REASONING, ATTENTION AND MEMORY BASED MACHINE LEARNING MODELS

A Project Report Submitted

in Partial Fulfilment of the Requirements

for the Degree of

BACHELOR OF TECHNOLOGY

in

Mathematics and Computing

by

Dhruv Kohli

(Roll No. 120123054)

to the

DEPARTMENT OF MATHEMATICS

INDIAN INSTITUTE OF TECHNOLOGY GUWAHATI

GUWAHATI - 781039, INDIA

April 2016

CERTIFICATE

This is to certify that the work contained in this project report entitled "**Reasoning, Attention and Memory Based Machine Learning Models**" submitted by **Dhruv Kohli** (**Roll No.: 120123054**) to Indian Institute of Technology Guwahati towards partial requirement of **Bachelor of Technology** in Mathematics and Computing has been carried out by him/her under my supervision and that it has not been submitted elsewhere for the award of any degree.

Guwahati - 781 039
April 2016

(Dr. Amit Sethi)
Project Supervisor

ABSTRACT

The main aim of the project is to explore and analyse the Reasoning, Attention and Memory (RAM) based models in machine learning with applications in the domain of sequence to sequence learning and question answering. We also present some ideas for a model based on RAM for learning the optimal policy of taking actions in a game.

Contents

List of Figures

Chapter 1

Introduction

Its a fact that every human being has the capability of memorizing facts as well as solving problems. Therefore, any model that tries to mimic the behavior of human brain must constitute a memory component and a problem solving component. The machine learning community represents the problem solving component with an artificial neural network and so far, has been successful in solving problems such as speech recognition, object classification, image annotation etc. But most of the existing machine learning models lack any interaction with the (potentially very large) long term memory component. That's where the reasoning, attention and memory based machine learning models come into picture.

The models discussed in this thesis comprises of a memory component and a controller, such as an artificial neural network, which takes inputs from external world, stores a representation or encoding of those inputs into the **memory**, interacts with the memory using a defined mechanism called **attention** process and produce outputs for the external world with **reasoning** where the reasoning follows from the graphical visualization of the interaction between controller and memory. Hence, our **memory** comprises

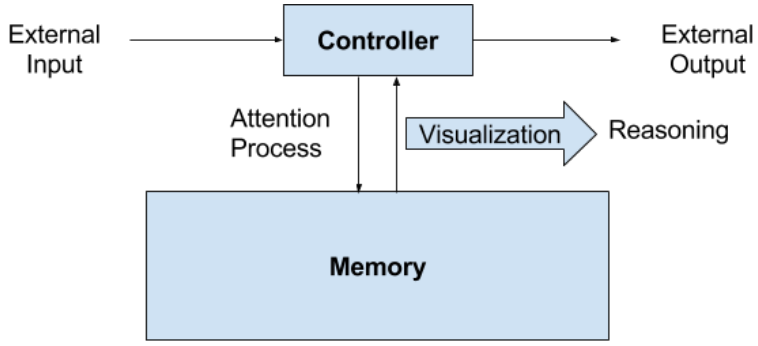

Figure 1.1: **General architecture of a RAM based model.**

of a set of entities where each entity is a representation or encoding of an external input seen by the controller, the **attention** process is the process or mechanism through which the controller interacts with the memory and the **reasoning** behind production of an external output follows from the graphical visualization of the interaction between controller and memory. Fig. 1.1 shows a general architecture of a RAM based model.

Following example demonstrates why RAM based machine learning models are superior to the existing machine learning models in terms of similarity with the way human beings think. Consider the comprehension:

1. John went to the kitchen.
2. John picked up the knife.
3. John dropped the knife on the floor.
4. Marry came into the kitchen.
5. Marry picked up the knife from the floor.
6. John went to the bathroom.
7. Marry went to dining room.

Now, try to answer the following query based on the above comprehension: Where is the knife? Your answer must have been "dining room".

Such question answering tasks have been successfully solved using recurrent neural networks. Recurrent neural networks (RNNs) are a class of artificial neural network architecture that-inspired by the cyclical connectivity of neurons in the brain-uses iterative function loops to store information. Though one might get high accuracy in terms of the correctness of the answer to the query using RNN, but RNN fails to provide an explicit explanation behind the production of a particular answer. On the contrary, RAM based model stores a representation of each sentence of the comprehension as an entity in the memory, searches for the entities in the memory relevant for answering the query and finally computes the answer to the query based on the searched relevant entities.

In this thesis, we focus on applying RAM based models in the domain of sequence to sequence learning where the aim is to find a mapping between a sequence of inputs to a sequence of outputs, in the domain of question answering where the aim is to build a model that can answer any query based on a given comprehension and in the domain of building game playing agents where the aim is to find an optimal policy which is a mapping from the state of the agent to a legal action so that, on following the optimal policy, the overall reward or score of the agent gets maximized.

1.1 Structure of the Thesis

Chapter 2 briefly reviews supervised sequence to sequence learning. Chapter 3 provides background material on artificial neural networks and recurrent neural networks. Chapter 4 and 5 investigates Neural Turing Machine with application in sequence to sequence learning and End to End Memory Networks with application in question answering tasks, respectively. Chapter 6

reviews reinforcement learning and building game playing agents using deep reinforcement learning and presents ideas for a RAM based model for learning the optimal policy for a game playing agent and shows its superiority to existing state of the art in terms of the similarity of the model with the way human beings perceive, reason and act.

Chapter 2

Supervised Sequence to Sequence Learning

This chapter provides background material and literature review of supervised sequence to sequence learning. Section 2.1 introduces supervised learning in general and section 2.2 introduces the problem of sequence to sequence learning in a supervised setting with examples from problems like Machine Translation and POS-Tagging.

2.1 Supervised Learning

Here, the goal is to learn a mapping from inputs x to outputs y, given a labeled set of input-output pairs $D = \{(x_i, y_i)\}_{i=1}^{N}$. Here D is called the training set, and N is the number of training examples. In the simplest setting, each training input x_i is a D-dimensional vector of numbers, representing, say, the height and weight of a person. These are called **features**, **attributes** or **covariates**. In general, however, x_i could be a complex structured object, such as an image, a sentence, an email message, a time series,

a molecular shape, a graph, etc.

Similarly the form of the output or **response variable** can in principle be anything, but most methods assume that y_i is a **categorical** or **nominal** variable from some finite set, $y_i \in \{1, \ldots, C\}$ (such as male or female), or that y_i is a real-valued scalar (such as income level). When y_i is categorical, the problem is known as **classification** or **pattern recognition**, and when y_i is real-valued, the problem is known as **regression**. Another variant, known as **ordinal regression**, occurs where label space Y has some natural ordering, such as grades A-F.

2.2 Sequence to Sequence Learning

Here, the goal is to learn a mapping from sequence of inputs $\{x_1, x_2, \ldots, x_{T_x}\}$ to a sequence of outputs $\{y_1, y_2, \ldots, y_{T_y}\}$, given a labeled set of input-output sequence pairs $D = \left\{\left(\left\{x_1^i, x_2^i, \ldots, x_{T_{x_i}}^i\right\}, \left\{y_1^i, y_2^i, \ldots, y_{T_{y^i}}^i\right\}\right)\right\}_{i=1}^{N}$. Here D is called the training set, and N is the number of training examples. Described below are some of the problems involving sequence to sequence learning.

Machine Translation

Consider a sentence i.e. sequence of words $\mathbf{x} = \{x_1, x_2, \ldots x_{T_x}\}$ in some language $L1$ and the corresponding translation of $\mathbf{x}$ i.e. $\mathbf{y} = \{y_1, y_2, \ldots y_{T_y}\}$, in some other language $L2$. In machine translation from $L1$ to $L2$, the aim is to find $\mathbf{y}$ from $\mathbf{x}$ as shown in Fig. 2.1. Traditional statistical techniques for machine translation models the probability distribution $P(\mathbf{y}|\mathbf{x})$ and for a given $\mathbf{x}$, searches for $\mathbf{y}^*$ that maximizes the conditional probability distribution i.e.

$$\mathbf{y}^* = argmax_y P(\mathbf{y}|\mathbf{x}) \tag{2.1}$$

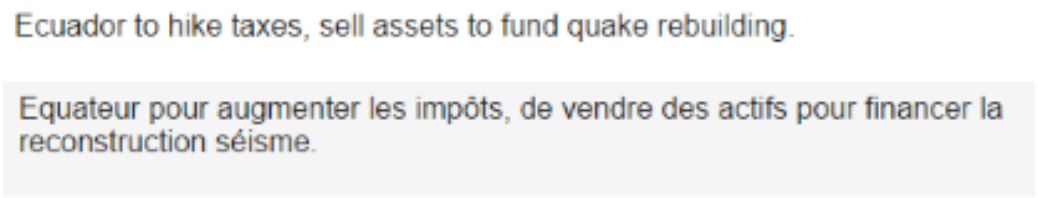

Figure 2.1: **Translation of text from English to French.**

In neural machine translation, we fit a parameterized model to maximize the conditional probability of sentence pairs using a parallel training corpus. Once the conditional distribution is learned by a translation model, given a source sentence a corresponding translation can be generated by searching for the sentence that maximizes the conditional probability. This neural machine translation approach consist of two components, the first encodes the variable length input sentence $\mathbf{x}$ to a fixed length vector and the second decodes the fixed length vector to the target sentence $\mathbf{y}$. In [4], RNN is used as an encoder and a separate RNN is used as a decoder. In this Encoder-Decoder framework, the encoder reads the input sentence, a sequence of vectors $\{x_1, x_2, \ldots x_{T_x}\}$ where $x_i \in \mathbb{R}^n$, uses RNN to compute the hidden states $\{h_1, h_2, \ldots h_{T_x}\}$ and finally computes a fixed length vector c from these hidden states as follows:

$$h_t = f\left(x_t, h_{t-1}\right) \tag{2.2}$$

$$c = q\left(\{h_1, h_2, \ldots h_{T_x}\}\right) \tag{2.3}$$

The decoder is trained to predict the next word $y_{t'}$ given the context vector c and all the previously predicted words $\{y_t, \ldots y_{t'-1}\}$. In other words, the decoder defines a probability over the translation $\mathbf{y}$ by decomposing the joint probability into the ordinal conditionals i.e.

$$P(\mathbf{y}) = \prod_{i=1}^{i=T_y} P(y_t | \{y_1, \ldots y_{t-1}\}, c) \tag{2.4}$$

A separate RNN is used to model the ordinal distribution where s_t represents the hidden states,

$$P(y_t | \{y_1, \ldots y_{t-1}\}, c) = g(y_{t-1}, s_t, c) \tag{2.5}$$

In [1], a bidirectional RNN is used to encode the input sequence. Consider an input sequence $\{x_1, x_2, \ldots x_{T_x}\}$, the forward hidden states $\left\{h_1^f, \ldots h_{T_x}^f\right\}$ obtained using an RNN which reads the input sequence in order from x_1 to x_{T_x} and the backward hidden states $\left\{h_1^b, \ldots h_{T_x}^b\right\}$ obtained using an RNN which reads input sequence in order from x_{T_x} to x_1, then the net hidden states are given by $h_t = \left[h_t^f, h_t^b\right]$. This way h_t contains the contextual information of both the preceding words and the following words with strong focus on x_t. Then, as in the decoder of [4], the decoder, here, defines each conditional probability as:

$$P(y_t | \{y_1, \ldots y_{t-1}\}, c) = g(y_{t-1}, s_t, c_t) \tag{2.6}$$

where s_t is the hidden state of a RNN. Unlike [4], here the context vector c_t is distinct for each target word y_t and is defined as the weighted sum of the hidden states produced by the encoder i.e.

$$c_t = \sum_{j=1}^{T_x} \alpha_{tj} h_j \tag{2.7}$$

where α_{tj} is defined as

$$\alpha_{tj} = \frac{exp(e_{tj})}{\sum_{k=1}^{T_x} exp(e_{tk})} \tag{2.8}$$

where

$$e_{tj} = a\left(s_{t-1}, h_j\right) \tag{2.9}$$

where a is a feed forward neural network jointly learnt with all other components. A graphical illustration of the model is shown in Fig. 2.2.

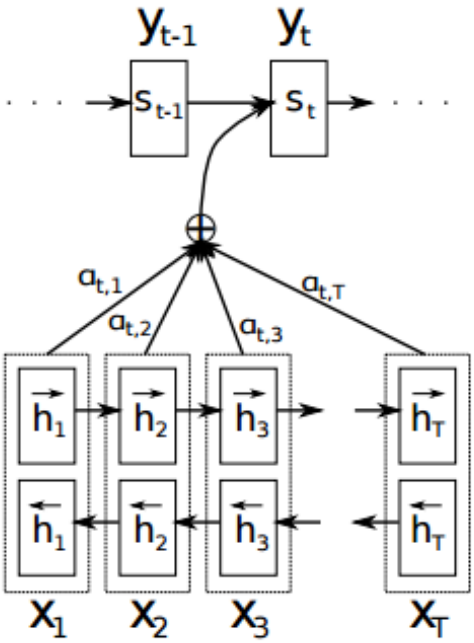

Figure 2.2: **Graphical illustration of the neural machine translation model using bidirectional RNN [1].**

The performance achieved by the above model is comparable to the existing state-of-the-art. In [17], an attention based model is used for machine translation using the concepts of Neural Turing Machine [9], and the results achieved are comparable to those in [1].

POS-Tagging

Here, given a text document comprising of sentences $\{s_1, s_2, \ldots s_n\}$ where

each sentence comprises of a sequence of words $\{s_{i1}, \ldots s_{in_i}\}$, the aim is to predict the correct Part-Of-Speech Tag corresponding to each word in each sentence as shown in Fig. 2.3. In [22], a bidirectional LSTM-RNN in roughly the same manner as described in [1] is used for solving the sequence to sequence learning problem of POS-Tagging.

```
>>> text = word_tokenize("They refuse to permit us to obtain the refuse permit")
>>> nltk.pos_tag(text)
[('They', 'PRP'), ('refuse', 'VBP'), ('to', 'TO'), ('permit', 'VB'), ('us', 'PRP'),
('to', 'TO'), ('obtain', 'VB'), ('the', 'DT'), ('refuse', 'NN'), ('permit', 'NN')]
```

Figure 2.3: **A sentence with POS-tags corresponding to each word of the sentence taken from NLTK.**

Chapter 3

Artificial Neural Networks

In this chapter, we review the theory of artificial neural networks, specifically, feedforward neural networks (FNNs). In section 3.1, we discuss the biological relationship of ANNs with neurons in a brain. In section 3.2, we elaborate on the theory of feedforward neural networks with forward propagation, activation functions, loss functions and backward propagation. In section 3.3, we state the commonly used gradient descent algorithms. In section 3.4, we state the commonly used techniques to initialize parameters of an ANN and in section 3.5, we state the procedure for training FNNs. Finally, in section 3.6, we give a brief overview of recurrent neural networks.

3.1 Perceptron

An artificial perceptron is an artificial version of biological neuron which takes a number of external inputs through the incoming connections, perform a weighted sum of the inputs where weights represents the strength of the connections (synapses) between neurons or between neurons and preceptors and finally applies an activation function such as sigmoid activation

function over the weighted sum producing an output based on which the perceptron is classified as activated or non-activated. In recent models, the binary state of an artificial perceptron is extended to a real valued state. In general, a perceptron can be represented with a function f which takes inputs $\{x_1, x_2, \ldots x_n\}$ through connections with weights (strengths) $\{w_1, w_2, \ldots w_n\}$ and a bias b and produces output $f\left(b + \sum_{i=1}^{n} w_i x_i\right)$ which represents the state of the perceptron.

3.2 Feedforward Neural Network

A feedforward network is represented by multiple layer of neurons (hidden layers) with an input and an output layer with connections feeding forward from one layer to the next as shown in Fig. 3.1. Input pattern is presented to the input layer, propagated through the hidden layers to the output layer. The output of a FNN depends on the current input only and does not depend on past inputs or future inputs. Therefore the FNN are not suitable for the task of sequence to sequence learning or question answering or building game playing agent.

A FNN with a particular set of weights defines a function from input to output. By altering the weights, a single FNN is capable of instantiating many different functions. It has been proven in [14] that an FNN with a single hidden layer containing a sufficient number of nonlinear units can approximate any continuous function on a compact input domain to arbitrary precision. For this reason FNNs are said to be universal function approximators.

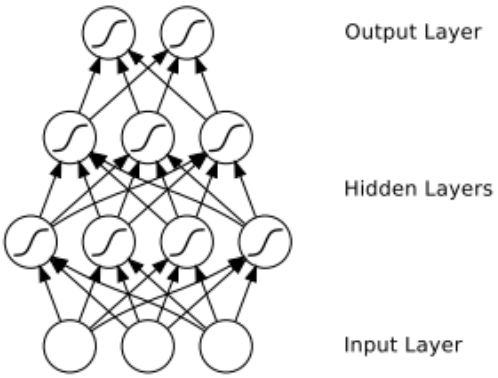

Figure 3.1: **A feedforward neural network.** The S-shaped curves in the hidden and output layers indicate the application of 'sigmoidal' nonlinear activation functions.

3.2.1 Forward Propagation

Consider an input vector to FNN $\mathbf{x} = \{x_1, \ldots x_n\}$. For hidden unit h in the first hidden layer of neurons with connections w^1_{ih} with x_i where i goes from 1 to n and bias b^1_h, the activation a^1_h is computed as

$$s^1_h = b^1_h + \sum_{i=1}^{n} w^1_{ih} x_i \tag{3.1}$$

$$a^1_h = \theta_h \left(s^1_h \right) \tag{3.2}$$

where θ_h is an appropriately chosen activation function depending on the problem statement. Similarly, for hidden unit h in the hidden layer H_l with connections $w^l_{h'h}$ with h' unit of previous layer H_{l-1} and bias b^l_h, the activation a^l_h is computed as

$$s^l_h = b^l_h + \sum_{h' \in H_{l-1}} w^l_{h'h} a^{l-1}_{h'} \tag{3.3}$$

$$a^l_h = \theta_h \left(s^l_h \right) \tag{3.4}$$

The process is repeated until the output activations are generated.

3.2.2 Activation Functions

Following are some of the most commonly used activation functions:

Sigmoid

$$\theta_h(x) = \sigma(x) = \frac{1}{1 + exp(-x)} \tag{3.5}$$

Tanh

$$\theta_h(x) = tanh(x) = \frac{exp(x) - exp(-x)}{exp(x) + exp(-x)} \tag{3.6}$$

Rectifier Linear Unit (RELU)

$$\theta_h(x) = relu(x) = max(x, 0) \tag{3.7}$$

Leaky-RELU

$$\theta_h(x) = lrelu(x, \alpha) = \mathbb{I}(x \geq 0)\, x + \mathbb{I}(x < 0)\, \alpha x \tag{3.8}$$

Linear

$$\theta_h(x) = x \tag{3.9}$$

Sine

$$\theta_h(x) = sin(x) \tag{3.10}$$

3.2.3 Loss Functions

Suppose y is the target and p is the output produced by the neural network with parameters Θ then following are the commonly used functions to compute the loss or cost with respect to the parameters of the model.

Euclidean squared error (L2 loss)

$$l_{L2}(y, p(\Theta)) = \|y - p(\Theta)\|_2^2 \tag{3.11}$$

Laplacian Loss (L1 loss)

$$l_{L1}(y, p(\Theta)) = \|y - p(\Theta)\|_1 \tag{3.12}$$

Binary Cross-Entropy Loss (Classification task)

$$l_{binary\_crossentropy}(y, p(\Theta)) = -\sum_i y_i \log(p(\Theta)_i) \tag{3.13}$$

where y is a one-hot encoded vector with $y_i = 1$ if the ith example has class C_i.

In probabilistic setting, L2 loss corresponds to the negative log likelihood of a normal distribution with mean $p(\Theta)$, L1 loss corresponds to the negative log likelihood of a laplacian distribution with mean $p(\Theta)$ and binary crossentropy loss corresponds to negative log likelihood of a multinomial distribution with class probabilities given by vector $p(\Theta)$.

3.2.4 Backward Propagation

Once an appropriate loss function is chosen, the main goal is to minimize the loss with respect to the parameters. Basically, the derivative of the loss with respect to each parameter of the model is computed and the parameters are then updated using one of the gradient descent algorithms. The computation of the gradient of loss with respect to each parameter is performed using backpropagation algorithm which makes use of chain rule of partial derivatives. As an example, consider the L2 loss l_{L2} and the output of the final

layer H_l be $p(= a_h^l)$ which is computed using equations 3.3 and 3.4. Now, to compute derivative of loss with respect to $w_{h'h}^k$ which connects $a_{h'}^{k-1}$ with s_h^k, chain rule of partial derivatives is used.

We start by computing the partial derivative of loss l_{L2} with respect to activations in the final layer a_h^l.

$$\Delta_h^l = \frac{\partial l_{L2}}{\partial a_h^l} = 2(a_h^l - y_h) \tag{3.14}$$

Then, the by making use of the chain rule of partial derivatives, derivative of loss with respect to pre-activations s_h^l is computed.

$$\delta_h^l = \frac{\partial l_{L2}}{\partial s_h^l} = \frac{\partial l_{L2}}{\partial a_h^l}\frac{\partial a_h^l}{\partial s_h^l} = \Delta_h^l \theta'(s_h^l) \tag{3.15}$$

The derivative of loss with respect to the parameters $w_{h'h}^l$ and b_h^l are computed as,

$$\frac{\partial l_{L2}}{\partial w_{h'h}^l} = \frac{\partial l_{L2}}{\partial s_h^l}\frac{\partial s_h^l}{\partial w_{h'h}^l} = \delta_h^l a_{h'}^{l-1} \tag{3.16}$$

$$\frac{\partial l_{L2}}{\partial b_h^l} = \frac{\partial l_{L2}}{\partial s_h^l}\frac{\partial s_h^l}{\partial b_h^l} = \frac{\partial l_{L2}}{\partial s_h^l} = \delta_h^l \tag{3.17}$$

The derivative of loss with respect to the activations a_h^{l-1} are computed as,

$$\Delta_h^{l-1} = \frac{\partial l_{L2}}{\partial a_h^{l-1}} = \sum_{h' \in H_l} \frac{\partial l_{L2}}{\partial s_{h'}^l}\frac{\partial s_{h'}^l}{\partial a_h^{l-1}} = \sum_{h' \in H_l} \delta_{h'}^l w_{hh'}^l \tag{3.18}$$

Then, the derivative of loss with respect to the pre-activations s_h^{l-1} are computed as,

$$\delta_h^{l-1} = \frac{\partial l_{L2}}{\partial s_h^{l-1}} = \frac{\partial l_{L2}}{\partial a_h^{l-1}} \frac{\partial a_h^{l-1}}{\partial s_h^{l-1}} = \Delta_h^{l-1} \theta'(s_h^{l-1}) = \theta'(s_h^{l-1}) \sum_{h' \in H_l} \delta_{h'}^l w_{hh'}^l \quad (3.19)$$

To compute the gradient of loss with respect to the parameters $w_{h'h}^k$ and b_h^k where k goes from $l-1$ to 1, the recursion developed in equation 3.19 with equations 3.18, 3.15, 3.16 and 3.17 are used.

3.3 Parameter Update Rules

Once the derivative of loss with respect to the parameters has been calculated, the next step is to update the parameters such that the loss with the new set of parameters is lesser. This is done by making a step on the surface of the loss function which is a function of the parameters from the current point to a point which has a lower associated loss. This is efficiently done using **gradient descent algorithms**. This process of gradient computation and gradient descent is repeated until the parameters converge to a point on the surface where the loss is minimum. Here, we describe some of the most commonly used gradient descent algorithms.

Stochastic Gradient Descent

$$\Theta_{t+1} = \Theta_t - \alpha \nabla L(\Theta_t) \quad (3.20)$$

Here, Θ_{t+1} is the new set of parameters obtained by subtracting α times the gradient of loss with respect to the parameters at old state $\nabla(\Theta_t)$ from the old set of parameters Θ_t. Here, α is the learning rate which has to be tuned. A large value of α leads to divergence of the parameters from the minima while a very small value of α leads to slow convergence. Hence, SGD is highly

sensitive to the learning rate α. [2]

Momentum update

$$V_{t+1} = \mu V_t - \alpha \nabla L(\Theta_t) \tag{3.21}$$

$$\Theta_{t+1} = \Theta_t + V_t \tag{3.22}$$

Here, V_t and μ can be interpreted as velocity at time t and friction coefficient that damps the velocity. This makes SGD more less sensitive to the learning rate α as it prevents oscillation of the update V_{t+1} in the steep direction and builds up the update V_{t+1} in shallow direction, hence taking significant steps. [2]

Adaptive Gradient Descent

$$\Theta_{t+1_i} = \Theta_{t_i} - \alpha \frac{(\nabla L(\Theta_t))_i}{\sqrt{\sum_{t'=1}^{t} (\nabla L(\Theta'_t))_i^2}} \tag{3.23}$$

Here, Θ_{t+1_i} is the ith component of the new set of parameters obtained. The step for ith component is divided by the square root of the sum of the square of the ith component of the gradient of loss with respect to the old set of parameters Θ_t. This helps to adapt the step for steep and shallow directions. But, this results in slow convergence of parameters due to gradual decrease of step size to zero. [5]

RMSprop

$$U_{t+1_i} = \eta U_t - (1 - \eta)(\nabla L(\Theta_t))_i^2 \tag{3.24}$$

$$\Theta_{t+1_i} = \Theta_{t_i} - \alpha \frac{(\nabla L(\Theta_t))_i}{\sqrt{U_{t+1_i}}} \tag{3.25}$$

Here, $U_{0_i} = 0 \; \forall \; i$ and η is the decay rate which decays the accumulated sum of squares of gradients so that the step size does not gradually decrease to zero but increase too. This speeds up the convergence of parameters [21]

Adaptive Moment (Adam)

$$M_{t+1_i} = \chi M_t - (1 - \chi)(\nabla L(\Theta_t))_i \tag{3.26}$$

$$V_{t+1_i} = \eta V_t - (1 - \eta)(\nabla L(\Theta_t))_i^2 \tag{3.27}$$

$$\Theta_{t+1_i} = \Theta_{t_i} - \alpha \frac{M_{t+1_i}}{\sqrt{U_{t+1_i}}} \tag{3.28}$$

Here, M_t represents momentum, V_t represents velocity, $M_{0_i} = 0$ and $V_{0_i} = 0$, $\forall \; i$, χ is the decay rate for momentum and η is the decay rate for velocity. This method is generalization of adaptive gradient descent. [15]

3.4 Parameter Initialisation

The first step of training a neural network is to randomly initialize the parameters which comprises of weights of the connections and bias terms. Usually, the weights are initialized with standard normal distribution and bias terms are initialized to a constant 0 or 0.1. But in convolutuional neural networks [16], some parameter initialization techniques like HeUniform,

HeNormal [11], GlorotUniform and GlorotNormal [7] are proven to be better in terms of speed of convergence to optimal parameters.

3.5 Training a Feedforward Neural Network

Given data $D = \{(x_i, y_i)\}_{i=1}^{N}$ where x_i represents the input and y_i denotes the targets, training a feedforward with paramters Θ, which constitutes all the weights connecting neurons and bias terms, comprises of the following steps:

1. Randomly initialize the parameters (refer 3.4)
2. Compute the outputs p_i for each input x_i using forward pass (refer 3.2.1)
3. Compute the loss or error in outputs from targets using an appropriate loss function (refer 3.2.3)
4. Compute the derivative of loss with respect to each parameter (refer 3.2.4)
5. Perform a gradient descent step using an appropriate gradient descent algorithm (refer 3.3)
6. Repeat from second step until the parameters converge or loss becomes less than some threshold.

3.6 Recurrent Neural Network

Recurrent neural networks, as the name implies, contains feedback connections from neurons in a layer to neurons in the same or previous layers. RNN

maps an entire history of past inputs to an output. Indeed, the equivalent result to the universal approximation theory for FNNs is that an RNN with a sufficient number of hidden units can approximate any measurable sequence to sequence mapping to arbitrary accuracy [10]. **The key point is that the recurrent connections allow a 'memory' of previous inputs to persist in the networks internal state, and thereby influence the network output.**

3.6.1 Unfolding

RNN can be viewed as FNN when unfolded. Fig. 3.2 shows a RNN and the corresponding unfolded version. Once unfolded, the training procedure for FNN can be applied to RNN too.

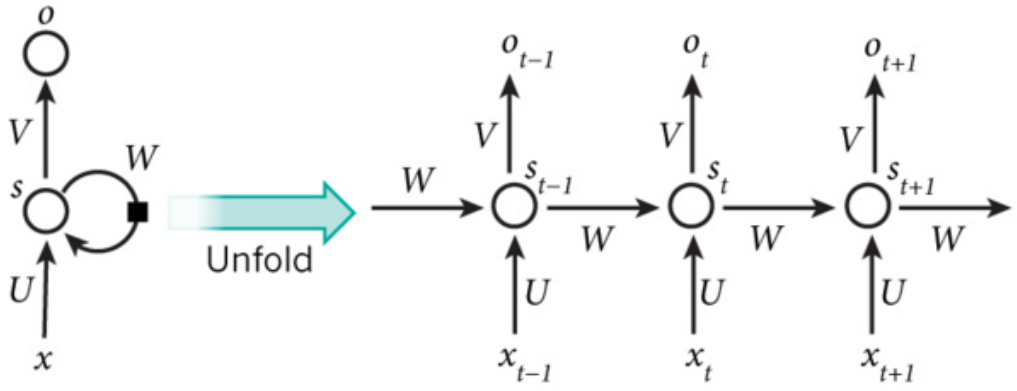

Figure 3.2: **RNN with unfolded version. [3]**

3.6.2 Vanishing or Exploding Gradient

Unfortunately, for standard RNN architectures, the range of context that can be in practice accessed is quite limited. The problem is that the influence of a given input on the hidden layer, and therefore on the network

output, either decays or blows up exponentially as it cycles around the networks recurrent connections. This effect is often referred to in the literature as the vanishing gradient problem [12]. The vanishing gradient problem is illustrated schematically in Fig. 3.3. To overcome this problem, Long-Short Term Memory Architecture was introduced. LSTM in itself is an exhaustive model which we do not intend to cover in this thesis. One can refer to [13] to get a detailed view of LSTM, from the design of the model to its training.

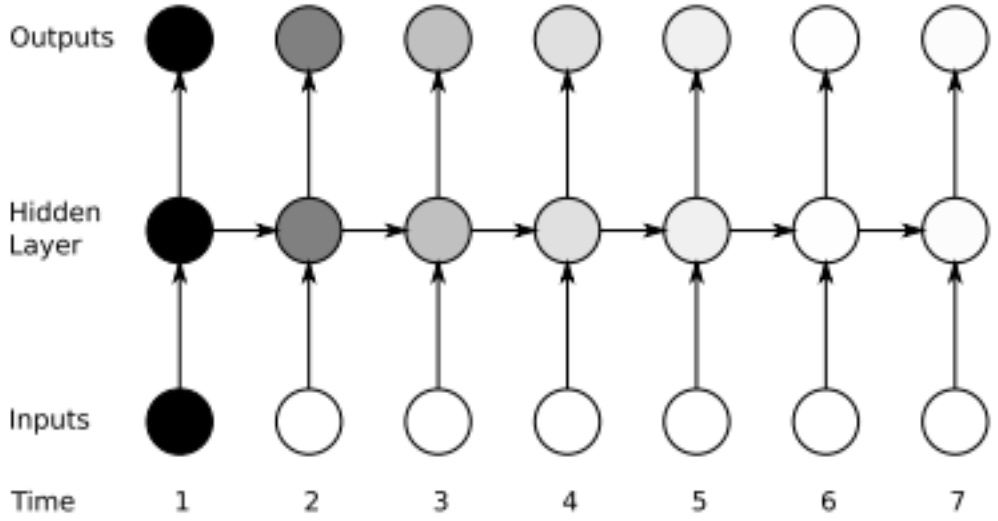

Figure 3.3: **Illustration of vanishing gradient in recurrent neural networks.** [8] The shading of the nodes in the unfolded network indicates their sensitivity to the inputs at time one (the darker the shade, the greater the sensitivity). The sensitivity decays over time as new inputs overwrite the activations of the hidden layer, and the network forgets the first inputs.

Chapter 4

Neural Turing Machine

The combination of neural networks, specifically recurrent neural networks, with a large addressable memory, which they can interact with by attentional processes, is called a Neural Turing Machine due to its analogy to Turing Machine with infinite memory tape. Unlike Turing Machine, an NTM is a differentiable computer that can be trained by gradient descent, yielding a practical mechanism for learning programs. In [9], preliminary results show that an NTM can infer simple algorithms such as copying, sorting and associative recall.

4.1 Architecture

A Neural Turing Machine architecture contains two basic components: a neural network controller and a memory bank. Fig. 4.1 presents a basic diagram of the NTM architecture. The controller not only interacts with an external world via input and output vectors but also interacts with a memory matrix using selective read and write operations. By analogy to the Turing machine, network outputs that parametrise these operations are referred as

"heads".

Note that Turing Machine has a defined controller in terms of the task that the machine has to perform. On the other hand, the controller in Neural Turing Machine is learnt based on the input output pairs corresponding to a particular task.

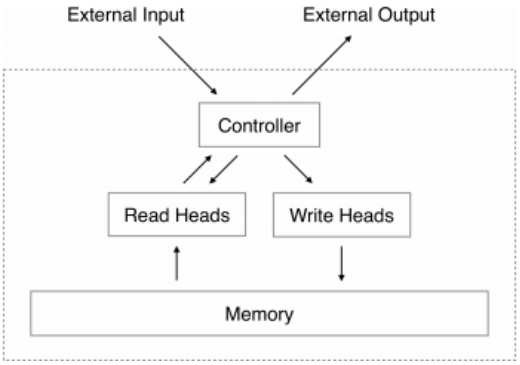

Figure 4.1: **Neural Turing Machine architecture.[9]**

Breifly, the controller in NTM is a neural network (feedforward, recurrent or LSTM), the memory is a matrix consisting of N memory slots each of an appropriate size M, the read and write heads are vectors of length N used by the controller to strongly focus on one memory slot or weakly focus on multiple memory slots.

4.2 Attention Mechanism

The process or mechanism with which the controller interacts with the memory is called attention mechanism. In NTM, the attention mechanism conmprises of procedures to read from memory, write into memory and updation of read and write heads.

4.2.1 Reading from Memory

Let $\mathbf{M}_t$ be the contents of the NxM memory matrix at time t, where N is the number of memory locations and M is the vector size at each location. Let w_t be a vector of weightings over the N locations emitted by a read head at time t. Since all the weightings are normalised, the N elements w_t(i) of $\mathbf{w}_t$ are given by:

$$\sum_i w_t(i) = 1 \tag{4.1}$$

The length M read vector $\mathbf{r}_t$ returned by the head is defined as a convex combination of the row-vectors $\mathbf{M}_t$(i) in memory:

$$\mathbf{r}_t \leftarrow \sum_i w_t(i)\mathbf{M}_t(i) \tag{4.2}$$

$\mathbf{r}_t$ is differentiable with respect to both the memory and the weighting. The derivative of $\mathbf{r}_t$ with respect to $\mathbf{w}_t$ is:

$$\frac{\partial r_t(i)}{\partial w_t(j)} = \mathbf{M}_t(i,j) \tag{4.3}$$

and with respect to $\mathbf{M}_t$ is:

$$\frac{\partial r_t(k)}{\partial \mathbf{M}_t(i,j)} = w_t(i)\mathbb{I}(k=j) \tag{4.4}$$

4.2.2 Writing into Memory

Given a weighting vector $\mathbf{w}_t$ emitted by a write head at time t, along with an erase vector $\mathbf{e}_t$ whose M elements all lie in the range (0,1), the memory

vectors $\mathbf{M}_{t-1}$(i) from the previous time-step are modified as follows:

$$\widetilde{\mathbf{M}}_t(i) \leftarrow \mathbf{M}_{t-1}(i)[1 - w_t(i)\mathbf{e}_t] \tag{4.5}$$

where **1** is a row vector of all 1-s, and the multiplication against memory location acts point-wise. Therefore, the elements of a memory location are reset to zero only if both the weighting at the location and the erase element are one; if either the weightings or the erase is zero, the memory is left unchanged. When multiple write heads are present, the erasures can be performed in any order, as multiplication is commutative.

$\widetilde{\mathbf{M}}_t$ is differentiable with respect to $\mathbf{M}_t$ $\mathbf{w}_t$ and $\mathbf{e}_t$ and the corresponding derivatives are:

$$\frac{\partial \widetilde{\mathbf{M}}_t(i,j)}{\partial \mathbf{M}_t(i',j')} = (1 - w_t(i)e_t(j))\mathbb{I}(i = i')\mathbb{I}(j = j') \tag{4.6}$$

$$\frac{\partial \widetilde{\mathbf{M}}_t(i,j)}{\partial w_t(k)} = -\mathbf{M}_{t-1}(i,j)e_t(j)\mathbb{I}(i = k) \tag{4.7}$$

$$\frac{\partial \widetilde{\mathbf{M}}_t(i,j)}{\partial e_t(k)} = -\mathbf{M}_{t-1}(i,j)w_t(i)\mathbb{I}(k = j) \tag{4.8}$$

Each write head also produces a length M add vector $\mathbf{a}t$, which is added to the memory after the erase step has been performed:

$$\mathbf{M}_t(i) \leftarrow \widetilde{\mathbf{M}}_t(i) + w_t(i)\mathbf{a}_t \tag{4.9}$$

Once again, the order in which the adds are performed by multiple heads is irrelevant. The combined erase and add operations of all the write heads produces the final content of the memory at time t.

$\mathbf{M}_t$ is differentiable with respect to $\mathbf{w}_t$, $\mathbf{a}_t$ and $\mathbf{e}_t$ and the corresponding

derivatives are:

$$\frac{\partial \mathbf{M}_t(i,j)}{\partial \widetilde{\mathbf{M}}_t(i',j'))} = \mathbb{I}(i=i')\mathbb{I}(j=j') \tag{4.10}$$

$$\frac{\partial \mathbf{M}_t(i,j)}{\partial w_t(k))} = w_t(i)\mathbb{I}(j=k) \tag{4.11}$$

$$\frac{\partial \mathbf{M}_t(i,j)}{\partial w_t(k))} = a_t(j)\mathbb{I}(i=k) \tag{4.12}$$

4.2.3 Addressing Mechanism

The addressing mechanism used to update read as well as write weighting vectors is presented as a flow diagram in Fig. 4.2.3. The weightings arise by combining two addressing mechanisms: "content-based addressing" and "location-based addressing". The content based addressing focuses attention on locations based on similarity between current values in memory matrix and the values emitted by the controller. The location based addressing is used to facilitate both simple iterations across the locations of the memory and random-access jumps. It includes the interpolation, convolutional shift and finally, sharpening. The following sub-sections provides a description of each stage.

Content Addressing

The weightings produced by the content addressing module will focus on those memory slots $\mathbf{M}_t$(i) which are similar to the length M key vector $\mathbf{k}_t$ (with respect to cosine similarity) and hence the content of the read vector (if formed by this weighting alone) will be similar to the content of the focused memory slot content. The precision of focus can be attenuated or amplified

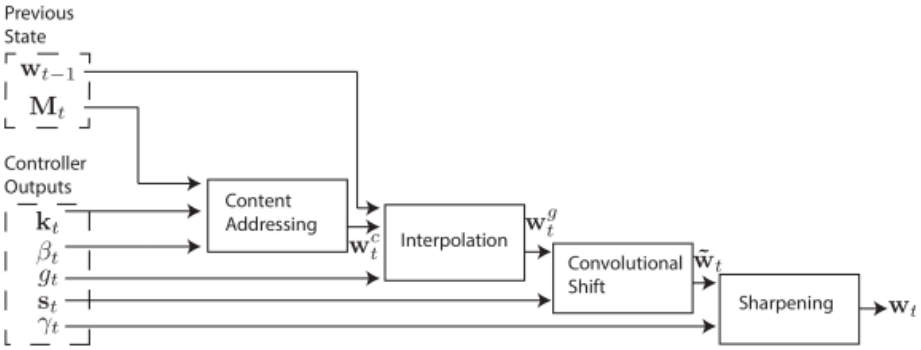

Figure 4.2: **Flow diagram of addressing mechanism.[9]** The key vector, $\mathbf{k}_t$ and key strength, β_t are used to perform content-based addressing of the memory matrix, $\mathbf{M}_t$. The resulting content-based weighting is interpolated with the weighting from the previous time step based on the value of the interpolation gate, g_t. The shift weighting, s_t determines whether and by how much the weighting is rotated. Finally, depending on γ_t, the weighting is sharpened and used for memory access.

with a positive key strength β_t.

As an example, suppose memory looks like $[A, B, C, D, ..., Z]$, 26 memory slots with one unique alphabetic character each, key vector is D and key strength is very large (note that infinite key strength changes softmax to max). Then, the weighting produced by content addressing module will approximately be $[0, 0, 0, 1, ..., 0]$ and corresponding read vector will be D.

$$w_t^c(i) \leftarrow \frac{\exp\left(\beta_t \mathbf{K}[\mathbf{k}_t, \mathbf{M}_t(i)]\right)}{\sum_j \exp\left(\beta_t \mathbf{K}[\mathbf{k}_t, \mathbf{M}_t(j)]\right)} \tag{4.13}$$

The cosine similarity measure is defined as:

$$\mathbf{K}[\mathbf{u}, \mathbf{v}] \leftarrow \frac{\mathbf{u}.\mathbf{v}}{\|\mathbf{u}\| \, \|\mathbf{v}\|} \tag{4.14}$$

The derivative of $\mathbf{w}_t^c$ with respect to β_t, $\mathbf{k}_t$ and $\mathbf{M}_t$ are:

$$\frac{\partial w_t^c(i)}{\partial \beta_t} \leftarrow \frac{\exp(\beta_t c_i)(\sum_j (c_i - c_j)\exp(\beta_t c_j))}{(\sum_j \exp(\beta_t c_j))^2} \tag{4.15}$$

where

$$c_i \leftarrow \mathbf{K}[\mathbf{k}_t, \mathbf{M}_t(i)] \tag{4.16}$$

Also,

$$\frac{w_t^c(i)}{\partial c_i} \leftarrow \frac{\sum_{j \neq i} \beta_t \exp(\beta_t c_j)}{(\sum_j \exp(\beta_t c_j))^2} \tag{4.17}$$

and

$$\frac{w_t^c(i)}{\partial c_j} \leftarrow \frac{-\beta_t \exp(\beta_t c_i)\exp(\beta_t c_j)}{(\sum_j \exp(\beta_t c_j))^2} \tag{4.18}$$

where

$$\frac{\partial \mathbf{K}[\mathbf{u},\mathbf{v}]}{\partial u(i)} \leftarrow \frac{1}{\|\mathbf{u}\|\,\|\mathbf{v}\|}\left[v(i) - \frac{u(i)(\mathbf{u}.\mathbf{v})}{\|\mathbf{u}\|^2}\right] \tag{4.19}$$

where $\mathbf{u}$=$\mathbf{k}_t$ and $\mathbf{v}$=$\mathbf{M}_t(i)$. Now, one can use chain rule to find the derivative of $\mathbf{w}_t^c$ with respect to $\mathbf{k}_t$ and $\mathbf{M}_t$.

Interpolation

Each head emits a scalar interpolation gate g_t in the range (0,1). The value of g_t is used to blend between the weighting $\mathbf{w}_{t-1}$ produced by the head at the previous time-step and the weighting $\mathbf{w}_t^c$ produced by the content system at the current step, yielding the gated weighting $\mathbf{w}_t^g$:

$$\mathbf{w}_t^g \leftarrow g_t \mathbf{w}_t^c + (1 - g_t)\mathbf{w}_{t-1} \tag{4.20}$$

The main aim of having this module is to give the addressing mechanism capability to completely neglect the weighting produced by content addressing based module and use the previous final weighting as a reference point. This

gives NTM the power to iterate over memory slots (this module just sets the reference point i.e. the weighting to start from while the next module takes care of the iteration part). One can easily observe that, if the gate is zero, the the content weighting is entirely ignored and weighting from previous time step is used and if the gate is one then the weighting from the previous iteration is ignored, and the system applies content based addressing. The derivative of $\mathbf{w}_t^g$ with respect to g_t, $\mathbf{w}_t^c$ and $\mathbf{w}_{t-1}$ are:

$$\frac{\partial w_t^g(i)}{\partial g_t} \leftarrow w_t^c(i) - w_{t-1}(i) \tag{4.21}$$

$$\frac{\partial w_t^g(i)}{\partial w_t^c(j)} \leftarrow g_t \mathbb{I}(i = j) \tag{4.22}$$

$$\frac{\partial w_t^g(i)}{\partial w_{t-1}(j)} \leftarrow (1 - g_t)\mathbb{I}(i = j) \tag{4.23}$$

Convolutional Shift

The controller emits a shift weighting s_t that defines a normalised distribution over allowed integer shifts. If we index N memory locations from 0 to N-1, the rotation applied to $\mathbf{w}_t^g$ by $\mathbf{s}_t$ can be expressed as the following circular convolution.

$$\widetilde{w}_t(i) \leftarrow \sum_{j=0}^{N-1} w_t^g(j) s_t(i - j) \tag{4.24}$$

As an example, suppose memory looks like $[A, B, C, D, ..., Z]$, 26 memory slots with one unique alphabetic character each and the weighting vector produced by interpolation module is $[0, 1, 0, 0, 0, ..., 0]$ (focusing on memory slot containing B). Now, suppose that shift weighting vector produced only allows shifts of 1 and the corresponding shift weight value be 1. That means

$\mathbf{s}_t(1) = 1$ and $\mathbf{s}_t(i) = 0 \ \forall$ i $\neq 1$. Then, using equation 4.24 new weighting vector produced will be $[0, 0, 1, 0, ..., 0]$. Also, note that if the shift weighting vector produced, only allows shifts of -1, 0 and 1 and the corresponding shift weight value be 0.1, 0.8 and 0.1 respectively. That means $\mathbf{s}_t(1) = 0.1$, $\mathbf{s}_t(-1) = 0.1$, st(0)=0.8 and $\mathbf{s}_t(i) = 0 \ \forall$ i $\neq$ to {-1,0,1}. Then, the new weighting vector will be $[0.1, 0.8, 0.1, 0, ..., 0]$ which is slightly blurred over three points. To combat this we require sharpening. Note that the derivative of $\widetilde{\mathbf{w}_t}$ with respect to $\mathbf{s}_t$ and $\mathbf{w}_t^g$ are (note that the subtraction is modulo N):

$$\frac{\partial \widetilde{w}_t(i)}{\partial s_t(k)} \leftarrow \sum_{k=0}^{N-1} w_t^g(i-k) \tag{4.25}$$

$$\frac{\partial \widetilde{w}_t(i)}{\partial w_t^g(j)} \leftarrow \sum_{j=0}^{N-1} s_t(i-j) \tag{4.26}$$

Sharpening

Each head emits one further scalar γ_t whose effect is to sharpen the weights as follows:

$$w_t(i) \leftarrow \frac{\widetilde{w}_t(i)^{\gamma_t}}{\sum_j \widetilde{w}_t(j)^{\gamma_t}} \tag{4.27}$$

The derivative of $\mathbf{w}_t$ with respect to $\widetilde{\mathbf{w}_t}$ and γ_t are:

$$\frac{\partial w_t(i)}{\partial \gamma_t} \leftarrow \frac{\widetilde{w}_t(i)_t^{\gamma}}{\sum_j \widetilde{w}_t(j)_t^{\gamma}} \left[\log(\widetilde{w}_t(i)) - \frac{\sum_j \log(\widetilde{w}_t(j)) \widetilde{w}_t(j)_t^{\gamma}}{\sum_k \widetilde{w}_t(k)_t^{\gamma}} \right] \tag{4.28}$$

$$\frac{\partial w_t(i)}{\partial \widetilde{w}_t(i)} \leftarrow \frac{\gamma_t \widetilde{w}_t(i)^{\gamma_t - 1}}{\sum_j \widetilde{w}_t(j)_t^{\gamma}} \left[1 - \frac{\widetilde{w}_t(i)_t^{\gamma}}{\sum_k \widetilde{w}_t(k)_t^{\gamma}} \right] \tag{4.29}$$

and when i $\neq$ j

$$\frac{\partial w_t(i)}{\partial \widetilde{w}_t(j)} \leftarrow \frac{-\gamma_t \widetilde{w}_t(i)^{\gamma_t} \widetilde{w}_t(j)^{\gamma_t - 1}}{\sum_k \widetilde{w}_t(k)_t^{\gamma}} \tag{4.30}$$

The combined addressing system of weighting interpolation and content and location-based addressing can operate in three complementary modes:

1. A weighting can be chosen by the content system without any modification by the location system.

2. A weighting produced by the content addressing system can be chosen and then shifted. This allows the focus to jump to a location next to, but not on, an address accessed by content; in computational terms this allows a head to find a contiguous block of data, then access a particular element within that block.

3. A weighting from the previous time step can be rotated without any input from the content-based addressing system. This allows the weighting to iterate through a sequence of addresses by advancing the same distance at each time-step.

4.3 Controller

The choices for the controller include LSTM, feedforward networks and recurrent network, each of which are explained in chapter 4.

4.4 Training Neural Turing Machine

Fig. 4.3 represents a simulation of external input sequence $\{X_1, X_2, \ldots X_n\}$ through a Neural Turing Machine. C_t represents hidden layer of neurons where C_0 represents hidden layer of neurons at time $t = 0$ which can also be interpreted as the biased hidden layer of neurons which acts as one of

the parameters of the NTM. Hidden layer of neurons C_t is used to produce an erase vector $\mathbf{e_{t+1}}$ and an add vector $\mathbf{a_{t+1}}$ which are used in updating the memory. The parameters $\mathbf{W_{eC}}$ and $\mathbf{b_{eC}}$ are used to generate $\mathbf{e_{t+1}}$ and the parameters $\mathbf{W_{aC}}$ and $\mathbf{b_{aC}}$ are used to generate $\mathbf{a_{t+1}}$. Here, the values in add vector are clipped between -1 and 1.

$$\mathbf{e_{t+1}} \leftarrow \sigma(\mathbf{W_{eC}}C_t + \mathbf{b_{eC}}) \tag{4.31}$$

$$\mathbf{a_{t+1}} \leftarrow (\mathbf{W_{aC}}C_t + \mathbf{b_{aC}}) \tag{4.32}$$

$\mathbf{M_0}$ represents the bias memory matrix which acts as one of the parameters of the NTM. Memory matrix at time t is given by $\mathbf{M_t}$ which is updated to $\mathbf{M_{t+1}}$ by applying equations 4.5 and 4.9 which use the write weighting vector $\mathbf{w_{w,t}}$, erase vector $\mathbf{e_{t+1}}$ and add vector $\mathbf{a_{t+1}}$. $\mathbf{r_t}$ represents read vector which is formed by applying equation 4.2 which uses the updated memory matrix $\mathbf{M_t}$ and the read weighting vector $\mathbf{w_{r,t-1}}$. Then, the external input X_t and the read vector $\mathbf{r_t}$ are used to update the hidden layer of neurons C_t using the parameters $\mathbf{W_{CX}}$, $\mathbf{b_{CX}}$, $\mathbf{W_{Cr}}$ and $\mathbf{b_{Cr}}$.

$$C_t \leftarrow relu(\mathbf{W_{CX}}X_t + \mathbf{b_{CX}} + \mathbf{W_{Cr}}\mathbf{r_t} + \mathbf{b_{Cr}}) \tag{4.33}$$

Using the updated hidden layer of neurons C_t and a set of parameters $\{\mathbf{W_{PC}}, \mathbf{b_{PC}}, \mathbf{W_{k_uC}}, \mathbf{b_{k_uC}}, \mathbf{W}_{\beta_\mathbf{u}\mathbf{C}}, \mathbf{b}_{\beta_\mathbf{u}\mathbf{C}}, \mathbf{W_{g_uC}}, \mathbf{b_{g_uC}}, \mathbf{W_{s_uC}}, \mathbf{b_{s_uC}}, \mathbf{W}_{\gamma_\mathbf{u}\mathbf{C}}, \mathbf{b}_{\gamma_\mathbf{u}\mathbf{C}}\}$, an external output P_t (assuming that the values in vectors of target sequence lie between 0 and 1), a set of outputs for updating the read weighting vector $\mathbf{H_{r,t}}$ and a set of outputs for updating the write weighting vector $\mathbf{H_{w,t}}$ are produced. Here, $\mathbf{H_{u,t}}$ comprises of a key vector $\mathbf{k_{u,t}}$, a key strength scalar $\beta_{u,t}$, a gating scalar $g_{u,t}$, a shifting vector $\mathbf{s_{u,t}}$ and a sharpening scalar $\gamma_{u,t}$ where $u = r$ for read weighting vector and $u = w$ for write weighting vector.

Note that the key vector is clipped between -1 and 1. These sets of outputs along with the memory matrix and read and write weighting vectors at previous time step are used to update the read and write weighting vectors by making use of the addressing mechanism defined in section 4.2.3

$$P_t \leftarrow \sigma(\mathbf{W_{PC}} C_t + \mathbf{b_{PC}}) \tag{4.34}$$

$$\mathbf{k_{u,t}} \leftarrow \mathbf{W_{k_uC}} C_t + \mathbf{b_{k_uC}} \tag{4.35}$$

$$\beta_{u,t} \leftarrow relu(\mathbf{W}_{\beta_\mathbf{u}\mathbf{C}} C_t + \mathbf{b}_{\beta_\mathbf{u}\mathbf{C}}) \tag{4.36}$$

$$g_{u,t} \leftarrow \sigma(\mathbf{W_{g_uC}} C_t + \mathbf{b_{g_uC}}) \tag{4.37}$$

$$\mathbf{s_{u,t}} \leftarrow \sigma(\mathbf{W_{s_uC}} C_t + \mathbf{b_{s_uC}}) \tag{4.38}$$

$$\gamma_{u,t} \leftarrow relu(\mathbf{W}_{\gamma_\mathbf{u}\mathbf{C}} C_t + \mathbf{b}_{\gamma_\mathbf{u}\mathbf{C}}) \tag{4.39}$$

The updated hidden layer of neurons, then, also produces an erase and an add vector using equations 4.31 and 4.32 which, with the updated write weighting vector, are used to update the memory matrix. This process goes on until the input sequence gets exhausted. Once an external output P_t has been produced for each element X_t in the input sequence, an error or loss $Loss_t$ is calculated between the external outputs P_t and the targets Y_t. Here, the loss function is chosen based on the task. Since we are assuming that the values in vectors of target sequence lie between 0 and 1, therefore, the binary cross entropy loss will be an appropriate choice.

$$Loss_t = binary\_crossentropy(P_t, Y_t) \tag{4.40}$$

Let P be the matrix of predicted external output vectors $[P_1, P_2, \ldots P_T]$ and Y be the matrix of target external output vectors $[Y_1, Y_2, \ldots Y_T]$, then

one can compute the loss corresponding to the complete output sequence by computing the binary cross entropy error (refer section 3.2.3) between P and Y.

$$overall\_loss, L = binary\_crossentropy(P, Y) \tag{4.41}$$

Now, the derivative of the *overall_loss* L is computed with respect to each parameter of the model by making use of the chain rule of partial derivatives (refer section 3.2.4) and the derivative equations in this chapter. After computing the derivative of L with respect to each parameter, the parameters of the model are updated using the RMSprop version of gradient descent as described in section 3.3.

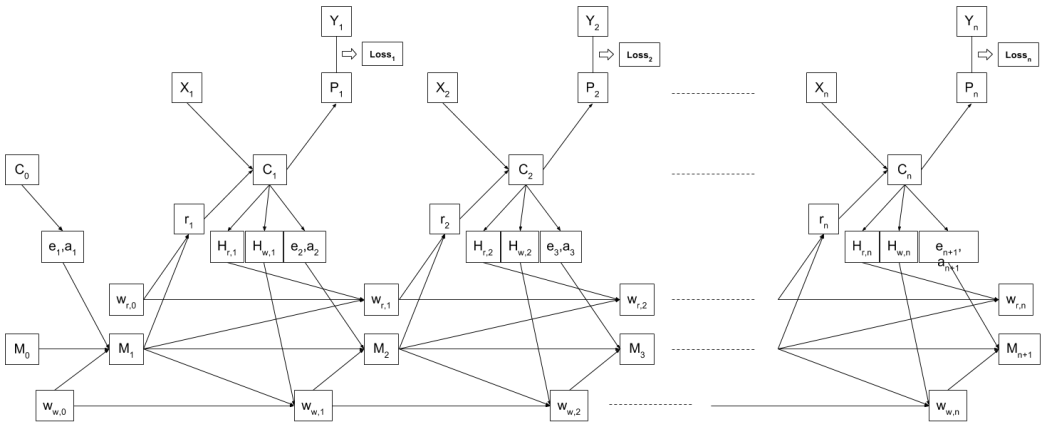

Figure 4.3: **Neural Turing Machine learning network.**

4.5 Experiments

We developed two versions of NTM, one with a single head which is used to read from as well as write to the memory and the second with two separate heads, one for reading from the memory and other for writing into the memory. We tested these two versions of NTM on tasks described below.

4.5.1 Copy Task

The memory matrix size was set to 128×20 i.e. 128 memory slots each of length 20. For training the first version of NTM, we took random binary sequences of length less than or equal to 20 and for the second version of NTM, we took binary sequences of length less than or equal to 5. One such input output sequence is shown in Fig. 4.4.

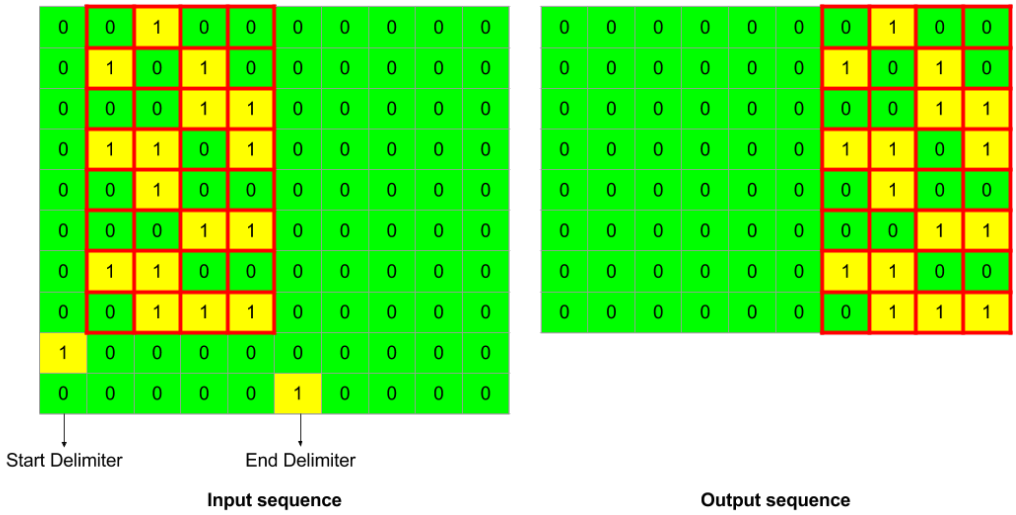

Figure 4.4: **Input and output sequence in copy task.**

Our first version of NTM was able to correctly predict the external output sequence with input sequences of length 116 and our second version of NTM was able to correctly predict the external output sequence with input sequences of length 34. Fig. 4.5 and Fig. 4.6 shows the learning curves in the two cases. Fig. 4.7 and Fig. 4.8 shows the corresponding plots which includes the visualization of the interaction between the controller and the memory matrix.

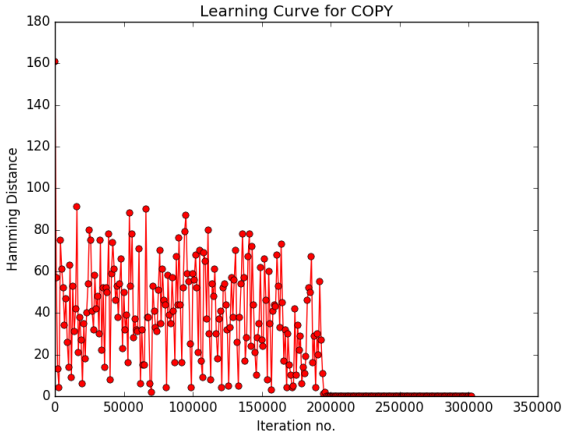

Figure 4.5: **Learning curve in copy task with our first version of NTM.**

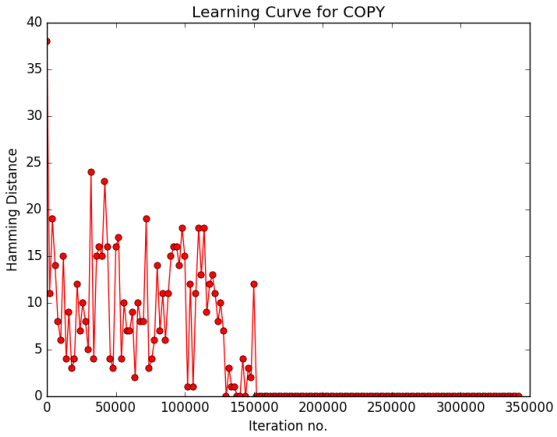

Figure 4.6: **Learning curve in copy task with our second version of NTM.**

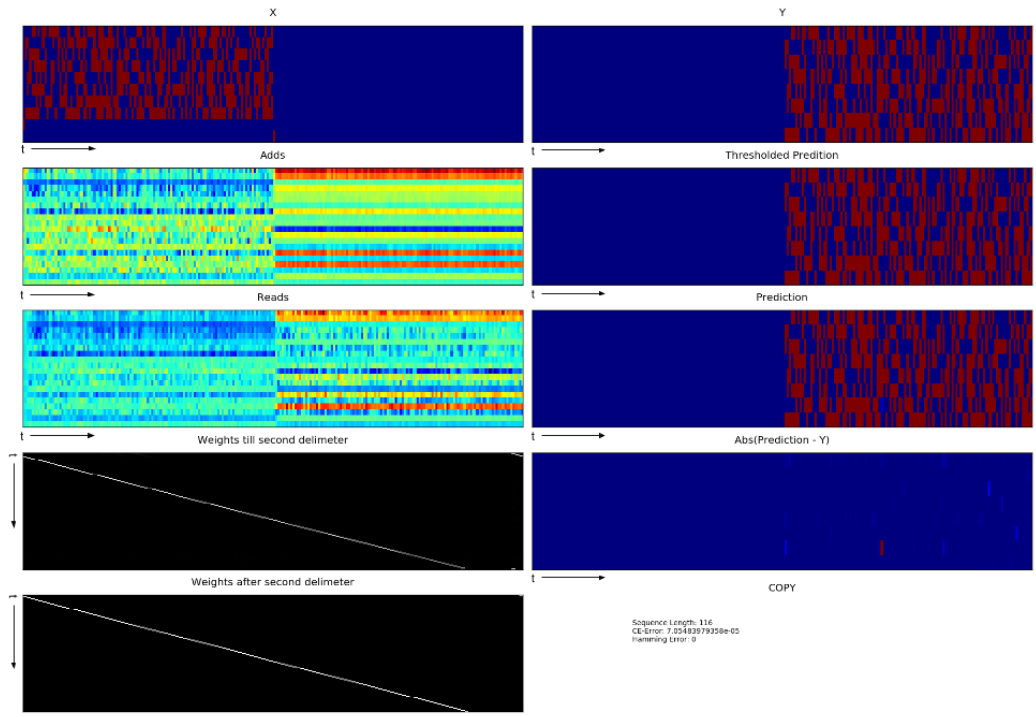

Figure 4.7: **Graphical visualization of copy task with first version of NTM.** External Input Sequence (X), Target Sequence (Y), Prediction Sequence (Prediction), Thresholded Prediction Sequence (Thresholded Prediction), Error (Abs(Prediction-Y)), Read vectors (Reads), Add vectors (Adds) and Weightings before and after ending delimiter

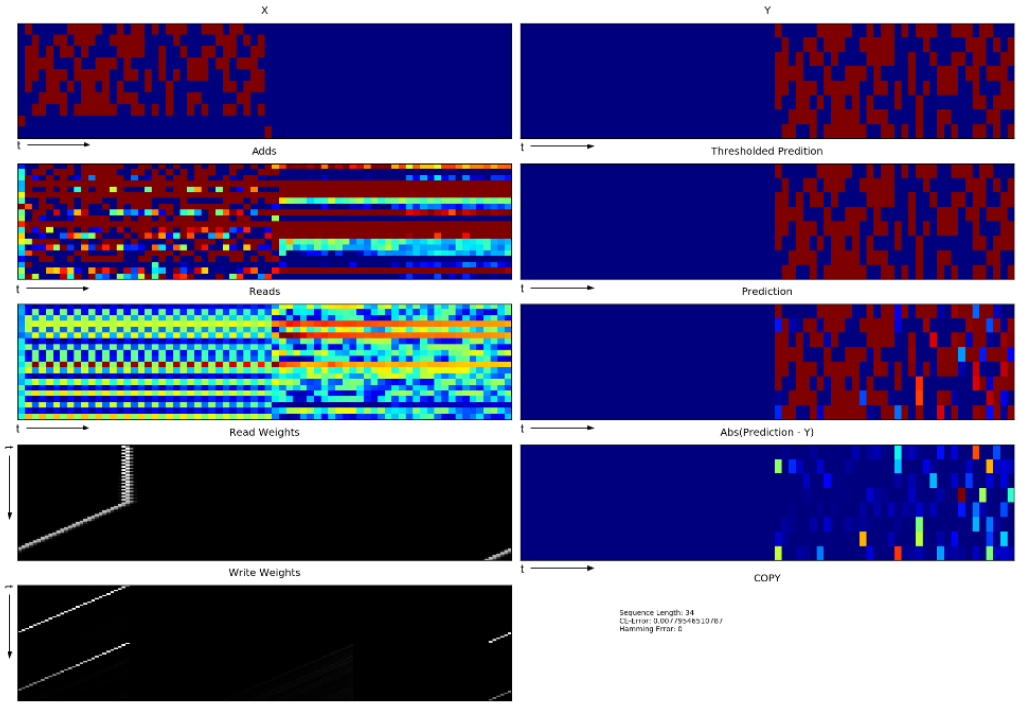

Figure 4.8: **Graphical visualization of copy task with second version of NTM.** External Input Sequence (X), Target Sequence (Y), Prediction Sequence (Prediction), Thresholded Prediction Sequence (Thresholded Prediction), Error (Abs(Prediction-Y)), Read vectors (Reads), Add vectors (Adds), Read Weighting vectors (Read Weights) and Write Weighting vectors (Write Weights)

One can infer from the graphical visualization of the interaction between the controller and the memory matrix that both versions of NTM have successfully learnt the following algorithm:

initialise: move head to location next to start delimiter
while end delimiter not seen **do**
    receive input vector
    write input to head location
    increment head location by 1
end while
return head to start location
while true **do**
    read output vector from head location
    emit output
    increment head location by 1
end while

Chapter 5

End to End Memory Networks for QA Task

This chapter explores another RAM based model called end to end memory networks and its application to question answering task. End to end memory networks is a novel model that has a recurrent neural network architecture where the recurrence reads from a possibly large external memory multiple times before outputting a symbol. The model being differentiable end to end with respect to the parameters is hence trainable end to end and requires less supervision than the memory networks discussed in [23]. Section 6.1 describes a single layered end to end memory network. Then, section 6.2 describes a multi-layered version. Section 6.3 briefly states the loss computation and training and finally, section 6.4 shows experimentation and results with end to end memory networks in question answering task.

5.1 Single Layered End to End Memory Networks

Suppose $\{s_1, s_2, \ldots s_n\}$ be an input sequence of sentences forming a comprehension where each sentence is a sequence of words from a vocabulary V, q be a query based on the comprehension which is again a sentence formed of words from V and y be the target one-word answer to the query q which belongs to the comprehension itself. A single layer of end to end memory networks, Fig. 5.1, comprises of three components: input memory representation, output memory representation and final answer prediction.

Input Memory Representation : Each word in the input sequence of sentences, query sentence and the target one-word answer is converted to the bag of words representation i.e. a vector of dimension equal to the number of words in the vocabulary with value of 1 for the corresponding word and 0 for all other words. Then the a memory vector $\mathbf{m_i}$ is computed corresponding to each sentence s_i using an embedding $\mathbf{A}$ as follows:

$$\mathbf{m_i} \leftarrow \sum_j \mathbf{l_j}.\mathbf{A}\mathbf{x_{ij}} \tag{5.1}$$

where

$$\mathbf{x_{ij}} = BAG\_OF\_WORDS(s_{ij}) \tag{5.2}$$

and

$$l_{kj} \leftarrow \left(1 - \frac{j}{J}\right) - \left(\frac{k}{d}\right)\left(1 - \frac{2j}{J}\right) \tag{5.3}$$

where J is the total number of words in the sentence s_i and d is the dimension of embedding $\mathbf{A}$. Similarly, the internal representation $\mathbf{u}$ of the query q is computed using embedding $\mathbf{B}$. Then, a weighting vector $\mathbf{p}$ over the memory

vectors is computed where p_i denotes the similarity of the memory vector $\mathbf{m_i}$ and the internal representation of query $\mathbf{u}$. Higher the value of p_i implies more relevant the sentence s_i is in answering query q.

$$p_i \leftarrow Softmax(u^T m_i) \tag{5.4}$$

Output Memory Representation : The output vectors $\mathbf{c_i}$ corresponding to each sentence s_i are computed using an embedding $\mathbf{C}$ in the same manner as the memory vectors are computed. Then, an intermediate response $\mathbf{o}$ is calculated as the weighted sum of the output vectors weighted by the weighting vector $\mathbf{p}$.

$$\mathbf{o} \leftarrow \sum_i p_i \mathbf{c_i} \tag{5.5}$$

Final answer prediction : The output vector $\mathbf{o}$ is summed with the internal representation of query $\mathbf{u}$ and the resulting vector is decoded using a decoding matrix $\mathbf{W}$. The decoded vector is then softmax-ed to produce the final response $\hat{\mathbf{a}}$.

$$\hat{\mathbf{a}} \leftarrow Softmax(\mathbf{W}(\mathbf{o} + \mathbf{u})) \tag{5.6}$$

The error or loss in the final response is computed using the binary crossentropy loss function (refer section 3.2.3) with $\hat{\mathbf{a}}$ and $BAG\_OF\_WORDS(y)$ as arguments. Note that the parameters of the single layered end to end memory network are $\mathbf{A}$, $\mathbf{B}$, $\mathbf{C}$ and $\mathbf{W}$.

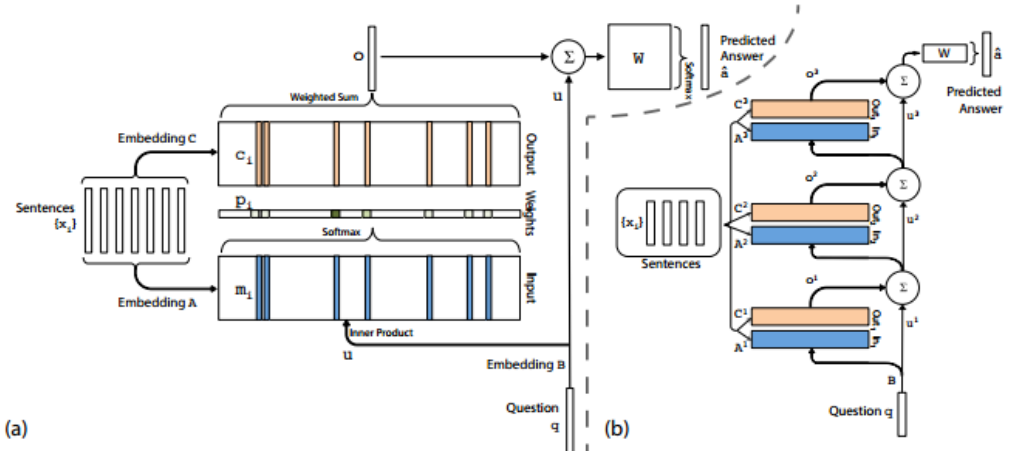

Figure 5.1: **End to End Memory Networks Architecture.[20]** (a) A single layer version. (b) A three layer version.

5.2 Multi-Layered End to End Memory Networks

The single layered end to end memory networks cannot predict correct answers to questions which require transitive implications. For example, consider the following comprehension,

1. Jerry is a mouse.

2. mice are afraid of wolves.

Now, consider the following query, "Jerry is afraid of whom?". This query requires transitive implication that "Jerry ⇒ mouse", "mice ⇒ afraid of wolves" ⇒ "Jerry ⇒ afraid of wolf". Multiple layers (or hops) are introduced to answer such questions.

Multi-layered end to end memory network is formed by stacking the single layers. Fig. 5.1 shows a single and a multi-layered (3 layered) end to end memory network. The parameters of the multi-layered end to end memory

network are $\mathbf{W}$, $\mathbf{B}$, $\mathbf{A_i}$ and $\mathbf{C_i}$ where i goes from 1 to the number of hops or layers. Thus, with multiple layers, the parameter space of the model blows up and to overcome this issue, a popular technique called **parameter tying** is applied. There are two types of parameter tying that can be used,

1. RNN-like: Here, $\mathbf{A_1} = \mathbf{A_2} = \ldots \mathbf{A_n}$ and $\mathbf{C_1} = \mathbf{C_2} = \ldots \mathbf{C_n}$.
2. Adjacent: Here, $\mathbf{C_{k+1}} = \mathbf{A_k}$, $\mathbf{B} = \mathbf{A_1}$ and $\mathbf{W^T} = \mathbf{C_n}$.

We used adjacent parameter tying in our implementation.

5.3 Training End to End Memory Networks

After computing the output $\hat{a}$ and given the target y, loss L is computed as (refer section 3.2.3),

$$L = binary\_crossentropy(\hat{a}, y) \tag{5.7}$$

The derivative of L with respect to each parameter is, then, computed (refer section 3.2.4) and the parameters are updated using stochastic gradient descent (refer section 3.3).

5.4 Experiments

To test the performance of end to end memory networks, we trained the model on the 20 tasks in Babi-Project Dataset [20], and computed the confusion matrices, precision, recall and f1-score [25] [24] for each task and got consistent results as in [20].

5.4.1 Question Answering based on Babi-Project Dataset

Here, we show the results in some of the tasks from Babi-Project Dataset.

Task 1: Single Supporting Fact

1. **Sample Input**
   (a) Comprehension
      i. Mary moved to the bathroom.
      ii. John went to the hallway.
   (b) Query: Where is Mary?
2. **Sample Output**
   (a) Answer: bathroom
   (b) Sentences used: i
3. **Confusion Matrix**

	C1	C2	C3	C4	C5	C6
C1	148	0	0	0	0	0
C2	0	169	0	1	0	0
C3	0	0	184	0	0	0
C4	0	0	0	153	0	0
C5	0	0	0	0	155	1
C6	1	0	0	0	0	180

4. **Score Matrix**

Class	Precision	Recall	F1-Score
C1	0.99	1.00	1.00
C2	1.00	0.99	1.00
C3	1.00	1.00	1.00
C4	0.99	1.00	1.00
C5	1.00	0.99	1.00
C6	0.99	0.99	0.99

Task 5: Three-args relation

1. **Sample Input**
   (a) Comprehension
      i. Bill travelled to the office.
      ii. Bill picked up the football there.
      iii. Bill went to the bedroom.
      iv. Bill gave the football to Fred.
   (b) Query: What did Bill give to Fred?
2. **Sample Output**
   (a) Answer: football
   (b) Sentences used: iv
3. **Confusion Matrix**

	C1	C2	C3	C4	C5	C6	C7
C1	129	8	12	13	0	0	0
C2	20	159	4	4	0	0	0
C3	5	15	94	14	0	0	0
C4	12	21	12	144	0	0	0
C5	0	0	0	0	135	1	0
C6	0	0	0	0	2	86	4
C7	0	0	0	0	3	0	95

4. **Score Matrix**

Class	Precision	Recall	F1-Score
C1	0.78	0.80	0.79
C2	0.78	0.85	0.82
C3	0.77	0.73	0.75
C4	0.82	0.76	0.79
C5	0.96	0.99	0.98
C6	0.99	0.93	0.98
C7	0.96	0.97	0.96

Task 10: Indefinite Knowledge

1. **Sample Input**

   (a) Comprehension

      i. Fred is either in the school or the park.

      ii. Mary went back to the office.

   (b) Query: Is Mary in the office?

2. **Sample Output**

(a) Answer: yes

(b) Sentences used: ii

3. **Confusion Matrix**

	C1	C2	C3
C1	107	34	5
C2	24	335	76
C3	4	44	363

4. **Score Matrix**

Class	Precision	Recall	F1-Score
C1	0.79	0.73	0.76
C2	0.81	0.77	0.79
C3	0.82	0.88	0.85

Task 15: Basic Deduction

1. **Sample Input**

(a) Comprehension

i. Mice are afraid of wolves.

ii. Gertrude is a mouse.

iii. Cats are afraid of sheep.

iv. Winona is a mouse.

v. Sheep are afraid of wolves.

vi. Wolves are afraid of cats.

vii. Emily is a mouse.

viii. Jessica is a wolf.

(b) Query: What is gertrude afraid of?

2. **Sample Output**

(a) Answer: wolf

(b) Sentences used: ii, i

3. **Confusion Matrix**

	C1	C2	C3	C4
C1	209	0	0	0
C2	0	238	0	0
C3	0	0	204	0
C4	0	0	0	341

4. **Score Matrix**

Class	Precision	Recall	F1-Score
C1	1.00	1.00	1.00
C2	1.00	1.00	1.00
C3	1.00	1.00	1.00
C4	1.00	1.00	1.00

Task 20: Agents Motivations

1. **Sample Input**

(a) Comprehension

i. Sumit is tired.

(b) Query: Where will sumit go?

2. **Sample Output**

(a) Answer: bedroom

(b) Sentences used: i

3. **Confusion Matrix**

	C1	C2	C3	C4	C5	C6	C7
C1	96	0	0	0	0	0	0
C2	0	158	0	0	0	0	0
C3	0	0	93	0	0	0	0
C4	0	0	0	154	0	0	0
C5	0	0	0	0	179	0	0
C6	0	0	0	0	0	152	0
C7	0	0	0	0	3	0	160

4. **Score Matrix**

Class	Precision	Recall	F1-Score
C1	1.00	1.00	1.00
C2	1.00	1.00	1.00
C3	1.00	1.00	1.00
C4	1.00	1.00	1.00
C5	1.00	1.00	1.00
C6	1.00	1.00	1.00
C7	1.00	1.00	1.00

Chapter 6

Generic Game Playing Agent using Deep Reinforcement Learning with RAM

This chapter provides with the background material and literature review of reinforcement learning for building game playing agents. Section 6.1 deals with markov decision process with relevant theorems and their proofs forming the base of reinforcement learning, section 6.2 describes the policy value and section 6.3 describes the state-action value function and an algorithm called Q-learning for approximating the optimal state-action value function. Then, section 6.4 deals with building a generic game playing agent (a single model capable of learning to play any game) using the concepts of neural networks specifically, convolutional neural networks from deep learning and markov decision process from reinforcement learning. Then, in section 6.5, some ideas are presented for building an agent that can not only learn to play any game but can also provide reasoning behind the strategy (the sequence of actions) being played by the agent. We start with a brief introduction of

reinforcement learning.

In reinforcement learning, the learner does not receive a labeled dataset, in contrast with supervised learning. Instead, he collects information through a course of actions by interacting with the environment. In response to an action, the learner or agent receives two types of information: his current state in the environment and a real-valued reward, which is specific to the task and its corresponding goal. Fig. 6.1 shows the general scenario of reinforcement learning. Unlike supervised learning, there is no fixed distribution according to which the instances are drawn; the choice of a policy defines the distribution.

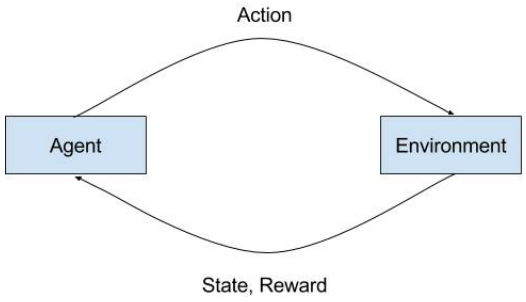

Figure 6.1: **Representation of general scenario of reinforcement learning.**

The objective of the agent is to maximize his reward and thus to determine the best course of actions, or policy, to achieve the objective. **However, the information he receives from the environment is only the immediate reward corresponding to the action just taken. No future or long-term reward feedback is provided by the environment.** The agent also faces the dilemma between exploring unknown states and actions to gain more information about the environment and exploiting the information already collected to optimize his reward. Two main settings are possible:

1. Environment model is known to agent. Then the problem is reduced

to **planning**.

2. Environment model is unknown to agent. Then, he faces **learning** problem. This will be our main concern in this thesis.

Note that environment model comprises of the state transition probability distribution and the reward probability distribution which are defined in the next section.

6.1 Markov Decision Process

A Markov Decision Process (MDP) is defined by:

1. Set of states, S.
2. Set of actions, A.
3. Start state, $s_0 \in S$.
4. Reward Probability

$$P(r_{t+1}|s_t, a_t) \ \ where\, r_{t+1} = r(s_t, a_t) \tag{6.1}$$

5. State transition probability

$$P(s_{t+1}|s_t, a_t) \ \ where\, s_{t+1} = \delta(s_t, a_t) \tag{6.2}$$

We also define $\pi : S \rightarrow A$ as the policy function mapping a state S to an action A. In a discrete time model, actions are taken at a set of decision epochs $\{0, \ldots, T\}$. We deal with infinite horizon i.e. T tends to infinity. Then, the agent's objective is to find a policy that maximizes his expected reward.

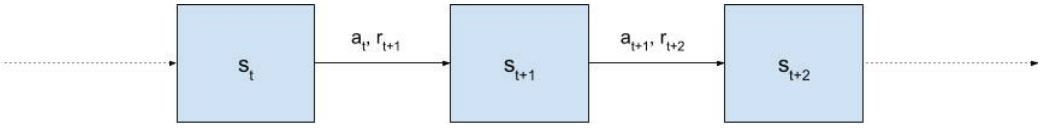

Figure 6.2: **Illustration of states and transitions of MDP at different times.**

6.2 Policy Value

We define the policy value $V_\pi(s)$ in the case of infinite horizon as the expected reward of the agent starting at state s and following policy π i.e.

$$V_\pi(s) = E\Big[\sum_{\tau=0}^{T-t} \gamma^\tau r(s_{t+\tau}, \pi(s_{t+\tau}))|s_t = s\Big] \tag{6.3}$$

where T tends to infinity.

Theorem 6.2.1. *The policy value $V_\pi(s)$ obey the following system of linear equation (Bellman's equation):*

$$\forall s \in S,\ \ V_\pi(s) = E[r(s, \pi(s))] + \gamma \sum_{s'} Pr\Big[s'|s, \pi(s)\Big] V_\pi(s') \tag{6.4}$$

Proof.

$$V_\pi(s) = E\Big[\sum_{\tau=0}^{T-t} \gamma^\tau r(s_{t+\tau}, \pi(s_{t+\tau}))|s_t = s\Big] \tag{6.5}$$

$$V_\pi(s) = E[r(s, \pi(s, \pi(s))] + \gamma E\Big[\sum_{\tau=0}^{T-t} \gamma^\tau r(s_{t+1+\tau}, \pi(s_{t+1+\tau}))|s_t = s\Big] \tag{6.6}$$

$$V_\pi(s) = E[r(s, \pi(s, \pi(s))] + \gamma E[V_\pi(\delta(s, \pi(s)))] \tag{6.7}$$

The second term in equation 6.7 is the second term of equation 6.4. □

The Bellman's equation can also be written as:

$$\mathbf{V} = \mathbf{R} + \gamma \mathbf{P}\mathbf{V} \tag{6.8}$$

where $\mathbf{P}$ is the state transition probability matrix, $\mathbf{R} = E[r(s, \pi(s))]$, γ is the discount for future rewards and $\mathbf{V}$ is the unknown policy value matrix

Theorem 6.2.2. *For a finite MDP, Bellman's equation admits a unique solution given by*

$$\boldsymbol{V}_0 = (\boldsymbol{I} - \gamma \boldsymbol{P})^{-1} \boldsymbol{R} \tag{6.9}$$

Proof. One just needs to show that $(\mathbf{I} - \gamma\mathbf{P})$ is invertible and the remaining part is trivial. Note that the infinite norm of P is:

$$\|\mathbf{P}\|_\infty = \max_s \sum_{s'} |P_{ss'}| = \max_s \sum_{s'} Pr\left[s^{'}|s, \pi(s)\right] = 1 \tag{6.10}$$

This implies that $\|\gamma\mathbf{P}\|_\infty = \gamma < 1$. The Eigenvalues of $\mathbf{P}$ are all less than 1 and $(\mathbf{I} - \gamma\mathbf{P})$ is invertible. □

The optimal policy value at a given state s is thus given by

$$V_{\pi^*}(s) = max_\pi V_\pi(s) \tag{6.11}$$

6.3 State-Action Value Function and Q-Learning

We define the optimal state-action value function Q^* for all $(s, a) \in S \times A$ as the expected return for taking action $a \in A$ at state $s \in S$ and then following the optimal policy:

$$Q^*(s, a) = E[r(s, a)] + \gamma \sum_{s' \in S} Pr[s^{'}|s, a]V^*(s^{'}) \tag{6.12}$$

Algorithm 1 Q-learning Pseudocode

```
procedure Q-LEARNING(π)
    Q ← Q_0                                          ▷ initialization e.g, Q_0 = 0
    for t ← 0 to T do
        s ← SELECT_STATE()
        for each step of epoch t do
            a ← SELECTACTION(π, s)                   ▷ Policy π from ϵ-greedy
            r' ← REWARD(s, a)
            s' ← NEXTSTATE(s, a)
            Q(s,a) ← Q(s,a) + α[r' + γ max_a' Q(s',a') − Q(s,a)]
            s ← s'
    return Q
```

Figure 6.3: **Q-Learning algorithm [19]**

One can observe that the optimal policy can be given by

$$\forall s \in S, \;\; \pi^*(s) = argmax_{a \in A} Q^*(s, a) \tag{6.13}$$

Thus, the knowledge of the state-vale function Q^* is sufficient for the agent to determine the optimal policy, without any direct knowledge of the reward or transition probabilities.

Q-learning algorithm descirbed in Fig. 6.3 is used for learning (approximating) the optimal state-action value function. A proof of the algorithm can be found in [19].

6.4 Deep Reinforcement Learning

In real world scenarios, the number of states are very large (infinite) which prevents us from representing the state-action value function as a matrix. This is the case with the atari games too. The number of states (configuration of the screen of the game) are infinite and hence, we represent our state-value function using a convolution neural network (instead of a matrix), which takes state-information (screen) as input and outputs scores representing the

expected reward corresponding to each action. The optimal action is the one corresponding to maximum score.

So, our optimal state-action value function will be given by

$$Q^*(s,a;\theta^*) = E[r(s,a);\theta^*] + \gamma \sum_{s^{'} \in S} Pr[s^{'}|s,a;\theta^*]V^*(s^{'}) \tag{6.14}$$

where,
θ^* represents the learnt parameters of our neural network. Also, the optimal policy is given by

$$\forall s \in S,\ \ \pi^*(s) = argmax_{a \in A} Q^*(s,a;\theta^*) \tag{6.15}$$

6.4.1 Model

Our model based on [18] is a convolution neural network which takes last 4 preprocessed frames of the screen as input (84x84x4), applies 32 convolution filters with kernel size (8,8) and stride (4,4) and rectifier non-linearity to get first hidden layers of neurons which is further convolved using 64 filters with kernel size (4,4) and stride (2,2) and rectifier non-linearity to get the second hidden layer of neurons which is further convolved using 64 filters with kernel size (3,3) and stride (1,1) and rectifier non-linearity to get third hidden layer of neurons which is fully connected to a layer of 512 neurons with rectifier non-linearity which is again fully connected to output layer of size equal to number of actions in the minimum legal action set. Each output unit represents expected reward obtained by the agent if it performs the action corresponding to that output unit. Here, the preprocessing involves conversion to grayscale and taking the maximum of the value of pixel in current frame and the last frame to avoid flickering effect in the games where

some objects occur in even or odd frames only.

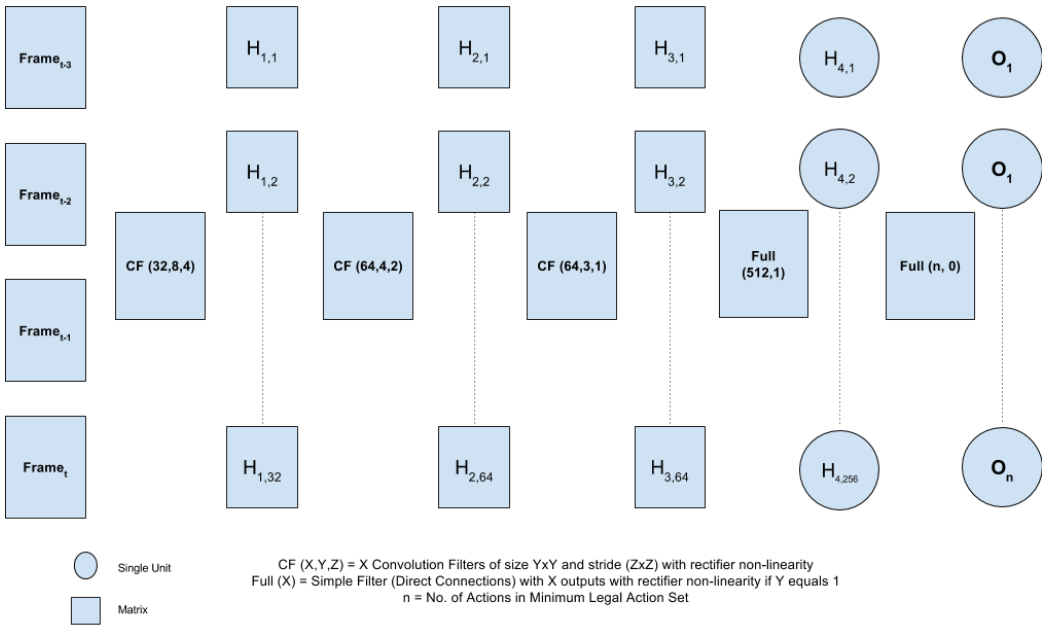

Figure 6.4: **Convolution neural network as the mind of the agent.**

6.4.2 Training Details

The parameter ϵ in our model simulates the exploration vs exploitation trade-off. A value of 1 for ϵ implies that the agent will make random actions hence explore the environment and a value of 0 for ϵ implies that the agent will chooses actions based on the model only and hence exploits the acquired information. While training, we decrease ϵ with some appropriate decay rate. So, our agent initially starts by making random actions and collects the information in the form of following tuple: $(S_t, a_t, r_t, S_{t+1}, g_t)$ where S_t denotes the last four states including the current state, a_t denotes the action taken in current state, r_t denotes the reward received, S_{t+1} denotes the last four frames including the next state that the agent got into and g_t is a boolean denoting whether the game is over or not. Also, the parameters of the con-

volution neural network that acts as the brain of the agent is initialized with random values. After making a move, the agent selects a random batch of information from the collected information and makes a gradient descent step on the euclidean loss between targets and predictions with respect to network parameters θ where the inputs and targets to convolution neural network are described below:

Let $j = 1, \ldots, m$ represents the indices of the examples in the sampled batch. Then, input is:

$$I_j = S_t^j \tag{6.16}$$

and target is (for action a_t^j only):

$$y^j = r_t^j \mathbb{I}(g_t^j = True) + (r_t^j + \gamma max_{a'} Q(S_{t+1}^j, a'; \theta)) \mathbb{I}(g_t^j = False) \tag{6.17}$$

The discount value, γ is set to 0.99. We took a batch size of 32, initial ϵ value of 1, and final ϵ value of 0.1. While choosing an action, a uniform random number is generated between 0 and 1. If the generated number is less than ϵ then a random action is taken otherwise an action prdicted by the state-action value function represented by CNN is taken. We used the rmsprop (refer section 3.3) version of gradient descent for updating the parameters. A learning rate of 0.00025, momentum decay rate of value 0.95, and velocity decay rate of value 0.95 were used in rmsprop.

6.4.3 Results

Fig. 6.5 shows the epoch number versus the total reward and mean Q-value received by the agent in the atari game *Breakout* during testing. Each test epoch comprises of a fixed number of frames to make the rewards in different epoch comparable. The increase in total reward per epoch shows that our

agent learnt to play the game. Fig. 6.6 shows the learning curve for the game *Breakout*. As expected, the learning curve doesn't give much insight on whether the agent has learnt to play in an optimal manner while the mean Q-value per epoch does. Then, Fig. 6.7 shows a sequence of frames while the agent has learnt to play the game.

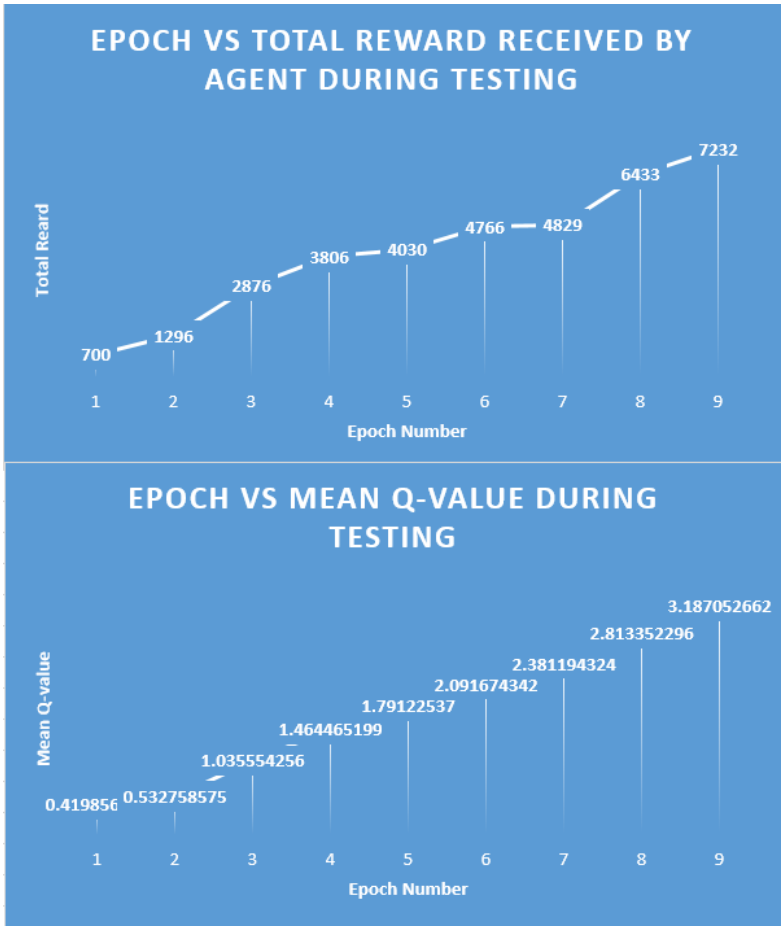

Figure 6.5: **Epoch number versus the total reward and mean Q-value received by the agent in the game *Breakout* during testing**

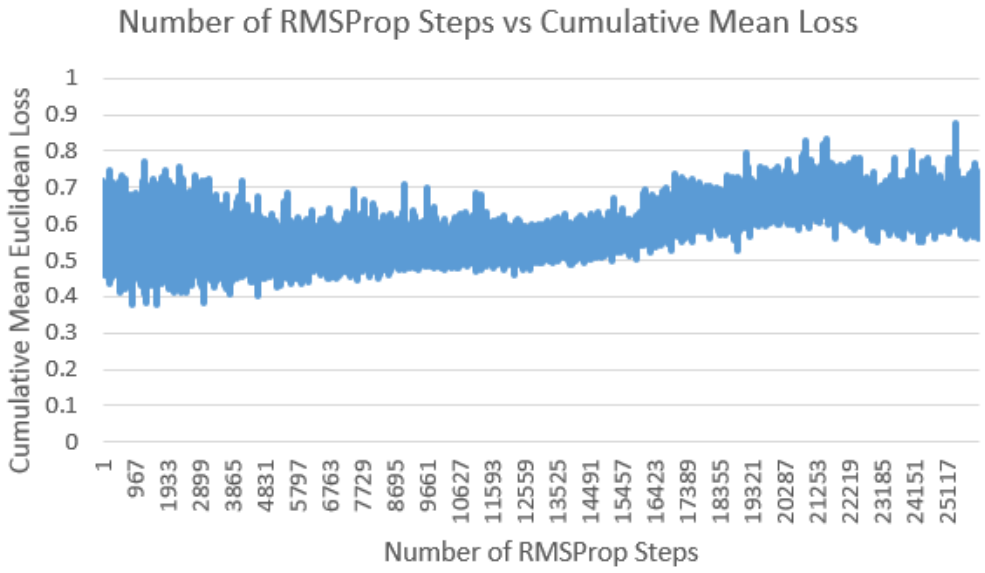

Figure 6.6: **Learning curve for the game *Breakout***

6.5 Deep Reinforcement Learning with RAM

Consider the game *Breakout* and the sequence of frames observed by our model in Fig. 6.7. Now, the model/agent based on deep reinforcement learning moves to the left. But why? A human agent might answer that since the ball is moving towards left the action taken is to move left. This is the part that the model doesn't answer. It has learnt some set of parameters that once fitted in the convolutional neural network predicts optimal action to take. But a RAM-based agent is capable of providing the reason that "since the ball is moving towards left, I should also move towards left to prevent death".

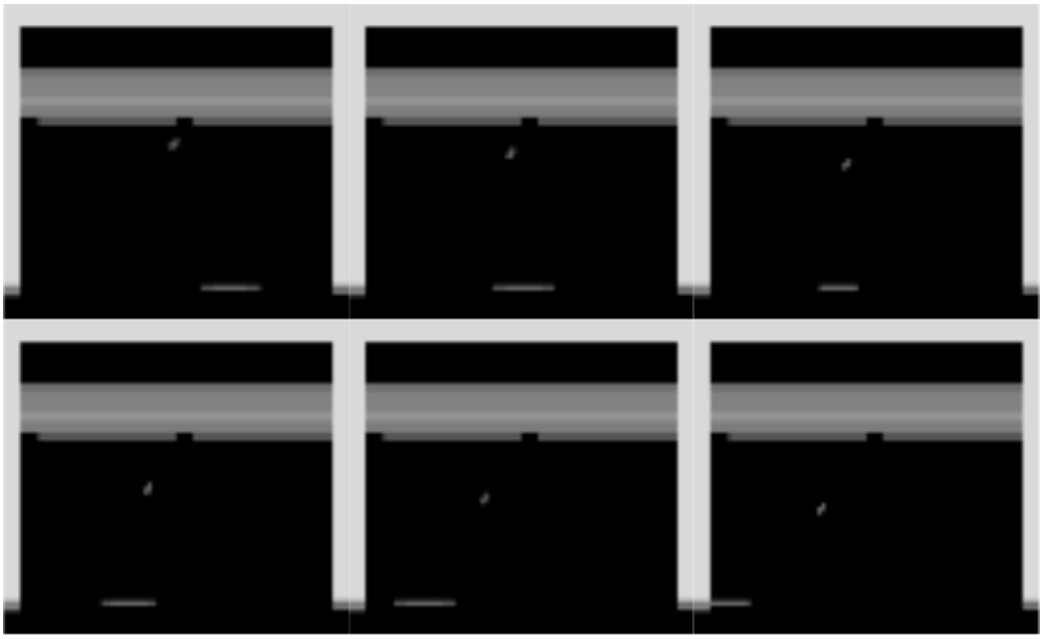

Figure 6.7: **A sequence of frames while the agent has learnt to play the game**

6.6 Model Ideas

Usually, the optimal decision taken by a game playing agent depends on a part of his state. For example, consider the atari game *Breakout* and let the screen of the game represents the state of the agent. Then, the optimal strategy of the agent for playing *Breakout* must depend on the part of the screen of the game displaying the ball and the slider. Thus, a RAM-based agent for playing *Breakout* should have attention on these two areas of the screen. If the graphical visualization of the attention mechanism between the controller and the memory of the RAM-based agent for playing *Breakout* shows that the two areas (ball and slider) of the screen are in focus, then it can be inferred that the agent is making decisions (actions) based on the position of the ball and the slider.

We tested the same model described in previous section after replacing the convolutional neural network with a neural turing machine which takes a sequence of last 4 frames as external input and produces a score corresponding to each legal action that represents the expected future reward as external output. The resulting model produced results consistent with those shown in Fig. 6.5. But we missed a very important aspect. NTM has read and write heads that focus on the memories not on the content of the memories. We require attention on the content of the memories not on the location of the memories. So, the most important change that one must introduce in NTM for building game playing agents that provide reasoning is to,

"Introduce weighting on the content of the memories."

Some other ideas of reasonable changes can be:

1. Using a technique called Clustering on Subsets of Attributes (COSA) [6] for the content addressing module so that the memories can also be focused based on the partial contents. The motivation behind this change follows from the way human beings are capable of detecting partially similar objects.

2. Number of allowed shifts in Neural Turing Machine should be equal to the number of memory slots in the memory so that the focus can shift to memory slot at arbitrary distance from the memory slot in focus.

Bibliography

[1] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. Neural machine translation by jointly learning to align and translate. *CoRR*, abs/1409.0473, 2014.

[2] Lon Bottou. Stochastic gradient descent tricks.

[3] Denny Britz. Recurrent neural networks tutorial, part 1 introduction to rnns.

[4] Kyunghyun Cho, Bart van Merrienboer, Çaglar Gülçehre, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. Learning phrase representations using RNN encoder-decoder for statistical machine translation. *CoRR*, abs/1406.1078, 2014.

[5] John Duchi, Elad Hazan, and Yoram Singer. Adaptive subgradient methods for online learning and stochastic optimization. Technical Report UCB/EECS-2010-24, EECS Department, University of California, Berkeley, Mar 2010.

[6] Jerome H. Friedman and Jacqueline J. Meulman. Clustering objects on subsets of attributes. *Journal of the Royal Statistical Society*, 66:815–849, 2004.

[7] Xavier Glorot and Yoshua Bengio. Understanding the difficulty of training deep feedforward neural networks. In *International conference on artificial intelligence and statistics*, pages 249–256, 2010.

[8] Alex Graves. *Supervised sequence labelling*. Springer, 2012.

[9] Alex Graves, Greg Wayne, and Ivo Danihelka. Neural turing machines. *CoRR*, abs/1410.5401, 2014.

[10] Barbara Hammer. On the approximation capability of recurrent neural networks. *Neurocomputing*, 31(1-4), 2000.

[11] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. *CoRR*, abs/1502.01852, 2015.

[12] Sepp Hochreiter, Yoshua Bengio, Paolo Frasconi, and Jürgen Schmidhuber. Gradient flow in recurrent nets: the difficulty of learning long-term dependencies.

[13] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

[14] K. Hornik, M. Stinchcombe, and H. White. Multilayer feedforward networks are universal approximators. *Neural Netw.*, 2(5):359–366, July 1989.

[15] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *CoRR*, abs/1412.6980, 2014.

[16] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in Neural Information Processing Systems*, page 2012.

[17] Fandong Meng, Zhengdong Lu, Zhaopeng Tu, Hang Li, and Qun Liu. Neural transformation machine: A new architecture for sequence-to-sequence learning. *CoRR*, abs/1506.06442, 2015.

[18] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A. Rusu, Joel Veness, Marc G. Bellemare, Alex Graves, Martin Riedmiller, Andreas K. Fidjeland, Georg Ostrovski, Stig Petersen, Charles Beattie, Amir Sadik, Ioannis Antonoglou, Helen King, Dharshan Kumaran, Daan Wierstra, Shane Legg, and Demis Hassabis. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533, 02 2015.

[19] Mehryar Mohri, Afshin Rostamizadeh, and Ameet Talwalkar. *Foundations of Machine Learning*. The MIT Press, 2012.

[20] Sainbayar Sukhbaatar, Arthur Szlam, Jason Weston, and Rob Fergus. Weakly supervised memory networks. *CoRR*, abs/1503.08895, 2015.

[21] T. Tieleman and G. Hinton. Lecture 6.5—RmsProp: Divide the gradient by a running average of its recent magnitude. COURSERA: Neural Networks for Machine Learning, 2012.

[22] Peilu Wang, Yao Qian, Frank K. Soong, Lei He, and Hai Zhao. Part-of-speech tagging with bidirectional long short-term memory recurrent neural network. *CoRR*, abs/1510.06168, 2015.

[23] Jason Weston, Sumit Chopra, and Antoine Bordes. Memory networks. *CoRR*, abs/1410.3916, 2014.

[24] Wikipedia. F1 score — wikipedia, the free encyclopedia, 2016. [Online; accessed 22-April-2016].

[25] Wikipedia. Precision and recall — wikipedia, the free encyclopedia, 2016. [Online; accessed 22-April-2016].