Advancing Neural Turing Machines: Learning a Solution to the Shortest Path Problem

Author:

Conor SEXTON

Supervisor:

Dr. Rozenn DAHYOT

A dissertation submitted in fulfilment of the requirements

for the degree of Master in Computer Science

in the

School of Computer Science and Statistics

Trinity College Dublin

Coláiste na Tríonóide, Baile Átha Cliath

The University of Dublin

Submitted to the University of Dublin, Trinity College, May, 2017

Declaration of Authorship

I, Conor SEXTON, declare that the following dissertation, except where otherwise stated, is entirely my own work; that it has not previously been submitted as an exercise for a degree, either in Trinity College Dublin, or in any other University; and that the library may lend or copy it or any part thereof on request.

Signed:

_____

Date:

_____

Summary

This dissertation explores the ability of the neural Turing machine to learn an algorithm involving complex data structures. It does this by training an implementation of the neural Turing machine to solve the shortest path problem from graph theory. Its ability to perform this task is evaluated both in absolute terms, and relative to a baseline long short-term memory network. The problem's constraints are then relaxed and a more general view is sought on the model's ability to reason over graph structures.

A detailed review of the research underpinning the neural Turing machine is presented. This review, along with a formal definition and analysis of the shortest path problem form the foundations of the work detailed in this dissertation.

A method for generating a dataset over which to train the neural Turing machine and long short-term memory models is defined. This method uses knowledge of the problem space to efficiently generate a large dataset. Having generated the dataset, a method is described for the sequencing of its elements for input to the two models.

Two neural networks are implemented using TensorFlow to solve the shortest path problem: a neural Turing machine network and a long short-term memory network. These implementations are described in detail. The descriptions look in particular at the structures built that are specific to solving the shortest path problem.

An iterative method for training and optimising both networks is given. It involves three stages: plotting the networks' learning curves, hyper-parameter optimisation, and final testing. The learning curve determines the networks' performance versus training set size. The optimal training set size for each network is then used in hyper-parameter optimisation using random search over 60 iterations. The best-performing hyper-parameter configuration for each network is used to test the networks on unseen testing sets. Having been trained over a dataset containing only 6-node graphs, the networks are tested on datasets containing 6, 7, 8, 9 and 10-node

graphs.

The results presented in this dissertation show that neither the neural Turing machine, nor long short-term memory network learned a reliable solution to the shortest path problem. Analysis of their performance does indicate that the neural Turing machine achieved a statistically significantly lower error rate than the long short-term memory network. This is likely owing to its ability to use its addressable memory as a lookup table. However, the long short-term memory network generalised better, out-performing the neural Turing machine on sets containing larger graphs. Further analysis shows that neural Turing machine was able to reason about graph structures better than long short-term memory. Furthermore, the gap in performance between the two models on 6-node graphs is widened when a partial solution is considered.

The dissertation concludes with a discussion of improvements that could be made to the model. A weakness is identified in the description of the problem to the network and approaches to a solution are outlined, with reference to relevant research.

Abstract

Conor SEXTON

Advancing Neural Turing Machines: Learning a Solution to the Shortest Path Problem

The growth of the internet and advances in processing power have seen the dawn of a machine learning age. Concepts and models, new and old, are being combined and reworked to produce interesting and useful results. As we struggle to make sense of the vast amount of data that we now store, there is a pressing need to develop models that can make sense of it for us.

One such model is the neural Turing machine. Combining a neural network with an addressable memory bank, it emulates a human's ability to maintain a working memory. Initial research has shown that by using this memory, it can learn its own algorithms to solve problems, based solely on input and output examples.

This work measures the neural Turing machine's ability to learn a solution to a problem more complex than those it has seen before - the shortest path problem from graph theory. A large number of problems can be expressed using graphs, so showing that a neural Turing machine can reason over their structure shows that they can reason over a much larger set of problems.

By analysing the performance of the neural Turing machine against a long short-term memory network, this work shows that the memory has a positive effect on the model's ability to both solve the problem, and understand graph structures.

Acknowledgements

I would like to thank my mother, whose endless supply of muffins fuelled many a long day. I would also like to thank my father for always being ready to help, in whatever capacity required. Without their knowledge and support, this would not have been possible.

I would also like to extend my thanks to Dr. Rozenn Dahyot for her guidance throughout the year.

Contents

Declaration of Authorship **i**

Summary **ii**

Abstract **iv**

Acknowledgements **v**

List of Figures **xi**

List of Tables **xiii**

List of Abbreviations **xiv**

1 Introduction **1**

1.1 Motivation 1

1.2 Research Question 2

1.3 Aims 2

1.4 Dissertation Structure 3

2 Foundational Research **4**

2.1 Deterministic Turing Machines 4

2.2 Neural Networks 6

2.2.1 Overview 6

2.2.2 Gradient Descent 8

2.2.3 Hidden Layers 11

2.2.4 Training Techniques 13

Curriculum Learning 13

Regularisation 13

Gradient Clipping . . . 14
2.3 Types of Neural Network . . . 14
2.3.1 Recurrent Neural Networks and Long Short-Term Memory . . . 14
2.3.2 Neural Turing Machines . . . 18
2.3.3 Attention . . . 22
2.4 Shortest Path Problem . . . 24
2.4.1 Definition . . . 24
2.4.2 Applications . . . 25
2.4.3 Dijkstra's Algorithm . . . 27
2.4.4 Complexity . . . 27

3 Design & Implementation **29**
3.1 Tools . . . 29
3.1.1 TensorFlow . . . 29
3.1.2 NTM-TensorFlow . . . 30
3.1.3 Pickle . . . 31
3.2 The Dataset . . . 32
3.2.1 The Problem Space . . . 32
3.2.2 Graph Element Generation . . . 34
3.2.3 Dataset Generation . . . 35
3.2.4 Sequencing the Shortest Path Problem . . . 38
3.2.5 Generating Input . . . 40
3.3 Neural Turing Machine Model . . . 40
3.3.1 NTM Cell . . . 41
3.3.2 NTM Container . . . 43
Initial Connections . . . 43
Answer Phase . . . 44
Loss and Error Functions . . . 47
Optimiser . . . 50
Evaluation . . . 51
3.3.3 Shortest Path Task . . . 52
3.3.4 Padding . . . 53

3.4 Long Short-Term Memory Model 54

4 Evaluation **55**

4.1 Procedure 55

4.1.1 Learning Curve 55

4.1.2 Hyper-Parameter Optimisation 56

4.1.3 Final and Generalisation Testing 57

4.1.4 Comparing Results 58

4.2 Results 59

4.2.1 Learning Curve 59

4.2.2 Hyper-Parameter Optimisation 62

4.2.3 Final and Generalisation Testing 62

5 Discussion **64**

5.1 Analysis of Results 64

5.2 Model Improvements 70

6 Conclusion **72**

6.1 Objective Assessment 72

6.2 Final Words 73

A Production Environment **78**

B Hyper-Parameter Search Results **79**

List of Figures

2.1 The schematic view of a Turing machine. 5
2.2 Linear classification in two-dimensional feature space. 7
2.3 A neural network with input, hidden and output layers. The number of *hidden units* of a network refers to the number of nodes in its hidden layer. 7
2.4 Gradient descent with two weight parameters. 9
2.5 The effect of momentum on stochastic gradient descent (Ruder, 2016). 10
2.6 The effect of a $tanh$ layer on the representation of data (Olah, 2016). . . 12
2.7 How multiple hidden layers change data's representation to make them linearly classifiable (Olah, 2016). 12
2.8 A neural network (A) and recurrent neural network (B). 15
2.9 Unrolling a recurrent neural network through time. 16
2.10 An external view of an LSTM Network. 16
2.11 An internal view an LSTM Network. Each coloured area represents a different gate. 18
2.12 A schematic view of the neural Turing Machine (Graves, Wayne, and Danihelka, 2014). 19
2.13 The neural Turing machine unrolled through time (Olah and Carter, 2016). 20
2.14 A flow diagram of the addressing mechanism of the neural Turing machine (Graves, Wayne, and Danihelka, 2014). 22
2.15 An unweighted graph. 25
2.16 A subsection of a map of the London underground. 26
2.17 A relationship described as the shortest path in a family tree. 26

3.1 An example of a TensorFlow computation graph. The code on the right runs the computation graph and requests the value of *output1*, given an input of 10. 31
3.2 Two example elements from the dataset made up of a graph, start and end nodes and the shortest path between them. 33
3.3 The dataset generation process with paths lengths up to 2. A unique dataset element is first generated and then distributed to the correct bin. The first bin in the training set has become full and has been deleted, as indicated by the cross appearing over it. As a result, the first bin of the validation set has begun to be filled. The second bin of the validation set will not begin to be filled until the second bin of the training set has been filled. This process continues until all bins have been deleted. 37
3.4 A partial view of the NTM model. The green circles labelled *input* and *target* are placeholder tensors. The target placeholders are not yet connected to anything. 44
3.5 A network with its output highlighted on a graph. The start and end nodes of the path sought are illustrated by the double circles in the graph. 45
3.6 A partial view of the NTM computation graph, showing the structure of the connections in the answer phase. The *converter* node contains within it the softmax, argmax and one-hot encoding functions necessary to convert the network's output into an acceptable input. The + node prepends phase bits to the tensors. 46
3.7 The process of creating the output mask. The target tensor on the left is reduced along its second axis with a max function. Then, the result is tiled and reshaped to match the dimensions of the output. T represents the length of the input sequence. 48
3.8 The partial view of the NTM computation graph showing the sequence loss calculation. 49

4.1 On the left is the uniform distribution between the log of the range limits 0.001 and 1 after 100,000 samples. On the right is the distribution of the exponentiation. 58
4.2 The learning curves for the NTM (a) and LSTM (b) models. 60

5.1 The number of correct solutions per path length. 65
5.2 An example of an element in which the solution path is of length 1. . . 65
5.3 The training loss of the optimal hyper-parameter settings for the NTM (a) and LSTM (b). SGD loss has a lot of fluctuation between examples so the plot has been smoothed to highlight trends. 66
5.4 The average percentage of output path that was correct, per length of solution path. 66
5.5 Walks (a), (b) and (c) are all valid given the graph shown on the left. . 67
5.6 The percentage of paths returned by the NTM (a) and LSTM (b) that were valid. 67
5.7 The division of valid walks among the defined classes for the NTM (a) and LSTM (b) networks. 68
5.8 The distance of each valid walk output to the end node. 69
5.9 Shortest paths to within 1 edge of the specified end node. 70

List of Tables

3.1 Examples of tensors ordered by rank. 30
3.2 The number of unique connected, labelled graphs C_n and corresponding number of elements P_n per number of nodes n. 34

4.1 The performance of each model (%) on the validation set, having been trained on differently sized training sets. 60
4.2 The hyper-parameter settings used for determining the models' learning curves. The number of layers refers to the controller in the case of the NTM, and the network itself in the case of the LSTM. The memory size is only applicable to the NTM model. 60
4.3 The t-statistics and corresponding p-values obtain from performing a paired t-test between the results of training over 40,000 examples and every other training set size. 60
4.4 The edge error rate, per path length, for each training set size. In bold are the best results for each path length. 61
4.5 T-statistics and corresponding p-values obtained from comparing the LSTM network's edge error rate (for path lengths 2, 3 and 5) having been trained with 10,000 examples, with that of every other training set size. 61
4.6 Top 5 NTM hyper-parameter settings. 62
4.7 Top 5 NTM hyper-parameter settings. 62
4.8 The test set error rates (%). 63

A.1 The hardware specifications of the instances used for training and testing. 78

B.1 Neural Turing machine hyper-parameter search results. 80

B.2 Long short-term memory hyper-parameter search results. 81

List of Abbreviations

LSTM	**L**ong **S**hort-**T**erm **M**emory
NTM	**N**eural **T**uring **M**achine
RNN	**R**ecurrent **N**eural-**N**etwork
SGD	**S**tochastic **G**radient-**D**escent

Chapter 1

Introduction

In this chapter, the motivation behind this dissertation is introduced. The chapter then goes on to define the question posed by the research, and derives from this question the set of goals that were pursued. Lastly, the contents of the dissertation are outline by chapter.

1.1 Motivation

Machine learning has seen a recent surge in popularity. It has been available in commercial software products since the 1990s, but it is only with the recent growth in the size and availability of datasets that it has reached widespread utility (Goodfellow, Bengio, and Courville, 2016, p. 19-20).

Neural networks constitute one machine learning model that stands out from the rest. From their conception, they have striven to replicate the performance of the human brain. Recent research has seen them exceeding that goal in tasks at which, classically, computers could not compete, such as image recognition (He et al., 2015).

Modifications to their structure see them again draw inspiration from human cognition. The neural Turing machine (NTM) attempts to model the cognitive psychology concept of a "working memory" to allow it to approach problems in the way that a human would. It couples a neural network with an addressable memory bank that allows it to store information for processing at different points in computation. This has had the effect that, by providing the NTM with an input and desired output, it can learn its own basic algorithms for tasks such as copying, sorting, and associative recall.

The algorithms already learned by the NTM are fundamental to computer science, but basic. The aim of this dissertation is therefore to continue to explore the reasoning ability of the NTM by applying it to a new and more complex problem. To that end, it details the training of an NTM network to learn its own algorithm for solving the shortest path problem from graph theory.

Graph theory is important because it provides a framework on which to model a large set of concepts, including those in the areas of networking, linguistics, and computability theory. By showing that the NTM can learn to analyse and reason about a graph, one is also showing it to be able to reason over a much larger set of concepts.

1.2 Research Question

This dissertation questions the impact of an addressable memory source on the ability of neural networks to reason over complex structures such as graphs.

1.3 Aims

Derived from the research question, the aims of this dissertation are to:

1. Survey the research of neural Turing machines, neural networks and other related theory.
2. Define a method to procedurally generate a dataset containing graph and shortest path elements.
3. Build neural Turing machine and long short-term memory models to learn to solve the shortest path problem.
4. Define and follow a methodology for training and optimising these models.
5. Evaluate the models in terms of:
   (a) Their outright performance.
   (b) The statistical difference in the performance between the two.
   (c) Their ability to more generally reason over graph structures.

1.4 Dissertation Structure

This dissertation is divided into six chapters. This section gives an overview of the contents of each chapter.

Chapter 2 reviews the research on which this work is based. The deterministic Turing machine, the classical model from which the neural Turing machine draws its inspiration, is described in full. A portion of the chapter is devoted to proving a grounding in the core concepts powering neural networks, before analysing how these concepts are combined to form more complex models. Finally, a formal definition of the problem posed to the network is given. Its known solutions and application are listed before its complexity is given context with in machine learning.

Chapter 3 outlines design and implementation of the components necessary for the research. The space occupied by the dataset and the processes involved in its generation are defined. A description of the construction of the networks looks in detail at tools and structures used to better solve the problem.

Chapter 4 describes the evaluation of the networks. This chapter is broken into two sections: a description of the procedures followed in gathering results and a presentation of the results themselves.

Chapter 5 discusses the results obtained in the context of the research presented in Chapter 2. It then suggests changes that could be made that could improve the performance of the networks given the results.

Finally, Chapter 6 summarises the key findings of the research with reference to the aims described in Section 1.3 and presents some final words on the topic.

Chapter 2

Foundational Research

The chapter lays the foundations on which this dissertation is built. The chapter begins by looking at the conceptual ancestor of the neural Turing machine - the deterministic Turing machine. Section 2.2 is devoted to giving an overview of neural networks and the ideas underpinning them, before Section 2.3 explores more complex variations in their design. Section 2.4 concludes the chapter with a formal definition of the shortest path problem and analysis that contextualises the proposal of this dissertation.

2.1 Deterministic Turing Machines

A one-tape deterministic Turing machine is a mathematical model of computation often used in complexity theory to formalise the notion of an algorithm (Garey, 1979, p. 23).

A Turing machine consists of a finite-state machine *controller*, a *read-write head*, and a bidirectional, infinite *tape*. The tape can be thought of as a list that extends infinitely in both directions. Each discrete index, or *cell*, in the list can contain a *symbol*. A symbol is an element of some accepted set of symbols specified in the initial Turing machine definition. The read-write head allows the controller to read and write symbols from and to cells on the tape. A high-level view of the architecture of the Turing machine can be seen in Fig. 2.1.

A Turing machine M is formally defined (Hopcroft and Ullman, 1979) by:

$$M = \langle Q, \Gamma, b, \Sigma, \delta, q_0, F \rangle \text{ where} \tag{2.1}$$

- Q is the finite, non-empty set of states.
- Γ is the finite set of allowable tape symbols.
- $b \in \Gamma$ is the *blank symbol*.
- $\Sigma \subseteq \Gamma \setminus \{b\}$ is the set of input symbols.
- $\delta : (Q \setminus F) \times \Gamma \to Q \times \Gamma \times \{L, R\}$ is the transition function, where L is left shift, R is right shift. If δ is not defined on the current state and the current tape symbol, then the machine halts.
- $q_0 \in Q$ is the start state.
- $F \subseteq Q$ is the set accepting states.

A Turing machine takes as input a string $x \in \Sigma^*$, which it then writes to the tape. At each subsequent time step, the Turing machine's controller performs actions and a state transition based on the transition function. The Turing machine's actions will be to either erase or overwrite the current symbol pointed to by the head, and then move the head one cell to either the left or right along the tape. Execution concludes when the controller has reached an accepting state.

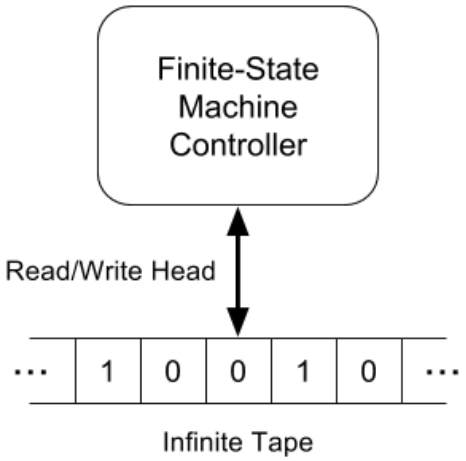

FIGURE 2.1: The schematic view of a Turing machine.

Turing's idea was to affix memory to a finite state machine. This difference is a subtle one, but it makes Turing machines a much more powerful model. For any computation that can be carried out by mechanical means, a Turing machine can be constructed that is capable of simulating that computation (Turing, 1937).

2.2 Neural Networks

This section introduces the concept of a neural networks. It begins with a basic explanation of their operation followed by a look in greater detail at the concepts underpinning them.

2.2.1 Overview

A neural network is a computational model that attempts to approximate a function $f(x) = y$ based on input and output examples. Many types of task can be solved using neural networks, but two of the most common are *classification* and *regression*. If the output y is of a set of discrete classes, problem is one if classification. An example of this type of problem is identifying objects in images (Krizhevsky, Sutskever, and Hinton, 2012). Regression aims to predict some continuous value based on its input. A classic example is predicting house prices based on location, size and other features. What follows is a description of neural networks in term of a classification problem, although the two only differ in the way that the output is interpreted.

In its simplest form, a neural network is a linear classifier with an input and output layer. It accepts as input *features* of a data element, and outputs a classification of that data element. It does this by trying to define a straight line in feature space that divides elements by class (as illustrated with two classes in two dimensions in Fig. 2.2). This line is defined by:

$$y = Wx + b. \tag{2.2}$$

A *loss function* is a function that measures how wrong the network's output was and therefore how well the line divides the classes of the set. When the loss over the entire set is high, it means the classes are poorly divided. When the loss is low, it means the classes are well divided. *Gradient descent* is a training process by which the model's *parameters* W (weight) and b (bias) are incrementally changed to minimize this loss function.

Linear classification works well until what is needed to be classified is not linearly separable in its current representation. It is at this point that a *hidden layer* is

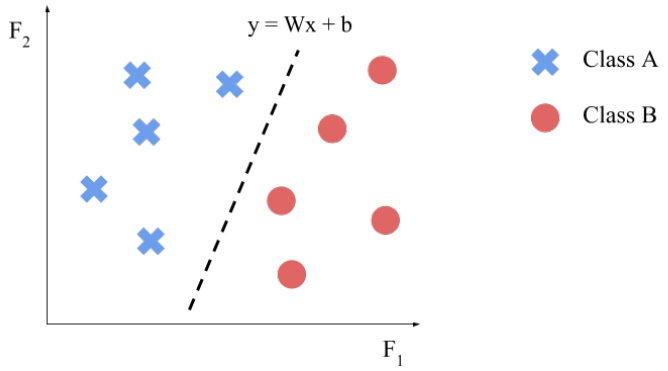

FIGURE 2.2: Linear classification in two-dimensional feature space.

added. A hidden layer sits between the input and output layers and applies some function to the representation of the data to make it linearly classifiable. The extent to which this function is applied, and to what features, is learnable using gradient descent. A neural network containing a hidden layer is illustrated in Figure 2.3.

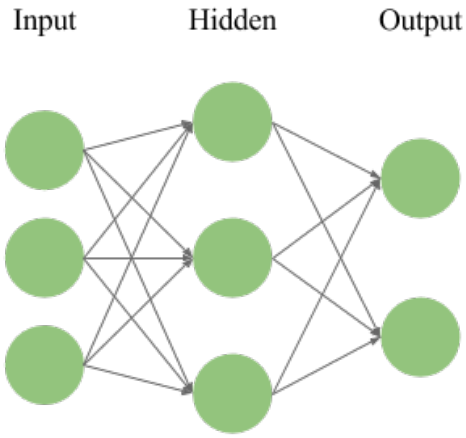

FIGURE 2.3: A neural network with input, hidden and output layers. The number of *hidden units* of a network refers to the number of nodes in its hidden layer.

Training neural networks is difficult and can be mathematically unstable. To aid in this task, a number of training techniques and algorithms have been introduced. The values that vary how and where these techniques and algorithms are applied are referred to as the model's *hyper-parameters*.

Neural networks constitute a machine learning model that derive power from the way that they can be configured and tuned using different hidden layers and functions to suit the task at hand. The coming sections examine how neural networks learn using these configurations.

2.2.2 Gradient Descent

The objective of training a neural network is to minimise some loss function. Gradient descent is an approach to optimisation that makes use of the function's derivatives. This section describes the idea behind gradient descent, and how it is modified to perform well on larger models and datasets.

To gain some intuition on this method, imagine a blind hillwalker. His objective is to reach the top of a hill but he cannot see where to go. When the slope of the hill is large, he knows to take long steps because he is still far from the top. As he approaches the top however, he takes smaller steps to avoid over-shooting the peak. In this way, the slope of the hill aids in finding the peak.

Suppose we have some function $y = f(x)$, where both x and y are real numbers. The derivative of this function, which gives the slope of $f(x)$ at point x, is denoted by $f'(x) = \frac{\delta y}{\delta x}$. When trying to minimise or maximise the value of $f(x)$ by changing the value of x, the derivative $f'(x)$ can be used to *scale* the changes made: $f(x+\alpha) \approx f(x) + \alpha f'(x)$, where α is the size of the unscaled change to be made (Goodfellow, Bengio, and Courville, 2016).

When training a neural network, the function being minimised is the loss function, expressed as a function of the model's parameters. The α term is referred to as the *learning rate*, and controls how much the parameters will be changed. In the hillwalker example, he was trying to reach the peak of the hill - or to maximise his height. Neural networks aim to minimise some loss function, so the negative slope is used as a scaling factor.

Batch gradient descent computes the derivative of the loss function $\mathcal{L}$ with respect to the model's parameters θ across the entire training set. The parameters are then updated according to the function:

$$\theta = \theta - \alpha \nabla_\theta \mathcal{L}(\theta) \tag{2.3}$$

This is an iterative process; the derivative is repeatedly found and applied to the parameters to minimise the loss - thus descending the gradient.

An example of this with just two weights, w_1 and w_2, and loss as a function of these weights $\mathcal{L}(w_1, w_2)$ is illustrated in Figure 2.4. The loss starts high, but each step

of updating the weights with the derivative of the loss with respect to the weights brings the loss closer to the minimum.

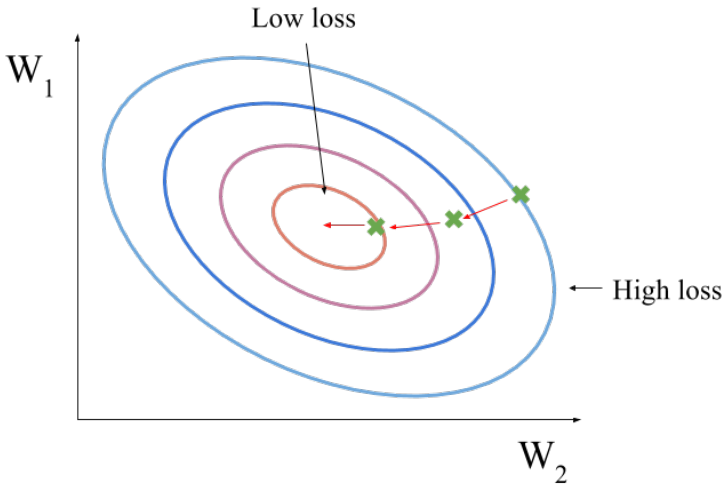

FIGURE 2.4: Gradient descent with two weight parameters.

A problem with batch gradient descent is that it does not scale well. It requires the loss to be computed over the entire training set for each update. As the number of training examples and model parameters increases, this becomes infeasibly costly. Furthermore, this must be done repeatedly to find a minimal loss.

Because of this, *stochastic gradient descent* (SGD) was developed. Stochastic gradient updates the model's parameters using a single random sample of the training set x_i and its target y_i:

$$\theta = \theta - \alpha \nabla_\theta \mathcal{L}(\theta; x_i; y_i) \tag{2.4}$$

A single pass of SGD over all of the training examples is called an *epoch*.

The derivative of the loss for an individual example is not likely to be representative of entire training set, meaning that the updates might not step the loss in the right direction. This is compensated for by performing many more, smaller updates. These updates are much cheaper to compute than the updates over the entire dataset, so more of them can be performed and, over time, they will tend towards the minimum. Neural networks optimised with SGD have been shown to perform well on large-scale problems (Bottou, 2010), but algorithm requires a number of modifications to make it more effective.

SGD performs badly when the optimum sits at the end of a long *ravine* (Sutton, 1986). A ravine is an area where the gradient is much steeper in one dimension than another. This is because the gradient at any point along either of the walls of the ravine will point across to the other wall, causing the loss to oscillate between the two.

Momentum (Qian, 1999) is one method for pushing the update along the ravine towards the minimum to prevent oscillation. Its effects are illustrated in Figure 2.5. An analogy for how it works is to imagine pushing a ball down a hill (Ruder, 2016). As the ball travels down the hill it gathers momentum until it reaches some terminal velocity. This is reflected in the parameter updates. A new momentum term γ is introduced that increases for dimensions whose gradients point in the same direction and reduces for updates whose gradients change directions. An *update vector* v keeps track of this momentum, and the update term at time t becomes:

$$v_t = \gamma v_{t-1} + \alpha \nabla_\theta \mathcal{L}(\theta) \tag{2.5}$$

$$\theta = \theta - v_t \tag{2.6}$$

(a) SGD without momentum (b) SGD with momentum

FIGURE 2.5: The effect of momentum on stochastic gradient descent (Ruder, 2016).

Some parameters will have more of an effect on the loss than others. To reflect this, many SGD-based optimisers use *adaptive learning rates*. Adaptive learning rate optimisers use different learning rates for each model parameter, slowly adjusting the rate using the consistency of the parameter's gradient. For example, RMSProp (Hinton, 2014) divides the learning rate of each parameter by an exponentially decaying average of its squared gradient. For gradient g, learning rate α and smoothing

term ϵ, the running average $E[g^2]_t$ and resulting update function are defined as:

$$E[g^2]_t = \beta E[g^2]_{t-1} + (1-\beta)g_t^2 \tag{2.7}$$

$$\theta_{t+1} = \theta_t - \frac{\alpha}{\sqrt{E[g^2]_t + \epsilon}} g_t \tag{2.8}$$

The β term above is the *weight decay*, another model hyper-parameter that can be tuned.

Gradient descent is a method for training neural networks by minimising a loss value expressed as a function of the model's parameters. Batch gradient descent does not scale well with size of dataset or model. Instead stochastic gradient descent is done to provide a cheap approximation. In an attempt to make this approximation more accurate, momentum and adaptive learning rates are used.

2.2.3 Hidden Layers

Hidden layers are used in neural networks to transform the representation of the data into one that is linearly classifiable. Different functions introduce non-linearity to achieve this. Hidden layers can then be combined to produce networks suited to the problem.

Hidden layers apply some function to the input from the previous layer to produce an output. A common approach is to first apply a linear transformation to the incoming data, followed by some non-linear squashing function. An example of a common hidden layer is a $tanh$ layer $tanh(Wx + b)$. This layer consists of:

1. A linear transformation by the weight matrix W.
2. A translation by the bias vector b
3. A point-wise application of $tanh$.

The effects of this layer on the data's representation can be seen in Figure 2.6.

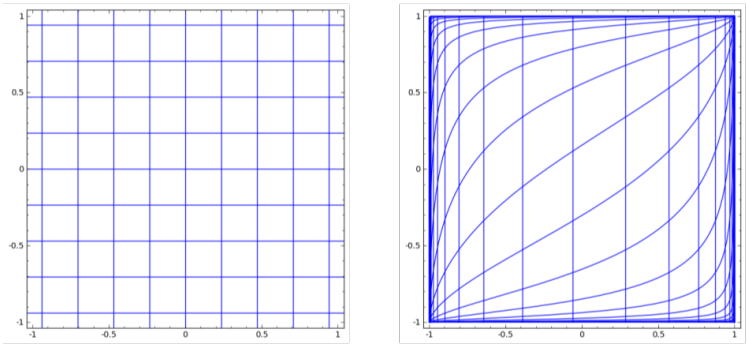

FIGURE 2.6: The effect of a $tanh$ layer on the representation of data (Olah, 2016).

Each layer has its own set of parameters that can tuned using gradient descent. The $tanh$ layer is one example of many different types of layer, which can be combined in different ways. It then becomes network's job to learn to find the parameters for these layers that produce a representation in which the data are linearly separable. The number of layers a network has is referred to as its *depth*.

In Fig. 2.7 it can be seen on the left that a straight line is not sufficient to separate two classes (represented by two curved lines) that are closely mixed. But by changing the *representation* of the data using a combination of hidden layers, the classes become easily separable using linear classification.

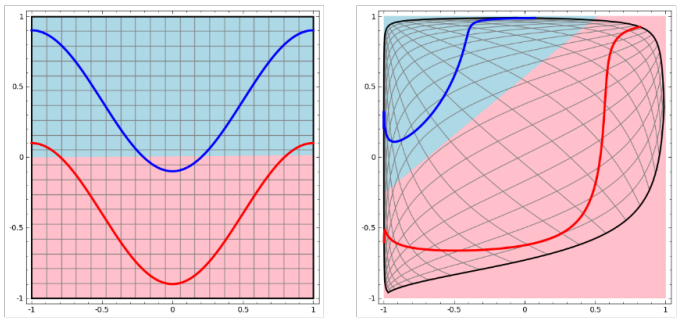

FIGURE 2.7: How multiple hidden layers change data's representation to make them linearly classifiable (Olah, 2016).

Combining layers in this way to produce deep networks has proven to be a powerful method. Networks constructed in this way have dramatically improved the state-of-the-art in speech recognition, visual object recognition and many other domains (Lecun, Bengio, and Hinton, 2015).

2.2.4 Training Techniques

As the depth of a network increases, so too does the difficulty of training it (Srivastava, Greff, and Schmidhuber, 2015). This section describes some of techniques and algorithms used to help a network to learn.

Curriculum Learning

Humans and animals learn much better when examples are presented to them in some meaningful order, rather than randomly. Using *curriculum learning* (Bengio et al., 2009)., training data for a task is structured as if being taught to a human. It breaks the data into discrete *lessons* that gradually introduce more complex concepts to the network. This has been shown to both increase the speed of convergence in training, and also find better local minima in non-convex loss space.

Regularisation

Regularisation is the concept of applying artificial constraints on the network to prevent what is known as *overfitting*. Overfitting occurs when the approximate function found by the network is excessively complex. This is signalled by lowering of the loss seen in training, with no corresponding reduction in the error rate in validation.

One common form of regularisation is known as ℓ_2 regularisation. ℓ_2 regularisation encourages the parameters to not take large values by updating the loss value with the ℓ_2-norm of the model's weight parameters. The new loss $\mathcal{L}'$ is defined as:

$$\mathcal{L}' = \mathcal{L} + \beta \frac{1}{2} \|\theta\|_2^2, \tag{2.9}$$

where

$$\|\theta\|_2^2 = \sum_{i=1}^{k} \theta_i^2 \tag{2.10}$$

W represents the model's parameters, of which there are k. The β is a tunable hyperparameters governing how much regularisation is applied.

Gradient Clipping

Gradient clipping is a technique used to prevent the problems of *vanishing* and *exploding* gradients.

These problems are apparent in deep and recurrent (discussed in section 2.3) networks when the model's gradients are repeatedly multiplied by numbers less-than or more-than 1. There is a limit to the accuracy of floating point numbers in computers, so if the gradient becomes too small it will "vanish" - or become zero. Similarly, if the gradients become too large, they can become infinite.

Without meaningful gradients, the model cannot learn. To prevent this, the gradients are clipped to maximum and minimum values, ensuring that they never get too large or too small

2.3 Types of Neural Network

The previous section discussed the mechanics of standard neural networks. This section introduces three variations on the original design: recurrent neural networls, long short-term memory, and the neural Turing machine. Their core structure is still that of a neural network, but they have been engineered to be better solvers of specific problems. Also discussed is a trend in neural network design, and the significance it has in the field as a whole.

2.3.1 Recurrent Neural Networks and Long Short-Term Memory

Traditional neural networks have proven to have good performance for many practical classification problems (Zhang, 2000), but they have limitations. This section discusses these limitations and describes the models created to remove them.

Traditional neural networks are only capable of classifying a fixed-length input, which makes them very rigid. If the data are sequential, we must concatenate them into a single vector for them to be accepted as input by the network. This concatenation is wasteful, as there may be statistical invariants through time that will be lost if the sequence is treated as a whole.

For example, if training a traditional neural network to count the number of 1s in a sequence of bits, the network can only work on a sequence with a fixed length.

Here, identifying if a bit is 1 or 0 is a statistical invariant between each length sequence - that is, it does not change with the length of sequence. But this insight is being lost as the network must be trained for each length of bit sequence independently. A network that has trained to count bits in a sequence of length 5 will not work on a sequence of length 30.

The more that is known about the data, the better the classifier can be made. If the data is known to be sequential, the classifier should capture this. It is with this in mind that *recurrent neural networks* (RNNs) were developed. The addition of a feedback loop from the network to itself makes RNNs particularly suited to handle sequences of input. In Fig. 2.8 we can see the difference between traditional and recurrent neural networks.

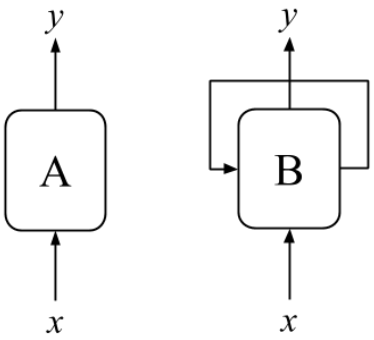

FIGURE 2.8: A neural network (A) and recurrent neural network (B).

By unrolling the RNN through time, we can see more clearly how it is the same as a sequence of connected networks performing the same task on a sequence of inputs while taking into account what has come before (see Fig. 2.9). This feedback loop gives RNNs greater power in the solving of problems, even making then Turing-complete (Siegelmann and Sontag, 1995).

In the bit-counting example, it allows the network to learn to identify if a single bit is 0 or 1. A sequence of arbitrary length can be fed to the network and it can reuse this ability for each successive bit. Then, it can use its recursive connection to maintain a count of bits for the entire sequence, outputting when necessary.

In more complex sequences, dependencies can exist. For example, in an English sentence the use of the pronouns "he" or "she" will depend on the person to whom the speaker is referring.

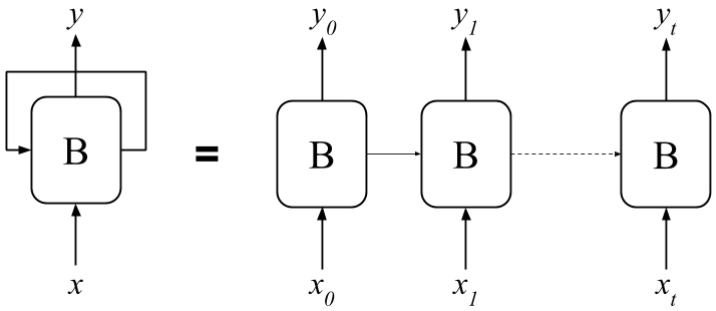

FIGURE 2.9: Unrolling a recurrent neural network through time.

A problem with RNNs is that they are bad at dealing with long-term dependencies (Bengio, Simard, and Frasconi, 1994). As dependencies get further away from each other in the input sequence, it becomes less likely that the network will account for them.

This tendency led to the development of *long short-term memory* (LSTM) (Hochreiter and Schmidhuber, 1997). An external view of LSTM can be seen in Figure 2.10. LSTM is similar to RNNs in that information from the previous input is fed back into the network. The difference, however, is that LSTM allows this information to be selectively modified based on the new input before being passed on to the next input. This modification happens to a varying degree. Furthermore, the degree to which this information, known as *cell state*, is modified is itself differentiable (able to be mathematically differentiated), allowing it to be learned using gradient descent. This mechanism allows LSTM to learn to hold information for longer when long-term dependencies do occur.

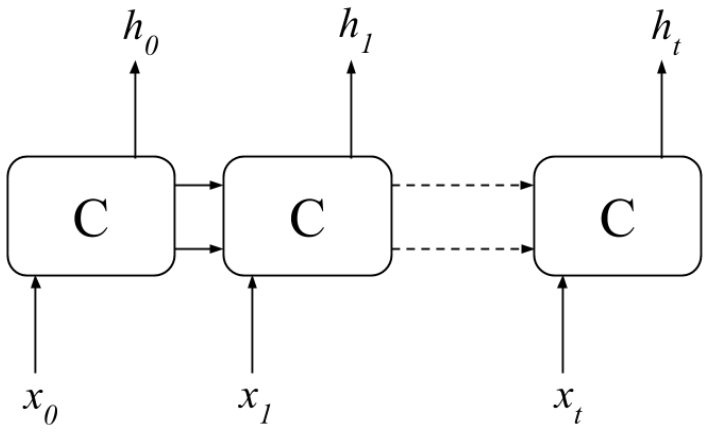

FIGURE 2.10: An external view of an LSTM Network.

Internally, LSTM contains three *gates*. Gates are ways to filter information, and are the mechanism by which the cell state is modified. The three gates are:

1. *Forget gate* - this is used to determine what parts of the cell state will be forgotten. It is implemented using a sigmoid layer and looks at the previous output and the current input to produce a number between 1 and 0 for each number in the cell state. A 1 indicates a number should completely retained, a 0 indicates a number should be completely forgotten. The resulting vector is then multiplied into the cell state. For previous output h_{t-1} and current input x_t, the output of the forget gate is described by:

$$f_t = \sigma(W_f[h_{t-1}, x_t] + b_f) \tag{2.11}$$

2. *Input Gate* - this gate decides what information will be stored in the cell state. In is made up of two parts: a sigmoid layer and a tanh layer. The sigmoid layer decides what values will be added to the cell state, the tanh layer prepares the input for storing in the cell, and the output of the two are multiplied together to produce the update vector which is itself added to the cell state. The output of the sigmoid layer i_t, and the output of the tanh layer $\bar{C}_t$ and update vector U_t are described by:

$$i_t = \sigma(W_i[h_{t-1}, x_t] + b_i) \tag{2.12}$$

$$\bar{C}_t = tanh(W_C[h_{t-1}, x_t] + b_C) \tag{2.13}$$

$$U_t = i_t \times \bar{C}_t \tag{2.14}$$

3. *Output Gate* - this gate decides what is output by the network. This is again made up of two parts: a sigmoid layer and a tanh layer. In this case the sigmoid layer decides what part of the cell state to output, and the tanh layer prepares the cell state for output. With cell state C_t, the output of the sigmoid layer o_t, and output of the network h_t are described by:

$$o_t = \sigma(W_o[h_{t-1}, x_t] + b_o) \tag{2.15}$$

$$h_t = o_t \times tanh(C_t) \tag{2.16}$$

The weight parameters W_f, W_i, W_C, W_o and bias parameters b_f, b_i, b_C, b_o can all be trained, meaning LSTM can learn how to manage its cell state. These gates are illustrated in Figure 2.11.

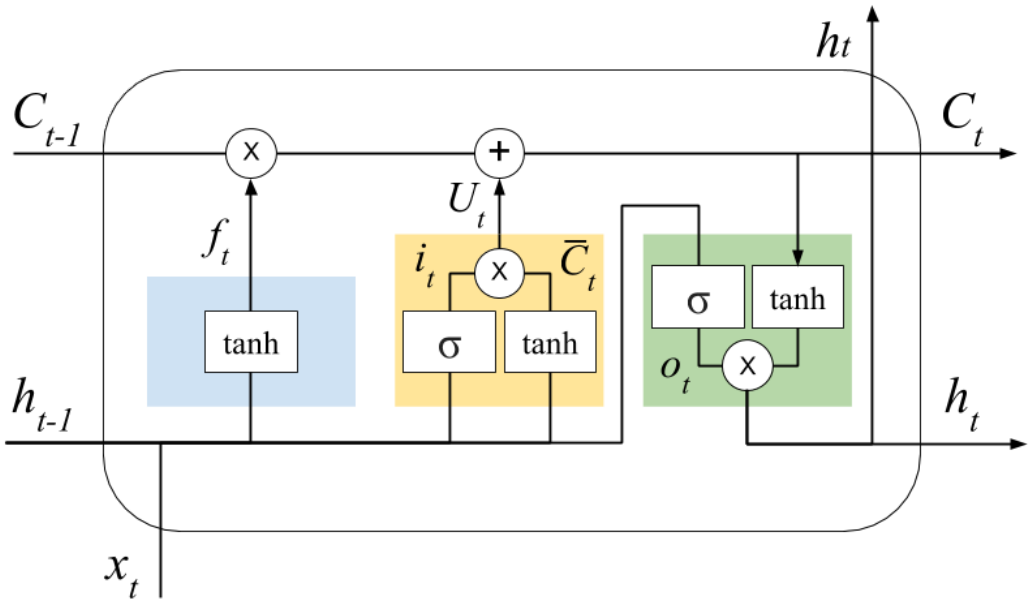

FIGURE 2.11: An internal view an LSTM Network. Each coloured area represents a different gate.

Neural networks are a powerful machine learning model. By adding new connections and layers, such as those present in RNNs and LSTM, they become adept at handling sequential, variable-length input.

2.3.2 Neural Turing Machines

This chapter has already discussed how enriching deterministic finite state machines with a memory tape increases their power, and how augmentations to neural networks have allowed them to learn to solve problems involving sequences of variable length. The neural Turing machine (NTM) (Graves, Wayne, and Danihelka, 2014) combines these augmentations to push even further the capabilities of machine learning. This section discusses the theory behind the NTM, gives an overview of its architecture and fully defines its operation.

"Working memory" is a concept in human cognition that describes holding information in the mind and mentally manipulating it, such as when working out a

mathematical problem or interpreting language. It is critical to our ability to see connections between seemingly unrelated things and to pull apart elements from an integrated whole - and hence to creativity because creativity involves disassembling and recombining elements in new ways (Diamond, 2013). It is often measured by the number of "chunks" of information in a single dimension that can be readily recalled (Miller, 1965). The NTM aims to give neural networks a working memory - some space separate to the core computation where they can manipulate and reorder information to solve problems.

The NTM is made up of two basic components: an LSTM *controller* and a rewritable, addressable *memory bank*. The controller interacts with the outside world via input and output vectors, as a standard neural network would, and interacts with its memory bank using read and write *heads*, similar to the deterministic Turing machine, described in Section 2.1. A high-level view of the NTM's architecture is illustrated in Fig. 2.12.

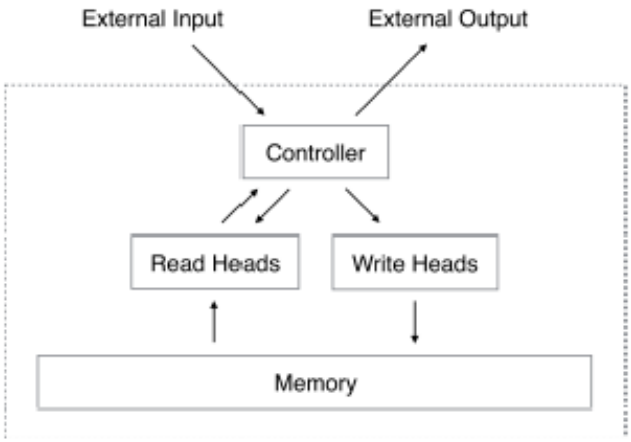

FIGURE 2.12: A schematic view of the neural Turing Machine (Graves, Wayne, and Danihelka, 2014).

For each input, the NTM considers the controller's cell state and the contents of memory. For each output, the NTM updates the cell state and write to memory. This is illustrated in Fig. 2.13 with a controller network C accessing a memory bank M. What is important is that every component is differentiable - meaning that every operation is learnable through gradient descent. Reading, writing and specifying a location in memory are all learnable by the controller.

The memory bank M_t is a matrix with N vectors of length M at time t. The sizes

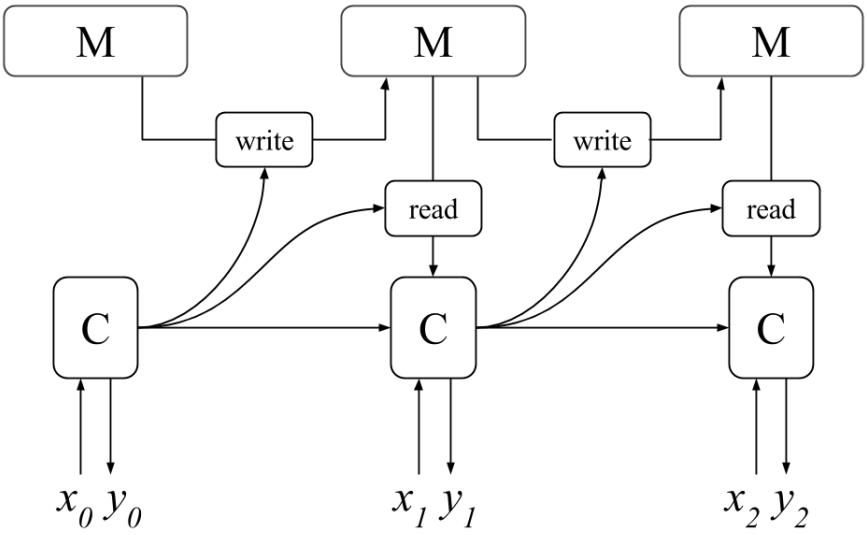

FIGURE 2.13: The neural Turing machine unrolled through time (Olah and Carter, 2016).

of N and M are fixed and defined during creation. When reading from M_t instead of reading from one single location, a normalised weight vector w_t emitted by the read head specifies how much we care about each location in memory:

$$r_t \leftarrow \sum_i w_t(i) M_t(i), \tag{2.17}$$

where r_t is the read vector obtained from memory. As such it can specify where to read from and by how much, and is differentiable with respect to both the weight and the memory bank.

Similarly, when writing to memory, the NTM writes everywhere at once, just to different extents - controlled by the weight vector. The process is split into two parts: *erasure* and *addition*. Initially an erasure vector e_t of length M is emitted by the read head and applied to the memory bank:

$$\bar{M}_t \leftarrow M_{t-1}(i)[1 - w_t(i)e_t]. \tag{2.18}$$

Next, the addition vector a_t also of length M is added to our erased memory bank:

$$M_t \leftarrow \bar{M}_t(i) + w_t(i)a_t. \tag{2.19}$$

Notice that, in both cases, where the operation's attention is being focused is controlled by a weight vector being emitted by each head. This weight vector is defined by the NTM's *addressing mechanism*. There are two ways by which an NTM can address its memory:

1. *Content-based addressing*: The head compares (using cosine similarity) each vector in memory, $M_t(i)$, to a length M key vector k_t emitted by the controller. The weight vector w_t^c is the normalised result of this comparison. A *key strength* value β_t is used to increase or decrease the precision of focus on the memory location:

$$w_t^c \leftarrow \frac{\exp(\beta_t K[k_t, M_t(i)])}{\sum_j \exp(\beta_t K[k_t, M_t(j)])}, \tag{2.20}$$

   where

$$K[u, v] = \frac{u \cdot v}{\| u \| \cdot \| v \|}. \tag{2.21}$$

2. *Location-based addressing*: This addressing mechanism is designed to allow iteration through memory locations, or random jumps. Location-based addressing can be used in two ways: it can specify an offset from the last memory address accessed, or it can specify that offset relative to a content-based address. It does this using an *interpolation gate* g_t in the range (0, 1):

$$w_t^g \leftarrow g_t w_t^c + (1 - g_t) w_{t-1}. \tag{2.22}$$

   Whichever one is selected is then used in the equation to move the head. The *shift weighting* s_t is a normalised vector emitted by the head that defines the degree to which an offset in either direction is performed:

$$\bar{w}_t(i) \leftarrow \sum_{j=0}^{N-1} w_t^g(j) s_t(i - j). \tag{2.23}$$

   The result from above is then sharpened using the scalar $\gamma_t \geq 1$ to prevent dispersion of weightings over time:

$$w_t(i) \leftarrow \frac{\bar{w}_t(i)^{\gamma_t}}{\sum_j \bar{w}_t(j)^{\gamma_t}} \tag{2.24}$$

A complete view of the memory addressing used by the NTM is illustrated in Fig. 2.14.

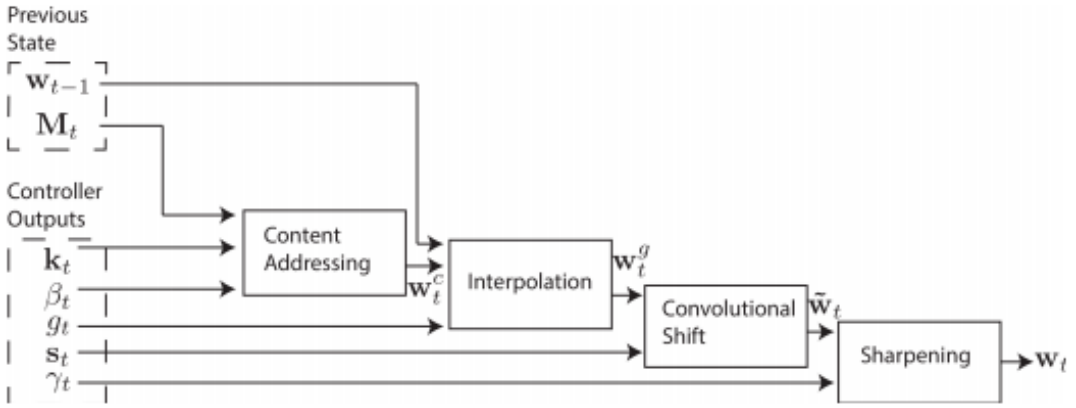

FIGURE 2.14: A flow diagram of the addressing mechanism of the neural Turing machine (Graves, Wayne, and Danihelka, 2014).

By combining concepts from automata theory, machine learning, and psychology, researchers have created a new model for machine learning. Using the read, write and addressing mechanisms described above, the NTM can learn how to store, re-read and manipulate input and intermediate data throughout the course of its computation using gradient descent. These new abilities were shown by the original paper (Graves, Wayne, and Danihelka, 2014) to allow the NTM to infer basic algorithms more effectively than LSTM.

2.3.3 Attention

As discussed in this chapter, augmentations to neural networks aim to reduce their limitations and increase their problem-solving power. These augmentations come in several different forms, of which the NTM is one example. A new group of neural networks is emerging that relies on the same underlying concept - that of *attention*. Attention-based models have shown near-human levels of ability on certain tasks using mechanisms inspired by human cognition.

When translating a sentence, a person will focus more on the word that they are reading at any one point in time than any other in the text. When describing their surroundings in a room, a person will look at each object that they list as they speak. When copying text, a person will remember several words at a time and copy them

sequentially from memory. Neural networks can behave in the same way using attention. Attention allows them to focus in different amounts on different points in input or memory to aid in completing a task. Attention is also differentiable, allowing the neural network to learn where it needs to focus.

This chapter has discussed attention implemented as memory addressing in the NTM - being able to read and write vectors in different amounts to different points in a memory bank. Other neural network augmentations exist that implement attention also. For example, Attentional Interfaces allow RNNs to focus on different sections of their input, Adaptive Computation Time (Graves, 2016) allows computation to performed to varying degrees at each step, and Neural Programmers (Neelakantan, Le, and Sutskever, 2015) build programs by learning where to call functions. These models are at the cutting edge of machine learning research.

These new, attention-based models have produced accurate solutions to problems at which humans have typically bested computers. These include problems such as automatically describing images (Xu et al., 2015) and language translation (Bahdanau, Cho, and Bengio, 2014). Furthermore, as discussed, the models have allowed computers to infer their own algorithms for solving problems (Graves, Wayne, and Danihelka, 2014).

Very often, the most effective forms of intelligence involve the interaction between some exact medium and the heuristic intuition of humans. For example, a human will use a pen and paper to solve a mathematical problem. Part of the computation is offloaded to the equations on the page, and the human uses their intuition to reason and decide what next step to take.

Classically, computers have been one such exact medium used by humans. Attention and neural networks are exciting because they allow computers to use exact media the way a human would - they model human intuition in a way executable by computers. They allow computers to have fuzzy interaction with exact media, where fractions of actions can be taken simultaneously and combined to varying extents.

The emergence of attention-based models has allowed machines to learn to solve problems with accuracy previously only possible by humans, in a way that emulates humans.

2.4 Shortest Path Problem

Among the aims of this dissertation is to train neural networks to solve the shortest path problem. This section first formally defines the problem and gives examples of its application. Following this, a known solution is analysed and the problem's complexity is discussed within the context of machine learning.

2.4.1 Definition

In graph theory, the shortest path problem is the problem of finding a path between two vertices in a graph such that the sum of the weights of the constituent edges of the path is minimized. A formal definition of the problem with given graph G is:

$$G = \langle V, E \rangle \tag{2.25}$$

$$P = \langle v_1, v_2, ..., v_n \rangle \in V \times V \times ... \times V \tag{2.26}$$

$$w(P) = \sum_{i=1}^{k} w(v_i, v_{i+1}) \tag{2.27}$$

$$\phi(u, v) = min \{w(P) : P \text{ is path from } u \text{ to } v\} \tag{2.28}$$

- V is the set of vertices.
- E is the set of edges.
- $P \subset V$ such that $(v_i, v_{i+1}) \in E, \forall v \in P$
- $w(P)$ is the weight of a path (note that in an unweighted graph we treat the weight of each edge as 1).
- ϕ is the shortest path between two vertices.

For example, applying the above formulae to the graph illustrated in Figure 2.15 gives:

$$\phi(D, F) = P \tag{2.29}$$

$$P = \{D, B, E, F\}. \tag{2.30}$$

$$w(P) = 3 \tag{2.31}$$

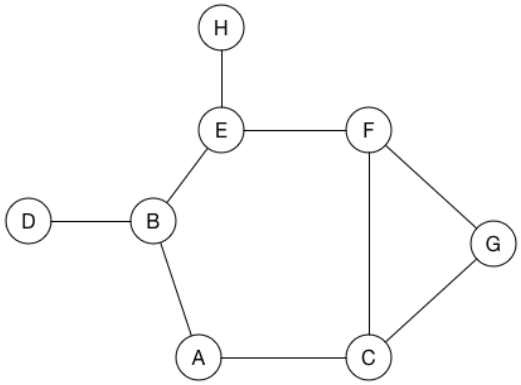

FIGURE 2.15: An unweighted graph.

The networks used in this dissertation are trained to produce a shortest path, as defined above, between two vertices of an unweighted, undirected, connected graph.

2.4.2 Applications

A large number of problems can be modelled using graph theory. This section illustrates some examples of problems being represented as graphs, and the meaning the shortest path has in these examples.

A common example of a graph structure is seen in subway maps. Illustrated in Figure 2.16 is a subsection of the London underground. In this illustration, we can consider the stations to be vertices of a graph, and the train lines to be edges. Finding the shortest route between two subways stations then becomes an instance of the shortest path problem.

Non-deterministic abstract machines can be modelled as graphs in which vertices represent states and edges represent transitions between those states. Finding the sequence of transitions that lead to some goal state for the machine is thus another instance of the shortest path problem.

FIGURE 2.16: A subsection of a map of the London underground.

An example of this the Rubik's cube. It can be modelled as a graph by making each vertex a different configuration of its faces. Each edge in this case represents a single turn of one of the cube's faces that connects two configurations. The process of solving the Rubik's cube is another instance of the shortest path problem. The start node is the initial jumbled configuration, the end node is the configuration with one colour on each face, and the solution is the shortest path between the two.

Relationships between entities can also be modelled using graphs. An example of this is a family tree. The relationship between any two people in the tree is described by the shortest path between them. This is illustrated in Figure 2.17.

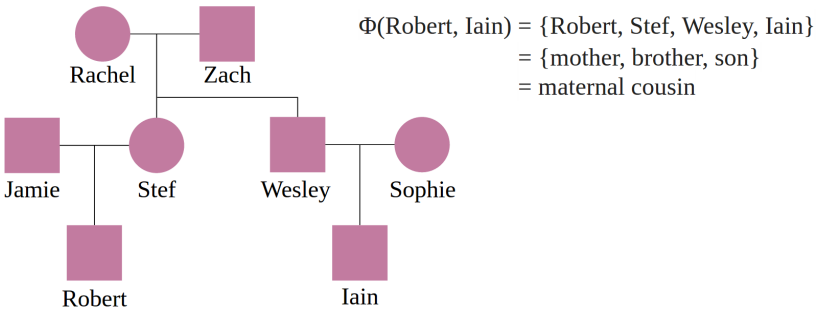

FIGURE 2.17: A relationship described as the shortest path in a family tree.

The foregoing are just a few examples of how problems can be modelled as graphs.

2.4.3 Dijkstra's Algorithm

The shortest path problem is an extensively-studied problem. There are a number of known solutions, depending on the type of graph, but they can broken down into two categories: single source and all-pairs. Single-source algorithms find the shortest path between a single vertex in the graph and all other vertices. All-pairs algorithms find the shortest path between every pair of vertices in a graph. This section describes Dijkstra's algorithm - a solution to the single-source shortest path problem - and analyses its complexity.

Dijkstra's algorithm (Dijkstra, 1959) works by gradually exploring vertices outwards from the source, prioritising those that are closer. The pseudocode for Dijkstra's algorithm is given in Algorithm 1 below.

Algorithm 1 Dijkstra's algorithm for the single-source shortest path problem.

```
1:  function DIJKSTRA(Graph, source)
2:      for vertex v in Graph do                ▷ Initialise all distances to infinity
3:          dist[v] = ∞
4:          previous[v] = undefined
5:      dist[source] = 0
6:      Q = the set of all vertices in the Graph
7:      while Q is not empty do                 ▷ Main loop
8:          u = vertex in Q with smallest dist
9:          remove u from Q
10:         for each neightbour v of u do       ▷ Where v is still in Q
11:             alt = dist[u] + dist_between(u, v)
12:             if alt < dist[v] then
13:                 dist[v] = alt
14:                 previous[v] = u
        return previous
```

In an undirected, weighted graph $G = \langle V, E \rangle$, the time complexity of Dijkstra's algorithm is $\mathcal{O}(V^2)$. This is not optimal, as this problem has been shown to be solvable in linear time (Thorup, 1999). However, in the special case of unweighted graphs, Dijkstra's algorithm adopts the complexity of breadth-first search, solving the problem optimally in linear time $\mathcal{O}(V + E)$.

2.4.4 Complexity

The shortest path problem may appear to be a trivial one given that there are many algorithms already developed to solve it. However, it is important to make the

distinction between providing an algorithm to find a solution to the shortest path problem, and teaching a machine to infer its own algorithm based on examples. The shortest path problem is of a higher order of complexity than any other problem solved by the NTM in its original presentation (Graves, Wayne, and Danihelka, 2014), based on structural complexity.

The choice of structural complexity may seem arbitrary, but that is not the case. The core aim of the NTM is to be able to learn to store and maintain data structures over the course of the computation. Therefore the complexity of the data structures that the NTM is capable of storing is indicative of the success of this aim. The speed and space requirements of the solution obtained are secondary, and so a measure of time or space complexity in the context of this dissertation are irrelevant.

The NTM has so far applied algorithms to simple, linear data structures such as lists and tables. These structures include up to one level of indirection - a elements pointing to at most one other element. The shortest path problem is much more complex in this regard as it involves a non-linear data structure in the form of a graph. Furthermore, in a connected graph, every element points to at least one other element or, more often, more than one element.

Therefore it can be seen that the shortest path problem is more complex than any problem faced by the NTM in its original definition.

Chapter 3

Design & Implementation

This chapter describes the design and implementation of systems necessary to examine the Research Question (Section 1.2). Section 3.1 outlines the tools used in the implementation. Section 3.2 defines the problem space and presents a method for efficiently generating a subset of this space, and a method for encoding this subset for input to a neural network. Sections 3.3 and 3.4 give detailed accounts of the implementation of the NTM and LSTM models.

3.1 Tools

3.1.1 TensorFlow

TensorFlow is a powerful open-source software library for machine learning developed by Google. It provides a number of pre-built functions to ease the task of constructing neural networks, and it is optimised to run across multiple processors and machines. The models used in this dissertation are implemented using TensorFlow's Python API. This section gives an overview of the concepts necessary in understand the models' design.

The two fundamental concepts in TensorFlow are *tensors* and the *computation graph*. Tensors can be thought of as a n-dimensional arrays or lists. A single scalar, such as the number 4, is known as a *rank*-0 tensor. A list of scalars, such as [1, 2, 3], has a rank of 1. Rank, therefore, describes the dimensionality of a tensor. The *shape* of a tensor describes the size of each dimension. Examples of tensors with corresponding ranks and shapes can be seen in Table 3.1.

Rank	Tensor	Shape
0	4	[]
1	[1, 2, 3]	[3]
2	[[1, 2, 3], [4, 5, 6]]	[2, 3]
3	[[[1, 2], [3, 4]], [[5, 6], [7, 8]]]	[2, 2, 2]

TABLE 3.1: Examples of tensors ordered by rank.

The computation graph is a schematic that describes the operation of a model. Nodes in the graph represent mathematical operations and tensors. Edges in the graph represent the connections between operations and tensors.

The programming of a TensorFlow neural network happens in two steps: building the computation graph and running the computation graph. When building the computation graph, the programmer specifies the entry points to the graph, called *placeholders*, the mathematical operations to be used and how those operations are connected using tensors. When running the computation graph, the graph is initialised, data is passed to the placeholders in a *feed dictionary* and then specific nodes are selected for evaluation.

Figure 3.1 shows a simple example of running a computation graph. Firstly, TensorFlow *session* is created. The session is responsible for allocating resources on whatever machine or group of machines the graph is to be executed on. All variables in the graph are then assigned their initial values using the *init* function. A feed dictionary mapping the value 10 to the placeholder *ginput* is passed to the graph and the value of *output1* is requested, with the result being stored in *answer* (in this case the value 20). Because the value of *output2* is never asked for, TensorFlow will optimise the graph by pruning the *output2* and division nodes during compilation. This is because neither node affects any result asked for.

Section 3.3 describes how these concepts are combined to implement NTM and LSTM models to learn a solution to the shortest path problem.

3.1.2 NTM-TensorFlow

The implementation described in this chapter is based upon an existing TensorFlow implementation of the NTM (Kim, 2015). In its original design, it successfully reproduced the results of the copy task presented in the original NTM paper (Graves,

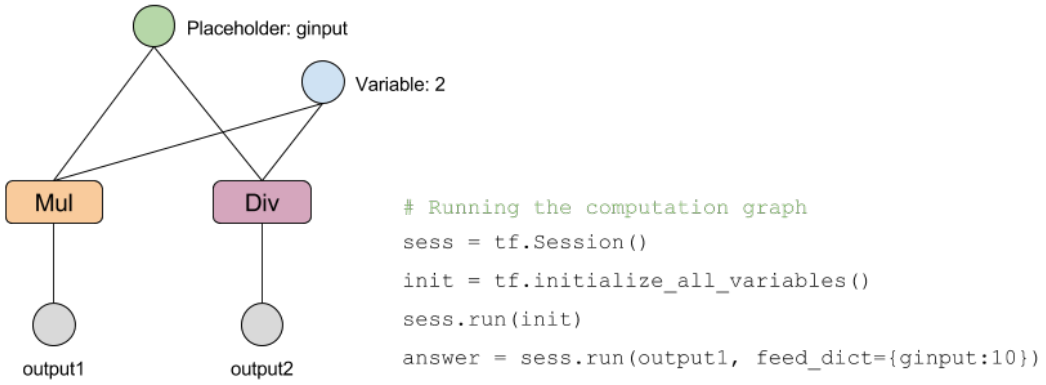

FIGURE 3.1: An example of a TensorFlow computation graph. The code on the right runs the computation graph and requests the value of *output1*, given an input of 10.

Wayne, and Danihelka, 2014). This implementation was chosen for adaptation as it was directly referenced by members of the Google Brain team (Olah and Carter, 2016).

The program consists of a main module that instantiates an NTM Container with an NTM Cell using various support modules. Task modules are used to describe training and testing procedures run with the NTM Container.

As is described in Section 3.3, new NTM Container and Task modules, as well as others, are defined within the existing structure of the program to better suit the shortest path problem.

3.1.3 Pickle

To store and load datasets and results, the application uses the Python Pickle module. This implements binary protocols for serialising and de-serialising Python object structures.

Pickling is the process by which a Python object hierarchy is converted to a byte stream and written to the disk. *Unpickling* is this operation in reverse.

Pickle is part of Python's standard library and can store arbitrarily complex Python data structures. Also, it is fast as it is mainly written in C. This makes Pickle the logical choice for storing objects generated by this application.

3.2 The Dataset

This section describes the method used to generate the dataset used for training and testing the models. It begins with a definition of the contents of the dataset elements and the space they occupy. It then describes how each element is generated individually, and how these elements are combined to form the dataset as a whole. Lastly, it describes how these elements are encoded for the network and the structures in place for marshalling sets of encoded input for training and testing.

3.2.1 The Problem Space

This section defines the contents of an element of the dataset used for training the LSTM and NTM models. It then calculates the absolute size of the problem space - i.e. the number of possible unique elements given their structure.

Each element of the dataset is made up of four parts:

1. Graph data - an object containing the description of the graph's topology. Each node in the graph is assigned a label from 0-9.
2. Start node - the start of the path.
3. End node - the end of the path.
4. Shortest path - the shortest path between the terminal nodes, stored as an edge list (there may multiple, but only 1 is used).

Every graph in the dataset contains 6 nodes and is *connected*, *undirected* and *unweighted*. A connected graph $G = \langle V, E \rangle$ is a graph such that for any two vertices $v_i, v_j \in V$ where $i \neq j$, there exists a path from v_i to v_j. Making all of the graphs connected guarantees that there will be a solution, assuming the terminal nodes are contained in the graph. Edges in undirected, unweighted graphs can be traversed in either direction and with uniform cost.

With 6 nodes, the topology of the graphs and associated shortest paths can vary significantly. The number of possible edges in an n-node graph ranges from $n - 1$ to $\frac{(n)(n-1)}{2}$. Therefore every graph in the dataset has from 5 to 15 edges. The length of shortest paths will then range from 1 to 5. Example elements from the dataset are illustrated in Figure 3.2.

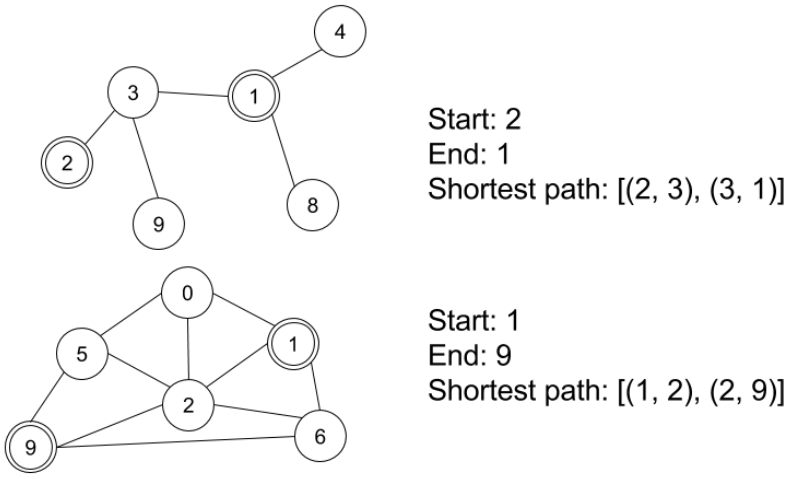

FIGURE 3.2: Two example elements from the dataset made up of a graph, start and end nodes and the shortest path between them.

Graph enumeration describes a class of problems in combinatorics that involve counting the number of a graphs that meet some set of criteria, typically as a function of the number of vertices of the graph. The number of different possible connected, labelled graphs with n-nodes satisfies the recurrence relation (Harary and Palmer, 1973):

$$C_n = 2^{\binom{n}{2}} - \frac{1}{n}\sum_{k=1}^{n-1} k\binom{n}{k}2^{\binom{n-k}{2}}C_k$$

Using this equation, the number of unique possible elements in the dataset can be calculated. For any single n-node graph, the number of possible terminal pairs is the number of possible combinations of 2 labels, multiplied by 2 as both directions are valid. C_n is evaluated for n labelled nodes, but the models accepts labels from 0-9. Therefore the space is expanded by the number of different possible combinations of n labels from a set of 10. The final equation for the number of possible unique elements for a graph size n is then:

$$P_n = 2C_n\binom{n}{2}\binom{10}{n}$$

Table 3.2 shows this function evaluated for up to 10 nodes.

Using 6 nodes per graph, there are approximately 168 million possible elements to sample from. Enumerating the problem space in this way gives context to the size of dataset used in Chapter 4.

n	C_n	P_n
2	1	90
3	4	2880
4	38	95,760
5	728	3,669,120
6	26,704	168,235,200
7	1,866,256	9,405,930,240
8	215,548,592	$\approx 6.339025 \times 10^{11}$
9	66,296,291,072	$\approx 4.773333 \times 10^{13}$
10	34,496,488,594,816	$\approx 3.104684 \times 10^{15}$

TABLE 3.2: The number of unique connected, labelled graphs C_n and corresponding number of elements P_n per number of nodes n.

3.2.2 Graph Element Generation

This section describes the method used in generating each individual element of the dataset. Firstly, a graph is generated and then, from it, the rest of the element.

The approach to graph generation described adheres to the *method of admissible choice* (MAC) (Rodionov and Choo, 2004). In this method, at each step, a choice is made that 1) keeps the graph in the given class and 2) does not break a limitation. In the case of the dataset used in this work, each choice of edge should keep the graph within the class of undirected connected graphs, and there should only be 6 nodes. Wherever possible, choices are randomised. Randomisation is important - if patterns are introduced into the dataset it can unintentionally affect how the network learns.

Graphs are generated in the Graph class. Using the method described above, the class's initialisation function can be broken into two parts: generation of the *minimum spanning tree* and allocation of the remaining edges. The minimum spanning tree of a graph is a subset of edges that connect all vertices without any cycles and with the minimum possible total edge weight. All connected graphs will contain at least one minimum spanning tree, so this can be done first. Then, a random number of remaining edges can be added to the graph, up to the maximum number possible. The pseudocode for generating the edge list of a Graph in this way can be seen in Algorithm 2 below.

Having generated the Graph, the rest of the element can now be constructed. Two nodes are selected at random from Graph's node list. These nodes, together with the graph, are given to a function to compute the shortest path between the two nodes in the form of a edge list. A version of Dijkstra's algorithm (described in

Algorithm 2 Generating random connected n-node graphs.

```
1: function GRAPH(n)
2:     edgeNum = sample(1, range(n − 1, n(n−1)/2 + 1))
3:     V = sample(n, range(0, 10))
4:     P = V × V
5:     E = []
6:     unvisited = V
7:     root = remove(1, unvisited)
8:     Add root to visited
9:     count = 0
10:    while unvisited is not empty do          ▷ Build the spanning tree
11:        v1 = remove(1, unvisited)
12:        v2 = sample(1, visited)
13:        Add (v1, v2) to E
14:        Add (v1, v2) to visited
15:        Remove (v1, v2) from P
16:        count++
17:    while count <= edgeNum do                ▷ Add any remaining edges
18:        e = remove(1, P)
19:        Add e to E
```

Chapter 2) is implemented to achieve this.

The method described in the section generates individual dataset elements. The next section describes how these elements are combined as a whole.

3.2.3 Dataset Generation

Dataset generation is handled by the *dataset_util* module. Research work encountered a number of challenges involving the generation of the dataset. Firstly, every element needed to be unique, but also randomly generated to ensure an even distribution. Secondly, curriculum learning should be supported, to enable better learning. Finally, the process should be as fast and memory efficient as possible. This section presents an approach that addresses each of these challenges.

As described in the previous section, the element generation process involves a lot of randomisation. This is necessary, but it means that there is no guarantee that a newly generated element is different to any of the previously generated elements.

To solve this, every time the program generates a new element, it checks to see if that same element has been generated previously. When generating hundreds of thousands of elements, this can become quite costly, as every previous element must be checked at each step. To make this faster, a hash table is used. Python's

dictionary data structure allows values to be keyed by any immutable type. So, when an element is generated, the graph and terminal nodes of the element are converted into a tuple (immutable in Python) and used as a the key into the hash map, setting its value to 1. Then, when subsequent elements are generated, the same procedure is followed and if there is a collision it is known that this element has already been created. Hash table lookups and insertions take an expected $\mathcal{O}(1)$ time, which makes this a fast solution.

The same graphs can be represented by differently-ordered edge lists. To ensure collisions in cases such as this, each graph also contains a reference to its own *adjacency matrix*. An adjacency matrix contains a row and column for each node label. If there exists an edge between node i and node j, then in adjacency matrix A, $A_{i,j} = 1$ and $A_{j,i} = 1$. The adjacency matrix is used in place of the edge list in the hashing operation.

Curriculum learning is implemented in this dissertation. Each successive lesson contains solutions with increasingly-long shortest paths. To begin with, the network is shown examples in which the solution shortest path is of length one. Next, it is shown examples in which the solution shortest path is of length two, and so on until it reaches $n - 1$ for graph size n. Furthermore, within each lesson, the graphs are ordered by number of edges, starting with the lowest.

Implementing curriculum learning in this way raised the question of how to divide the dataset between each lesson. When comparing the elements in terms of shortest path length, the data are *intrinsically imbalanced*. Any dataset that exhibits unequal distribution between its classes is considered imbalanced, it is intrinsic if that imbalance is a direct result of the nature of the dataspace. If we consider the path length to be the data's class, the number of elements of class 1 will be significantly higher than of class 5, using 6-node graphs. Training over imbalanced data can significantly compromise the performance of most standard learning algorithms (He and Garcia, 2009).

To solve this, *random undersampling* is used. In this method, random elements of the majority class are discarded so that the number of elements of each class is equal. Therefore the dataset contains an equal number of elements with each possible path length.

Generating the very large dataset required to train and test the network requires a lot of memory. To minimize this, the generation function exploits the fact that curriculum learning is used to create a memory and time-efficient algorithm. It begins by creating three sets, each represented as lists: a training set, a validation set and a test set. Each set is then made-up of 5 bins - one for each path length - that are lists of elements. As each unique element is generated, it is distributed into its corresponding bin. Firstly, the training set bins are filled, then the validation set bins and then the test set bins. As each bin becomes full (according to overall size of the set divided by the number of bins), it is written to the disk using Pickle. The bin is then deleted from memory and the garbage collector is manually called. It proceeds in this way until there are no bins left and the entire dataset has been generated and stored on the disk. This process is illustrated in Figure 3.3.

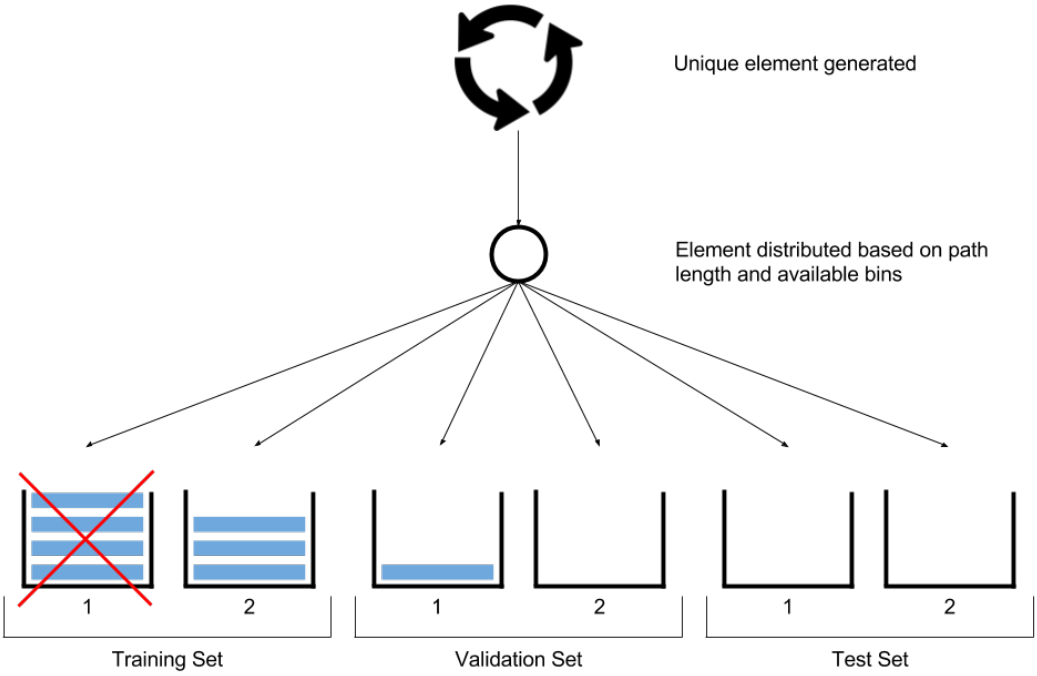

FIGURE 3.3: The dataset generation process with paths lengths up to 2. A unique dataset element is first generated and then distributed to the correct bin. The first bin in the training set has become full and has been deleted, as indicated by the cross appearing over it. As a result, the first bin of the validation set has begun to be filled. The second bin of the validation set will not begin to be filled until the second bin of the training set has been filled. This process continues until all bins have been deleted.

This method is efficient in time and space, within the bounds of its stochastic

nature. This method has both the feature that no element will be unnecessarily discarded, making it efficient in time, and that bins are deleted as quickly as possible, making it efficient in space. Because the path length distribution is skewed, the lower path length bins will fill up faster. By making the bins independent, this method uses the fact that some elements occur more frequently to free memory faster. Furthermore, because each set is given a priority, no more than five bins will ever be active at any one time, keeping average memory usage low.

Using this method, a balanced dataset that supports curriculum learning can be created efficiently.

3.2.4 Sequencing the Shortest Path Problem

Because both the NTM and LSTM models are forms of recurrent neural network, they require their input in the form of a sequence. The shortest path problem is sequenced similarly to how graph tasks were sequenced for processing by a differentiable neural computer (Graves et al., 2016). This section first defines how a graph is represented as a sequence, and then problem as a whole.

Vertices in the graph are represented by a label from 0-9. Edges in the graph are represented by pairs of labels. The entire graph is then represented by a sequence of edges. Because undirected graphs are used, edges will not be duplicated in the edge sequence - i.e.

$$(1, 4) = (4, 1), \tag{3.1}$$

so either one or the other can be present, but not both. The shortest path solution is also represented as an edge list in this way.

Labelling the digits from 0-9 is convenient for people because we understand how to interpret them as just labels, but a network learning to interpret graphs does not have this intuition. By giving the nodes ordinal values, it is open to be interpreted that a node labelled 9 is somehow greater than a node labelled 4. To remove this possibility, the digits of each label are represented by a 10-way *one-hot encoding*. One-hot encodings transform categorical features to a format that works better with machine learning algorithms. Each class is given a column in a boolean vector. If an element is of a particular class, the corresponding column in its one-hot encoding

vector contains a 1, and all other values are 0. In the case of node labels for this problem, each label value 0-9 corresponds to an index in a 10-bit vector. For example, a node labelled 6 would be encoded as [0000001000] (note, this uses vector notation so indexes are counted left to right). Each edge is therefore represented by a 20-bit vector

Each dataset element produces two distinct sequences: an input sequence and a target sequence. The input sequence is broken down into four phases: a graph description phase, a query phase, a planning phase and an answer phase. During the graph description phase the input vectors (containing one-hot encoded edges) are fed to the network in a random order. In the query phase, two vertices (the start and end of the path sought) are given to the network in a single 20-bit vector. During the 10-step planning phase, no input is given to the network and it is given time to perform computation and attempt to determine the shortest path. Then, during the answer phase, the network is asked for its solution.

The target sequence only contains the sequenced shortest path solution and is padded with zeros at the beginning so that it lines up with the answer phase of the input. This is so that the output of the output of the network can be directly compared with the target output.

Each label requires 10 inputs to the graph, and each edge requires 20. Two additional bits are required to indicate the current phase of input. So in total the input vectors are of size 22 for each step. The target vectors are then of size 20.

The input sequence's overall length will be dependent on both the number of edges in the graph, and the length of the shortest path. For a graph with n nodes, the maximum length of input sequence L_{max} can be calculated by summing the maximum size of each phase:

$$\begin{aligned} L_{max} &= \frac{n(n-1)}{2} + 1 + 10 + (n-1) \\ &= \frac{n^2 + n + 20}{2} \end{aligned} \tag{3.2}$$

Using this method, elements of the dataset can be converted into sequences suitable for processing by both the neural Turing machine and long short-term memory

networks.

3.2.5 Generating Input

When training and testing the networks, the dataset is represented using the Dataset class. This class is responsible for loading pickled datasets, encoding their elements, and marshalling them for input to the networks.

The Dataset is initialised with a directory containing pickled dataset files. The Dataset object then provides three functions:

1. *get_train_data*
2. *get_val_data*
3. *get_test_data*

Each function takes a *size* argument that specifies the number of examples to be included in the returned sets. They load the corresponding pickled dataset bins (either train, test or validation bins) and extract an equal number of elements from each, summing to the size of set asked for. The elements are then ordered by number of edges in the graph. These two steps are done as part of curriculum learning. Each element is then sequenced using the method described in Section 3.2.4, and a list of input sequences and a corresponding list of target sequences are returned.

As is discussed in Section 3.3.3, the Dataset object is used to easily gather inputs for training and testing the network.

3.3 Neural Turing Machine Model

The three core modules in the application are the NTM Cell, the NTM Container and the shortest path Task. The NTM Cell and the NTM Container build the model's computation graph and the Task module is used for running the computation graph.

Neural Turing machines are types of recurrent neural network. If we consider them, as we did in Section 2.3.2, as rolled out through time, then describing their TensorFlow computation graph involves first describing each discrete time-step. This is done in the NTM Cell. How these time-steps are connected to input, output and

each other is described by the NTM Container. This section describes first the smallest building block, the NTM Cell, and then use it to gradually build up a view of the entire computation graph. Lastly, how the Task makes use of this computation graph in training for and solving the problem is described.

3.3.1 NTM Cell

This dissertation makes use of a pre-existing implementation of an NTM Cell (Kim, 2015). This section gives a general overview of the NTM Cell's construction including initialisation, input and output, construction of its components and interaction with other cells. Section 3.3.2 goes into more depth about how the NTM Cell is used in the construction of the models in this dissertation.

Within the NTM Cell are contained the controller, memory, and read and write heads of the NTM. Initialising an NTM Cell involves defining the number of controller layers, number of controller hidden units and dimensions of the NTM's memory. At this point, no operations have been added to the computation graph.

The NTM Cell's *call* function simulates one time-step in the network. Therefore, it accepts a single element of the sequence input to the network, and returns the corresponding element of the output sequence for that input. It also accepts a state input, received from the previous time-step, and returns an updated state, to be passed to the next time-step. The input and output are represented by tensors. The state is represented by a dictionary of tensors, made up of:

- Memory - a tensor representation of the NTM's addressable memory bank, with the dimensions given in initialisation of the cell.
- Read weights - the weights governing how much and what parts of a memory address will be read.
- Write weights - the weights governing how much and what parts of a memory address will written to or erased.
- Read list - the list of all tensors read from the NTM's memory at each time-step.
- Output list - the list of all outputs of the network at each time-step.

- Hidden states - a list of tensors containing the hidden states of the LSTM controller at each time-step.

The first call to the NTM Cell should contain a reserved *start symbol*. This initialises the memory and creates a the starting state dictionary.

The *call* function constructs the NTM Cell's computation graph. Firstly, the LSTM controller is defined. It contains the input, forget and output gates defined in Section 2.3.1. These gates are created using the current input and the last element of the read list. The outputs of these gates are then combined to produce an updated hidden state and a new controller output. Both are added to their respective lists, and these list are added to a new state dictionary.

Next, the read and write heads used to interact with the memory are constructed. These contain the gates described in Section 2.3.2 necessary for addressing the memory. The read head uses the read weights to read from the position defined by the addressing gates. The value obtained is appended to the read list and new read weights are derived. The write head uses the write weights, addressing gates, and last output to construct the add and erase gates. Also, it derives new write weights. The outputs of the add and erase gates are applied to the memory. Then, the updated memory and weights are added to the new state dictionary.

The output returned by the NTM Cell is the last output in the output list, with a linear transformation applied. Therefor, the output of the NTM Cell is in the form of *logits*. Logits are the unscaled outputs of linear neural network layers.

For every gate defined in the NTM Cell, and its output, a linear transformation is applied. The weights and biases used in these linear transformations are known as the model's *parameters*. It is these parameters that are changed as part of learning using gradient descent.

Every time the NTM Cell is called, a new computation graph as described above is created. However, each set of gates contained in each new computation graph needs to use the same set of model parameters in its operation as the last. If it does not, each individual time-step in the sequence will have its own set of parameters to be trained. To solve this, the NTM Cell uses TensorFlow's *get_variable* function. When called, this function can be passed a *name* string. If a variable has already been

created with this name in the entire computation graph, it is returned. Otherwise, a new variable with that name is created and returned. With this in mind, each gate is given its own name and a single linear transformation function is defined. Every time the function is used, a name is given to it and it finds the weights and biases corresponding to that name. In this way, the cell can be called multiple times, with multiple controllers, read and write heads and gates constructed, but the same weights and biases will always be used all of them.

The NTM Cell forms the heart of the model. It defines the computation graph of a single time-step, where a single input and state are passed through the controller and memory of the NTM to produce a new output and updated state. This layer of abstraction then allows one to arrange and combine these time-steps to create a larger computation graph for solving the shortest problem.

3.3.2 NTM Container

If the NTM Cell is the heart of the model, the NTM Container is the body. It defines how input data are accepted, how those data flow through the model, and how the output data are used to test and train the model. This section steps through the construction of the various components of the NTM Container and describes their operation in detail.

Initial Connections

When initialised, the NTM Container is given an NTM Cell, relevant hyper-parameter settings, the length of the sequence to be input and the maximum size of graph to be accepted.

Once initialised, the *build_model* function creates the computation graph using the NTM Cell and other parameters given during initialization. Firstly, a reserved *start symbol* is input to the cell. The start symbol initialises the cell's memory bank and returns the dictionary containing its initial state. Then, for every element in the sequence up to the answer phase, the function:

1. Creates placeholder tensors for the input and target vectors. The input tensor is of shape [22] and the target tensor is of shape [20]. The input tensor is appended to a list of inputs, and the target tensor to a list of targets.

2. Passes the input placeholder and state dictionary to the cell, which returns a tensor and a dictionary containing its output and its updated state respectively. The output tensor is of shape [20].

For the first vector in the sequence, the state input to the cell is the initial state received after passing the cell the start symbol. For each subsequent step, the state passed to cell is the output state generated from the previous input. The model, as described up to this point, is illustrated in Figure 3.4.

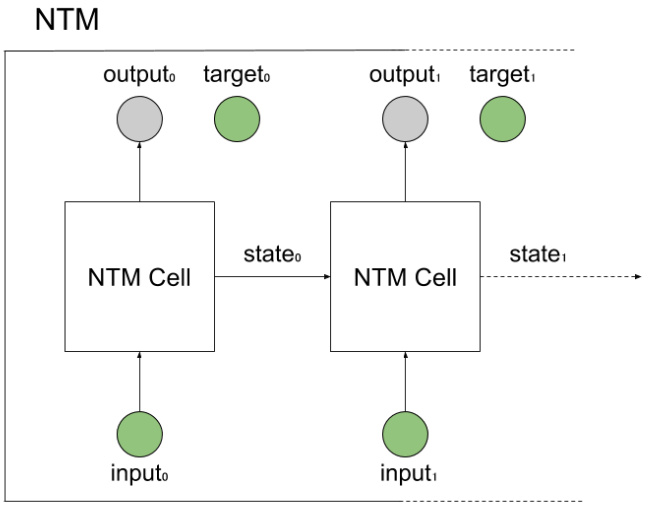

FIGURE 3.4: A partial view of the NTM model. The green circles labelled *input* and *target* are placeholder tensors. The target placeholders are not yet connected to anything.

Answer Phase

A constraint on the solution to the shortest path problem is that the output of each time-step is dependent on the output of the previous time-step. That is, the correct choice of next edge to take is dependent on what node you are currently at, which is decided by what edges you have previously taken.

Consider the example illustrated in Figure 3.5. The shortest path from node 0 to node 2 is sought. In its answer phase, the model has already output the edge [0, 1],

indicating it is now at node 1. When deciding the next edge to select, knowing that it is currently at node 1 is useful information as it indicates what edges are available for selection - i.e. [1, 2] or [1, 5].

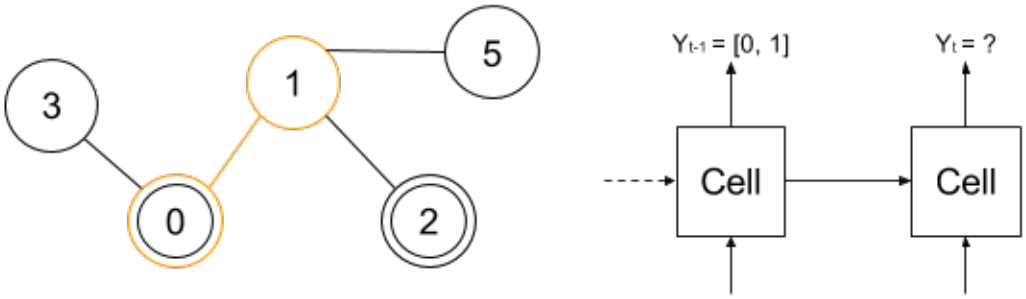

FIGURE 3.5: A network with its output highlighted on a graph. The start and end nodes of the path sought are illustrated by the double circles in the graph.

This constraint is reflected in the structure of the connections in the answer phase of the network. The answer phase is the portion of the end of the input sequence when the network is asked for its response to the query. When training the model, the input for a time-step t in the answer phase is the target tensor for time-step $t-1$. This is tells the network the correct answer to the *last* step, so that it can make an informed prediction about its *current* step. Then, during testing, instead of using the target, the output tensor of time-step $t-1$ is given as input to time-step t. The first step in the answer phase is given an input containing only the phase bits, as there is no previous path edge from which to draw. Constructing the connections in this way, differently for training and testing, allows the network to learn to trust the input during training, and only be concerned with outputting the correct current prediction. Then, during testing, its own correct predictions should propagate through the time-steps.

However, as discussed in the previous section, the output of the NTM cell is a tensor of logits. This raw output can have a wide range of values, but the network only accepts correctly encoded node label pairs. Therefore, an extra step is required before feeding it back into the network. Firstly, the output logits are split into two separate shape [10] tensors and passed through individual *softmax* layers. The softmax function is a logistic function that accepts and n-dimensional vector containing arbitrary real values and squashes them into an n-dimensional vector in which the

values sum to 1. It is described by the equation (Goodfellow, Bengio, and Courville, 2016):

$$softmax(x)_i = \frac{\exp x_i}{\sum_{j=1}^{n} \exp x_j} \tag{3.3}$$

The results of the softmax functions can be thought of as the probability distribution of for each possible label for each node of an edge. From there, the *argmax* function is used to return the index with the highest probability - the most probable label - in each shape [10] tensor. Each index returned by argmax is then one-hot encoded and they are recombined into a shape [20] tensor - representing the predicted edge.

Finally, both the target tensor and newly-refined answer from time-step $t-1$ must be prepended with [1,1] to indicate the input is in the answer phase. The answer phase as described is illustrated in Figure 3.6.

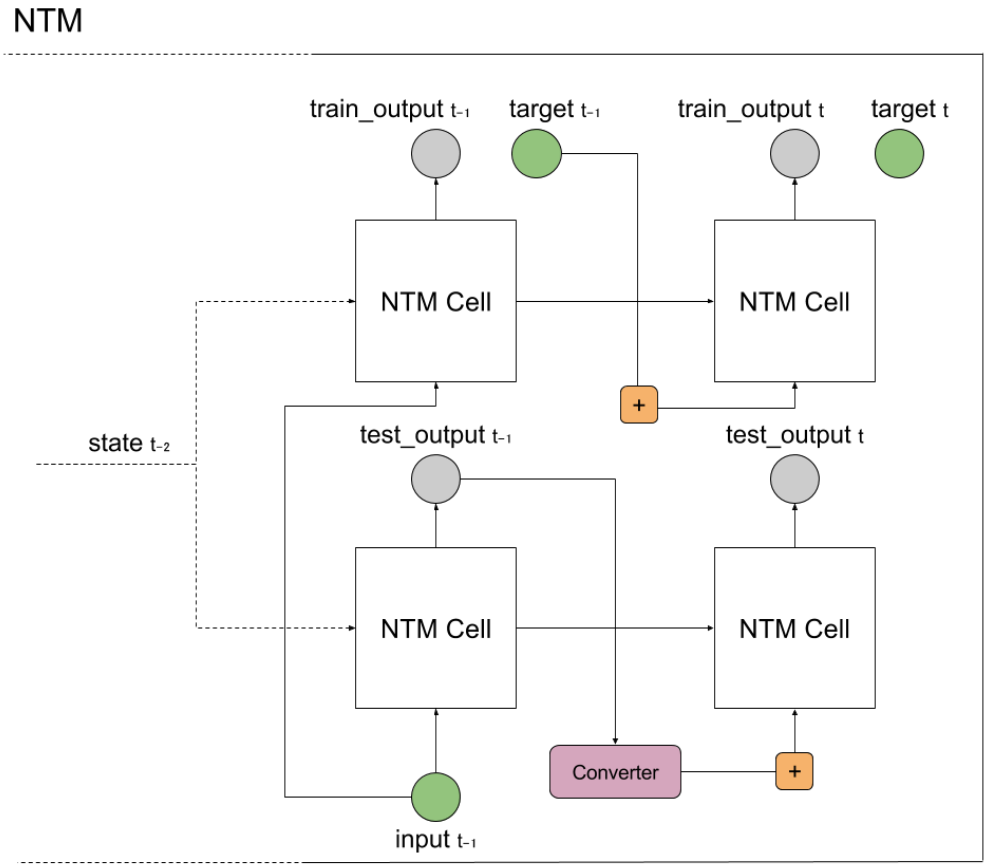

FIGURE 3.6: A partial view of the NTM computation graph, showing the structure of the connections in the answer phase. The *converter* node contains within it the softmax, argmax and one-hot encoding functions necessary to convert the network's output into an acceptable input. The + node prepends phase bits to the tensors.

Loss and Error Functions

Structuring the answer phase as described creates two distinct lists of output placeholders: testing output placeholders and training output placeholders. Where A is the length of the answer phase and T is the length of the input sequence, the first $T - A$ elements in each of these lists will be the same. The training output is used in the model's loss function, whereas the testing output is used in the model's error function.

The loss function of a network is a measure of how wrong it was. The loss function used in this model is described as the log-probability of predicting an edge, and therefore the sum of the log-probability of correctly predicting each digit. For an input sequence x, with output sequence y and target sequence z (both of length T), the loss function is:

$$\mathcal{L}(x, z) = -\sum_{t=1}^{T} \left\{ A(t) \sum_{d=0}^{1} \log[Pr(z_t^d \mid y_t^d)] \right\} \tag{3.4}$$

where z_t^d is the target at time t for digit d, y_t^d is the softmax distribution over digit d returned by the network at time t, and $A(t)$ is an indicator function whose value was 1 during the answer phase and 0 otherwise (so that output returned when not needed is ignored).

To calculate the loss, the model's training answer must be extracted from the list of training output placeholders. This is so that the loss is only calculated over the answer phase of the output. This is done by first using the TensorFlow *stack* function to transform the outputs from a list of rank-1 tensors, to a rank-2 tensor. This new tensor is of shape [T, 20], where T is the length of the sequence and 20 dimension of the output. The target list is transformed in the same way, and then reduced to a rank-1 tensor of shape [65] using *reduce_max* along its second axis. This is now a tensor that contains a 1 in all of the positions where an answer is expected, and a 0 everywhere else. Using *tile* and *reshape* function, this is then expanded back out to a rank-2 tensor that contains a shape [20] tensor of all 1s in all of the positions where an answer is expected. This can now be used to mask the output tensor by multiplying the two together, element-wise. The generation of the mask is illustrated in Figure 3.7.

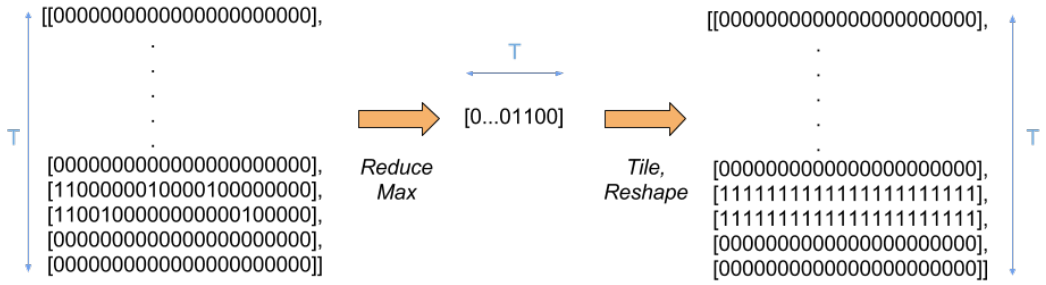

FIGURE 3.7: The process of creating the output mask. The target tensor on the left is reduced along its second axis with a max function. Then, the result is tiled and reshaped to match the dimensions of the output. T represents the length of the input sequence.

Once the answer has been extracted, the *sequence_loss* function is used to calculate the loss over the entire sequence. This function accepts a list of output tensors, a list of target tensors and a reference to a loss function. It applies the given loss function to every pair of elements from the output and target lists and returns the sum of the results. The loss function passed to the *sequence_loss* function should therefore calculates the inner terms of the overall loss functions defined in Equation 3.4 above:

$$- A(t) \sum_{d=0}^{1} \log[Pr(z_t^d \mid y_t^d)] \tag{3.5}$$

The negative log-likelihood of one distribution given another is also known as the *cross-entropy* between those two distributions. As such, TensorFlow's *softmax_cross_entropy_with_logits* function is used to calculate the inner terms. This built-in function first applies a softmax layer to the given logits and then calculates the cross-entropy between the result and the given target. These two calculations being performed in tandem is common in neural network design, but there exist numerically unstable corner cases in doing its calculation. The *softmax_cross_entropy_with_logits* function handles these corner cases in a mathematically sound way while also optimising the two functions. The indicator function $A(t)$ is handled by the mask that has been applied to the output.

As the cross-entropy is to be calculated over each digit, the output logits and target tensors are first split into two shape [10] tensors each. Then, *softmax_cross_entropy_with_logits* is called twice and results are summed. The resulting value is then returned to the *sequence_loss* function, which sums it with the other loss values calculated at each time-step in the sequence. This process is illustrated in Figure 3.8. This implementation also makes use of ℓ_2 regularization, which is applied at this point. It is calculated using the output layer's weights with the *l2_loss* function, multiplied by a tunable β value and added to the final loss.

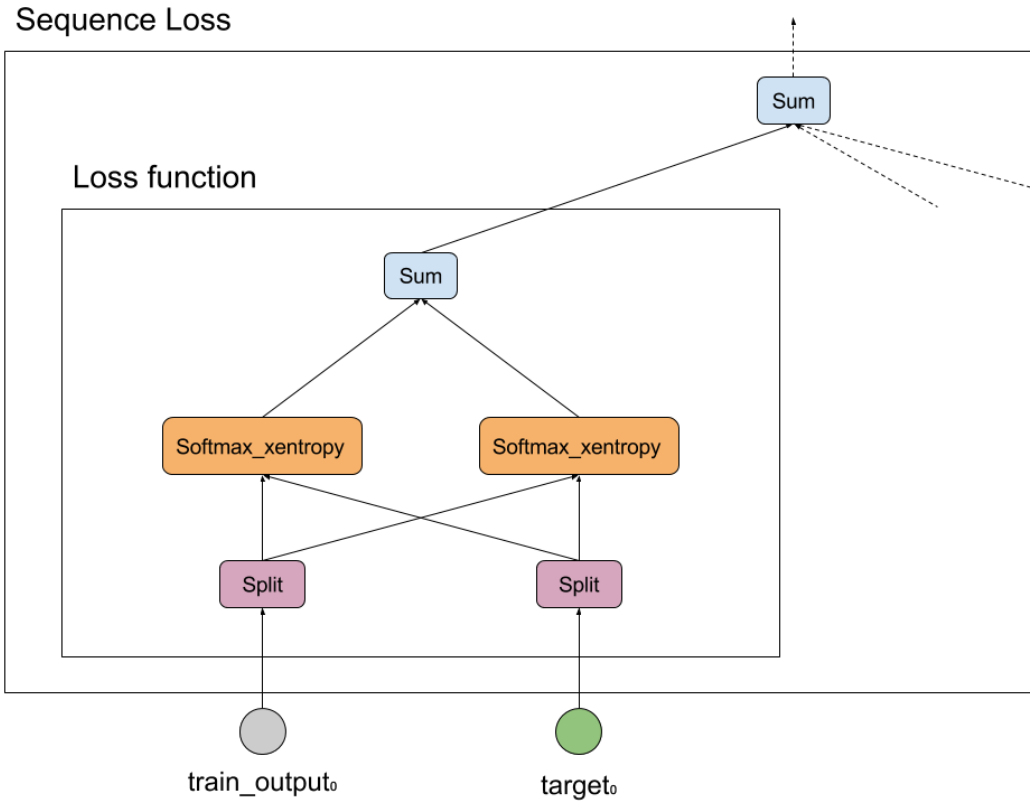

FIGURE 3.8: The partial view of the NTM computation graph showing the sequence loss calculation.

Having defined the connections necessary to calculate the loss with the list of training outputs, the list of testing outputs is used to define the error function. The two values returned from the error function are:

1. Error position - a tensor of shape $[n - 1]$, where n is the number of nodes in the graph. Each index corresponds to an output in the answer phase. A 1 is contained at every index for which the output was wrong, a 0 otherwise.

2. Final output - a tensor of shape [P, 2], where A is the length of the solution path.

Obtaining these values involves performing a series of operations over the test output and target tensors. Firstly, the output is split, is passed through a softmax layer and has argmax applied in the same method used during the answer phase. It is important to note, however, that this operation was being performed over rank-1 tensors during the answer phase, but it is now being performed over the entire output sequence. This adds a dimension to each of the operations as both the test output and target tensors are of shape [T, 20]. The result of these three operations is two tensors of shape [T], where the value at index i corresponds to a node label output by the network at time-step i.

Copies of these tensors are made, and they are combined using the *stack* function in the first dimension. This leaves a single tensor of shape [T, 2] where the pair of values at index i correspond to the network's edge prediction of time-step i. The leading and following zeros are stripped to produce the final output.

Having isolated the network's predicted node labels, the target labels are found using the same splitting and argmax procedure. The softmax layer is not necessary, as the target distribution is already known to be in the correct range. The predicted label and target label tensors are now combined using an element-wise *not_equal* function. This leaves two shape [T] tensors containing a 1 at the indices where the network's node label prediction was incorrect.

Finally, the two node-error tensors are combined using and element-wise *logical_or* function. This gives a single shape [T] tensor containing a 1 at the indices where the network's edge prediction was incorrect. This is stripped of its leading zeros to give the error position output.

Optimiser

There are three components that go intro training a model: the model parameters, the loss function, and the optimiser. The model's parameters and loss function have already been defined, and here they are combined with an optimiser to allow the model to be trained.

Firstly, the model parameters defined in the NTM Cell must be gathered. TensorFlow has a feature that, when creating a variable, allows you to set a *trainable* boolean. Any variables for which this boolean is marked "true" can then be easily gathered into a single tenor with the *get_variables* function. This functionality is designed exactly for this purpose: so that model parameters can be defined in different parts of the code and easily gathered when defining the optimisation function.

Next, the loss function and gathered model parameters are given to the *gradients* function. This function calculates the partial derivatives of the loss with respect to each of the model's parameters. It is at this point that gradient clipping is applied. The *clip_by_value* function clips the tensors' values to the specified maximum and minimum values - in this case 10 and -10.

TensorFlow provides support for a number of optimisers that implement the various different types of gradient descent algorithm. The NTM makes use of the RMSProp optimiser, which is initialised with values for starting learning rate, weight decay, and momentum. Then, the optimisation function for the model is defined as the RMSProp optimiser's *apply_gradients* function. For each input during training, this optimisation function is evaluated causing gradients to be applied to the model parameters, thus training the model.

Evaluation

At this point, the entire computation graph has been defined. During training and testing, different values and functions can be selected for evaluation.

For example, during training the optimisation function will be called. This, in turn, will cause the gradients to be evaluated, which will cause the loss to be evaluated, which will cause the output to be evaluated and so on, back through the computation graph to the entry points (placeholders or constants).

Importantly, only nodes depended upon by the function evaluated will be evaluated themselves. When testing the model, the model will not be optimised. Therefore, the loss, gradient, optimisation and other nodes will be pruned from the computation graph as it is initialised. Similarly, the error function and associated nodes *will* be evaluated, where this was not the case during training.

What functions are evaluated and when is defined in the shortest path Task.

3.3.3 Shortest Path Task

The shortest path Task is responsible for running the computation graph. It receives as input a fully-constructed network and the current session and performs one of two actions: training or testing.

Training involves passing inputs and target outputs to the network and training its parameters using its optimisation function. It operates as follows:

1. Initialise all variables - call TensorFlow's *init_all_variables* function.
2. Create a Dataset object - specify a file location from which to load the dataset.
3. Get training data - the Dataset object is asked for and returns input and target sets of the specified size. Each element of the input set contains a fully-encoded input sequence for the network, with each element of the target containing the corresponding target output sequence.
4. For input sequence element of the input set:
   (a) Create a feed dictionary - this maps each element of the input sequence and corresponding target sequence to the relevant placeholders in the network.
   (b) Evaluate - The network's optimisation and loss functions are run within the session using the feed dictionary.
   (c) Save - the network's loss is recorded. Additionally, the models parameters are saved to a *checkpoint* file every ten thousand iterations. TensorFlow checkpoint files allow you to store information, including model parameters or the computation graph itself, to an external file that can then be reloaded.

In this way the network is trained over each element of the training set and the training loss is recorded.

Testing the network follows a similar procedure, although using different data and evaluating different functions:

1. Load - the model's parameters are loaded from a checkpoint file.

2. Initialise all variables - call TensorFlow's *init_all_variables* function.

3. Create a Dataset object - specify a file location from which to load the dataset.

4. Get testing data - input and target sets are return, similarly to training, except they have either been drawn from the validation set or the test set, depending on the stage of experimentation (as discussed in Chapter 4).

5. For each input sequence element of the input set:

   (a) Create a feed dictionary - this maps each element of the input sequence and corresponding target sequence to the relevant placeholders in the network.

   (b) Evaluate - The network's error function is evaluated.

   (c) Save - the output of the error function is appended to a list of outputs

6. Store results - the list of outputs is pickled.

Storing raw results in this way allowed for greater flexibility for analysis later in the project.

With the network's computation graph fully defined, the shortest path Task is used to first train the network using a training set generated by the Dataset object, and store its parameters to a checkpoint file. Having trained the network, its parameter can be loaded from a checkpoint file and tested on either a validation or test set obtained from the Dataset object.

3.3.4 Padding

Section 3.3.2 described passing output and target vectors as input to the network during the answer phase to improve its performance. One requirement of this approach is knowing where the answer phase is going to be at the point of constructing the graph so that the necessary connections can be made. However, as described in Section 3.2.4, the sequence length will vary based on the number of edges in the graph.

One solution to this requirement is to build a model for each possible sequence length, and pass the input sequence to the corresponding model based on its size.

This method is wasteful, however. Instead, a method is used that is similar to the one seen in the TensorFlow module implementing sequence-to-sequence models, introduced in (Cho et al., 2014). By this method, the description phase of the input sequence is padded using zero vectors to be the length of the longest possible edge list. Similarly, the answer phase itself is padded to the length of the longest possible shortest path, which will be of length $n - 1$.

The answer phase is now static across all input sequences, ensuring the connections described in Section 3.3.2 work as intended.

3.4 Long Short-Term Memory Model

As part of this dissertation, an LSTM model is implemented to use as a baseline. Rather than implement a complete network from scratch, the similarities between the LSTM and NTM models are exploited. The only difference between the two is that the NTM has an addressable memory bank, whereas the LSTM does not. The controller of the NTM is itself an LSTM network.

Section 3.3.1 describes the construction of the NTM Cell. To implement the LSTM model, an "LSTM mode" is added to the NTM Cell that, when active, does not create the connections between the controller and the memory. Without the memory attached, the network is now a standard LSTM network. Thus the construction and execution of the LSTM network is the exact same as that of the NTM, except that the cell passed to the Container has LSTM mode activated.

Implementing the LSTM network like this has the benefit that it is guarantees that all aspects of the networks are identical, except for the difference under test - the memory.

Chapter 4

Evaluation

This chapter details the evaluation of the work performed as part of the research for this dissertation. The aim of the evaluation undertaken is to firstly optimise the NTM model to achieve best-possible performance in solving the shortest path problem and, secondly, to assess this performance after optimisation. This evaluation is given context by a comparison to an LSTM network trained on the same problem. The chapter begins with an outline of the procedure followed in achieving these aims, followed by a presentation of the results of this procedure.

4.1 Procedure

Evaluation is performed in three stages. This section outlines each of these stages, their purpose, and the methodology observed in their execution. The networks are trained using stochastic gradient descent trained for a single epoch. The evaluation described in Sections 4.1.1 and 4.1.2 are carried out using a validation set of size 40,000. The evaluation described in Section 4.1.2 is carried out on five separate test sets of size 70,000. A detailed specification of the hardware and software used in these evaluations is given in Appendix A.

4.1.1 Learning Curve

The first stage of evaluation determines what size of training set to use when training the models. *Learning curves* show the model performance as function of the training sample size and can be used to help determine the sample size need to train a good model (Beleites et al., 2013).

Each model is first trained using training sets of increasing size with a common default set of hyper-parameters. The smallest set used contains 10,000 examples, and the largest set used contains 50,000 examples. Each model is trained five times, at intervals of 10,000 from the smallest set through the largest set.

The performance of the model at each training set size is evaluated using the validation set. The results are compared and the best training set size for each model selected.

4.1.2 Hyper-Parameter Optimisation

Having found the best training set size for each model, the hyper-paramters of each must be optimised. In this process, different values for each of the models' hyper-parameters are chosen and trained over. Their performance is then measured using the validation set. Tuning the hyper-parameters of a neural network is a time-consuming process. A typical approach is to take pairs of hyper-parameters and exhaustively test each possible combination over a certain discrete range of values for each. This is referred to as *grid search*, and is often combined with a *manual search*, in which hyper-parameters are individually changed by the researcher. This is computationally expensive, however, as the number of pairs of values increases exponentially with each additional hyper-parameter.

Instead, the NTM and LSTM models' hyper-parameters are optimised using *random search*. In this process, settings for each hyper-parameter are chosen at random from a range of possible values. Research has shown (Bergstra and Bengio, 2012) that a random search involving 60 hyper-parameter samples performs as well and, in some cases, better than a combination of grid search and manual search. There is a probabilistic explanation for this: for any distribution over a sample space with a finite maximum, the maximum of 60 random observations lies within the top 5% of the true maximum, with 95% probability (Zheng, 2015). Furthermore, as each hyper-parameter sample is completely independent, this becomes an *embarrassingly parallel* problem. This is the set of problems that can be easily divided into components that can be executed concurrently (Herlihy and Shavit, 2012). Using this method greatly decreased time spent training.

Thus, the hyper-parameters are given by sampling from the following distributions:

1. The number of layers in the NTM controller/LSTM network is chosen from 1, 2 or 3 with equal probability.
2. The learning rate is chosen log-uniformly from 0.00001 to 1.
3. The RMSProp momentum is chosen uniformly from 0.001 to 1.
4. The RMSProp weight decay is chosen uniformly from 0.001 to 1.
5. The number of hidden units in the NTM controller/LSTM network is chosen log-uniformly from 100 to 600.
6. The ℓ_2 regularization of the output weight matrices of the final layer is chosen to be 0 (with probability 0.5), or chosen log-uniformly from 10^{-7} to 10^{-4}

Sampling log-uniformly involves drawing uniformly from the log domain between $log(A)$ and $log(B)$, where A and B are the range limits, and exponentiating to get a number between A and B. This gives a higher probability for lower values, as is favourable with some hyper-parameters such as learning rate. This is illustrated in Figure 4.1. The ranges used are adapted from (Bergstra and Bengio, 2012) with some changes made. For example, the upper limit of hidden units is lowered from 4000 to 600. This is because training neural networks with a large number of hidden units has high memory demands, and the hardware used to evaluate this dissertation was limited in that regard.

Optimising the hyper-parameters in this way afforded a high confidence that an optimum solution had been found.

4.1.3 Final and Generalisation Testing

In the last stage of evaluation, optimum hyper-parameter settings are used to determine the models' final error rate and their ability to generalise.

Up until this point, the models have been trained using the training set, and had their performance measured against a validation set. In the final testing stage, the models' performance is evaluated against the test set.

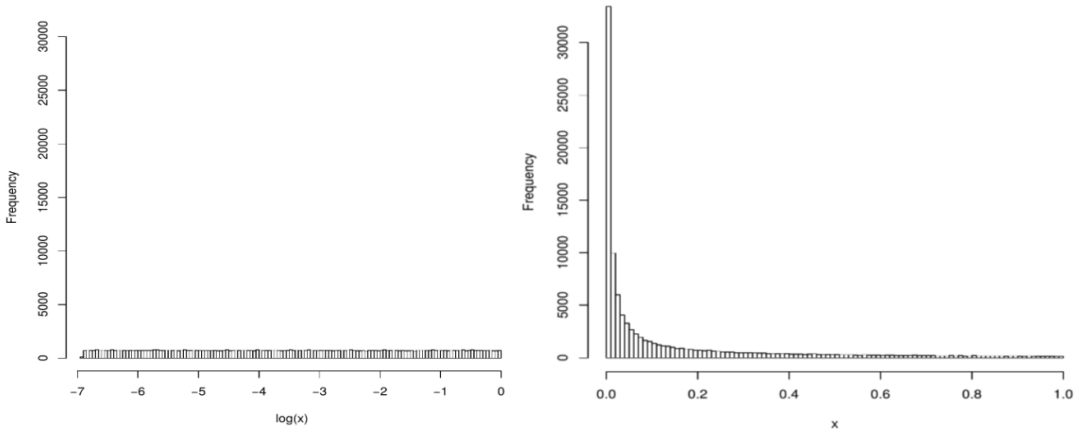

FIGURE 4.1: On the left is the uniform distribution between the log of the range limits 0.001 and 1 after 100,000 samples. On the right is the distribution of the exponentiation.

The fundamental goal of any machine learning model is to generalise to problems unseen (Domingos, 2012). The dataset used to train the models in this project contained only graphs with 6 nodes. To test the models' ability to generalise even beyond unseen graphs of the same size, their performance when shown graphs with 7, 8, 9 and 10 nodes is also measured.

The results of this final stage of evaluation are thus the ultimate measure of the models' performance in solving the shortest path problem.

4.1.4 Comparing Results

The method used to compare results across different training conditions is the *paired t-test*. Letting x and y be two sets of results from running the model over the same validation or test set having been trained under different conditions, the procedure is as follows:

1. Calculate the difference $d_i = x_i - y_i$ between each two outputs given the same input.

2. Calculate the mean difference $\bar{d}$.

3. Calculate the standard deviation of the difference, s_d, and use it to calculate the standard error of the mean difference, $SE(\bar{d}) = \frac{s_d}{\sqrt{n}}$, where n is the number of samples.

4. Calculate the t-statistic, given by $T = \frac{\bar{d}}{SE(\bar{d})}$.

5. Calculate the degrees of freedom $df = n - 1$.

6. Using tables of t-distribution, obtain a p-value using T and df.

The null hypothesis in the case of these tests is that there is no difference, in either direction, between the performance of two models. The alternative hypothesis is then that there is a difference. The P-value is the probability that the results observed would be at least as inconsistent with the alternative hypothesis, assuming the alternative hypothesis is true.

Using paired t-tests, it can be determined whether or not changes to the training of a model result in statistically significant changes in validation or test set performance.

4.2 Results

This section presents the results obtained in each stage of experimentation described in Section 4.1.

4.2.1 Learning Curve

Figure 4.2 shows the plotting of the learning curves from both the NTM and LSTM models, corresponding to the values shown in Table 4.1. The hyper-parameters used for these test were unchanged from the original paper (Graves, Wayne, and Danihelka, 2014) and are shown in Table 4.2. These hyper-parameters are not optimised but should provide a valid starting point for this test.

As can be seen from Table 4.1, the NTM model achieved the lowest error rate when trained using 40,000 examples. To test if this result was significant, a paired t-test was performed between the results of the model when trained over 40,000 examples and every other training set size using the method described in Section 4.1.4. The resulting t-statistics and corresponding p-values are shown in Table 4.3. As can be seen from these calculations, the NTM model performs with a statistically significantly lower error rate when trained over 40,000 examples than any other training set size.

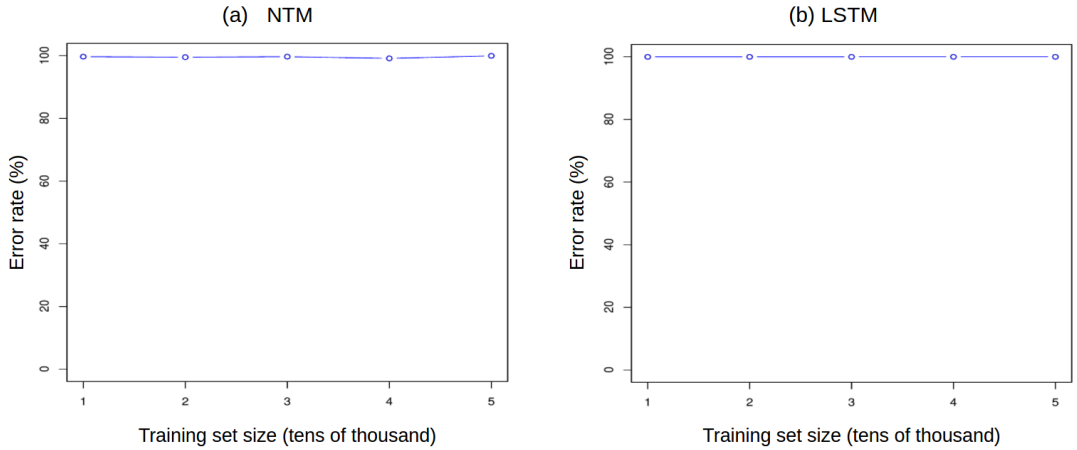

FIGURE 4.2: The learning curves for the NTM (a) and LSTM (b) models.

Model	10k	20k	30k	40k	50k
NTM	99.72	99.55	99.72	99.16	99.98
LSTM	99.99	100	100	100	100

TABLE 4.1: The performance of each model (%) on the validation set, having been trained on differently sized training sets.

Layers	Hidden Units	Memory Dimension	Learning Rate	Momentum	Decay	ℓ_2
1	100	128	0.0001	0.9	0.95	0

TABLE 4.2: The hyper-parameter settings used for determining the models' learning curves. The number of layers refers to the controller in the case of the NTM, and the network itself in the case of the LSTM. The memory size is only applicable to the NTM model.

Value	10k	20k	30k	50k
T	-10.5030	-6.8474	-10.6827	-17.8064
p	<0.0001	<0.0001	<0.0001	<0.0001

TABLE 4.3: The t-statistics and corresponding p-values obtain from performing a paired t-test between the results of training over 40,000 examples and every other training set size.

Performing the same calculations on the LSTM learning curve results showed no statistical significance in the difference between performance of the model with 10,000 examples and any other training set size. Therefore to determine the size of training set to use, a second error rate is defined that afforded greater granularity in the results. This second error rate, the *edge error rate*, is defined as the percentage of incorrect edges returned by the network, per path length. The best performing model is then defined as the model that has the outright lowest error rate in the

largest number of path lengths. The results of this measurement are shown in Table 4.4.

As can be seen in the results, the model trained using 10,000 examples had the lowest error rate in three of the five path lengths. To test the whether these differences are statistically significant, again paired t-tests were performed using the these results and the results after every other training set size in the same path length. The results of this analysis, listed in Table 4.5, show that 2 of the three error rates were statistically significantly lower than every other training set size.

Through training both models using different training set sizes, and analysing the results for statistical significance the optimum training set size has been found for each model. Moving forward to the next stage of experimentation, the NTM model is trained using 40,000 examples, while the LSTM model is trained using 10,000 examples.

Path Length	10k	20k	30k	40k	50k
1	100	100	100	100	100
2	**99.81**	99.86	99.88	100	100
3	**98.54**	99.34	99.32	99.26	99.29
4	96,43	96.12	95.96	**95.77**	96.21
5	**98.8**	99.22	99.3	99.29	99.28

TABLE 4.4: The edge error rate, per path length, for each training set size. In bold are the best results for each path length.

Value	Path Length	10k	20k	30k	40k
T	2	-1.11	-1.57	-5.49	-5.49
p	2	0.27	0.12	<0.0001	<0.0001
T	3	-5.42	-5.15	-4.38	-4.78
p	3	<0.0001	<0.0001	<0.0001	<0.0001
T	5	-5.67	-6.78	-6.57	-6.49
p	5	<0.0001	<0.0001	<0.0001	<0.0001

TABLE 4.5: T-statistics and corresponding p-values obtained from comparing the LSTM network's edge error rate (for path lengths 2, 3 and 5) having been trained with 10,000 examples, with that of every other training set size.

4.2.2 Hyper-Parameter Optimisation

Tables 4.6 and 4.7 show the five best-performing hyper-parameter configurations for the NTM and LSTM respectively. A full list of all configurations and resulting error rates over all 60 runs is available in Appendix B.

The lowest error rate achieved by the NTM network was 96.48%. This is statistically significantly better than the next-best performance (p = <0.0001). The lowest error rate achieved by the LSTM network was 98.2%. This too is statistically significantly better than the next-best performance (p = 0.0034).

Rank	Layers	Hidden	Memory Dimension	LR	Momentum	Decay	ℓ_2	Error Rate (%)
1	3	222	128	1.6e-5	0.2582	0.9247	0	96.48
2	3	138	128	1.0e-5	0.3068	0.8409	0	97.15
3	2	114	128	0.002505	0.0111	0.0727	4.94e-6	97.88
4	3	154	128	0.000431	0.0806	0.4295	3.554e-5	98.21
5	2	309	256	0.002525	0.0826	0.0258	1.209e-5	99.29

TABLE 4.6: Top 5 NTM hyper-parameter settings.

Rank	Layers	Hidden	LR	Momentum	Decay	ℓ_2	Error Rate
1	2	163	0.002961	0.4594	0.2516	1.29e-6	98.2
2	2	411	0.001428	0.017	0.5667	0	98.47
3	3	170	0.000914	0.2541	0.3717	3.392e-5	99.38
4	2	150	0.0033	0.0021	0.0093	1.4e-7	99.39
5	1	122	0.002267	0.4257	0.1471	0	99.51

TABLE 4.7: Top 5 NTM hyper-parameter settings.

The best-performing hyper-parameter settings of each network after 60 iterations of random search have been found. The networks trained using these settings are used in the final stage of evaluation.

4.2.3 Final and Generalisation Testing

The performance of the networks on the 6-node test set, and the test sets containing graphs with more nodes, are shown in Table 4.8.

One thing to note is that the test sets containing graphs with more than 6 nodes are not balanced. This is because there are only 10 possible labels for a node and as the number of nodes in the graph increases, the number of possible graph elements with longer paths decrease. It decreases to the point where there are not enough $n-1$ and $n-2$ length graphs to be able to balance the set. As a result, the performance of then networks on the 6-node set is not directly comparable with that on the test

sets. The performances of the networks on the 7, 8, 9 and 10-node test sets can be meaningfully compared.

Network	6-node	7-node	8-node	9-node	10-node
NTM	96.44	93.47	96.14	97.58	98.14
LSTM	98.23	94.62	94.87	94.81	95.41

TABLE 4.8: The test set error rates (%).

As the results show, the NTM network out-performed the LSTM network on graphs with 6 and 7 nodes. The LSTM network did generalise better than the NTM network, achieving a lower error rate on the 8, 9 and 10-node test sets. All differences are statistically significant to a very high confidence ($p = <0.00001$). The next section analyses these results and give them context within the problem as a whole.

Chapter 5

Discussion

This chapter discusses the results of the dissertation work. It begins with an analysis of the results presented in Chapter 4. Following this, it includes suggestions as to how the models' performance could be improved in light of these results.

5.1 Analysis of Results

This chapter analyses in greater depth the performance of the two networks on the 6-node test set. By breaking down the results further, it aims to show more clearly how the network performed in the context of the theory discussed in Chapter 2. The results are first broken down by length of solution path, and the resulting distribution is analysed in the context of the entire problem. The requirements of the problem are then relaxed and a more general view of how each network was able to reason about graph structures is given. This analysis is then used to show how each network performed at a partial solution.

As shown in Section 4.2.3, the NTM network achieved an error rate of 96.44% on the 6-node set, where the LSTM network achieved an error rate of 98.23%. Within this set are elements with shortest path lengths of 1 through 5, Figure 5.1 shows the networks' performance broken down by length of solution path.

In the cases where the solution path is a single edge, the correct shortest path output is actually the query itself. This is illustrated in Figure 5.2. In these cases, to know that the query is the answer requires the network to know that the query is contained in the graph that was describe to it. This is a perfect example of the content-based addressing of which the NTM is capable. It could write the entire graph description to its memory bank as it receives it. Then, it could use the query

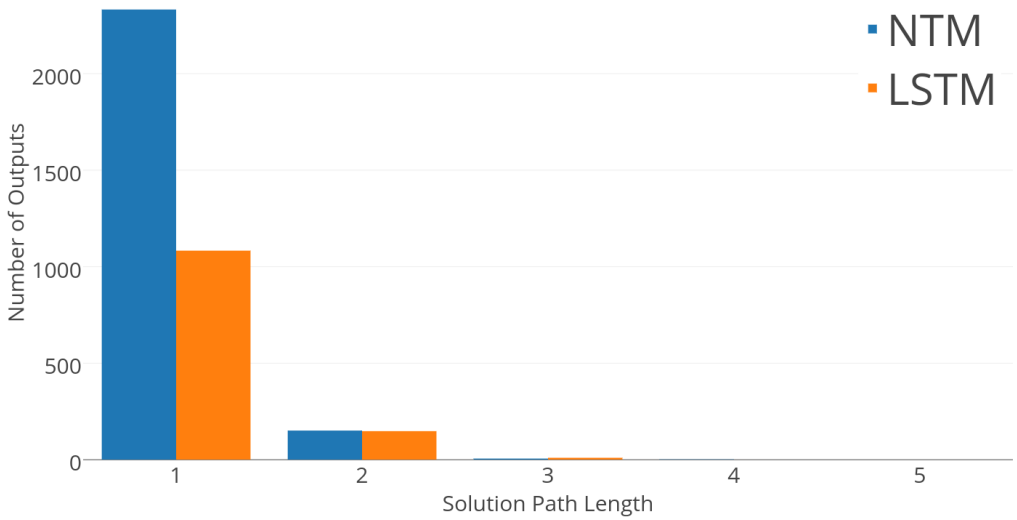

FIGURE 5.1: The number of correct solutions per path length.

it receives as a key into its memory, addressing based on content. If that query is present in the graph it has stored, it knows that it is the solution.

For the LSTM network, this is a much more difficult task. It has only its cell state in which to store information and has less sophisticated means of accessing this information. This could explain why the NTM correctly identifies roughly twice as many length-one shortest paths as the LSTM.

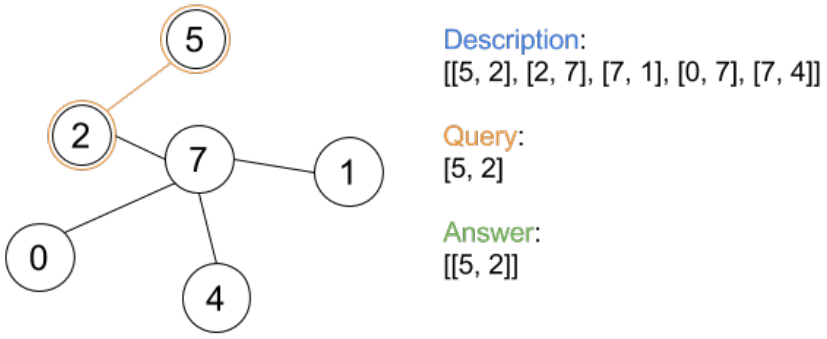

FIGURE 5.2: An example of an element in which the solution path is of length 1.

However, this approach to finding a solution does not work for any other path length. This is evident in the plots of the networks' losses shown in Figure 5.3. As the first curriculum learning lesson ends, the networks are introduced to training examples in which the solution shortest paths are of length 2. At this point, both loss values jump considerably as very little of what was learned in the first lesson

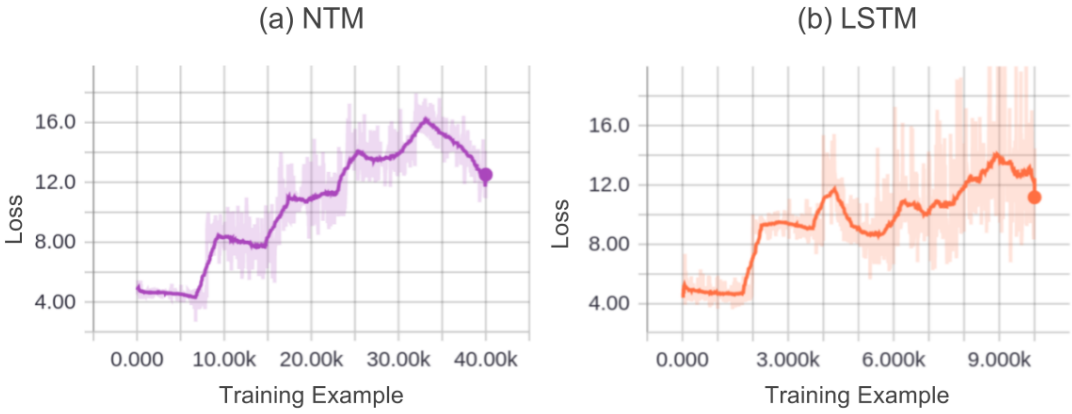

FIGURE 5.3: The training loss of the optimal hyper-parameter settings for the NTM (a) and LSTM (b). SGD loss has a lot of fluctuation between examples so the plot has been smoothed to highlight trends.

can be transferred to the second lesson. The loss still jumps with each subsequent lesson, but less so, indicating that the later lessons have more in common.

For training examples with solution paths longer the 1 edge, the training loss of the LSTM network appears to remain lower than that of the NTM. However, this does not translate to better outputs. Figure 5.4 shows the average fraction of the output path that was correct (given as a percentage), per path length. From this it can be seen that for elements with longer solution path lengths, the LSTM network still outputs fewer correct edges on average than the NTM. This could be explained by overfitting - a concept describe in Section 2.2.4. The use of ℓ_2 regularisation attempts to curb overfitting in the networks, but additional techniques such as *cross-validation* (Ng, 1997) could be used to reduce it further.

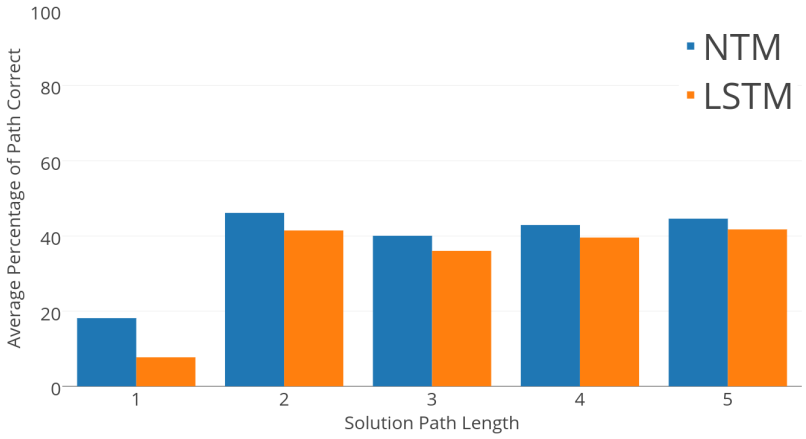

FIGURE 5.4: The average percentage of output path that was correct, per length of solution path.

One of the aims of this research was to see if NTMs could learn to reason about graph data. Beyond looking at how well the network was able to solve the problem, it is also relevant how *sensible* its answers were in the context of graph structures. For this reason, the number of outputs from the network that were valid *walks*. A valid walk in this case is defined as a sequence of edges output by the network in which each edge is contained in the graph given to the network, and each edge is connected to the previous edge in the sequence by a node. Examples of valid walks given a graph are illustrated in Figure 5.5. As can be seen from the results shown in Figure 5.6, the NTM network output significantly more valid walks than the LSTM network. This indicates some ability on the NTM's part to reason about a graph structure.

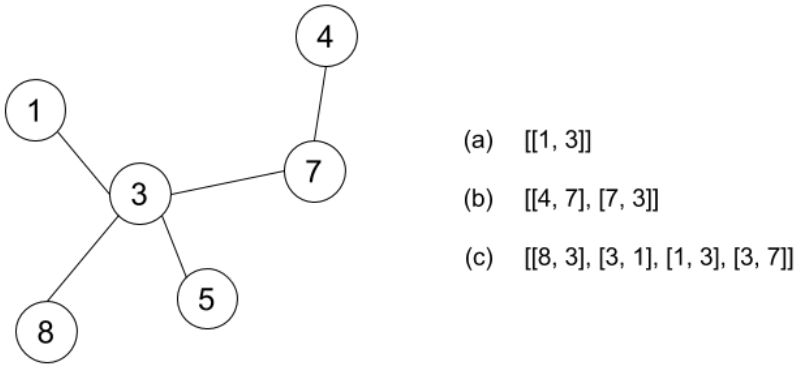

FIGURE 5.5: Walks (a), (b) and (c) are all valid given the graph shown on the left.

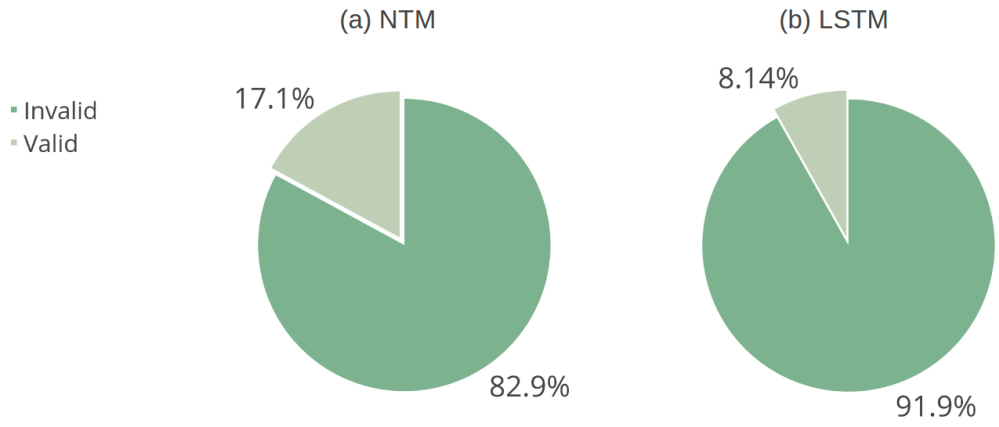

FIGURE 5.6: The percentage of paths returned by the NTM (a) and LSTM (b) that were valid.

Some of these walks will be solutions to the problem, and some will be completely unrelated. To make sense of this, four classes of walk are defined:

1. *Complete* - these are the solutions to the problem. They are the shortest path between the start and end nodes.

2. *Deviations* - these are walks that started from the correct node, but did not finish at the end node. These had the potential to be correct answers, but made a wrong decision at some point along the way.

3. *False Starts* - these are walks that did not start from the correct node, but finished at the end node.

4. *Lost* - these are walks the neither started from the correct node, nor finished at the end node.

Figure 5.7 shows how the valid walks are divided among these classes. An interesting statistic to note is that of the valid walks output by the NTM, 16.9% ended at the correct node, having start from an incorrect start node. This means that, of all the valid paths that did not start at the correct node, slightly less than half finished at the correct end node. In a 6-node graph, this is more than can be expected by chance. As the graphs are undirected, a shortest path from v_1 to v_2 is the reverse of the shortest path from v_2 to v_1. A high number of paths ending at the correct node could indicate the network is solving the problem by travelling from the end node to the start node in some cases.

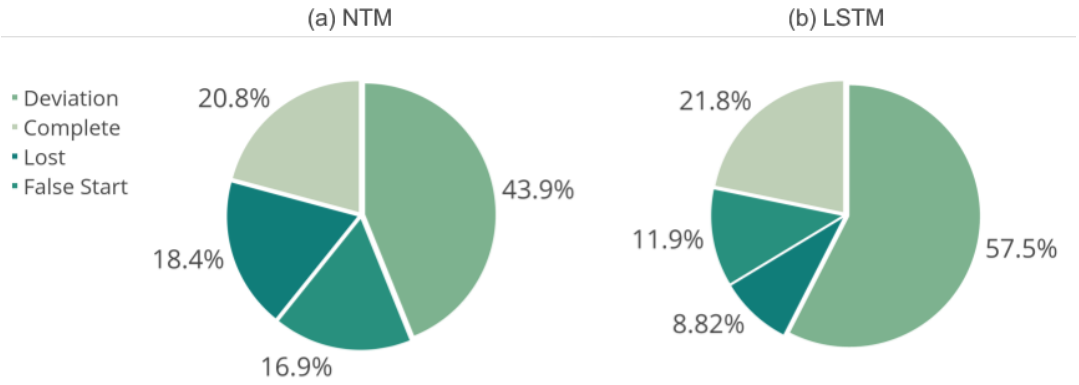

FIGURE 5.7: The division of valid walks among the defined classes for the NTM (a) and LSTM (b) networks.

Consider the subway example from Chapter 2. A solution that brings us to within one or two stations of our final destination is better than none at all. Using the complete and deviation paths, the distribution of paths over their distance to the end node is calculated. This describes how close the network got to the end node, given that it output a valid path that started from the correct node. This distribution is shown in Figure 5.8.

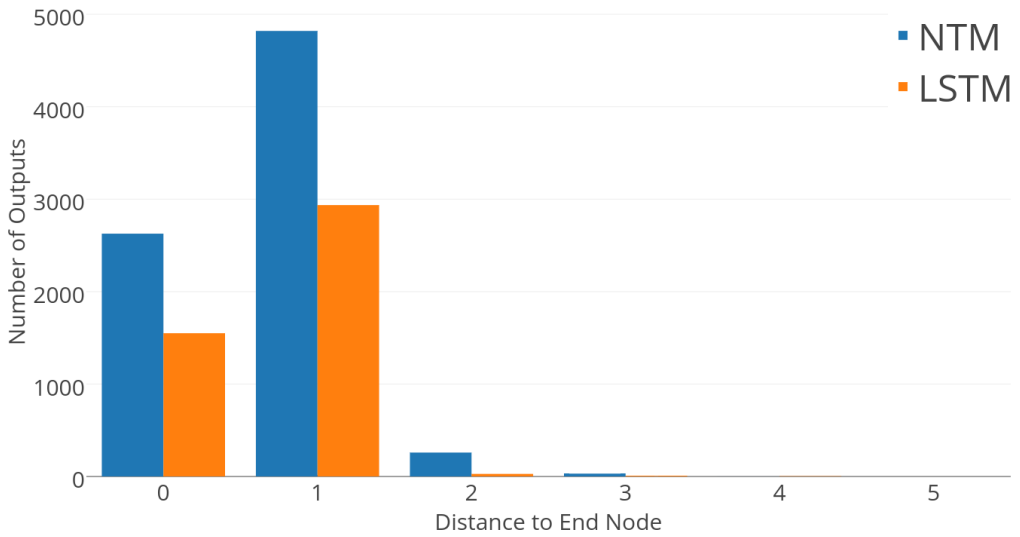

FIGURE 5.8: The distance of each valid walk output to the end node.

Using this information it is determine how often the network can find the shortest path to a node within one edge from the specified end node. This is done by summing the number of complete paths, with the number of deviations that finished at node one edge from the correct end node. The results of this calculation are in are in Figure 5.9. This metric shows that the NTM network has more potential for immediate improvement than the LSTM network, with a higher percentage of paths getting to within one node of correct end node.

The results of the evaluation performed show that the NTM network can find the shortest path between two nodes in a graph with a lower error rate than the LSTM network. Having broken down the results, it appears that the NTM network is using its memory as an aid in simple solutions, but does not learn to find more complicated paths with great success. Analysing the output in the context of the graph shows that the NTM network has learned to give more reasonable answers

than the LSTM network. In some cases these answers are partial solutions - paths that lead to within one edge of the full solution.

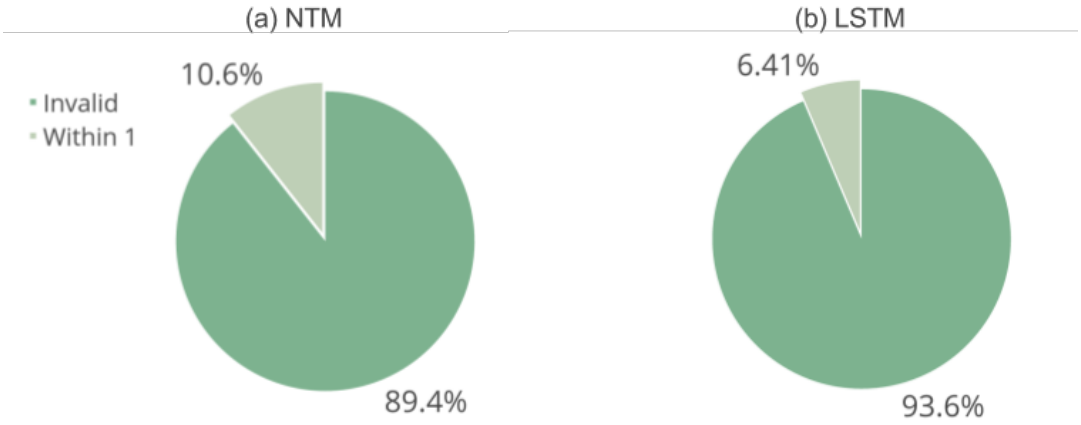

FIGURE 5.9: Shortest paths to within 1 edge of the specified end node.

5.2 Model Improvements

As seen in the previous section, the NTM network does outperform the LSTM network at both finding shortest paths, and reasoning over graph structures. Neither network, however, performs well in absolute terms - the error rate on the 6-node test set was over 95% for both networks. This section discusses a possible reason for this performance, and suggests changes that could be made to the network that would improve it.

Every learner must embody some knowledge or assumptions beyond the data it's given in order to generalise beyond the data (Domingos, 2012). One weakness in the implementation described in this work is that not enough is done to convey to the network what the problem is it is trying to solve. As seen in Chapter 4, the NTM network had limited success in finding shortest paths, but was much more capable of returning sensible output in the context of the graph. This disconnect shows that the network can learn to reason about graphs, but maybe that it did not understand the problem that was asked of it.

Chapter 3 described how the answer phase is structured to try and embody knowledge about the problem in the networks, but other architectures take this concept further. Pointer networks (Vinyals, Fortunato, and Jaitly, 2015) output discrete tokens corresponding to positions in the input sequence. They have been shown to

be able to approximate solutions to other problems from graph theory, such as the Travelling Salesman Problem. One approach that could be taken to better convey the problem to the networks could be to more closely model their structure to the problem.

The loss function used by the networks is a generic sequence loss that measures the distance between the output and target vectors. There is scope to tailor this loss function to more specifically describe the problem to the network.

Recent research involving similar networks (Graves et al., 2016) borrows techniques from reinforcement learning to give more context to the network's output and loss when dealing with graphs. Edge choices are described as actions, and the network's output as a policy. For each choice of action, the network's output is renormalised over the available actions and the most probable is chosen. The loss then becomes a function over the networks policy and a target optimal policy. For some tasks, a second network that learns to predict an expected value for each action is introduced, adding even greater context to the loss.

The results presented in this dissertation indicate that the NTM network has the potential to do better at the task of finding the shortest path in graph if the task more clearly conveyed to it. Other neural network models introduce more relevant structure or more context-sensitive loss functions to better convey task objectives, and the networks described in this work could benefit from these approaches. Depending on the degree to which each approach was adopted, they could result in significant changes to the model structure, taking it further away from its original definition. For this reason, they are not explored in this dissertation, but they offer a clear avenue for improvement for future work.

Chapter 6

Conclusion

Chapter 6 concludes this dissertation. It opens with an assessment and summary of the work performed, with reference to the original objectives of the research, and closes with some final thoughts on the topic.

6.1 Objective Assessment

Having defined a research question and set of objectives in Chapter 1, this section aims to contextualise the work described in this dissertation with respect to these definitions.

This dissertation opens with a survey of the the research surrounding the NTM. This section describes the concepts necessary to understand the NTM, from those underpinning it all the way up to those defining its context in the world of machine learning as whole. This information, along with a problem definition, lays the foundations of the work done as part of this dissertation.

Chapter 3 describes the successful implementation of a method for procedurally generating a dataset necessary to learn a solution to the shortest path problem. It does so efficiently in time and space and with guarantees of uniqueness.

Chapter 3 goes on to detail a method for implementing both NTM and LSTM models using TensorFlow. These implementations incorporate ideas introduced in Chapter 2, and are designed with structures to specifically handle problem posed to them.

Chapter 4 defines a comprehensive method for training and optimising neural networks, informed by current research. This method iteratively refines different settings that affect neural networks' performance. The chapter then goes on to present

the results achieved when following this method.

Chapter 5 evaluates these results, in terms of outright performance, relative difference, and the models' abilities to understand graph structures. Neither the NTM's nor the LSTM's performance proved significant in absolute terms - neither model appeared able to learn an algorithm to reliably find the shortest path in a given graph.

The results, however, a statistically significant difference between the performance of the two models. In this case, the NTM's memory and addressing mechanism appear to have given it the upper hand by allowing it to use its memory as a lookup in cases where the solution paths were of length one. However, the LSTM proved better at generalising to examples with a greater number of nodes, outperforming the NTM on test sets with 8, 9 and 10-node graphs.

Beyond solving the problem, this dissertation also explores the networks' abilities to reason over graph structures. In this regard, the NTM again outperformed the LSTM, and with a wider margin than in any other measure. More often than the LSTM, the NTM was able to produce answers that, if not right, had a chance of being sensible given the graph.

6.2 Final Words

Machine learning is an exciting field of research. It is not a new field, but the recent growth in size of available datasets has made it more relevant than ever before.

Neural networks constitute a model at the centre of machine learning. Configurable in a myriad of different ways, new neural networks, and their capabilities, are constantly being explored.

This dissertation serves as another such exploration. It defines a system and method for exploring the ability of the neural Turing machine to use its addressable memory to reason over graph structures. The results of this dissertation showed that the neural Turing machine could not do this reliably, but that it performed better than long short-term memory. Furthermore, the results show some ability for the neural Turing machine to comprehend graph structures, indicating potential for future research.

Bibliography

Bahdanau, Dzmitry, Kyunghyun Cho, and Yoshua Bengio (2014). "Neural Machine Translation by Jointly Learning to Align and Translate". In: *CoRR* abs/1409.0473.

Beleites, Claudia et al. (2013). "Sample size planning for classification models". In: *Analytica Chimica Acta* 760, pp. 25 –33.

Bengio, Y., P. Simard, and P. Frasconi (1994). "Learning long-term dependencies with gradient descent is difficult". In: *IEEE Transactions on Neural Networks* 5.2, pp. 157–166.

Bengio, Yoshua et al. (2009). "Curriculum Learning". In: *Proceedings of the 26th Annual International Conference on Machine Learning*. ICML '09. Montreal, Quebec, Canada: ACM, pp. 41–48. ISBN: 978-1-60558-516-1.

Bergstra, James and Yoshua Bengio (2012). "Random Search for Hyper-parameter Optimization". In: *J. Mach. Learn. Res.* 13, pp. 281–305. ISSN: 1532-4435.

Bottou, Léon (2010). "Large-Scale Machine Learning with Stochastic Gradient Descent". In: *Proceedings of COMPSTAT'2010: 19th International Conference on Computational StatisticsParis France, August 22-27, 2010 Keynote, Invited and Contributed Papers*. Ed. by Yves Lechevallier and Gilbert Saporta. Heidelberg: Physica-Verlag HD, pp. 177–186. ISBN: 978-3-7908-2604-3.

Cho, Kyunghyun et al. (2014). "Learning Phrase Representations using RNN Encoder-Decoder for Statistical Machine Translation". In: *CoRR* abs/1406.1078.

Diamond, Adele (2013). "Executive Functions". In: *Annual Review of Psychology* 64.1, pp. 135–168.

Dijkstra, Edsger. W. (1959). "A note on two problems in connexion with graphs." In: *Numerische Mathematik* 1, pp. 269–271.

Domingos, Pedro (2012). "A Few Useful Things to Know About Machine Learning". In: *Commun. ACM* 55.10, pp. 78–87. ISSN: 0001-0782.

Garey Michael RJohnson, David S (1979). *Computers and intractability*. 1st ed. W.H. Freeman.

Goodfellow, Ian, Yoshua Bengio, and Aaron Courville (2016). "Deep Learning". Book in preparation for MIT Press.

Graves, Alex (2016). "Adaptive Computation Time for Recurrent Neural Networks". In: *CoRR* abs/1603.08983.

Graves, Alex, Greg Wayne, and Ivo Danihelka (2014). "Neural Turing Machines". In: *CoRR* abs/1410.5401.

Graves, Alex et al. (2016). "Hybrid computing using a neural network with dynamic external memory". In: *Nature* 538.7626, pp. 471–476.

Harary, Frank and Edgard M Palmer (1973). *Graphical enumeration*. 1st ed. Academic press, p. 7.

He, H. and E. A. Garcia (2009). "Learning from Imbalanced Data". In: *IEEE Transactions on Knowledge and Data Engineering* 21.9, pp. 1263–1284. ISSN: 1041-4347.

He, Kaiming et al. (2015). "Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification". In: *CoRR* abs/1502.01852.

Herlihy, Maurice and Nir Shavit (2012). *The Art of Multiprocessor Programming, Revised Reprint*. 1st ed. Elsevier Science.

Hinton, Geoffrey (2014). *Neural Networks for Machine Learning*. URL: `http://www.cs.toronto.edu/~tijmen/csc321/slides/lecture_slides_lec6.pdf`.

Hochreiter, Sepp and Jürgen Schmidhuber (1997). "Long Short-Term Memory". In: *Neural Computation* 9.8, pp. 1735–1780.

Hopcroft, John E and Jeffrey D Ullman (1979). *Introduction to automata theory, languages, and computation*. Addison-Wesley.

Kim, Taehoon (2015). *NTM-tensorflow*. `https://github.com/carpedm20/NTM-tensorflow`.

Krizhevsky, Alex, Ilya Sutskever, and Geoffrey E Hinton (2012). "ImageNet Classification with Deep Convolutional Neural Networks". In: *Advances in Neural Information Processing Systems 25*. Ed. by F. Pereira et al. Curran Associates, Inc., pp. 1097–1105.

Lecun, Yann, Yoshua Bengio, and Geoffrey Hinton (2015). "Deep learning". In: *Nature* 521, pp. 436–444. ISSN: 0028-0836.

Miller, George A (1965). *The magical number seven, plus or minus two*. 1st ed. College Division of Bobbs-Merrill Co.

Neelakantan, Arvind, Quoc V. Le, and Ilya Sutskever (2015). "Neural Programmer: Inducing Latent Programs with Gradient Descent". In: *CoRR* abs/1511.04834.

Ng, Andrew Y. (1997). "Preventing "Overfitting" of Cross-Validation Data". In: *Proceedings of the Fourteenth International Conference on Machine Learning*. ICML '97. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., pp. 245–253. ISBN: 1-55860-486-3.

Olah, Chris (2016). *Neural Networks, Manifolds, and Topology*. URL: `http://colah.github.io/posts/2014-03-NN-Manifolds-Topology/` (visited on 11/10/2016).

Olah, Chris and Shan Carter (2016). *Attention and Augmented Recurrent Neural Networks*. http://distill.pub/2016/augmented-rnns/.

Qian, Ning (1999). "On the momentum term in gradient descent learning algorithms". In: *Neural Networks* 12.1, pp. 145 –151. ISSN: 0893-6080.

Rodionov, Alexey S. and Hyunseung Choo (2004). "On Generating Random Network Structures: Connected Graphs". In: *Information Networking. Networking Technologies for Broadband and Mobile Networks: International Conference ICOIN 2004, Busan, Korea, February 18-20, 2004. Revised Selected Papers*. Ed. by Hyun-Kook Kahng and Shigeki Goto. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 483–491. ISBN: 978-3-540-25978-7.

Ruder, Sebastian (2016). "An overview of gradient descent optimization algorithms". In: *CoRR* abs/1609.04747.

Siegelmann, H.T. and E.D. Sontag (1995). "On the Computational Power of Neural Nets". In: *Journal of Computer and System Sciences* 50.1, pp. 132–150.

Srivastava, Rupesh Kumar, Klaus Greff, and Jürgen Schmidhuber (2015). "Training Very Deep Networks". In: *CoRR* abs/1507.06228.

Sutton, Richard S. (1986). "Two Problems with Backpropagation and Other Steepest-Descent Learning Procedures for Networks". In: *Proceedings of the Eighth Annual Conference of the Cognitive Science Society*. Hillsdale, NJ: Erlbaum.

Thorup, Mikkel (1999). "Undirected Single-source Shortest Paths with Positive Integer Weights in Linear Time". In: *J. ACM* 46.3, pp. 362–394. ISSN: 0004-5411.

Turing, A. M. (1937). "On Computable Numbers, with an Application to the Entscheidungsproblem". In: *Proceedings of the London Mathematical Society* s2-42.1, pp. 230–265.

Vinyals, Oriol, Meire Fortunato, and Navdeep Jaitly (2015). "Pointer Networks". In: *Advances in Neural Information Processing Systems 28*. Ed. by C. Cortes et al. Curran Associates, Inc., pp. 2692–2700.

Xu, Kelvin et al. (2015). "Show, Attend and Tell: Neural Image Caption Generation with Visual Attention". In: *CoRR* abs/1502.03044.

Zhang, G.P. (2000). "Neural networks for classification: a survey". In: *IEEE Transactions on Systems, Man and Cybernetics, Part C (Applications and Reviews)* 30.4, pp. 451–462.

Zheng, Alice (2015). *Evaluating Machine Learning Models*. O'Reilly Media. URL: `https://www.oreilly.com/ideas/evaluating-machine-learning-models` (visited on 05/08/2017).

Appendix A

Production Environment

This appendix describes the software versions and hardware used in the development and evaluation of the this dissertation.

The models were programmed initially using TensorFlow r0.11, but were later updated to support version r1.0. All development and testing was performed using Ubuntu 16.04 and Python 3.5.

A local machine was used for writing code and testing smaller models. A set of eight separate servers instances was used for training and testing the full models, which required more memory than the local machine contained. Two of the eight instances were provided by Trinity College Dublin's OpenNebula system. The remaining six instances were m3.2xlarge instances provided by Amazon Web Services (AWS). The hardware specifications of the local machine and each group of instances are given in Table A.1.

Provider	Architecture	CPU	Clock Speed	RAM	vCPU/CPU Cores
TCD	Intel Xeon	CPU X5650	2.67GHz	32GB	4
AWS	Intel Xeon	E5-2670 v2 (Ivy Bridge)	2.5GHz	30GB	8
Personal	Intel Core i7	i7-3630QM	2.4GHz	8GB	4

TABLE A.1: The hardware specifications of the instances used for training and testing.

As described in Section 4.1.2, the hyper-parameter optimisation process used is highly parallelisable. To take advantage of this, training was executed in batches concurrently on all eight servers. Code was distributed to the servers using a Git repository, pushed to only by the local development machine.

Appendix B

Hyper-Parameter Search Results

This appendix presents the full set of results and corresponding hyper-parameter settings from the 60 iterations of random hyper-parameter search performed. Table B.1 shows the results of the neural Turing machine hyper-parameter search. Table B.2 shows the results of the long short-term memory hyper-parameter search. The results are ordered by ascending error rate.

There is a known issue with the neural Turing machine TensorFlow implementation that the work in this dissertation uses. This causes the loss value for the network to go to NaN (Python syntax for *not a number*) for some hyper-parameter configurations. The results marked with "n" in the error rate column are configurations that resulted in NaN loss.

Error Rate (%)	ℓ_2	LR	Momentum	Decay	Hidden Units	Layers	Memory Dimension
96.48	0	1.6e-05	0.2582	0.9246	222.0	3	128
97.15	0	1.0e-5	0.3068	0.8409	138.0	3	128
97.88	4.93e-06	0.002504	0.011	0.0727	114.0	2	128
98.21	3.553e-05	0.000431	0.0806	0.4295	154.0	3	128
99.29	1.208e-05	0.002525	0.0825	0.0258	309.0	2	256
99.3	4.815e-05	0.001133	0.0925	0.6823	116.0	1	128
99.32	1.62e-06	0.001014	0.0031	0.0171	363.0	3	128
99.48	0	1.617676	0.2057	0.4159	307.0	3	128
99.61	8.684e-05	0.001043	0.7640	0.0039	264.0	3	128
99.66	0	0.003189	0.0012	0.0782	271.0	3	128
99.73	0	0.009569	0.0041	0.1883	111.0	2	128
99.74	1.515e-05	0.007087	0.0012	0.7477	181.0	2	128
99.76	0	0.613627	0.4184	0.0521	460.0	1	128
99.76	2.4e-07	0.001290	0.9554	0.1622	207.0	1	128
99.77	0	0.002783	0.8543	0.0013	121.0	3	128
99.77	1.5e-07	0.485567	0.0135	0.0029	548.0	3	128
99.78	0	0.001284	0.7552	0.01701	183.0	3	128
99.78	0	0.003758	0.0128	0.02015	388.0	3	128
99.78	2.225e-05	0.001360	0.0162	0.992	242.0	3	128
99.79	3.4e-07	0.229341	0.0075	0.01663	104.0	3	128
99.79	0	0.163570	0.027	0.0141	289.0	2	128
99.79	0	0.171580	0.0045	0.0075	149.0	2	128
99.79	0	0.052307	0.0063	0.123	177.0	3	128
99.80	0	0.095483	0.0185	0.3049	168.0	3	128
99.80	0	0.038765	0.0251	0.0515	496.0	3	128
99.80	2.63e-06	0.074532	0.1223	0.0028	129.0	3	128
99.80	1.213e-05	0.005220	0.7647	0.574	367.0	2	128
99.80	0	0.035474	0.0072	0.2096	469.0	2	128
99.80	8.55e-06	0.008163	0.4446	0.0251	508.0	2	128
99.81	3e-07	0.010164	0.6344	0.0451	300.0	3	128
99.83	5.953e-05	0.000177	0.5634	0.5534	132.0	1	256
99.84	7.638e-05	0.000688	0.0069	0.0091	147.0	1	128
99.86	0	0.378308	0.0945	0.0099	234.0	3	128
100	0	0.233577	0.0488	0.3156	413.0	3	128
100	0	0.695759	0.0034	0.0051	105.0	3	128
100	0	0.000219	0.964	0.005	370.0	3	128
n	3.297e-05	0.020638	0.7576	0.1961	305.0	1	256
n	0	0.237963	0.9383	0.0969	203.0	1	256
n	1.213e-05	0.005220	0.7647	0.5740	367.0	2	128
n	1.85e-06	0.030341	0.0545	0.0896	206.0	1	128
n	0	0.025219	0.0036	0.0049	418.0	2	256
n	9.7e-07	0.207107	0.0874	0.7650	164.0	1	128
n	5.50e-06	0.015848	0.0086	0.0023	307.0	2	128
n	3.57e-06	0.002161	0.0045	0.0017	226.0	1	256
n	0	0.009015	0.2088	0.0024	116.0	2	128
n	0	0.002480	0.0021	0.0305	566.0	1	128
n	7.592e-05	0.016454	0.0052	0.1209	107.0	2	256
n	1.4e-07	0.136737	0.0843	0.0761	451.0	2	256
n	4.2e-07	0.026760	0.0251	0.0266	209.0	1	256
n	8.9e-07	0.048027	0.0213	0.0024	134.0	2	128
n	0	0.010874	0.0086	0.0117	252.0	1	128
n	1.5e-07	0.000262	0.0161	0.0016	281.0	1	128
n	4.548e-05	0.000378	0.0067	0.2481	324.0	3	128
n	0	1.024091	0.3068	0.8409	138.0	3	128
n	9.688e-05	0.000527	0.1278	0.0152	449.0	1	128
n	3.053e-05	0.000144	0.3489	0.8387	228.0	2	256
n	2.7e-07	0.000951	0.5397	0.6813	225.0	2	256
n	2.9e-07	0.000375	0.0571	0.6026	109.0	2	256
n	1.128e-05	0.000191	0.6144	0.7518	301.0	1	256
n	3.6e-07	0.014120	0.2281	0.021	103.0	1	256

TABLE B.1: Neural Turing machine hyper-parameter search results.

Error Rate (%)	ℓ_2	LR	Momentum	Decay	Hidden	Layers
98.2	1.29e-06	0.002961	0.4594	0.2516	163.0	2
98.47	0	0.001428	0.017	0.5667	411.0	2
99.38	3.392e-05	0.000914	0.2541	0.3717	170.0	3
99.39	1.4e-07	0.0033	0.0021	0.0093	150.0	2
99.51	0	0.002267	0.4257	0.1471	122.0	1
99.51	0	0.000493	0.0995	0.5090	101.0	3
99.51	0	0.000494	0.8567	0.2798	160.0	3
99.57	0	0.007783	0.0017	0.0043	167.0	1
99.66	0	0.008507	0.0085	0.1183	108.0	3
99.69	0	0.002712	0.3652	0.3023	142.0	2
99.71	1.0e-07	0.008508	0.1071	0.2114	289.0	1
99.73	0	0.000871	0.8311	0.5009	289.0	3
99.73	1.1e-07	1.1e-05	0.0953	0.4795	104.0	2
99.73	9.630e-05	0.004599	0.2267	0.1855	294.0	2
99.74	0	0.063860	0.0659	0.0015	103.0	2
99.74	0	0.001425	0.0462	0.3153	344.0	1
99.74	1.22e-06	3.9e-05	0.1599	0.0897	297.0	3
99.76	3.14e-06	0.045055	0.0315	0.6191	551.0	3
99.76	0	0.246500	0.4421	0.0068	359.0	3
99.76	0	0.055512	0.4611	0.0059	436.0	2
99.76	1.09e-05	0.693108	0.0392	0.6101	209.0	2
99.76	0	0.132101	0.6338	0.0383	115.0	2
99.76	0	0.402081	0.0073	0.1764	384.0	2
99.76	0	0.333717	0.0341	0.0036	442.0	2
99.76	0	0.012658	0.0023	0.0017	221.0	1
99.76	0	0.005906	0.1831	0.0903	433.0	3
99.76	1.45e-06	0.892879	0.4716	0.3604	364.0	3
99.77	3.1e-07	0.600767	0.0168	0.2209	159.0	1
99.77	0	0.000208	0.9361	0.9697	282.0	3
99.77	0	0.020555	0.0806	0.2704	328.0	3
99.77	0	0.172471	0.3899	0.1668	412.0	2
99.77	1.568e-05	0.006719	0.0268	0.0038	231.0	2
99.77	1.4e-07	0.019442	0.0238	0.0018	337.0	2
99.77	1.19e-06	0.016317	0.0132	0.1291	110.0	3
99.77	1.4e-07	0.912625	0.0073	0.0037	503.0	3
99.77	0	0.011477	0.0152	0.3812	382.0	3
99.78	0	0.023196	0.0435	0.0107	391.0	2
99.78	0	0.006998	0.2413	0.0105	202.0	3
99.79	9.2e-07	0.073483	0.1070	0.0051	106.0	1
99.79	0	0.337348	0.0033	0.0018	107.0	2
99.79	1.85e-06	0.004208	0.8881	0.0333	100.0	2
99.79	0	0.009819	0.0358	0.0023	210.0	2
99.8	1.403e-05	0.005758	0.0012	0.0021	237.0	3
99.8	0	0.346232	0.0022	0.1319	122.0	1
99.8	0	0.003622	0.0434	0.6474	381.0	3
99.8	5.2e-07	0.006756	0.0154	0.0047	142.0	2
99.81	3.1e-07	0.051436	0.1203	0.5355	579.0	1
99.82	1.0e-07	0.000388	0.0525	0.3085	523.0	1
99.88	0	0.000378	0.3007	0.0477	406.0	3
99.89	0	3.0e-05	0.4631	0.4924	556.0	2
99.89	1.79e-06	7.5e-05	0.0412	0.6536	573.0	2
99.91	3.0e-07	8.1e-05	0.8149	0.7902	188.0	3
99.94	0	2.0e-05	0.5832	0.4997	473.0	1
99.94	0	1.6e-05	0.9003	0.3825	317.0	1
99.98	0	0.000215	0.8860	0.1574	262.0	2
99.99	5.51e-06	3.2e-05	0.2524	0.3894	203.0	1
99.99	0	0.000452	0.8267	0.9071	234.0	1
100	0	2.3e-05	0.9479	0.8934	359.0	2
100	1.5e-07	6.0e-05	0.4294	0.7570	217.0	2
100	0	0.000351	0.4709	0.4190	348.0	2

TABLE B.2: Long short-term memory hyper-parameter search results.