INSTITUTO POLITÉCNICO NACIONAL
CENTRO DE INVESTIGACIÓN EN COMPUTACIÓN

TESIS

"Neural Networks with External Dynamic Memory to Solve Algorithmic Problems"

PARA OBTENER EL GRADO DE:
"Maestría en Ciencias de la Computación"

PRESENTA
Ing. Uriel Corona Bermúdez

DIRECTORES DE TESIS
Dr. Ricardo Menchaca Méndez Dr. Rolando Menchaca Méndez

Ciudad de México noviembre de 2018

SIP-14 bis

INSTITUTO POLITÉCNICO NACIONAL

SECRETARÍA DE INVESTIGACIÓN Y POSGRADO

ACTA DE REVISIÓN DE TESIS

En la Ciudad de México siendo las 12:00 horas del día 02 del mes de julio de 2018 se reunieron los miembros de la Comisión Revisora de la Tesis, designada por el Colegio de Profesores de Estudios de Posgrado e Investigación del:

Centro de Investigación en Computación

para examinar la tesis titulada:

"Neural Networks with External Dynamic Memory to Solve Algorithmic Problems"

Presentada por el alumno:

CORONA	**BERMÚDEZ**	**URIEL**
Apellido paterno	Apellido materno	Nombre(s)

Con registro:

B	1	6	0	6	3	0

aspirante de: **MAESTRÍA EN CIENCIAS DE LA COMPUTACIÓN**

Después de intercambiar opiniones los miembros de la Comisión manifestaron ***APROBAR LA TESIS***, en virtud de que satisface los requisitos señalados por las disposiciones reglamentarias vigentes.

LA COMISIÓN REVISORA

Directores de Tesis

[illegible]	[illegible]
Dr. Ricardo Menchaca Méndez	Dr. Rolando Menchaca Méndez
[illegible]	[illegible]
Dr. Juan Carlos Chimal Eguía	Dr. Francisco Hiram Calvo Castro
[illegible]	[illegible]
Dr. Erik Zamora Gómez	Dr. Mario Eduardo Rivero Ángeles

ESTADOS UNIDOS MEXICANOS

PRESIDENTE DEL COLEGIO DE PROFESORES

[illegible]

Dr. Marco Antonio Ramírez Salinas

INSTITUTO POLITECNICO NACIONAL
CENTRO DE INVESTIGACION
EN COMPUTACION
DIRECCION

INSTITUTO POLITÉECNICO NACIONAL

SECRETARÍA DE INVESTIGACIÓN Y POSGRADO

CARTA CESIÓN DE DERECHOS

En la Ciudad de México el día 22 del mes noviembre del año 2018, el (la) que suscribe Uriel Corona Bermúdez alumno (a) del Programa de Maestría en Ciencias de la Computación con número de registro B160630, adscrito a Centro de Investigación en Computación, manifiesta que es autor (a) intelectual del presente trabajo de Tesis bajo la dirección de Ricardo Menchaca Méndez y Rolando Menchaca Méndez y cede los derechos del trabajo intitulado Neural Networks with External Dynamic Memory to Solve Algorithmic Problems, al Instituto Politécnico Nacional para su difusión, con fines académicos y de investigación.

Los usuarios de la información no deben reproducir el contenido textual, gráficas o datos del trabajo sin el permiso expreso del autor y/o director del trabajo. Este puede ser obtenido escribiendo a la siguiente dirección urielcoro@outlook.com. Si el permiso se otorga, el usuario deberá dar el agradecimiento correspondiente y citar la fuente del mismo.

[illegible]

Uriel Corona Bermúdez

Nombre y firma

Resumen

Implementamos y desarrollamos una Computadora Neural Diferenciable, estudiamos su arquitectura, características y capacidades. Se realizaron pruebas mediante un conjunto de tareas algorítmicas básicas y se llevó a cabo una comparación con su modelo predecesor; La Máquina de Turing Neuronal. Se proporciona un marco teórico extenso sobre el funcionamiento de la Computadora Neural Diferenciable. Los resultados preeliminares muestran el cómo la Computadora Neural Diferenciable es capáz inferir algoritmos simples al igual que las Máquinas de Turing Neuronales pero de manera más eficaz y eficiente, y también cómo este modelo tiene la capacidad de generalizar resultados sin la necesidad de entrenarlo nuevamente.

Keywords: Redes neuronales, Computadora Neural Diferenciable, Aprendizaje de Máquina

Abstract

We developed a Differentiable Neural Computer and studied its architecture, characteristics, and capabilities. We tested the DNC through a set of basic tasks and made a benchmark versus its predecessor; The Neural Turing Machine. Also, we provide an extensive theoretical framework about how the Differentiable Neural Computer works. Preliminary results show that Differentiable Neural Computers can infer simple algorithms like Neural Turing Machines but in a more efficient and efficacy way, and also has the capability of generalizing results without retraining the model.

Keywords: Neural networks, Differentiable Neural Computer, Machine learning

Acknowledgments

I would like to thank Dr. Ricardo Mechaca and Dr. Rolando Menchaca for their help through the master's degree, their help with this research, and for all the provided knowledge. To the ”Centro de Investigación en Cómputo” and the ”Instituto Politécnico Nacional”, my alma mater. To the ”Consejo Nacional de Ciencia y Tecnología” for making this possible.

Dedication

For my mother, my family, and my friends who always are there, supporting me with their words, companionship, and love.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Introduction

Based on traditional computer architecture, in 2014, Alex Graves et. al. published an article called ”Neural Turing Machines” [1], where they introduced a neural network-based machine architecture and how it works when solving complex and well-structured problems. The main idea was extending the current neural network capability by an external memory expansion, this should be useful when storing information in long-time periods. Alex Graves said that DNC’s architecture is similar to Von Neumann classical computer architecture, with the main difference that Neural Turing Machine is fully differentiable, which implies that it can be trained with any current neural network training method.

The Neural Turing Machine is, in general, a recurrent neural network. The main Neural Turing Machine architecture component is the controller. The controller is a neural network that uses the previous and current time-step information for producing an information vector, that posteriorly will be used for calculating the current time-step output, and additionally, will be used for accomplishing the memory interaction by generating the reading parameters and the writing parameters. What is pretended with the Alex Graves’ investigation, is to demonstrate that a neural network can learn how solving algorithmic tasks using examples as structured inputs, like Zaremba and Sutskever [2] did, with the difference that Zaremba and Sutskever used Long-Short Term Memories and Alex Graves proposed his own architecture. With this in mind, Alex Graves added an additional memory to a neural network, with the confidence that it could work as a traditional computer memory unit, that is, for storing variables through an algorithm execution, what could be useful for maintaining long-time information and its dependencies.

In the original article, the Neural Turing Machine’s functionality isn’t good explained [1], but in 2017, Omar Gutierrez [3] graded with a thesis where he proposed his own NTM’s components. Omar Gutierrez’s components worked with two specific tasks proposed originally by Alex Graves [1].

In 2016, Alex Graves et.al. published another article called ”Hybrid computing using a neural network with dynamic external memory” [4], where he showed the Neural Turing Machine’s evolution;

the Differentiable Neural Computer (DNC). DNC's model was based on the same Neural Turing Machine's principle: Extends a neural network with an external memory. The main differences between Neural Turing Machine and Differentiable Neural Computer are, the memory addressing access mechanisms, a secondary memory used when sequential memory readings, and memory usage efficiency. DNC's changes helped for accomplishing better results than previous models.

Differentiable Neural Computer showed its capability for understanding structured data, and for learning that data usage. Its performance was good when solving synthetic question answering, graphs paths, and puzzle experiments when using small instances. They argue that for solving bigger problems, is just needed to modify the external memory size. The input data was structured, which indicates the model's capability when understanding structured data. The memory usage and the information interpretation is shown in the article videos.

Given that neural networks compose the state of art in many topics, is important to know all DNC's mechanisms for attacking those problems where the most studied models don't work. The DNC's benefit is the full differentiation, it means, that the model can be trained using backpropagation algorithm, only by feeding it with examples(in supervised learning), so, for the proposed Alex Graves' tasks the model can learn how solving them by itself. Now, Differentiable Neural Computer was only tested with simple algorithms, but getting the DNC's mechanisms knowledge will help us to understand it better, and adapt it for solving new problems.

1.2 Problem

In traditional computers, the computing is done by the processor, using a directional memory, this allows storing information as variables during the execution. Variables have a very important role when collecting and storing information: preserves data in a registers table, storing previous calculus information for its future use, and storing individual information that must be globally available through the application [5]. By contrast, with a computer, in neural networks, that information is mixed in neural network's weights and in the neuron functionality, so, while the task memory demand increase, neural networks cannot store new information dynamically, neither the learning algorithms can act independently of generated tasks values. Despite the utility of neural networks when learning tasks, these are limited in variables representation, structured data understanding, and storing information over long time periods [4].

In order to attack this problem, Alex Graves introduced a model named Differentiable Neural Computer [4], that is in general, a neural network with logical mechanisms and an extended external memory, which helps when solving harder problems than a classical neural network. The extended DNC's memory helps the model for long time periods information storing(such as a computer does), which increases the data 's model storing capability, useful for interpreting structured inputs and examples-based algorithmic tasks learning. It's important to understand the DNC, its mechanisms, how they work and test its capabilities, with the aim of attacking different problems varieties that current neural network models can’t.

1.3 Goals

Understand and explain the ideas behind Differentiable Neural Computers. In particular, we want to study the way in which DNC’s interact with an external memory unit.

1.3.1 Subgoals

- Study the main Neural Turing Machine’s architectural components.
- Implement the Neural Turing Machine.
- Use the Neural Turing Machine for testing a set of basic tasks, including the ”associative recall” and the ”copy” tasks.
- Study the main Differentiable Neural Computers’ architectural components.
- Implement the Differentiable Neural Computer.
- Use the Differentiable Neural Computer for testing a set of basic tasks, including the ”associative recall” and the ”copy” tasks.
- Compare Neural Turing Machine’s and Differentiable Neural Computer’s performance when training and testing the basic tasks.
- Use the Differentiable Neural Computer in a basic task for validating generalization.

1.4 Justification

Neural networks are important in sensorial processing, sequential learning, and reinforcement learning fields, but despite its potential, they have limited capability when storing useful information and data interpretation in long time-scales [4].

Algorithmic solutions are used in many tasks, despite whether they are efficient or not, but in order to solve our problem, we must use them. NP problems lay here. NP is the set of problems where exists an efficient certifier [6], that means, they can be efficiently checked independently whether they can be solved efficiently or not. The most common approaches for tackling NP-hard problems are, using approximation algorithms and general meta-heuristics. While approximation algorithms provide rigorous approximation factors, by exploiting relevant problem structures, meta-heuristics move in the problem solutions space by following analogies from nature or other places (like neural networks). These techniques have been evaluated experimentally and have demonstrated their usefulness for solving practical problems [7].

Given that neural networks are meta-heuristics models, they are useful for solving NP problems and other problems' variety because of their approximation solutions capabilities. Also, neural networks are an important study and research topic, but they don't have the representing structured data and long timescales storing capability. This problem was treated by Alex Graves when he introduced Neural Turing Machines and two years later its evolution: Differentiable Neural Computer. The limitation of neural networks memory was tackled with an external memory, which also includes differentiable mechanisms for managing it. These features gave a powerful differentiable model, and in consequence trainable [4]. Given the metaheuristics models importance in many areas, especially neural networks importance, study Alex Graves' architecture becomes important in order to attack tasks that requires algorithmic methods, and in consequence, an extense memory.

So, could a DNC solve problems that current methods cannot? In order to answer this question, we will get you the knowledge that Differentiable Neural Computer involves and will evaluate whether it could become a metaheuristics solutions powerful and a principal tool.

1.5 Contributions

The contributions that are pretended to cover with this study are:

- A Differentiable Neural Computer detailed description and how it works based on existent information and experimental results, going from general to particular aspects.
- A task test set of different sizes used when evaluating the model's performance.
- An own Differentiable Neural Computer implementation, free and well explained, with the aim of strengthen the knowledge for those interested people on use, modify or study it.

1.6 Organization

This thesis is separated into six chapters. The first chapter contains the abstract, goals, and justification. It is the first Differentiable Neural Computer approach.

In the second chapter, the theoretical framework can be found. There we explain the necessary theoretical bases.

The third chapter was written for explaining all about Differentiable Neural Computer: model, mechanisms, memory functionality, etc. All information is divided according to the explained functionality, like, memory writing, memory reading, output generation, etc.

The fourth chapter shows how DNC was developed. To get or explore the full code, you can access the next repository `https://github.com/cobu93/DNC`.

In the fifth chapter, all tested tasks are described. Also, the comparison between NTM's and DNC's tests are explained, and the results are exposed. Also, some DNC's features are tested.

Finally, the last chapter includes the conclusion about this thesis, the final contributions, and future work.

Chapter 2

Theoretical Framework

2.1 Neural Networks

The neurocomputing beginning is considered with McCulloch y Pitts' published article [8] in 1943. There they showed how the neurons work in function of arithmetical and logical functions. They introduced the term ”neuron net” and found that all networks work under the same principle. Combined with specific learning Hebb's theory [9], used in neurons synapses, we can explain the neural

Artificial neural networks are computing models, biological neural networks inspired. These artificial neural networks are designed for learning such as biological models do; with reverberatory activity persistence or repetition [9], that means, that if we provide enough certain activity's information to a neural network, it will learn how to develop that task.

2.1.1 Feedforward network

Basic neural network models are called feedforward neural networks, or multilayer perceptrons(MLPs), are the quintessential deep learning models. The feedforward network's goal is approximating some function f^*. A feedforward network defines a mapping $y = f(\boldsymbol{x}; \boldsymbol{w})$ and learns the parameters values $\boldsymbol{w}$, what results in the best approximation function. These models are called feedforward because the information flows through the function being evaluated from $\boldsymbol{x}$, after, the intermediate computations used for defining f, and finally, to the output y. There are no feedback connections to itself [10].

Feedforward neural networks are called networks because they are typically represented by composing together many different functions. The model is associated with a directed acyclic graph, describing how the functions are composed together.

Specifically, they can be described as a functional transformations combination [11]. First, we construct M linear inputs variables combinations $\boldsymbol{x} = (x_1, \cdots, x_D)^T$ as next:

$$a_j^{(1)} = \sum_{i=1}^{D} w_{ji}^{(1)} x_i + w_{j0}^{(1)}$$

where $j = 1, \cdots, M$, and the superscript (1) indicates the first layer network parameters. We shall refer to w_{ji} parameters as weights and w_{j0} ones as biases. Each one is then transformed using a differentiable, and usually, nonlinear activation function $f^{(1)}(\cdot)$ to get

$$z_j^{(1)} = f^{(1)}\left(a_j^{(1)}\right)$$

Where $\boldsymbol{z}^{(1)} = (z_1, \cdots, z_M)$ is used as the next layer input, which in context is called hidden layer, and output is used for the next layer and so on.

So, it is written as a generalized function

$$a_k^{(l)} = \sum_{j=1}^{N} w_{kj}^{(l)} z_j^{(l)} + w_{k0}^{(l)}$$

$$z_k^{(l)} = f^{(l)}\left(a_k^{(l)}\right)$$

Figure 2.1 shows a feedforward neural network graphic model, just as we said, it has a directed acyclic graph representation.

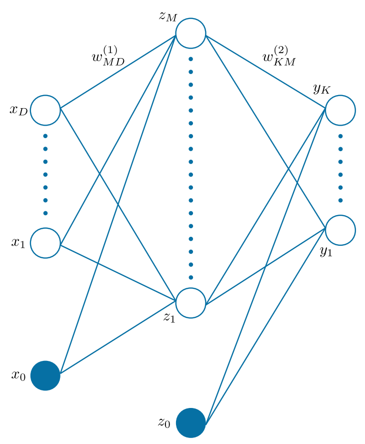

Figure 2.1: A feedforward neural network graphical model.

2.1.2 Activation functions

Feedforward networks have introduced the hidden layer concept, and this requires to choose the activation functions that will be used for computing the hidden layer values [10]. Below, some activation functions are introduced. These functions will be used through all document.

2.1.2.1 Hyperbolic tangent

The hyperbolic tangent definition is

$$tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$$

$$\begin{array}{cc} \text{Domain:} & \mathbb{R} \\ \text{Image:} & (-1, 1) \end{array}$$

Figure 2.2 shows a function plot

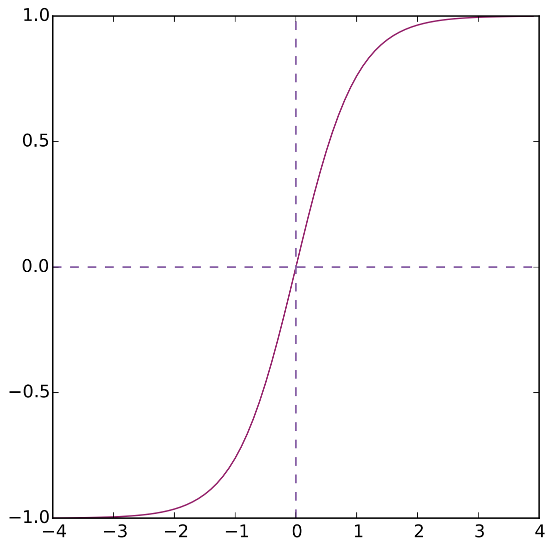

Figure 2.2: The hyperbolic tangent plot.

2.1.2.2 Oneplus

The oneplus function definition is

$$oneplus(x) = 1 + log(1 + e^x)$$

$$\begin{array}{rc} \text{Domain:} & \mathbb{R} \\ \text{Image:} & (1, \infty) \end{array}$$

The figure 2.3 shows a function plot

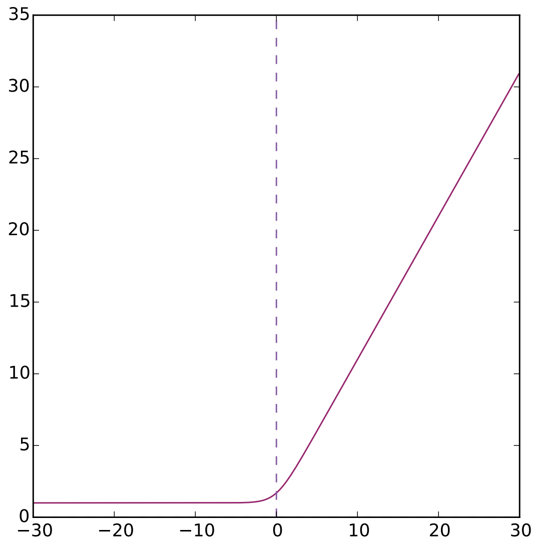

Figure 2.3: The oneplus function plot.

2.1.2.3 Sigmoid

The sigmoid function definition is

$$\sigma(x) = \frac{1}{1 + e^{-x}}$$

$$\begin{array}{rc} \text{Domain:} & \mathbb{R} \\ \text{Image:} & (0, 1) \end{array}$$

The figure 2.4 shows a function plot

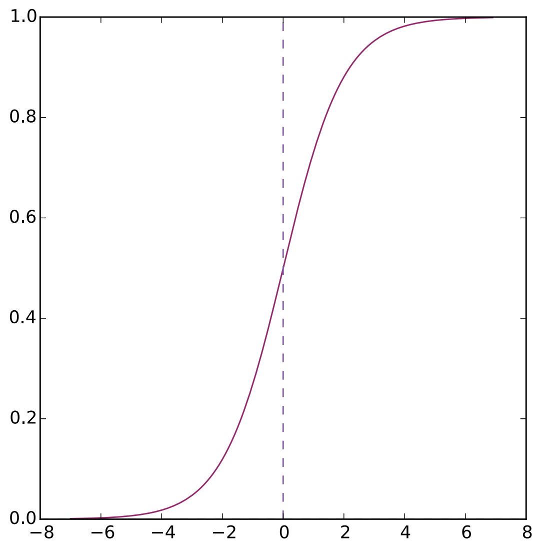

Figure 2.4: The sigmoid function plot.

2.1.2.4 Softplus

The softplus definition function is

$$sofplus(x) = log(1 + e^x)$$

$$\begin{array}{cc} \text{Domain:} & \mathbb{R} \\ \text{Image:} & (0, \infty) \end{array}$$

The figure 2.5 shows a function plot

The softmax function can be described as a probability function, where the output is a normalized vector.

$$softmax(\boldsymbol{x})_j = \frac{e^{x_j}}{\sum_{k=1}^{K} e^{x_k}}$$

$$\begin{array}{cc} \text{Domain:} & \{\boldsymbol{x} = (x_1, \cdots, x_K) \in \mathbb{R}^K\} \\ \text{Image:} & \{\boldsymbol{z} = (z_1, \cdots, z_K) \in \mathbb{R}^K : \sum_{j=1}^{K} z_j = 1, z_j \geq 0, \forall j\} \end{array}$$

In this document, we will use next softmax notation

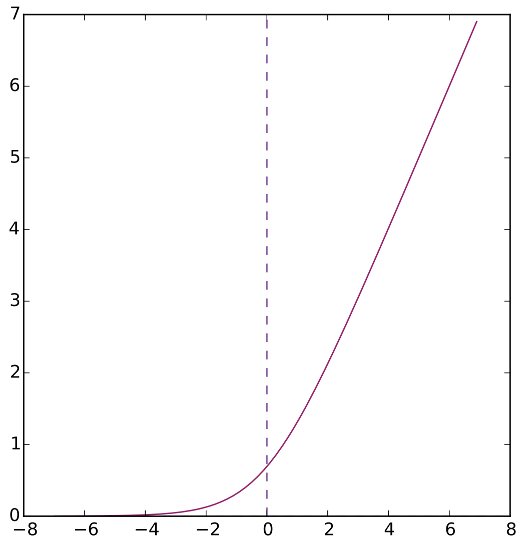

Figure 2.5: The softplus function plot.

$$\mathcal{S}_N = \left\{ \boldsymbol{z} \in \mathbb{R}^N : \boldsymbol{z}_i \in [0,1], \sum_{i=1}^{N} \boldsymbol{z}_i = 1 \right\}$$

2.1.3 Recurrent neural networks

The recurrent neural networks(RNN) are a neural networks family used for sequential processing data [10]. A recurrent neural network looks like a feedforward neural network, with the exception that they contain backward pointing connections [12].

The figure 2.6 shows an RNN basic architecture. RNN backward connections can be appreciated in the left part of figure 2.6. Also, a recurrent neural network can be imagined as a feedforward network unrolled it over the time, just as in the right part of the figure.

The RNN's parameters and activation functions are shared along the time, that is, for all inputs, the same updating rule is applied, and depending on fed sequential data(past and present data), the weights will be updating. This shared information and the sequential data use allows maintaining dependencies over some time-steps and give us information about future behavior depending on past behavior.

Given that the current output depends on present and past input data, and updating rule, we can express current output as

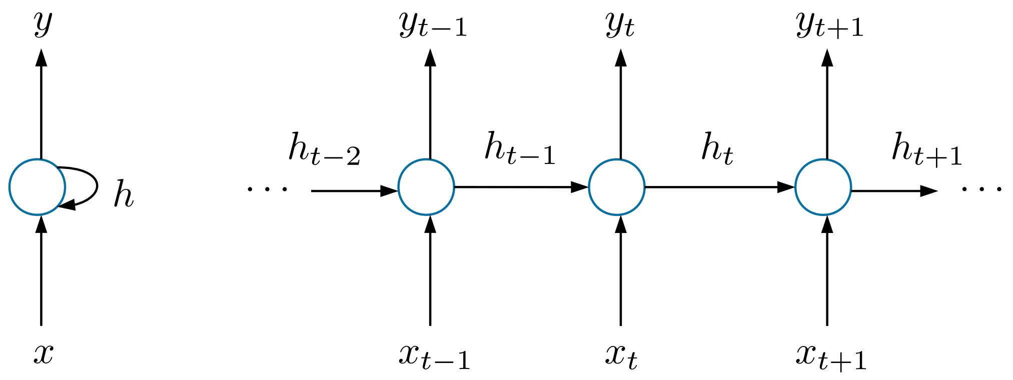

Figure 2.6: The Recurrent Neural Network architecture.

$$\boldsymbol{h}_t = g(\boldsymbol{h}_{t-1}, \boldsymbol{x}_t; \boldsymbol{\theta})$$

where all learnable parameters are expressed by $\boldsymbol{\theta}$.

The RNN's simple output can be expressed by the last-time output, and the last time output depends on the previous one and so on, so, that behavior equation is expressed below

$$\boldsymbol{h}_t = \boldsymbol{y}_t = f(W_x\boldsymbol{x}_t + W_h\boldsymbol{h}_{t-1} + \boldsymbol{b})$$

More complex models can be designed, despite that the model shown here is a very simple RNN model, for example, making different $\boldsymbol{h}_t$ from $\boldsymbol{y}_t$, and applying one or more stacked activation functions, that is, stacking more cells over the recurrent cell output, we can get results like shown in figure 2.7

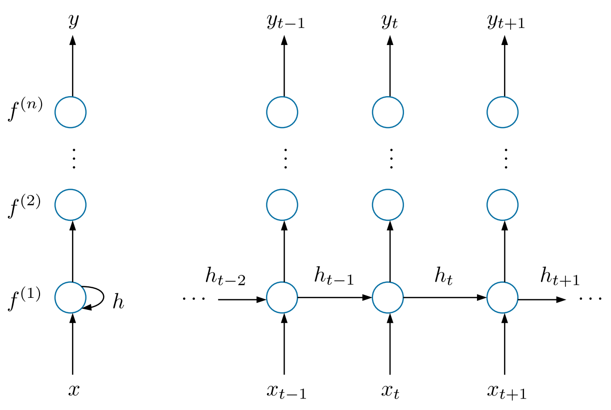

Figure 2.7: The Recurrent Neural Network: $\boldsymbol{y}_t \neq \boldsymbol{h}_t$.

Its equations are different, so, for that model, we can consider these two equations

$$\boldsymbol{h}_t = f^{(1)}(W_x\boldsymbol{x}_t + W_h\boldsymbol{h}_{t-1} + \boldsymbol{b})$$

$$\boldsymbol{y}_t = f^{(n)}\left(W_{f^{(n)}}f^{(n-1)}\left(\cdots f^{(2)}(W_{f^{(2)}}\boldsymbol{h}_t + \boldsymbol{b}^{(2)})\right) + \boldsymbol{b}^{(n)}\right)$$

The backward connections can be applied in whatever layer. The output from an RNN can be obtained in different ways. The first way is to simultaneously take an inputs sequence and produce an outputs sequence(figure 2.8 top-left network). This retrieve type is useful, for example, for predicting time series such as stock prices. You feed the network with the last N days prices and the output will be the predicted ones.

Alternatively, you could feed the network with an inputs sequence, and ignore all outputs except the last one(figure 2.8 top-right network). To get a movie sentimental score, a words sequence, corresponding to a movie review, can be fed to the neural network (e.g., from -1 [hate] to $+1$ [love]).

The third method is when you feed the network with a single input, at the first time-step (and zeros for all other time steps), and recover the full output sequence (figure 2.8 bottom-left network). This is a vector-to-sequence network. The input could be an image and its caption will be the output.

Lastly, you could have a sequence-to-vector network, called an encoder, followed by a vector-to-sequence network, called a decoder (figure 2.8 bottom-right network). This method can be used for translating a sentence from one language to another. You feed the network with a sentence in one language, then the encoder converts this sentence into a single vector representation. and finally. the decoder converts this vector into another language sentence [12].

2.2 Long Short-Term Memory

The Long Short-Term Memory (LSTM) cell can be considered as a black box. It can be used as a basic RNN cell, with the exception, that it will perform much better, what means that the training will converge faster and the long-term data dependencies will be detected [12].

To talk of an LSTM as an RNN basic cell means that it can be considered as an activation function, with the difference that it is a more complex mechanism that includes multiple operations. The information is kept in this cell over a long time-step unlike a simple RNN because the information is stored only in shared weights.

The figure 2.9 shows how an LSTM cell looks like. One of the differences is that two headers are kept by this cells' type, $\boldsymbol{h}_t$ and $\boldsymbol{c}_t$, where the short-term state and long-term state are represented. The mechanism can be thought as [12]:

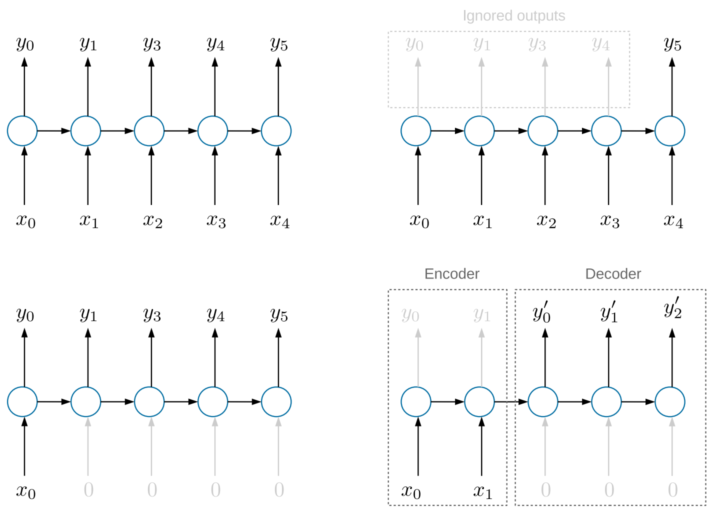

Figure 2.8: The Recurrent Neural Network outputs.

- The control of what long-term state part should be erased lay in the forget gate(f_t).
- The control of what g_t part should be added to the long-term state lay on the input gate(i_t). This is why we said that it was only ”partially stored”.
- Finally, the control of what long-term state parts should be read and outputted at this time-step is the output gate(o_t) job.

The LSTM cell equations is below

$$\begin{gathered}
\boldsymbol{i}_t = \sigma\left(W_{xi}^T \boldsymbol{x}_t + W_{hi}^T \boldsymbol{h}_{t-1} + \boldsymbol{b}_i\right) \\
\boldsymbol{f}_t = \sigma\left(W_{xf}^T \boldsymbol{x}_t + W_{hf}^T \boldsymbol{h}_{t-1} + \boldsymbol{b}_f\right) \\
\boldsymbol{o}_t = \sigma\left(W_{xo}^T \boldsymbol{x}_t + W_{ho}^T \boldsymbol{h}_{t-1} + \boldsymbol{b}_o\right) \\
\boldsymbol{g}_t = tanh\left(W_{xg}^T \boldsymbol{x}_t + W_{hg}^T \boldsymbol{h}_{t-1} + \boldsymbol{b}_g\right) \\
\boldsymbol{c}_t = \boldsymbol{f}_t \otimes \boldsymbol{c}_{t-1} + \boldsymbol{i}_t \otimes \boldsymbol{g}_t \\
\boldsymbol{y}_t = \boldsymbol{h}_t = \boldsymbol{o}_t + tanh\left(\boldsymbol{c}_t\right)
\end{gathered}$$

The figure 2.10 shows a high-level basic LSTM diagram used in an example. For get the LSTM output, we can use those simple RNN methods showed before, so, the LSTM cell is just an RNN cell modification which provides more ”memory”.

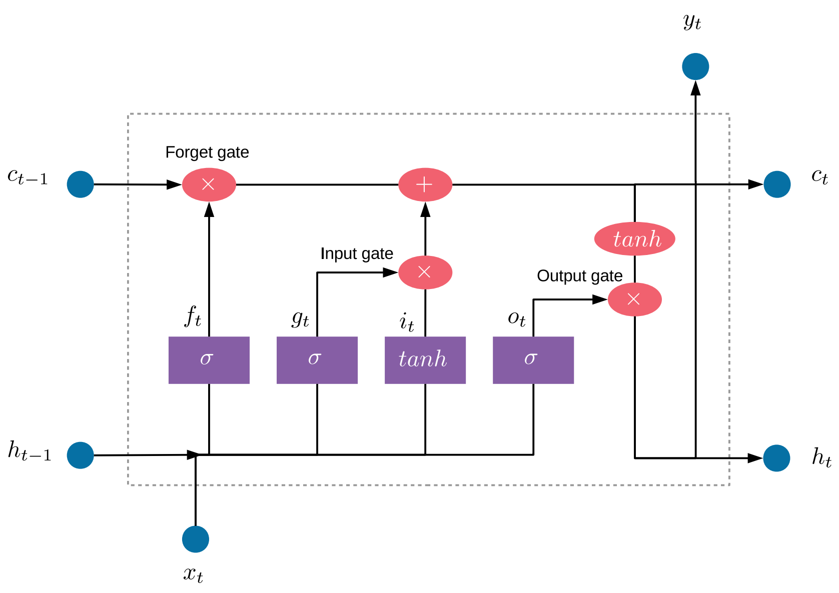

Figure 2.9: A Long Short-Term Memory cell

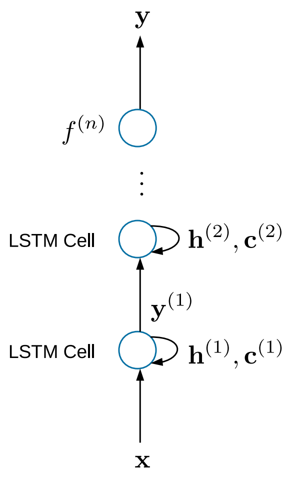
$\mathbf{y}$
$f^{(n)}$
LSTM Cell
$\mathbf{h}^{(2)}, \mathbf{c}^{(2)}$
$\mathbf{y}^{(1)}$
LSTM Cell
$\mathbf{h}^{(1)}, \mathbf{c}^{(1)}$
$\mathbf{x}$

Figure 2.10: A Long Short-Term Memory basic usage.

2.3 Neural Turing Machine

The Neural Turing Machine was the first Alex Graves proposed mode, this was in 2014 [1]. A Neural Turing Machine (NTM) is a machine learning model that includes a neural network with a joined external dynamic memory that can be accessed by differentiable addressing mechanisms, so, it can be trained. For representing and handling complex structured data, NTM can learn how managing its memory like a computer, with the main difference that it does use examples [3].

The Neural Turing Machine's principal features are:

- It is a full differentiable model.
- Can be trained using gradient descent and backpropagation algorithms.
- Instead of using deterministic methods, the NTM can learn how managing the memory.
- Merge neural network and programmable computer capabilities.
- It has finite memory.

An NTM has two basic components: A neural network that works as a controller(neural controller) and a memory(memory matrix). The neural controller interacts with the environment via the input and output vectors, but either decide where reading or writing content in the memory through attention processes.

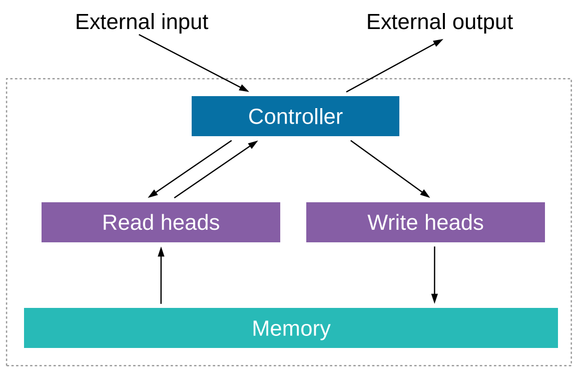

Figure 2.11: The Neural Turing Machine's architecture.

In Alex Graves' article [1] isn't well specified how Neural Turing Machine works, he only gave an overview, but, Omar Gutierrez's thesis [3] shows an option for developing it and how it possibly works.

Omar Gutierrez provided a high level diagram (figure 2.12) and proposed next elements:

- Neural controller.

- Output layer.
- Write header.
- Read header.
- Write addressing mechanism.
- Read addressing mechanism.
- Write operation.
- Read operation.
- Memory.

Taking figure 2.12 as reference, on each time step t, the proposed functionality is:

1. The neural controller receives the input $\boldsymbol{x}_t$ and the read vector $\boldsymbol{r}_t$.
2. The controller generates the vector state $\boldsymbol{h}_t$.
3. The output layer uses $\boldsymbol{h}_t$ for generating $\boldsymbol{Y}_t$.
4. The header receives $\boldsymbol{h}_t$ for generating $\hat{\boldsymbol{k}}_t$, $\hat{\beta}_t$, $\hat{g}_t$, $\hat{\boldsymbol{s}}_t$ and $\hat{\gamma}_t$ (If it is a write header, additionally generates $\hat{\boldsymbol{e}}_t$ and $\hat{\boldsymbol{a}}_t$,).
5. Different layers processes the parameters for specified rules adjusting. Finally the head emits $\boldsymbol{k}_t$, β_t, g_t, $\boldsymbol{s}_t$ and γ_t (If it is a write header additionally generates $\boldsymbol{e}_t$ and $\boldsymbol{a}_t$).
6. The addressing mechanism(reading or writing) uses they own parameters $\boldsymbol{k}_t$, β_t, g_t, $\boldsymbol{s}_t$, γ_t and some last time vectors; M_{t-1} and $\boldsymbol{w}_{t-1}^w$ if it is writing addressing, or $\boldsymbol{w}_{t-1}^r$ if it is a reading mechanism, for generating $\boldsymbol{w}_t^w$ and $\boldsymbol{w}_t^r$ respectively.
7. The writing operation takes $\boldsymbol{w}_t^w$, $\boldsymbol{e}_t$, $\boldsymbol{a}_t$ and M_{t-1} for obtaining the newmemory matrix state M_t.
8. The reading operation gets M_t and $\boldsymbol{w}_t^r$ for emitting $\boldsymbol{r}_t$

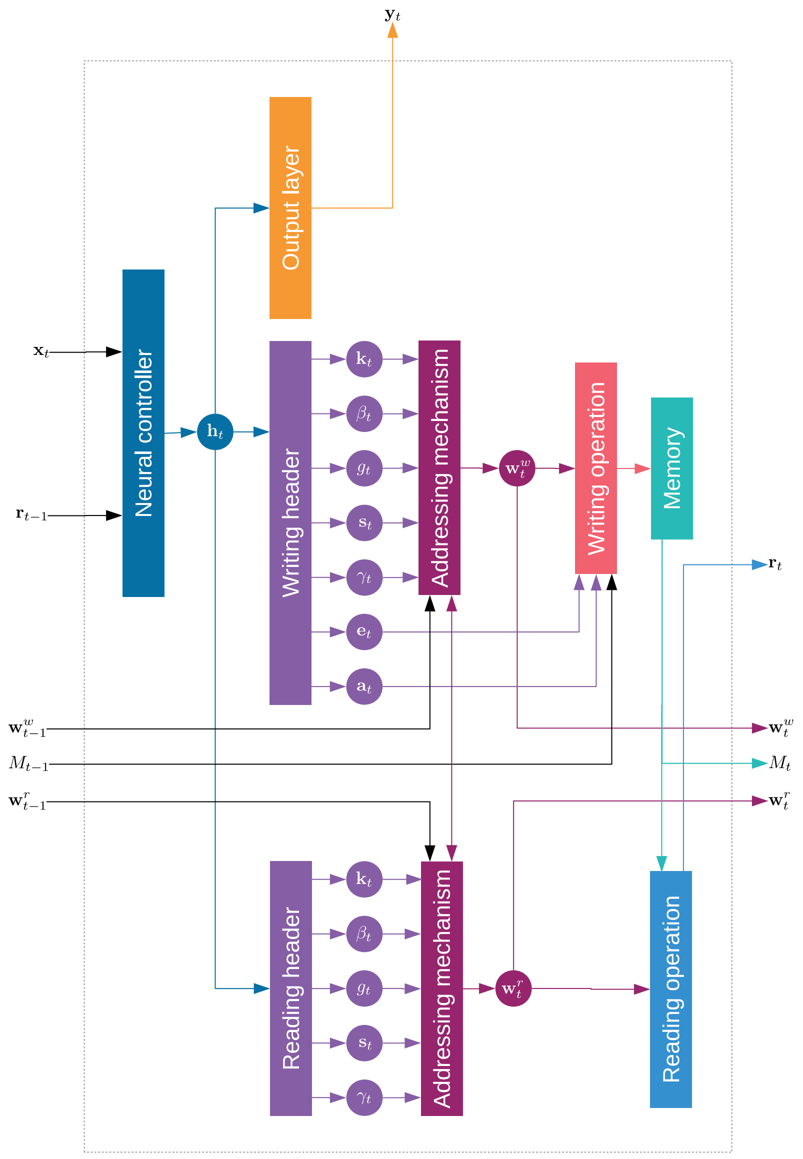

Figure 2.12: Omar Gutierrez's Neural Turing Machine diagram

Chapter 3

Differentiable Neural Computer

3.1 Introduction

In 2016, Alex Graves published an article [4] where he proposed a new architecture for neural networks, it was taken as the evolution of the Neural Turing Machine and the new architecture was named Differentiable Neural Computer (DNC), which consists of a neural network that can read from and write to an external memory matrix, analogous to a conventional computer random-access memory. Like a conventional computer, it can use its memory for representing and manipulating complex data structures, but, like a neural network, it can learn how to do so from data.

Alex Graves and his colleagues demonstrated that the DNC is able for solving multiple problems, as answering synthetic questions, find the shortest path between two specified points in a directed graph, inferring missed links in randomly generated graphs and complete puzzles. Their results demonstrate that DNCs have the capability for solving complex, and well-structured tasks, that are inaccessible for neural networks without an external readable-writeable memory.

The principal model's characteristic is that it is analogous to a classical computer, that is, the computation and the memory are modularly separated. The computation is performed by a processor, which in order to bring operands in and out of play uses an addressable memory conferring two important benefits: the use of extensible storage to write the new information and the ability to treat the contents of memory as variables. The variables are critical to an algorithm generality: To perform the same procedure on the data invariantly to the content and the algorithm only has to change the address it reads from.

In contrast to the computers, the computational and the memory resources of artificial neural networks are mixed together in the network weights and the neuron activity. This is a major liability: as the task memory demands increase, storage cannot be allocated dynamically for this networks, nor easily learn algorithms that act independently of the values realized by the task variables. The neural networks are limited in their ability to represent variables and data structures as well as to store data over long timescales without interference.

Alex Graves' model aims to combine the advantages of neural and computational processing by providing a neural network with an external memory. Is important to mention that the whole system is differentiable, and can, therefore, be trained end-to-end with gradient descent, allowing to the network to learn how to operate and organize the memory in a goal-directed manner.

3.2 Architecture

The Differentiable Neural Computer is a model that aims to couple an external memory and a neural network. The behavior of the network and the memory size are independent as long as the memory is not filled to capacity, which is why the memory is considered as external.

While we considered the external memory as a random-access memory the neural network referred to as the controller can be thought as a differentiable CPU whose operations are learned by training methods.

Whereas the conventional computers use unique addresses for accessing memory contents, the DNC uses differentiable attention mechanisms for defining distributions over the memory rows in a memory matrix. These distributions, called weightings, represent the degree to which each location is involved in a reading or writing operations.

Below, the process involved in each DNC's operation will be described for explaining completely how mechanisms were developed in for generating a functional model.

3.2.1 Controller

The controller is a neural network that is analogous to a CPU, that is, it manages the memory access, but, the main idea, is that neural network should achieve that only with examples (in supervised learning) or by the environment information(reinforcement learning).

Let N the controller, at every time-step t it receives a vector $\boldsymbol{x}_t \in \mathbb{R}^X$ from the dataset or the environment and produces an output vector $\boldsymbol{y}_t \in \mathbb{R}^Y$ that parameterizes either a predictive distribution for a target vector $\boldsymbol{z}_t \in \mathbb{R}^Y$ (supervised learning) or an action distribution (reinforcement learning). Additionally, the controller also receives a set of R read vectors $\boldsymbol{r}_{t-1}^1, \boldsymbol{r}_{t-1}^2, \cdots, \boldsymbol{r}_{t-1}^R$ where $\boldsymbol{r}_{t-1}^i \in \mathbb{R}^N$ represents the rows in the memory matrix $M \in \mathbb{R}^{N \times W}$ obtained via read-heads on the previous time-step. It then emits an interface vector ξ_t that defines its interactions with the memory at the current time-step via reading, writing and addressing operations.

For notational convenience, the read and the input vectors are concatenated for generating a single controller input vector $\boldsymbol{\chi}_t = [\boldsymbol{x}_t; \boldsymbol{r}_{t-1}^1; \cdots; \boldsymbol{r}_{t-1}^R]$. The controller can be any type of neural network as we mention later. Next equations models the Alex Graves proposed controller:

$$
\begin{aligned}
\boldsymbol{i}_t^l &= \sigma\left(W_i^l[\boldsymbol{\chi}_t;\boldsymbol{h}_{t-1}^l;\boldsymbol{h}_t^{l-1}]+\boldsymbol{b}_i^l\right)\\
\boldsymbol{f}_t^l &= \sigma\left(W_f^l[\boldsymbol{\chi}_t;\boldsymbol{h}_{t-1}^l;\boldsymbol{h}_t^{l-1}]+\boldsymbol{b}_f^l\right)\\
\boldsymbol{o}_t^l &= \sigma\left(W_o^l[\boldsymbol{\chi}_t;\boldsymbol{h}_{t-1}^l;\boldsymbol{h}_t^{l-1}]+\boldsymbol{b}_o^l\right)\\
\boldsymbol{g}_t^l &= tanh\left(W_g^l[\boldsymbol{\chi}_t;\boldsymbol{h}_{t-1}^l;\boldsymbol{h}_t^{l-1}]+\boldsymbol{b}_g^l\right)\\
\boldsymbol{c}_t^l &= \boldsymbol{f}_t^l\boldsymbol{c}_{t-1}^l+\boldsymbol{i}_t^l\boldsymbol{g}_t\\
\boldsymbol{h}_t^l &= \boldsymbol{o}_t^l + tanh\left(\boldsymbol{c}_t^l\right)
\end{aligned}
$$

Where $\boldsymbol{h}_t^0 = \boldsymbol{0}$ for all t and $\boldsymbol{h}_0^l = \boldsymbol{s}_0^l = \boldsymbol{0}$ for all l. The W indicates the learnable weights (for example, W_i^l symbolizes the weights matrix going into the layer-l input gates) and $\boldsymbol{b}$ denotes the learnable biases.

Almost all our tests used a feedforward neural controller, but the LSTM controller was not discarded. Also, a one-layer LSTM controller was used, and the figure 3.2 shows how it is. The next equations are followed by our feedforward controller:

$$
\boldsymbol{h}_t = tanh\left(W_h\boldsymbol{\chi}_t+\boldsymbol{b}_h\right) = tanh\left(W_h[\boldsymbol{x}_t;\boldsymbol{r}_{t-1}^1;\cdots;\boldsymbol{r}_{t-1}^R]+\boldsymbol{b}_h\right)
$$

Where, as in LSTM controller, W_h indicates the learnable weights matrix and $\boldsymbol{b}_h$ denotes the learnable biases vector. The $\boldsymbol{h}_t$ vector initialization is defined by $\boldsymbol{h}_0 = \boldsymbol{0}$.

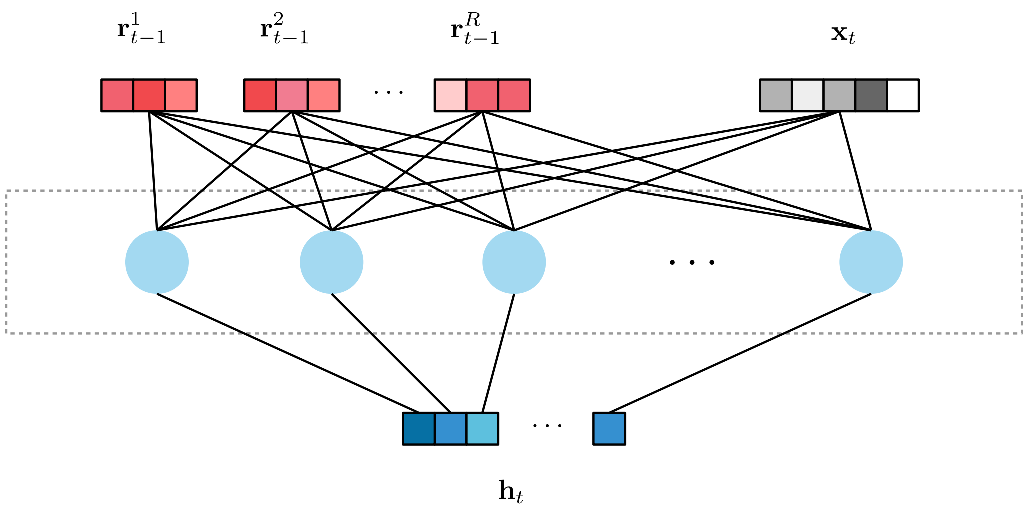

Figure 3.1: A graphic model for a feed forward neural network controller.

The model suggests that at each time-step, despite the controller type, an output vector $\boldsymbol{\upsilon}_t \in \mathbb{R}^Y$ and an interface vector $\boldsymbol{\xi}_t \in \mathbb{R}^{(W\times R)+3W+5R+3}$ must be emitted. For an LSTM those vectors are defined as:

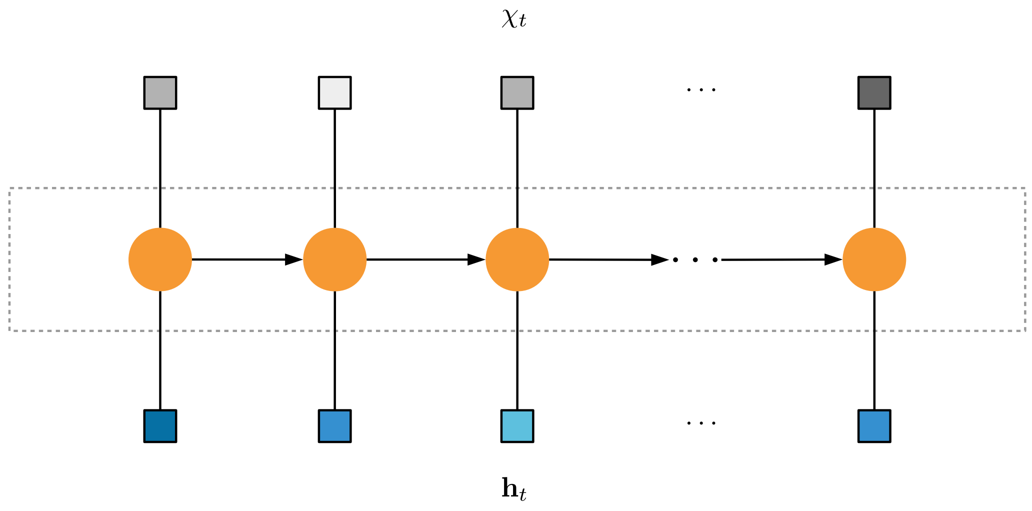

Figure 3.2: A graphic model for an one layer LSTM controller.

$$\boldsymbol{\upsilon}_t = W_{\boldsymbol{y}}[\boldsymbol{h}_t^1; \cdots ; \boldsymbol{h}_t^l]$$
$$\boldsymbol{\xi}_t = W_{\boldsymbol{\xi}}[\boldsymbol{h}_t^1; \cdots ; \boldsymbol{h}_t^l]$$

The output can be taken as a function of the complete history $(\boldsymbol{\upsilon}_t, \boldsymbol{\xi}_t) = \mathcal{N}([\boldsymbol{\chi}_1; \cdots ; \boldsymbol{\chi}_t]; \boldsymbol{\theta})$ in case of a recurrent controller, or a function of the last-time state in case of the feed-forward controller $(\boldsymbol{\upsilon}_t, \boldsymbol{\xi}_t) = \mathcal{N}(\boldsymbol{\chi}_t; \boldsymbol{\theta})$, where $\boldsymbol{\theta}$ defines the set of trainable parameters. Finally, the sum of the $\boldsymbol{\upsilon}_t$ vector and the concatenation of the current read vectors $[\boldsymbol{r}_t^1; \cdots ; \boldsymbol{r}_t^R]$ multiplied by the $RW \times Y$ weight matrix W_r defines the output vector $\boldsymbol{y}_t$.

$$\boldsymbol{y}_t = \boldsymbol{\upsilon}_t + W_r[\boldsymbol{r}_t^1; \cdots ; \boldsymbol{r}_t^R]$$

This arrangement allows the DNC for conditioning its output decisions on memory that has just been read, so, the only way to pass information back to the controller is creating a cycle in the process, used for generating the $\boldsymbol{\upsilon}_{t+1}$ vector.

3.2.2 Interface parameters

The controller generates the interface parameters vector $\boldsymbol{\xi}_t$. The dimensionality $\boldsymbol{\xi}_t \in \mathbb{R}^{(W \times R)+3W+5R+3}$ is because it must be divided as follows (The parameters dimensions are shown in table 3.1):

$$\boldsymbol{\xi}_t = \left[\boldsymbol{k}_t^{r,1}; \cdots ; \boldsymbol{k}_t^{r,R}; \hat{\beta}_t^{r,1}; \cdots ; \hat{\beta}_t^{r,R}; \boldsymbol{k}_t^w; \hat{\beta}_t^w; \hat{\boldsymbol{e}}_t; \boldsymbol{\upsilon}_t; \hat{f}_t^1; \cdots ; \hat{f}_t^R; \hat{g}_t^a; \hat{g}_t^w; \hat{\boldsymbol{\pi}}_t^1; \cdots ; \hat{\boldsymbol{\pi}}_t^R\right]$$

To ensure that each one of these pieces lies in the correct domain, they are processed in order for getting the next parameters:

Parameter	Preprocessed parameter	Final parameter	Dimension	Domain
R read keys	$\boldsymbol{k}_t^{r,i}; 1 \leq i \leq R$	$\boldsymbol{k}_t^{r,i}$	$W \times R$	$\boldsymbol{k}_t^{r,i} \in \mathbb{R}^W$
R read strengths	$\hat{\beta}_t^{r,i}; 1 \leq i \leq R$	$\beta_t^{r,i} = oneplus(\hat{\beta}_t^{r,i})$	R	$\beta_t^{r,i} \in [1, \infty)$
A write key	$\boldsymbol{k}_t^w$	$\boldsymbol{k}_t^w$	W	$\boldsymbol{k}_t^w \in \mathbb{R}^W$
A write strength	$\hat{\beta}_t^w$	$\beta_t^w = oneplus(\hat{\beta}_t^w)$	1	$\beta_t^w \in [1, \infty)$
An erase vector	$\hat{\boldsymbol{e}}_t$	$\boldsymbol{e}_t = \sigma(\hat{\boldsymbol{e}}_t)$	W	$\boldsymbol{e}_t \in [0, 1]^W$
A write vector	$\boldsymbol{v}_t$	$\boldsymbol{v}_t$	W	$\boldsymbol{v}_t \in \mathbb{R}^W$
R free gates	$\hat{f}_t^i; 1 \leq i \leq R$	$f_t^i = \sigma(\hat{f}_t^i)$	R	$f_t^i \in [0, 1]$
An allocation gate	$\hat{g}_t^a$	$g_t^a = \sigma(\hat{g}_t^a)$	1	$g_t^a \in [0, 1]$
A write gate	$\hat{g}_t^w$	$g_t^w = \sigma(\hat{g}_t^w)$	1	$g_t^a \in [0, 1]$
R read modes	$\hat{\boldsymbol{\pi}}_t^i; 1 \leq i \leq R$	$\boldsymbol{\pi}_t^i = softmax(\hat{\boldsymbol{\pi}}_t^i)$	$3 \times R$	$\boldsymbol{\pi}_t^i \in \mathcal{S}_3$

Table 3.1: The interface parameters processing.

In our propossed model, the parameters are not generated as controller information purely extracted, what we do is to extract an $\boldsymbol{h}_t$ vector(showed in last section) interpreted as main information from reading vectors, then that vector is passed as to multiple neural networks for getting desired parameters. Figure 3.3 shows a graphical model of how this is done.

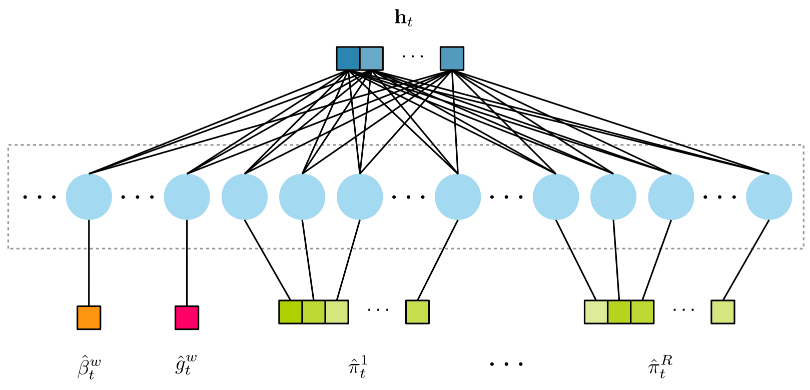

Figure 3.3: A representative model of the interface parameters generation.

In following sections, how these parameters are used is explained.

3.2.3 Memory Addressing

The model uses a combination of content-based addressing and dynamic memory allocation for determining where to write in memory, and a combination of content-based addressing and temporal memory linkage for determining where to read. These mechanisms, all of which are parameterized by the interface parameters obtained from the vector $\boldsymbol{\xi}_t$ which was emitted by the controller.

3.2.3.1 Content-based addressing

The content lookup(for reading or writing) on memory $M \in \mathbb{R}^{N \times W}$ use the following function:

$$\mathcal{C}(M, \boldsymbol{k}, \beta)[i] = \frac{e^{\mathcal{D}(\boldsymbol{k}, M[i,\cdot])\beta}}{\sum_j e^{\mathcal{D}(\boldsymbol{k}, M[j,\cdot])\beta}}$$

where $\boldsymbol{k} \in \mathbb{R}^W$ is a lookup key obtained via interface parameters vector either write or read operation, $\beta \in [1, \infty)$ is a scalar representing a key strength also obtained by the interface parameters vector either read or write operation, $M[i, \cdot], M[j, \cdot] \in \mathbb{R}^W$ represents the i^{th} and j^{th} rows of memory matrix respectively, and finally, $\mathcal{D}$ is the cosine similarity defined by:

$$\mathcal{D}(\boldsymbol{u}, \boldsymbol{v}) = \frac{\boldsymbol{u} \cdot \boldsymbol{v}}{||\boldsymbol{u}||||\boldsymbol{v}||}$$

The weighting $\mathcal{C}(M, \boldsymbol{k}, \beta) \in \mathcal{S}_N$ defines a normalized probability distribution over the memory locations.

Figure 3.4 represents graphically how content-based addressing works. The cosine similarity is a similitude measure, then, when we compare the key $\boldsymbol{k}$ with each memory row the closer rows are the third and the fifth, the other rows are almost orthogonal, so, the similitude is low, that is the reason because the probability distribution represented by the content vector $\boldsymbol{c}$ is more concentrated in the third and the fifth slots. The intensity factor β functionality is for increasing the difference between the most similar vectors and the less ones.

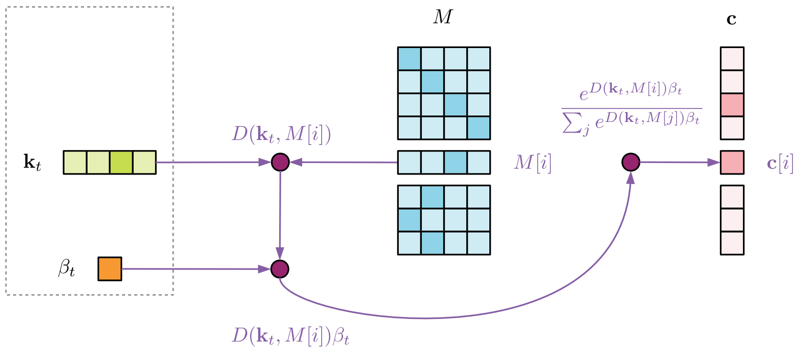
M
$\mathbf{c}$
$\frac{e^{D(\mathbf{k}_t, M[i])\beta_t}}{\sum_j e^{D(\mathbf{k}_t, M[j])\beta_t}}$
$\mathbf{k}_t$
$D(\mathbf{k}_t, M[i])$
$M[i]$
$\mathbf{c}[i]$
β_t
$D(\mathbf{k}_t, M[i])\beta_t$

Figure 3.4: A graphic model for the DNC content-based addressing.

3.2.3.2 Dynamic memory allocation

The dynamic memory allocation is an operation used exclusively for writing weights. It allow the controller for free and allocate memory as needed, this is by developing an analogous of the ”free list” method, it is, to keep the information in those slots where the information is accesed constinuously and free(erase) information on those where information is less accessed. Let $\boldsymbol{u}_t \in [0,1]^N$ the memory usage vector at time t, and define $\boldsymbol{u}_0 = \mathbf{0}$.

When the controller emits the interface parameters vector, it includes a free gates f_t^i set, one per reading head, that determines whether the most recently read locations can be freed. The free gates are used for defining the retention vector $\boldsymbol{\psi}_t \in [0,1]^N$ that represents by how many each location will not be freed, and it is defined by:

$$\boldsymbol{\psi}_t = \prod_{i=1}^{R} \left(\mathbf{1} - f_t^i \boldsymbol{w}_{t-1}^{r,i} \right)$$

Where additionally, the i^{th} last time read weights vector is $\boldsymbol{w}_{t-1}^{r,i}$.

Figure 3.5 shows the memory retention vector generation. Here, the free gates f are represented in three values, the first(f_t^1) is high, the second(f_t^i) is medium and the third(f_t^R) is low. The read weights are equal for simplicity. The forget gates and the read weighting product means how much the read weighting vector will be forgotten, and its complement represents how much they will be maintained. That causes that the first complement $(1 - f_t^1 \boldsymbol{w}_{t-1}^{r,1}[j])$ is low, the second one is medium and the third is high. When these values are multiplied the retention vector will be obtained. Here the retention vector has medium values because at least, one read weighting slot uses the i^{th} space. The retention vector means by how much each memory matrix column slot will be kept.

Once defined and obtained the memory retention vector $\boldsymbol{\psi}_t$ we can define the usage vector $\boldsymbol{u}_t$ as:

$$\boldsymbol{u}_t = \left(\boldsymbol{u}_{t-1} + \boldsymbol{w}_{t-1}^{w} - \boldsymbol{u}_{t-1} \odot \boldsymbol{w}_{t-1}^{w} \right) \odot \boldsymbol{\psi}_t$$

Where the element-wise product is denoted by $\odot$. The information will be more retained while $\boldsymbol{\psi}_t[i] \approx 1$. The operations inside the parentheses can be interpreted as how much each location was used in the last time-step. The multiplication defines that those locations with more usage must be retained. The usage of each location will increase up to a maximum of 1 with every writing and can only be subsequently decreased to a minimum of 0 using the free gates. The $\boldsymbol{u}_t$ vector elements are therefore bounded in the range $[0,1]$.

Figure 3.6 shows a graphical model of how usage vector is generated. The first done operation($\boldsymbol{u}_{t-1} \odot \boldsymbol{w}_{t-1}^{w} \in [0,1]^N$) indicates a direct proportionality between how much was used a memory row slot(a memory column) in the last time-step and how much we want to write in there. If we were using a slot and we want to write in the same memory slot then the value will be high, if we were not using a

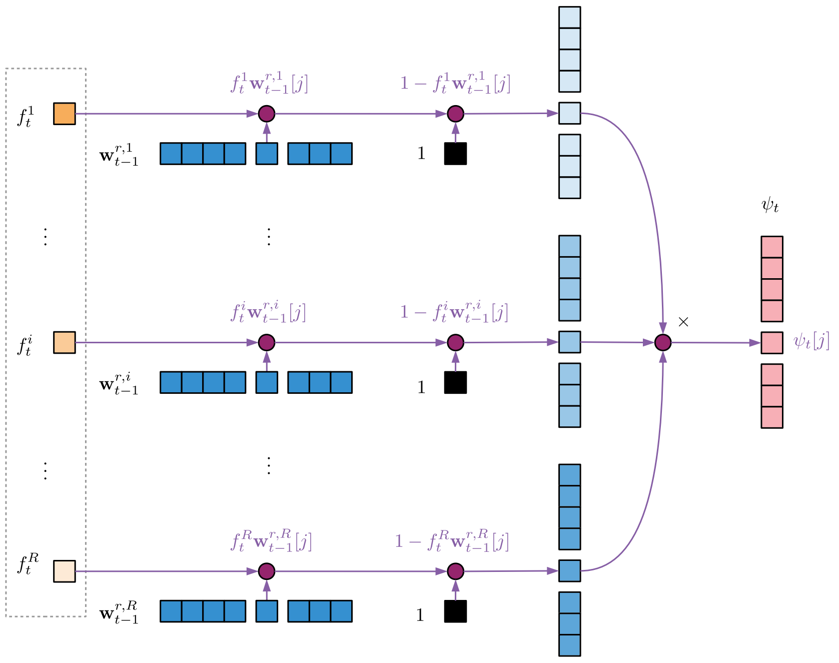
f_t^1
$\mathbf{w}_{t-1}^{r,1}$
$f_t^1 \mathbf{w}_{t-1}^{r,1}[j]$
$1 - f_t^1 \mathbf{w}_{t-1}^{r,1}[j]$
1
f_t^i
$\mathbf{w}_{t-1}^{r,i}$
$f_t^i \mathbf{w}_{t-1}^{r,i}[j]$
$1 - f_t^i \mathbf{w}_{t-1}^{r,i}[j]$
1
f_t^R
$\mathbf{w}_{t-1}^{r,R}$
$f_t^R \mathbf{w}_{t-1}^{r,R}[j]$
$1 - f_t^R \mathbf{w}_{t-1}^{r,R}[j]$
1
$\times$
ψ_t
$\psi_t[j]$

Figure 3.5: A graphic model for the DNC memory retention vector generation.

memory slot and we want to write on it, and vice-versa, the value will be medium, finally, if we were not using a memory slot and we do not want to write in there the value will be low. All combinations are represented in the orange vector. Is important mentioning that given the usage vector domain and the write weighting domain $\boldsymbol{w} \in \Delta_N$ this operation follows the restriction $(\boldsymbol{u}_{t-1} \odot \boldsymbol{w}_{t-1}^w)[j] \leq \boldsymbol{u}_{t-1}, \boldsymbol{w}_{t-1}^w$.

The second operation $(\boldsymbol{w}_{t-1}^w - \boldsymbol{u}_{t-1} \odot \boldsymbol{w}_{t-1}^w)$ indicates, in general terms, how much I can write where I want to write in each memory slot. This attenuates the write weighting and can be considered as a writing verification because it validates if is necessary depending on the previous use. The brown vector indicates this operation. All combinations are included because of the write weighting $\boldsymbol{w}_{t-1}^w$ dependence.

The third operation $(\boldsymbol{u}_{t-1} + \boldsymbol{w}_{t-1}^w - \boldsymbol{u}_{t-1} \odot \boldsymbol{w}_{t-1}^w)$ gives meaning to all these calculations. It can be interpreted as the memory use increase depending on if we want to write in a memory slot.

Finally, the last operation will increase the difference between the low and high values in the third operation, depending on if the model decides to keep the information or not. This makes sense because if a memory slot does not need to be retained, it has no use and vice-versa.

Once $\boldsymbol{u}_t$ has been determined, the free list $\boldsymbol{\phi}_t \in \mathbb{Z}^N$ is defined by sorting the memory locations indices in ascending order of usage, where the least used location is in the $\boldsymbol{\phi}_t[1]$ index. Then the allocation weighting vector $\boldsymbol{a}_t \in \Delta_N$ is defined by:

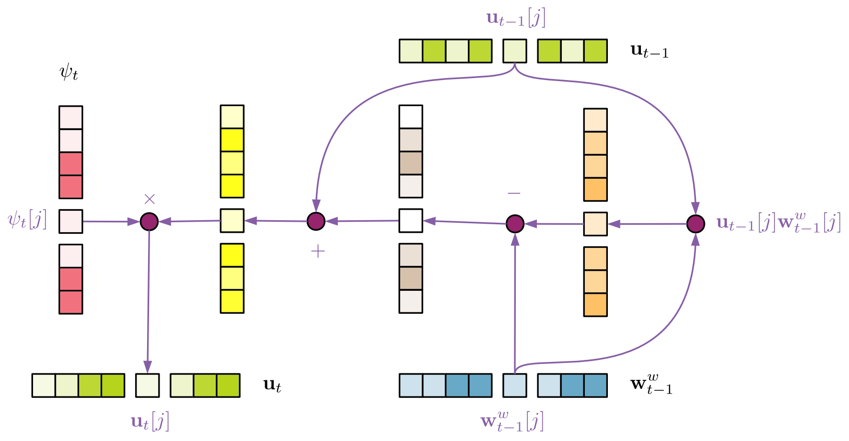

Figure 3.6: A graphic model for the DNC memory usage vector generation.

$$\boldsymbol{a}_t[\boldsymbol{\phi}_t[j]] = (1 - \boldsymbol{u}_t[\boldsymbol{\phi}_t[j]]) \prod_{i=1}^{j-1} \boldsymbol{u}_t[\boldsymbol{\phi}_t[i]]$$

And it is used to provide new allocations for writing. While all usages are 1, then $\boldsymbol{a}_t = \mathbf{0}$ and the controller can no longer allocate memory without first freeing used locations. Δ_N is the non-negative orthant of $\mathbb{R}^N$ with the unit simplex as boundary and it is defined as:

$$\Delta_N = \left\{ \boldsymbol{\alpha} \in \mathbb{R}^N : \boldsymbol{\alpha}_i \in [0,1], \sum_{i=1}^{N} \boldsymbol{\alpha}_i \leq 1 \right\}$$

Figure 3.7 shows a graphic representation of allocation weighting vector calculation. In the left figure part, we can look what the free list $\boldsymbol{\phi}$ is doing, it is only sorting the data in ascending order. After that, the complement $1 - \boldsymbol{u}_t[\boldsymbol{\phi}_t[j]]$ means that we can use that space for new data storing and when it is multiplied by the product $\prod_{i=1}^{j-1} \boldsymbol{u}_t[\boldsymbol{\phi}_t[i]]$ it will be attenuated, this is, while more using has a location is less probable that we can write in there because exists slots with less use. The allocation vector $\hat{\boldsymbol{a}}$ means that we can use that memory space for information storing. Finally, the allocation vector is sorted again to put it back in the original order.

3.2.3.3 Write weighting and writing memory

Write weighting is associated with a vector generation that will be used to write in memory. A mechanism can write to newly allocated locations, or to locations addressed by content, or it can choose not to write at all. First, a write content weighting $\boldsymbol{c}_t^w \in \mathcal{S}_N$ is constructed using the write key $\boldsymbol{k}_t^w$ and the write strength β_t^w(The controller generates these parameters and are included in the interface parameters):

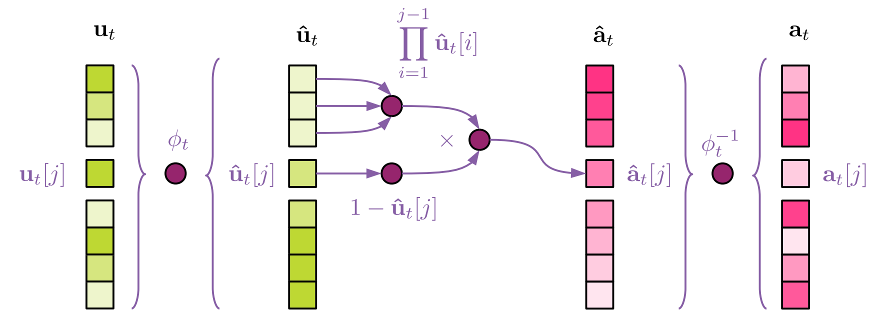

Figure 3.7: A graphic model for the DNC allocation weighting generation.

$$\boldsymbol{c}_t^w = \mathcal{C}(M_{t-1}, \boldsymbol{k}_t^w, \beta_t^w)$$

Then $\boldsymbol{c}_t^w$ is interpolated with the allocation weighting $\boldsymbol{a}_t$ to determine the write weighting $\boldsymbol{w}_t^w \in \Delta_N$.

$$\boldsymbol{w}_t^w = g_t^w \left[g_t^a \boldsymbol{a}_t + (1 - g_t^a)\boldsymbol{c}_t^w\right]$$

Where $g_t^a \in [0, 1]$ is the write gate obtained from the interface parameters. If the write gate is 0, then nothing is written, regardless of the other write parameters; it can, therefore, be used to protect the memory from unnecessary modifications. The operation inside parenthesis can be interpreted as the total content that allocation weighting or content lookup will maintain.

Figure 3.8 shows the write weighting generation. Here, the interpretation is simple, the weight weighting generation decides how much will write from the content vector $\boldsymbol{c}$ and how much from the allocation vector $\boldsymbol{a}$ via the allocation gate g^a. In the figure, we can see that the allocation gate was set near to 1 for ignoring almost all the content vector. If it is set near to 0 then the allocation vector will be the ignored. Finally, how much importance has the sum of both vectors is decided by the write gate g^w, to add that to the write weighting $\boldsymbol{w}^w$, that will decide in which memory slot is better to write. In this case, the allocation weighting has more importance, so the write weighting is more similar to it.

Finally, the next operation produces the next time-step memory state:

$$M_t = M_{t-1} \otimes (E - \boldsymbol{w}_t^w \boldsymbol{e}_t^T) + \boldsymbol{w}_t^w \boldsymbol{v}_t^T$$

Where $\otimes$ denotes the element-wise product and E is an $N \times W$ ones matrix. This operation makes sense if we interpret it as how much we will erase, with the erasing vector $\boldsymbol{e}$, of the current memory M_t and how much we will add, with the write weighting $\boldsymbol{v}$, depending on the write weighting $\boldsymbol{w}^w$. The first operation $M_{t-1} \otimes (E - \boldsymbol{w}_t^w \boldsymbol{e}_t^T)$ will delete the slots that are not needed and the adding of

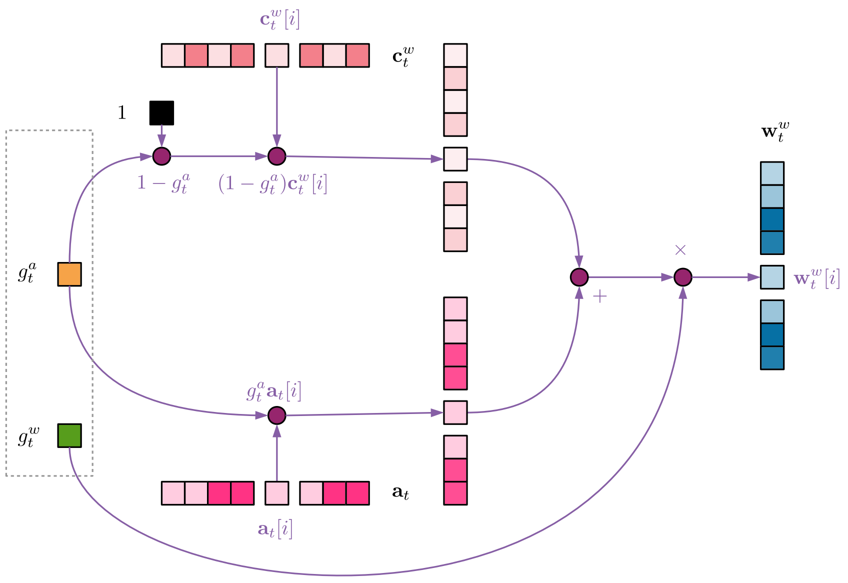

Figure 3.8: A graphic model for the DNC write weighting generation.

$\boldsymbol{w}_t^w \boldsymbol{v}_t^T$ will ad the new content.

3.2.3.4 Temporal memory linkage

The temporal memory linkage is defined as a multiple ”calls” association from memory locations to an order in which those were called. It can be taken as the memory location storage sorting. This information is useful in many situations, for example when an instructions sequence must be recorded and retrieved in order.

With that target, the model uses a temporal link matrix $L_t \in [0,1]^{N\times N}$ to keep consecutively modified memory locations tracking.

$L_t[i,j]$ represents the degree of which location i was the location written after the location j, and each L_t row and column defines the location weighting: $L_t[i,\cdot] \in \Delta_N$ and $L_t[\cdot,j] \in \Delta_N$ for all i, j and t. To define L_t, a precedence weighting $\boldsymbol{p}_t \in \Delta_N$ is required, where the element $\boldsymbol{p}_t[i]$ represents the degree of which location i was the last one written. The next recurrence relation defines the precedence $\boldsymbol{p}_t$ vector:

$$\boldsymbol{p}_0 = \mathbf{0}$$

$$\boldsymbol{p}_t = \left(1 - \sum_i \boldsymbol{w}_t^w[i]\right)\boldsymbol{p}_{t-1} + \boldsymbol{w}_t^w$$

Where $\boldsymbol{w}_t^w$ is the write weighting.

The first operation $\sum_i \boldsymbol{w}_t^w[i]$ indicates, in general, how much the memory was written, then, the complement is obtained and can be interpreted as how much was not written. Once the complement was obtained the last time-step precedence vector is multiplied, this is because if a lot of new content will be written then the current precedence will have less value because all the new content will dominate over the last content, so, the precedence vector needs to be attenuated a lot, and vice-versa. Finally, to each precedence vector slot will be added the same write weighting index slot, this, to indicate by how much each memory row has changed and which one can be considered the last one written. When the last operation is done the memory changes over the time-steps are mixed in the precedence vector, it is the reason for its name.

The precedence vector generation is shown in figure 3.9. The write weighting was set with low values, so, after the sum complement, the obtained value will be near to 1 which causes almost the same precedence vector after the product. Finally, the precedence vector incrementation will be homogeneous because all write weightings locations are increased in the same quantity.

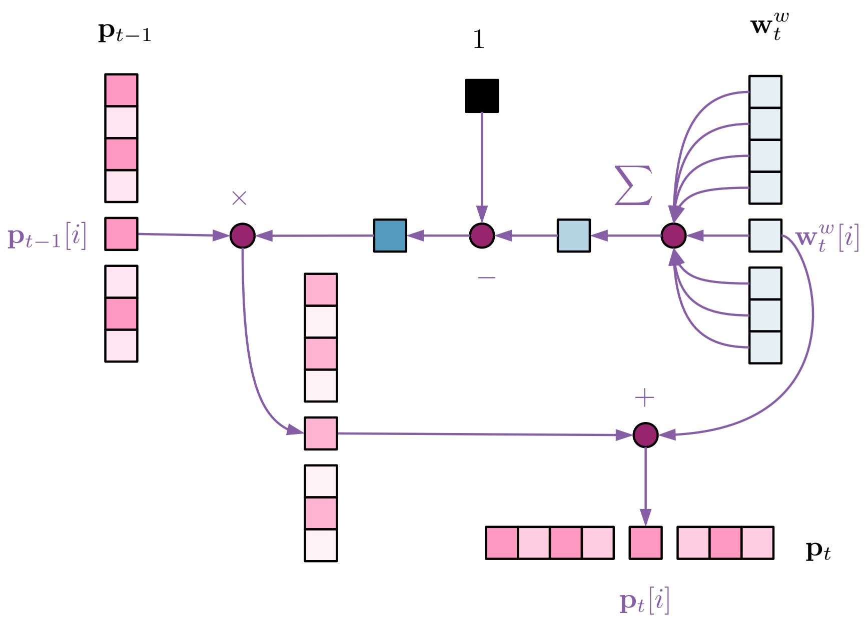

Figure 3.9: A graphic model for the DNC precedence vector calculation.

At every time-step a location is modified, the link matrix is updated to remove old links to and from that location. Then the new links from the last-written location are added. To implement that logic the following recurrence relation is defined:

$$L_0[i,j] = 0; \forall i, j$$
$$L_t[i,i] = 0; \forall i$$
$$L_t[i,j] = (1 - \boldsymbol{w}_t^w[i] - \boldsymbol{w}_t^w[j])L_{t-1}[i,j] + \boldsymbol{w}_t^w[i]\boldsymbol{p}_{t-1}[j]$$

First, this operation $(1 - \boldsymbol{w}_t^w[i] - \boldsymbol{w}_t^w[j])L_{t-1}[i,j] = (1 - (\boldsymbol{w}_t^w[i] + \boldsymbol{w}_t^w[j]))L_{t-1}[i,j]$ means the same as in the precedence vector. It is a linkage memory slot information attenuation. If the position (i,j) will be written by the write weighting the linkage memory will be attenuated a lot because the new information is which will have more importance than the old because a lot of information in that position was the last-written. Finally, the $\boldsymbol{w}_t^w[i]\boldsymbol{p}_{t-1}[j]$ addition is the operation that generates the strength in the linkage between the previous steps and the current one.

Figure 3.10 shows how the linkage memory is updated. The write weighting $\boldsymbol{w}^w$ wants to write in the first and the fifth slots, so there a lot of new content will be written, that is the reason because in the linkage matrix L the probability in the spaces $(1,5)$ and $(5,1)$ will be lower; Is not possible that one precedes to another if both were written in the same time-step. Also, the first and fifth rows and columns have low probabilities too because there is no precedence on that spaces and much information will be written. In consequence, the last column has the higher probabilities, that is because the precedence vector indicates that this slot precedes the current operation.

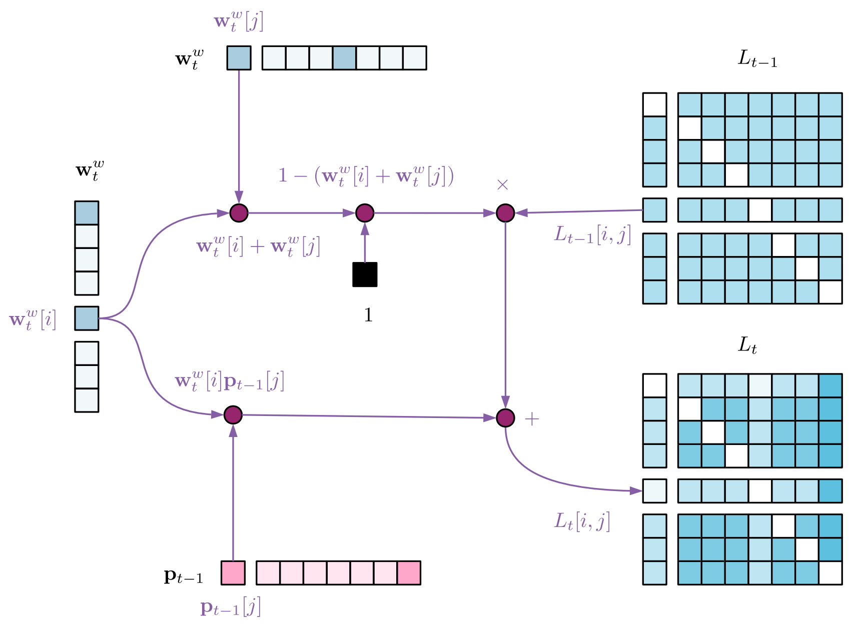

Figure 3.10: A graphic model for the DNC linkage memory updating.

The self-links are excluded (the link matrix diagonal) because it is unclear how to follow a transition from a location to itself. The L_t rows and columns represent the weights of the temporal links going into and out from particular memory slots, respectively. Given L_t, the backward weighting $\boldsymbol{b}_t^i \in \Delta_N$ and forward weighting $\boldsymbol{f}_t^i \in \Delta_N$ for the read head i are defined as:

$$\boldsymbol{f}_t^i = L_t \boldsymbol{w}_{t-1}^{r,i}$$
$$\boldsymbol{b}_t^i = L_t^T \boldsymbol{w}_{t-1}^{r,i}$$

Where $\boldsymbol{w}_t^{r,i}$ is the i^{th} previous time-step read weighting and those vectors can be interpreted as how are related those memory slots from i to j and vice-versa.

3.2.3.5 Read weighting and reading memory

To read from the memory is needed to define the read weighting. R read weighting vectors are needed for sending to the next time-step and they will define how much the content lookup can read backward memory or forward memory slots usage. By first, the read key $\boldsymbol{k}_t^{r,i} \in \mathbb{R}^W$ is used to compute each read head i content weighting $\boldsymbol{c}_t^{r,i} \in \Delta_N$:

$$\boldsymbol{c}_t^{r,i} = \mathcal{C}(M_t, \boldsymbol{k}_t^{r,i}, \beta_t^{r,i})$$

Where each $\boldsymbol{k}_t^{r,i}, \beta_t^{r,i}$ were generated by controller via interface parameters. Each read head also receives a read mode vector $\boldsymbol{\pi}_t^i \in \mathcal{S}_3$, which is also taken from interface parameters, and interpolates among the backward weighting $\boldsymbol{b}_t^i$, the forward weighting $\boldsymbol{f}_t^i$ and the content read weighting $\boldsymbol{c}_t^{r,i}$, thereby determining the read weighting $\boldsymbol{w}_t^{r,i} \in \mathcal{S}_3$

$$\boldsymbol{w}_t^{r,i} = \boldsymbol{\pi}_t^i[1]\boldsymbol{b}_t^i + \boldsymbol{\pi}_t^i[2]\boldsymbol{c}_t^{r,i} + \boldsymbol{\pi}_t^i[3]\boldsymbol{f}_t^i$$

If $\boldsymbol{\pi}_t^i[2]$ dominates the read mode, then the weighting reverts to content lookup using $\boldsymbol{k}_t^{r,i}$. If $\boldsymbol{\pi}_t^i[3]$ dominates, then the read head iterates through memory locations in the order they were written, ignoring the read key. If $\boldsymbol{\pi}_t^i[1]$ dominates, then the read head iterates in the reverse order.

Figure 3.11 shows the read weighting generation. It is simple, depending on which read mode dominates is from where the most content will be extracted. In the example, the dominating read mode is the first, so, the backward reading will predominate.

With those read weighting vectors, we obtain R next time-step read vectors using:

$$\boldsymbol{r}_t^i = M^T \boldsymbol{w}_t^{r,i}$$

Each one of the generated read vectors is appended to controller input at the next time-step, giving access to the memory contents and creating the recurrence in the Differentiable Neural Computer.

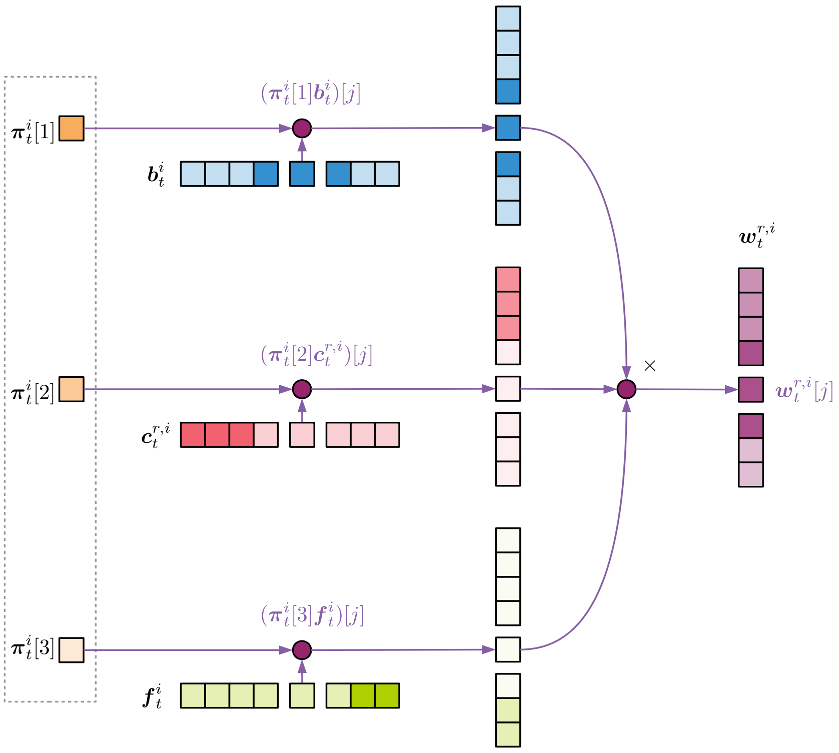

Figure 3.11: A graphic model for DNC read weighting generation.

Chapter 4

The Differentiable Neural Computer implementation

4.1 Introduction

For developing our Differentiable Neural Computer, Python was used as the programming language and Tensorflow as the machine learning framework. Each component was developed as a Differentiable Neural Computer module, this for generating independence among each component, just in case of replacing, do it independently.

Maybe, you want to experiment with this architecture by testing it with the different showed examples, or create your own model by combining different components, so, next is a free access link for our development `https://github.com/cobu93/DNC`.

In the next sections, I am explaining, part by part, how each component was programmed, and the most representative code parts. If you want to download or check the code it is available in the previous paragraph link.

4.2 General

First, for generating any feedforward network, a simple code was developed. By specifying the description of layers a feedforward neural network is built.

```
class FeedForwardNN():

 def __init__(self, layers_desc, scope):
  self.layers = layers_desc
  self.scope = scope
  self.h = {}
 def run_feed_forward_nn(self, inputs):
  self.h.clear()

  with tf.variable_scope(self.scope):

   for i in range(0, len(self.layers)):

    if self.layers[i].is_input:
     self.h[i] = inputs

    else:
     with tf.variable_scope(self.layers[i].name):
      self.h[i] =  self.layers[i].activation(
          Utility._linear(
           self.h[i - 1],
           self.layers[i].size,
           self.layers[i].has_bias
          )
        )

  return self.h[len(self.layers) - 1]
```

Here `Utility._linear` is a well-known function that used to be in the Tensorflow framework and was removed. This function automatically creates linear combinations, define the learnable weights matrices and biases, making the feedforward networks definition easier. We defined the `Utility` class for using it for all not-related Differentiable Neural Computers logic but necessary functionality.

4.3 Controller

The controller is a feed forward network defined by

```
self.controller = FeedForwardController(
 layers_desc = [
  InputLayerDescription('input_header'),
  OutputLayerDescription(self.hsize, 'header', tf.nn.tanh)
 ],
 name='controller'
)
```

where `FeedForwardController` class is an interface for creating a feedforward network. Here, the controller is defined as a neural network that receives an input vector, sends it through a tanh activation function, and outputs an `hsize` size vector. `hsize` is a configurable parameter that is useful for generating more or less representative information. It is util for generating interface parameters and the DNC's output.

4.4 Output layer

For generating the DNC's output, the output layer is defined. The next description layers define it:

```
self.output_layer = DNCOutputLayer(
 layers_desc = [
  InputLayerDescription('input_y'),
  OutputLayerDescription(self.osize,
   'output',
   tf.nn.tanh, has_bias=False
  )
 ],
 name='dnc_output'
)
```

The `DNCOutputLayer` generates a feedforward network that receives the header and read vectors as the input. Then it generates the output from the passed header vector through the defined layer(of size `osize`) and summing the concatenated read vectors and a bias for generating the current time-step DNC output.

The `DNCOutputLayer` developed operations represent the next model equation:

$$\boldsymbol{y}_t = \boldsymbol{v}_t + W_r[\boldsymbol{r}_t^1; \cdots ; \boldsymbol{r}_t^R]$$

4.5 Interface parameters

The interface parameters vector were not obtained from a simple neural network. Each parameter was obtained and processed by a different feedforward neural network. To realize this the `InterfaceParameters` class was defined. Multiple feedforward networks were defined, passing the layers descriptions, for generating the different parameters using the `FeedForwardNN` class. Once all description layers definitions were determined, the only class use is for running the neural networks.

The layers description code is the next:

```
self.interface_parameters = InterfaceParameters(
  key_w_layers_desc=[
    InputLayerDescription('input_key'),
    OutputLayerDescription(
     self.mcolumns,
     'write_key',
     tf.nn.tanh
    )
  ],

  intensity_w_layers_desc=[
    InputLayerDescription('input_intensity'),
    OutputLayerDescription(
     self.SCALAR_SIZE,
     'write_intensity',
     tf.nn.softplus
    )
  ],

  erase_layers_desc=[
    InputLayerDescription('input_erase'),
    OutputLayerDescription(
     self.mcolumns,
     'erase',
     tf.nn.sigmoid
    )
  ],

  add_layers_desc=[
    InputLayerDescription('input_add'),
    OutputLayerDescription(
     self.mcolumns,
     'add',
     tf.nn.tanh
    )
  ],

  allocation_layers_desc=[
    InputLayerDescription('input_allocation'),
    OutputLayerDescription(
     self.SCALAR_SIZE,
     'allocation',
     tf.nn.sigmoid
    )
```

```
],

write_gate_layers_desc=[
  InputLayerDescription('input_write_gate'),
  OutputLayerDescription(
   self.SCALAR_SIZE,
   'write_gate',
   tf.nn.sigmoid
  )
],

free_gate_layers_desc=[
  InputLayerDescription('input_free_gates'),
  OutputLayerDescription(
   self.SCALAR_SIZE  * self.rvectors,
   'free_gates',
   tf.nn.sigmoid
  )
],

key_r_layers_desc=[
  InputLayerDescription('input_key'),
  OutputLayerDescription(
   self.mcolumns * self.rvectors,
   'read_keys',
   tf.nn.tanh
  )
],

intensity_r_layers_desc=[
  InputLayerDescription('input_intensity'),
  OutputLayerDescription(
   self.SCALAR_SIZE  * self.rvectors,
   'read_intensity',
   lambda x: 1 + tf.log(1 + tf.exp(x))
  )
],

read_mode_layers_desc=[
  InputLayerDescription('input_read_mode'),
  OutputLayerDescription(
   3  * self.rvectors,
     'read_mode',
     tf.nn.softmax
    )
  ],

  name='interface_parameters'

)
```

An important feature is that a simple output vector is returned from each feedforward neural network, so, for those multiple-vectors parameters, a linearization is applied for generating a single output vector via reshaping operation. This is the read modes and read keys case, where the linearization is performed by:

```
r_modes = tf.reshape(r_modes, [self.rvectors, 3])
r_keys = tf.reshape(r_keys, [self.rvectors, self.mcolumns])
```

After done, all parameters can be used for writing and reading in memory.

4.6 Write weighting and writing memory

First, the needed operations are those where we need to write in the memory. Here new information is generated, that is, the next time-step memory state, the next time-step write weighting and the next time-step usage vector. The class `Writer` is defined for achieving this.

First, inside the `Writer` class, the content vector is generated.

```
content_vector = ContentAddressing.address(
 key=w_key,
 intensity=w_intensity,
 memory=memory,
 epsilon=epsilon
)
```

After obtaining the content vector, as was mentioned before, for writing, is necessary to allocate memory dynamically. While the allocating memory operation is performed, the allocation weighting and the usage vector are created.

```
allocation_weights, usage_vec = DynamicAllocation.allocate(
 free_gates=r_free_gates,
 last_read_weights=r_weights,
 last_usage_vector=usage_vector,
 last_write_weights=w_weights
)
```

For generating the write weighting vector and the next time-step memory the next operations are done, each one according to an operation mentioned before. The model equations are included above each code.

$$\boldsymbol{w}_t^w = g_t^w[g_t^a \boldsymbol{a}_t + (1 - g_t^a)\boldsymbol{c}_t^w]$$

```
write_weights = tf.multiply(
 write_gate,
 tf.multiply(allocation_gate, allocation_weights) +
        tf.multiply(1 - allocation_gate, content_vector)
)
```

$$M_t = M_{t-1} \otimes (E - \boldsymbol{w}_t^w \boldsymbol{e}_t^T) + \boldsymbol{w}_t^w \boldsymbol{v}_t^T$$

```
n_memory = tf.multiply(
 memory,
 tf.ones(tf.shape(memory)) -
  tf.matmul(tf.transpose(write_weights), erase_vector)
 ) +
 tf.matmul(tf.transpose(write_weights), add_vector)
```

Finally, we return the new time-step parameters

```
return write_weights, usage_vec, n_memory
```

4.7 Read weigthing and reading memory

Finally, is necessary to read the memory. Here, the new information generated is the next time-step read weighting, the next time-step read vectors, the next time-step precedence vector, and the next time-step linkage memory. The class `Reader` is defined for achieving this.

Inside the `Reader` class, the content vector is generated.

```
content_vector = ContentAddressing.address(
 key=r_keys,
 intensity=r_intensities,
 memory=memory,
 epsilon=epsilon
)
```

Next, as we mentioned before, is necessary to generate the temporal linkage matrix.

```
n_precedence, n_linkage_matrix, forward_weights, backward_weights =
TemporalLinkage.link(
 w_weights=n_w_weights,
 last_precedence=precedence_vector,
 last_linkage_matrix=linkage_matrix,
 last_r_weights=r_weights
)
```

Also, the new precedence vector is generated.

After, the next operations are done for getting the read weighting, and the next time-step read vectors(fed back to the controller). Finally, the information is returned.

$$\boldsymbol{w}_t^{r,i} = \boldsymbol{\pi}_t^i[1]\boldsymbol{b}_t^i + \boldsymbol{\pi}_t^i[2]\boldsymbol{c}_t^{r,i} + \boldsymbol{\pi}_t^i[3]\boldsymbol{f}_t^i$$

$$\boldsymbol{r}_t^i = M^T \boldsymbol{w}_t^{r,i}$$

```
r_weights = tf.multiply(
 tf.transpose([read_mode_vector[:, 0]]), backward_weights) +
 tf.multiply(tf.transpose([read_mode_vector[:, 1]]), content_vector) +
 tf.multiply(tf.transpose([read_mode_vector[:, 2]]), forward_weights)

r_vecs = tf.transpose(
 tf.matmul(tf.transpose(memory), tf.transpose(r_weights))
)

return r_weights, r_vecs, n_precedence, n_linkage_matrix
```

Chapter 5

Testing

Here we showed the evaluating results of this architecture over multiple tests. Principally, we compare the DNC versus its predecessor; The Neural Turing Machine. Is important to mention that both models are those developed by me, but guided in other documents [4] [3].

Both models implementations, tests parameters, plots, and descriptions are in the next URLs `https://github.com/cobu93/DNC` and `https://github.com/cobu93/NTM`.

5.1 Generalities

Here, two algorithmic tasks were tested in the DNC and the NTM models: the copy sequences and the associative recall. Both models were trained in the same conditions:

1. The controller was feedforward, with the representative information size of 100, that is, it has 100 output neurons.
2. The controller has a single layer.
3. The memory size is 50×20.
4. The used activation function, for generating the representative information, was $tanh$.
5. The sigmoid function is the activation function for both models. Despite this, the DNC performs extra operations on this output.
6. The used cost function was the mean squared error(MSE) for the DNC and the cross-entropy for the NTM because of the output domain.
7. RMSProp was the used learning algorithm.
8. For both models, the learning rate was set to 0.0001, the decay and the momentum were set to 0.9.

5.2 Copy task

5.2.1 Goal

Prove that DNC is able for copying an input sequence, given in multiple steps, to the output.

5.2.2 Justification

Since the NTM can recover and store information, is necessary to test if the DNC, which is the NTM's evolution, can perform the same task, and compare its performance versus the NTM.

5.2.3 Input

The inputs are random binary vectors sequences, followed by a delimiter flag that indicates the input end. After this, the same vectors number as the input vectors, of the same size, is zero-filled and appended at the end, that is, for representing the time-steps task duration when recovering the input, and additionally, is where the output will be written.

For example, a 3×5 input sequence (3 vectors of 5 bits each one) will be:

$$\left[\begin{array}{ccccc|c}
0 & 1 & 1 & 0 & 0 & 0 \\
0 & 0 & 1 & 0 & 1 & 0 \\
0 & 0 & 0 & 1 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 1 \\
0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0
\end{array}\right]$$

The first 3 rows (from up to bottom) are those vectors that will be copied. The extra zero-column at right will be used, in the fourth row, for putting the delimiter flag. The last 3 zero-rows is where the output will be written.

The tested inputs in this document were:

- Small instances: 3×5 sequences.
- Medium instances: 5×5 sequences.
- Large instances: 10×5 sequences.

5.2.4 Output

The expected output is very similar to the input, but in this case, the extra column is deleted. Also, the input vectors are set to zero and the recovered information will be set in the last rows.

For example, the expected output for the above proposed input is:

$$\left[\begin{array}{ccccc|c} 0 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{array}\right] \xrightarrow{output} \begin{bmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 \end{bmatrix}$$

5.3 Details

For the NTM and the DNC, the next table shows the trainable parameters (weights and biases).

Model	Input size	Parameters
Differentiable Neural Computer	3×5	40688
	5×5	
	10×5	
Neural Turing Machine	3×5	21986
	5×5	
	10×5	

Table 5.1: The input sequences parameters.

The parameters number doesn't variate because the difference isn't between bits but in the vectors number, that is, the input width won't increment and the parameters total will be kept in both models.

The training for all sequences sizes, was done while 100000 epochs with exploration finality, that is, the overfitting, a faster converge, and precision.

5.4 Results

For first sequences size(3×5), the DNC model shows a faster convergence. In 20000 epochs the DNC's testing error was $2.397(10^{-3})$ and the training error was $1.4003(10^{-3})$, while in the same step the NTM's testing error was 0.09632 and for training was 0.9505. The epochs number where the NTM reaches an error near to the DNC error($1.293(10^{-3})$) was in 569500, also the NTM's error variation between training and testing changes abruptly beside the DNC, which looks more stable. Figure 5.1 shows both models training error plots and their details. With exploration finality, the full training was done during 100000 epochs.

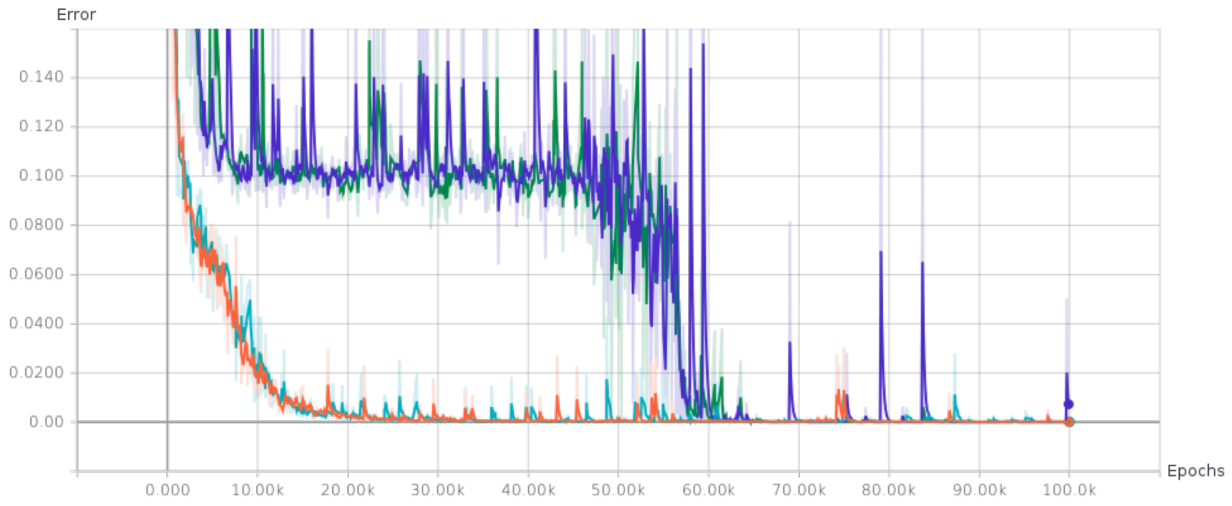

Figure 5.1: Copy task learning error plot for a 3×5 size sequence.
🟦 Neural Turing Machine test. 🟩 Neural Turing Machine train. 🟥 Differentiable Neural Computer test. 🟦 Differentiable Neural Computer train.

In second sequences size(5×5), the DNC shows a faster convergence but both models error were not enough low. After the full training, the DNC's testing error was 0.03952, and the training error was 0.03964 while in the same step the NTM's testing error was 0.1919 and the training error was 0.1901. Here, the NTM's training error and NTM's testing error variation also change abruptly beside the DNCs errors, which looks more stable. Figure 5.2 shows both models training errors plots.

Finally, the third sequences size(10×5), the DNC shows a faster convergence but both models error were not enough low. After the full training, the DNC's testing error was 0.08057 and the training error was 0.07063, while in the same step the NTM's testing error was 0.3214 and the training error was 0.3081. Here, the NTM's training and NTM's testing error variation were not different from the previous input sequence size, here, the DNC looks more stable too. Figure 5.3 shows both models training errors plots.

5.5 Conclusion

Despite the Neural Turing Machine parameters are lower than the Differentiable Neural Computer ones, the NTM convergence time is longer and its stability is too. Also, is important to examine the learning performance because the input size was the only variation among experiments, that means, that the learning performance can be considered as a meta parameters'(learning rate, decay, etc.)

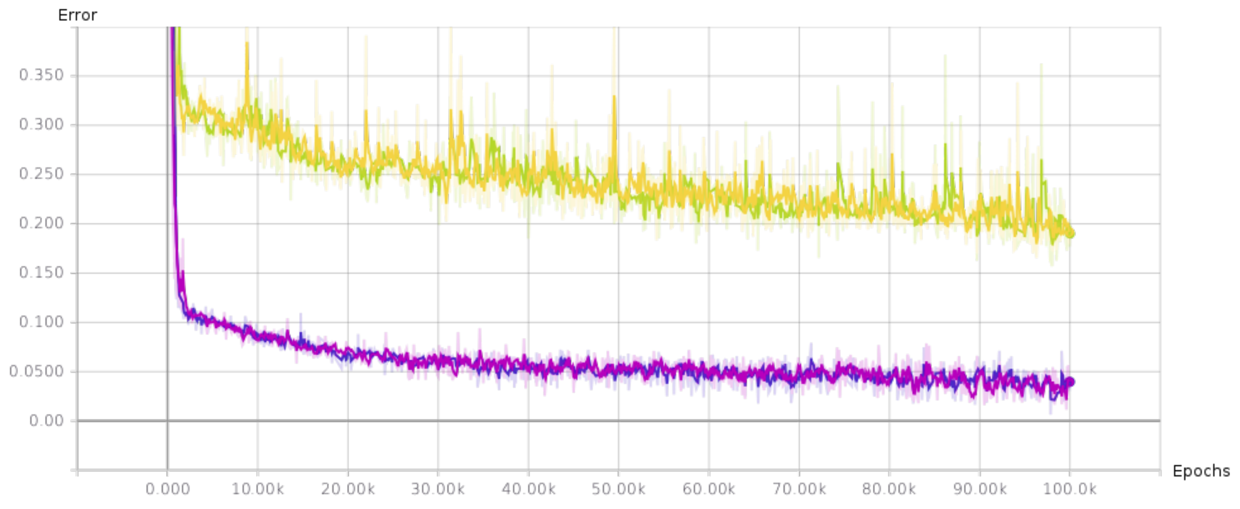

Figure 5.2: Both models copy task learning error plot for a 5×5 size sequence.
🟨 Neural Turing Machine test. 🟩 Neural Turing Machine train. 🟪 Differentiable Neural Computer test. 🟦 Differentiable Neural Computer train.

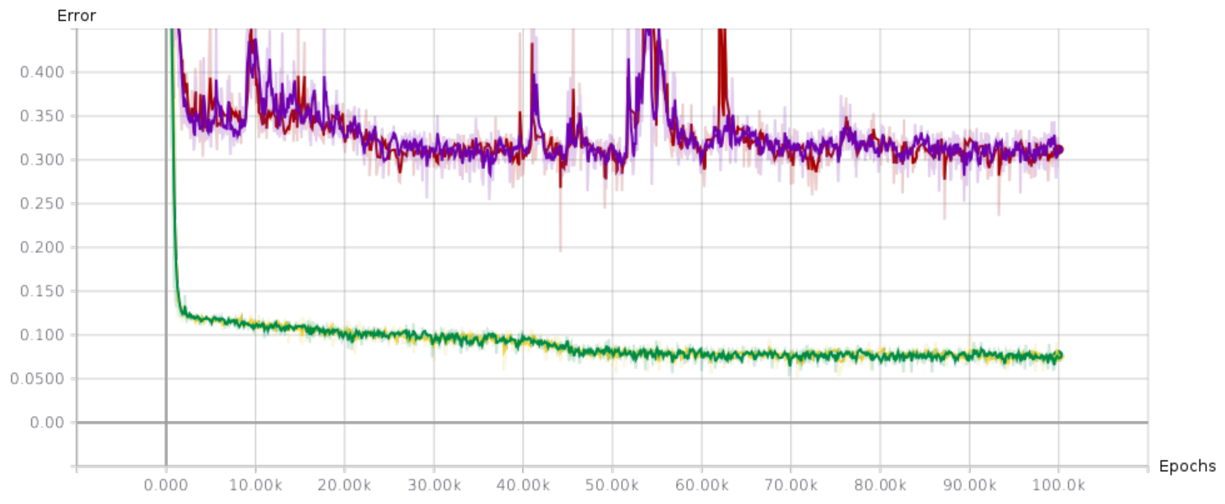

Figure 5.3: Both models copy task learning error plot for a 10×5 size sequence.
🟪 Neural Turing Machine test. 🟥 Neural Turing Machine train. 🟩 Differentiable Neural Computer test. 🟨 Differentiable Neural Computer train.

function, or, in the other hand a model parameters' function(memory size, information vector size, etc.). If we make a learning performance prediction using the figures 5.2 and 5.3 the extrapolation could match with the figure 5.1, indicating that meta parameters are important for obtaining a faster convergence, but in order for getting a small error, the model parameters are what really cares.

5.6 Associative recall task

5.6.1 Goal

Prove that DNC can recover a vector, from a vectors sequence, based on the requested one.

5.6.2 Justification

Since the NTM can recover a vector, from a given vectors sequence, is necessary to test if DNC, which is the evolution of the NTM, is also able to do the same task and measure how well it is done versus the NTM.

5.6.3 Input

The inputs are sequences of random binary vectors followed by a delimiter flag, that will mark the input end. The delimiter row always will contain the recovered vector index. The recovering index is indicated by the next way: The first bit(from left to right) indicates that first vector(from up to bottom) will be recovered, the second bit indicates that the recovered vector is the second one and so on. After this, a one filled-zero vector is appended for representig the time-steps task duration when recovering the requested vector, and additionally, is where the output will be written.

For example, a 3×3 input sequence (3 vectors of 5 bits each one) will be:

$$\left[\begin{array}{ccc|c} 1 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ X & X & X & 1 \\ 0 & 0 & 0 & 0 \end{array}\right]$$

The first 3 rows(from up to bottom) represents the input vectors. A right extra-zero is added because that column will be used to put the delimiter flag in the fourth row. The X represents zero for those ignored vectors and one for desired recovering vector. The last row, the zero vector appended, is where the output will be set.

The tested inputs in this document are:

- Small instances: 3×3 sequences.
- Medium instances: 5×5 sequences.
- Large instances: 10×10 sequences.

5.6.4 Output

A recovered vector is the expected output. The extra column is deleted. Also, the input vectors are set to zero and the recovered information will be set in the last row.

Some examples are:

$$\left[\begin{array}{ccc|c} 1 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \end{array}\right] \xrightarrow{output} \left[\begin{array}{ccc} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \\ 1 & 0 & 0 \end{array}\right]$$

$$\left[\begin{array}{ccc|c} 1 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 \end{array}\right] \xrightarrow{output} \left[\begin{array}{ccc} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 1 & 0 \end{array}\right]$$

5.7 Details

For the NTM model and the DNC model, the next table shows the trainable parameters (weights and biases).

Model	Input size	Parameters
Differentiable Neural Computer	3×3	40088
	5×5	40688
	10×10	42188
Neural Turing Machine	3×3	21586
	5×5	21986
	10×10	22986

Table 5.2: The input sequences parameters.

The training for all sequences sizes was done during 100000 epochs.

5.8 Results

For first sequences size(3×3), we can observe that the DNC and the NTM stability was not really well, but, by the end of the training epochs, the DNC shows more stability and a lower error. At the training end, the DNC testing error was $2.5354(10^{-3})$ and the DNC training error was $3.9231(10^{-4})$, while the NTM reports a testing error of $2.9052(10^{-3})$, and a training error of 0.07504. Figure 5.4 shows the error plots for training phase of both models and its details.

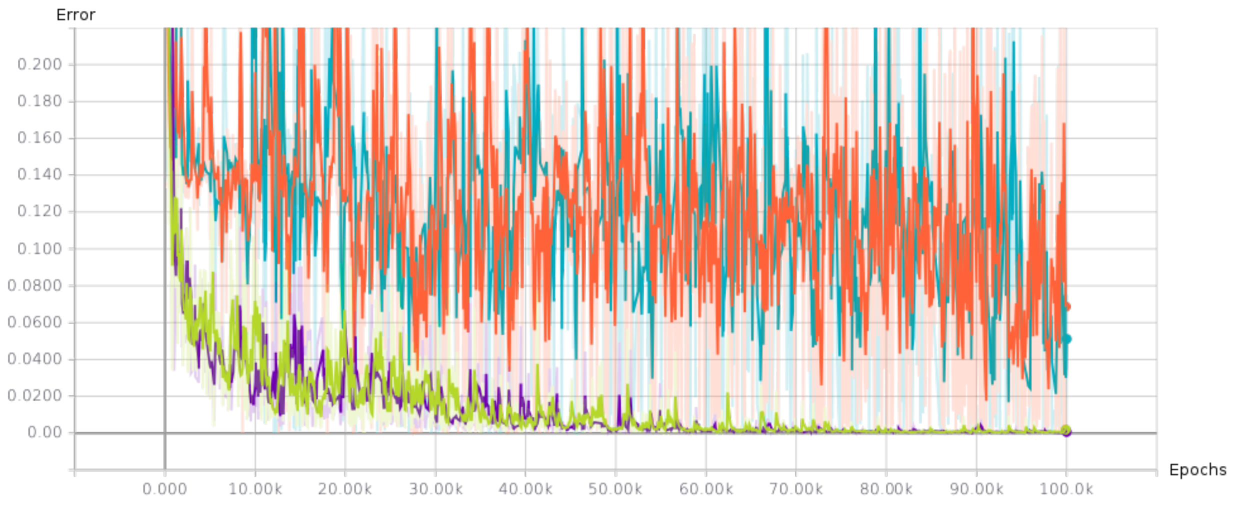

Figure 5.4: Both models associative recall task learning error plot for a 3×3 size sequence. 🟧 Neural Turing Machine test. 🟦 Neural Turing Machine train. 🟩 Differentiable Neural Computer test. 🟪 Differentiable Neural Computer train.

For the second sequences size(5×5), we can observe that the NTM and the DNC stability isn't really well, but, by the end of training epochs the DNC shows more stability and a lower error. At the end of training, the DNC testing error was $8.7321(10^{-3})$ and the training error was 0.02914 while the NTM reports a testing error of 0.0967 and a training error of 0.1164. Figure 5.5 shows the DNC and the NTM training phase error plots and its details.

For last sequences size(10×10), the stability looks better and at the training end, the DNC shows more stability and a lower error. At the training end, the DNC's testing error was 0.02122, and the training error was 0.02079, while the NTM reports a testing error of 0.05838, and a training error of 0.05706. Figure 5.6 shows the training phase error's plots of both models and its details.

5.9 Conclusion

As in the last experiment, the NTM parameters are lower than DNC parameters but the last one has, apparently, more stability and its convergence is faster. As before, despite the learning parameters, the

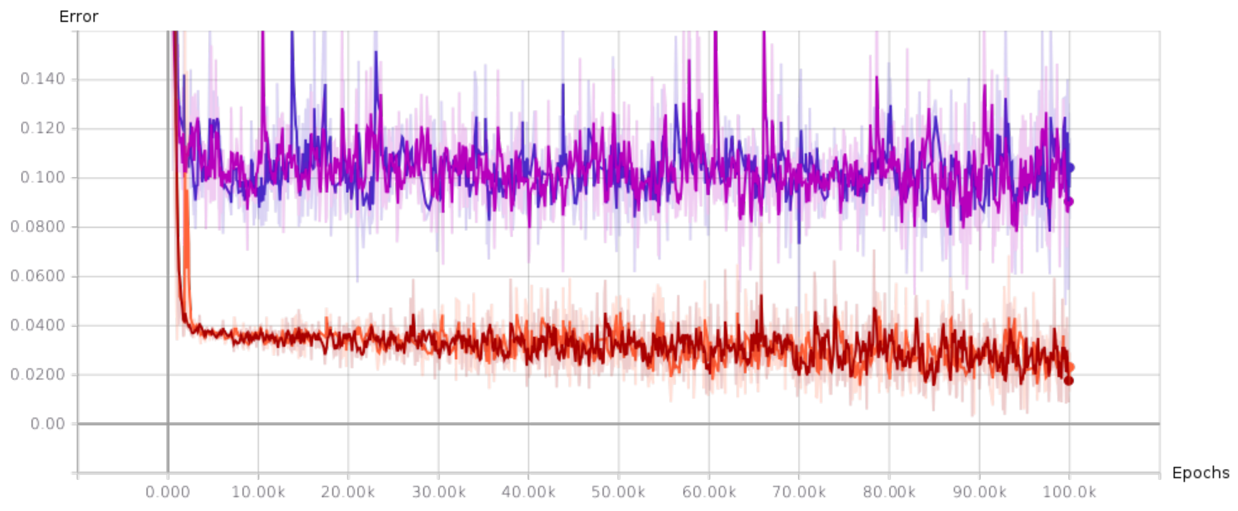

Figure 5.5: Both models associative recall task learning error plot for a 5×5 size sequence. 🟪 Neural Turing Machine test. 🟦 Neural Turing Machine train. 🟥 Differentiable Neural Computer test. 🟧 Differentiable Neural Computer train.

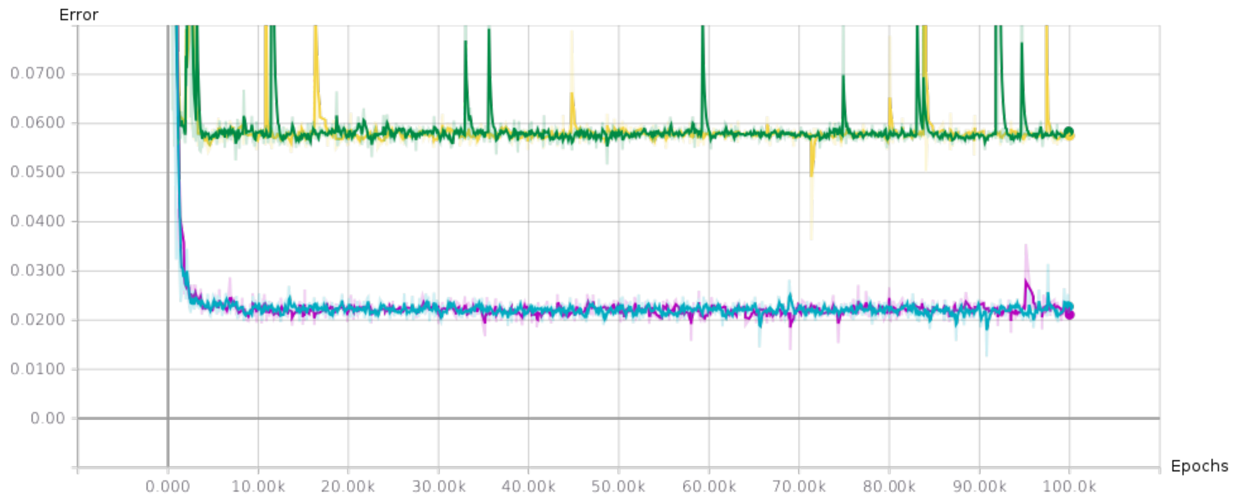

Figure 5.6: Both models associative recall task learning error plot for a 10×10 size sequence. 🟩 Neural Turing Machine test. 🟨 Neural Turing Machine train. 🟦 Differentiable Neural Computer test. 🟪 Differentiable Neural Computer train.

DNC's performance is much better, maybe modifying the model parameters the performance will be better, and the error will be lower, but is clear that the DNC's performance is better than the NTM's performance, so, in conclusion, the NTM improvements made for developing the DNC model were satisfactory.

5.10 Generalization with a feedforward controller

5.10.1 Goal

Prove that the DNC can generalize in the copy task, that is, if a bigger input is given, talking on vectors number, it still generating results with a low error rate.

5.10.2 Justification

One of the most important DNC's feature showed in Alex Graves' paper is the DNC's capability for generalizing results without retraining the full model. It is a very important result because a neural network can take a long training time, but if the DNC can generalize, cheaper resources are needed, and for solving big problems we can use smaller instances, what gives us a perfect model for solving a big problems variety.

5.10.3 Input

The inputs follow the same rules described before(in the copy task), but after training, in order to test the generalization, the inputs' size will be variated.

For copy task the tested inputs in this document are:

- Original instances: 5×5 sequences.
- Smaller instances: 3×5 sequences.
- Larger instances: 10×5 sequences.

The associative recall task cannot be generalized because while more vectors are needed either more bits.

5.10.4 Output

The expected output is the same as in the copy task.

5.11 Details

First, we trained a model for sequences of size 5×5, the learning rate was set to $1(10^{-4})$, and the momentum and decay were set to 0.9. The parameters number is 67348. The information vector size was 100 and the memory size was 100×40.

The training was done during 100000 epochs to get a low error rate.

5.12 Results

First, the training error was 0.01318 and the testing error was $6.4049(10^{-3})$. Figure 5.7 shows a training error plot.

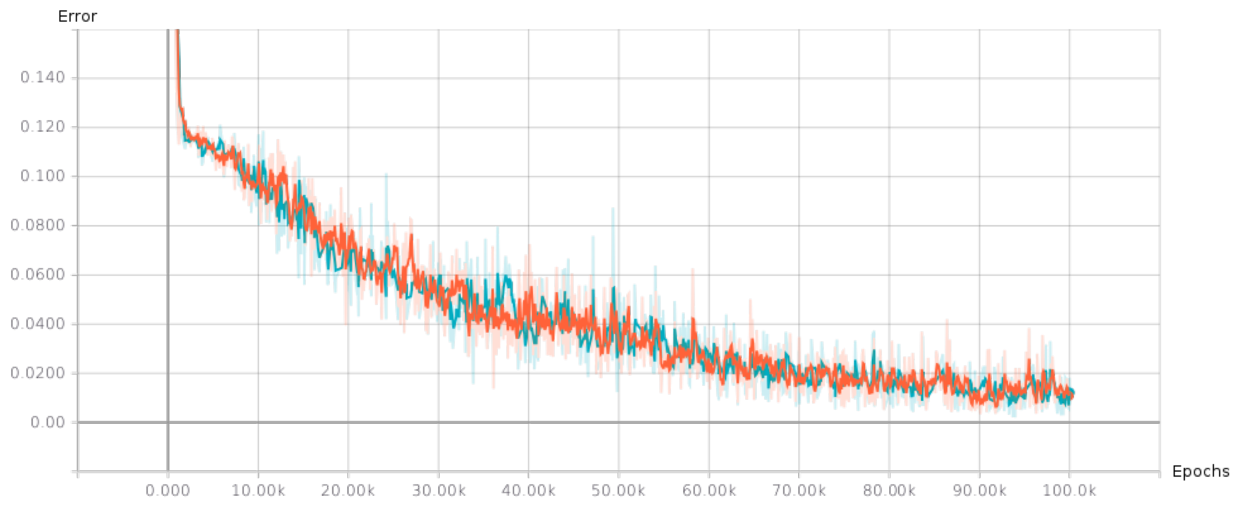

Figure 5.7: The copy task learning error plot for a 5×5 size sequence.
🟥 Differentiable Neural Computer test. 🟦 Differentiable Neural Computer train.

The first tested sequences were those of the size 5×5. I ran 1000 examples and of those tests, the obtained error was 0.011737. A rounded output example is:

$$\left[\begin{array}{ccccc|c} 1&1&0&0&1&0\\ 1&1&1&1&0&0\\ 0&0&0&0&0&0\\ 1&0&1&0&0&0\\ 1&0&0&1&0&0\\ 0&0&0&0&0&1\\ 0&0&0&0&0&0\\ 0&0&0&0&0&0\\ 0&0&0&0&0&0\\ 0&0&0&0&0&0\\ 0&0&0&0&0&0 \end{array}\right] \xrightarrow{target} \begin{bmatrix} 0&0&0&0&0\\ 0&0&0&0&0\\ 0&0&0&0&0\\ 0&0&0&0&0\\ 0&0&0&0&0\\ 0&0&0&0&0\\ 0&0&0&0&0\\ 1&1&0&0&1\\ 1&1&1&1&0\\ 0&0&0&0&0\\ 1&0&1&0&0\\ 1&0&0&1&0 \end{bmatrix} \xrightarrow{output} \begin{bmatrix} 0&0&0&0&0\\ 0&0&0&0&0\\ 0&0&0&0&0\\ 0&0&0&0&0\\ 0&0&0&0&0\\ 0&0&0&0&0\\ 0&0&0&0&0\\ 1&1&0&0&1\\ 1&1&1&1&0\\ 0&0&0&0&0\\ 1&0&1&0&0\\ 1&0&0&1&0 \end{bmatrix}$$

As we can observe, compared with what we want to obtain, the output is really close.

The next proved sequences were of size 3×5, where from 1000 examples, the obtained error was 0.213543. The error was too high but taking a look in the output it's easy to know why.

$$\left[\begin{array}{ccccc|c} 1&1&1&0&0&0\\ 0&1&0&0&0&0\\ 1&1&1&1&1&0\\ 0&0&0&0&0&1\\ 0&0&0&0&0&0\\ 0&0&0&0&0&0\\ 0&0&0&0&0&0 \end{array}\right] \xrightarrow{target} \begin{bmatrix} 0&0&0&0&0\\ 0&0&0&0&0\\ 0&0&0&0&0\\ 0&0&0&0&0\\ 1&1&1&0&0\\ 0&1&0&0&0\\ 1&1&1&1&1 \end{bmatrix} \xrightarrow{output} \begin{bmatrix} 0&0&0&0&0\\ 0&0&0&0&0\\ 0&0&0&0&0\\ 0&0&0&0&0\\ 0&0&0&0&0\\ 0&0&0&0&0\\ 1&1&1&0&0 \end{bmatrix}$$

The observed phenomena in this example(and what happens in almost all examples) is the out of phase output, that is, the zero padding is the same as in the original size(5) what causes a displaced output, it is, the vector that should be in the fourth row is now in the sixth one.

Now, proving the last sequences of size 10×5, with a sample of 1000 examples, we get an error of 0.356087. The error increases but a similar out of phase is observed.

$$\left[\begin{array}{ccccc|c}
0 & 0 & 0 & 0 & 1 & 0 \\
1 & 1 & 0 & 1 & 1 & 0 \\
0 & 1 & 0 & 0 & 0 & 0 \\
1 & 1 & 1 & 1 & 1 & 0 \\
1 & 0 & 1 & 0 & 1 & 0 \\
0 & 1 & 0 & 0 & 0 & 0 \\
1 & 1 & 0 & 1 & 1 & 0 \\
0 & 0 & 1 & 0 & 0 & 0 \\
1 & 0 & 0 & 0 & 0 & 0 \\
1 & 1 & 1 & 1 & 1 & 0 \\
0 & 0 & 0 & 0 & 0 & 1 \\
0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0
\end{array}\right]
\xrightarrow{target}
\begin{bmatrix}
0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 1 \\
1 & 1 & 0 & 1 & 1 \\
0 & 1 & 0 & 0 & 0 \\
1 & 1 & 1 & 1 & 1 \\
1 & 0 & 1 & 0 & 1 \\
0 & 1 & 0 & 0 & 0 \\
1 & 1 & 0 & 1 & 1 \\
0 & 0 & 1 & 0 & 0 \\
1 & 0 & 0 & 0 & 0 \\
1 & 1 & 1 & 1 & 1
\end{bmatrix}
\xrightarrow{output}
\begin{bmatrix}
0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 1 \\
1 & 1 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 \\
1 & 1 & 1 & 1 & 1 \\
1 & 1 & 1 & 0 & 1 \\
0 & 0 & 1 & 1 & 0 \\
0 & 1 & 1 & 1 & 1 \\
0 & 1 & 1 & 1 & 1 \\
0 & 1 & 0 & 0 & 0 \\
0 & 0 & 1 & 1 & 1 \\
0 & 0 & 1 & 1 & 0 \\
1 & 0 & 1 & 1 & 1 \\
0 & -1 & 1 & 1 & 1 \\
0 & -1 & 1 & 1 & 1 \\
0 & -1 & 1 & 1 & 1
\end{bmatrix}$$

In this case, the zero-padding continues as in the original output, what shows a displaced output but there isn't less information, so, we can adjust the padding for showing the needed output, being as below:

$$\left[\begin{array}{ccccc|c}
0 & 0 & 0 & 0 & 1 & 0 \\
1 & 1 & 0 & 1 & 1 & 0 \\
0 & 1 & 0 & 0 & 0 & 0 \\
1 & 1 & 1 & 1 & 1 & 0 \\
1 & 0 & 1 & 0 & 1 & 0 \\
0 & 1 & 0 & 0 & 0 & 0 \\
1 & 1 & 0 & 1 & 1 & 0 \\
0 & 0 & 1 & 0 & 0 & 0 \\
1 & 0 & 0 & 0 & 0 & 0 \\
1 & 1 & 1 & 1 & 1 & 0 \\
0 & 0 & 0 & 0 & 0 & 1 \\
0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0
\end{array}\right]
\xrightarrow{target}
\begin{bmatrix}
0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 1 \\
1 & 1 & 0 & 1 & 1 \\
0 & 1 & 0 & 0 & 0 \\
1 & 1 & 1 & 1 & 1 \\
1 & 0 & 1 & 0 & 1 \\
0 & 1 & 0 & 0 & 0 \\
1 & 1 & 0 & 1 & 1 \\
0 & 0 & 1 & 0 & 0 \\
1 & 0 & 0 & 0 & 0 \\
1 & 1 & 1 & 1 & 1
\end{bmatrix}
\xrightarrow{output}
\begin{bmatrix}
0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 1 \\
1 & 1 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 \\
1 & 1 & 1 & 1 & 1 \\
1 & 1 & 1 & 0 & 1 \\
0 & 0 & 1 & 1 & 0 \\
0 & 1 & 1 & 1 & 1 \\
0 & 1 & 1 & 1 & 1 \\
0 & 1 & 0 & 0 & 0 \\
0 & 0 & 1 & 1 & 1
\end{bmatrix}$$

Here we can appreciate the similitude between the target and the output, which is not the same but is an approximation. The error reduces, and the new error rate obtained is 0.123861.

5.13 Conclusion

The generalization is one of the most important model's feature, so, if we try to get it with a feedforward controller, the generalization is not well performed. The generalization gives a high error and a bad approximation. Maybe could be a good option if the solution accuracy is not very important, but maybe, using another controller type, or a more complex one, it could be better.

5.14 Generalization with an LSTM controller

5.14.1 Goal

Prove that the DNC's generalization is equal or better when using a Long Short-Term Memory controller instead of a feedforward controller.

5.14.2 Justification

As mentioned before, the generalization is an important DNC's model feature, so, for getting a better performance, given that in the last test we did not obtain the expected result, it's necessary developing the original Alex Graves' proposed model [4]. This will verify the model generalization, what can be useful for considering this model as an important tool when solving larger problems using small instances.

5.14.3 Input

The inputs follow the same rules described before(in the copy task), but after training, in order to test the generalization, the inputs' size will be variated.

For copy task the tested inputs in this document are:

- Original instances: 5×5 sequences.
- Smaller instances: 3×5 sequences.
- Larger instances: 10×5 sequences.

The associative recall task cannot be generalized because while more vectors are needed either more bits.

5.14.4 Output

The expected output is the same as in the copy task.

5.15 Details

First, we trained a model for sequences of size 5×5, the learning rate was set to $1(10^{-4})$, and the momentum and decay were set to 0.9. The parameters number is 786960. The information vector size was 100 and the memory size was 100×40.

The training was done for 100000 epochs to get a low error rate.

5.16 Results

First, the training error was $4.5965(10^{-6})$, and the testing error was $4.493(10^{-6})$. Figure 5.8 shows a training error plot.

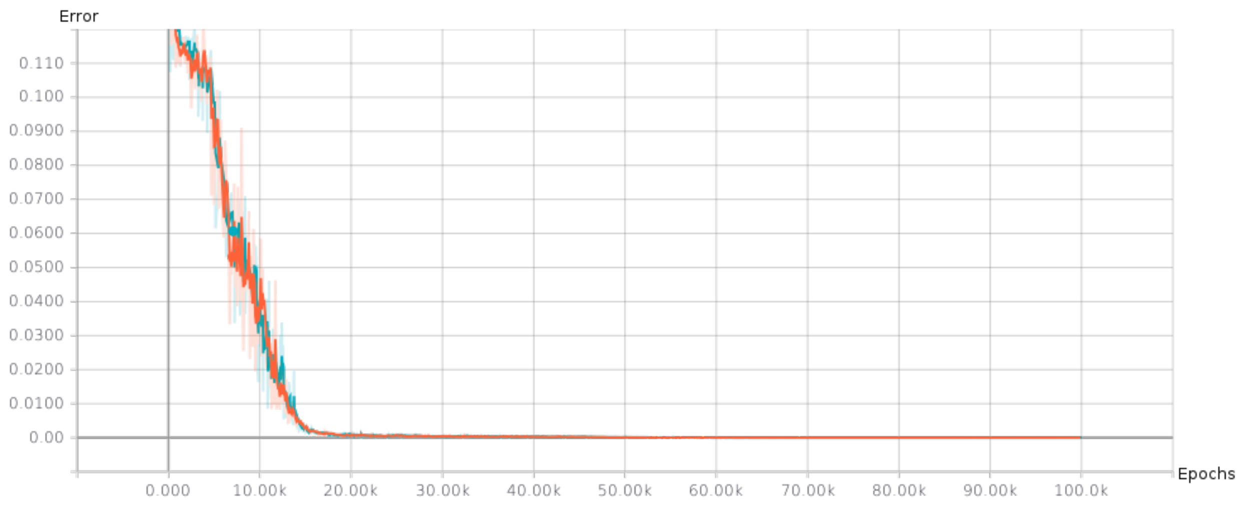

Figure 5.8: The copy task learning error plot for sequences of 5×5 size .
🟥 Differentiable Neural Computer test. 🟦 Differentiable Neural Computer train.

We can appreciate, that the convergence was much faster than feedforward controller, and the error is lower.

The first tested sequences were those of size 5×5. I ran 1000 examples and of those tests, the obtained error was $5.798107(10^{-6})$. I used the last test examples in order to get a good comparison between the feedforward and the LSTM controller. The first example output was the same in both controllers.

The next tested sequences were of size 3×10, where from 1000 examples the obtained error was 0.188272. The error was too high but if we take a look at the outputs structure, it is easy to observe why. The error was lower than using a feedforward controller but we can observe the same feedforward controller phenomena: The zero-padding is from the original size, so, some information was lost. The output is similar to the feedforward controller one.

Now, proving the last sequences of size 10×5, with a sample of 1000 examples, we get an error of 0.268362. The error increases but a similar out of phase is sxhowed.

$$\left[\begin{array}{ccccc|c}
0 & 0 & 0 & 0 & 1 & 0 \\
1 & 1 & 0 & 1 & 1 & 0 \\
0 & 1 & 0 & 0 & 0 & 0 \\
1 & 1 & 1 & 1 & 1 & 0 \\
1 & 0 & 1 & 0 & 1 & 0 \\
0 & 1 & 0 & 0 & 0 & 0 \\
1 & 1 & 0 & 1 & 1 & 0 \\
0 & 0 & 1 & 0 & 0 & 0 \\
1 & 0 & 0 & 0 & 0 & 0 \\
1 & 1 & 1 & 1 & 1 & 0 \\
0 & 0 & 0 & 0 & 0 & 1 \\
0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0
\end{array}\right]
\xrightarrow{target}
\left[\begin{array}{ccccc}
0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 1 \\
1 & 1 & 0 & 1 & 1 \\
0 & 1 & 0 & 0 & 0 \\
1 & 1 & 1 & 1 & 1 \\
1 & 0 & 1 & 0 & 1 \\
0 & 1 & 0 & 0 & 0 \\
1 & 1 & 0 & 1 & 1 \\
0 & 0 & 1 & 0 & 0 \\
1 & 0 & 0 & 0 & 0 \\
1 & 1 & 1 & 1 & 1
\end{array}\right]
\xrightarrow{output}
\left[\begin{array}{ccccc}
0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 \\
1 & 1 & 0 & 1 & 1 \\
0 & 1 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 1 \\
0 & 1 & 0 & 0 & 0 \\
1 & 1 & 0 & 1 & 1 \\
0 & 0 & 1 & 0 & 0 \\
1 & 0 & 0 & 0 & 0 \\
1 & 1 & 1 & 1 & 1 \\
0 & 0 & 0 & 0 & 1 \\
0 & 0 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0
\end{array}\right]$$

In this case, the zero-padding continues as in the original output, what produces a displaced output, but there isn't less information, so, we can adjust the padding to show the needed output, being as below:

$$\left[\begin{array}{ccccc|c}
0 & 0 & 0 & 0 & 1 & 0 \\
1 & 1 & 0 & 1 & 1 & 0 \\
0 & 1 & 0 & 0 & 0 & 0 \\
1 & 1 & 1 & 1 & 1 & 0 \\
1 & 0 & 1 & 0 & 1 & 0 \\
0 & 1 & 0 & 0 & 0 & 0 \\
1 & 1 & 0 & 1 & 1 & 0 \\
0 & 0 & 1 & 0 & 0 & 0 \\
1 & 0 & 0 & 0 & 0 & 0 \\
1 & 1 & 1 & 1 & 1 & 0 \\
0 & 0 & 0 & 0 & 0 & 1 \\
0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0
\end{array}\right]
\xrightarrow{target}
\begin{bmatrix}
0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 1 \\
1 & 1 & 0 & 1 & 1 \\
0 & 1 & 0 & 0 & 0 \\
1 & 1 & 1 & 1 & 1 \\
1 & 0 & 1 & 0 & 1 \\
0 & 1 & 0 & 0 & 0 \\
1 & 1 & 0 & 1 & 1 \\
0 & 0 & 1 & 0 & 0 \\
1 & 0 & 0 & 0 & 0 \\
1 & 1 & 1 & 1 & 1
\end{bmatrix}
\xrightarrow{output}
\begin{bmatrix}
0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 \\
1 & 1 & 0 & 1 & 1 \\
0 & 1 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 1 \\
0 & 1 & 0 & 0 & 0 \\
1 & 1 & 0 & 1 & 1 \\
0 & 0 & 1 & 0 & 0 \\
1 & 0 & 0 & 0 & 0 \\
1 & 1 & 1 & 1 & 1
\end{bmatrix}$$

Here we can appreciate the similitude between the target and the output, which is a really well approximation. Here, the lost information is lower than in the feedforward controller. Despite the error, what in part was caused by the incorrect padding, the approximation is very precise. The error reduces and the new error rate obtained is 0.036332.

5.17 Conclusion

The generalization is a really useful property and a powerful one, depending, in a big part, in the controller type or the controller capabilities. When using an LSTM controller the model capabilities increases beside the feedforward controller, so, the controller can be considered as a really important model's part, but it is more complex, then depending on our goal, the controller must be selected carefully. Finally, about the generalization, is easy to observe that it can be obtained by selecting carefully the model parameters and the model modules, and despite it is a very unprecise output, it approximates really good when using the LSTM controller.

5.18 Stability

5.18.1 Goal

Prove the DNC's stability over multiple training instances.

5.18.2 Justification

The stability is an important feature of any machine learning model, given that it warranties that won't care how many times we train or test the model on a specific problem, the result will be consistent.

5.18.3 Details

Using the copy task with the input sequences of 3×5 size, and the feedforward controller and the LSTM controller, we will test the model's stability by training it three times.

5.19 Results

Using the feedforward controller we get the plot shown in figure 5.9. There, we can observe that the error shown on the three training times, is close one of each other, so, using a feedforward controller will provide us stable results.

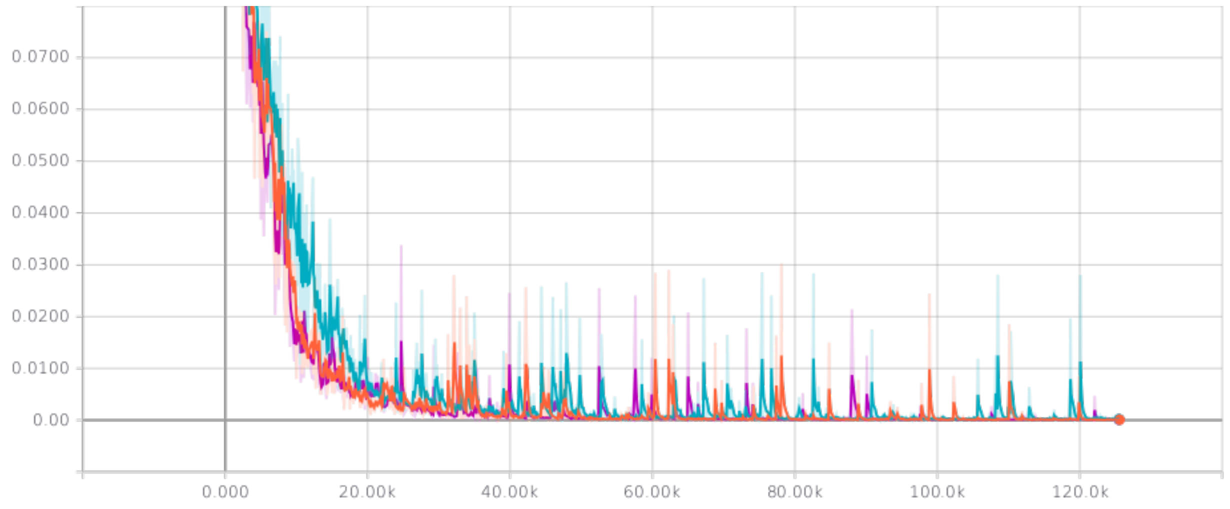

Figure 5.9: Feed-forward controller model's error trained three times with sequences of 3×5 size.

As second test, we proved the LSTM controller, the results are shown in figure 5.10. There, we can observe that the error on three training times, is close one of each other, a faster convergence is maintained, then we can conclude that using an LSTM controller will provide us stable results also.

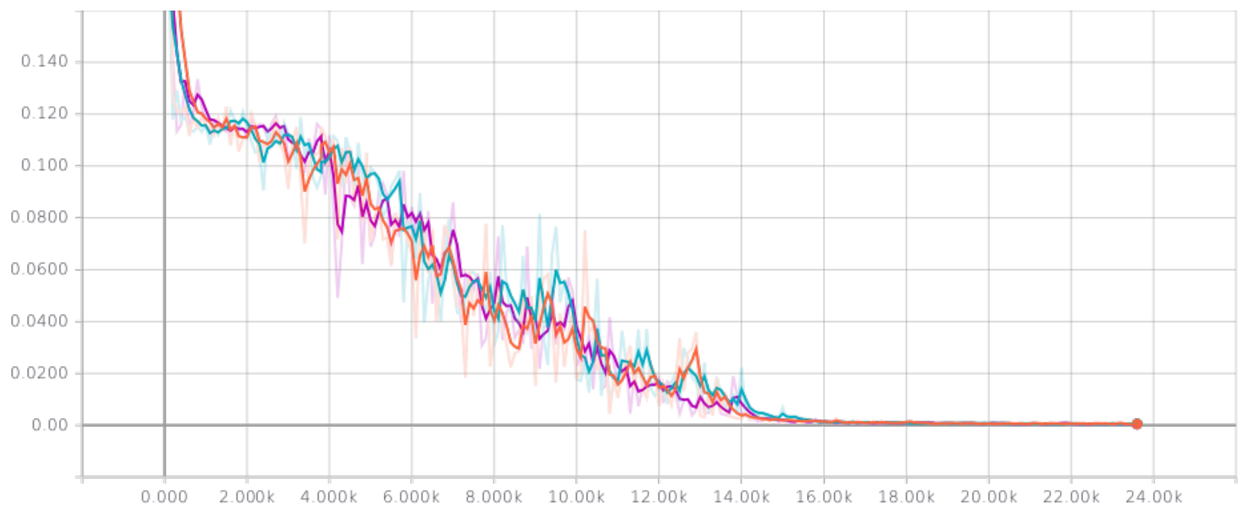

Figure 5.10: LSTM controller model's error trained three times with sequences of 3×5 size.

5.20 Conclusion

The stability in this model is not a problem because the results showed before provide us a strong confidence on the consistence of its results. Given that, we can also conclude, no matter how many times we run previous test, they are trustworthy.

Chapter 6

Conclusion

In 2016 Alex Graves' article was published [4], there, he introduces a model that supposed to be very powerful and with new features. It was his previous model, Neural Turing Machine, evolution; the Differentiable Neural Computer. It is a neural network model extended with an external memory, and using designed methods, the model can learn how to use it for running algorithmic tasks.

In this thesis, we verify the main DNC's properties presented in Alex Graves' article.

Here we test the model's features over multiple tasks. In order to do this, we implemented a Differentiable Neural Computer(available in `https://github.com/cobu93/DNC`). After multiple tests, we conclude that all claimed features are applicable, and choosing the correct model's configuration and meta-parameters, good results were obtained.

The main feature is that all used modules can be interchanged depending on what we want to do. We can adapt the addressing mechanisms, the controller, the output function, etcetera because the main idea is offering a general architecture and not a strict model for solving problems. With this principle, is important to choose the correct controller because, our tests showed, that big differences occur when variating this component.

The LSTM controller gave us the most important model's feature: Generalization. Generalization means that once we had trained the model with specific inputs and outputs, we can solve similar problems with different inputs and outputs sizes without retraining. It is here where it's important to choose the correct components depending on our problem. Additionally, when this controller is selected, the convergence was much faster.

So, this showed us that all claimed characteristics are ascertainable, and how we expected, is necessary a correct model's configuration in order to get the desired results. The DNC is a very powerful model, it can be used for algorithmic tasks or reinforcement learning, the added memory is very useful for maintaining information in long time periods and the addressing mechanisms were well adapted because they made an easier understanding.

6.1 Contribution

The contributions done with this study were:

- A detailed description of Differentiable Neural Computer, and how it works based on existent given information and experimental results.
- A comparison between DNC and NTM.
- A task set of different sizes used for evaluating the model performance.
- An own Neural Turing Machine implementation.
- A demonstration of DNC functionality when solving algorithmic tasks.
- An own Differentiable Neural Computer implementation using Python and Tensorflow.
- A DNC's feedforward and LSTM controller comparison.
- A proof of DNC generalization.

6.2 Future work

An interesting and discussed architecture future work, is its modular use when solving bigger problems, especially in those problems where a deterministic solution is hard or impossible. Imagine the generalization applied on those problems where their solving complexity increases exponentially with the input size; Could be enough training the model using small instances and get solutions for bigger ones! It would be very nice.

In addition, the DNC's model was tested on multiple tasks using structured inputs. The obtained results demonstrate, the architecture success when storing and managing structured inputs, so, for those problems that can be expressed in a general structure, it is a great advantage because it is enough training the model with a simple problem and applies the solution in equivalent problems.

Finally, a free implementation was provided. Everybody is free for testing, modifying or contributing in the given code!.

Bibliography

[1] A. e. a. Graves, "Neural turing machines," *Google DeepMind*, 2014.

[2] W. Zaremba and I. Sutskever, "Learning to execute," *ICLR*, 2015.

[3] O. Gutierrez, "Aprendizaje de tareas algorítmicas con máquinas de turing neuronales," Ph.D. dissertation, Centro de Investigación en Computación, 2017.

[4] A. e. a. Graves, "Hybrid computing using a neural network with dynamic external memory," *Nature*, vol. 538, pp. 471 – 476, 2016.

[5] R. Bangia, *Computer Fundamentals and Information Technology*, 1st ed. Firewall Media, 2008.

[6] E. Kleinberg, J.; Tardos, *Algorithm Design*, 1st ed. Pearson, 2005.

[7] T. Gonzalez, *Handbook of Approximation Algorithms and Metaheuristics*. Chapman & Hall/CRC, 2007.

[8] W. McCulloch, W.; Pitts, "A logical calculus of the ideas immanent in nervous activity," *Bulletin of Mathematical Biophysics*, vol. 5, pp. 115 – 133, 1943.

[9] D. Hebb, *The organization of behavior: A neuropsychological theory*. Taylor and Francis, 2009.

[10] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning (Adaptive Computation and Machine Learning series)*. Massachusetts: The MIT Press., 2016.

[11] M. Bishop, *Pattern Recognition and Machine Learning*. Springer, 2006.

[12] A. Géron, *Hands-On Machine Learning with Scikit-Learn and TensorFlow*. O'Reilly, 2017.