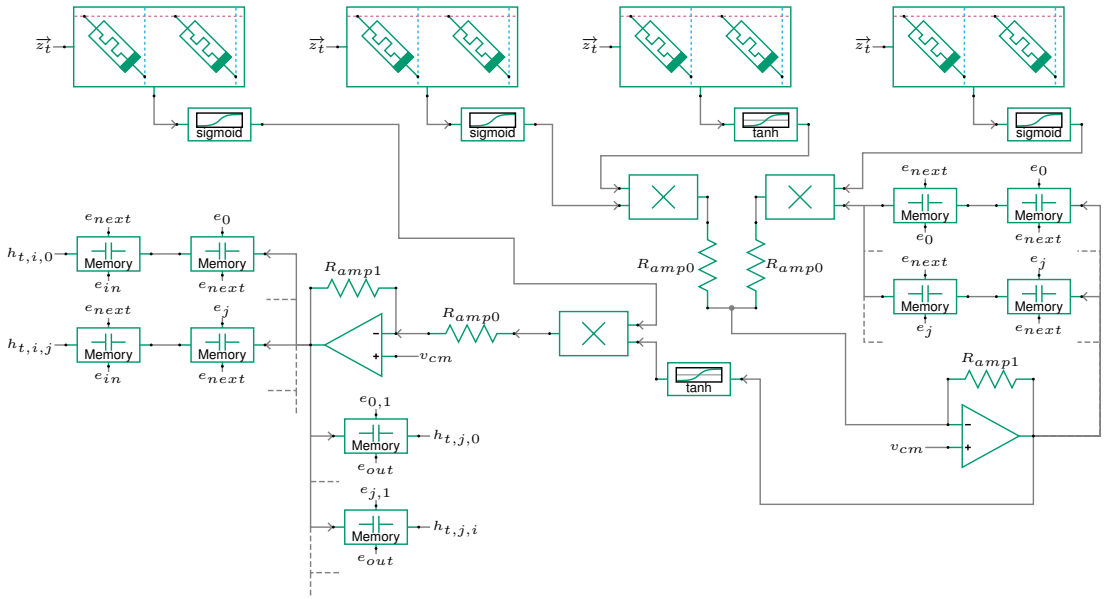
TÉCNICO LISBOA



Memristors-based recurrent modules for neural computing

Valentin BARBAZA

Thesis to obtain the Master of Science Degree in

Electrical and Computer Engineering

Supervisors: Dr. Diogo Miguel Bárbara Coroas Prista Caetano
Dr. Ruxandra Georgeta Barbulescu

Examinatiom Committee

Chairperson: Prof. Teresa Maria Canavarro Menéres Mendes de Almeida
Supervisor: Dr. Diogo Miguel Bárbara Coroas Prista Caetano
Members of the Committe: Dr. Paulo Ferreira Godinho Flores

November 2023

Declaration

I declare that this document is an original work of my own authorship and that it fulfills all the requirements of the Code of Conduct and Good Practices of the Universidade de Lisboa.

Life is really simple, but we insist on making it complicated.

Confucius

Acknowledgments

I would like to thank the Instituto Superior Tecnico (IST) for allowing to join their double degree program, that made it possible for me to widen my computer knowledge from computer science to electronics. I would also like to thank Instituto de Engenharia de Sistemas e Computadores (INESC) to have let me join their rank, to both INESC-ID and INESC-MN groups to have given the necessary tools to succeed with this thesis. I would also like to thank all the other student working in the ASIC (Advanced Sensor Interfaces and Circuits) group with whom I could share problems and were happy to help. I would lastly like to thank Dr Ruxandra Barbulescu and Dr Diogo Caetano, to have trusted me with this very interesting topic despite my recent arrival in the university.

Abstract

In this work we propose an analog structure for memristor based recurrent modules targeting neural computing. The system is fully analog and implements a working Long Short-Term Memory (LSTM) circuit block and a work in progress Gated Recurrent Unit (GRU) circuit block. Both of those blocks contain memristors to be used as weights in a analog Vector Matrix Multiplication (VMM) capable circuit. These circuit blocks allow to run very fast computation of Recurrent Neural Networks of any size, in a relatively small integrated circuit. As part of the LSTM and GRU blocks, an analog activation function circuit was designed. This specific circuit is capable of reproducing sigmoid and hyperbolic tangent (tanh) like functions, with similar shapes and the same output ranges. The work also include the implementation of a memory cell used to store an analog value for a short period of time. The LSTM block can be serialized or not with the ability to choose the level of serialization. Serializing the system allows to save onChip area at the cost of execution time. To the author's knwoledge this is the first analog implementation of the behavior of C. elegans using the LSTM block. Such an analog system provides ground for real time implementation of nervous systems.

Keywords

Analog neural computing, Embedded neural computing, Memristor-based recurrent modules, Analog LSTM, Analog GRU

Resumo

Neste trabalho, propomos uma estrutura analógica para módulos recorrentes baseados em memristors para computação neuromórfica. O sistema é totalmente analógico e implementa um de circuito LSTM funcional bem como um circuito GRU, ainda em desenvolvimento. Ambos os blocos contêm memristors para serem usados como pesos em um circuito VMM analógico capaz. Esses blocos permitem a execução de cálculos necessários para Redes Neurais Recorrentes de dimensão arbitrária, e de forma rapida e eficiente, em circuito integrado. Como parte dos blocos LSTM e GRU, foi projetado um circuito analógico que implementa funções de ativação. Este circuito específico é capaz de reproduzir funções semelhantes às sigmoid e tanh, com formas e intervalos de valor similares. O trabalho também inclui a implementação de uma célula de memória usada para armazenar um valor analógico por um curto período de tempo. Os cálculos do bloco LSTM podem ser executado em série ou paralelo, com um grau de serialização arbitrário. A serialização do sistema permite economizar área no chip, a custo do tempo de execução. Adicionalmente, até à data de publicação esta é a primeira implementação analógica do comportamento do Caenorhabditis elegans (C. elegans) usando o bloco LSTM em circuito integrado. Um sistema analógico deste tipo estabelece a base para a implementação em tempo real de sistemas nervosos.

Palavras chave

Computação neural analógica, Computação neural embarcada, Módulos recorrentes baseados em memristores, LSTM analógica, GRU analógica

Contents

List of Tables

List of Figures

Abbreviations

AI Artificial Intelligence

ASIC Application-Specific Integrated Circuit

C. elegans Caenorhabditis elegans

CIFG Coupled Input-Forget Gate

CMOS Complementary metal-oxide-semiconductor

CPU Central Processing Unit

FCM Forward Crawling Motion

FGR Full Gate Recurrence

FPGA Field Programmable Gate Array

GPT Generative Pre-trained Transformer

GPU Graphics Processing Unit

GRU Gated Recurrent Unit

INESC Instituto de Engenharia de Sistemas e Computadores

IST Instituto Superior Tecnico

LIDM Linear Ion Drift Model

LLaMA Large Language Model Meta AI

LLM Large Language Model

LSTM Long Short-Term Memory

MSE Mean Square Error

NFG No Forget Gate

NIAF No Input Activation Function

NIG No Input Gate

NN Neural Network

NOAF No Output Activation Function

NOG No Output Gate

NP No Peephole

opAmp operational amplifier

RMSE Root Mean Square Error

RNN Recurrent Neural Network

STBM Simmons Tunnel Barrier Model

tanh hyperbolic tangent

TEAM ThrEshold Adaptive Memristor Model

VMM Vector Matrix Multiplication

VR Virtual Reality

VTEAM Voltage ThrEshold Adaptive Memristor Model

List of Symbols

$\odot$	Point wise multiplication or Hadamart product
$\overrightarrow{1}$	Unit vector, containing only ones
A	Area
$\overrightarrow{b}$	Neural Network bias vector
$\overrightarrow{c}$	LSTM cell state vector
C	Electrical capacitance
$\overrightarrow{\hat{c}}$	LSTM candidate cell state vector
$\overrightarrow{f}$	LSTM forget gate vector
G	Electrical conductance
$\overrightarrow{h}$	Recurrent Neural Network hidden state vector
$\overrightarrow{ch}$	GRU candidate hidden state vector
H	Neural Network hidden perceptron
i	Electrical current
$\overrightarrow{i}$	LSTM input gate vector
I	Neural Network input perceptron
L	Electrical inductance
M	Memristor's internal resistance, memristance
$\overrightarrow{o}$	LSTM output gate vector
O	Neural Network output perceptron
$\overrightarrow{p}$	LSTM peephole weight vector
q	Electrical charge
$\overrightarrow{r}$	GRU reset gate vector
R	Electrical resistance
t	Time
v	Electrical voltage
w	Neural Network weight
W	Neural Network weight matrix
$\overrightarrow{x}$	Neural Network input vector
$\overrightarrow{z}$	GRU update gate vector
ϕ	Electrical flux
κ	Memristor's internal conductance, memductance

1

Introduction

Contents

1.1 Motivation

Nowadays, it is impossible not to have heard about Artificial Intelligences (AIs), they are everywhere and everybody talks about them. It is common practice to advertise products using AI powered tools. *ChatGPT* from **OpenAI** is a recent example and is getting very popular even among casual computer users. Everyone is using this tool. An older example include home assistants that have been around for about ten years. Those products (such as **Amazon**'s *Alexa*, **Google**'s *nest* system and others) are not computing their answers locally, hence creating avoidable internet trafic. It is clear that AI is becoming the predominant technology in the modern world and future innovations will depend on it.

AIs are very power hungry algorithms which limits their use in embedded systems and power efficient devices. Furthermore, the time an AI algorithm takes to compute its output quickly rises, especially on lower end chip such as the ones used in embedded systems. So much so that the most complex ones only run on online and very powerful servers, like the aforementioned *ChatGPT* and home assistants.

There are several options to reduce execution speed and energy consumption such as running the algorithm on a Graphics Processing Unit (GPU) rather than on a Central Processing Unit (CPU). Using a Field Programmable Gate Array (FPGA) or Application-Specific Integrated Circuit (ASIC) are other ways to improve power consumption and execution time. The latter is the most restricting, as it is name implies, but has the best results.

Using an ASIC allows the use of analog computation. Analog computation offers great advantages compared to digital computers as it is much faster while being known to be very power-efficient.

Home assistants could greatly benefit from being able to quickly compute answers without having to connect to an online server. Locally computing answers would drastically improve the usefulness of the device. It could even allow to be used on the go, in a car for example.

Another great use for local AI computation would be to use Large Language Models (LLMs), the technology behind *ChatGPT*, for any local use. LLMs are not the focus of this thesis, but are a long term potential objective of using analog computation for AI.

Making analog chips with a very low power consumption also allows to use them in embedded systems such as video surveillance cameras. A small chip could be installed inside the system that could be able to do gait detection [4–6]. The video surveillance camera would then be able to know who is showing up on the camera and if it is not a known person send an alert.

Such a chip could also be used for video stabilization [7]. It would allow to directly store stabilized video, and not have to use online servers like described in [7]. Any action camera would benefit greatly from such an improvement since they are mostly used to film scenes with lots of movements.

Another use for AI that works great in embedded use is for camera enhanced 3D camera localization [8]. That is useful especially for Virtual Reality (VR) headsets that need to know their location in the room. The embedded chip would allow this computation to be much faster than other methods, and thus give more realistic results.

Some research [9], focuses on reproducing parts of the nervous system of a small organism. This could be applied on larger organisms, or, in the long term, reproduce part of the human brain. Using analog computers, small chips could be used to replace a part of one's brain non functional part.

The focus of this thesis is to create a working simulation of an analog Long Short-Term Memory (LSTM) and Gated Recurrent Unit (GRU) circuit using **Cadence**'s *Virtuoso* as simulation software. Having those preliminary results would allow to fabricate and test such a chip. Once a working prototype is manufactured, such a circuit could highly improve embedded AI use.

1.2 Main Contributions

The thesis bring contributions to the following points:

- I created a script that can generate spice netlists for LSTM or feed forward Neural Networks of any size and importing weights as resistance values from a specified file containing the architecture and the weights.
- The script can generate an LSTM capable analog circuits and then run a simulation of the circuit in *Virtuoso* giving encouraging results.
- The script can also generate an GRU capable analog circuits and then run a simulation of the circuit in *Virtuoso*.
- I developed an analog activation function circuit that supports hyperbolic tangent (tanh) and sigmoid ranges working at $1.8V$.
- I recreated a Vector Matrix Multiplications (VMMs) capable analog circuit using a matrix of memristors.
- I designed a memory cells used to store analog data for a short amount of time, they are used the LSTM and GRU circuits.
- I coded in python [10] (tensorflow library[11]) to create a piece wise functions to reproduce in the tensorflow the analog activation functions.
- I used **Cadence**'s *Virtuoso* to run software simulation of different Neural Networks (NNs) capable analog circuits.
- A paper is being prepared to report the advancements made in this work.

1.3 Thesis Outline

This thesis is going to tackle the work that went into creating a software simulation of an analog computer capable of running Recurrent Neural Networks.

In chapter 2, all the technologies used and thus required to know in order to understand the thesis are detailed in the state of the art.

Next, in chapter 3, the work that was done for the thesis is going to be thoroughly explained. Starting with the different circuit that were designed for the thesis. Then the different scripts and python code that were created, either to run the digital NNs or to generate the analog system into a netlist.

Then chapter 4 contains the results, obtained from the digital inference and the analog simulation, from both the number of airline passenger and the Caenorhabditis elegans (C. elegans) datasets.

Finally, in chapter 5, the conclusions and improvements that can be made to the different parts of the thesis.

2

State of the art

Contents

2.1 Artificial Intelligence

Artificial Intelligence (AI) is very popular, and is getting more attention, being used as selling point by lots of services, the word is thrown around so much that it has lost its meaning. So let us see what are AIs exactly.

AI is defined as theories and strategies used to simulate human intelligence, in other words, a computer or software that can take a decision by itself based on previously set rules. The definition is very wide and has lots of ways to be implemented.

The most promising type of AI is machine learning, which in a broad sense means teaching the software to make decisions. It is defined as the use and development of computer systems that are able to learn and adapt without following explicit instructions, by using algorithms and statistical models to analyse and draw inferences from patterns in data.

In the last 10 years, the research concerning machine learning has considerably increased.

Figure 2.1: Machine learning and its subsets

Figure 2.1 is a visual representation of machine learning and all the disciplines of machine learning. The thesis will focus, as the title implies, on the Recurrent Neural Networks.

2.2 Neural Networks

Neural Networks (NNs) are a set of units known as neurons. Those neurons are linked to each other with arcs known as synapses, thoses synapses each have a weight associated to them. The set of neurons interconnected with their synapses is what is called a Neural Network. Figure 2.2, shows a simple representation of a NN, the artificial neurons are the represented by the colored circles. On figure 2.2 each arrow represent a synapse.

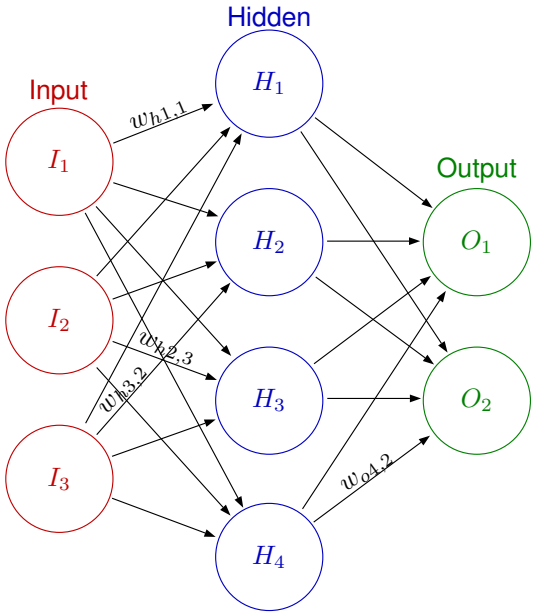


Figure 2.2: Simple Neural Network

NNs contains several layers :

- Input layer : This layer is simply the different inputs.
- Hidden layer : This layer can be (and usually is) wider than the one in figure 2.2. This is the layer that can be modified the most, by adding layers or increasing the amount of neurons in a layer.
- Output layer : This layer is where you can find the result from the NN.

The weights of the synapses have to be multiplied with the previous neuron and then added to each other to produce the next stage. Using the names defined in figure 2.2, the output is linked to the input by equations (2.1) and (2.2). First, the hidden layers' neurons need to be computed (equation (2.1)).

$$\begin{bmatrix} H_1 \\ H_2 \\ H_3 \\ H_4 \end{bmatrix} = \begin{bmatrix} w_{h1,1} & w_{h1,2} & w_{h1,3} \\ w_{h2,1} & w_{h2,2} & w_{h2,3} \\ w_{h3,1} & w_{h3,2} & w_{h3,3} \\ w_{h4,1} & w_{h4,2} & w_{h4,3} \end{bmatrix} \cdot \begin{bmatrix} I_1 \\ I_2 \\ I_3 \end{bmatrix} \tag{2.1}$$

Similarly the output is computed like in equation (2.2)

$$\begin{bmatrix} O_1 \\ O_2 \end{bmatrix} = \begin{bmatrix} w_{o1,1} & w_{o1,2} & w_{o1,3} & w_{o1,4} \\ w_{o2,1} & w_{o2,2} & w_{o2,3} & w_{o2,4} \end{bmatrix} . \begin{bmatrix} H_1 \\ H_2 \\ H_3 \\ H_4 \end{bmatrix} \tag{2.2}$$

Those matrix multiplication are called VMM because it is the result of the multiplication of a vector and a matrix, thus giving us another vector.

The model presented in figure 2.2 is modular and can be scaled up as much as required. It is generally accepted that the more neurons a NN has, the more complex the problems it can solve can be.

The weights values are obtained through training. In supervised learning, in order train the weights it is required to have a dataset of inputs/outputs values that are correct. The outputs are the target values, the values that the NN needs to compute when being executed. The training starts once the weights have been initialized (usually randomly picked values). The training is an iteravtive process, each iteration being called an epoch. Each of those epoch usally consists of the following steps :

1. Run the NN to get an output vector.
2. Measure the error (known as loss) by comparing the current prediction with the targeted output using the chosen error algorithm (Root Mean Square Error (RMSE), Mean Square Error (MSE), etc).
3. Run the backpropagation algorithm to change each individual weight.

The number of epochs required depends on the complexity of the problem the NN is solving.

Once trained, the NN can be tested by feeding the NN input data it has not seen before.

2.3 Recurrent Neural Network

Recurrent Neural Networks (RNNs) are, as the name suggests, a type of NN using recurrent connections. They are a NN with at least one cycle within the structure, outputs of previous time step are used as input for the next time step, these outputs are generally known are hidden states. Those feedback connections are the main difference from feedforward NN.

This type of NN is used when dealing with an unknown amount of inputs. Especially useful when treating time series [12]. Example of RNNs uses are speech recognition, automatic language translation [13] and shape recognition, especially for handwriting recognition.

Traditionnal RNNs have the ability to model sequential events by propagating through time, for example forward and backward propagation. This is achieved by connecting these sequential events with the hidden state like in equation (2.3).

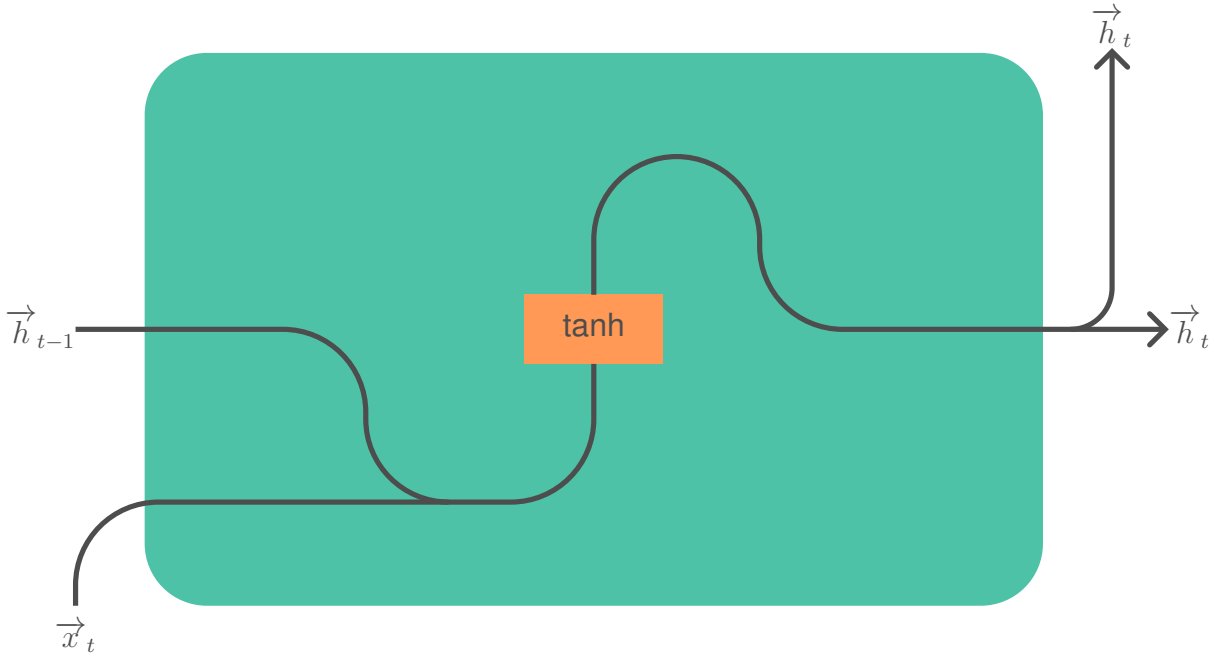
$$\overrightarrow{h}_t = f(\overrightarrow{x}_t, \overrightarrow{h}_{t-1}) \tag{2.3}$$

The hidden state ($\overrightarrow{h}_t$) carries all the past informations for the next time step. It also serves as the output of the RNN.

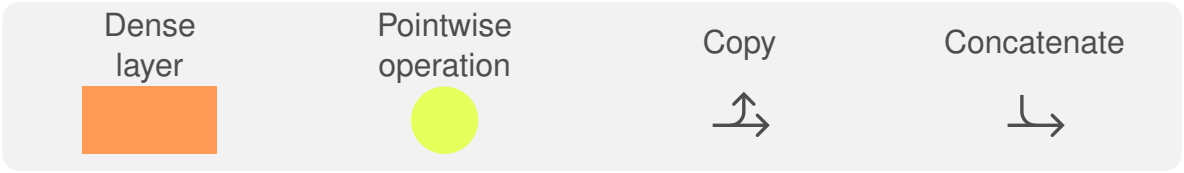
They are trained the same way NN are, measuring the error, backpropagating and adjusting the weights accordingly.

2.3.1 Simple Recurrent Neural Network

The simple RNN works just like a tanh activated feedforward NN with a feedback connection.



(a) RNN cell



(b) Legend

Figure 2.3

Equation (2.4) shows the equation that the simple RNN runs at every time step.

$$\overrightarrow{h}_t = tanh([\overrightarrow{x}_t, \overrightarrow{h}_{t-1}] \cdot W + \overrightarrow{b}) \tag{2.4}$$

Where $t \in \mathbb{N}^*$ is the time index, W the weight matrix, $\overrightarrow{b}$ the bias vector, $\overrightarrow{x}$ the input vector and $\overrightarrow{h}$ the hidden state of the RNN.

2.3.2 Vanishing gradient problem

The Vanishing gradient problem is a problem that comes when dealing with time dependent data [14]. When a big amount of time dependent data is fed to the RNN, the weights cannot be updated properly. The older the data, the lower it will impact how much the weight must change. Rendering

the old input data almost useless. That is the reasom why, simple RNNs must be used with relatively short time series.

Some RNNs were designed to tackle this issue. This is the case of the LSTM and GRU which were created with internal mechanisms to regulate the flow of information and gradients.

2.4 Long Short-Term Memory

Long Short-Term Memorys (LSTMs) are a type of RNN used to analyze sequences of data. They are capable of predicting data based on previous data points.

The first LSTM was invented in 1997 by Hochreiter and Schmidhuber [15]. LSTMs changed a lot through the years to become what they are now [16].

LSTMs was created to fix the vanishing gradient problem. LSTMs alleviate this issue by adding a cell state, this state gives it the ability to choose what input is important and which one is not, thus giving it a long term memory. This ability gave the uncommon name of Long Short-Term Memory as it has both long and short term memory. This is what makes LSTMs adequate for sequence data. They can analyze the data and keep the information from the last time step to make a better decision afterwards. The most comprehensible example is considering a sentence.

An LSTM is more complicated than just a simple feedforward Neural Network, it has several gates, which all compute a VMM. There is also a cell state whose job is to hold a value for the next step.

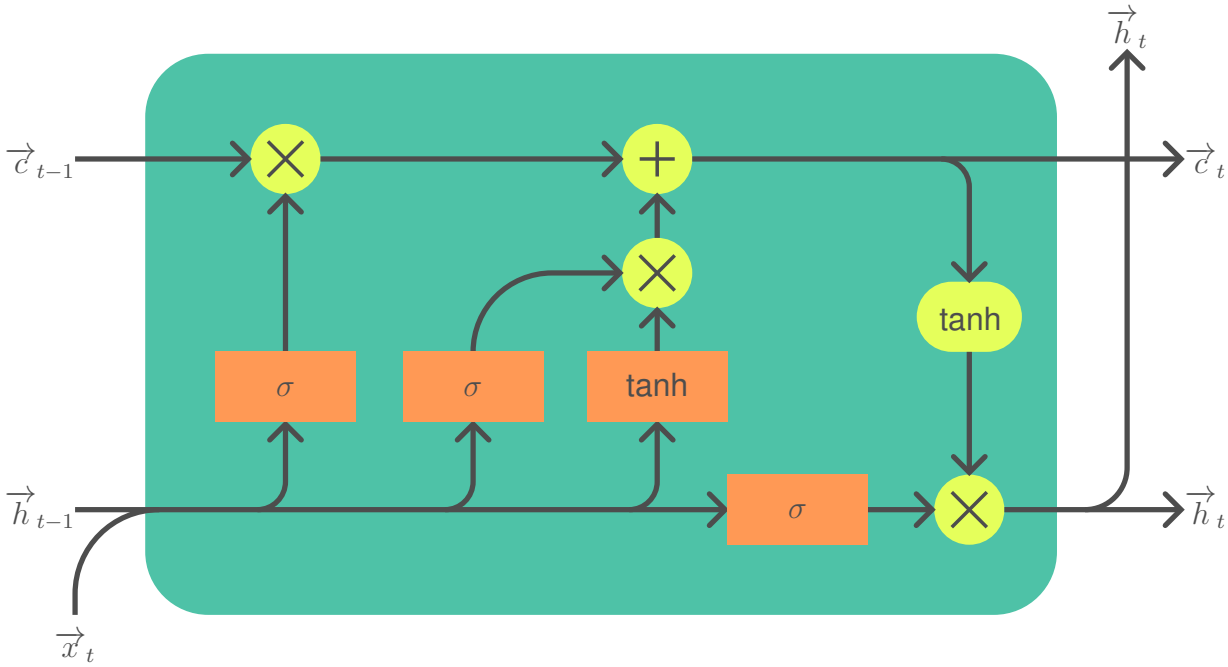


Figure 2.4: LSTM cell, adapted from [2]

Figure 2.4 shows the complexity of the LSTM architecture. In an LSTM, each gate is a different NN and then acivated with either a tanh or a sigmoid activation function. Each input to the cell is a vector. Those vectors are of varying sizes depending on several factors. For example, both h_t and c_t are of the same size as the number of hidden states (sometimes referred to as cell state) for any $t \geq 0$. The input vector (x_t) is of size of the sample we want to feed for each time step.

2.4.1 Equations

The equations of an LSTM are quite unusual. Let's start with the more classic gate equations. They are the ones that behave like the more classic NN. The input (equation (2.5)), forget (equation (2.6)) and output (equation (2.7)) gates are described below.

$$\overrightarrow{i}_t = \sigma([\overrightarrow{x}_t, \overrightarrow{h}_{t-1}] \cdot W_i + \overrightarrow{b}_i) \tag{2.5}$$

$$\overrightarrow{f}_t = \sigma([\overrightarrow{x}_t, \overrightarrow{h}_{t-1}] \cdot W_f + \overrightarrow{b}_f) \tag{2.6}$$

$$\overrightarrow{o}_t = \sigma([\overrightarrow{x}_t, \overrightarrow{h}_{t-1}] \cdot W_o + \overrightarrow{b}_o) \tag{2.7}$$

Where $(W_i, \overrightarrow{b}_i)$, $(W_f, \overrightarrow{b}_f)$ and $(W_o, \overrightarrow{b}_o)$ are the pair of weights matrixes and bias vectors for the input, forget and output gates respectively. $\overrightarrow{x}_t$ is the input vector and $\overrightarrow{h}_t$ is the hidden state vector.

The next equation describes the candidate cell state (equation (2.8)), that will next be used to compute the cell state (equation (2.9)).

$$\overrightarrow{cc}_t = tanh([\overrightarrow{x}_t, \overrightarrow{h}_{t-1}] \cdot W_c + \overrightarrow{b}_c) \tag{2.8}$$

$$\overrightarrow{c}_t = \overrightarrow{f}_t \odot \overrightarrow{c}_{t-1} + \overrightarrow{i}_t \odot \overrightarrow{cc}_t \tag{2.9}$$

Where W_c and $\overrightarrow{b}_c$ are the weights matrix and bias vector for the candidate cell state.

The final step of the LSTM is to compute the hidden state (equation (2.10)).

$$\overrightarrow{h}_t = \overrightarrow{o}_t \odot tanh(\overrightarrow{c}_t) \tag{2.10}$$

We set x_1 as the first input and define $\overrightarrow{h}_0$ as a zero only vector.

2.4.2 Usage

Using LSTM can be a bit tricky. Due to its sequential nature, it takes several time steps. Every hidden state (h_t) is passed to the next time step as an input. This hidden state can be used to compute an output at any time step t. Figure 2.5 shows a visual representation of an LSTM going from the current time step to the following time step.

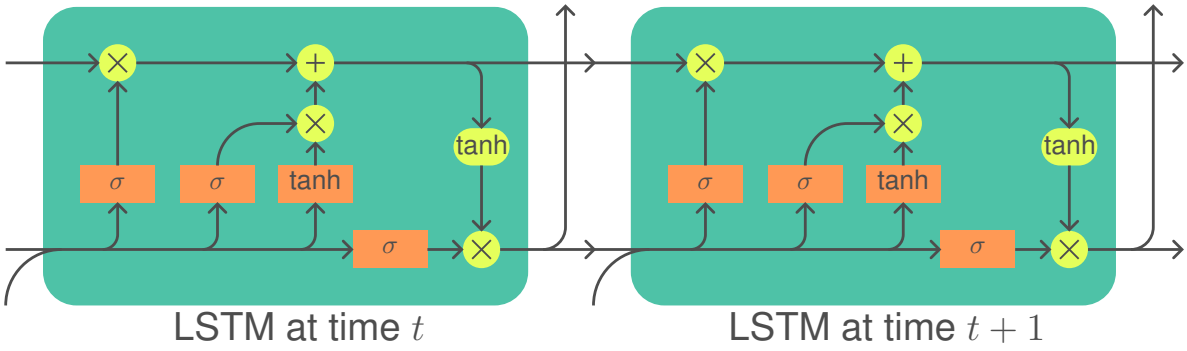


Figure 2.5: Unfolded LSTM, legend in figure 2.3b

2.4.3 Variants

LSTMs come in a few different flavors of implementations. Usually, when LSTMs are mentionned, the version used is the No Peephole (NP) version. This is the most common version as those are the equations described on the wikipedia page [2] and used in libraries like Keras [17] or PyTorch [18]. In fact, there are at least 8 other variations of LSTMs [19]. The difference between them varies from one to the other, some of them are detailled next.

2.4.3.A Vanilla LSTM

This variant of the LSTM was originally the only LSTM network, hence its name. It differs from the classic LSTM with its use of peephole weights, hence the name of the classic LSTM being No Peephole. The Vanilla LSTM is thus defined by the following equations [19, 20] :

$$\overrightarrow{i}_t = \sigma([\overrightarrow{x}_t, \overrightarrow{h}_{t-1}] \cdot W_i + \overrightarrow{b}_i + \overrightarrow{c}_{t-1} \odot \overrightarrow{p}_f) \tag{2.11}$$

$$\overrightarrow{f}_t = \sigma([\overrightarrow{x}_t, \overrightarrow{h}_{t-1}] \cdot W_f + \overrightarrow{b}_f + \overrightarrow{c}_{t-1} \odot \overrightarrow{p}_i) \tag{2.12}$$

$$\overrightarrow{o}_t = \sigma([\overrightarrow{x}_t, \overrightarrow{h}_{t-1}] \cdot W_o + \overrightarrow{b}_o + \overrightarrow{c}_t \odot \overrightarrow{p}_o) \tag{2.13}$$

With $\overrightarrow{p}_f$, $\overrightarrow{p}_i$ and $\overrightarrow{p}_o$ being the peephole weights vectors. Their size is the same as the size of the hidden state vector ($\overrightarrow{h}_t$). The equations that have not been rewritten simply stay the same.

2.4.3.B Full Gate Recurrence LSTM

The Full Gate Recurrence (FGR) LSTM is one of the most complex variant of the LSTM. This is due to the amount of feedback connections. This is basically a more complex version of the Vanilla LSTM. The modified equations are :

$$\overrightarrow{i}_t = \sigma([\overrightarrow{x}_t, \overrightarrow{h}_{t-1}] \cdot W_i + \overrightarrow{i}_{t-1} \cdot W_{ii} + \overrightarrow{f}_{t-1} \cdot W_{if} + \overrightarrow{o}_{t-1} \cdot W_{io} + \overrightarrow{b}_i + \overrightarrow{c}_{t-1} \odot \overrightarrow{p}_i) \tag{2.14}$$

$$\overrightarrow{f}_t = \sigma([\overrightarrow{x}_t, \overrightarrow{h}_{t-1}] \cdot W_f + \overrightarrow{i}_{t-1} \cdot W_{fi} + \overrightarrow{f}_{t-1} \cdot W_{ff} + \overrightarrow{o}_{t-1} \cdot W_{fo} + \overrightarrow{b}_f + \overrightarrow{c}_{t-1} \odot \overrightarrow{p}_f) \tag{2.15}$$

$$\overrightarrow{o}_t = \sigma([i\overrightarrow{x}_t, \overrightarrow{h}_{t-1}] \cdot W_o + \overrightarrow{i}_{t-1} \cdot W_{oi} + \overrightarrow{f}_{t-1} \cdot W_{of} + \overrightarrow{o}_{t-1} \cdot W_{oo} + \overrightarrow{b}_o + \overrightarrow{c}_{t-1} \odot \overrightarrow{p}_o) \tag{2.16}$$

With W_{fi} being the weight matrix of the input feedback connection to compute the forget gate, and the same goes for the other weight matrixes.

Once again, the equations that have not been rewritten are unchanged.

2.4.3.C Other small variants

- Removing the activation function from the Vanilla LSTM for :
  - the candidate cell and thus becomes $\overrightarrow{cc}_t = ([\overrightarrow{x}_{t-1}, \overrightarrow{h}_{t-1}] \cdot W_c + \overrightarrow{b}_c)$, this is called the No Input Activation Function (NIAF)
  - the output of the previous cell state and thus becomes $\overrightarrow{h}_t = \overrightarrow{o}_t \odot \overrightarrow{c}_t$, and is called No Output Activation Function (NOAF)

- Using units instead of the respective gates :
  - No Input Gate (NIG) : $\overrightarrow{i}_t = 1$
  - No Output Gate (NOG) : $\overrightarrow{o}_t = 1$
  - No Forget Gate (NFG) : $\overrightarrow{f}_t = 1$

2.5 Gated Recurrent Unit

The Gated Recurrent Unit (GRU) is another type of RNN. It is also known to reduce the effect of the vanishing gradient problem. It was first introduced to improve translation techniques [13].

The GRU is very often compared to the LSTM, being sometimes assimilated as a type of LSTM [19]. Their performance was found to be very similar in most situations [21], making those two types of RNNs coexistant in the modern machine learning world.

There are two versions of the GRU, both are found on the internet, they are known as the encoder and decoder version [13]. They were originally designed to encode the message to translate and then decode in the translation. PyTorch only supports the decoder version [22], while the Keras library supports both [23] chosen by changing an argument.

2.5.1 Encoder GRU

The encoder GRU is the version of the GRU most widely described on the internet. It contains an update gate (equation (2.17)), a reset gate (equation (2.18)), a candidate activation gate (equation (2.19)). The hidden state is then computed (equation (2.20)) using the previous results.

$$\overrightarrow{z}_t = \sigma([\overrightarrow{x}_t, \overrightarrow{h}_{t-1}] \cdot W_z + \overrightarrow{b}_z) \tag{2.17}$$

$$\overrightarrow{r}_t = \sigma([\overrightarrow{x}_t, \overrightarrow{h}_{t-1}] \cdot W_r + \overrightarrow{b}_r) \tag{2.18}$$

$$\overrightarrow{ch}_t = tanh(\overrightarrow{x}_t \cdot W_{hx} + (\overrightarrow{r}_t \odot \overrightarrow{h}_{t-1}) \cdot_{hh} + \overrightarrow{b}_h) \tag{2.19}$$

$$\overrightarrow{h}_t = (\overrightarrow{1} - \overrightarrow{z}_t) \odot \overrightarrow{h}_{t-1} + \overrightarrow{z}_t \odot \overrightarrow{ch}_t \tag{2.20}$$

Where $(W_z, \overrightarrow{b}_z)$, $(W_r, \overrightarrow{b}_r)$,$(W_{hx}, W_{hh}, \overrightarrow{b}_h)$ are the weights matrixes and bias vectors for the update, reset and candidate activation gates respectively.

A visual representation of the encoder GRU cell is available in figure 2.6.

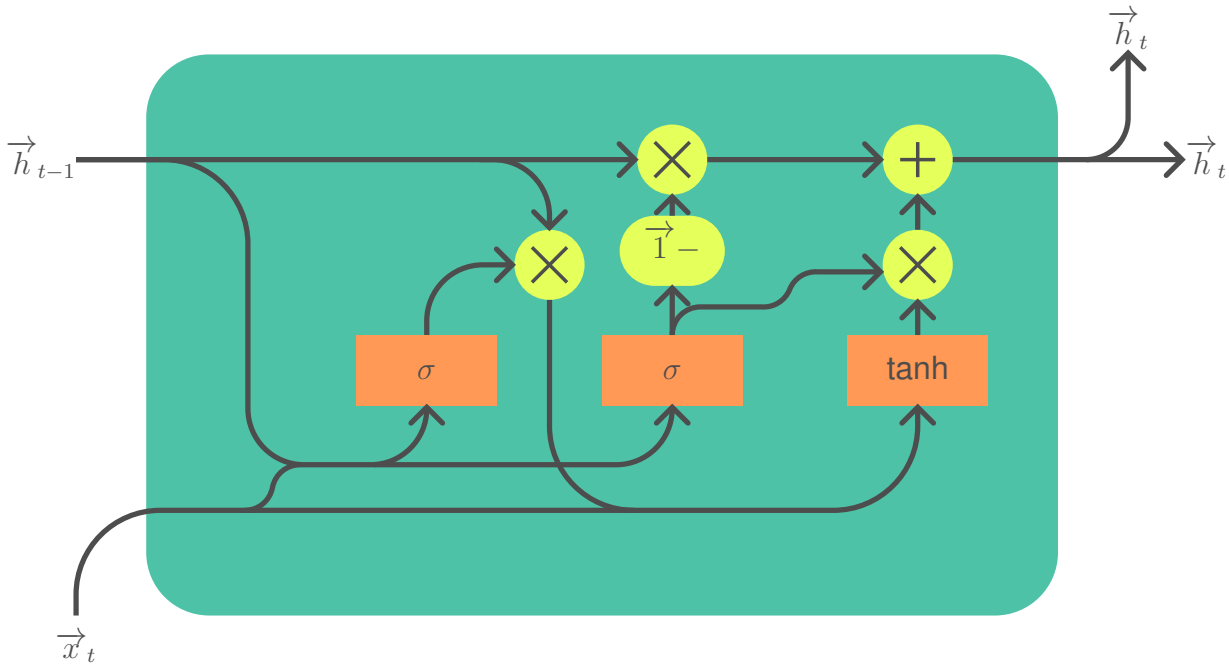


Figure 2.6: Encoder GRU cell, legend in figure 2.3b

Sometimes the equation (2.20) can be found in another form (equation (2.21))[22]. This, however, has no impact on the final results, it means the weights are going to be trained differently for the update gate.

$$\overrightarrow{h}_t = \overrightarrow{z}_t \odot \overrightarrow{h}_{t-1} + (\overrightarrow{1} - \overrightarrow{z}_t) \odot \overrightarrow{h}_t \tag{2.21}$$

2.5.2 Decoder GRU

The decoder GRU, while being less described, is the version used in pyTorch, which is getting very popular.

The candidate activation gate (equation (2.22)) is the only difference from the encoder GRU.

$$\overrightarrow{ch}_t = tanh(\overrightarrow{x}_t \cdot W_{hx} + \overrightarrow{b}_{hx} + \overrightarrow{r}_t \odot [\overrightarrow{h}_{t-1} \cdot W_{hh} + \overrightarrow{b}_{hh}]) \tag{2.22}$$

A visual representation of the decoder GRU cell can be found in figure 2.7.

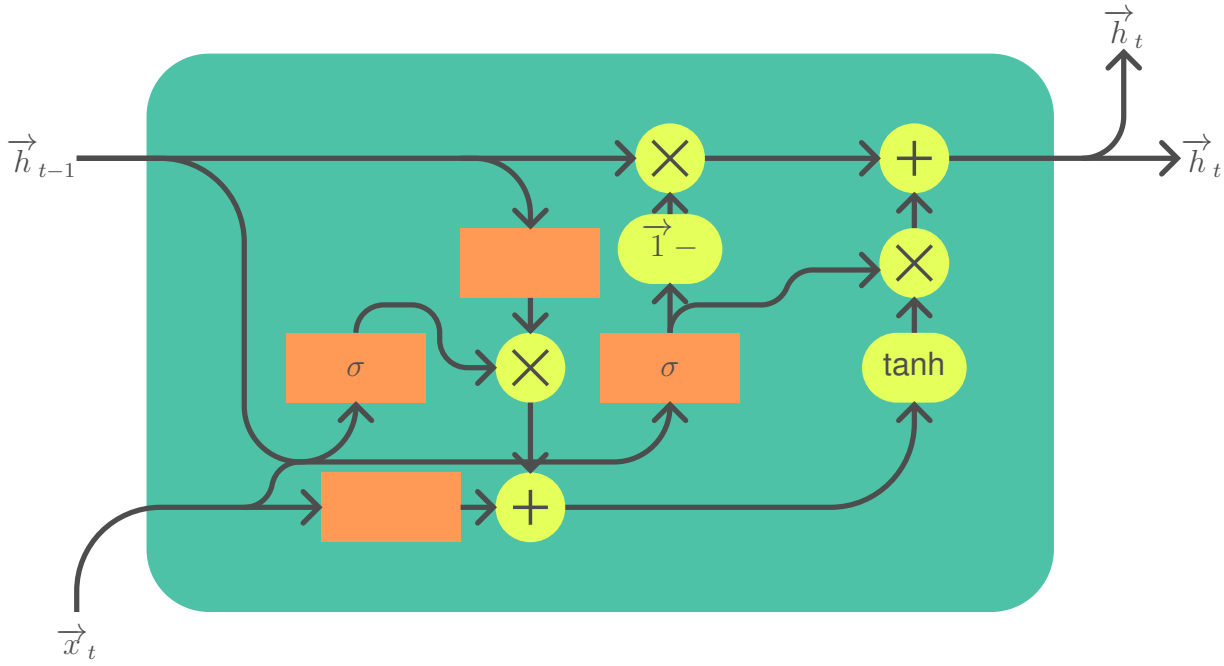


Figure 2.7: Decoder GRU cell, legend in figure 2.3b

2.5.3 Similarities with LSTM

While the GRU is its own type of RNN, it has been compared and assimilated to an LSTM [19]. Indeed, the subtype of LSTM it has been associated with is called the Coupled Input-Forget Gate (CIFG) LSTM. If the input and forget gates are combined into one ($\overrightarrow{f_t} = \overrightarrow{1} - \overrightarrow{i_t}$), they can be assimilated as the update gate of the GRU. The reset gate would then be the LSTM's output gate.

Equations (2.23) to (2.27) are the GRU's equations with the LSTM's names.

$$\overrightarrow{f}_t = \sigma([\overrightarrow{x}_t, \overrightarrow{h}_{t-1}] \cdot W_f + \overrightarrow{b}_f) \tag{2.23}$$

$$\overrightarrow{o}_t = \sigma([\overrightarrow{x}_t, \overrightarrow{h}_{t-1}] \cdot W_o + \overrightarrow{b}_o) \tag{2.24}$$

$$\overrightarrow{i}_t = \overrightarrow{1} - \overrightarrow{f}_t \tag{2.25}$$

$$\overrightarrow{cc}_t = tanh(\overrightarrow{x}_t \cdot W_{cx} + (\overrightarrow{o}_t \odot \overrightarrow{h}_{t-1}) \cdot W_{ch} + \overrightarrow{b}_c) \tag{2.26}$$

$$\overrightarrow{h}_t = (\overrightarrow{i}_t) \odot \overrightarrow{h}_{t-1} + \overrightarrow{f}_t \odot \overrightarrow{cc}_t \tag{2.27}$$

Differences between the two RNNs are :

- The GRU has no output activation function.
- The GRU has combined hidden state and cell state.
- The GRU's candidate hidden state is point wise multiplied with the reset gate before passing into the dense layer.

2.6 Memristors

Memristors are the lesser known fourth fundamental passive component of electronics, among resistors, capacitors and inductor. It was first theorized in 1971 by L. Chua from UC Berkeley, in [24]. The name comes from the blend of *memory* and *resistance*. The theory behind this component was extracted from a missing component to link the four fundamental circuit variables, voltage (v), charge (q), current (q) and flux (ϕ). Figure 2.8 shows the four fundamental variables are on each side of the square, with the ones on opposite sides being linked by the following equations :

$$d\phi = v \cdot dt \tag{2.28}$$

$$dq = i \cdot dt \tag{2.29}$$

Resistors, capacitors and inductors were already very established and well known components, so it was theorized that a fourth device should then exist to physically link flux (ϕ) and charge (q). The flux in this case is not a magnetic flux and is defined as such : $d\phi = v \cdot dt \Rightarrow \phi = \int v\, dt$.

The component stayed theoretical until 2008 when it was implemented in a physical device for the first time [25]. It took 37 years to have an actual working device.

There is then an extension of this theoretical device to another, the memristive device. It was theorized in 1976 by L. Chua and S. M. Kang [26]. The difference between the memristor and the memristive devices is its internal behavior. Memristive device are commonly referred to as memristors as well.

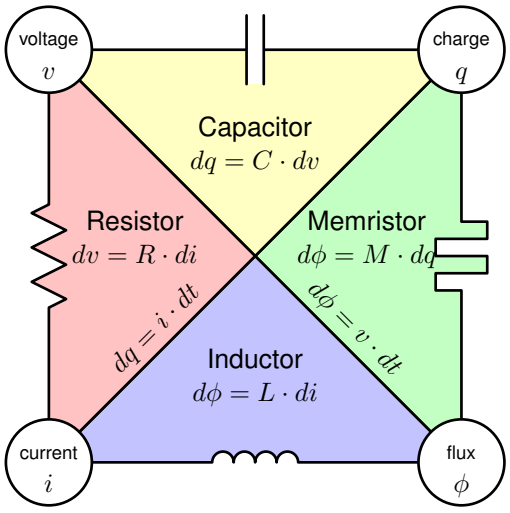


Figure 2.8: Fundamental passive components, adapted from [3]

2.6.1 Equations

A memristor links the flux (ϕ) and charge (q) by the memristance. This memristance is defined with the following equation :

$$M(q) = \frac{d\phi}{dq} \tag{2.30}$$

It can be compared with the other fundamental components like resistor ($R(i) = \frac{dv}{di}$), capacitor ($\frac{1}{C(q)} = \frac{dv}{dq}$) and inductor ($L(i) = \frac{d\phi}{di}$). We can then extract a more useful equation in an actual circuit :

$$v(t) = M(q(t)) \cdot i(t) \tag{2.31}$$

Similarly, a memductance can be defined as such :

$$\kappa(\phi) = \frac{dq}{d\phi} \tag{2.32}$$

A memristive device is slightly differently defined, it still uses a memristance, but here the memristance also depends on an internal state called x. This gives us this equation :

$$v(t) = M(x, i) \cdot i(t) \tag{2.33}$$

The internal state (x) is not linked to flux or charge in the case of a memristive device.
Once again, we can also define the memristive device using a memductance :

$$i(t) = \kappa(x, v) \cdot v(t) \tag{2.34}$$

In all of the previous equations, v is the voltage in Volt (V), i is the current in Ampere (A), ϕ is the flux in Weber (Wb), q is the charge in Coulomb (C), M is the memristance in Ohm (Ω) and κ is the memductance in Siemens (S or Ω^{-1}).

2.6.2 Behavior

A memristor is defined as a non-linear two-terminal fundamental electrical component. It behaves as a resistance with memory (hence its name), meaning that it changes its resistance based on how much charge went through it. This enables us to manipulate the resistance of the component. The huge benefit of memristors is the ability to retain its internal resistance, the device can be left without power for a long period of time (retention time of minimum 10 years according to [27]). When the device is powered backup, it will have the same resistance it had before.

Memristive devices have a similar behavior, the memristive device's resistance will change depending on the internal state (x). That internal state changes based on how much and how long voltage signals or currents are applied to the memristive device.

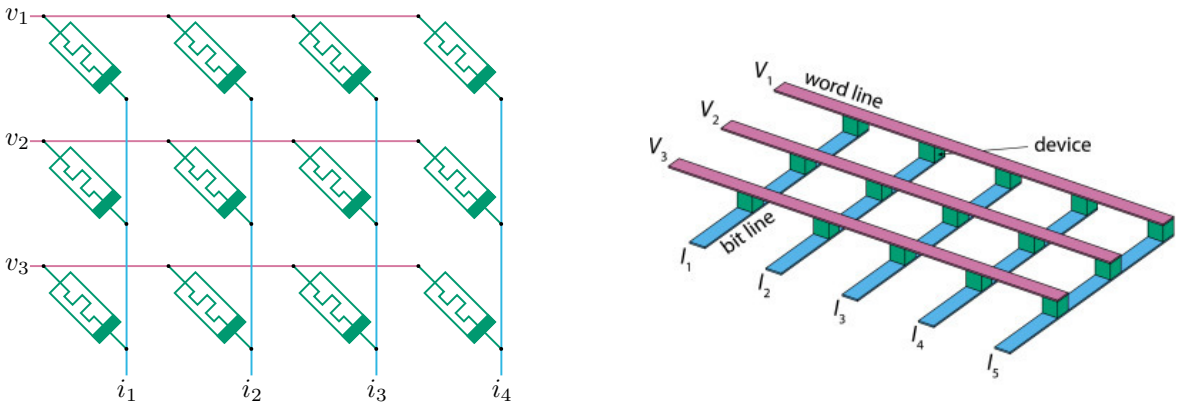
2.6.3 Usage

The main research for memristor usage is using them as ReRAM. The idea behind ReRAM is to use memristors as Non-Volatile memory. It uses two states of the device with known resistances (R_{on} and R_{off}), giving it binary property. Reading the memory simply requires setting a voltage and reading the output current. It is better than current solutions (HDD, SSD) as it has a much lower latency. It is better than traditionnal DRAM because it keeps the information even when turned off. This makes ReRAM a good replacement for both RAM and HDDs/SSDs, thus eliminating von Neumann bottleneck due to the von Neumann architecture that is used in all modern computers.

They can also be used to set a resistance to be able to perform analog multiplication. Setting them in a crossbar array makes them a very strong candidate to be used in neuromorphic computing.

2.7 Memristors Crossbar Array

Setting memristors in a crossbar array allows to perform analog VMM, also called Multiply and Accumulate. This circuit works great for this use as all the computation is done almost instantly and using physics laws. Figure 2.9 shows what a typical crossbar array looks like.



(a) Schematics, inspired from [28]

(b) 3 dimensional representation, from [28]

Figure 2.9: Memristor crossbar array

It uses physical properties of electrical systems to perform analog computing. In what follows I will use the circuit node in figure 2.10 to explain the theory behind the memristor crossbar array.

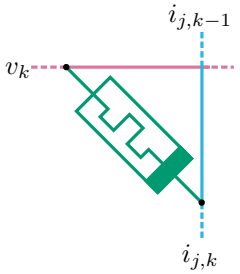


Figure 2.10: Memristor crossbar node of the k^{th} line and j^{th} column

First of all, a voltage is applied on the k^{th} line, because every column is virtually grounded, the voltage applied to the memristor, of a set resistance R_k, is v_k. As such and using Ohm's law, we know that the current flowing into the memristor (i_k) is bound by the following equation :

$$v_k = R_k \cdot i_k \Rightarrow i_k = v_k \cdot (\frac{1}{R_k}) = v_k \cdot G_k \tag{2.35}$$

With G_k being the conductance of memristor, defined as $G_k = \frac{1}{R_k}$.

This line then joins the column where a current of $i_{j,k-1}$ is flowing, then according to Kirchhoff's current law the resulting current is :

$$i_{j,k} = i_{j,k-1} + i_k = i_{j,k-1} + v_k \cdot G_k \tag{2.36}$$

By applying this process to the whole system we find that the current at the bottom of one column is :

$$i_1 = G_1 \cdot v_1 + G_2 \cdot v_2 + G_3 \cdot v_3 \tag{2.37}$$

With G_1, G_2 and G_3 being the conductance of the 3 memristors in the first column.

Using such an analog circuit is a huge gain in time, for two main reasons :

- By being analog, this circuit allows for analog computations which are much faster than their digital counterparts. As explained in this section, the circuit only uses physical properties of the different components and can compute large VMMs in a very short amount of time.
- The other advantage of such a system is the use of the memristors, that make the need to copy data from another memory useless as the memristors have a long term memory themselves. This is one of the ways to break down the Von Neumann bottleneck.

2.8 Memristor's model

There are several ways to model a memristor [25, 29–31]. The way the memristor works depends on the materials used to fabricate the memristor. Each type of memristor has slightly different behavior. Models were thus created to simulate, as accurately as possible, the memristor in electrical circuit simulation software such as **Cadence**'s *Virtuoso* or SPICE.

Some examples of memristor's model are :

- Linear Ion Drift Model (LIDM), the model of the first memristor [25].
- Simmons Tunnel Barrier Model (STBM), is another model of the TiO_x memristor created in 2008 by **HP** [25, 31].
- ThrEshold Adaptive Memristor Model (TEAM), is a model that can easily adapt to several types of memristive devices while focusing on fast computation [29].
- Voltage ThrEshold Adaptive Memristor Model (VTEAM), is a later improvement of the TEAM. The internal resistance depends on voltage, unlike TEAM which depends on current [30].

The VTEAM being the most versatile model, its implementation would fit most memristor types.

2.8.1 Equations

It has been established that the internal resistance also depends on an internal state x (equation (2.33)).

$$\frac{dx}{dt} = f(x, v) \tag{2.38}$$

Where v is the voltage, t is time.

The function f is described in its simplest form as in equation (2.39).

$$f(x, v) = g(x) \cdot s(v) \tag{2.39}$$

Where g is the window function and s the sensitivity function.

- The window function delimits the operational boundaries of the memristor.
- The sensitivity function, also known as voltage sensitivity describes the effect of voltage on the internal state's variations.

The model is based on previous work [32] that aims to reproduce as many memristive device types. They have chosen to use the resistive state (R) as the internal state [33]. Based on this assumption the equations (2.40) and (2.41) are extracted from the memristive device. This model is of course very basic and ignores some physical dependencies such as temperature.

$$i(R,v) = \begin{cases} \frac{a_p}{R} \cdot sinh(b_p \cdot v) & v \geq 0 \\ \frac{a_n}{R} \cdot sinh(b_n \cdot v) & v < 0 \end{cases} \tag{2.40}$$

$$\frac{dR}{dt} = f(R,v) = s(v) \cdot g(R,v) \tag{2.41}$$

The sensitivity function was chosen to be a voltage-dependent exponential function, like described in equation (2.42).

$$s(v) = \begin{cases} A_p \cdot (e^{t_p \cdot |v|} - 1) & v > 0 \\ A_n \cdot (e^{t_n \cdot |v|} - 1) & v < 0 \\ 0 & v = 0 \end{cases} \tag{2.42}$$

The window function used is a state-dependent quadratic function as shown in equation (2.43).

$$g(R,v) = \begin{cases} (R - r_p(v))^2 & v > 0 \\ (R - r_n(v))^2 & v < 0 \\ 0 & v = 0 \end{cases} \tag{2.43}$$

Where r_p and r_n are voltage dependent in a polynomial manner. They represent the boundaries of the state variable. All the other parameters in equations (2.40) to (2.43) that have not been declared are free fitting variables. They have to be set depending on the type of memristor used.

2.8.2 Verilog-A integration

The model now needs to be adapted to work as Verilog-A code. Verilog-A is a hardware desciption language, very popular in the industry for its simplicity and flexibility when using it with circuit simulator such as **Cadence**'s *Virtuoso*.

The focus of this specific implementation will be the resistance range $[20k\Omega, 120k\Omega]$. The boundaries functions are found in equation (2.44).

$$\begin{cases} r_p(v) = r_{p,0} + r_{p,1} \cdot v & v > 0 \\ r_n(v) = r_{r,0} + r_{n,1} \cdot v & v \leq 0 \end{cases} \tag{2.44}$$

Where $r_{p,0}$, $r_{p,1}$, $r_{n,0}$ and $r_{n,1}$ are free fitting variables that need to be extracted from the physical device.

The changes of resistive states from the initial resistance (R_0) are computed by equation (2.45) extracted from [32]. It uses the bias voltage (v_b) of the pulse applied to the memristor. The pulse has a duration of t.

$$R(t)_{|V_b} = \begin{cases} \frac{R_0+s(V_b)\cdot r_p(V_b)\cdot(r_p(V_b)-R_0)\cdot t}{1+s(V_b)\cdot(r_p(V_b)-R_0)\cdot t} & V_b > 0, R < r_p(V_b) \\ \frac{R_0+s(V_b)\cdot r_n(V_b)\cdot(r_n(V_b)-R_0)\cdot t}{1+s(V_b)\cdot(r_n(V_b)-R_0)\cdot t} & V_b \leq 0, R > r_n(V_b) \\ R_0 & else \end{cases} \tag{2.45}$$

The final Verilog-A code is available in [32].

2.8.3 Behavior

The model implements, as with physical memristors, a threshold voltage value from which its resistive states starts changing. This threshold value is v_{read}. For this specific model the threshold voltage is set to $v_{read} = 500mV$.

Every pulse is composed of 2 parts and lasts for a total of $1ms$:

- The reading of the resistive state, by applying a triangular wave that is bound by the threshold value v_{read}. The triangular wave starts from $0V$, goes to v_{read} then to $-v_{read}$ before going back to $0V$.
- The writing to the memristor. During this part the resistive state will change. This is done by applying a square wave of variable width (t_Δ) with a set bias voltage (v_{bias}).

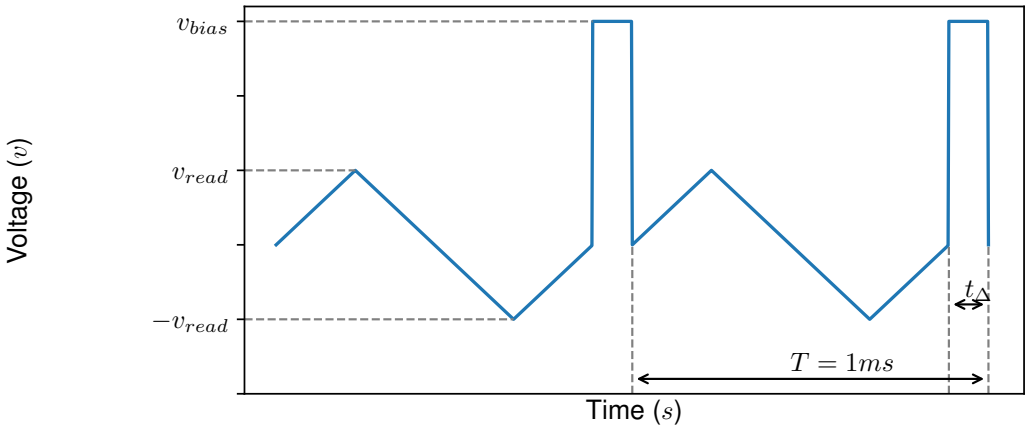


Figure 2.11: Pulse shape graph

The figure 2.11 is the visual representation of what each pulse, the value is read during the first half of the pulse.

The resistive state of the memristor changes depending on the pulse width (t_Δ) and the bias voltage (v_{bias}) used for each pulse.

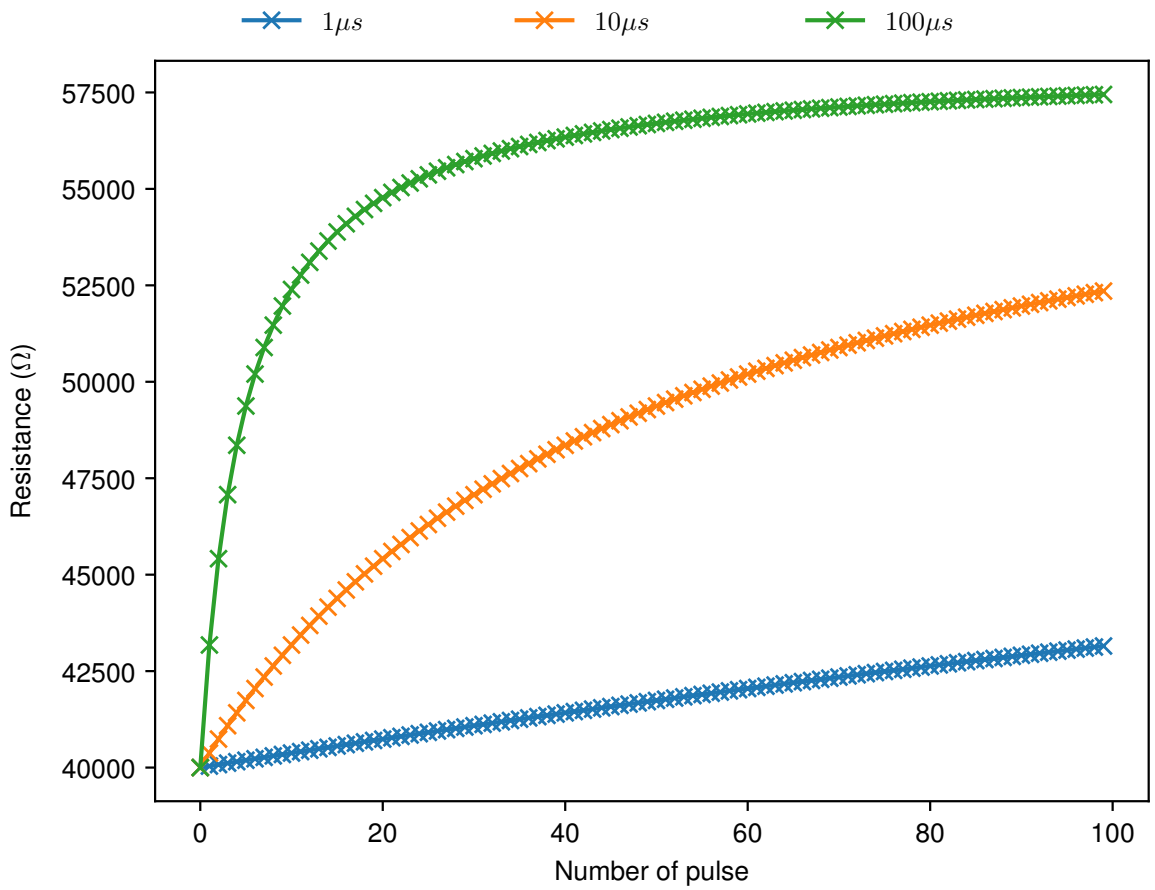


Figure 2.12: Memristor's resistive state under different pulses width (Fixed bias voltage $v_{bias} = v_{dd} = 1.8V$)

The effect of the pulse width variation is shown in figure 2.12. Indeed, the larger the pulse widths, the faster the internal resistance reaches the bounding resistance (equation (2.44)). The bounding resistance being the same as all curves use the same bias voltage ($v_{bias} = v_{dd} = 1.8V$), the curves will all eventually reach the max resistance.

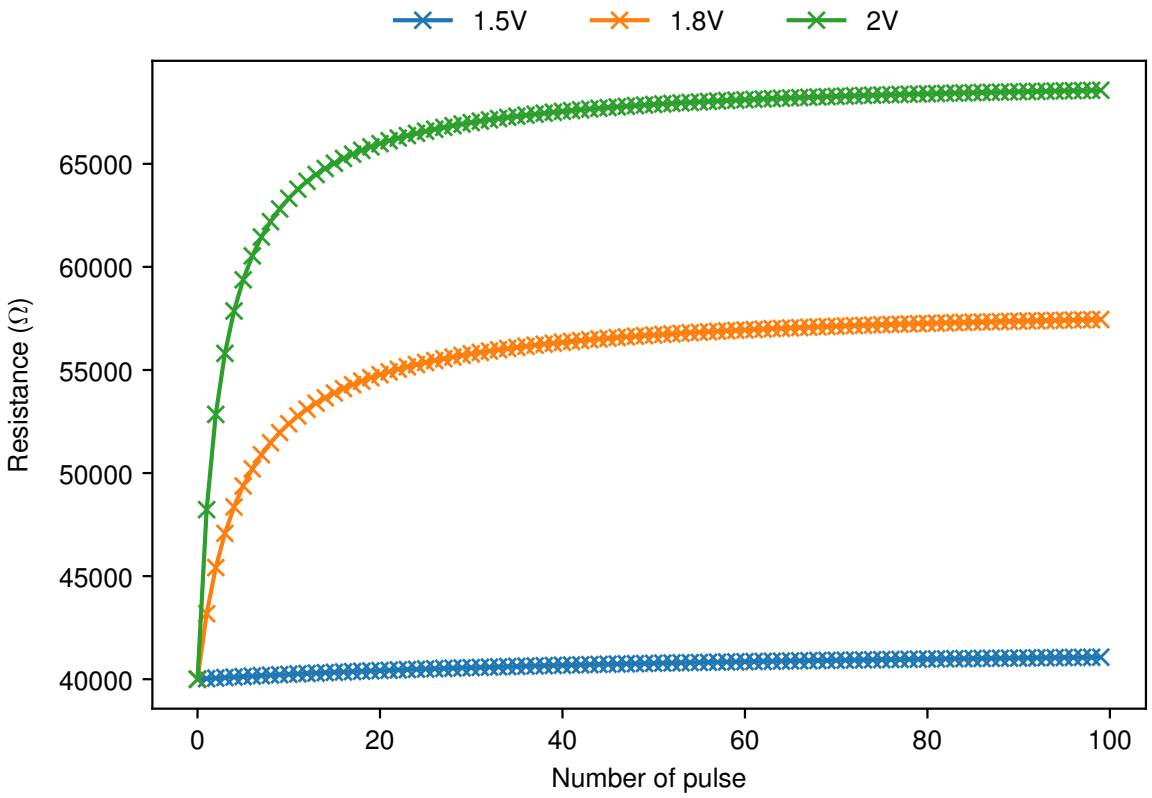


Figure 2.13: Memristor's resistive state under different voltages (Fixed pulse width $t_\Delta = 100\mu s$)

The voltage dependence of the memristor's model is described in figure 2.13. The influence of the bias voltage is, as equations (2.43) and (2.44) show, the way to change the resistance range of the memristor, until the physical limit is reached of course. Applying higher voltage also allows to reach a set resistance faster as the slopes are bigger the higher the bias voltage gets. The resistive state can also be lowered by applying negative voltages.

3

The design

Contents

In this chapter, I'm going to describe the different design decisions taken during the study of the thesis.

The system will be working with a v_{dd} of $1.8V$. Such a value was chosen because this is a low power system.

The way values are encoded in the analog system will be descibed here as it serves for the entire thesis. In order for the system to support negative numbers we're going to use a v_{cm} set to $\frac{v_{dd}}{2}$. That means that $v_{cm} = \frac{v_{dd}}{2} = 0.9V$. This v_{cm} will then describe a zero. A step of one was chosen to be $0.1V$ in the analog circuit. Table 3.1 shows the conversions from a real number to its voltage equivalent.

Real value	Voltage
0	$0.9V$
1	$1.0V$
x	$v(x) = \frac{x}{10} + v_{cm}$
$real(v) = (v - v_{cm}) \cdot 10$	v

Table 3.1: Real/Voltage Conversion Table.

Since the system cannot reach voltage outside of the operating range with the intended behavior, the voltage is then restricted to $v \in [0, 1.8]$. This means that the range of real value that the systems can handle is $x \in [-9, 9]$.

Any data inputed in the analog system should have been previously checked to make sure it stays in the range of accepted values.

In order to create a fully functional LSTM or GRU circuit, there are a few components that are required.

An analog activation function circuit to have an analog equivalent to the activation function necessary to the good behavior of LSTMs and GRUs.

A memory cell to allow to store the analog value from one time step to the next.

A circuit for the crossbar array, to have a more general circuit that can easily be adapted.

Other, less significant, components required for the circuits to work.

3.1 The activation functions

Activation functions play a great role in the results obtained [34]. It is the reason why making good activation function circuits is fundamental. Of course the easy way would be to simply design a hard sigmoid (section C.2), the issue being that the hard sigmoid is much worse than the regular sigmoid, especially for regression problems [35]. The same goes for the tanh and hard tanh functions.

Designing an analog activation function as close to the original is very important for the final result's quality.

3.1.1 Circuit

The circuit used is the same as the one in [36], the circuit is the one shown in figure 3.1. The technology used being different, all the parameters had to be determined empirically to best fit a sigmoid shape. The parameters can be found in table 3.2.

Due to the nature of the functions we want to generate, we will use the same circuit for both a sigmoid and a tanh like functions. The two different functions are generated by changing two parameters.

The functions generated are the same shape and only differ by their output range.

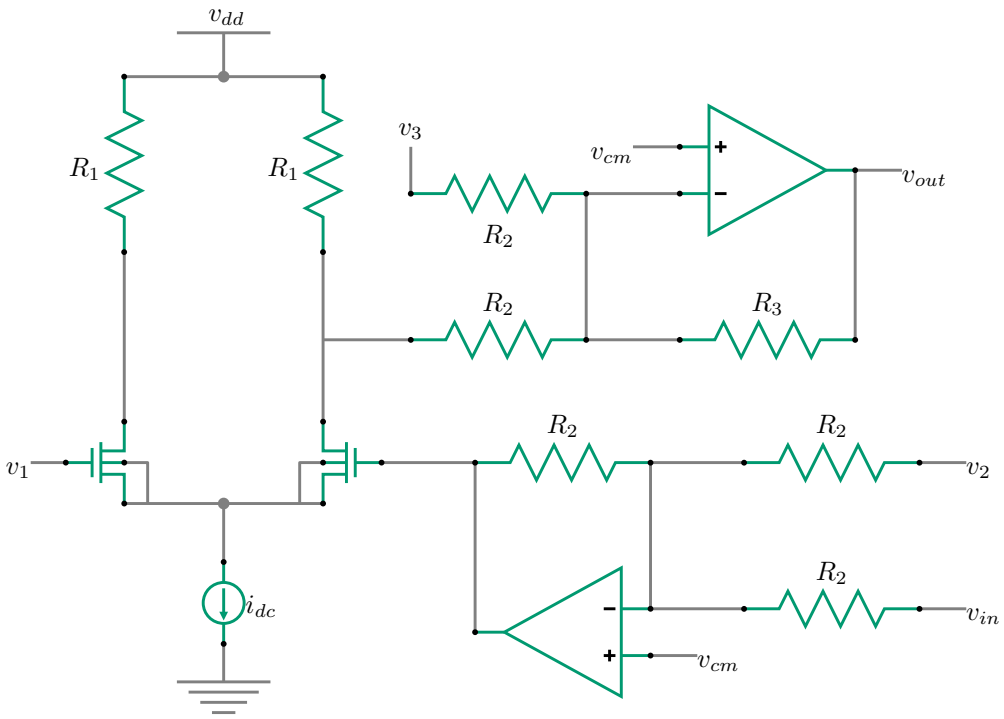


Figure 3.1: Activation functions circuit

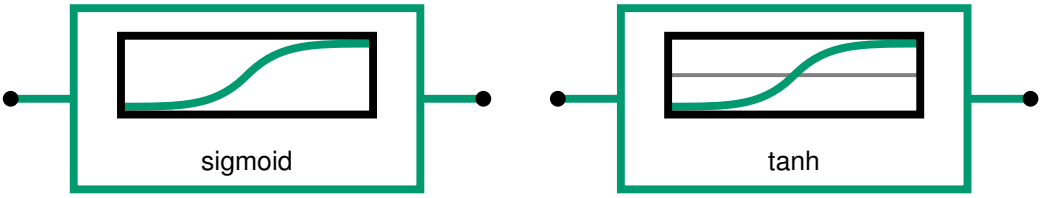
Parameter	Sigmoid	tanh
v_1	$1.1V$	
v_2	$635mV$	
v_3	$0.8V$	$550mV$
i_{dc}	$150uA$	
w	$900nm$	
l	$60nm$	
R_1	$5k\Omega$	
R_2	$10k\Omega$	
R_3	$2k\Omega$	$4k\Omega$

Table 3.2: Circuits parameters

Here, w and l are, respectively, the width and length of the two NMOS of the circuit.

3.1.2 Symbols

The symbols for the sigmoid and the tanh are separated for better understanding.



(a) Sigmoid symbol (b) tanh symbol

Figure 3.2: Activation functions symbols with the input and output pins on either side depending on the flow of the current for better readability

The approximate onChip area of this circuit is :

$$A_{af} = 2 \cdot A_{R_1} + 2 \cdot A_{CMOS} + 5 \cdot A_{R_2} + A_{R_3} + 2 \cdot A_{opAmp} \tag{3.1}$$

With $A_{CMOS} = w \cdot l$

In order to combine the the onChip area of both the sigmoid and tanh activation functions, it is assumed that the variation of area taken by the resistor (R_3) is negligible compared to the overall active area.

3.1.3 Usage

This circuit outputs a voltage that depends on the input voltage passed on. The relation between the two is shown in figure 3.3. The graph also shows the RMSE of each graph compared to the ideal result.

Note that the actual output of the circuit in figure 3.1 is inverted arround V_{cm} a circuit such as the one shown in section A.2 inverts the the signal to its intended values.

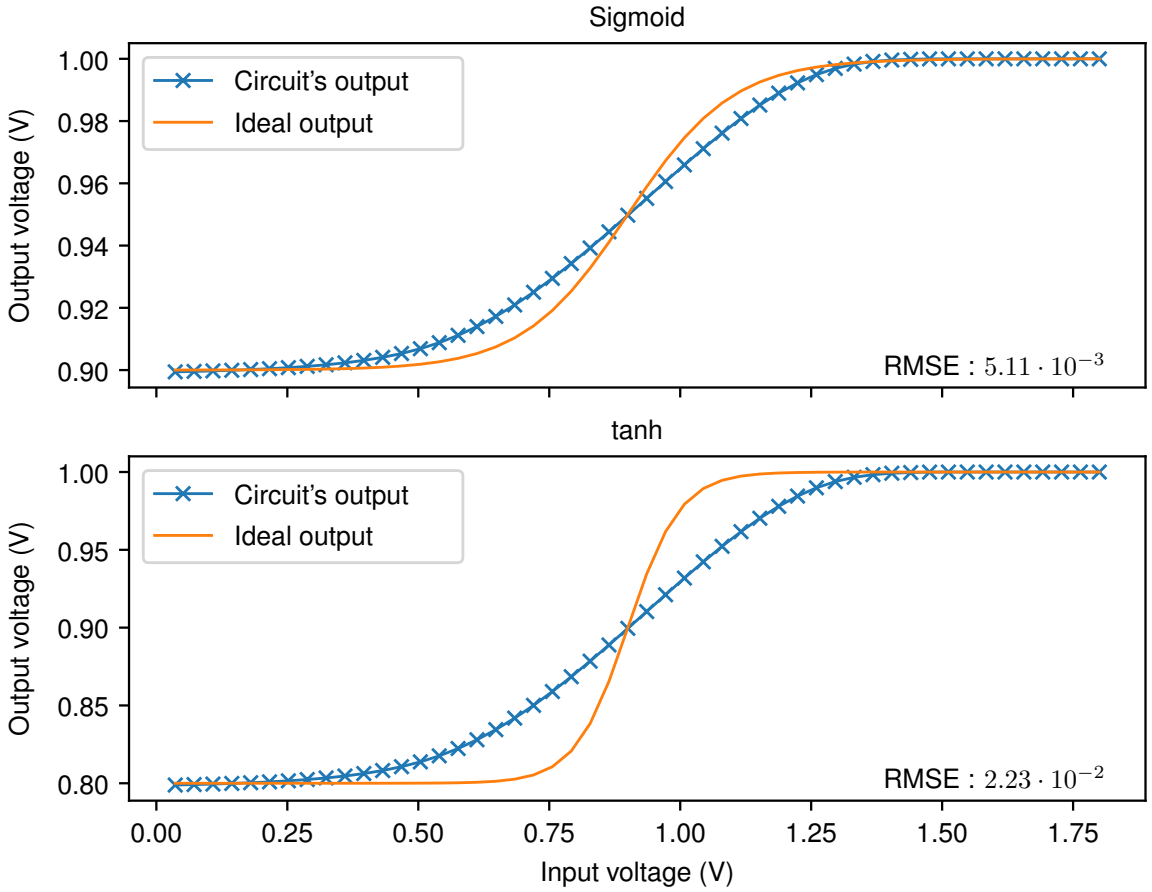


Figure 3.3: Input/Output graph of the activation function circuit for both sigmoid and tanh functions

These functions are still a bit different from the original functions (especially for the tanh). However that does not matter too much as the trainning will be happening with the extracted analog functions, all weights will be set in the circuit. This is the reason why such a difference does not matter. As long as the curves have the similar shape, the result will not be drastically affected.

3.2 Memory cells

The memory cell is a circuit that is able to store an analog value for a limited time. It works using capacitors that have the ability to store a voltage for a short period of time.

3.2.1 Circuit

The circuit is shown in figure 3.4. The value/voltage is trapped in the capacitor using Complementary metal-oxide-semiconductor (CMOS) switches.

The CMOS switches found in the circuit have width of $w = 200nm$ and a length of $l = 60nm$.

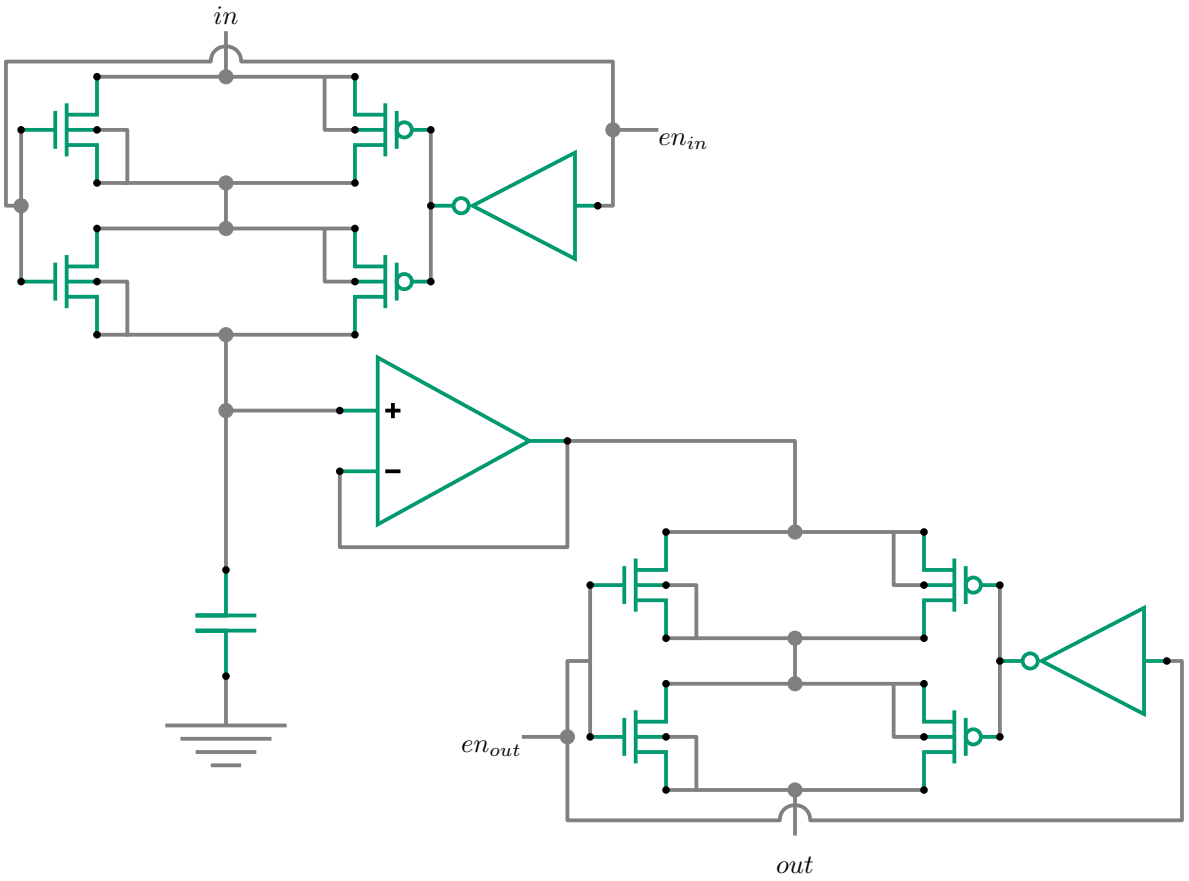


Figure 3.4: Memory cell circuit

The cirucit has a two CMOS switches design to avoid voltage leakage through the swicthes. The system has voltage leakage when only one CMOS swicth, and thus leads to a large memory leak. This is due to the high voltage difference between the two sides of the CMOS swicth. Using two CMOS switches allows for this difference to be mitigated (figure 3.5).

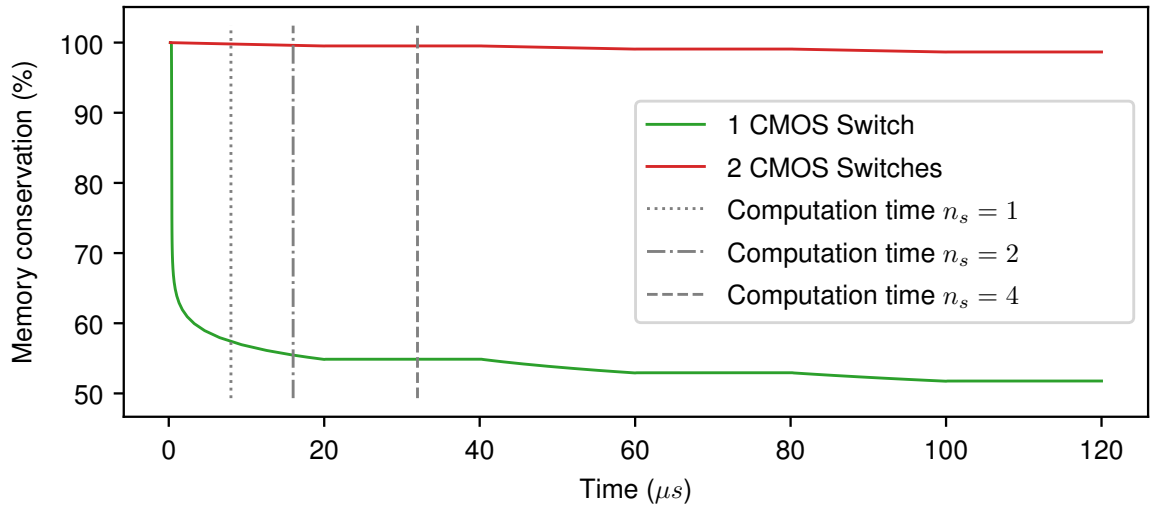


Figure 3.5: Memory conservation in a memory cell with 1 CMOS switch vs 2 CMOS swicthes

3.2.2 Symbol

The symbol for this circuit is designed to show a capacitor because it is its memory mechanism.

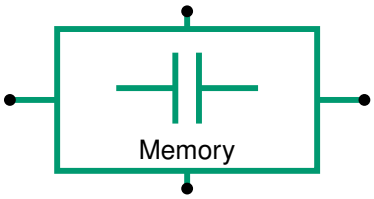


Figure 3.6: Memory cell symbol with the input enable pin (top) and the output enable pin (bottom). The left and right pins can be either input or output. The component is non-linear.

For this circuit the approximate onChip area is :

$$A_{memcell} = 8 \cdot A_{CMOS} + 2 \cdot A_{inv} + A_{opAmp} + A_{cap} \tag{3.2}$$

With $A_{CMOS} = w \cdot l$

3.2.3 Usage

This circuit is pretty straight forward, when the input enable pin is high (up to v_{dd}), then the first switch is open and the capacitor is being charged, if it is low, then the switch is closed and the capacitor keeps the voltage and thus the analog data. When the output enable is high, the buffer (section A.1) forwards the voltage stored in the capactor to the output pin without emptying the capacitor.

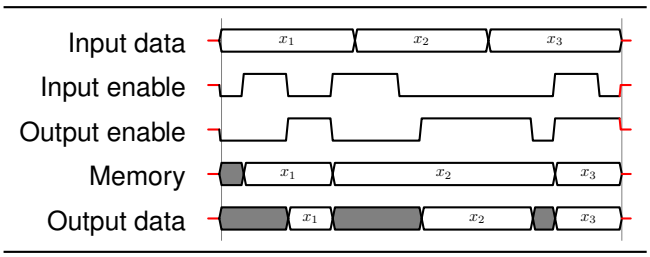


Figure 3.7: Time diagram that shows the logic of the memory cell

3.3 Voltage-driven crossbar circuit

The crossbar circuit theory has already been explained in the section 2.7. This section describes how the crossbar circuit is actually implemented. The final circuit is the one in figure 3.8. The circuit depends on three parameters :

- n_i : The number of input for our crossbar array (not including bias for a more general circuit).
- n_o : The number of parallel output for our crossbar array.
- n_s : The serial size of our crossbar system.

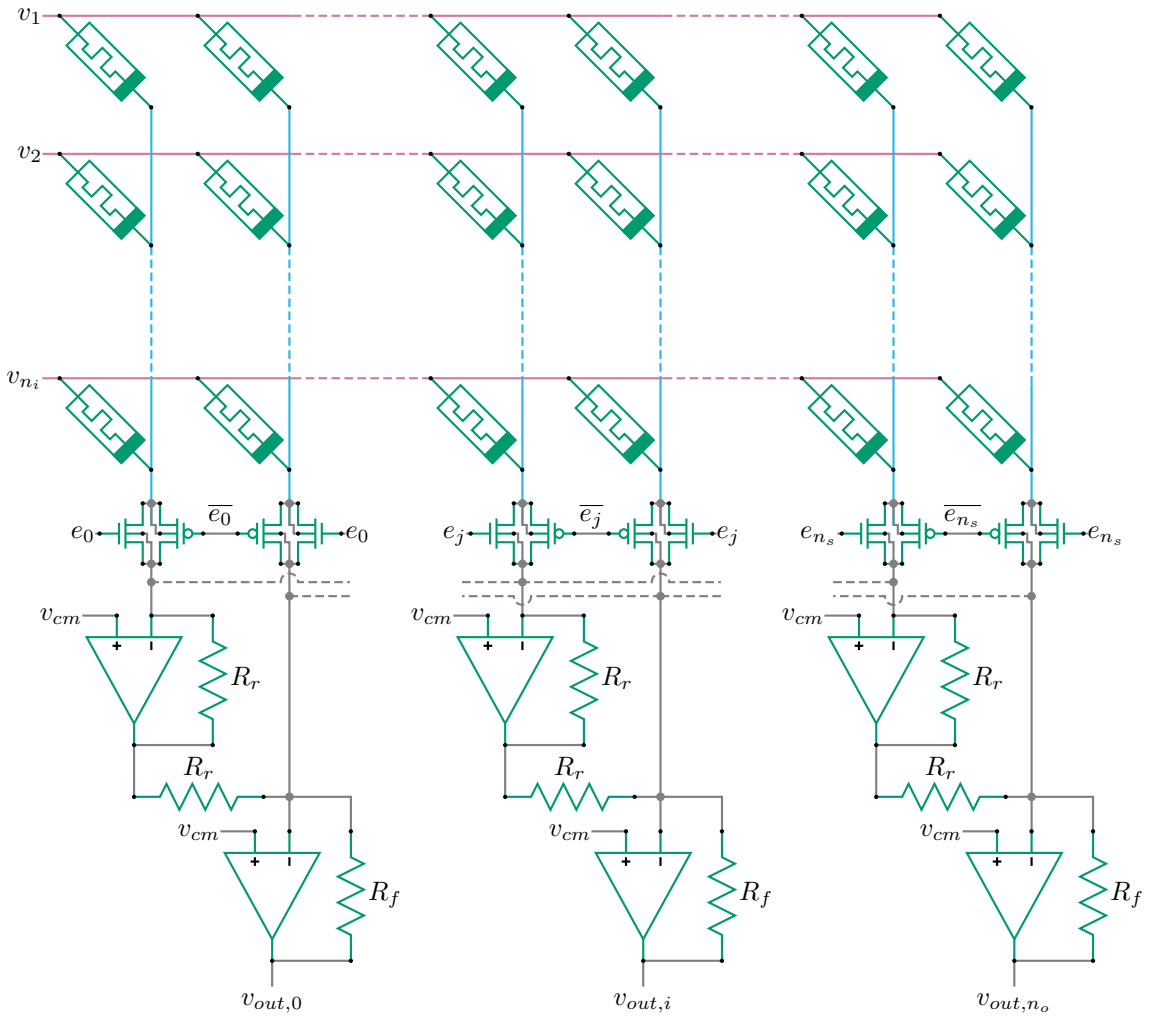


Figure 3.8: Circuit of the crossbar array used in the final system (n_i, n_o, n_s)

The parallel and serial are explained later (section 3.6.3).

The enable flags ($e_j, \forall j \in [\![0, n_s]\!]$ and similarly $\overline{e_j}$) are there to show when the CMOS switches are open, the states of those flags is shown in figure 3.10.

3.3.1 Two memristors per synapse

We could choose between one or two memristor per synapse. Using two memristor per synapse doubles the area but doubles the weight range (table 3.3) and allows to easily use negavite weights. The output voltage will be centered around V_{cm} and be compliant with the standard set in table 3.1.

This design is the one that is used in [1]. Let's assume that a given memristor has a resistance range of $R \in [R_{min}, R_{max}]$, that means its conductance range is $G \in [G_{min}, G_{max}]$ (with $G_{min} = \frac{1}{R_{max}}$ and $G_{max} = \frac{1}{R_{min}}$). This design works using two operational amplifier (opAmp) connected to v_{cm} with the positive pin and the negative pin to the output of the crossbar array. Equations (3.3) to (3.5) are describing how this architecture works. A simplified version of the double memristors per synapse circuit is also available in figure 3.9.

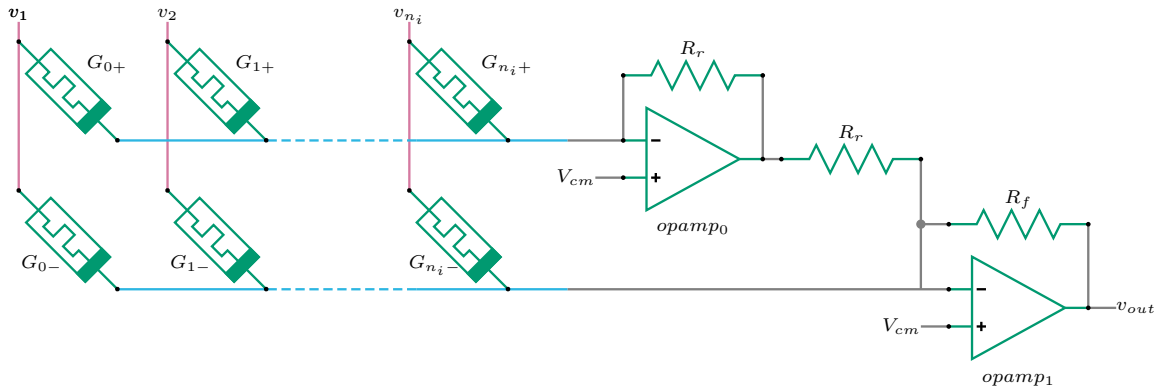


Figure 3.9: Simplified circuit of a double memristor per synapse architecture

Using v_k as the voltage for the input line k. The highest opAmp is identified as $opAmp_0$ and the lowest $opAmp_1$.

For the sake of simplicity, the following equations considers the ground to be v_{cm}.

$$v_{opAmp_0} = -R_r \cdot i_+ \tag{3.3}$$

$$i_{R_f} = i_- + \frac{v_{opAmp_0}}{R_r} = i_- - i_+ \tag{3.4}$$

$$v_{opAmp_1} = v_{out} = -R_f \cdot (i_- - i_+) = R_f \cdot (i_+ - i_-) = R_f \cdot \sum_{k=0}^{n} (G_{k+} - G_{k-}) \cdot v_k \tag{3.5}$$

With $i_+ = \sum_{k=0}^{n} G_{k+} \cdot v_k$ and $i_- = \sum_{k=0}^{n} G_{k-} \cdot v_k$.

	Two memristors per synapse	One memristor per synapse
Maximum weight	$G_{max} - G_{min}$	$G_{max} - \overline{G}$
Minimum weight	$G_{min} - G_{max}$	$G_{min} - \overline{G}$
Range	$2 \cdot (G_{max} - G_{min})$	$G_{max} - G_{min}$

Table 3.3: Synaptic weights precision (extracted from [1])

3.3.2 Serialization

This circuit has the option to be serialized with varying degrees. The idea of serializing the circuit came from [36]. Serializing the circuit reduces the number of components required and thus reduces the final onChip area. Serializing the system increases the time it takes to compute the output.

Serializing the system means not computing all values of the output vector at the same time, but instead computing group by group, the groups' size are n_o. The first output group is computed during e_0 and the i^{th} group is computed during e_i. The timing of when the outputs are available is found in figure 3.10. Those flag control the CMOS swicthes present in figure 3.8, the switches control which output group is outputed.

The CMOS switches are here to open the necessary input gates when the output is required, the CMOS switches are controlled as in figure 3.10.

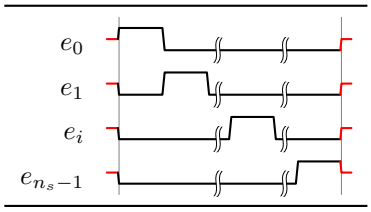


Figure 3.10: Enable flags timing for any value of n_s in a single time step

When the system is used fully in parallel, the CMOS switches are not required can then be removed to lower the final onChip area.

3.3.3 Symbol

The symbol (figure 3.11) defined for the voltage based memristor crossbar array used in this thesis is more compact and helps the readability of the circuits that require a crossbar array. It depends on several parameters, the number of inputs (n_i), the number of outputs (n_o) and the serial size (n_s).

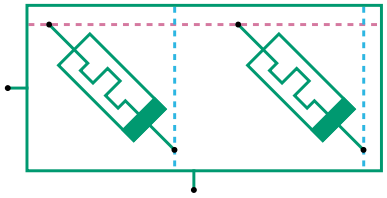


Figure 3.11: Symbol used for the crossbar array, the input pin is a bus of size n_i and the output pin is a bus of size n_o

The approximate onChip area for the crossbar circuit depends on the previously defined parameters.

$$A_{xbar}(n_i, n_o, n_s) = \begin{cases} 2 \cdot n_i \cdot n_o \cdot A_{memristor} + n_o \cdot (2 \cdot [A_{opAmp} + A_{R_r}] + A_{R_f}) & n_s = 1 \\ 2 \cdot n_i \cdot n_o \cdot n_s \cdot A_{memristor} + 2 \cdot A_{CMOS} \cdot n_o \cdot n_s & \\ \quad + n_o \cdot (2 \cdot [A_{opAmp} + A_{R_r}] + A_{R_f}) & \forall n_s > 1 \end{cases} \tag{3.6}$$

With $A_{CMOS} = w \cdot l$.

3.3.4 Usage

3.3.4.A Input voltages

The input voltages must be monitored very carefuly in order not to accidentally change a memristor's internal resistance. Indeed, memristors have a voltage threshold (V_{read}) above which the internal resistance changes [29–32]. This means that the input voltages must follow the inequality in equation (3.7).

$$|v_{mem}| = |v_i - v_{cm}| \leq v_{read} \tag{3.7}$$

Where v_{mem} is the terminal voltage of a memristor.

3.3.4.B Dense layer

A dense layer is simply a VMM. This means that a dense layer in a NN can be assimilated with a crossbar array. Dense layers are mostly used as by fully parallel voltage-driven crossbar array in this thesis to keep the timing simple, otherwise the timing of the enable flags, and later on the LSTM's memory cells (section 2.4), would get too complicated.

3.3.4.C Resistors vs Memristors

In this work, the analog system will be simulated for the inference of the NN, thus the weights will not have to change during the simulation. Because the weights are represented in the analog circuit by the internal resistances of the memristors, the memristors can be replaced by resistors with a set resistance for the simulation.

3.4 Inverter

This section simply describes a very well know electrical component, the inverter. When it a voltage of V_{dd} is applied to the input, the output is grounded and the same goes for the other way around.

(a) Inverter's circuit

(b) Inverter's symbol

Figure 3.12

Like mentionned, the circuit for the inverter is very straight forward (figure 3.12a). The symbol is nothing fancy as well as it is the classic symbol (figure 3.12b) for such a circuit.

The approximate onChip area for the inverter is :

$$A_{inv} = 2 \cdot A_{CMOS} \tag{3.8}$$

With $A_{CMOS} = w \cdot l = 2 \cdot 10^{-7} \cdot 6 \cdot 10^{-8} = 1.2 \cdot 10^{-14} m^2$.

The onChip area of the inverter is then $A_{inv} = 2.4 \cdot 10^{-14} m^2$.

3.5 Verilog-A models

Due to a lack of time, some of the more common component were not designed by me but instead were simulated using a Verilog-A model. Verilog-A is a hardware description laguage very popular in the industry. The only components that use a verilog model are the voltage multiplier and the opAmp.

3.5.1 Operational amplifier

This component is the very famous opAmp. This specific component was not designed for the thesis, it required a current range that did not allow to use ones already made by members of the research group. The choice was then to use a verilog-A model, and then design one if time allows it.

$$v_{out} = \mu \cdot (v_+ - v_-) \tag{3.9}$$

Equation (3.9) is the equation used in the verilog-A model. It makes it so it behaves like an ideal opAmp. For this thesis μ has been set to $\mu = 10^5$.

3.5.2 Voltage multiplier

This component while far less popular than the latter, is just as useful for our specific use. It allows us to multiply, as its name implies, two voltages. It is used to compute the pointwise multiplications of the LSTM or GRU (figures 2.4, 2.6 and 2.7).

However this part is tricky. Indeed, because of the voltage conversion (table 3.1), the equation needs to take the conversion into consideration. The final equation needs to be equation (3.10) in order to have $1 \cdot 1 = 1$ ($v_{in_1} \cdot v_{in_2} = 1V$, with $v_{in_1} = v_{in_2} = 1V$)

$$v_{out} = 10 \cdot (v_{in_1} - v_{cm}) \cdot (v_{in_2} - v_{cm}) + v_{cm} \tag{3.10}$$

This is taken care of using in reality two parts, the actual voltage multiplier (equation (3.11)) and a non inverting amplifier (equation (3.12)), the circuit of which is available in section A.2.

$$v_{voltMult} = -(v_{in_1} - v_{cm}) \cdot (v_{in_2} - v_{cm}) + v_{cm} \tag{3.11}$$

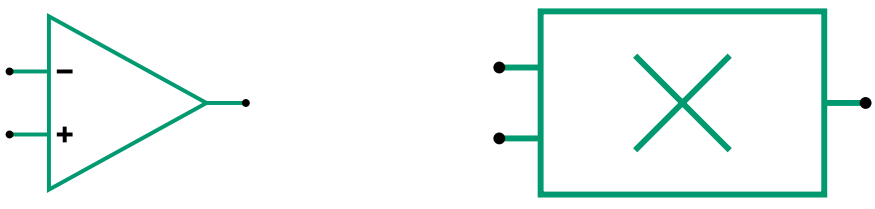
$$v_{out} = -(v_{voltMult} - v_{cm}) \cdot 10 + v_{cm} \tag{3.12}$$

Where v_{out} is the output voltage the inverting amplifier (section A.2), $v_{voltMult}$ is the out voltage of the voltage multiplier itself and v_{in_1} and v_{in_2} are the input voltages.

The equation (3.11) is assumed possible because of the actual voltage multiplier's datasheet available at [37].

3.5.3 Symbols

The symbols for those two components are the ones in figure 3.13.



(a) opAmp's symbol

(b) Voltage multiplier's symbol

Figure 3.13: Symbols used for the verilog-A components

The opAmp uses its classic symbol (figure 3.13a) while the voltage multiplier uses a custom symbol (figure 3.13b).

3.6 LSTM analog implementation

3.6.1 Circuit

This section describes the circuit of an LSTM with an input vector of size n_i, a n_h hidden states, a serial size of n_s and n_{ts} time steps. $n_o = n_h/n_s$ is going to be used for future references in this section. In order for the crossbar array to be used n_o must be an integer, in other words, n_s must divide n_h. The circuit (shown in figure 3.17) is pretty complex and contains numerous parts that require explaination.

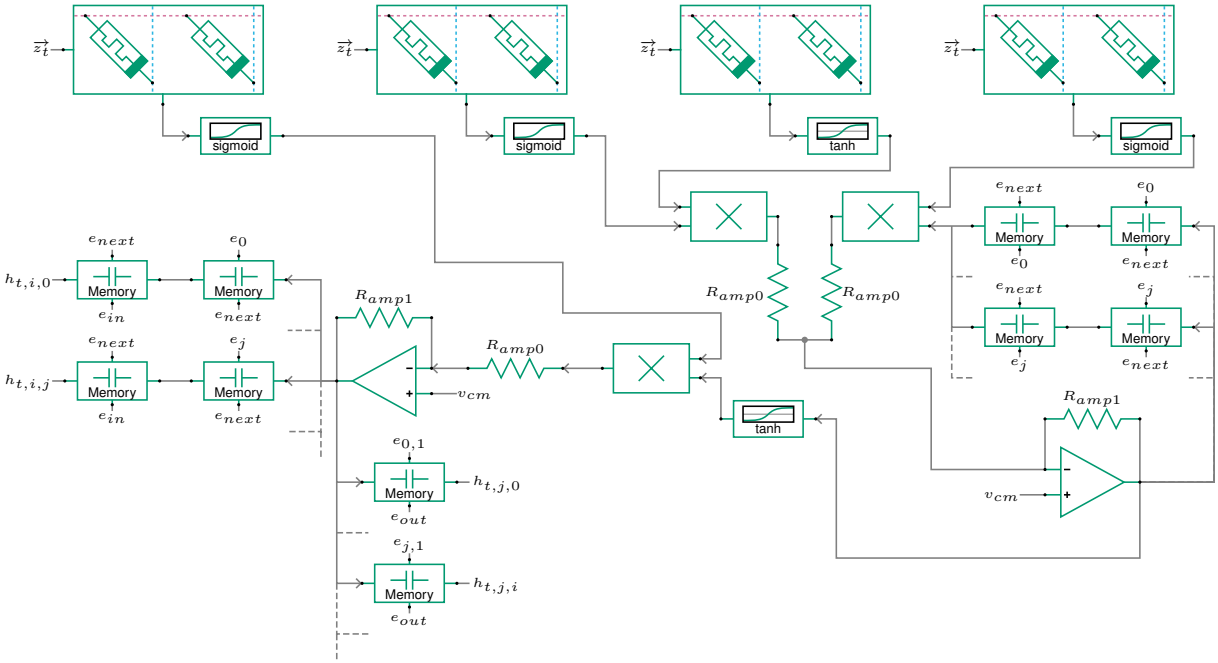


Figure 3.14: LSTM circuit

The system is built using crossbar array (section 3.3) with ($n_i + n_h + 1$,n_o, n_s) as parameters. The different vectors and variables present in the schematic have to be described :

- $\overrightarrow{x_t}$: This is the input vector for the LSTM circuit at time t. It has a size of n_i.
- $\overrightarrow{h_t}$: This is the hidden layer vector for the feedback connections, it is defined as $\overrightarrow{h_t} = (\overrightarrow{h_{t,i}})\forall i \in [\![0, n_o - 1]\!]$, with $\overrightarrow{h_{t,i}} = (h_{t,i,j})\forall j \in [\![0, n_s - 1]\!]$.
- $\overrightarrow{z_t}$ is the input of the crossbar but not the input of the LSTM. This vector is there to lighten the informations on the schematic (figure 3.17). $\overrightarrow{z_t}$ is defined by $\overrightarrow{z_t} = (\overrightarrow{x_t}, \overrightarrow{h_{t-1}}, 1)$.
- $e_{j,0}$ and $e_{j,1}$ are two enable flags that respectively represent the first and second half of e_j.
- e_{in} and e_{out} are the flags used to enable the hidden state values to go to the input (feedback connection) or to the output of the circuit.

- e_{next} is the enable flag on in between two time steps.
- R_{amp0} and R_{amp1} are the two resistances used to amplify the output voltage of the voltage multipliers. The ratio of the resistances value must of $\frac{R_{amp1}}{R_{amp0}} = 10$. The resistances must stay arround the values of the resistances used around the circuit, especially those of the memristors.

The wires coming into the crossbar are a bus of size $n_i + n_h + 1$ and the output of the crossbar is a bus of size n_o (figure 3.11). This is why everything in the system apart from the crossbar arrays is only shown once in figure 3.17 but in reality those components are present n_o times. Those extra components are needed in order for the parallel channels to work.

3.6.2 Doubling memory cells

3.6.2.A Feedback hidden states

In order for the system to work in serial mode, the memory cells of the output need to be doubled. This is done in figure 3.17, and allows for the hidden states to be saved for the next stage. Indeed, if using the system in serial mode with only one line of memory cells, the old hidden state ($\overrightarrow{h_{t-1}}$) is slowly overwritten by the current hidden state ($\overrightarrow{h_t}$). As soon as the first n_o serial values are computed, they are going to override the old ones, still required for the following serial values. Figure 3.15 shows that the value in the memory cell changes too early in the cycle and gives a bad input for the next serial values. The value given to the input of the LSTM should always be the old hidden state ($\overrightarrow{h_{t-1}}$).

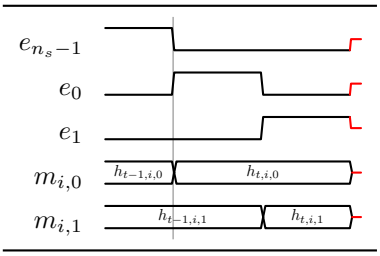


Figure 3.15: Time diagram of the values in the different memory cells ($m_{i,j}$ being the value stored in the i^{th} parallel memory cells for the j^{th} serial value). It is assumed that $n_s > 1$ (the system is in serial mode). This graph does not take into account the pauses e_{next} between the time steps.

Using two memory cells when in serial mode allows for this issue to be elevated. The values are transfered from one level to an other at the end of the time step. That way, the values are updated safely during each time step, without changing the final output.

When the system is running is a fully parallel mode, doubling the memory cells still causes a problem. The enable in and enable out flags cannot be high at the same time when the memory cell has a feedback connection, the signal would interfere with it self and change its own value. Potential solutions are discussed in section 5.2.5.

3.6.2.B Cell states

Cell states have the same solution to a similar problem. Here the problem is that the memory cell has a feedback connection and needs to be activated within one serial step. The value of the old cell

state needs to be used to compute the current values for the hidden states ($\overrightarrow{h_t}$) and cell states ($\overrightarrow{c_t}$). The issue is that at the value in the memory cell needs to be conserved until the next serial step. (section 5.2.5).

Once again, when the system is running in fully parallel mode, only one memory cell line is required for a normal behavior.

3.6.3 Serialization/Parallelization

Using an LSTM in serial mode is very beneficial as it divides the number of point wise circuit by n_s. There is a compromise to do in order to choose the serial size (n_s) :

- When the serial size (n_s) increases, the inference time increases with a factor of $O(n_s)$ while the onChip area decrease with the same factor.
- When the serial size (n_s) decreases, similarly, the onChip area increases with a factor of $O(n_s)$ while the inference time decreases by the same factor.

The overall onChip area is linked to the number of hidden states (n_h) by $O(n_h^2)$. This means that the serial size (n_s) will be set depending on the limiting factor of our system (onChip area or inference time).

3.6.4 Symbol

The symbol of the LSTM circuit can be found in figure 3.16. It depends on several parameters, the number of inputs (n_i), the number of hidden states (n_h), the serial size of the system (n_s) and the number of time steps (n_{ts}).

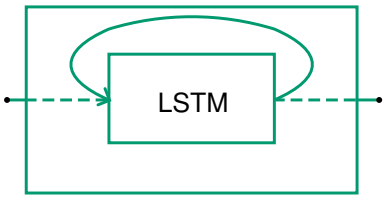


Figure 3.16: Symbol used for the LSTM circuit

The approximate onChip area for the crossbar circuit depends on the previously defined parameters.

$$\begin{gathered} A_{lstm}(n_i, n_h, n_s) = 4 \cdot A_{xbar}(n_i, \tfrac{n_h}{n_s}, n_s) \\ +\tfrac{n_h}{n_s} \cdot (5 \cdot A_{af} + 3 \cdot A_{voltMult} + 3 \cdot A_{R_{amp0}} + 2 \cdot A_{R_{amp1}} + 2 \cdot A_{opAmp}) \\ +5 \cdot n_h \cdot A_{memcell} \end{gathered} \tag{3.13}$$

3.7 GRU analog implementation

3.7.1 Circuit

This section describes the circuit of an encoder GRU with an input vector of size n_i, a n_h hidden states and n_{ts} time steps. The decoder GRU could also be implemented in an analog circuit, but the

choice was made to focus on the encoder GRU. The GRU is by its nature very similar to the LSTM.

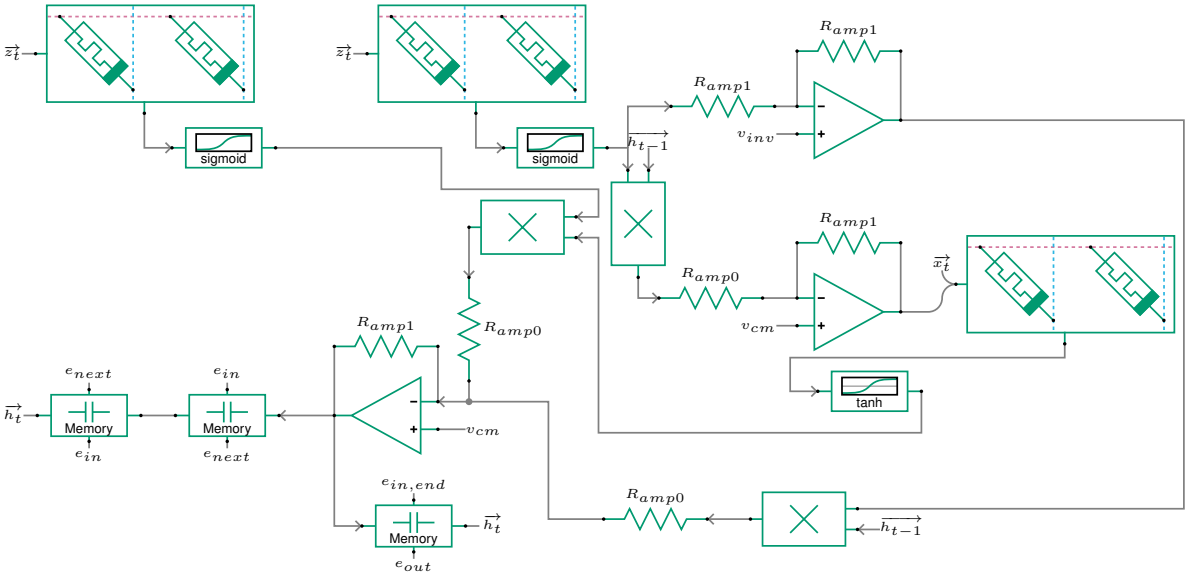


Figure 3.17: GRU circuit

The system is built, once again, using crossbar array (section 3.3) with ($n_i + n_h + 1$,n_h, 1) as parameters.

The different vectors and variables present in the schematic have to be described :

- $\overrightarrow{x_t}$: This is the input vector for the LSTM circuit at time t. It has a size of n_i.
- $\overrightarrow{h_t}$: This is the hidden layer vector for the feedback connections, it is defined as $\overrightarrow{h_t} = (h_{t,i}) \forall i \in [\![0, n_h - 1]\!]$.
- $\overrightarrow{z_t}$ is the input of the crossbar but not the input of the LSTM. This vector is there to lighten the informations on the schematic (figure 3.17). $\overrightarrow{z_t}$ is defined by $\overrightarrow{z_t} = (\overrightarrow{x_t}, \overrightarrow{h_{t-1}}, \overrightarrow{b})$.
- e_{in} and e_{out} are the flags used to enable the hidden state values to go to the input (feedback connection) or to the output of the circuit.
- $e_{in,end}$ is the second half of the e_{in} flag.
- e_{next} is the enable flag on in between two time steps.
- R_{amp0} and R_{amp1} are the two resistances used to amplify the output voltage of the voltage multipliers. The ratio of the resistances value must of $\frac{R_{amp1}}{R_{amp0}} = 10$. The resistances must stay arround the values of the resistances used around the circuit, especially those of the memristors.
- v_{inv} is the voltage used to perform the $1 - x$ operation.

The wires are all buses of different sizes. All the components expect from the crossbar, are only shown once but are in the real circuit present for every wire of the bus.

3.7.2 The $1 - x$ operation

A clever trick had to be employed in order to succesfully compute the function. The function is defined as in equation (3.14).

$$f(x) = 1 - x \tag{3.14}$$

The solution found to do compute the operation consist of using an opAmp as an inverter around V_{inv}, the formula is available in equation (3.15).

$$f_v(v) = -(v - v_{inv}) + v_{inv} = 2 \cdot v_{inv} - v \tag{3.15}$$

Writing the same operation in voltage gives us equation (3.16).

$$f_v(v) = (1 - v_{cm}) - (v - v_{cm}) + v_{cm} = 0.1 - v + 2 \cdot v_{cm} \tag{3.16}$$

It can thus be found that the value for v_{inv} is $v_{inv} = v_{cm} + 0.05$

3.7.3 Serialization/Parallelization

As opposed to the to the LSTM, the GRU can not be serialized. This is due to a mathematical issue. Indeed in the equation of the candidate hidden state is a problem for the serialization of the circuit. The equation (2.19) from section 2.5 is repeated in equation (3.17) for better lisibility of the thesis.

$$\overrightarrow{ch}_t = tanh(\overrightarrow{x}_t \cdot W_{hx} + (\overrightarrow{r}_t \odot \overrightarrow{h}_{t-1}) \cdot W_{hh} + \overrightarrow{b}_h) \tag{3.17}$$

As equation (3.17) shows, the reset gate's results vector needs to be multiplied with the the hidden state's previous vector. However, while the hidden state's previous vector is fully available in the memory cells, the reset gate's vector needs to be computed using the current inputs.

If the circuit were serialized, a full cycle would be required to compute the reset gate, that is itself required to compute the candidate hidden state.

That is why, the GRU would take twice as much time to compute a time step if it were serialized. It is not worth it to double the computation time to save a small amount of onChip area.

3.7.4 Symbol

The symbol of the GRU circuit is available in figure 3.16. It depends on a few parameters namely, the number of inputs (n_i), the number of hidden states (n_h) and the number of time steps (n_{ts}).

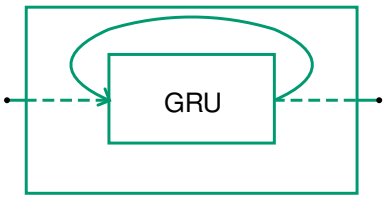


Figure 3.18: Symbol used for the GRU circuit

The approximate onChip area for the crossbar circuit depends on the previously defined parameters.

$$\begin{gathered} A_{gru}(n_i, n_h) = 4 \cdot A_{xbar}(n_i, n_h, 1) \\ +\frac{n_h}{n_s} \cdot (5 \cdot A_{af} + 3 \cdot A_{voltMult} + 3 \cdot A_{R_{amp0}} + 2 \cdot A_{R_{amp1}} + 2 \cdot A_{opAmp}) \\ +5 \cdot n_h \cdot A_{memcell} \end{gathered} \tag{3.18}$$

3.8 Weights generation

The choice for the weight generation was to use Keras API with the tensorflow framework. However, both tensorflow and pyTorch were tried and were giving similar results, the final choice of using tensorflow was made because it is the most popular among the research group.

3.8.1 Training the weights

Generating the weights requires to train a NN. This is done in Keras by creating the model, in other words the architecture required to solve the problem. This model contains the description of the NN, such as the type of RNN and the Dense layers that would come before of after the RNN. Two out of the three RNNs available in Keras are trained in this work, the LSTM and the GRU.

Training can only be done outside of the circuit as this state of the project. However, in order to train the weights as closely as they would have been on a real circuit, the activation functions used for the training are the custom activation functions generated by the dedicated circuit (section 3.1). In other words, the activation functions used are the ones shown in figure 3.3. This allows to train the weights as they almost (considering the activation functions are recreated using 51 simulated points) as they would have been in the real circuit.

All the weight trainings are done using MSE loss function and Adam optimizer [38].

3.8.2 Weights constaint

The weights need to be contrained during the training to make sure that the voltage of the analog circuit stays within its operating voltage range ($[0, v_{dd}]$). To avoid any unwanted behavior when the system supposedly goes out of range, the weights in the LSTM are contrained. This constrain is not fixed and depends on the architecture it is generated with.

The worst case scenario for an LSTM is when every value reaches the voltage threshold (v_{read}) (section 3.3) and every weight is maximized (w_{max}). When that is the case the output of the VMMs is found in equation (3.19).

$$v_{read} \cdot w_{max} \cdot (n_i + n_h) + w_{max} + v_{cm} = v_{dd} \tag{3.19}$$

We can then determine the maximum acceptable value for the weights (w_{max}), this value has thus been determined to be the one in equation (3.20).

$$w_{max} = \frac{v_{dd} - v_{cm}}{v_{read} \cdot (n_i + n_h) + 1} \tag{3.20}$$

The analog system being completly centered and symetrical around v_{cm} the value for the minimum weight is the opposite to the maximum one (equation (3.21))

$$w_{min} = -w_{max} = -\frac{v_{dd} - v_{cm}}{v_{read} \cdot (n_i + n_h) + 1} \tag{3.21}$$

A way to remove those constraints is discussed in section 5.2.6. Having a constraint on the weights limits the performance of the final NN.

3.8.3 Exporting the weights

Once all the parameters (number of hidden states, number of dense layers, etc) have been set. The weights can be exported using the required weight repartition layed out in section 3.9. A description of the architecture is also saved along the weights. This file can now be used as the input for the netlist generator (section 3.9).

The code used for all the weights generations is available at [39]. The code containing the functions to save the weights to a file is available in section B.2.

3.9 Netlist generation

In order to be able to run the simulation with different kinds of NN architecture. The point of the part is thus to explain how the netlist generator tool works. The tool also work to generate any architecture with the supported layers (listed below).

This is done by generating a SPICE netlist using a python script. A SPICE netlist was chosen because of **Cadence**'s *Virtuoso* limitations. Indeed, Verilog netlists can be imported with a downside, component's parameters cannot be imported. This is very limiting for this thesis' use case (required to set the resistors' resistance and thus the weights). SPICE netlists can import parameters for the components and is open source and by such well documented. The generator script takes in a few parameters :

- The first parameter is the number of input we use our system. This is the size of the first input vector ($\overrightarrow{x_t}$).
- The next parameter is the number of time steps. Everything about the LSTM time steps is all explained in section 2.4.
- The serial size of the crossbar arrays, as described in section 3.3, used in the LSTM network. This parameter must divide the number of hidden states in the LSTMs layers.
- The files containing different informations about the model. This file contains the type of NN architecture and the weights associated with each layer. The weights in the files have to be organized using a specific model (section 3.9.2).
- Finally the name of the file in which the output of the script (the netlist) will be written to.

3.9.1 Available layers

As of the writing of the thesis, the netlist generator can generate the netlist for the NP LSTM and the GRU. Other type of NNs can be added to the code.

3.9.2 Weights storage

The weights are stored in a single file. This file contains a brief description of the architecture being used. The first index stores this desciption. The following indexes are for the weights of each layer, in the order given in the description. Depending on the layer used, the weights are stored a bit differently :

- Dense layer : This is basically a VMM so the weights are just sorted linearly in a list. The list is of size n^2, where n is the size of the input vector. Equation (3.22) shows the position (i) of the weight (w_i) in the matrix. This is represented in the description as "Dense(n_o)" where n_o is the number of output of the layer.
- LSTM layer : An LSTM layer contains four VMMs which can be assimilated and thus stored like a dense layer. The weights for each VMM is then stored in a sub list. This is represented in the description as "LSTM(n_h)" where n_h is the number of hidden states of the layer.
- GRU layer : An GRU layer contains three VMMs which can be assimilated and thus stored like a dense layer. The weights for each VMM is then stored in a sub list. This is represented in the description as "GRU(n_h)" where n_h is the number of hidden states of the layer.

$$\begin{bmatrix} w_0 & w_1 & \dots \\ \vdots & w_i & \vdots \\ \dots & w_{n^2-2} & w_{n^2-1} \end{bmatrix} \tag{3.22}$$

All the code can be found on my github page [40].

3.10 Conversion from weights to resistance

The weight to resistances equations has been found using equation (3.23).

$$w = R_f \cdot (G_+ - G_-) = \frac{R_f}{R_+} - \frac{R_f}{R_-} \tag{3.23}$$

First, we know from equation (3.5) that the weight is represented by equation (3.23). This means that the weights are limited in values to :

- $w_{max} = R_f \cdot (G_{max} - G_{min})$
- $w_{min} = -w_{max} = R_f \cdot (G_{min} - G_{max})$

For the rest of this chapter we consider that R_f is set to the middle point of R_{max} and R_{min} meaning that $R_f = \frac{R_{min}+R_{max}}{2}$.

Since we only have one equation (equation (3.23)) for 2 unknowns (R_+ and R_-), we need to set a second equation. This is done by centering the resistances around R_f, this means that equation (3.24) is the second equation, that makes our problem now solvable.

$$R_f = \frac{R_- + R_+}{2} \tag{3.24}$$

$$\begin{cases} w = \frac{R_f}{R_+} - \frac{R_f}{R_-} \\ R_f = \frac{R_- + R_+}{2} \end{cases} \tag{3.25}$$

By solving equation (3.25), we find the equation (3.26) that gives the values for R_- and R_+. All the steps for solving the equations can be found in section C.1.

The real value to voltage conversion (table 3.1) does not affect this part of the system as the crossbar array already works in voltages. Replacing the resistor variable in equation (3.5) by w using equation (3.23) would not change the results.

$$\begin{cases} R_+ = (w + 1 - \sqrt{w^2 + 1}) \cdot \frac{R_f}{w} \\ R_- = 2 \cdot R_f - R_+ \end{cases} \tag{3.26}$$

While solving the system (section C.1), we get two potential equations for R_+. Graphing them shows that one of them is outside of the memristor's resistance range ($[R_{min}, R_{max}]$). Leaving us with only one equation because the other one is physically unreachable.

In the present work, this step is done in python and integrated in the netlist generator script (section 3.9). The resolution of the memristor (the precision of the resistances at which the memristor can be set) is simulated by limiting the generated resistances values to having only two significant figures.

The python implementation is available in section C.1.

3.11 The datasets

To make sure that the terminal voltage does not reach the threshold as explained in section 3.3. The voltage threshold (V_{read}) was chosen to be $V_{read} = 0.1V$ because the memristors used in this thesis are purely theorical. The dataset needs to be formated not to exceed this value when converted to voltage (table 3.1), the dataset needs to be trained while not exceeding 1.

3.11.1 Airline passengers

The first dataset contains a time series of international airline passengers from January 1949 to December 1960, the data is recorded monthly and is in thousands. The dataset then contains twelve years of monthly data so 144 sample points. Although the results of this problem are not necessarely tailored to the system, in the sense that it is not time sensitive information, it is nonetheless a problem that can, in a straightforward manner, demonstrate the predictive capacity of the proposed model. It is extracted from the tutorial available at [41]. The dataset is available at [42], the name of the file containing the dataset is *airline-passengers.csv*.

This is a forecasting problem, the role of the LSTM is to predict the number of passenger flying the next month being given previous months' passenger count.

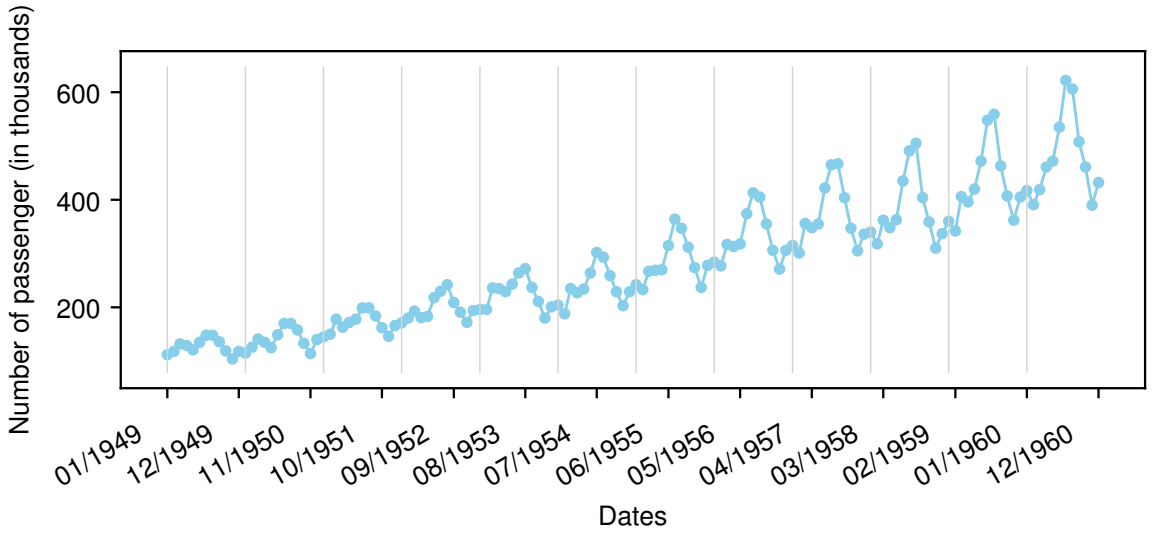


Figure 3.19: The airline dataset. The vertical lines represent a full year.

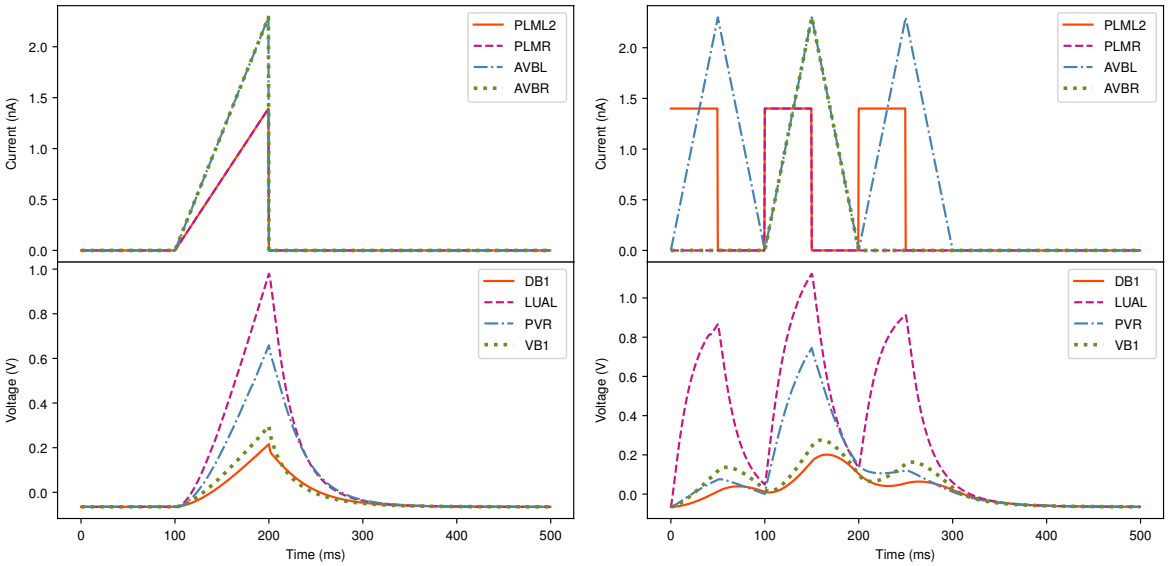
The data in the dataset has to be formatted to be used with the LSTM. The dataset contains 144 sample points. It has been transformed into 142 data points for training. This has been done by taking each values three by three. The first two values are two timesteps for the input vector and the third value being used the target value. Two third of the dataset is being used for training and the other third is used for validation.

3.11.2 Caenorhabditis elegans

This dataset is far more interesting and complex than the latter. This data set aims to use LSTMs to mimic the behavior of real neurons. As explained in [9], Caenorhabditis elegans (C. elegans) are simple organisms that are getting very popular for whole brain organization studies. The point of this problem is to reproduce the nervous system of the C. elegans. This is done using recorded data of the input of 4 neurons and the output of 4 other neurons.

The dataset is great for our study because :

- It comes from a very recent paper from the research group in which this work is also being developed.
- It aims at reproducing the behavior of the brain of a simple organism (C. elegans). In the (very) long term, a full parts of the human brain could be replaced by a very low powered chip.



(a) Training dataset sample

(b) Test dataset sample

(c) Training dataset sample

(d) Validation dataset sample

Figure 3.20: C. elegans dataset samples

The dataset is already formatted in a time depedent manner and contains a set of fourty set of input/output. Each input/output set contains $500ms$ of data with a time step of $0.5ms$, making a thousand points for each neurons. This data is recorded after applying current to the input neurons (PLML2, PLMR, AVBL, AVBR), and monitoring the output of four neurons (DB1, LUAL, PVR, VB1) that are known to have strong activity during a specific behavior of the nematode known as Forward Crawling Motion (FCM). The simulation is done using a known model of C. elegans connectome (complete overview of the brains connections) [9].

Figure 3.20 shows four sets of input/output out of the fourty that were recorded for the dataset.

The inputs for the systems are then going to be the current values for the four input neurons at every of the thousand time steps and plans to get an output as close as possible to the modeled

output for all time steps.

3.12 Running the simulation

The simulation is ran using **Cadence**'s *Virtuoso* simulator, using a parametric simulation to simulate all the inputs. The use of the parametric simulation allows to change the values for the data we want to use as input.

In order to run the simulation some variables need to be set. The time step (T) value was chosen to be $T = 8\mu s$, the pause between two LSTM steps is set to be $\frac{T}{8} = 1\mu s$. The value for the maximum and minimum resistances value of the memristor used (R_{max} and R_{min}, respectively) are set to be in acordance with the **KNOWM®**'s memristors [43]. The resistance values used for the simulation are thus $R_{max} = 10^6\Omega = 1M\Omega$ and $R_{min} = 10^4\Omega = 10k\Omega$.

Once everything is set, the simulation can be started. The result are recorded by looking at the output net when at the time of output activation. The name of the output nets is outputed by the netlist generator script (section 3.9).

4

Results

Contents

This chapter contains all the results of the datasets presented in section 3.11. The results of both the digital predictions and analog predictions. The analog predictions are the outputs of the analog circuit. Both the analog and digital predictions are using the same set of weights. Those weights were all trained digitally using the tensorflow python library.

4.1 Airline dataset

4.1.1 Network configuration

The layers used to solve this problem are listed below :

- An LSTM with four hidden states ($n_h = 4$) and an input with feature size of one and two time steps.
- A Dense layer with an output size of one.

Figure 4.1 is a graphical representation of the model just described.

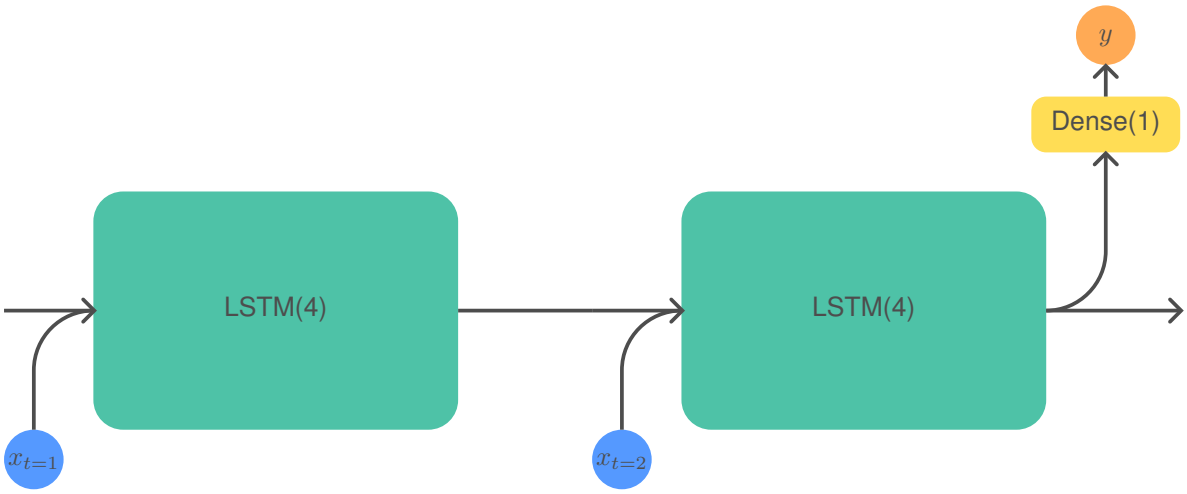


Figure 4.1: Model used to solve the airline passengers problem

The same network configuration using a GRU layer instead of an LSTM layer. The number of hidden states is the same.

The NN was trained for 300 epochs.

The timing of the flags can be quite hard to keep track of, for this reason all of them are going to be shown for the full duration of the system execution to get one output.

Figure 4.10 shows the time diagram of the entire execution using the parameters discussed earlier.

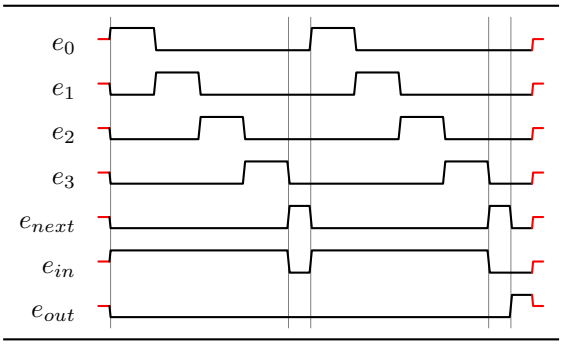


Figure 4.2: Flags time diagram for the airline problem with $n_s = 4$

4.1.2 Digital results

The digital results of the NN once trained are very important. They are the results that the analog system will have to reproduce. Furthermore, this part is used to measure the impact of the custom activation functions used for training (section 3.1).

4.1.2.A LSTM predictions

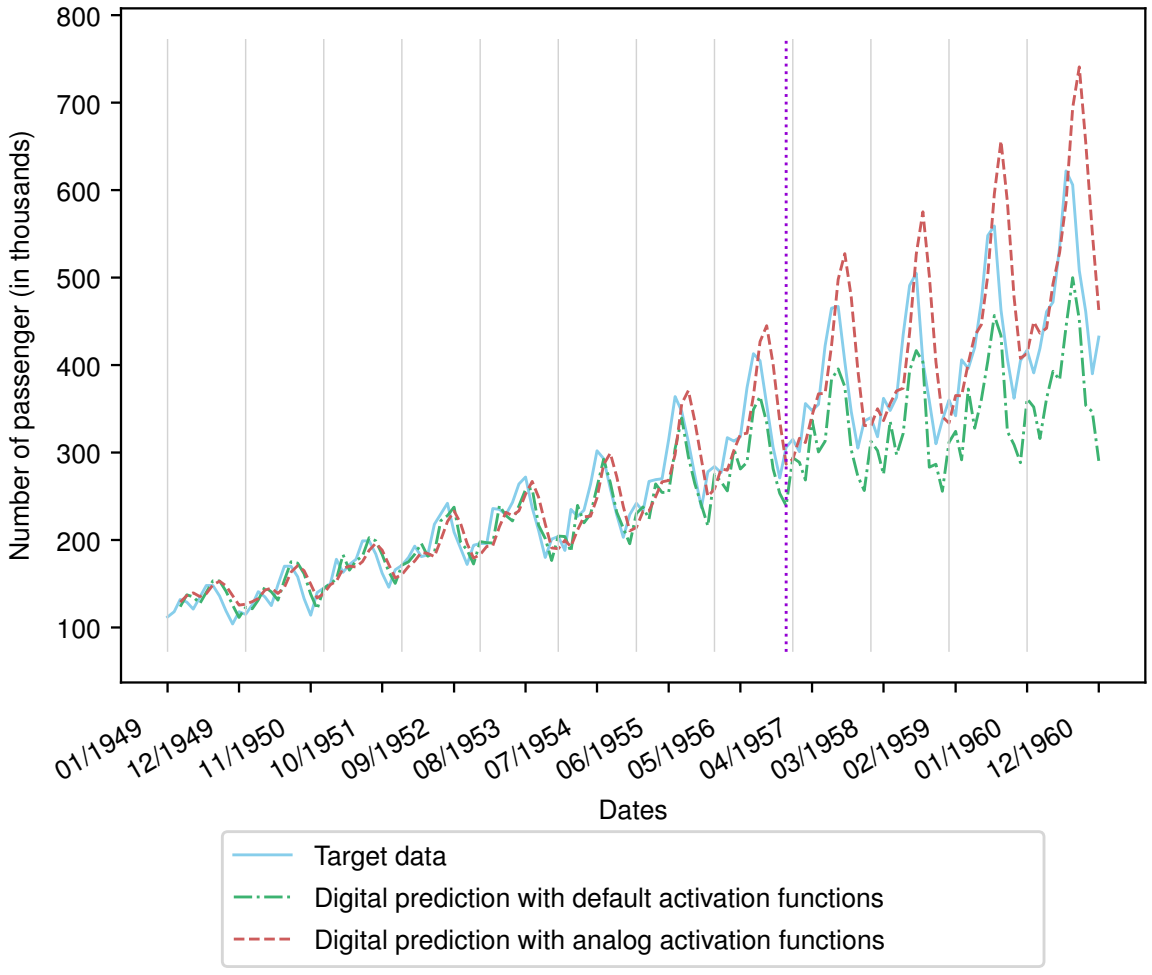


Figure 4.3: Graph of the digital predictions for airline dataset. The dotted vertical line shows the limit of the data used for training and the one used for validation.

Figure 4.3 shows the results of the NN next to the orinal data. The results are visually very close. The error is quantified using RMSE function. All those curves error to each other are measured and displayed in table 4.1.

RMSE	Target data	Digital prediction with default activation functions	Digital prediction with analog activation functions
Target data		54.6	52.3
Digital prediction with default activation functions	54.6		78.6
Digital prediction with analog activation functions	52.3	78.6	

Table 4.1: RMSEs of each curve to the others

Both predictions have similar RMSE to the target curve. They have a RMSE of 54.6 for the LSTM using the default activation functions (sigmoid and tanh functions) and 52.3 for the one using the analog activation functions (section 3.1). The error difference is so low (about two thousands passengers) that the difference may just come from the training and the initial values of the weights. They both approach the target curve pretty well. These error could be lowered by training for more epochs. This is not done because it diverges too much from the focus of the thesis.

The two predictions are very different from each other, the error rate between the two is of 78.6, which is significantly higher than the RMSE of the curves to the target curve.

Computing the NN with or without the analog activation functions does not seem to affect the error of the predictions, experience has shown that the LSTM using the analog activation functions seem to predict higher curves than the target and the LSTM using the default activation functions tends to predict curves that are under the target, the reasons for such a difference are unknown. This is simply an observation after a few different runs.

4.1.2.B GRU predictions

Unlike for the predictions using an LSTM layer, the weights were only trained using the analog activation functions. The effect of using the default activation functions is already shown with the LSTM layer.

The graph of the resulting prediction is the one in figure 4.4, the error rate is also in the figure.

Interestingly, the results look much better when using the GRU layer. The error rate of $36,6$ confirms that as it is almost half of what it was with the LSTM layer (table 4.1). That simply means that the GRU layer is better at handling this problem, as was previously stated in section 2.5.

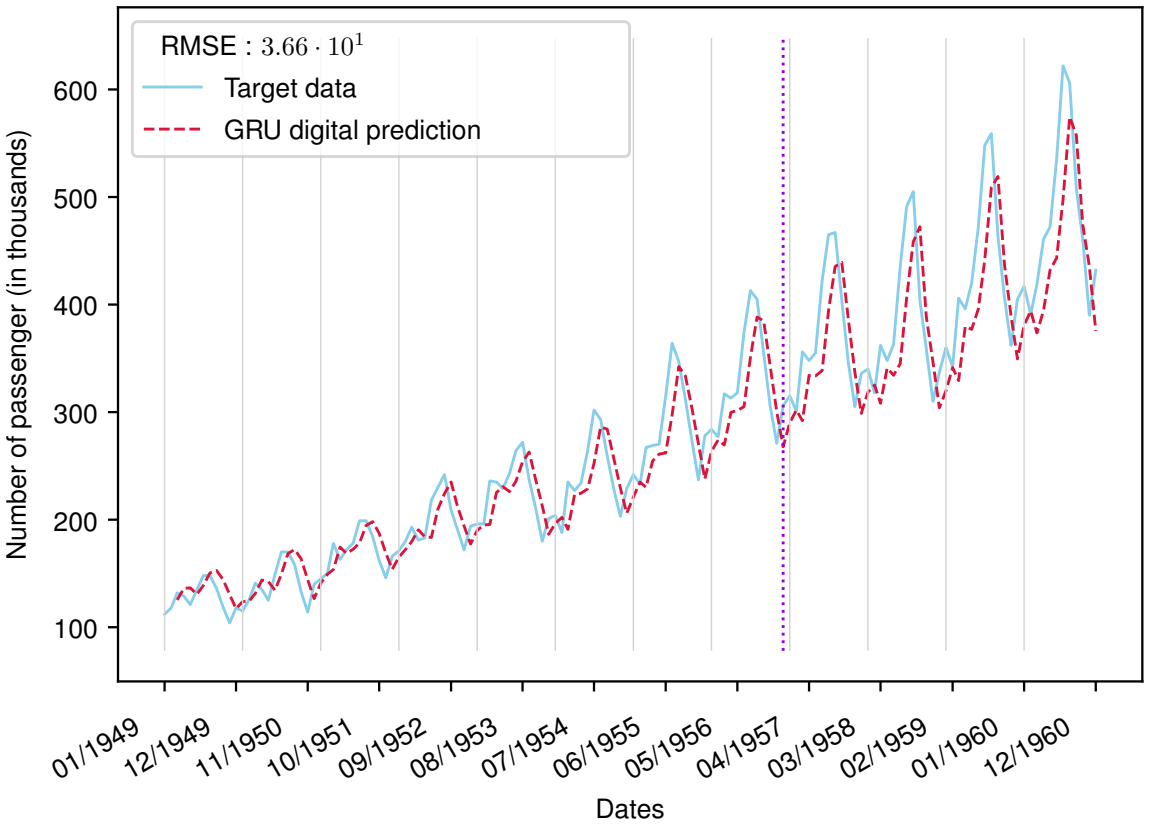


Figure 4.4: Graph of the digital predictions along with the RMSE for the airline dataset using a GRU layer. The dotted vertical line shows the limit of the data used for training and the one used for validation.

4.1.3 Analog results trained with default activation functions

This subsection shows the importance of using the analog activation functions in the training phase of the NN. The LSTM layer being the only to have been trained using the default activation functions the only results presented here are using an LSTM layer.

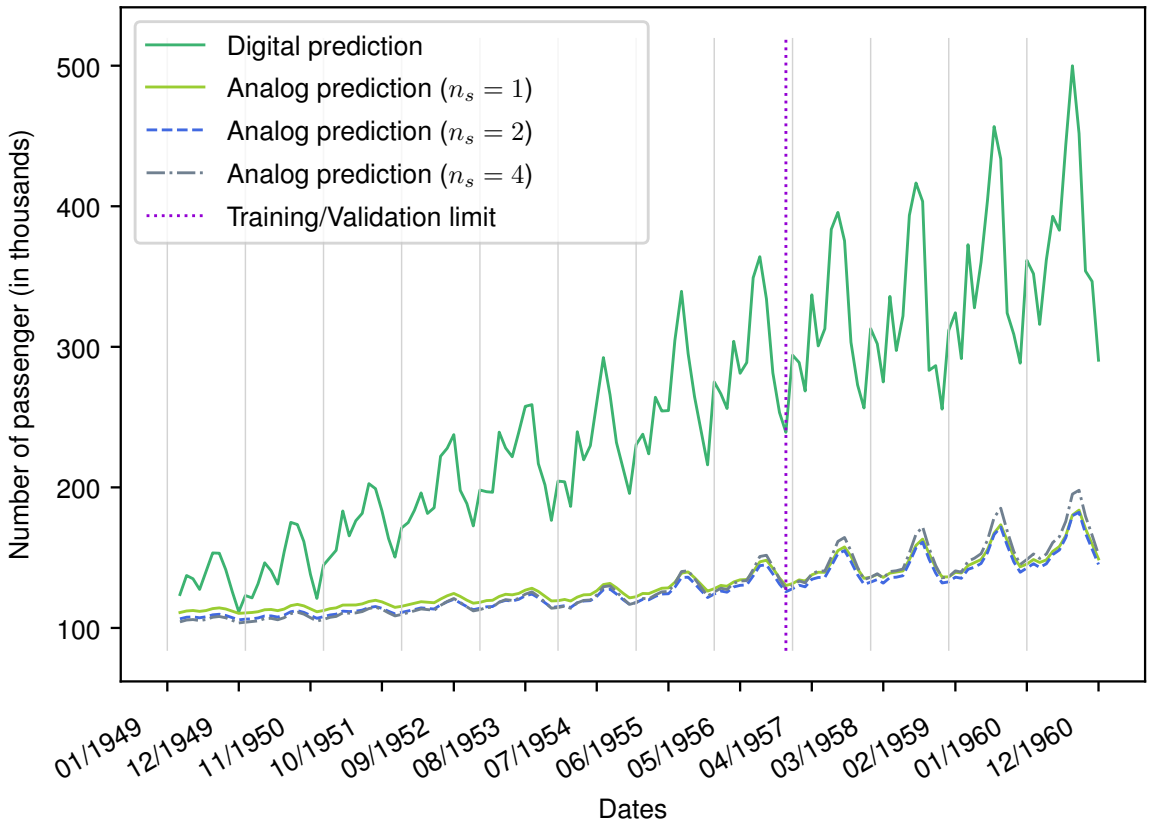


Figure 4.5: Analog predictions trained with the default activation functions.

As expected, the analog predictions are very far off the target curve. Which, in this case, is not the target dataset but the digital prediction ran with the default activation functions. Indeed, the goal of the analog system is to reproduce as closely as possible the digital prediction.

RMSE	Analog prediction		
	$n_s = 1$	$n_s = 2$	$n_s = 4$
Digital prediction with default activation functions	141	144	140

Table 4.2: RMSEs of each analog prediction to their associated digital prediction

As can be observed on figure 4.5, the analog predictions are not on par with what is expected at all. This is confirmed by the error rates found in table 4.2, the error rates are not in an acceptable range at all. This error would be even higher to the target dataset as the digital prediction using the default activation functions is already under than the target curve.

However, the prediction do not seem to be completely off. The curves seem to follow the shape of its target, and by extension of the target data. This deserves a closer look.

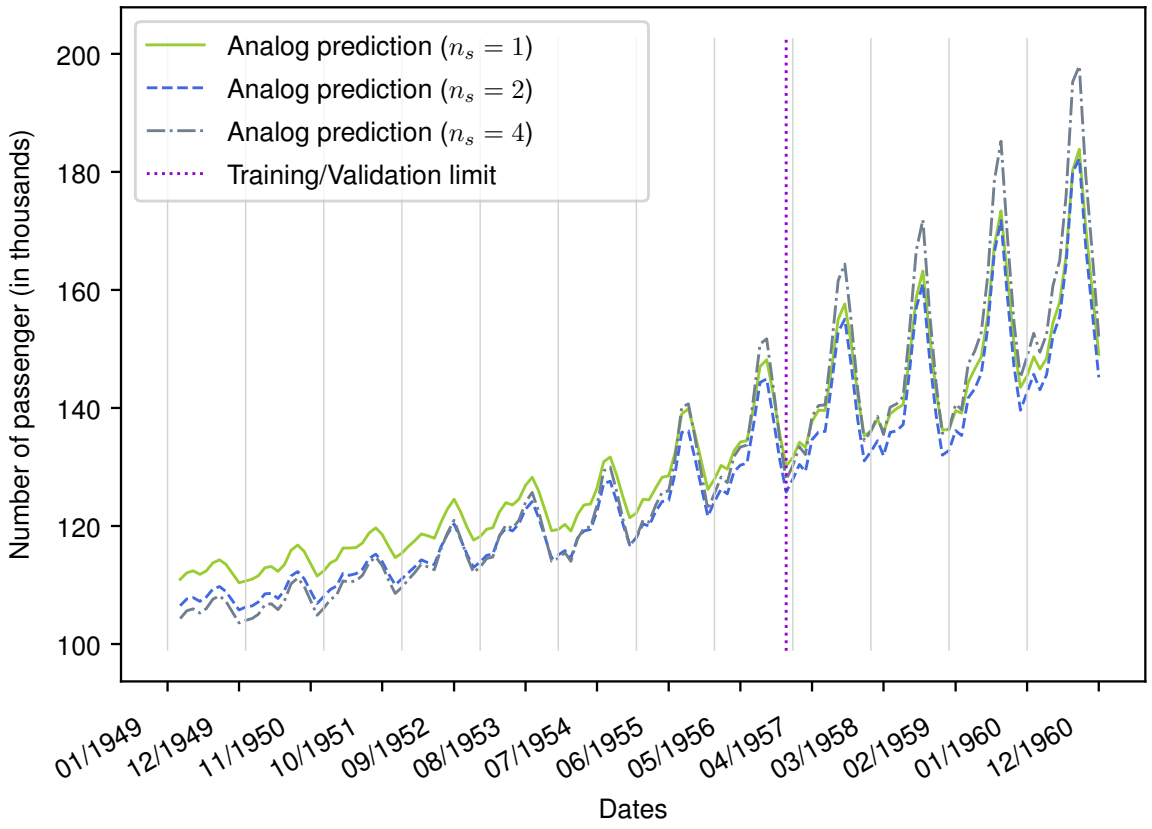


Figure 4.6: Analog predictions trained with the default activation functions zoomed on the the analog predictions.

As expected, in figure 4.6 the trend of the curves is very similar to the original dataset (figure 3.19) and the predictions are very close to each others no matter the serial size used (n_s).

RMSE		Analog prediction		
		$n_s = 1$	$n_s = 2$	$n_s = 4$
Analog prediction	$n_s = 1$		3.97	5.05
	$n_s = 2$	3.97		4.78
	$n_s = 4$	5.05	4.78	

Table 4.3: RMSEs of each analog prediction to the others depending on the serial size (n_s)

This similarity is further proven by the errors shown in table 4.3. The erros are 3.97, 4.78 and 5.05. Those error are all around four thousands passengers, which is negligible in our case. That shows a certain stability of the system when using different values of serial size (n_s). The error rate is also quite low because of the lower results obtained.

The predicted results generated by the analog system seem to be on a different scale while still following the overall aspect of the dataset. The down scale of those analog prediction would also mean that the errors of those predictions to each other is also scaled down.

4.1.4 Analog results trained with analog activation functions

4.1.4.A LSTM predictions

This part will show the result of the inference when the weights were trained using the analog activation functions.

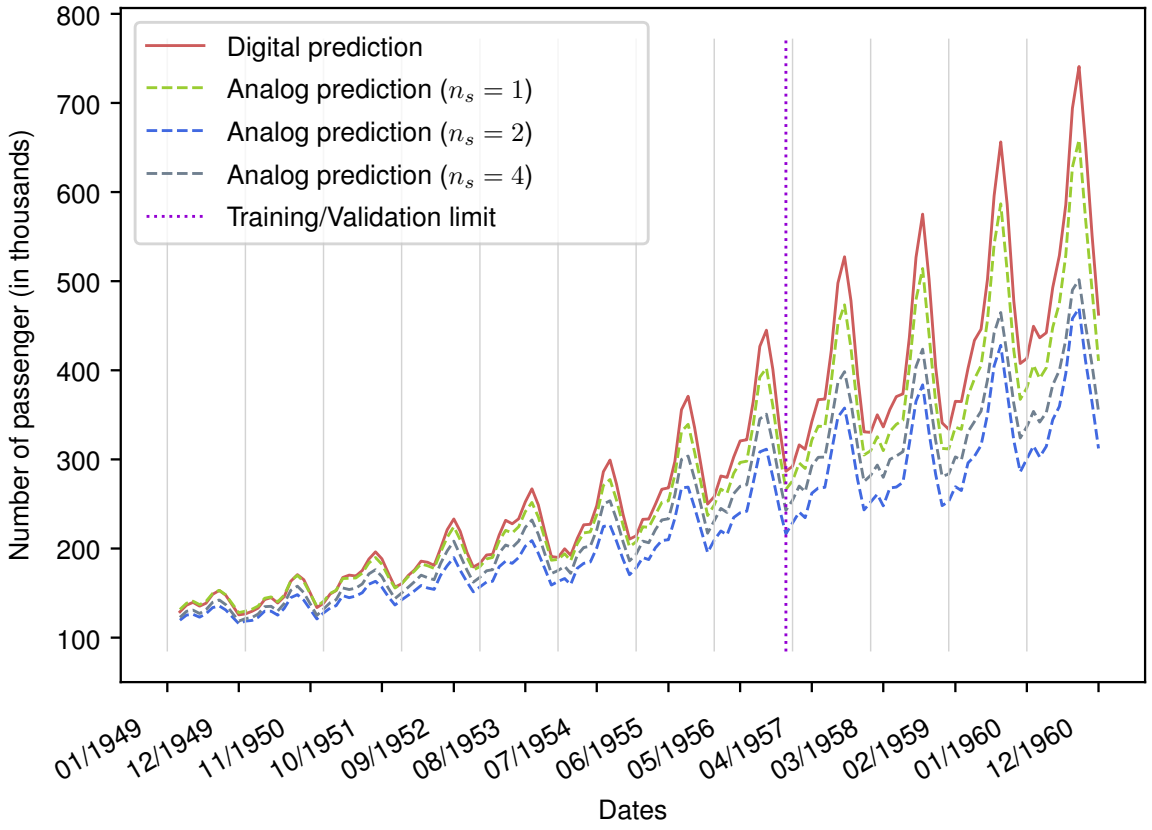


Figure 4.7: Analog predictions trained with the analog activation functions.

As can be observed in figure 4.7, the analog results are, in this case, way closer to the target values. The curves are very close to eahc other and are a definite improvement from the the ones trained using default activation functions.

RMSE	Analog prediction		
	$n_s = 1$	$n_s = 2$	$n_s = 4$
Digital prediction with analog activation functions	28.4	92.6	68.5

Table 4.4: RMSEs of each analog prediction to their associated digital prediction

Table 4.4 contains the errors of the analog predictions. It can be observed that the results the closest to the target are the ones generated using a serial size of one ($n_s = 1$). Indeed, its error (28.4) is lower than the error from the digital results (52.3) to the original target values, almost half of it. The other ones are still very close to the targeted curve, but are still lower. The errors are higher when running the system in serial mode. This higher inaccuracy is probably due to the data staying for longer in the memory cells as the time steps get longer.

RMSE		Analog prediction		
		$n_s = 1$	$n_s = 2$	$n_s = 4$
Analog prediction	$n_s = 1$		64.6	40.6
	$n_s = 2$	64.6		25.8
	$n_s = 4$	40.6	25.8	

Table 4.5: RMSEs of each analog prediction to the others depending on the serial size (n_s)

Once again, the results are quite close to each others, but as expected the errors (table 4.5) are higher then the ones found in table 4.3. This is due to the fact that the errors depend on the scale of the results, the predcitions here having a much greater range (table 4.4). The farthest appart ($n_s = 1$ to $n_s = 2$ with an error of 64.6 thousands passengers) have an error barely higher than the error of digital results to the target dataset (table 4.1)

The overall observation of those results compared to the previous ones (section 4.1.3) are how much closer to their target they are. It still is not quite perfect but simply changing the activation functions used for training improved the results by a lot. It can thus be theorized that the error will greatly decrease if the circuit is trained with the same parameters as the ones it is using for inference. In other words, the error will probably be very low when the training will happen inSitu (section 5.2.3).

4.1.4.B GRU predictions

Like explained in section 3.7, the GRU layer cannot be serialized in an analog circuit. The simulation were only ran once, for the only version available.

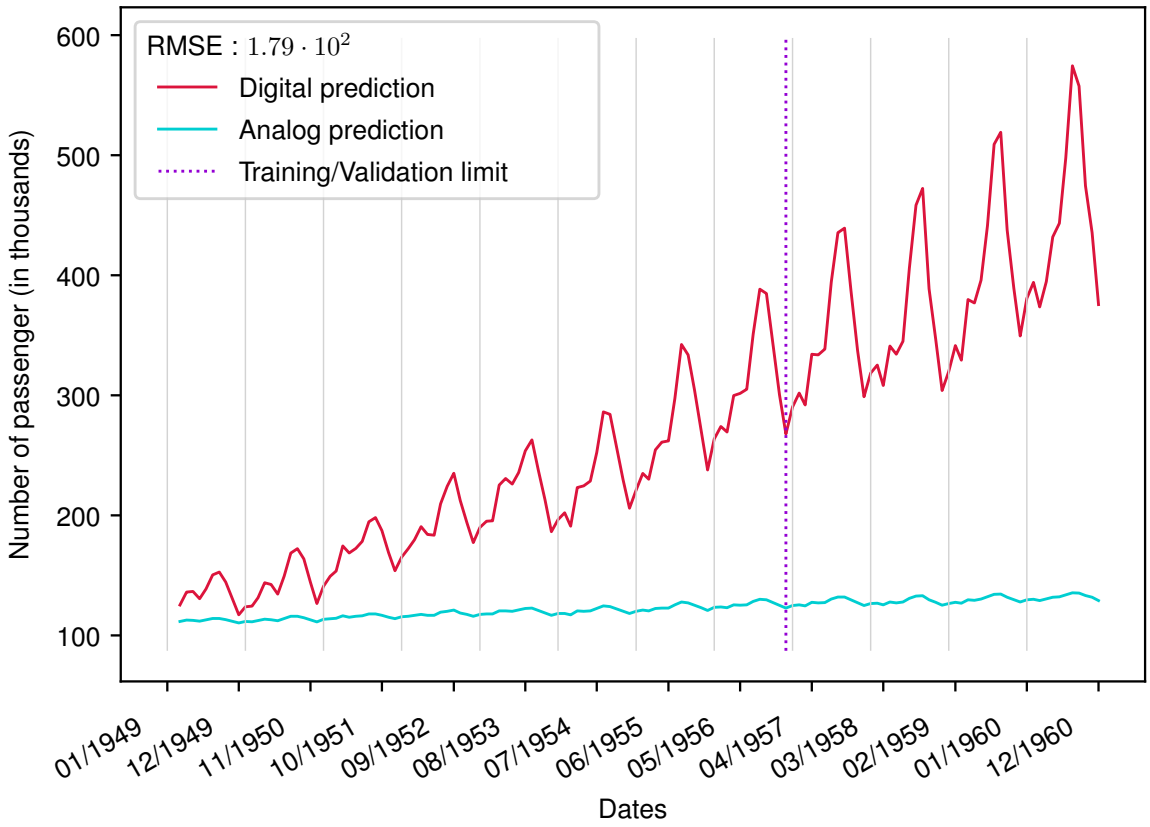


Figure 4.8: Analog prediction of the GRU analog circuit

Figure 4.8 contains the analog predcitions of the GRU circuit. The results are even worse than the results of simulation using weights generated with the default activation functions. Indeed, the error rate found in figure 4.8 is much greater than the ones in table 4.2. This means that while the circuit produces the right output shape, there is something wrong with the current version of the circuit.

4.2 C. elegans dataset

Due do the complexity of this dataset and the results associated are only trained using the analog activation function. Furthermore, the usefulness of using those activation functions was already demonstrated in section 4.1.

4.2.1 Network configuration

The layers used to solve this problem are listed below :

- An LSTM with eight hidden states and an input with feature size of four (because of the four input neurons that are considered), one thousand time steps and every time steps outputs a value for the next layer.
- A time distributed dense layer with an output size of four. It is time distributed because it needs to compute every time steps and not only the last.

The LSTM was chosen with a number of hidden states of eight because it gets the best results [9].

Figure 4.9 is a graphical representation of the model just described.

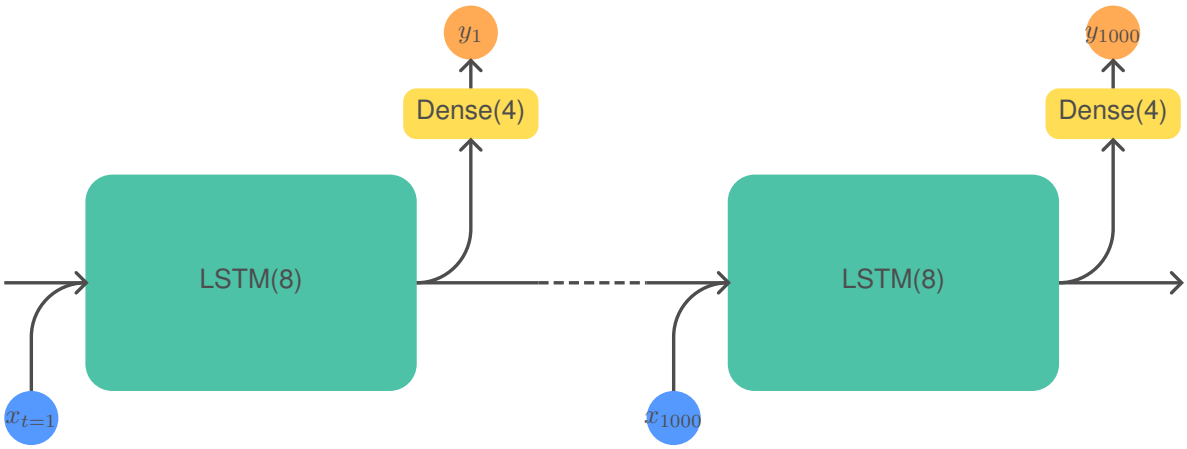


Figure 4.9: Model used to solve the C. elegans problem

The NN was trained for 1000 epochs.

The timing of the flags can be quite hard to keep track of, for this reason all of them are going to be shown for the full duration of the system execution to get one output.

Figure 4.10 shows the time diagram of the entire execution using the parameters discussed earlier.

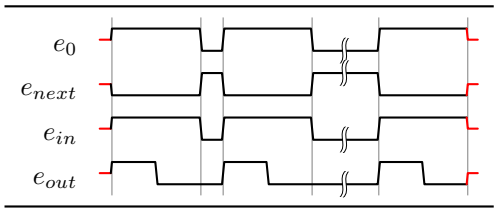
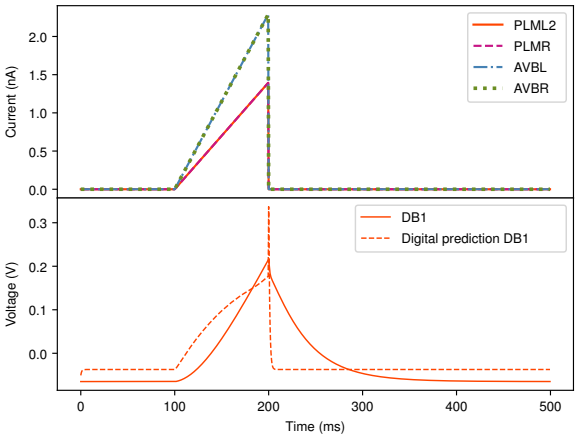


Figure 4.10: Flags time diagram for the celegans problem with $n_s = 1$

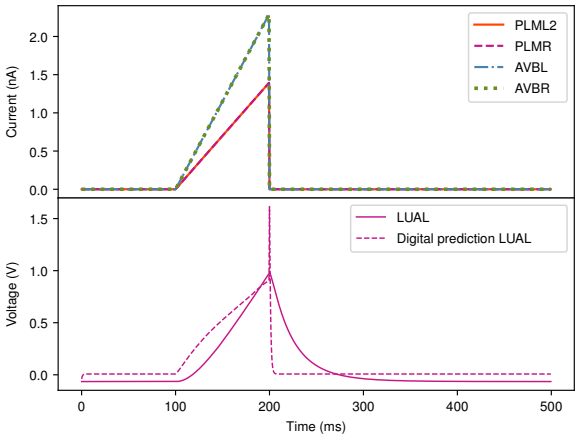
4.2.2 Digital results

The results will be analyzed by focusing on two data sequence, the first being simple inputs expecting simple outputs and the second being more complex, those sequences are the ones in figures 3.20a and 3.20d. We will identify those as sequence 5 and sequence 15 respectively.

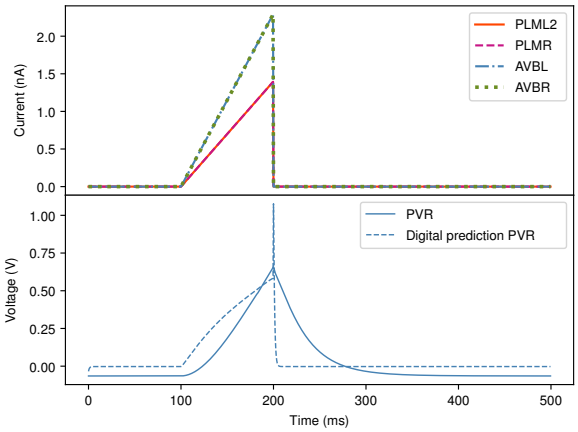
The digital results of the C. elegans dataset are quite numerous, not all of them can be displayed and analyzed. For this reason, only the two previously selected sequences will be displayed as the results of the dataset are not the focus of this thesis. The full work concerning the C. elegans dataset and problem is done in [9].



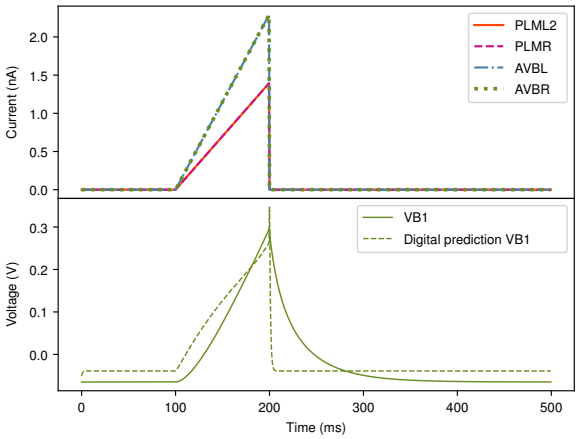
(a) Analog prediction for the DB1 neuron



(b) Analog prediction for the LUAL neuron



(c) Analog prediction for the PVR neuron



(d) Analog prediction for the VB1 neuron

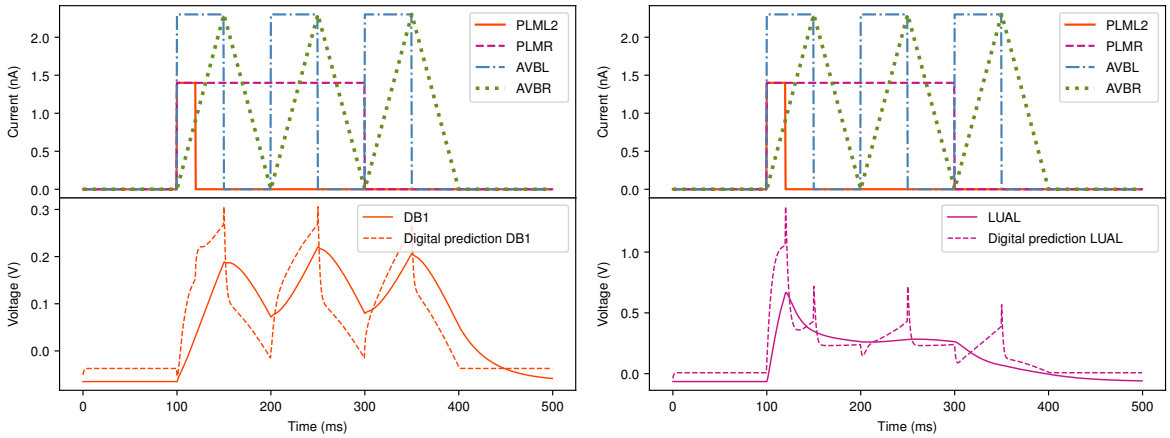
Figure 4.11: C. elegans digital responses spread out for each output neuron

Figures 4.11 and 4.12 are the digital predictions of the selected dat sequences. They all show a decent results, their RMSE can be found at table 4.6.

Neuron	BD1	LUAL	PVR	VB1
Sequence 5	$5.02 \cdot 10^{-2}$	$1.68 \cdot 10^{-1}$	$1.32 \cdot 10^{-1}$	$5.07 \cdot 10^{-2}$
Sequence 15	$6.98 \cdot 10^{-2}$	$1.39 \cdot 10^{-1}$	$1.86 \cdot 10^{-1}$	$6.63 \cdot 10^{-2}$

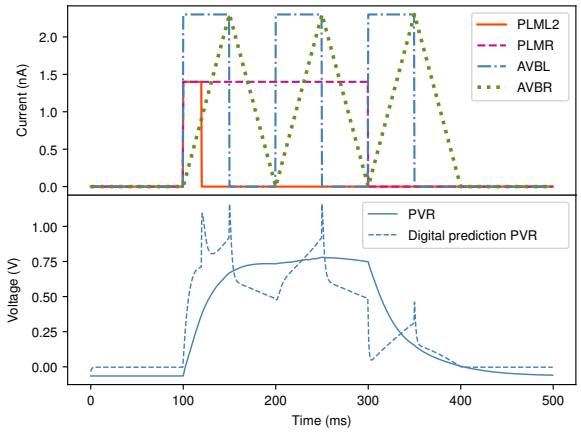
Table 4.6: RMSEs of each neuron's prediction for the two selected sequences

A pattern can be observed here, the error rates are lower for the DB1 and VB1 neurons.

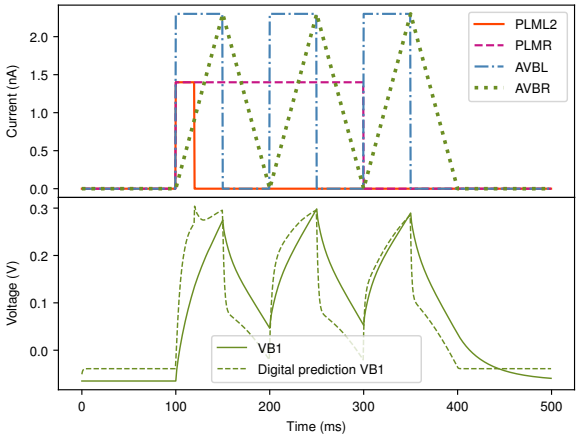


(a) Analog prediction for the DB1 neuron

(b) Analog prediction for the LUAL neuron



(c) Analog prediction for the PVR neuron



(d) Analog prediction for the VB1 neuron

Figure 4.12: C. elegans digital responses spread out for each output neuron

4.2.3 Analog results of sequence 5

It has been established in section 4.1, that the analog circuit with a serial size of one ($n_s = 1$) shows a lower RMSE to the digital predictions than the digital predictions.

Due to the complexity of the dataset, the simulation were only ran using a serial size of one ($n_s = 1$) to get the best looking results.

This part will show the result of the inference when the weights were trained using the analog activation functions.

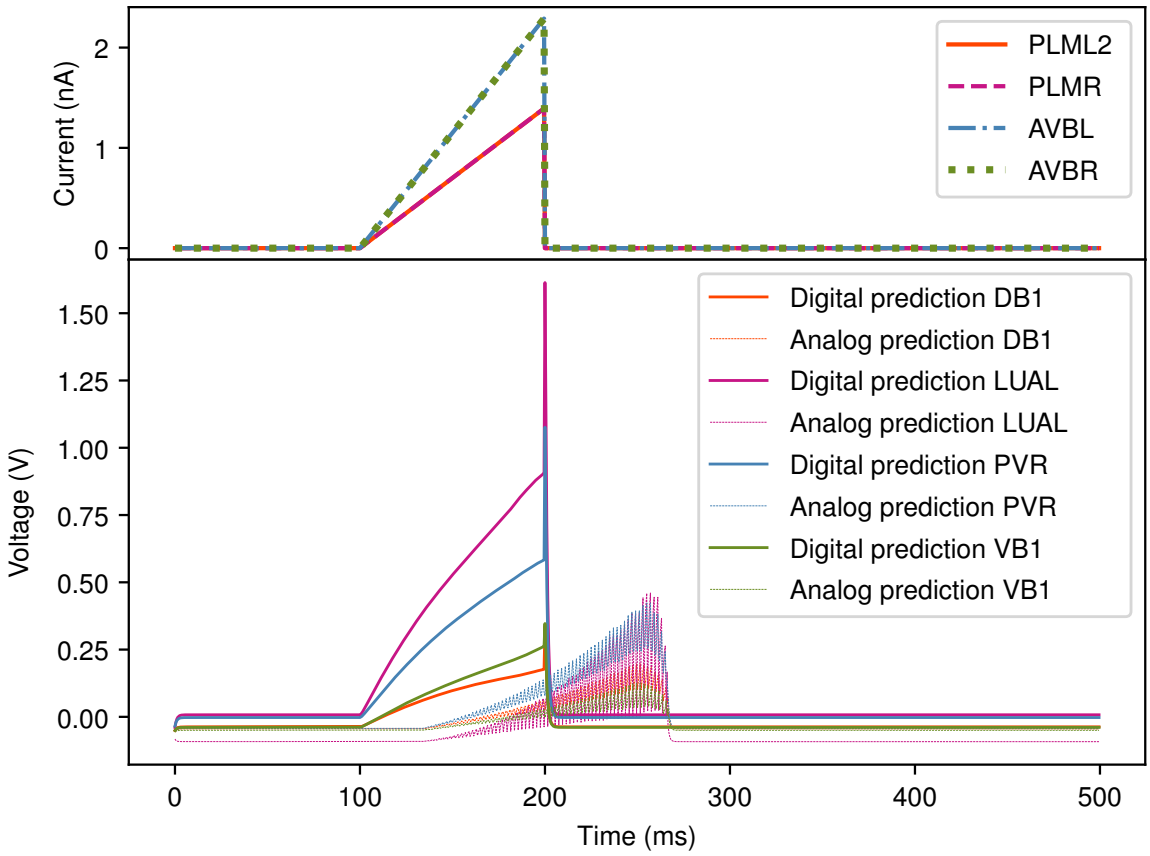
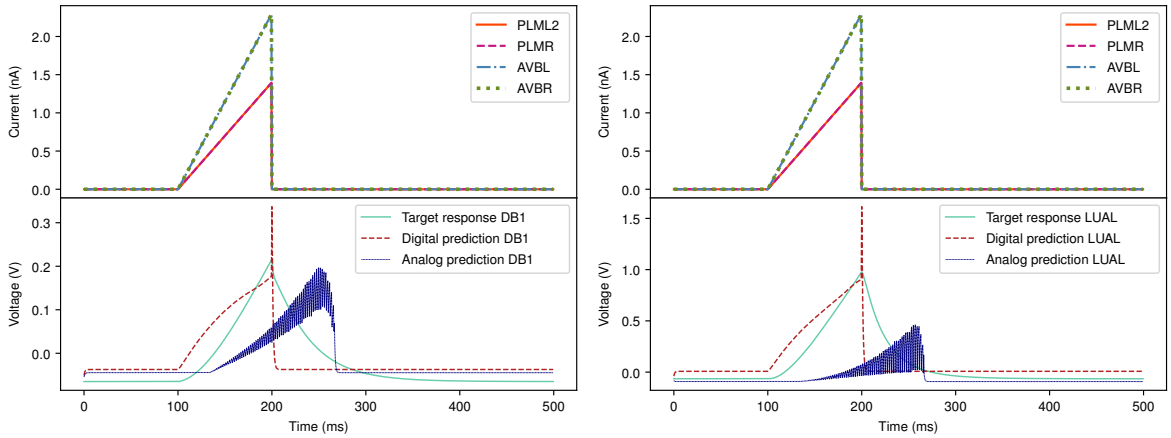


Figure 4.13: C. elegans analog responses with their digital counterparts

Figure 4.13 contains the predictions of both the digital and analog NN for all four output neurons that are being monitored. Figure 4.13 having all predicted responses to the stimuli on the same graph makes things very unintelligeable.

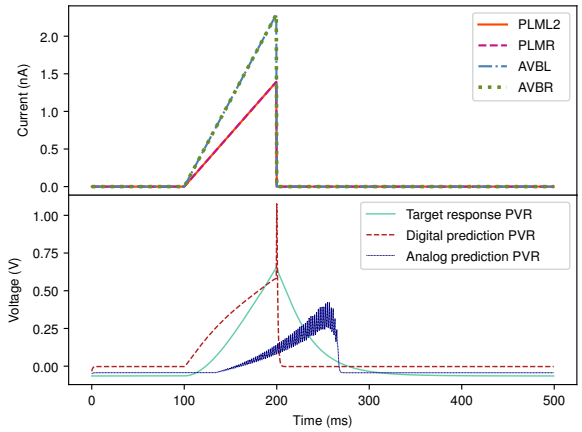
The response have thus been sperated into four graphs. One for every neuron for better readability. The target response was also added to all four graphs to highlight the objective of the NN. Those graphs can be found in figure 4.14.

All curves seem to all have a similar issue, the signal is late to go up. However, the reason for this behavior is still unknown. This can be adjusted manually in the graph until the reason for this issue is found.

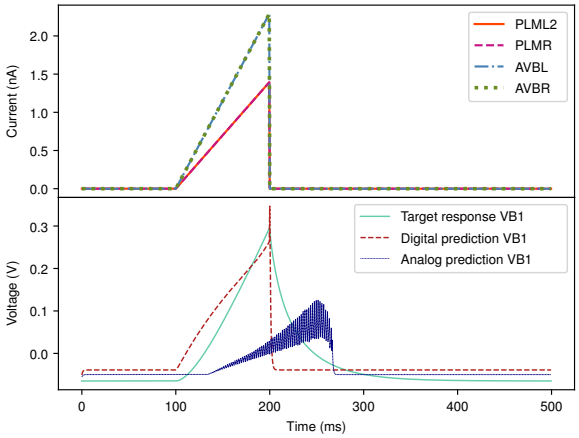


(a) Analog prediction for the DB1 neuron

(b) Analog prediction for the LUAL neuron



(c) Analog prediction for the PVR neuron



(d) Analog prediction for the VB1 neuron

Figure 4.14: C. elegans analog responses with their digital counterparts spread out for each output neuron

All the analog predicted responses seem to have the same problem. When the response is flat, the signal looks very good, while with a small vertical displacement. That displacement could be attributed to the difference of computations from the digital training to the analog inference (section 4.1). However, when the analog response sends out a pulse, the said pulse is very unstable. It continuously flickers up and down. This can be observed by zooming on the graph like in figure 4.15.

This phenomenen is very strange and has not been identified so far. The main theory is that the data is not stored properly in the memory cells, thus at every LSTM time step the supposed output is slightly altered. This would create this flickering of the outputed data.

The data is not useless for that matter. It was decided to smooth the data by averaging every output point with its neighboring points. This simple average will smooth out the curves to get a decently readable result, that can then be compared with the digitally predicted response. The python code for this function is available in section B.1.

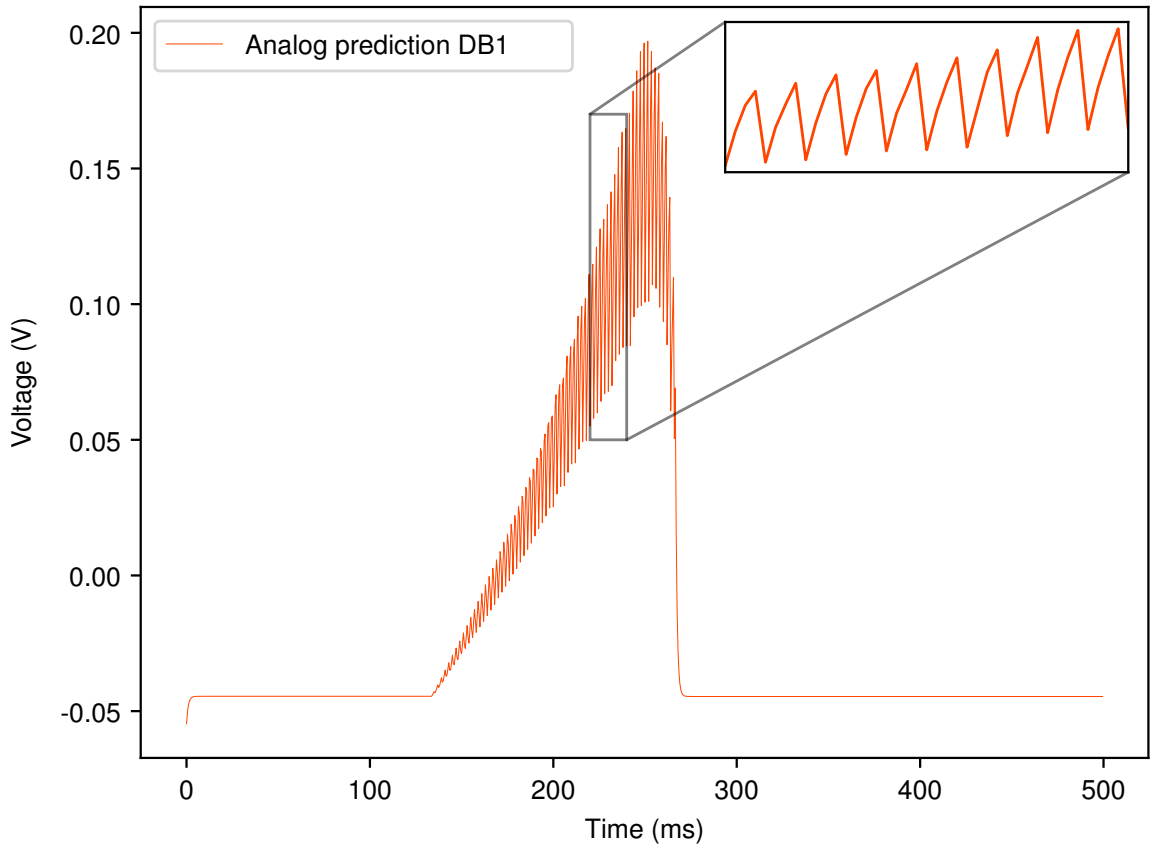


Figure 4.15: C. elegans DB1 analog response with zoom on the pulses

The adjusted (smoothed out and slided to the left to match the digital prediction best) and exploitable analog predictions are available in figure 4.17. Theses graphs allow for a much easier and better visualisation of the results.

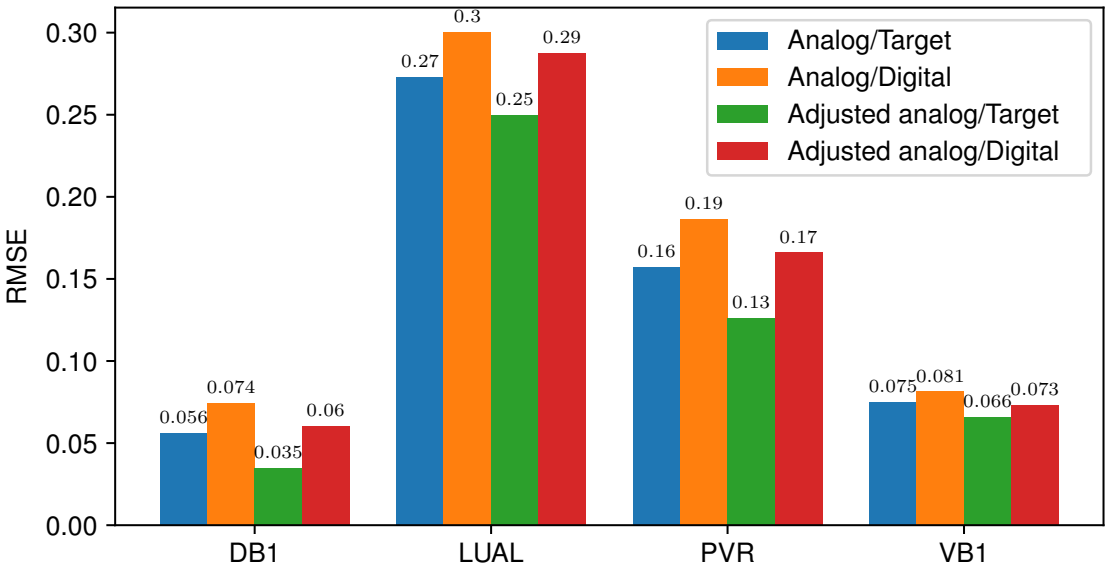
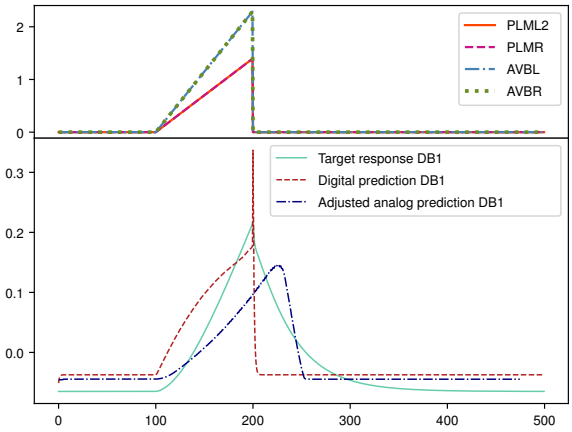
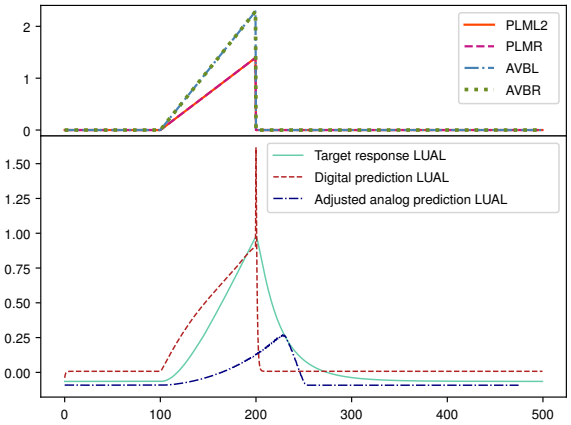


Figure 4.16: RMSEs of each neuron's analog prediction and adjusted analog prediction for sequence 5

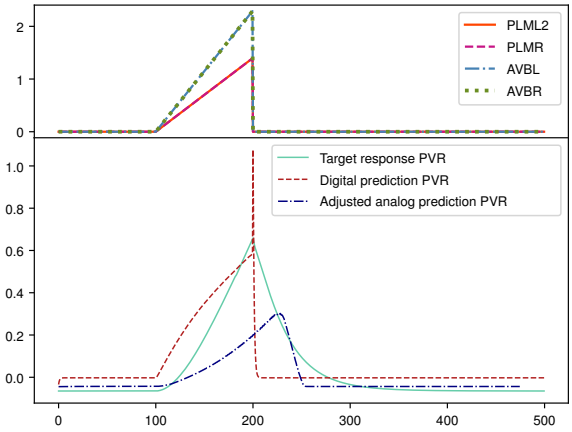
The figure 4.16 shows that adjusted or not, the errors are in the expected range. Adjusting the results slightly reduces the error, mainly due to the sliding of the results. Smoothing out the curve does not affect the error as it simply does a local average of each value. The erros are, once again higher when obtained with the analog simulation than with the digital inference.



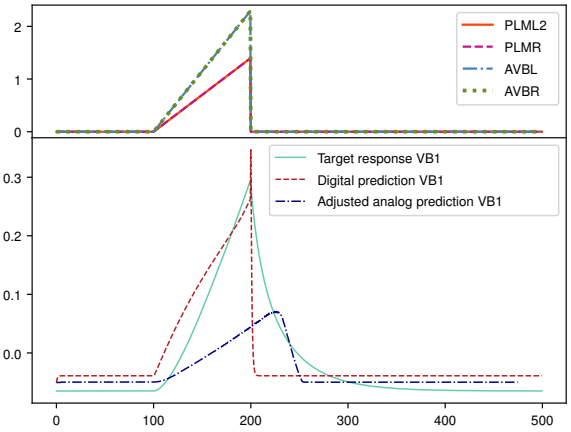
(a) Averaged analog prediction for the DB1 neuron



(b) Averaged analog prediction for the LUAL neuron



(c) Averaged analog prediction for the PVR neuron



(d) Averaged analog prediction for the VB1 neuron

Figure 4.17: C. elegans smoothed out and slided analog responses averaged with 21 neighbors, spread out for each output neuron

4.2.4 Analog results of sequence 15

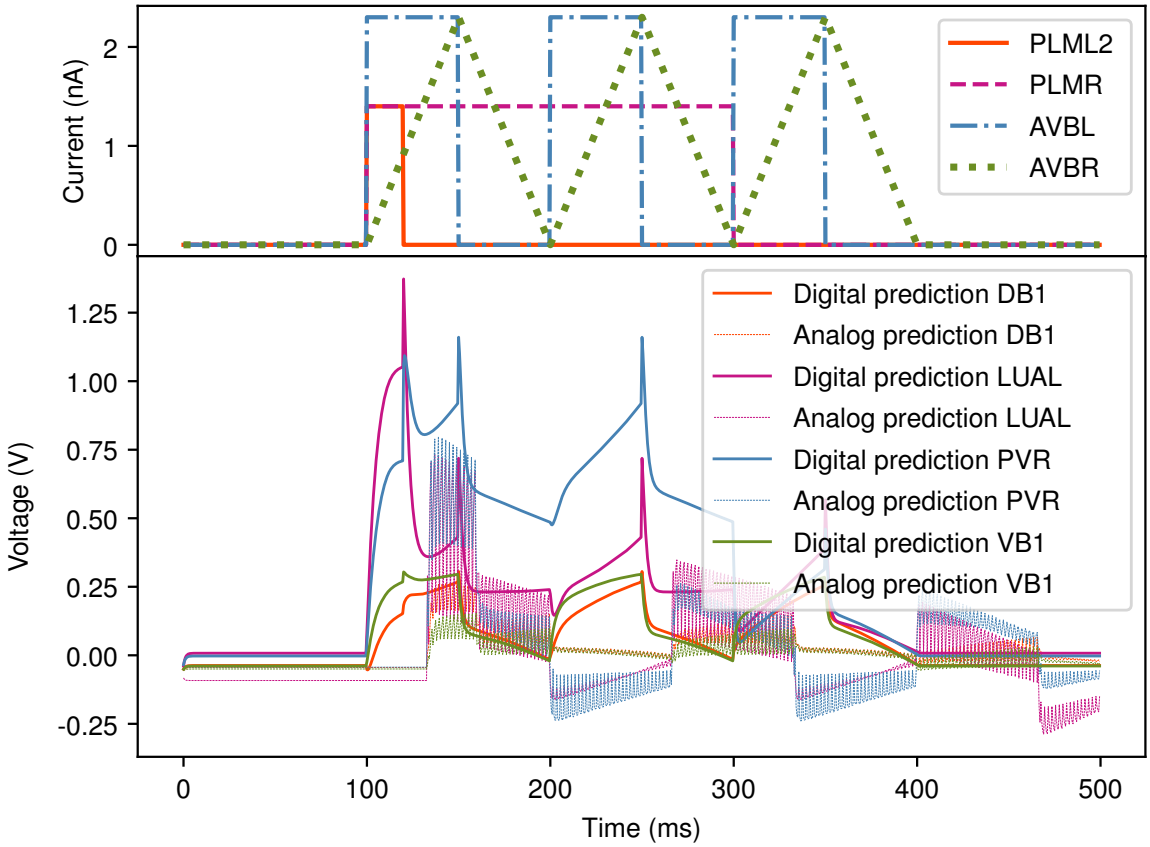
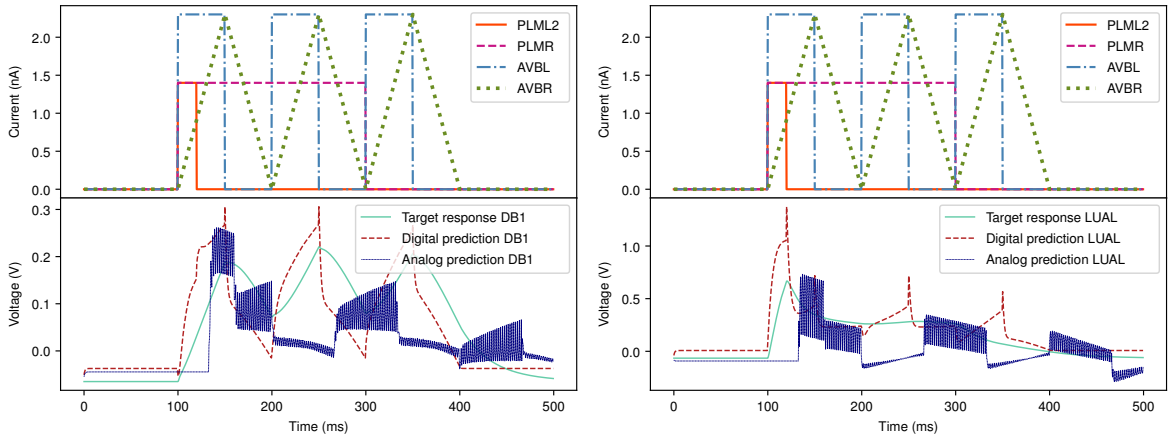


Figure 4.18: C. elegans analog responses with their digital counterparts

Figure 4.18 contains the predictions of both the digital and analog NN for all four output neurons that are being monitored. For the reasons mentioned in the previous section, for the sake of readability, the predictions are spread out on separate graphs.

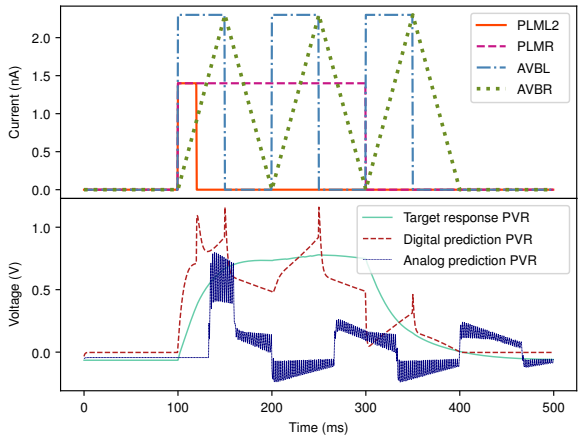
The response have thus been sperated into four graphs. One for every neuron for better readability. The target response was also added to all four graphs to highlight the objective of the NN. Those graphs can be found in figure 4.19.

All curves seem to all have a similar issue, the signal is late to go up. However, the reason for this behavior is still unknown. This can be adjusted manually in the graph until the reason for this issue is found.

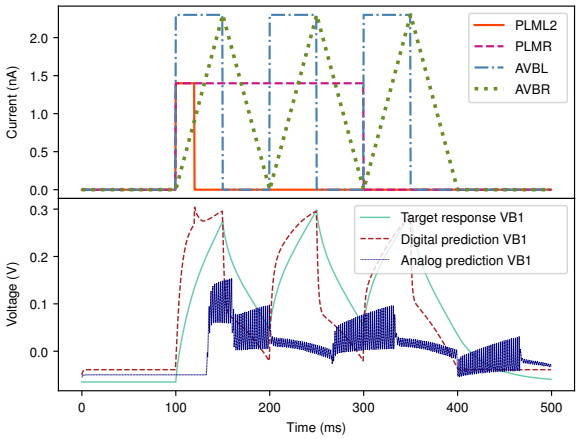


(a) Analog prediction for the DB1 neuron

(b) Analog prediction for the LUAL neuron



(c) Analog prediction for the PVR neuron



(d) Analog prediction for the VB1 neuron

Figure 4.19: C. elegans analog responses with their digital counterparts spread out for each output neuron

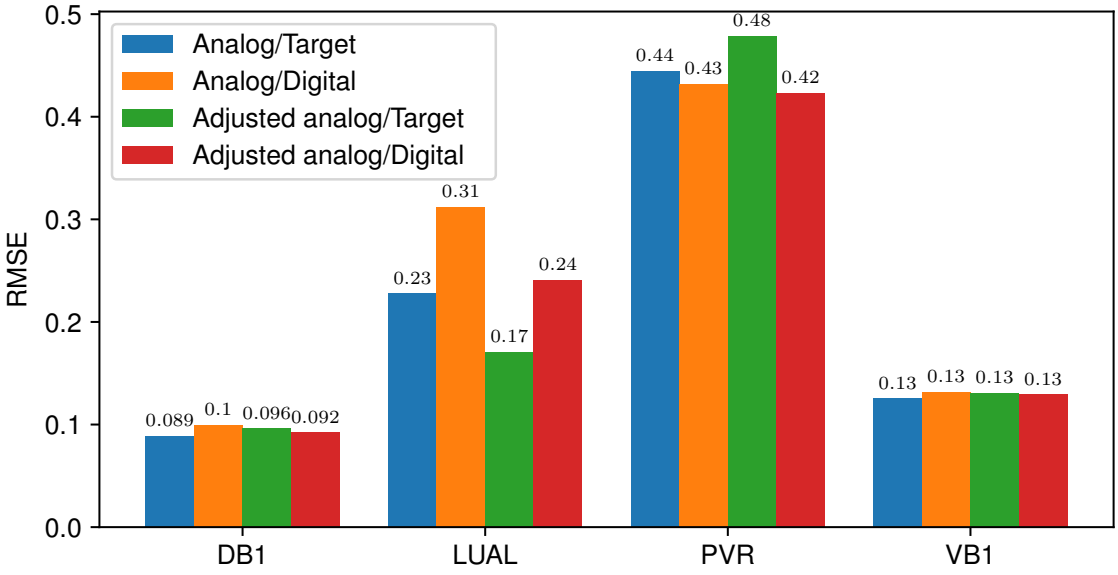
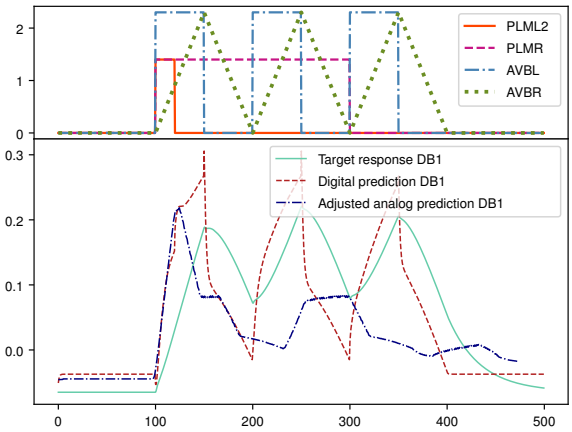
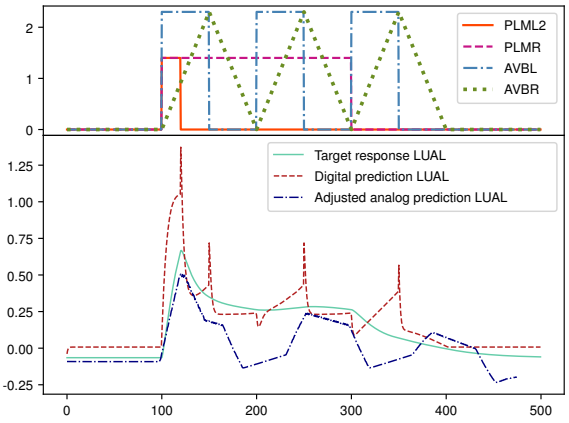


Figure 4.20: RMSEs of each neuron's analog prediction and adjusted analog prediction for sequence 15

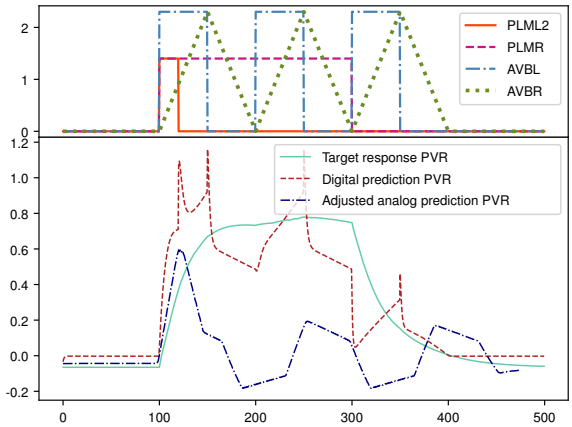
The figure 4.20 shows that adjusted or not, the errors are in the expected range. The PVR output neuron seems to be hardest to reproduce during this sequence as its error is much greater than for the others.



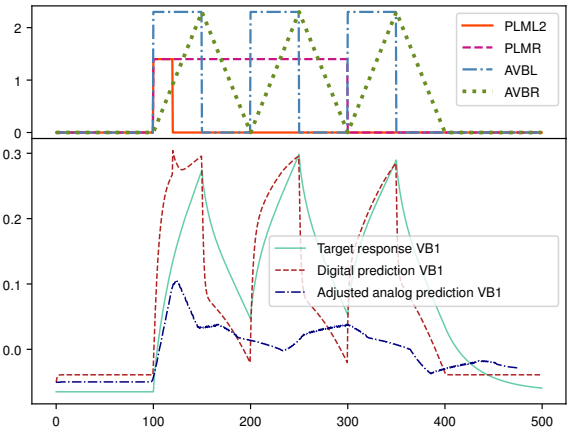
(a) Averaged analog prediction for the DB1 neuron



(b) Averaged analog prediction for the LUAL neuron



(c) Averaged analog prediction for the PVR neuron



(d) Averaged analog prediction for the VB1 neuron

Figure 4.21: C. elegans smoothed out and slided analog responses averaged with 21 neighbors, spread out for each output neuron

The results are a bit worse for this second sequence. That was expected as the inputs and the expected outputs are far more complex.

5

Conclusions

Contents

5.1 Conclusions

5.1.1 Performances

5.1.1.A LSTM

The circuit's performance has been evaluated in the previous chapter. The results highly depend on the parameters at play. For example the complexity of the dataset used, out of the two used in the thesis, the airline dataset is the simple dataset while the C. elegans dataset is the more complicated datset.

The best performance is the with the airline dataset, using the circuit with a serial size of one ($n_s = 1$) (table 4.4). While the results are not quite right, it is assumed that this issue will be elevated once inSitu training is implemented (section 5.2.3).

More complex datsets like C. elegans, the issue is getting more present. Indeed, the dataset feeds to the circuit four inputs across a thousand time steps and outputs just as much data. Any inaccuracy is scaled up.

With a simple input sequence, such as sequence 5 (figure 3.20a), the resulting predictions are quite good and on par with what is expected. The predictions are a bit late and it is not clear what is causing this specific issue. A theory is that the memory cells deteriorate its stored value by a very small amount every time step, thus the stored value is affected a lot after a large amount of time steps.

More complex input sequences produce even more inaccuracies. Indeed, when working with sequence 15 (figure 3.20d), the inaccuracies explode and gives out a barely usable prediction. The curve generated does not fit its digital counterpart anymore.

The results obtained demonstrate that the LSTM circuit block can be run with low error when dealing with simple inputs. The circuit is ready for a full simulation, meaning also simulating training with the analog circuit.

5.1.1.B GRU

The GRU circuit, while being present, is still a work in progress. The outputed predictions are scaled down for a still unknown reason, but show the right output shape. Once those issues are dealt with and have been fixed, the GRU circuit will be ready for an analog training as well.

5.1.2 Execution time

The inference time is quite tricky to deal with. While the analog circuit's inference time is set, it gets more complicated when trying to get the digital inference time. The digital inference time depends on the computer it is being run on. The CPUs used in embedded sytems, the main application for this work, are usually not very fast as they are intended to be low powered. This parts will thus only dicuss of the inference of the analog circuit.

5.1.2.A Airline inference

The airline problem feeding two inputs to the circuit to get a single output, it will have to be run 142 times to get all the output data. The inference time depends on the serial size being used :

- $n_s = 1$: The output can be read as a voltage on the output net from $t_0 = 18\mu s$ to $t_1 = 22\mu s$.
- $n_s = 2$: The output can be read as a voltage on the output net from $t_0 = 34\mu s$ to $t_1 = 38\mu s$.
- $n_s = 4$: The output can be read as a voltage on the output net from $t_0 = 66\mu s$ to $t_1 = 70\mu s$.

Those are the times it takes to compute one of the outputs. It takes 144 times longer to get all of the results.

5.1.2.B C. elegans inference

This sequence is a bit more complicated, for every input sequence with a thousand time steps, there are a thousand output vectors being read. Here, the inference time also depends on the serial size.

Serial size	$n_s = 1$	$n_s = 2$	$n_s = 4$	$n_s = 8$
Start of first output	$9\mu s$	$17\mu s$	$33\mu s$	$65\mu s$
End of first output	$13\mu s$	$21\mu s$	$37\mu s$	$69\mu s$
Output period	$9\mu s$	$17\mu s$	$33\mu s$	$65\mu s$
Start of last output	$9ms$	$17ms$	$33ms$	$65ms$
End of last output	$9ms + 4\mu s$	$17ms + 4\mu s$	$33ms + 4\mu s$	$65ms + 4\mu s$

Table 5.1: First and last read times for the different serial sizes

Table 5.1 shows the first and last output time range with the period. All of the output times can be determined using the first time range and using the period. A sequence consists of $500ms$ as explained in section 3.11.2, the time steps are physically $500\mu s$ appart, in the case where the circuit would be used to do real time inference, the circuit would have to be slowed down to be on sync with the inputs. The circuit could could be used in a real time use for this time sensitive problem.

5.1.3 onChip area

The LSTM and GRU blocks' onChip area are impossible to estimate, as parts of the circuits are made of verilog-A models. However, to get a general idea of the area of the chip, it can be assumed that the area of the circuit mainly depends on the the area of a memrisor, because the area of a memristor is much greater than the ones of the other components. Since the number of memristors is the number of weights in the circuit, the minimum area of any NN can be determined. The area for a memristor was determined using the feature size of the memristors that can be fabricated at Instituto de Engenharia de Sistemas e Computadores (INESC). The typical memristor that can be fabricated at INESC is $3\mu m$, which would make the approximate area for a memristor $A_{memristor} = 9\mu m^2 = 9 \cdot 10^{-12} m^2$

Table 5.2 contains several NNs models with their number of parameters, a minimum onChip area and an estimated feature size. None of those values have any scientific ground, they are just here to give general idea of the kind of circuit thta could be fabricated. Those areas only considers the area of the memristors, using a two memristor per synapse architecture like the one used in the thesis.

The feature size represents the length of the side of the chip if the chip was manufactured in a square.

The values of LLMs, such as Large Language Model Meta AI (LLaMA)-2 **Meta**'s LLM and the two last versions of Generative Pre-trained Transformers (GPTs) the LLM that powers *ChatGPT*, were included to show how the area of the chip scales up. Those models do not use RNNs, and futhermore, give out ridiculous areas. Fabricating such cicuits in analog is not feasible witht the current technologies.

Model	Parameters	Minimum area	Approximate feature size
Airline GRU $n_h = 4$	77	$1386\mu m^2$	$37.2\mu m$
Airline LSTM $n_h = 4$	101	$1818\mu m^2$	$42.6\mu m$
C. elegans GRU $n_h = 8$	348	$6264\mu m^2$	$79.1\mu m$
C. elegans LSTM $n_h = 8$	452	$8136\mu m^2$	$90.2\mu m$
LLaMA-2	$7 \cdot 10^9$	$1260cm^2$	$35.5cm$
	$13 \cdot 10^9$	$2340cm^2$	$48.4cm$
	$70 \cdot 10^9$	$1.26m^2$	$1.12m$
GPT-3	$175 \cdot 10^9$	$3.14m^2$	$1.77m$
GPT-4	$1 \cdot 10^{12}$	$18.0m^2$	$4.24m$

Table 5.2: Estimated onChip area for different models of NNs

Nevertheless, it is interresting to look at those approximation. It shows that the simple models used in this work would take a very small onChip area, even though this a very raw number. It also shows the current limit with state of the art NNs at the current time.

The onChip area of small NNs such as the ones used in this work are very reasonable and could easily be fabricated for embedded use.

5.2 Future work

5.2.1 Changing the verilog-A files for real circuit implementation

Making the system actually one hundred percent implemented for inference is the obvious next step in making a fully analog LSTM computer.

This means designing both an opAmp for the current flowing through the circuit and a voltage multiplier working in the right voltage range.

5.2.2 Circuit controller

The final circuit will need an external controller to manage diffenrent things in the circuit. This was not created in this thesis but will have to be in order to have a functionning circuit once fabricated.

The controller will increase the power consumption of the overall system. However, due to the slow (precision of about $1\mu s$) nature of the system requirements, the controller will be very low power,

and will not affect the overall power consumption of the final system.

The controller will have several roles in the system :

- The first role will be to manage the different flags. This means that the controller will have to set those to high or low in order to control the system. This includes flags like the enable flags of the voltage driven crossbar array (section 3.6.3) and those for the memory cell control.
- Anoter use for this controller is to reset the memory cell to V_{cm} (0 as real value equivalent, table 3.1). This will be done by enabling the input and then applying V_{cm} to the input, this will set the capacitor to V_{cm}. This has to be done everytime the circuit is ran.
- The controller will also be used to control the memristors internal resistance and control the weights. Depending on the type of memristor, the method and time to change the internal resistance differs. That will only be use for inSitu trainning(section 5.2.3).

5.2.3 inSitu trainning

Of course the first thing that comes to mind to improve the system and making it actually usable for a real world application would be doing inSitu training, meaning that the weight training would also be happening on the chip. Training on the chip also means that the weights, hence the memristors in the circuit, need to be changed between each epoch. That requires to have a backpropagation algorithm. Here, two options are available :

- The easiest option would be to use the external controller (section 5.2.2) to make the backprogation computation needed. This is not the best option, as the backpropagation would be happening in digital. This digital computation would have to happen as fast as the analog computation and thus a much faster controller would be required, meaning a bigger power consumption.
- The ideal way of implementing the backpropagation would be using an analog system similar to the one that is used for the inference. This would still mean having a controller to change the internal resistance of the memristors, this version would require a slower clock and making the controller lower powered. Using an analog implementation might also harm the system, as the backpropagation algorithm will probably be simpler and reduce the efficiency of the system.

In order for the inSitu training to work, the weights also need to change. Future work also also include looking for ways to change the weights in the fastest possible way. Idealy the algorithm will not need to compute the weight to resistance conversion (section 3.10) and directly know how the change the resistances' values.

This is a potential idea on how to improve the time it takes to compute a single epoch. Indeed changing the resistance value in a memristor, takes time depending on the type of memristor that is being used. If the weight to resistance is computed the same way as in the thesis (section 3.10), so that each pair of resistance is centered around the middle point of the memristor's resistance range.

Instead of having to do that, the idea is to only change one of the memristor's internal resistance so that the final weight is still the same. Equation (3.23) needs to be respected in any situation. The memristor's resistances will not be centered (equation (3.24)) anymore.

5.2.4 Other LSTMs variants

As discused before(section 3.8), they are only two of the LSTM variants available in Keras. Nonetheless, it is possible to write custom layer function for Keras. Creating a layer from scratch is very time consuming and not so interresting for the scope the results needed. It can be done to extend the reach of the work.

5.2.5 Avoiding doubling memory cells

Here there are two different problems that have two similar solutions, using a capacitor to keep the voltage in the node for a longer time :

- For the hidden state feedback memory cells, a potential solution would be adding a single capacitor before the input pin of the memory cell and then enabling the input during the pause between time steps (e_{next}). This would allow the enable in and enable out flags to be high at different times. This solution only works in fully parallel mode.
- For the cell state memory cells, adding a capacitor and a buffer (section A.1) after the output of the memory cell group. This capacitor is here to store the voltage after the output enable flag goes to low. The buffer is here to output the voltage with a high impedance.

Those solutions are purely theorical, they have not been tested due to lack of time.

5.2.6 Removing weight constraints

Removing the weights constraints is tricky since the the opAmp is based on a verilog-A model (section 3.5.1). Indeed, the model needs to be changed in order not to output voltages outside of the system's voltage range ($[0, v_{dd}]$). Once that is done, the opAmp should behave more like a real one. The opAmps should now be able to output a saturated signal.

However those saturated levels need to be added to the training of the weights. This can be done in Keras by creating a custom layer in the circuit that keeps the values within a the defined range ($[-9, 9]$ for this thesis). The training will thus happen with the values being saturated as it would happen in the circuit.

This solution is not implemented in the circuit as the saturating signals might affect the real circuit. This is would allow the circuit to access a larger range of weights and make the training more performant.

5.2.7 Serializing the GRU circuit

The GRU circuit, once fully working and implemented in its current form, can be serialized.

The GRU circuit cannot be serialized because of the candidate hidden state needs a fully computed reset gate to be computed. The solution would be to use the reset gate crossbar array with a serial size of one. While all other crossbar arrays can be serialized. This will allow to reduce the amount of pointwise components.

Bibliography

[1] R. Hasan, T. M. Taha, and C. Yakopcic, "On-chip training of memristor crossbar based multi-layer neural networks," *Microelectronics Journal*, vol. 66, pp. 31–40, Aug. 2017. [Online]. Available: https://doi.org/10.1016/j.mejo.2017.05.005

[2] Jul 2023. [Online]. Available: https://en.wikipedia.org/wiki/Long{_}short-term{_}memory

[3] Oct 2023. [Online]. Available: https://en.wikipedia.org/wiki/Memristor

[4] P. Phillips, S. Sarkar, I. Robledo, P. Grother, and K. Bowyer, "The gait identification challenge problem: data sets and baseline algorithm," in *Object recognition supported by user interaction for service robots*. IEEE Comput. Soc. [Online]. Available: https://doi.org/10.1109/icpr.2002.1044731

[5] A. Kale, A. Sundaresan, A. Rajagopalan, N. Cuntoor, A. Roy-Chowdhury, V. Kruger, and R. Chellappa, "Identification of humans using gait," *IEEE Transactions on Image Processing*, vol. 13, no. 9, pp. 1163–1173, Sep. 2004. [Online]. Available: https://doi.org/10.1109/tip.2004.832865

[6] C. Li, Z. Wang, M. Rao, D. Belkin, W. Song, H. Jiang, P. Yan, Y. Li, P. Lin, M. Hu, N. Ge, J. P. Strachan, M. Barnell, Q. Wu, R. S. Williams, J. J. Yang, and Q. Xia, "Long short-term memory networks in memristor crossbar arrays," *Nature Machine Intelligence*, vol. 1, no. 1, pp. 49–57, Jan. 2019. [Online]. Available: https://doi.org/10.1038/s42256-018-0001-4

[7] Z. Shi, F. Shi, W.-S. Lai, C.-K. Liang, and Y. Liang, "Deep online fused video stabilization," 2021. [Online]. Available: https://arxiv.org/abs/2102.01279

[8] D. Wang, G. Liu, B. D. Prasad, T. Xiao, and Y. Yang, "FFSCore-LSTM: An enhanced LSTM-based camera relocalization networks via front feature smoothing core," *Measurement*, vol. 210, p. 112542, Mar. 2023. [Online]. Available: https://doi.org/10.1016/j.measurement.2023.112542

[9] R. Barbulescu, G. Mestre, A. L. Oliveira, and L. M. Silveira, "Learning the dynamics of realistic models of c. elegans nervous system with recurrent neural networks," *Scientific Reports*, vol. 13, no. 1, Jan. 2023. [Online]. Available: https://doi.org/10.1038/s41598-022-25421-w

[10] G. Van Rossum and F. L. Drake, *Python 3 Reference Manual*. Scotts Valley, CA: CreateSpace, 2009.

[11] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard *et al.*, "Tensorflow: A system for large-scale machine learning," in *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, 2016, pp. 265–283.

[12] W. R. MoskolaÃ¯, W. Abdou, A. Dipanda, and Kolyang, "Application of deep learning architectures for satellite image time series prediction: A review," *Remote Sensing*, vol. 13, no. 23, p. 4822, Nov. 2021. [Online]. Available: https://doi.org/10.3390/rs13234822

[13] K. Cho, B. van Merrienboer, C. Gulcehre, D. Bahdanau, F. Bougares, H. Schwenk, and Y. Bengio, "Learning phrase representations using rnn encoder-decoder for statistical machine translation," 2014. [Online]. Available: https://arxiv.org/abs/1406.1078

[14] "Gradient flow in recurrent nets: The difficulty of learning LongTerm dependencies," in *A Field Guide to Dynamical Recurrent Networks*. IEEE, 2009. [Online]. Available: https://doi.org/10.1109/9780470544037.ch14

[15] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural Computation*, vol. 9, no. 8, pp. 1735–1780, Nov. 1997. [Online]. Available: https://doi.org/10.1162/neco.1997.9.8.1735

[16] F. Gers, "Learning to forget: continual prediction with LSTM," in *9th International Conference on Artificial Neural Networks: ICANN '99*. IEE, 1999. [Online]. Available: https://doi.org/10.1049/cp:19991218

[17] [Online]. Available: https://keras.io/api/layers/recurrent{_}layers/lstm/

[18] [Online]. Available: https://pytorch.org/docs/stable/generated/torch.nn.LSTM.html

[19] K. Greff, R. K. Srivastava, J. Koutnik, B. R. Steunebrink, and J. Schmidhuber, "LSTM: A search space odyssey," *IEEE Transactions on Neural Networks and Learning Systems*, vol. 28, no. 10, pp. 2222–2232, Oct. 2017. [Online]. Available: https://doi.org/10.1109/tnnls.2016.2582924

[20] A. Graves and J. Schmidhuber, "Framewise phoneme classification with bidirectional LSTM and other neural network architectures," *Neural Networks*, vol. 18, no. 5-6, pp. 602–610, Jul. 2005. [Online]. Available: https://doi.org/10.1016/j.neunet.2005.06.042

[21] N. Gruber and A. Jockisch, "Are GRU cells more specific and LSTM cells more sensitive in motive classification of text?" *Frontiers in Artificial Intelligence*, vol. 3, Jun. 2020. [Online]. Available: https://doi.org/10.3389/frai.2020.00040

[22] [Online]. Available: https://pytorch.org/docs/stable/generated/torch.nn.GRU

[23] [Online]. Available: https://keras.io/api/layers/recurrent{_}layers/gru/

[24] L. Chua, "Memristor-the missing circuit element," *IEEE Transactions on Circuit Theory*, vol. 18, no. 5, pp. 507–519, 1971. [Online]. Available: https://doi.org/10.1109/tct.1971.1083337

[25] D. B. Strukov, G. S. Snider, D. R. Stewart, and R. S. Williams, “The missing memristor found,” *Nature*, vol. 453, no. 7191, pp. 80–83, May 2008. [Online]. Available: https://doi.org/10.1038/nature06932

[26] L. Chua and S. M. Kang, “Memristive devices and systems,” *Proceedings of the IEEE*, vol. 64, no. 2, pp. 209–223, 1976.

[27] A. V. Fadeev and K. V. Rudenko, “To the issue of the memristor’s HRS and LRS states degradation and data retention time,” *Russian Microelectronics*, vol. 50, no. 5, pp. 311–325, Sep. 2021. [Online]. Available: https://doi.org/10.1134/s1063739721050024

[28] D. Joksas and A. Mehonic, “badcrossbar: A python tool for computing and plotting currents and voltages in passive crossbar arrays,” *SoftwareX*, vol. 12, p. 100617, Jul. 2020. [Online]. Available: https://doi.org/10.1016/j.softx.2020.100617

[29] S. Kvatinsky, E. G. Friedman, A. Kolodny, and U. C. Weiser, “TEAM: ThrEshold adaptive memristor model,” *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 60, no. 1, pp. 211–221, Jan. 2013. [Online]. Available: https://doi.org/10.1109/tcsi.2012.2215714

[30] S. Kvatinsky, M. Ramadan, E. G. Friedman, and A. Kolodny, “VTEAM: A general model for voltage-controlled memristors,” *IEEE Transactions on Circuits and Systems II: Express Briefs*, vol. 62, no. 8, pp. 786–790, Aug. 2015. [Online]. Available: https://doi.org/10.1109/tcsii.2015.2433536

[31] H. Abdalla and M. D. Pickett, “SPICE modeling of memristors,” in *2011 IEEE International Symposium of Circuits and Systems (ISCAS)*. IEEE, May 2011. [Online]. Available: https://doi.org/10.1109/iscas.2011.5937942

[32] S. Maheshwari, S. Stathopoulos, J. Wang, A. Serb, Y. Pan, A. Mifsud, L. B. Leene, J. Shen, C. Papavassiliou, T. G. Constandinou, and T. Prodromakis, “Design flow for hybrid CMOS/memristor systems—part i: Modeling and verification steps,” *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 68, no. 12, pp. 4862–4875, Dec. 2021. [Online]. Available: https://doi.org/10.1109/tcsi.2021.3122343

[33] I. Messaris, S. Nikolaidis, A. Serb, S. Stathopoulos, I. Gupta, A. Khiat, and T. Prodromakis, “A TiO2 ReRAM parameter extraction method,” in *2017 IEEE International Symposium on Circuits and Systems (ISCAS)*. IEEE, May 2017. [Online]. Available: https://doi.org/10.1109/iscas.2017.8050789

[34] B. Ding, H. Qian, and J. Zhou, “Activation functions and their characteristics in deep neural networks,” in *2018 Chinese Control And Decision Conference (CCDC)*. IEEE, Jun. 2018. [Online]. Available: https://doi.org/10.1109/ccdc.2018.8407425

[35] R. Maksutov, “Deep study of a not very deep neural network. part 2: Activation functions,” Jul 2018. [Online]. Available: https://towardsdatascience.com/deep-study-of-a-not-very-deep-neural-network-part-2-activation-functions-fd9bd8d406fc

[36] K. Adam, K. Smagulova, and A. James, “Generalised analog LSTMs recurrent modules for neural computing,” *Frontiers in Computational Neuroscience*, vol. 15, Sep. 2021. [Online]. Available: https://doi.org/10.3389/fncom.2021.705050

[37] Feb 2011. [Online]. Available: https://www.analog.com/en/products/ad734.html

[38] D. P. Kingma and J. Ba, “Adam: A method for stochastic optimization,” 2014. [Online]. Available: https://arxiv.org/abs/1412.6980

[39] V. Barbaza, “LSTM weights generator.” [Online]. Available: https://github.com/bicheTortue/LSTM-weights-generator

[40] ——, “Spice LSTM netlist generator code.” [Online]. Available: https://github.com/bicheTortue/memristor-LSTM-generator

[41] J. Brownlee, “Time series prediction with lstm recurrent neural networks in python with keras,” Aug 2022. [Online]. Available: https://machinelearningmastery.com/time-series-prediction-lstm-recurrent-neural-networks-python-keras/

[42] ——, “Machine learning datasets.” [Online]. Available: https://github.com/jbrownlee/Datasets

[43] [Online]. Available: https://knowm.org/memristors/

[44] [Online]. Available: https://www.tensorflow.org/api{_}docs/python/tf/keras/activations/hard{_}sigmoid

[45] [Online]. Available: https://pytorch.org/docs/stable/generated/torch.nn.Hardtanh.html

Circuits

A.1 Buffer

This is a classic circuit that only takes one opAmp wired as figure A.1. It is used to forward a voltage without interfering the node of the input voltage.

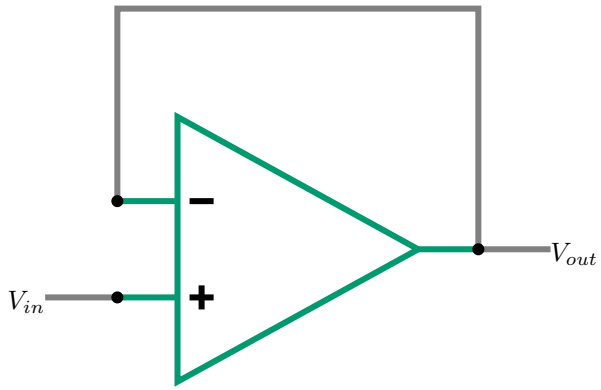


Figure A.1: Buffer circuit

The input is V_{in} and the output is V_{out}.

A.2 Linear combination of inputs

The linear combination of inputs is a very general circuit that is used in a plethera of different situations. A modulable circuit with a variable number of input $(n+1)$ in available in figure A.2.

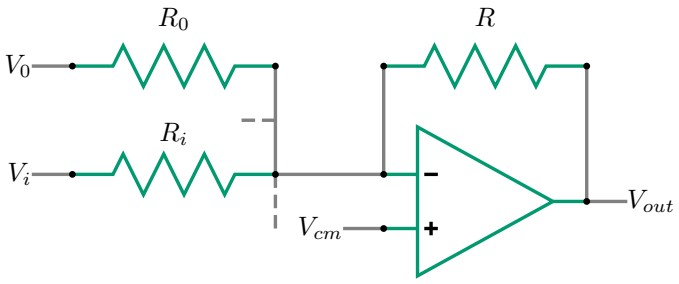


Figure A.2: Summing inverter circuit

This circuit links the inputs and its output with equation (A.1).

$$V_{out} = V_{cm} - R \cdot \sum_{i=0}^{n-1} \frac{V_i - V_{cm}}{R_i} \tag{A.1}$$

In this work itself it is used in at least three different use cases :

- As a voltage mirror around V_{cm}, using a single input $(n = 1)$ and using $R_0 = R$. This is used in section 3.1.
- As a inverting amplifier after a voltage multiplier (section 3.5.2). Using one input $(n = 1)$ and $R > R_0$. It can be found in sections 3.6 and 3.7.

- It is also used as a summing inverter with amplification after a voltage multiplier. Using two inputs ($n = 2$) and $R_0 = R_1 < R$ for the resistances. This is also found in the circuit in sections 3.6 and 3.7.

B

Codes and algorithms

B.1 Smoothing the curve

The code for smoothing the curve takes in a list and the averaging range. The function returns the averaged list.

```
def smooth_line(df, n):
    size = len(df)
    out = df.copy()
    for i in range(size):
        if i < n:
            out[i] = mean(df[0 : i + n])
        elif size - i - 1 < n:
            out[i] = mean(df[i - n : size - 1])
        else:
            out[i] = mean(df[i - n : i + n])
    return out
```

Listing B.1: Python function to smooth out curve

B.2 Saving weights to a file

There are a few functions that show how the weights are saved into a file. First depending on the type of layer, the weights need to be set in a list in the correct order. Once those list are correctly set everything is saved in a binary file.

```
import pickle

import tensorflow as tf
from tensorflow.keras.layers import Layer
from tensorflow.keras.layers import Dense, LSTM, GRU
from keras.layers import TimeDistributed


def getLSTMWeights(lstm, nbHidden):
    W, U, b = lstm.get_weights()
    nbGates = 4
    out = [None] * nbGates

    for i in range(nbGates):
        out[i] = []
        for j in range(nbHidden):
            out[i].extend(W[:, i * nbGates + j])
            out[i].extend(U[:, i * nbGates + j])
            out[i].append(b[i * nbGates + j])
    return out


def getGRUWeights(layer, nbHidden):
    W, U, b = layer.get_weights()
    nbGates = 3
    out = [None] * nbGates

    for i in range(nbGates):
        out[i] = []
        for j in range(nbHidden):
            out[i].extend(W[:, i * nbGates + j])
            out[i].extend(U[:, i * nbGates + j])
            out[i].append(b[i * nbGates + j])
    return out


def getDenseWeights(nn, nbOutput):
    W = nn.get_weights()[0]
    b = list(nn.get_weights()[1])
    out = []
    for i in range(nbOutput):
        out.extend(W[:, i])
```

```
        out.append(b[i])
    return out


def saveTofile(layers, filename):
    out = [[]]
    for layer in layers:
        if type(layer) == TimeDistributed:
            nbOut = layer.output_shape[-1]
            out[0].append("tDense(" + str(nbOut) + ")")
            out.append(getDenseWeights(layer, nbOut))
        elif type(layer) == Dense:
            nbOut = layer.output_shape[1]
            out[0].append("Dense(" + str(nbOut) + ")")
            out.append(getDenseWeights(layer, nbOut))
        elif type(layer) == LSTM:
            nbHid = layer.units
            out[0].append("LSTM(" + str(nbHid) + ")")
            out.append(getLSTMWeights(layer, nbHid))
        elif type(layer) == GRU:
            nbHid = layer.units
            out[0].append("GRU(" + str(nbHid) + ")")
            out.append(getGRUWeights(layer, nbHid))

    with open(filename, "wb") as file:  # Pickling
        pickle.dump(out, file)
```

Listing B.2: Python functions used to save Keras' weights to a file

B.3 Weights to resistances

The function takes in a weight and returns a tuple with the two resistances values. The global variables are set to set values chosen in this work.

```
Rmin = 10**4
Rmax = 10**6
Rf = (Rmax + Rmin) / 2


def wei2res(w):
    Rpos = (w + 1 - np.sqrt(w**2 + 1)) * Rf / w
    Rpos = float("%.2g" % Rpos)
    Rneg = 2 * Rf - Rpos
    return (Rpos, Rneg)
```

Listing B.3: Python function to convert a weight in two resistances

C

Mathematical tools

C.1 Solving weight to resistance

$$\begin{cases} w = \frac{R_f}{R_+} - \frac{R_f}{2 \cdot R_f - R_+} \\ R_- = 2 \cdot R_f - R_+ \end{cases} \tag{C.1}$$

$$\begin{cases} w = \frac{2 \cdot R_f^2 - 2 \cdot R_+ \cdot R_f}{2 \cdot R_f \cdot R_+ - R_+^2} \\ R_- = 2 \cdot R_f - R_+ \end{cases} \tag{C.2}$$

$$\begin{cases} 2 \cdot R_f \cdot R_+ \cdot w - R_+^2 \cdot w = 2 \cdot R_f^2 - 2 \cdot R_+ \cdot R_f \\ R_- = 2 \cdot R_f - R_+ \end{cases} \tag{C.3}$$

$$\begin{cases} R_+^2 \cdot w - R_+ \cdot 2 \cdot R_f \cdot (w+1) + 2R_f^2 = 0 \\ R_- = 2 \cdot R_f - R_+ \end{cases} \tag{C.4}$$

$$R_+^2 \cdot w - R_+ \cdot 2 \cdot R_f \cdot (w+1) + 2R_f^2 = 0 \tag{C.5}$$

We must now solve equation (C.6).

$$\Delta = 4 \cdot R_f^2 \cdot (w+1)^2 - 8 \cdot w \cdot R_f^2 = 4 \cdot R_f^2 \cdot (w^2+1) \geq 0 \tag{C.6}$$

Giving the two solutions :

$$R_{+,0} = \frac{R_f \cdot (w+1) - R_f \cdot \sqrt{w^2+1}}{w} \tag{C.7}$$

$$R_{+,1} = \frac{R_f \cdot (w+1) + R_f \cdot \sqrt{w^2+1}}{w} \tag{C.8}$$

C.2 Hard sigmoid and tanh functions

The hard sigmoid and tanh functions are simplifications of the the classic sigmoid and tanh used for faster computation. They come with a downside, they are less efficient then their original functions.

They are defined as equation (C.9) for the hard sigmoid and equation (C.10) for the hard tanh [44, 45].

$$hard\sigma(x) = \begin{cases} 0, & x \leq -2.5 \\ \frac{x}{5} + \frac{1}{2}, & 2.5 > x > -2.5 \\ 1, & x \geq 2.5 \end{cases} \tag{C.9}$$

$$hardtanh(x) = \begin{cases} -1, & x \leq -1 \\ x, & 1 > x > -1 \\ 1, & x \geq 1 \end{cases} \tag{C.10}$$

Some versions of the hard sigmoid are defined differently from equation (C.9), the slope depends on the library used [**?**].

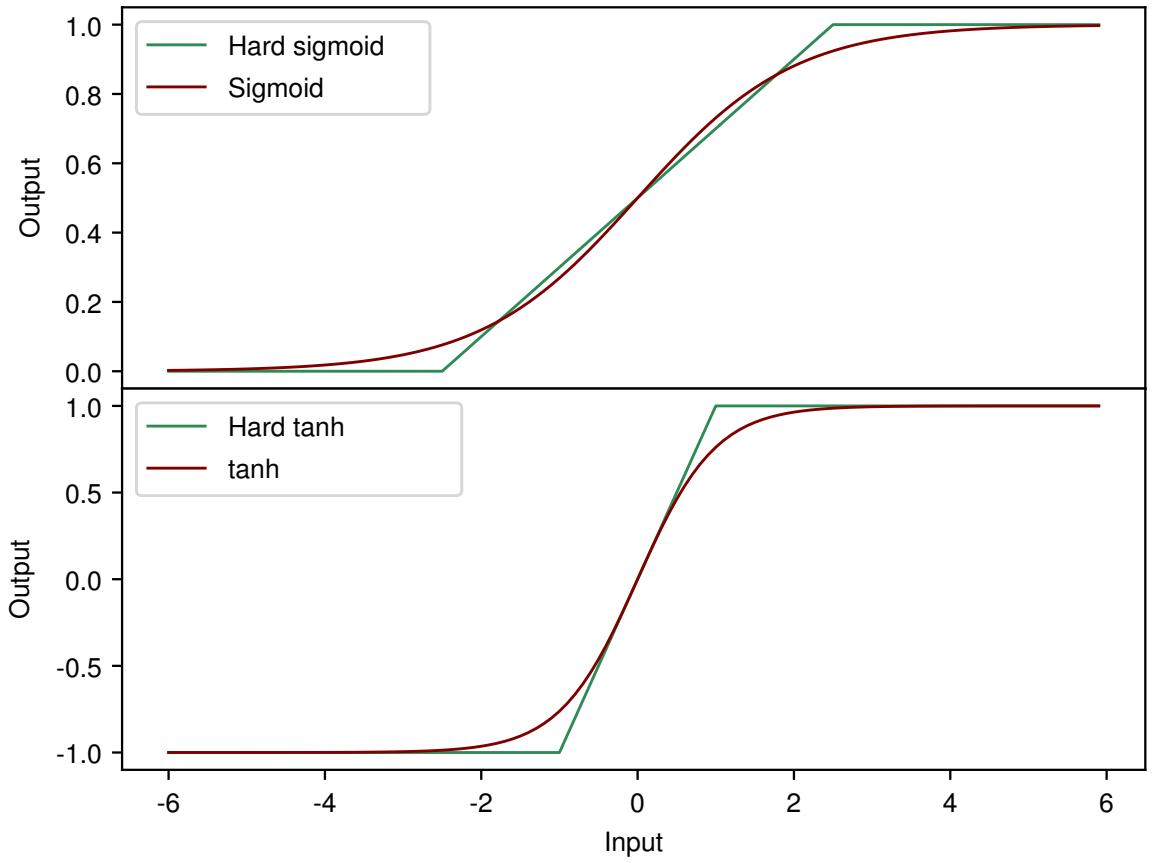


Figure C.1: Hard sigmoid and hard tanh with their original function

A visual representation helps seeing how close those functions are, figure C.1 displays the two together with the function they are modeled after.