Diseño y simulación de redes neuronales basadas en memristores
Design and simulation of memristor-based neural networks.

Trabajo de Fin de Grado
Curso 2022–2023

Autor
Pablo Alex Lázaro
Ignacio Jiménez Gallo

Director
Guillermo Botella Juan
Francisco Jiménez Molinos

Grado en Ingeniería Informática y Software
Facultad de Informática
Universidad Complutense de Madrid

Diseño y simulación de redes neuronales basadas en memristores
Design and simulation of memristor-based neural networks.

Trabajo de Fin de Grado en Ingeniería Informática y Software

Autor
Pablo Alex Lázaro
Ignacio Jiménez Gallo

Director
Guillermo Botella Juan
Francisco Jiménez Molinos

Convocatoria: *Junio 2023*

Grado en Ingeniería Informática y Software
Facultad de Informática
Universidad Complutense de Madrid

29 de MAYO de 2023

Agradecimientos

Queremos expresar nuestro más sincero agradecimiento a nuestros directores de TFG, Guillermo y Francisco, por su invaluable ayuda y compromiso en este proyecto. A lo largo de casi un año de trabajo, su orientación y apoyo han sido fundamentales para el desarrollo y éxito de nuestro trabajo.

Resumen

Diseño y simulación de redes neuronales basadas en memristores

En los últimos tiempos las redes neuronales han ido tomando una importancia cada vez mayor en ámbitos como el reconocimiento de patrones o la visión por computadora. Sin embargo, su uso implica importantes gastos tanto energéticos como de hardware, limitando los ámbitos en los que se puede emplear esta tecnología.

En este contexto, se plantea la viabilidad de utilizar circuitos analógicos basados en memristores como alternativas eficientes en la inferencia de redes neuronales. Los memristores destacan por su configurabilidad y bajo consumo.

Con el objetivo de estudiar la factibilidad del uso estos circuitos, se ha adaptado un modelo fisíco para simular con gran exactitud el comportamiento de los memristores comerciales de la empresa KNOWM. Utilizando este modelo, se han diseñado y simulado múltiples redes neuronales, obteniendo resultados muy satisfactorios.

Palabras clave

Memristor, Algoritmo Genético, Redes neuronales, Variabilidad, Inferencia.

Abstract

Design and simulation of memristor-based neural networks.

In recent times, neural networks have been gaining increasing importance in fields such as pattern recognition and computer vision. However, their usage entails significant energy and hardware costs, limiting the domains in which this technology can be employed.

In this context, the feasibility of utilizing analog circuits based on memristors as efficient alternatives in neural network inference is being considered. Memristors stand out for their configurability and low power consumption.

To study the feasibility of using these circuits, a physical model has been adapted to accurately simulate the behavior of commercial memristors from KNOWM. Using this model, multiple neural networks have been designed and simulated, yielding highly satisfactory results.

Keywords

Memristor, Genetic Algorithm, Neural Network, Variability, Inference.

Contents

List of figures

List of tables

Chapter 1

Introduction

The field of machine learning and its applications in different areas, such as computer vision, natural language processing or robotics, is generating increasing interest in recent times. This has led to the development of advanced techniques and tools for the implementation of artificial neural networks. However, these digital neural networks require large amounts of energy and expensive compute resources, which limits their use in scenarios where energy efficiency is key.

Memristors have recently emerged as a promising alternative in the field of analog computation. Their memristive properties make them theoretically ideal building blocks for analog neural networks. These dedicated analog circuits, where the data and computation are combined forgoing the von Neumann architecture, are orders of magnitude more energy efficient and cheaper to manufacture.

1.1. Motivation

The training process of neural networks is an expensive process, both in terms of energy usage and hardware requirements, but it only needs to be done once. Inference, on the other hand, is performed every time the models are used. Many models in production are being used continuously and by many machines at once, so finding ways to reduce the energy and hardware expenses needed for inference is vital.

In some contexts, such as low power smart devices, the energy efficiency is a necessity. New breakthroughs in this field could bring new capabilities to these types of devices.

There are already various techniques to achieve energy improvements, such as reducing precision and software optimizations, but the hardware constrains of the von Neumann architecture remain.

For many applications in industry, such as video encoding or cryptocurrency mining, single purpose accelerators are designed to speed up calculations and increase energy savings. Similarly, modern mobile chips come with small AI accelerators to perform matrix multiplications faster and consuming less power.

A memristor based analog neural network circuit could be employed in a similar

manner, as a co-processor that would dramatically reduce the costs of running neural networks. The idea of using analog computation to perform inference on neural networks has already been explored in many studies, such as Moreno et al. (2021)

Memristors, with their unique properties, offer several advantages for analog computation over digital systems. Their non-volatility allows them to retain their state even when power is lost, making them ideal for memory storage. Inherently operating in an analog fashion, memristors allow for continuous data representation, which is more efficient for certain computations like neural networks, compared to the binary data representation in digital systems. Furthermore, memristors can be miniaturized and densely packed, providing more computational power in a smaller space. They also consume less power compared to digital transistors, enhancing energy efficiency. Perhaps most intriguingly, memristors can mimic the behavior of biological synapses, making them ideal for neuromorphic computing, a form of analog computation that emulates the brain's architecture. This paradigm is being the subject of recent studies such as Botella et al. (2009)

The main advantage of the memristors when compared to other analog alternatives is that they are both reconfigurable and non-volatile.

In summary, research on the viability of memristors in the field of neural networks is an important and current topic that can have significant implications in the development of computing and electronics.

1.2. Objective

The main objective of this Bachelor's Thesis is to design, simulate and compare inference results of analog neural networks based on memristors with equivalent transistor-based digital neural networks. We will use commercially available memristors to test whether they are a real alternative in their current state.

1.3. Work plan

To achieve the objective, the following steps will be taken:

1. Study the properties and behavior of the device, using commercially available memristors from KNOWM and the Memristor Discovery software.

2. Characterize the device by collecting real data, programming automated scripts and visualizing the results.

3. Adjust a physical model of the memristor to faithfully represent our device.

4. Design circuits to perform inference on analog neural networks, simulate and compare results.

Chapter 2

State of the Art

Next, we will explain the research perspective of the project carried out. The objective is to demonstrate an understanding of the theory necessary for the design of neural networks based on memristors.

2.1. Introduction to Neural Networks

Neural networks are computational models inspired by the human brain, they are widely used in the field of machine learning and can be described as universal function approximators.

These networks consist of interconnected nodes, called neurons, which work together to solve complex problems Haykin (1999). The ability of neural networks to adapt makes them powerful tools in areas such as pattern recognition or natural language processing.

The behavior of these networks is based on the communication and processing of information through weighted connections between neurons. Each neuron receives multiple inputs, which are processed through a non-linear activation function to generate an output. The generated outputs are then sent as inputs to other neurons, creating a transmission and transformation chain of information throughout the network. In supervised learning, the weights of the neural networks are adjusted by comparing the outputs obtained with the desired results and using techniques such as gradient descent.

This field has received a significant boost due to advances in computer technology and the availability of large volumes of data to train the models. In recent years, new types of neural networks have been developed, such as recurrent neural networks, convolutional neural networks, diffusion models, transformer networks and deep architectures Goodfellow et al. (2016); Bishop (2006).

The importance of neural networks in the field of artificial intelligence lies in their ability to learn and generalize from data, enabling them to tackle complex problems and discover patterns in the data. This represents an advancement in various fields, including solutions to problems that are difficult to obtain using traditional algorithmic approaches.

The development of increasingly larger and more complex neural networks is driving the need for specialized hardware technologies that enable efficient implementations of these models.

In this context, a very promising line of research has emerged in the development of hardware for neural networks using memristors.

2.2. Memristor

Memristors are passive electric circuit components, such as inductors, capacitors and resistors. Their existence was theorized in 1971 by Professor Leon Chua in the article Chua (1971), titled: "Memristor-The Missing Circuit Element" and they relate the electric charge and the magnetic flux together. There wasn't a physical implementation of them until very recently, when in 2008, a research team from Hewlett-Packard experimentally demonstrated the existence of these devices Strukov et al. (2008).

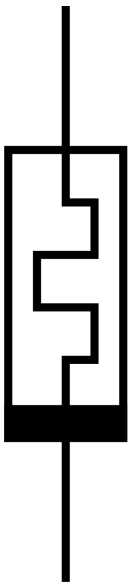


Figure 2.1: Memristor symbol

From a practical point of view, the memristor can be understood as a variable resistor whose resistance depends, in a nonlinear way, on the amount of current that has previously flowed through it. Additionally, it is a non-volatile device, meaning it retains its state even when disconnected from the current. These properties explain the name that Chua gave them: *memristor*, combining *memory* and *resistor*.

This memory of its history is contained in its physical configuration, so it needs to be both:

1. Physically reconfigurable when subjected to a certain voltage level. This reconfiguration must be reversible, going in both directions, from higher to lower resistivity when the voltage is positive and vice versa when it is negative.

2. Stable, not changing its state once the current flow through it stops.

These properties would exist in an ideal device; however, in practice, current implementations have the risk of reaching a physical configuration that becomes irreversible in some cases, such as a state of very high resistance. They also tend to slightly vary their state once disconnected from the current. Information about this internal state of the memristor is captured in a state variable μ. Its behavior is described by two equations:

$$1.\ I(t) = G(\mu(t)) * V(t)$$

$$2.\ \frac{\mathrm{d}}{\mathrm{d}t}\mu(t) = F(\mu(t), V(t))$$

The first equation indicates that the current depends on the conductance (the inverse of resistance) that the memristor has in state μ at time t and the voltage $V(t)$ applied at that moment. The second equation represents the change in the internal state of the device based on the current state μ and the voltage $V(t)$ it is being subjected to. These two equations give the device its non-linear behavior.

The operation of memristors is based on the phenomenon known as the "memristive effect." This effect occurs due to the properties of certain materials, such as metallic oxides, to change their resistance based on the direction and amount of electric charge passing through them. When a voltage is applied to a memristor, it undergoes a physical reconfiguration of its structure, changing its resistance, depending on the direction of the electric current.

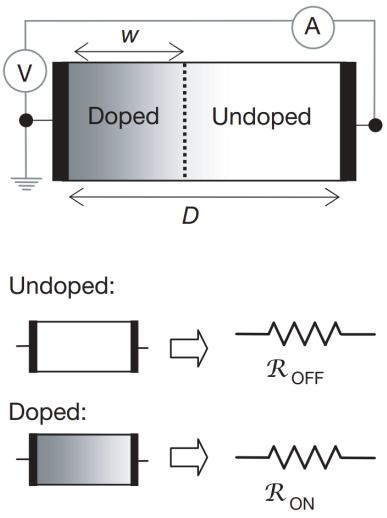


Figure 2.2: Simplified internal configuration of a memristor from Strukov et al. (2008)

The ability to "remember" its previous resistance even after the applied voltage is removed is a key characteristic of memristors. This allows them to maintain their resistance despite the absence of an external power source, making them ideal candidates for the creation of artificial synapses in neural networks. Yang y Strukov (2013); Pershin y Di Ventra (2011).

Significant progress has been made in recent times in the fabrication of these devices, thus opening new possibilities in the field of neuromorphic computing and the design of hardware-efficient neural networks. Indiveri y Liu (2015).

These devices have attracted great interest and study in artificial neural networks. This is because they offer potential advantages in terms of energy efficiency, storage density, and parallel processing, enabling the implementation of high-performance neural networks. Pershin y Di Ventra (2011).

Memristors can emulate the behavior of artificial synapses. Synapses are essential elements in neural networks as they facilitate communication between neurons and information processing. Traditionally, these neural synapses have been implemented using transistors and capacitors, but this emerging technology offers a promising alternative Yang et al. (2022); Hu et al. (2018).

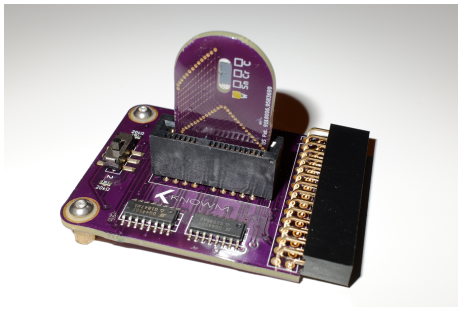
Most of the simulations performed on the papers that explore this idea, like in Hasan et al. (2017), use resistors or very rough models to approximate the behaviour of memristors. We will use state of the art physical models in our simulations, accounting for the inherent variability of the device.

Chapter 3

Memristor Discovery

3.1. Calibration

In this first process, the focus was on getting acquainted with memristor technology. To achieve this, the Memristor Discovery pack from KNOWM was used, which includes a chip with eight pairs of memristors, a board with assigned resistors, and a manual for the software application. To utilize this tool, the logic analyzer of the Analog Discovery 2 from Digilent was employed.



Memristor Discovery

Analog Discovery 2

Figure 3.1: Memristor Discovery and Analog Discovery 2

The setup of the Analog Discovery 2 using the Waveforms software was initiated. Firstly, the logic analyzer was connected to a laptop via USB. Once connected, an automatic test of wave generation functions was performed. To do this, the thirty-pin connector was used, connecting waveform generator 1 to oscilloscope 1 and waveform generator 2 to oscilloscope 2. Next, the software was configured to generate a square wave and a sinusoidal wave, as shown in figure 3.2.

Next, the logic analyzer was calibrated to ensure accurate measurements. A multimeter was used to measure the voltage generated at each step of the calibration process to correct the generated signals.

With the calibrated logic analyzer and the memristors functioning properly, we can now begin to conduct our own experiments using KNOWM's memristors through the Memristor Discovery application. In order to familiarize ourselves with this

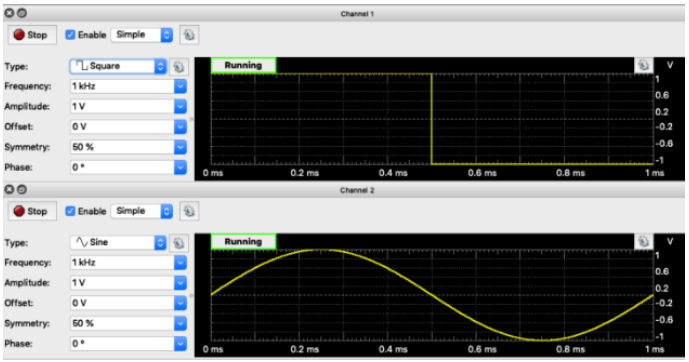


Figure 3.2: Waveform generators configured for self-test.

hardware, we will perform various tests, ranging from obtaining hysteresis curves to designing our own simple neural network.

3.2. DC Experiment

First, we started by exploring the DC option of the KNOWM Memristor Discovery software. This functionality allows us to characterize the electrical properties of a memristor easily and efficiently. So, to initiate our DC experiment, we apply a DC voltage to a memristor and then measure the resulting current. The current-voltage (I-V) and conductance-voltage (G-V) graphs are then plotted to show the memristor's characteristic pinched hysteresis behavior.

The following figures show the I-V and G-V graphs for a memristor driven with an input sinusoidal waveform. The I-V graph shows that the memristor has a non-linear resistance, also appreciated in the G-V graph that shows how conductance (G) does not vary proportionally to the applied voltage (V). The hysteresis loop shows that the memristor has a memory effect, meaning that its resistance depends on its previous state.

This DC experiment is a powerful tool for characterizing memristors. It can be used to measure a variety of important parameters, including the resistance, capacitance, switching threshold, and non-linearity. This information can be used to design circuits that take advantage of the memristor's unique properties.

It is worth noting that the yellow-colored graphs represent the data obtained if the resistor were not applied, while the blue-colored graphs represent the behavior of the memristor in conjunction with the resistor.

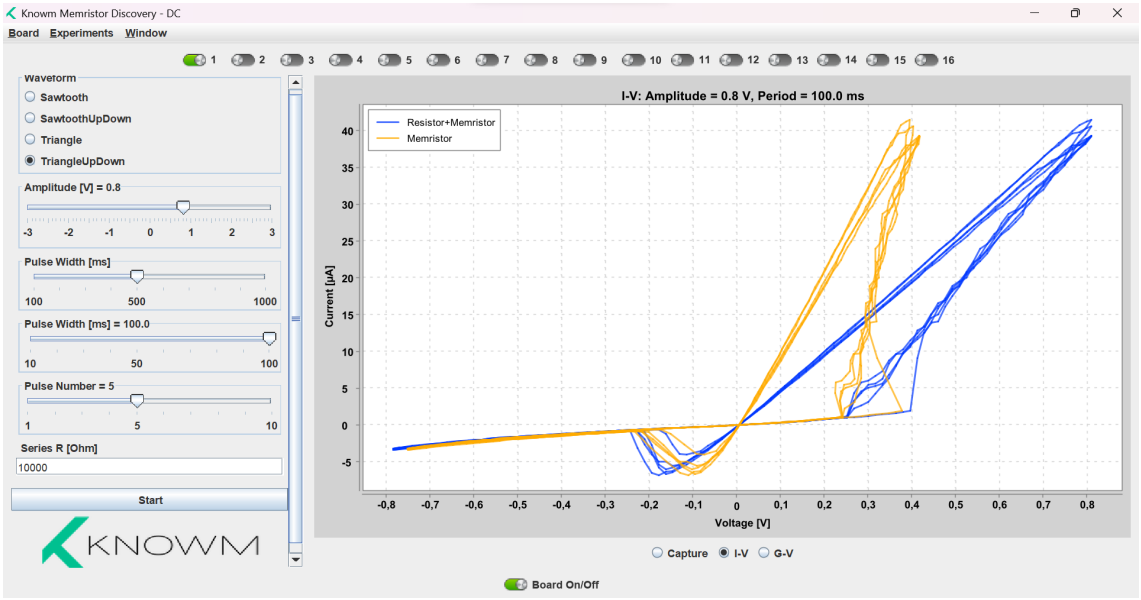


Figure 3.3: DC Response I-V.

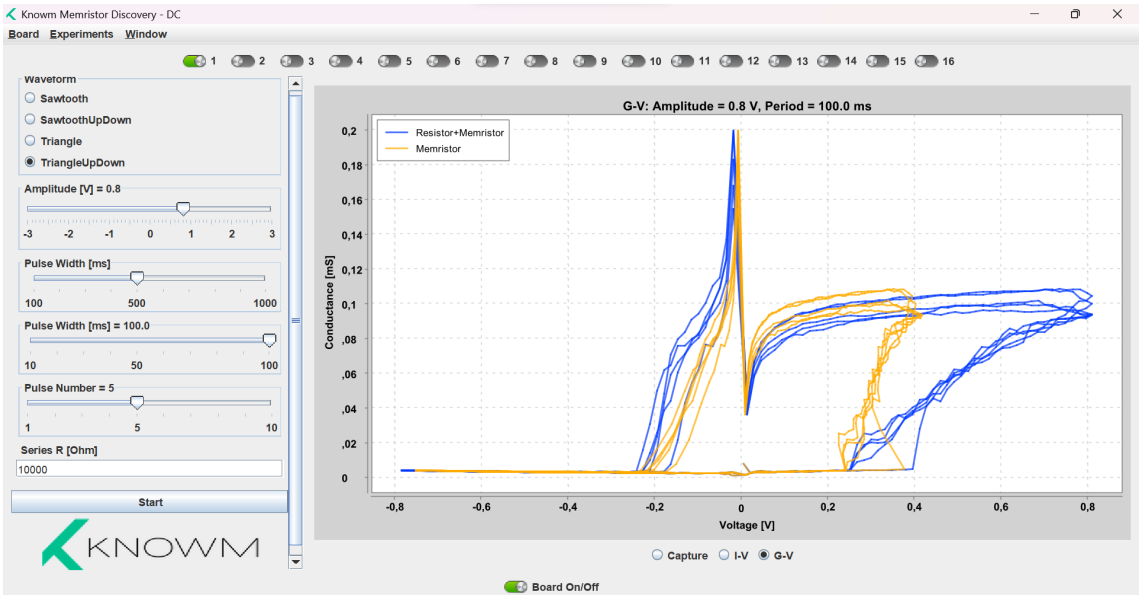


Figure 3.4: DC Response G-V.

3.3. AC Experiment

Next, we proceed to investigate the possibilities of the Hysteresis option within the same software. To do this, we will create a new experiment. In it, a memristor is driven by an AC voltage source and the resulting current is measured. The current-voltage (I-V) graphs are then plotted to show the memristor's characteristic pinched hysteresis behavior.

The frequency of the AC voltage source affects the memristor's hysteresis behavior. To test it, we applied three types of ramps, that is, three different frequencies. These frequencies were 10Hz, 100Hz, and 1000Hz. At low frequencies the memristor

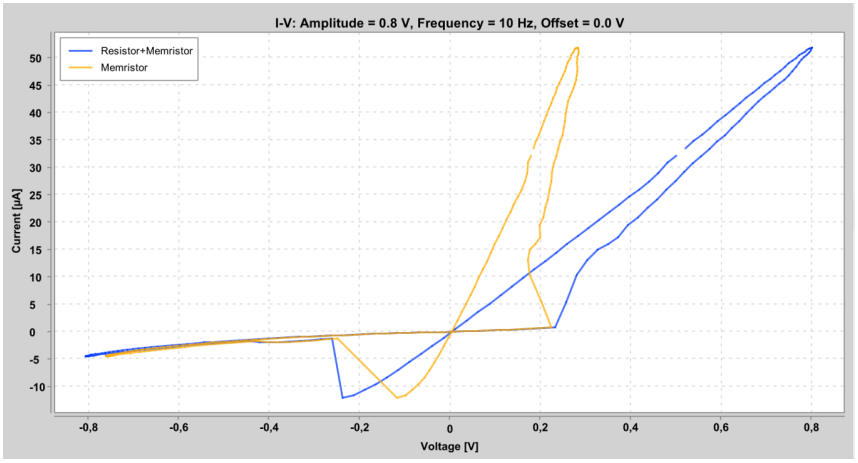


Figure 3.5: AC Experiment 10Hz

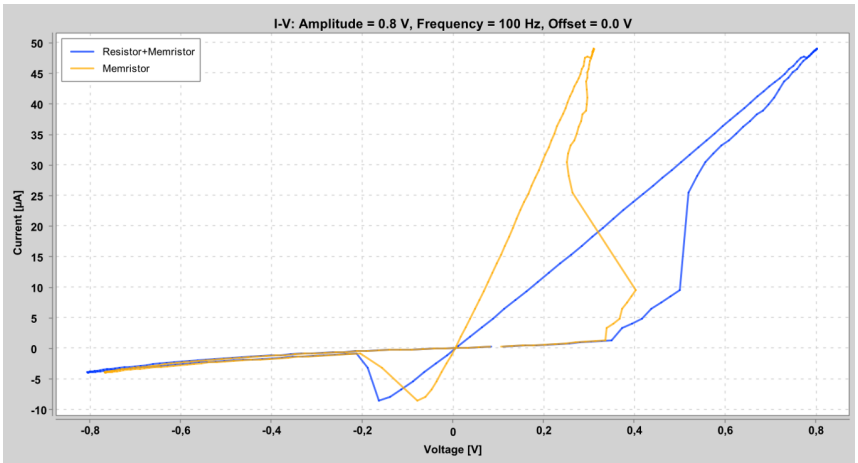


Figure 3.6: AC Experiment 100Hz

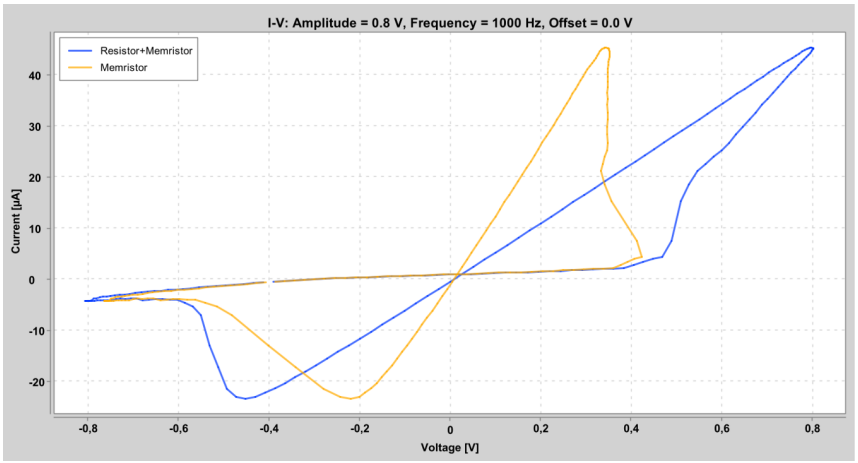


Figure 3.7: AC Experiment 1000Hz

has a larger hysteresis loop. This is because the memristor has time to switch between resistance states at low frequencies. While at high frequencies, the memristor has a smaller hysteresis loop, because the memristor spends less on either state and

more time switching between the resistance states. This can be verified by comparing the different plots in Figures 3.5, 3.6, 3.7, where it can be observed that the hysteresis curve for a frequency of 1000Hz is flatter than the curve for 10Hz.

3.4. Resistance Programing

Continuing with the option of Hysteresis, a new experiment was conducted. In this experiment, a memristor is driven by a DC voltage source and the resulting current is measured. The current is then used to calculate the memristor's resistance. The memristor's resistance can be programmed by changing the DC voltage source.

This Resistance Programming experiment works by applying a DC voltage to the memristor. The DC voltage causes the memristor to switch between two resistance states. The resistance state that the memristor switches to depends on the polarity of the DC voltage.

The Resistance Programming experiment can be used to program the resistance of a memristor in a variety of ways. One way to program the resistance is to apply a DC voltage to the memristor for a specific amount of time. The longer the DC voltage is applied, the larger the change in the memristor's resistance. The HRS

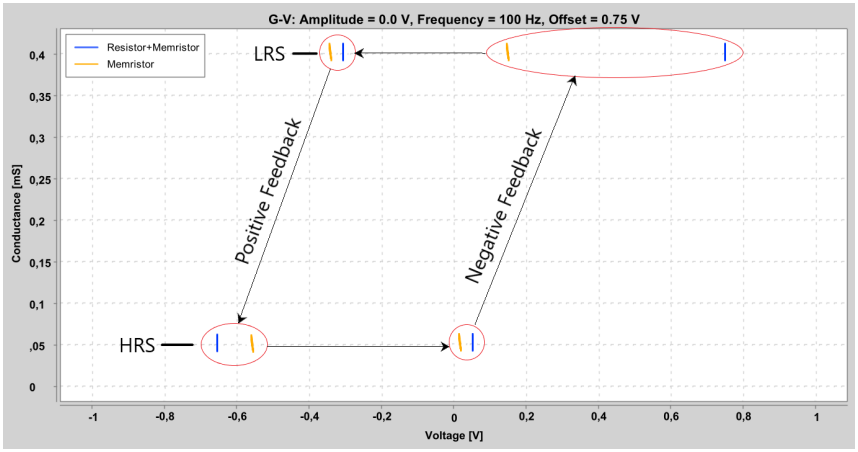


Figure 3.8: Resistance Programming

and LRS refer to the high resistance state and the low resistance state, respectively. The HRS is the state in which the memristor has a high resistance, and the LRS is the state in which the memristor has a low resistance. The memristor can be programmed to switch between the HRS and LRS by applying a voltage to the memristor.

3.5. Pulse Response

We continue exploring the Pulse option. For this experiment, a memristor is driven with a series of pulses of varying amplitude and width. The response of the memristor to these pulses is then measured and analyzed.

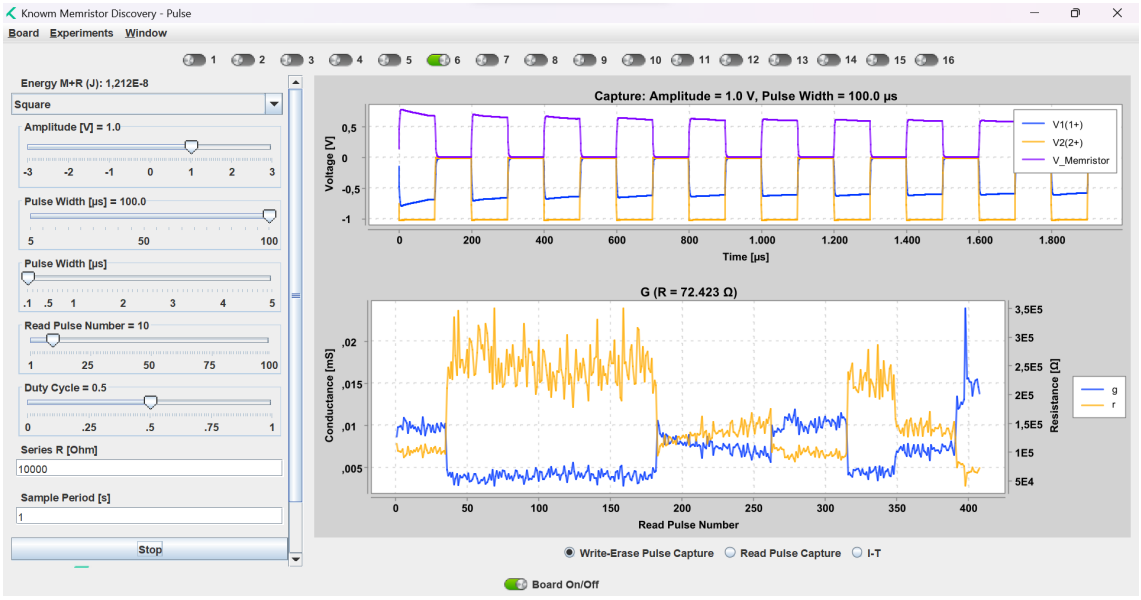


Figure 3.9: Pulse Response

The graph shows at image 3.9 the relationship between the conductance and resistance of a memristor as a function of the number of read pulses applied to the memristor. The graph shows that the conductance of the memristor increases with each pulse, while the resistance decreases. This is due to the fact that the memristor is a non-volatile memory device, meaning that it can store information even after the power is removed.

3.6. Synapse Experiment

This experiment was run in Synapse12 option of the KNOWM software. It consists of a series of elemental kT-RAM instructions that are used to drive the synapses, followed by a series of FLV read instructions that are used to measure the synaptic state and synaptic pair conductances. This experiment is repeated multiple times to observe the continuous response of the synapses.

The significance of this experiment is that it provides a detailed characterization of the functionality of kT-RAM differential-pair memristor synapses. The figure 3.10 demonstrates that the synapses can be driven with a variety of elemental instructions, and that the synaptic state and synaptic pair conductances can be accurately measured with FLV read instructions. This experiment also demonstrates that the synapses exhibit a continuous response, which is a key requirement for their use in neuromorphic computing applications.

3.7. Classifier Experiment

Finally, we started experimenting with the Classify12 option, that is a software program that allows users to test the performance of a memristor-based classifier.

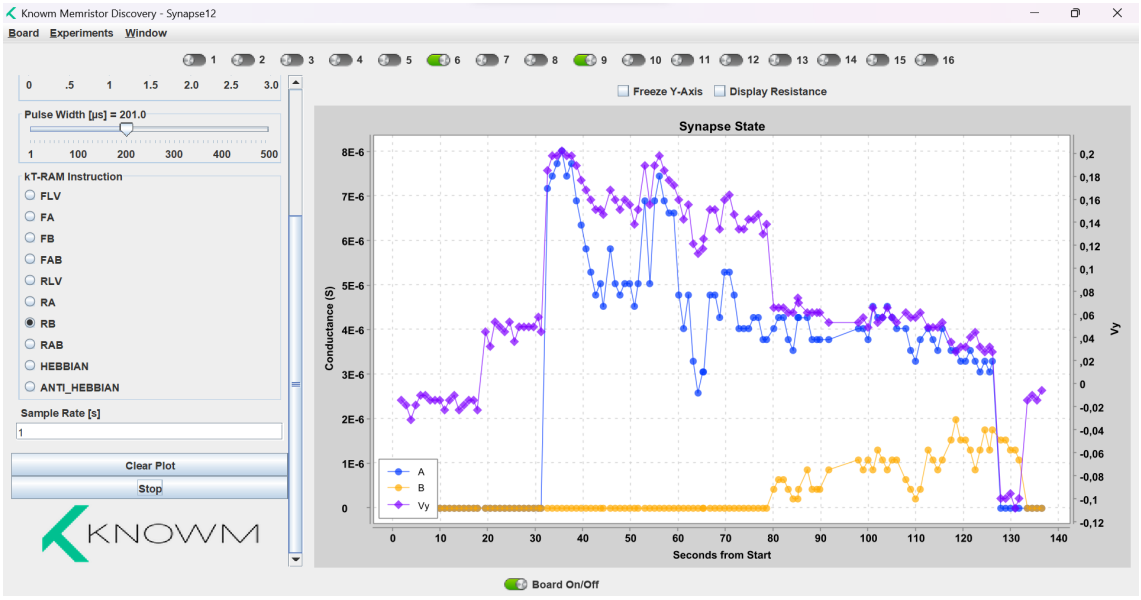


Figure 3.10: Synapse Experiment

The program uses a 1X16 linear array chip to create an 8-synapse neuron. The user can select the forward and reverse pulse voltage amplitudes and pulse widths for each synapse.

So, we create a Classifier Experiment that is significant because it allows us to test the performance of a memristor-based classifier without having to design our own hardware.

Dataset	True Patterns	False Patterns
Ortho2Pattern	[0,1,2,3]	[4,5,6,7]
AntiOrtho2Pattern	[4,5,6,7]	[0,1,2,3]
Ortho4Pattern	[4,5],[6,7]	[0,1],[2,3]
AntiOrtho4Pattern	[0,1],[2,3]	[4,5],[6,7]
Ortho8Pattern	[4],[5],[6],[7]	[0],[1],[2],[3]
AntiOrtho8Pattern	[0],[1],[2],[3]	[4],[5],[6],[7]

Table 3.1: Classifier Experiment Datasets

To use the Classifier Experiment, we first need to select the forward and reverse pulse voltage amplitudes and pulse widths for each synapse. Afterwards, we select one of the training data set shown in table 3.1 . The training data set should consist of a set of input and output pairs. The input pairs are the patterns that the classifier will be trained to recognize. The output pairs are the desired outputs for each input pattern.

When the training data set has been selected, we start the training process. The training process will take several iterations. After each iteration, the classifier will be updated to improve its performance. The training process will stop when the classifier has reached the desired epoch.

Once the training process is completed, without resetting the weights obtained

in the memristor pairs, we run it with a new data set. The new data set should be used for training. The classifier's performance on the new data set shows how well the classifier will fit a new target despite having been trained before.

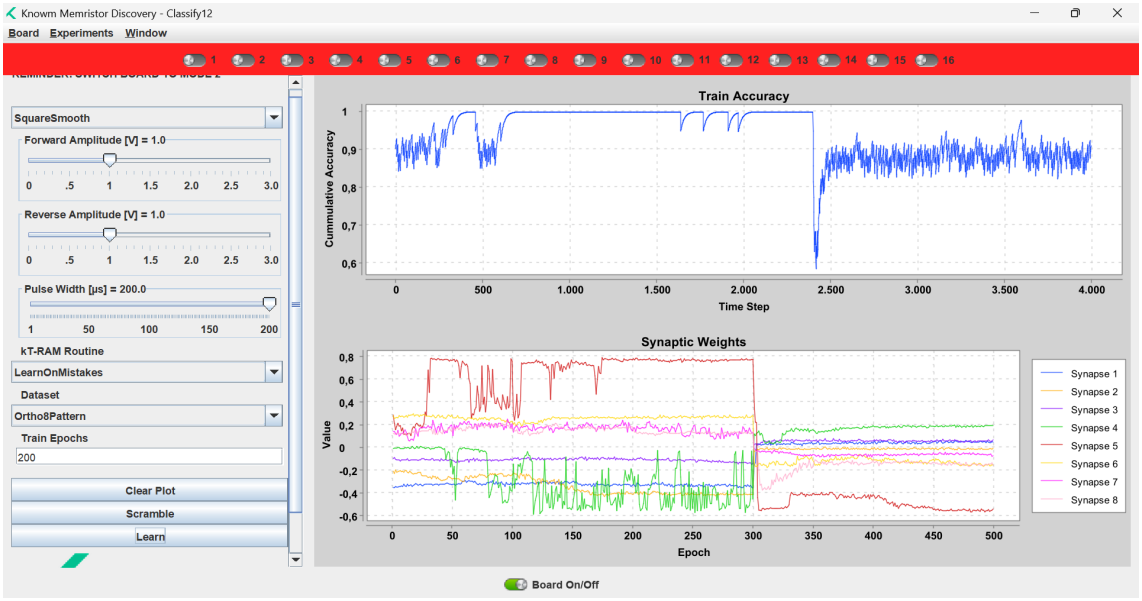


Figure 3.11: Resistance Programming

In Figure 3.11, you can see how the classification is produced using the Ortho8Pattern dataset, where synapses 4, 5, 6, and 7 had to classify as true (positive), while the rest of the synapses had to classify as false (negative). Shortly after starting the execution, you can observe how it reaches 100% accuracy. Without resetting the resistance weights, the dataset is changed to AntiOrtho8Pattern, which should invert the values to be classified, and it can be observed how all synapses modify their weights, obtaining an accuracy close to 90%.

To delve deeper into the functioning of memristors as synapses, we designed a simple classification experiment by creating both a new dataset shown in table 3.2 and a new learning method. We wrote our code with the extensions in Java in the source code of the program.

Dataset	True Patterns	False Patterns
myDataSet	[0, 2, 4], [2, 4, 6]	[1, 3, 5], [3, 5, 7]

Table 3.2: Custom Dataset Memristor

The learning method called LearnComboReverse consists of learning only when it must classify as false, or when it fails.

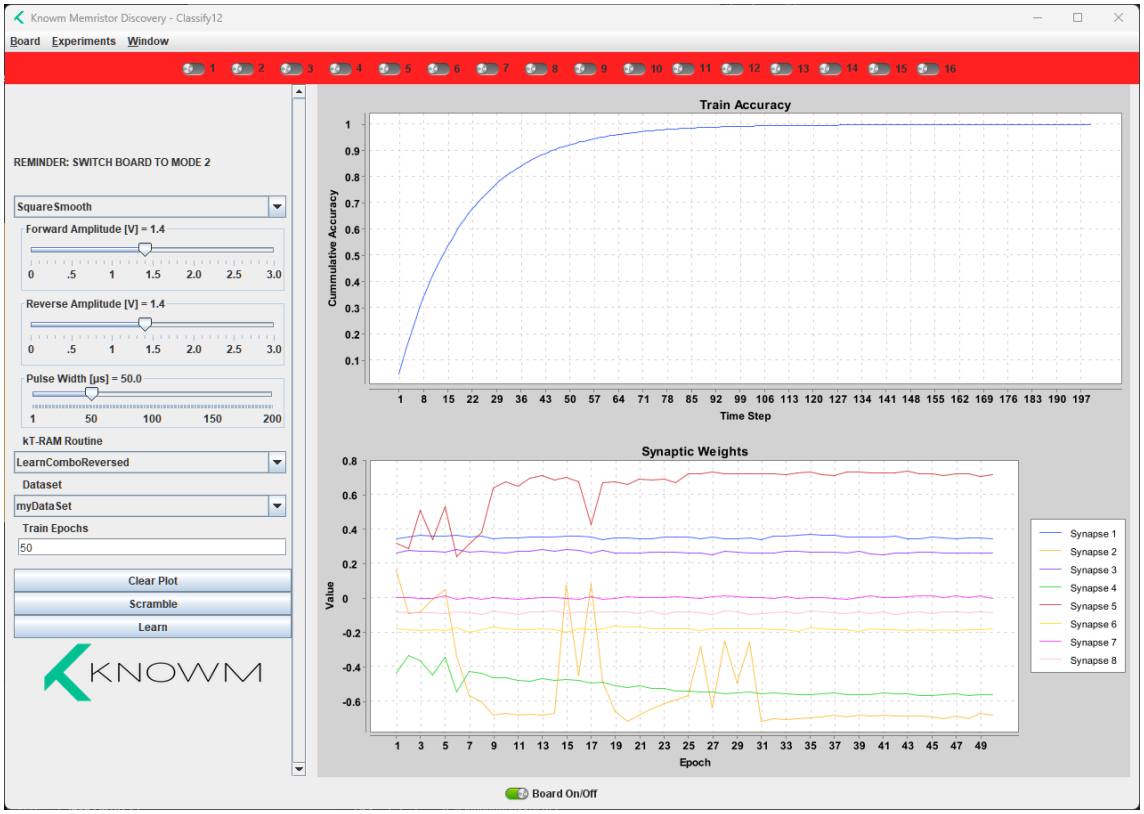


Figure 3.12: Custom Classification

Chapter 4

Device Characterization

Once the experimentation phase was done and an intuitive understanding of the device had been achieved, we set out to experimentally characterize it. This consists of obtaining the numerical data needed to determine some of the devices' most important properties, such as the set and reset voltages and the HRS and LRS currents ratio for various voltage ramps. These results are obtained by means of automated extraction scripts that are executed on the experimental data we collected from the device.

4.1. Data Collection

The DC Experiment environment from Knowm Memristor Discovery program provided us with the perfect means to visualize and obtain the data we needed. We first scanned through a few of the sixteen memristors on the chip until we selected one that showed good and consistent behavior. For that memristor, we applied 200 pulses of ramp voltage stress (triangle shaped voltages) with an amplitude of $0.8V$ through its $10KW$ resistor; and we did so for three different pulse periods: $10ms$, $100ms$ and $1000ms$, which result in ramps of $0.8V/s$, $8V/s$ and $80V/s$ respectively.

The program only allowed us to apply ten pulses at a time, so we slightly modify its source code to see if we could obtain the two hundred cycles in a single execution, eliminating the downtime involved in saving the data and rerunning the execution every time. We thought this downtime could affect the results due to the non-ideal nature of our devices, resulting in noticeable jumps in the measurements due to the tendency of our memristors to lose some conductance when idle. After examining the source code of the Knowm Memristor Discovery and Waveforms programs, both written fully in Java, we came across a hardware buffer size limitation on the Discovery board itself, where the results were stored before being sent via USB to the computer. We managed to double the number of cycles to 20 but couldn't go any further. The effects we feared are slightly noticeable in some of the results, but for the most part, it didn't present a real issue.

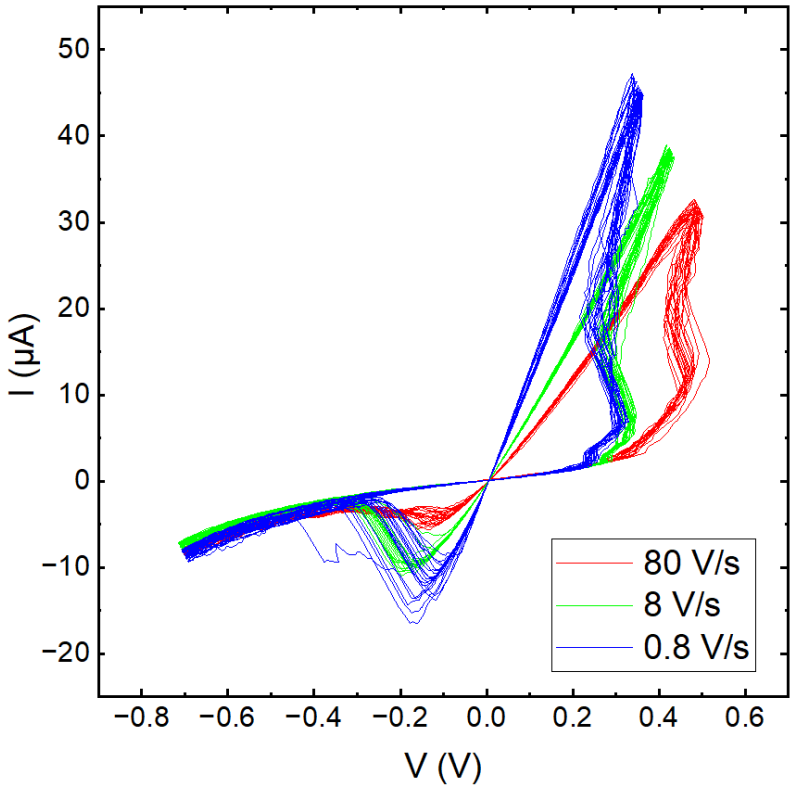


Figure 4.1: Measured current for the applied voltage ramps

4.2. Data and script adaptation

The first properties we obtained were the set and reset voltages of the device. These are the points at which the device starts to meaningfully transition from the HRS to the LRS for the set voltage and vice-versa for the reset voltage. This information is interesting since it provides us with an interval ($V_{reset} - V_{set}$) of the voltages we can apply to the device expecting to not see any meaningful changes in its internal state.

To obtain these two points, we had to find the parameters that would better met our specific needs using a MATLAB script for the methods from Maldonado et al. (2022). We required the input data to be separated by the set and reset of each cycle and for the currents to be in absolute values. For that purpose, we developed a Python script.

Each method provided us with different results for each cycle. We selected the ones that better represent the experimental measurements, which were the third method from the paper for both the set and reset. For the set, the method selected the points where there was a maximum separation from the straight line that joined the first and end points of each set curve, as shown in 4.2. For the reset, the method simply selected the points with the highest current.

We were also interested in showing the cycle to cycle variability at a given voltage points, and more precisely, at a reading voltage point, such as $0.05V$ or $0.1V$ in both HRS and LRS. For this purpose, we had to write another python script to estimate

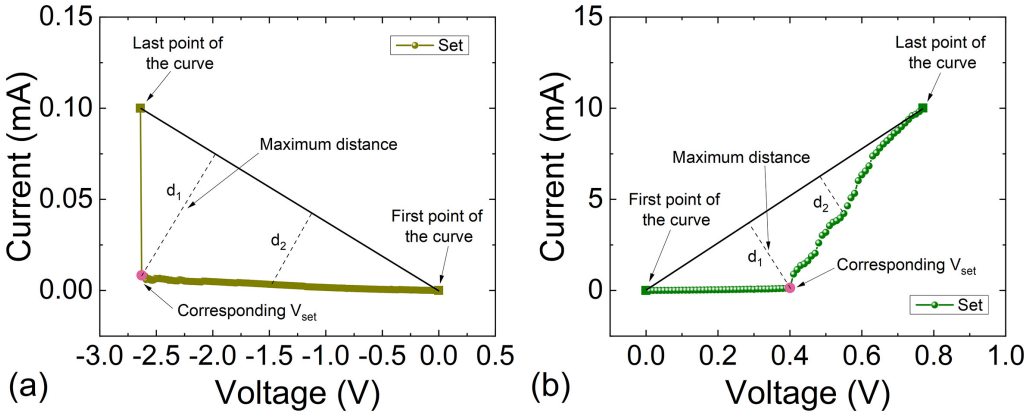


Figure 4.2: Method for selecting the set point based on maximum separation

the current at these points using a linear interpolation on the real data since we didn't have the values for those exact voltages.

4.3. Results

The results presented here, are soon to be published in a paper named "Characterization and modeling of variability in commercial self-directed channel memristors" in the 14th Spanish Conference on Electron Devices (CDE 2023). Jiménez-Gallo et al. (2023).

We chose to use a cumulative distribution function (CDF) to illustrate both the voltage set and reset of the device across all the 200 cycles. The results, seen in figure 4.3, are fairly consistent and in line with the manufacturer's advice to use $1V$ as probing voltage. The resulting interval of voltages in which the device can operate without meaningfully changing its resistance is roughly from $-0.1V$ to $0.25V$. This information will be useful when designing the neural networks.

It is also interesting to see in figure 4.4 how this voltage points vary from cycle to cycle. Although a slight general tendency can be intuited, the variability seems to be consistent across all voltage ramps.

Lastly, the current estimated at $0.1V$ and $0.05V$ at HRS and LRS can be seen in figure 4.5 for the $8V/s$ measurements. Except for some cycles around cycle 100, they appear very consistent. However, when we plot the ratios of currents in LHR and HRS for $0.1V$, we can see much more variability (figure 4.6). There are a few reasons for this, firstly we have to take into account the significant measurement noise when dealing with these small currents and the fact that interpolation was performed to obtain these values. We also think that in this case some of the variability can be attributed to the interrupted measurement process that we were forced to do, since some of the jumps appear to coincide exactly with multiples of 20.

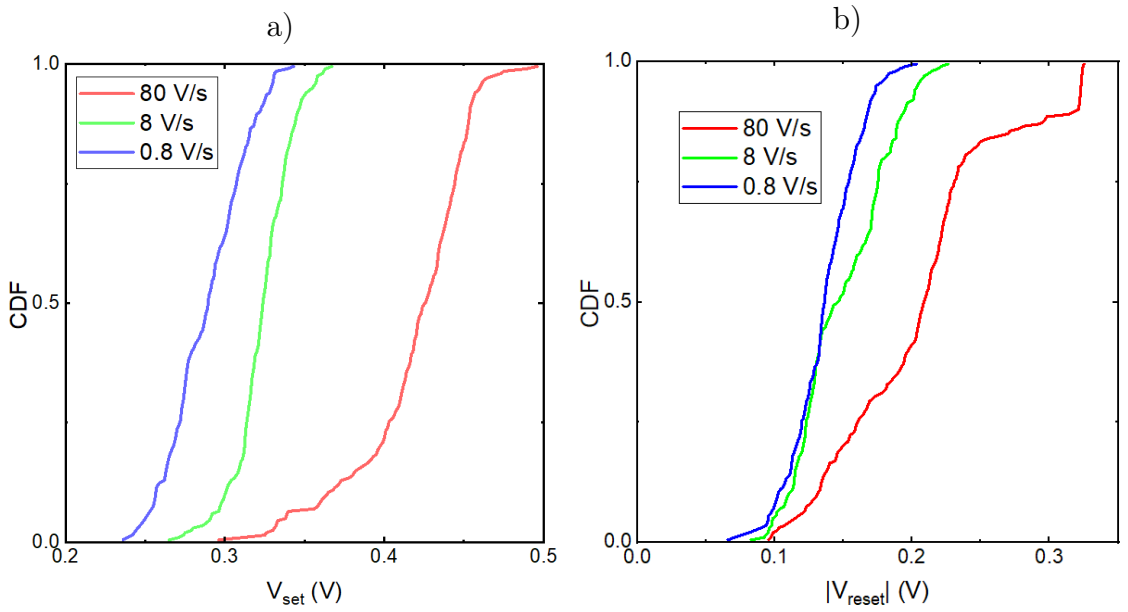


Figure 4.3: Cumulative distribution functions of the calculated set (a) and reset (b) voltages

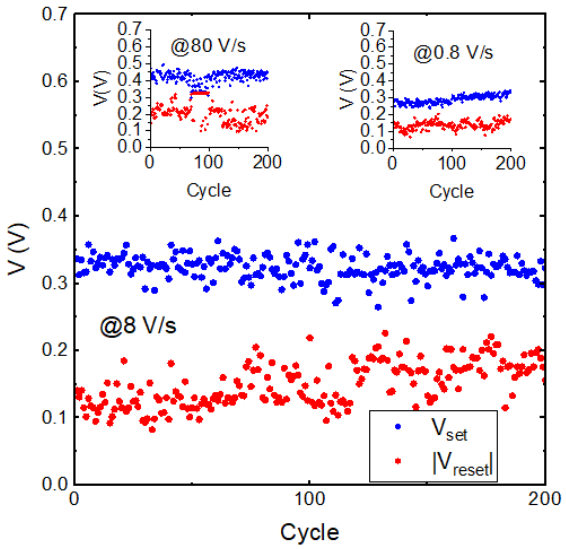


Figure 4.4: Set and reset voltages versus cycle number, for the three voltage ramp rates

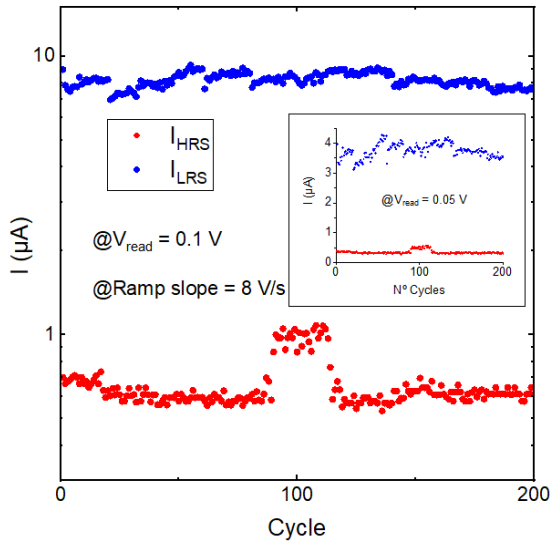


Figure 4.5: Cycle to cycle variability of the current measured at 0.1V in HRS and LRS

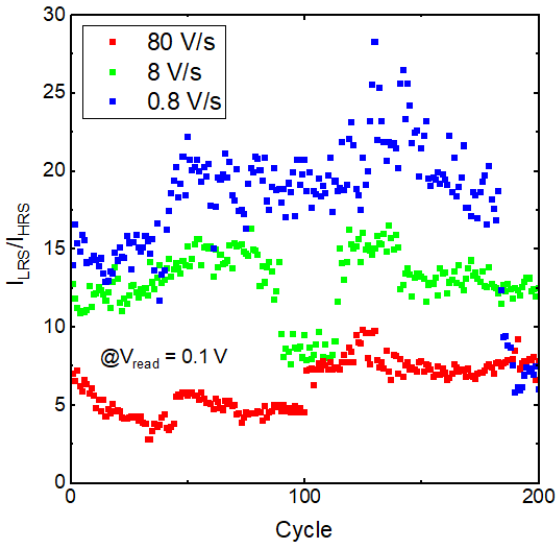


Figure 4.6: figure
HRS and LRS measured currents ratio versus cycle number for 0.1V

Chapter 5

Physical Model

Due to the limitation of having only sixteen memristors for the creation of minimally complex neural networks such as MNIST, a physical model has been created in LTspice to emulate the behavior of memristors.

To do this, we relied on the parameters described in F.L. Aguirre (2022). As seen in the model shown in Figure 5.1 and 5.2, we applied a voltage of 1.2V to the memristor, obtaining the response shown in Figure 5.3 using the LTspice simulator, graphing a curve very similar to the characteristic hysteresis curve of memristors on the I - V(app) axes.

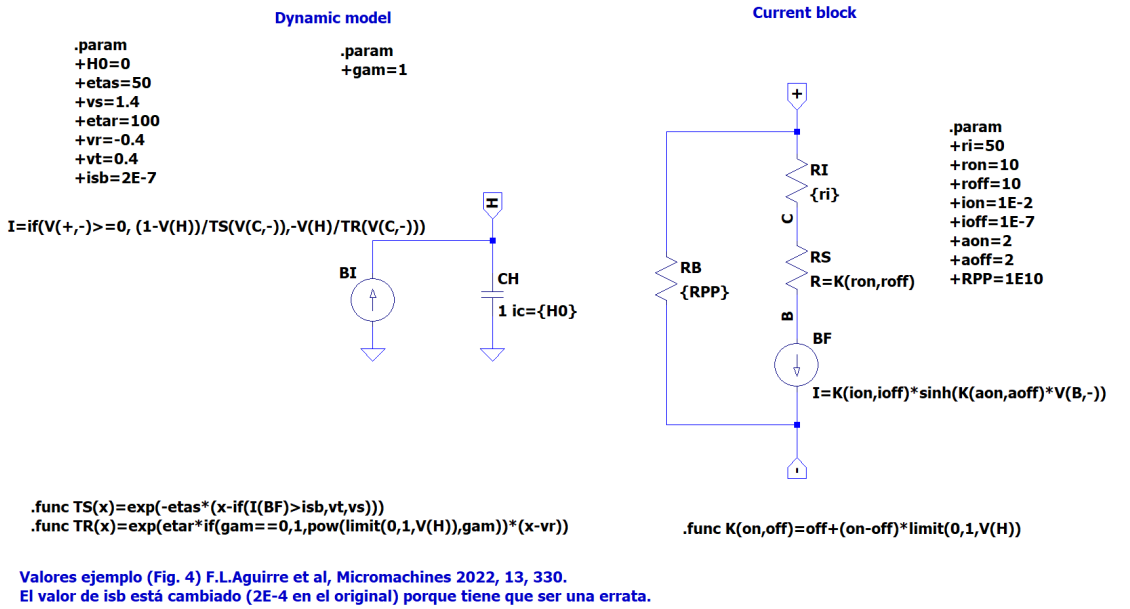


Figure 5.1: Memristor Physical Model

However, the model is not yet complete as commercial memristors exhibit a certain level of variability despite having stochastic components. This fact causes them to behave differently during two different cycles and leads to different memristors functioning differently despite being under the same conditions.

Due to this situation, we have emulated this variability behavior within our

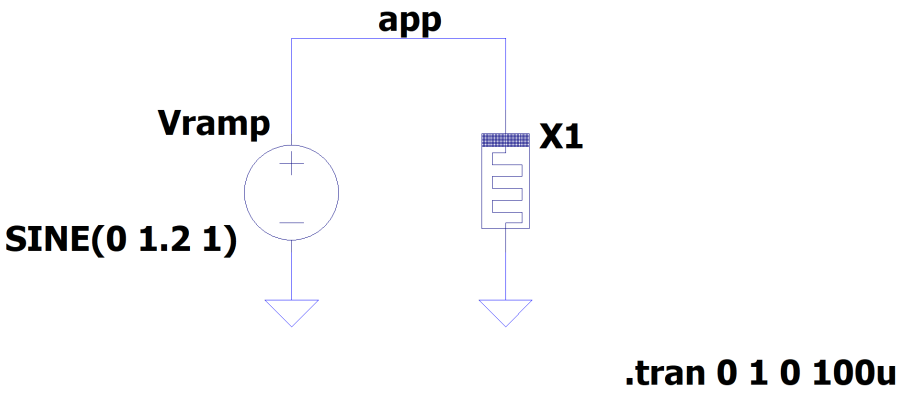


Figure 5.2: Physical Model

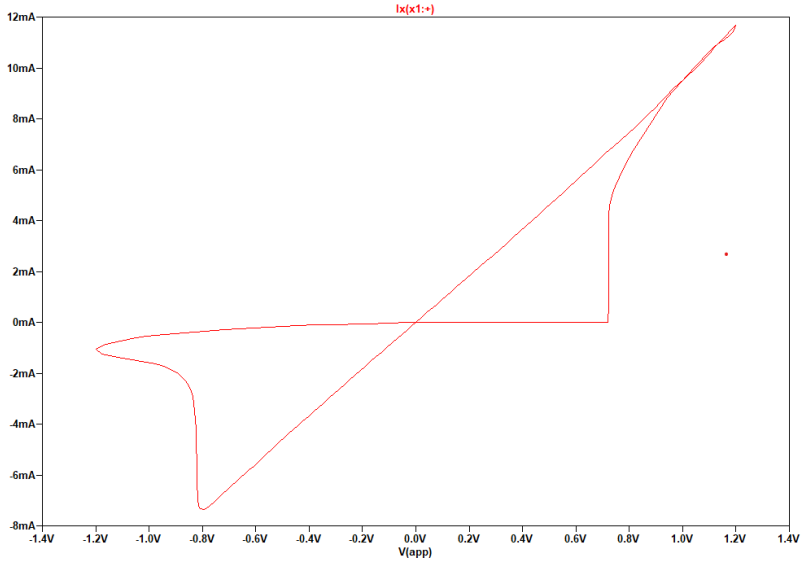


Figure 5.3: LTspice simulation

model. To do so, we have added the parameters mentioned in the paper Roldán (2023). In order to appreciate the variability, unlike the previous model, we execute the model a given number of times, in the case of Figure 5.6, one hundred times. It can be clearly observed that the result is very different from our empirical data. However, this will be addressed by using a genetic algorithm to adjust the initial parameters as to match our devices as accurately as possible.

To perform a simulation with variability, we use the Gaussian function 'gauss' in LTspice. This function generates a random number from a Gaussian distribution with a standard deviation determined by the parameter used. The function is applied to the new parameters that serve to emulate variability.

The data used as a starting point to create the model are from paper F.L. Aguirre (2022), which are shown in Table 5.1.

To utilize this model created in LTspice, it was decided to integrate it with Python in order to streamline parameter changes and, in the future, facilitate the development of genetic algorithms and neural network creation. For this purpose,

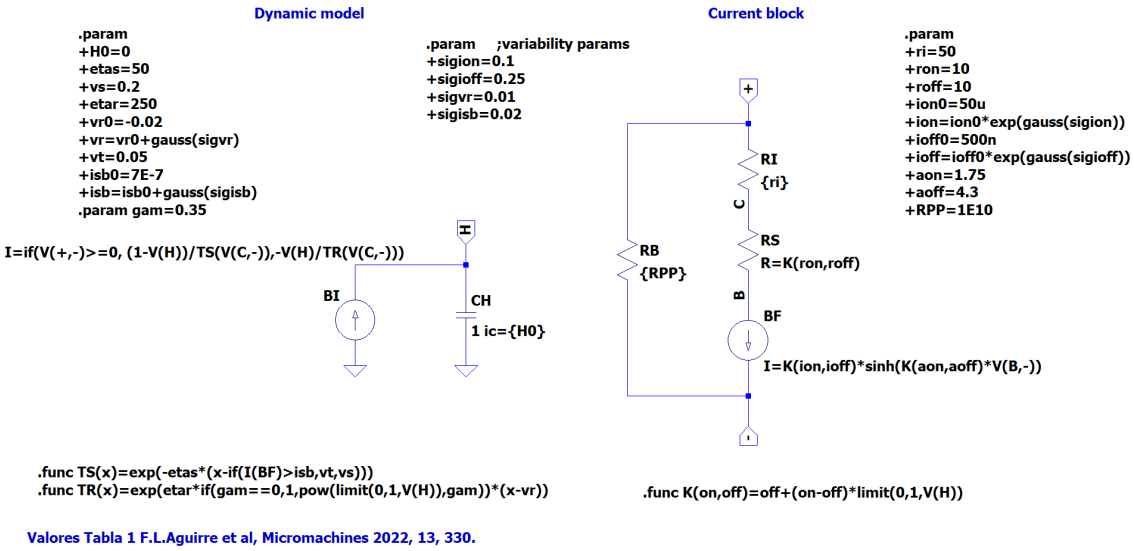


Figure 5.4: Memristor Physical Model with variability

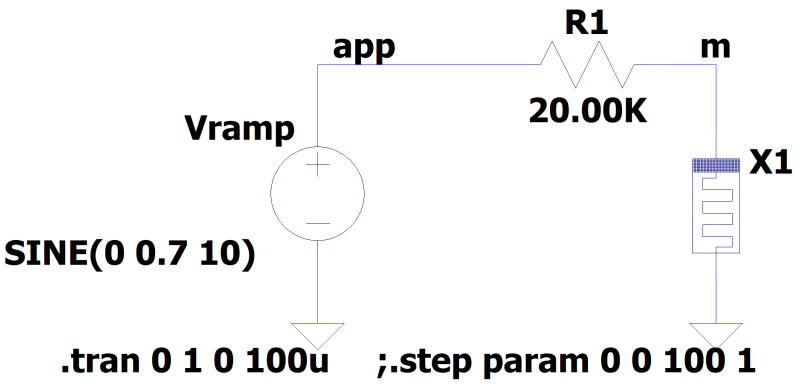


Figure 5.5: Physical Model with variability

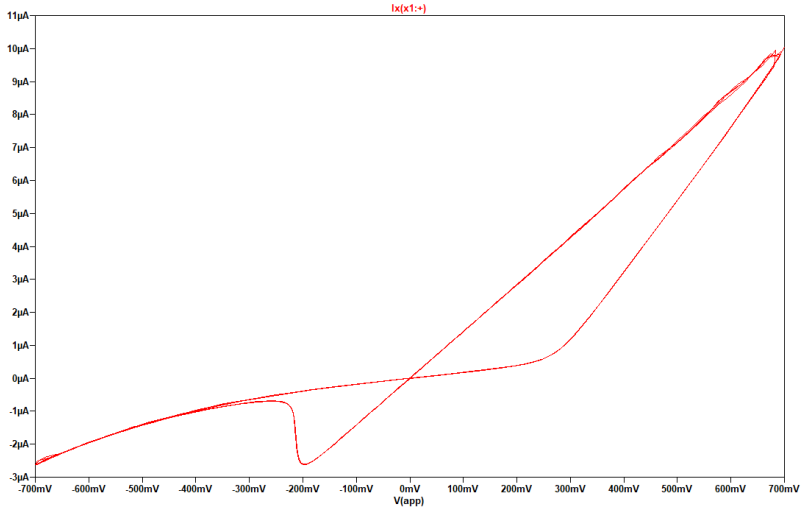


Figure 5.6: LTspice simulation with variability

roff = 10Ω	ri = 50Ω	ron = 10Ω	ion0 = 4.95µA
ioff0 = 2.48µA	aon = 1.71	aoff = 2.58	H0 = 0
etas = 50	etar = 250	Vs = 0.2V	Vr0 = -0.02V
Vt = 0.05V	isb = 7µA	gam = 0.35	sigion = 0.1
sigioff = 0.25	sigvr = 0.01	sigsb = 0.02	RPP = 1E10

Table 5.1: Init variability params

the PyLTSpice Python library was used, which enables a quick and straightforward connection with the models created in LTspice. Additionally, this library allows running the model with desired parameters directly from Python. Thanks to these features, parameter adjustments were made by creating a genetic algorithm.

5.1. Genetic Algorithm

A genetic algorithm was implemented to find the best parameters for the physical model.

A genetic algorithm is an optimization algorithm that mimics the process of natural selection. It works by iteratively creating a population of solutions, evaluating their fitness, and then using genetic operators to create a new population of solutions. The genetic operators are used to introduce variation into the population, which helps to ensure that the new population can contain solutions that are better than the old population. The process is repeated until a satisfactory solution is found.

The algorithm was implemented in python, and the following steps were taken:

1. In each generation, the learning rate was decreased incrementally.

2. The current in $Vramp$ and the voltage in *app* were obtained during the execution in LTspice with the current population, obtaining the simulated I-V curves.

3. The loss of each individual was calculated by mapping the simulation curve onto the real ones and calculating the square differences between them.

4. The average of losses was updated, and the population was sorted from the lowest to the highest loss.

5. The best individual was updated, and the half with the worst loss was eliminated while the remaining half was mutated.

The algorithm was repeated until a satisfactory solution was found.

The following image shows the results obtained, being the blue graph the original data and the red one the data obtained with the model after the execution of the genetic algorithm.

Next, this genetic algorithm was modified to incorporate the variability present in commercial memristors such as the one used by KNOWM. Memristors have the

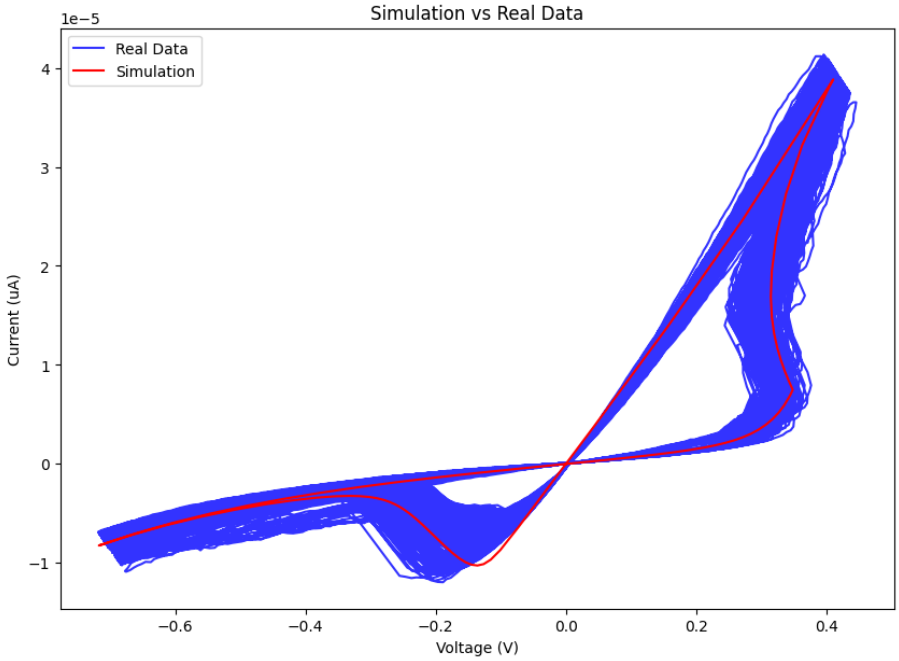


Figure 5.7: Genetic Algorithm results vs empirical curves

ability to modify their resistance based on the electric charge passing through them. Despite the stochastic nature of these devices, there are variations between different cycles or among different devices under the same conditions, thus demonstrating the aforementioned variability.

This new genetic algorithm was applied to all the parameters of our current parameters, resulting in the following values, as shown in the figure, with a good agreement between the experimental and simulated curves.

roff = 10Ω	ri = 50Ω	ron = 10Ω	ion0 = 50µA
ioff0 = 2.5µA	aon = 1.72	aoff = 2.7	H0 = 0
etas = 17	etar = 70	Vs = 0.14V	Vr0 = -0.03V
Vt = 0.1V	isb = 6.2µA	gam = 0.29	sigion = 0.05
sigioff = 0.07	sigvr = 0.01	sigsb = 2.04e-07	RPP = 1E10

Table 5.2: Genetic variability params

The 'steps' in each LTspice simulation could not be parallelized because all the parallel simulated steps used the same seed. As a result, the same values were obtained for the sigion, sigioff, sigvr, and sigisb parameters responsible for emulating the spoken variability. This meant that for each individual, all the steps performed used the same Gaussian distribution, resulting in no variability.

However, the execution of different individuals in the population could be parallelized, greatly speeding up the simulations. This was done from Python by launching multiple instances of LTspice, one for each individual. We were able to achive a speed up of up to $6.25X$.

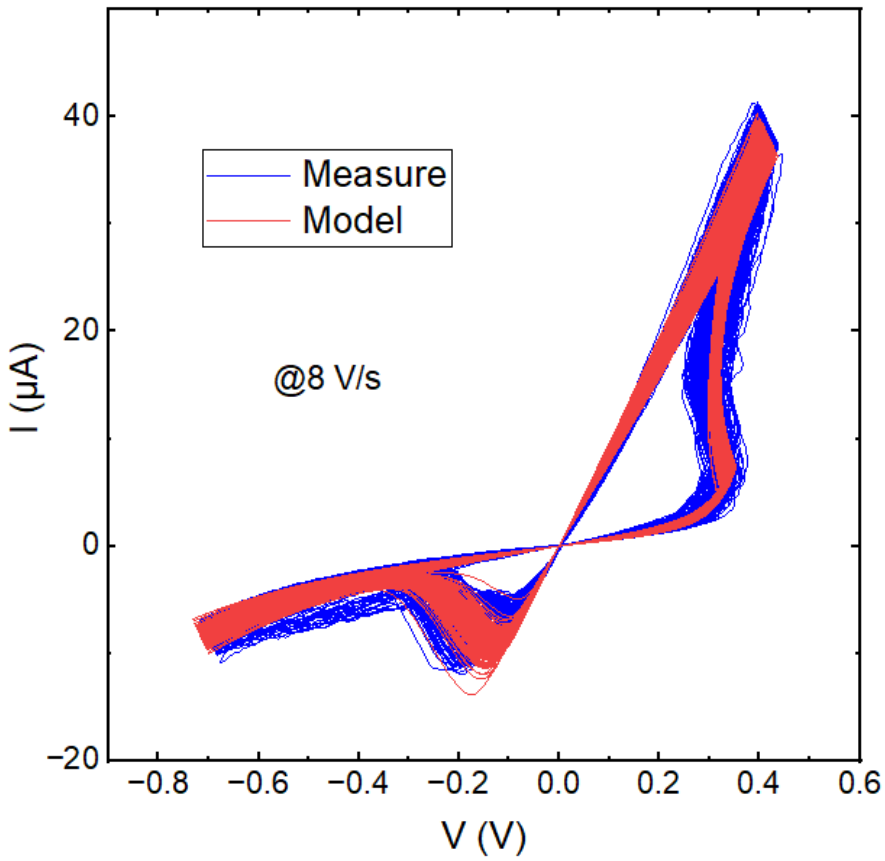


Figure 5.8: Genetic Algorithm with variability

5.2. Results

After using the genetic algorithm with variability, we obtain the final data for our model. The data shown in Table 5.2 allow us to perform a simulation very similar to the hysteresis curves obtained during the experimental phase with the commercial KNOWM memristor.

The model, using these parameters, is finally represented as shown in Figure 5.9.

Using the best parameters in this new model, we obtain the following curves shown in Figure 5.10 with 200 steps and variability. As can be seen, these curves accurately simulate the behavior of a real memristor. These data are shown in Figure 5.8, directly comparing them with the experimental curves obtained with the KNOWM memristor. The blue curves represent the actual measurements, while the red curves represent the model.

As mentioned before, due to the limited availability of physical memristors to simulate a complex neural network like MNIST, it is necessary to use a physical model that accurately emulates the behavior of a memristor. With the obtained data, it is considered that the resulting model is faithful enough to the commercial memristor used to evaluate these devices as a viable alternative in neural network creation.

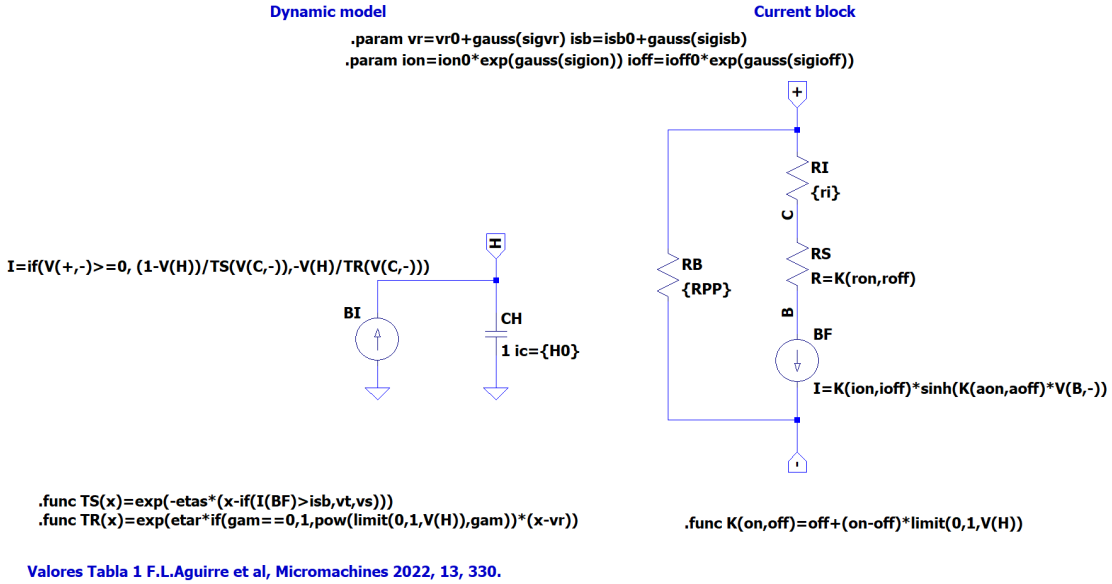


Figure 5.9: Memristor Physical Model with variability

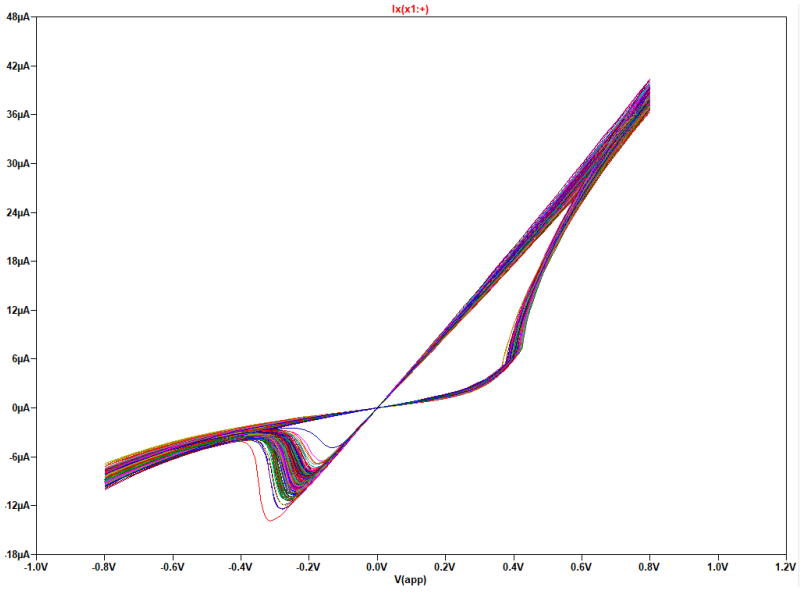


Figure 5.10: Memristor Physical Model Hysteresis

Chapter 6

Neural Network Implementation

We now have a suitable physical model for our memristors and the parameters that best fit our empirical data in simulations. Moving forward, we will no longer be constrained by the limited number of memristors on the board and their configuration.

We can start to design and test circuits by means of hardware simulation in LTSpice, using as many memristors as our processing power will allow and in whichever configuration we desire.

6.1. Inference in Neural Networks

We will design a circuit to perform neural network inference by means of analog computations. Before explaining the circuit design, we will do a brief explanation on how an inference step is calculated form a mathematical point of view.

6.1.1. Fully Connected Neural Network

A neural network is made up of an ordered sequence of layers, which are in turn composed of an array of neurons. Each layer of neurons is activated, one after the other, depending on the values of the output of the immediately previous layer. In a fully connected neural network, each and every one of it's layers is fully connected to the next one, meaning all neurons receive as inputs the outputs of every other neuron in the preceding layer. Each neuron activation value is calculated as follows:

$$z = \sum_j w_j x_j + b$$

Where x_j is the value of the output of the j^{th} neuron in the previous layer, w_j is the neuron's weight associated with the j^{th} output and b is the bias parameter of the neuron. If we have w and x as vectors, we need simply to calculate their dot product and then add b.

Afterwards, an activation function, such as sigmoid or relu, is applied to z to produce the final output that will be fed as an input to the next layer.

The calculations for each layer can be easily made in parallel, since the activation of one neuron doesn't depend in any way on the activation of other neurons within

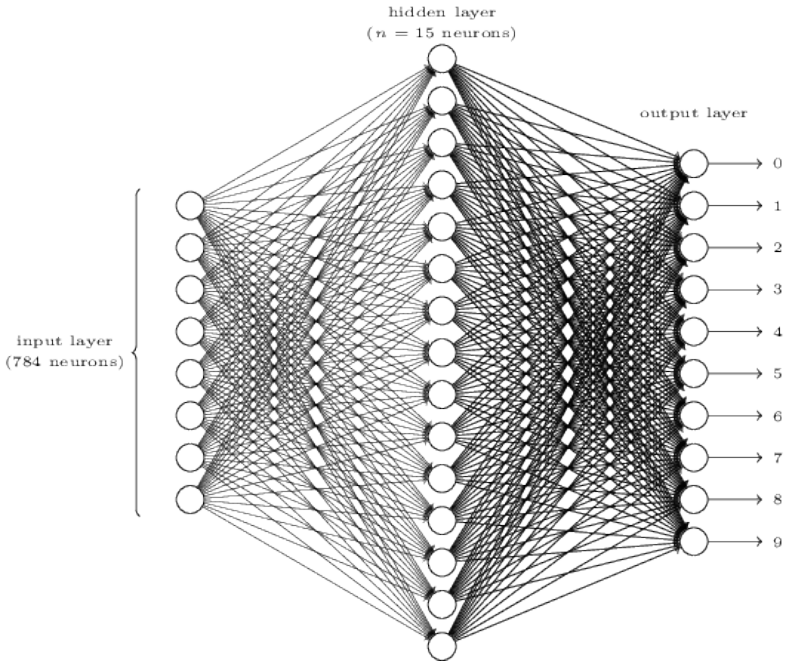


Figure 6.1: Neural Network illustration from Nielsen (2015)

the same layer. This way, instead of calculating dot products for each neuron, we perform a matrix multiplication as follows:

$$z = W_l * A_{l-1} + B_l$$

Where W_l is the matrix containing the weights of the neurons of layer l. Each row corresponds to a different neuron, and each column to the associated neuron of the previous layer. A_{l-1} is the vector of outputs from the previous layer, or the inputs if l is the first layer. Lastly, B_l is the vector of biases of the layer l each of them corresponds to a neuron.

The inference process is the calculation of the output of given an input. To do this, we only need to follow these steps for each layer in order, starting with $l = 1$:

1. Calculate $z = W_l * A_{l-1} + B_l$.
2. Apply the activation function to z to get A_l.
3. If l isn't the last layer, do $l + 1$.

6.1.2. Convolutional Neural Network

The other type of neural networks that we will implement are convolutional neural networks. These types of networks usually consist of a set of convolutional layers followed by some fully connected layers. They are widely used for image classification tasks, among many other things, because they retain the ability to generalize at a higher training efficiency.

The inference in these last fully connected layers works in the same way as previously described, so we will only focus on the convolutional layers in this section.

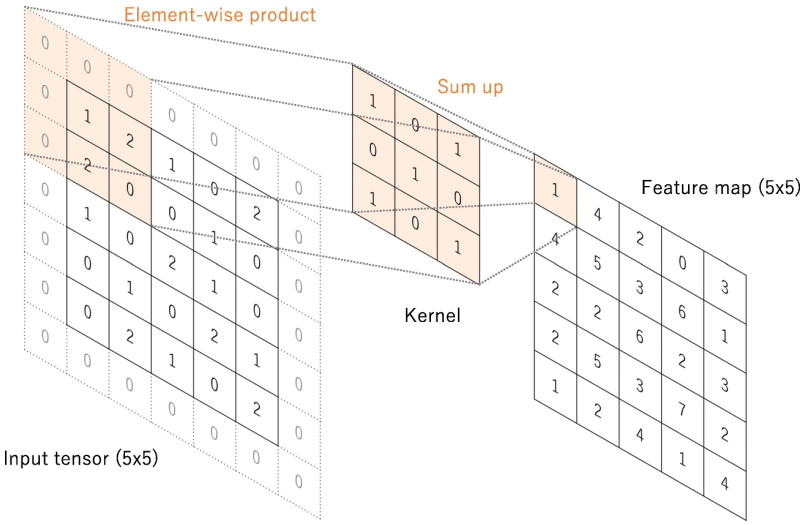


Figure 6.2: Convolutional layer illustration from Yamashita et al. (2018)

Convolutional layers use kernels, which are square matrices of a given size, typically 3x3 or 5x5. The activation value of a neuron is calculated only using a kernel sized window of neurons from the previous layer. The weights of the kernel are common to all neurons, which simplifies the structure. It is often explained as a sliding window, that multiplies with the input image element-wise, adds up all the terms and adds a bias.

Usually, more than one kernel is used, generating outputs with more than one channel, i.e., with more than one value for each position in the resulting feature map. This way, we have a third dimension, the channels. There will be as many kernels as $Channels_{in} * Channels_{out}$, meaning that for every channel of our feature map, there will be one kernel for each channel in the input; their resulting values will be added up.

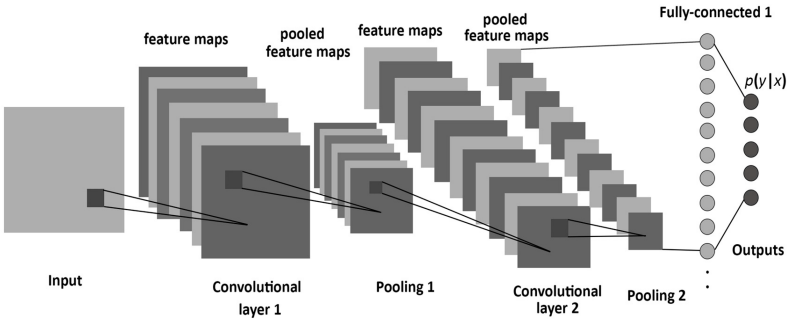


Figure 6.3: Convolutional network illustration from Albelwi y Mahmood (2017)

Unless some type of padding is used, the size of the output feature map will be smaller than that of the input. The reason is that values are not calculated for positions at the edges where the kernel window does not entirely fit.

To further reduce the size of the feature maps, techniques such as pooling or strides can be used. A pooling layer divides the feature map into windows, usually $2x2$, and reduces them to a single value, either by taking the maximum value of the

the window or by averaging them all. Strides allow us to achieve a similar result with a very small hit to quality, less complexity and less computations. The idea is that, instead of applying the kernels to every single possible position, we skip some in between.

Taking all of this into account, the resulting feature map will be a tensor made up of as much matrices as output channels and each of the matrices will have a size equal to the size input matrix minus the lost edges divided by the stride.

A forward pass in a convolutional layer is as follows:

Algorithm 1 Convolutional layer

1: $k \leftarrow size_{kernel} \div 2$
2: $(size_{fmap} \leftarrow size_{input} - k * 2) \div stride$ ▷ Size of the resulting feature maps
3: **for** $cout$ `in` $Channels_{out}$ **do**
4: **for** i `in` $size_{fmap}$ **do**
5: **for** j `in` $size_{fmap}$ **do**
6: **for** cin `in` $Channels_{in}$ **do**
7: **for** ki `in` $[-k, k]$ **do**
8: **for** kj `in` $[-k, k]$ **do** $H[cout][i][j] \leftarrow H[cout][i][j] + input[Cin, i*stride+ki+k, j*stride+kj+k]*kernel[Cout][Cin][ki+k][kj+k]$
9: **end for**
10: **end for**
11: **end for**
12: $H[cout][i][j] \leftarrow H[cout][i][j] + B1[cout]$ ▷ Adding the biases
13: **end for**
14: **end for**
15: **end for**

6.2. Hardware Design

Once we know what operations we will need to implement in our circuit, we can start with the design. We will use a crossbar structure with synapses of two memristors, similar to the one used in Hasan et al. (2017) and which can be seen in figure 6.4.

This structure is very widely used in analog computation, as it allows for very simple matrix multiplication. The inputs are the voltages in nodes A, B and C. By Ohm's law, the resulting current will be V/R or, using the conductance: $I = V * G$. All the currents that carry the value of an input multiplied by the conductance of the corresponding memristor are added up by virtue of being connected to the same cable.

6.2.1. Design of a Fully Connected Neural Network

Going back to the preceding explanation of inference, our inputs and activations of the previous layer are represented as the magnitude of the input voltages, which

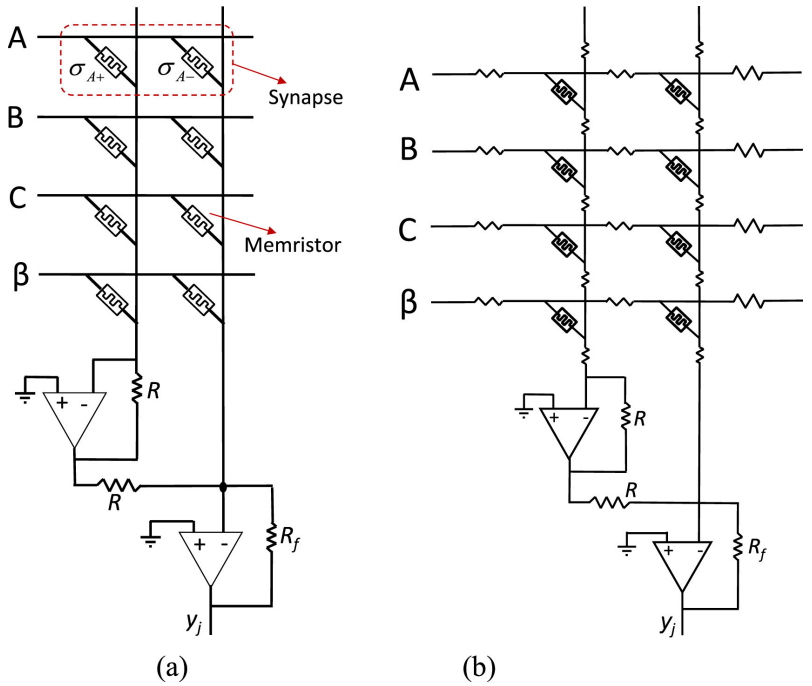


Figure 6.4: Memristor-based neuron circuit. A, B, C are the inputs and yj is the output. Figure from Hasan et al. (2017)

can be both positive or negative. Our weights are represented by the value of the conductance of the pair of memristors that form each synapse.

A neuron is a pair of columns connected to an operational amplifier. Its weights are its synapses, placed in the row corresponding to their associated input. In figure 6.4 only one neuron is depicted, adding more is as simple as replicating the structure, attaching more pairs of columns to the left.

We use the synapses to be able to represent negative weights: if the conductance of the first memristor $G1$ is higher than the one of the second $G2$, it represents a positive weight of value $G1 - G2$, otherwise, the weight will be negative. To achieve this, we need a sub-circuit that receives the currents from both columns and subtracts the value of second from the first one. In turn, this device, known as an operational amplifier, will give us the result in terms of a voltage, ready to be fed to the next layer of neurons or to be read as the final output. The magnitude of this voltage will be the resulting current multiplied by a resistance R_f.

The bias term of each neuron can be treated as a weight, meaning its value is determined by the conductance of a synapse. But, instead of receiving a variable input, it is always supplied a constant amount of voltage, making it independent of the input. Its addition to the dot products is carried out by being connected to the same cable.

Putting all this together, the output of a neuron, pending the activation function, will be the following:

$$z = R_f * ([\sum_j V_j * G1_j] + \beta * GB1 - ([\sum_j V_j * G2_j] + \beta * GB2))$$

or

$$z = R_f * ([\sum_j V_j * (G1_j - G2_j)] + \beta * (GB1 - GB2)$$

Where β is a constant, $GB1$ and $GB2$ are the conductances of the memristors that make up the synapse that represents the bias term. The same for each pair of $G1_j$ and $G2_j$, which are the weights.

We now have a way of calculating $z = W_l * A_{l-1} + B_l$. It's worth noting that the cross-bar structure doesn't perfectly map to the matrix W_l, since the weights of a neuron were the rows of W_l but are now the columns of the cross-bar. We are only missing a way of applying the activation function. There are electrical circuits than can effectively approximate these functions, but since it isn't necessarily relevant for our purpose, we chose to simulate then behaviorally instead of designing the circuits for them.

To have a complete circuit for a neural network with more than one layer, we simply concatenate another cross-bar structure that takes as inputs the voltage outputs of the last one, as in figure 6.5.

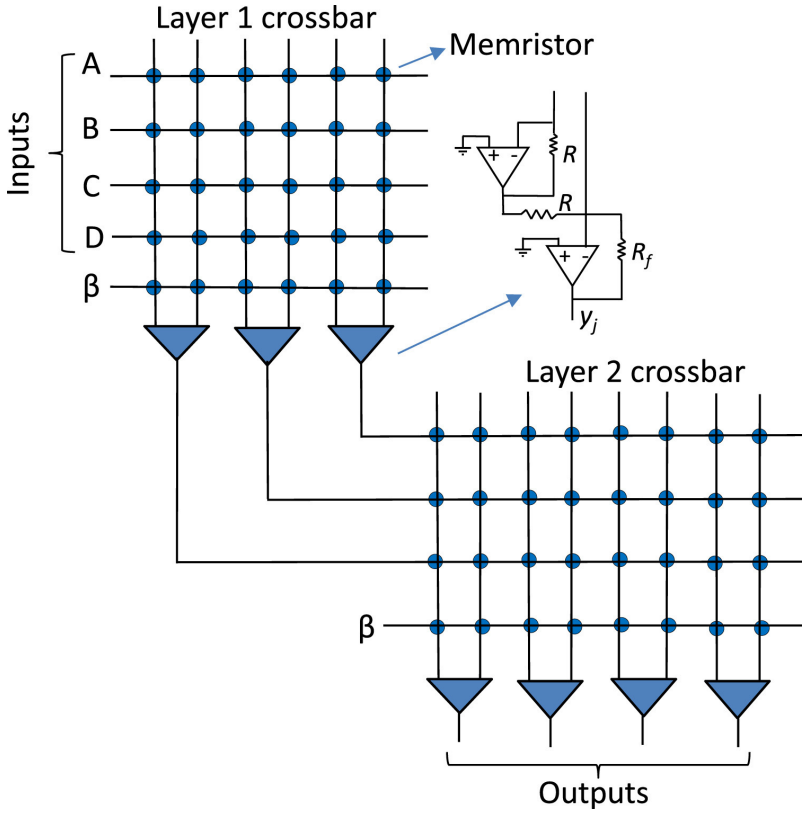


Figure 6.5: Multi-layer neural network. Figure from Hasan et al. (2017)

6.2.2. Design of a Convolutional Neural Network

Looking at algorithm 1 one would be correct in assuming that its circuit design won't be as straight forward as that of the fully connected network. There is hardly an easy way of visualizing it.

Regardless of that, the calculation can be parallelized not very differently than how we did for the fully connected layers. We won't need to use any other sub-circuit

or technique than the ones we already have, since we will only need to do addition and multiplication and we can use the same operational amplifier as before.

The difficulty of implementing this circuit lies in its interconnections. However, as we will explain in the following section, we are not designing the circuits manually, but rather using scripts, which will make this task feasible.

Some noteworthy differences are in the kernels, which contain the synapses representing the weights. Whereas in the fully connected layers each weight could be unique, the kernels and its synapses will need to be replicated for as many times as output values the output feature map contains. This is not a problem for us, since we are only interested in performing inference, but it is an important consideration for training.

6.2.3. Time and Power Consumption Estimations

We will now estimate the time of inference and the energy usage of a fully connected neural network with a hidden layer of 20 neurons.

For the time we are going to do a first order estimation treating the columns that represent each neuron as if they were in series with respect to the input. In reality they are neither in series nor in parallel, which complicates this analysis. The total time of the circuit will be determined by the result of $T = R * C$ where R is the total resistance of the interconnection cables, which we will assume to be 5Ω, and C is the total capacitance. To calculate the total capacitance, we must add the capacitance of all the individual memristors that are in parallel.

To do this, we have to look back at the circuit design. Since we are treating each neuron as parallel to each other, and they all have the same number of memristors, we only need to take in to account the total number of memristors for a single neuron. This will be the same as weights the neuron has, which in turn, corresponds to as many inputs as it processes. Each weight is composed of two memristors, but they are in series. Therefore, the amount of memristors in parallel in a given layer will be equal to the number of inputs it has. For a $16x16$ image, we have 256 memristors in parallel.

Now, the second layer will receive 20 inputs from the first layer, which means that it will have 20 memristors in parallel. Adding this up, we have a total of 276 of them in parallel. If we assume a typical capacitance of $20pF$ for each memristor, then $T = 5 * 276 * 20pF$ which gives us a result of $27,6ns$, or a frequency of $36,2MHz$.

For the power consumption, we measure the current in the operational amplifier, which has a value of $0.000754A$ and we multiply it by the voltage supplied $5V$, which gives a result of $3.77mW$ for each neuron. Since in this example we have 20 hidden neurons plus 10 output neurons, the total power consumption is $0.1131W$

6.3. LTSpice Implementation

The size and complexity of the circuits for the neural networks make it impractical if not impossible to draw. Instead, we will use Netlists, which allow us to describe a circuit and its interconnections using plain text and can be used to compile circuits

and do simulations in LTSpice.

In the Netlist, aside from the connectivity description, we can add simple components such as voltage sources or resistors, but for the rest, we will need to call sub-circuits. We only need two of them: one for the memristors and another one for the operational amplifiers.

We already have the circuit for the memristor, as seen in chapter 5 and we can easily extract a Netlist from it. We chose to use it coupled with the 10KΩ resistor with which our real memristors came with, since that is the configuration we have characterized. It receives as a parameter the value of H0 so that we can give it an initial state, i.e., set a given conductance.

For the operational amplifier, we built the circuit shown in figure 6.6, from which we extracted its Netlist. We also apply the activation function in here, reason why we have slight variations of this sub-circuit, differing only in this respect.

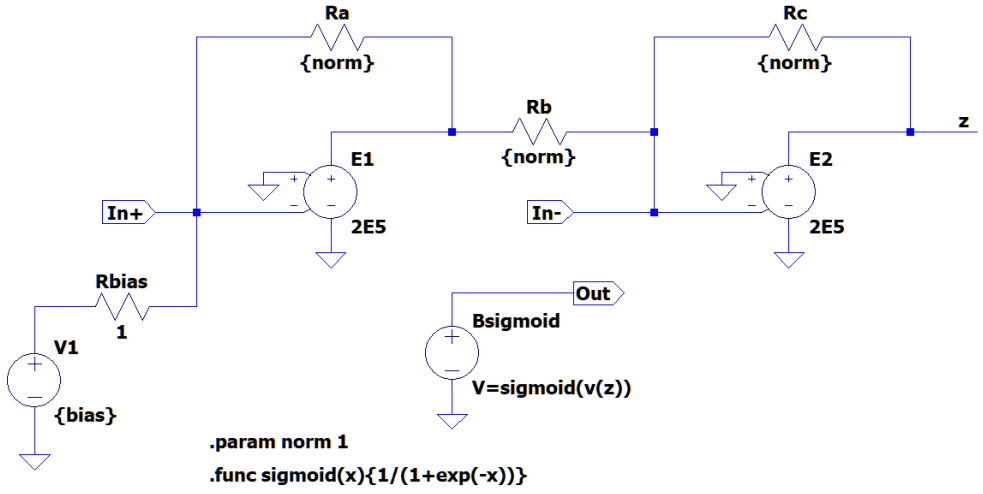


Figure 6.6: Operational Amplifier

As an example of how a Netlist describes a circuit, the following text is equivalent to figure 6.6:

.subckt neuron In+ In- Out PARAMs: norm=1 bias=0
E1 N001 0 0 In+ 2E5
Ra N001 In+ norm
E2 z 0 0 In- 2E5
Rc z In- norm
Rb N001 In- norm
Bsigmoid Out 0 V=sigmoid(v(z))
Rbias N002 In+ 1
V1 N002 0 bias
.func sigmoid(x)1/(1+exp(-x))
.backanno
.ends

It receives as parameters the value of bias, which we never used since we manage them as described in subsection 6.2.1, and the *norm* parameter, which is the value that will multiply the result of the addition; R_f in subsection 6.2.1.

To write the Netlist for the circuit of each neural network, we wrote script in Python that receives the following information as parameters:

1. The name of the file to write on.
2. The size of the image.
3. The amount of input channels (1 for B&W, 3 for RGB).
4. Number of output neurons.
5. The weights and biases of each layer, adapted to their corresponding H0 equivalents, see section 6.4.

With this information, the script calls functions to write each of the layers, depending on the type of network. We successfully implemented functions to be able to do a linear layer form both a preceding convolutional layer or another linear layer, a convolutional layer, and the operational amplifiers for both types of layers.

The functions that create the linear and convolutional layers aren't any more complicated than following the steps and algorithms detailed in section 6.1. Instead of having variables where we add up the products of our inputs and weights, we now simply use the same output cable where the currents add up. For example, a synapse representing a negative weight will be written like this:

Xwh39+ H627 H739+ memristor PARAMS: Hvalue=0

Xwh39- H627 H739- memristor PARAMS: Hvalue=1.959469e-06

Where $Xwh39+$ and $Xwh39-$ are the names of the two memristors that make up a synapse; $H627$ is the input cable for both. In the output of the first memristor, node $H739+$ we will get the current that results when multiplying the voltage in the input with the conductance of the memristor with $H0 = 0$, which will be very close to 0. The same goes for $H739-$, except this time we will have some current. For a positive synapse, the $H0$ of the second memristor would be 0 and the first one would be greater than 0; and if the weight happened to be 0, then they would both be 0.

For the biases, the input cable is always *Bias*, which has a fixed voltage.

The operational amplifiers work in a similar way, we only need to connect them to the corresponding inputs and outputs:

XNeuron3 H5+ H5- OUTPUT0 simple PARAMS: norm=1000 bias=0

Where $XNeuron3$ is the name, $H5+$ and $H5-$ are the two inputs carrying the currents that will be subtracted and the result will be multiplied by $norm = 1000$ and converted to voltage. The activation function in this case would be the identity since the type is *simple*.

With these building blocks, we can very easily implement different architectures that can differ in input channels, size of images, size of kernels, weights and biases, number and type of layers, output neurons and activation functions.

The Netlists we built for the circuits are, in fact, sub-circuits themselves. In section 6.6 we will explain the reasons, which have to do with the management of the inputs for the simulations.

6.4. Parameter Convertion

We now have the circuit designs and the means to build Netlists for them. We need only to acquire the appropriate weights and biases to be able to start the simulations.

Doing a short recapitulation, our memristor sub-circuits only take as an argument the $H0$, but we have defined the weights and biases of the memristor synapses as their conductivity, not their $H0$. Therefore, we need a way of making the conversion from conductivity to $H0$.

The range of values that $H0$ can take is between 0 and 1. To see how they relate to the conductivity, we run an LTSpice simulation on our memristor model that steps through 10.000 values of $H0$. We measure the conductance of the device at each point, while under a reading voltage of $0.1V$. The results can be seen in figure 6.7 where the variability is very noticeable.

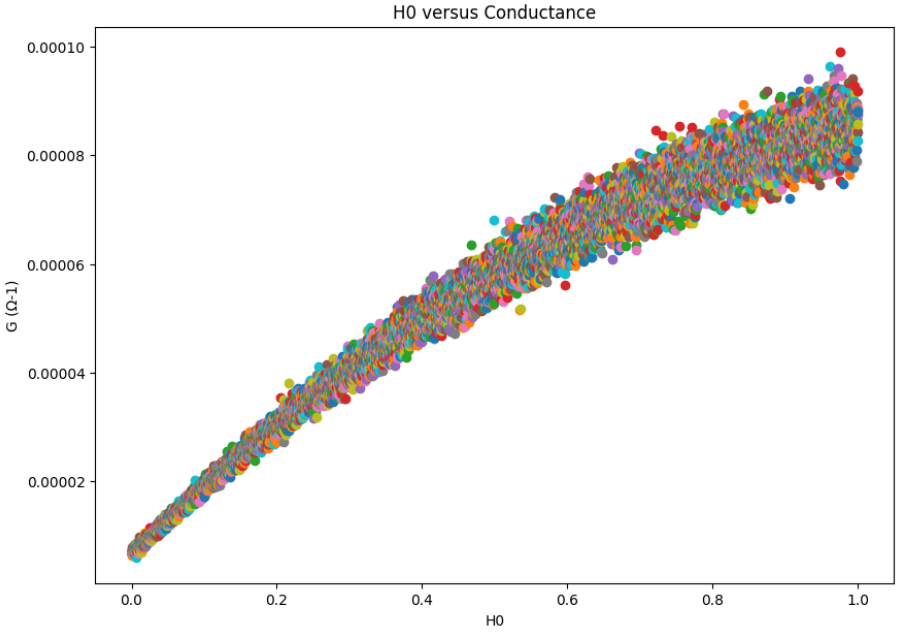


Figure 6.7: H0 (state variable of the device) versus the conductance

We chose to train a small neural network to approximate the value of $H0$ that should be set when trying to achieve a given conductance. The results are in figure 6.8. We will use this model to convert the weights we will obtain in the next section to be able to pass them as arguments to our Netlist builders.

From figure 6.7 we have also obtained the effective range for our weights and biases, which can't be any greater than $1e-4$. We chose to place the limit at $8e-5$, since that seemed like a good cut off point that, regardless of the variability, was

almost always reachable with $H0 = 1$. As we are using two memristor synapses and are able to represent negative values, our real range is $(-8e-5, 8e-5)$.

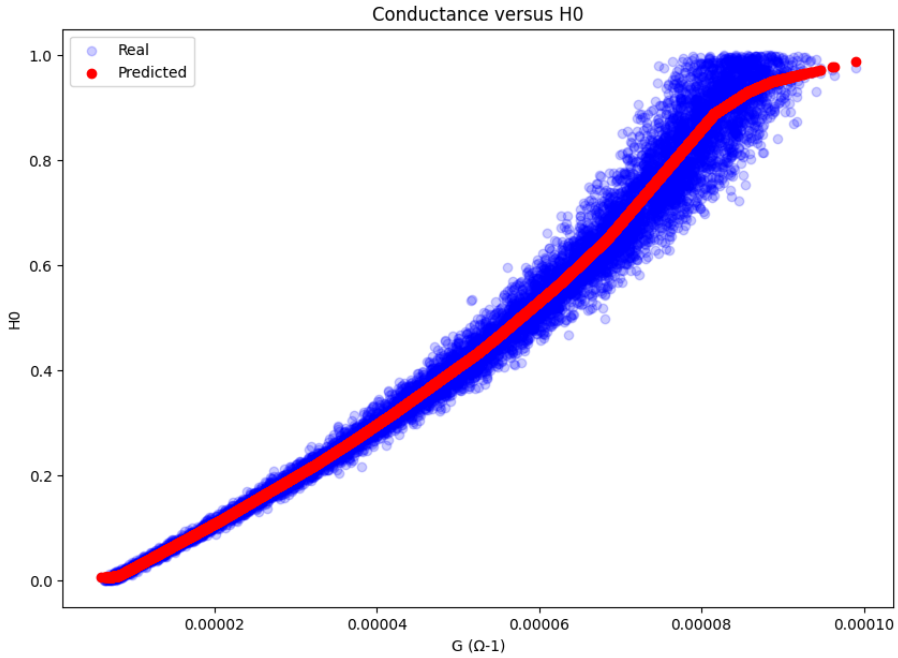


Figure 6.8: Results of the model trained to approximate H0 for a given conductance

6.5. Digital Training

Now that we know the restrictions for our inputs, weights and biases, we can try and obtain the parameters for the simulations of our circuits. For this purpose, we will use the PyTorch framework.

As previously mentioned, we will use the MNIST data set for our experiments, since it is a widely recognized and computationally light problem. Furthermore, we will be resizing the images form their $28x28$ original pixel size to $16x16$ and $12x12$. The reason behind this decision is the fact that the time it takes to simulate an inference step exhibits a quadratic growth in relation to the size of the input. We will use the high-quality LANCZOS interpolation method to reduce the quality penalty of the resize as much as possible.

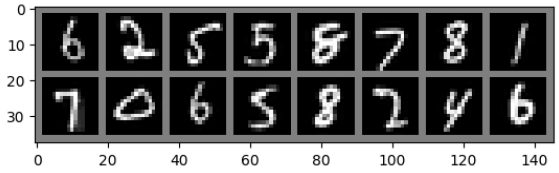


Figure 6.9: Scaled 16x16 MNIST examples

In the MNIST data set from the Torchvision package that we will be using, the images are in a gray scale with values from 0 to 1. These are the values that we

will feed as inputs to our circuits, and, as we saw in section 4.3, our memristors will change state under more than $0.25V$, which will alter the weights, impacting the performance of the network. To avoid this, we will apply a transformation to ensure that the inputs are contained within the safe interval from $-0.1V$ to $0.1V$.

The next step is to ensure the weights and biases are also constrained to the interval established in the previous section, $(-8e-5, 8e-5)$, since that is the available range for the conductance of our synapses. There is another important factor that further restricts the permissible interval of the biases, according to our physical design and as explained in subsection 6.2.1, the synapse that represents the bias is multiplied by a constant input voltage. Therefore, since that voltage has to be smaller than $0.25V$ to not introducing state changes, the real range of the biases will be further reduced. We chose $0.1V$ to simplify calculations, meaning our biases will have to be between $-8e-6$ and $8e-6$.

Since neural networks are not linear functions, we cannot simply train the network normally and then divide the value of the weights and biases to make them fit their intervals. This is because the activation functions are non-linear by design. Instead, we will attempt to train the networks directly with the restrictions.

This process required a great deal of trial and error, but we found a way of doing it without any loss in the performance of the network when compared to an equivalent one trained without restrictions. To do it, we had to follow these three steps:

1. Initialize the weights and biases with a normal distribution with mean 0 and standard deviation of $2e-5$ for the weights and $2e-6$ for the biases.

2. Use a weight and bias clipper after each gradient descent step.

3. Avoid the vanishing of the activations layer to layer.

The first point is necessary so that the starting value of the parameters meet our restrictions. The second point is to ensure that the weights and biases stay within the valid range. To achieve this, we define a simple function that goes through all the parameters of the network and, if any of its weights is larger than $8e-5$, it sets its value to be exactly that and vice versa with weights smaller than $-8e-5$. It does the same with the biases, but with their maximum and minimum values. We call this function after every step of the gradient descent, i.e., every time the parameters get updated. We can see the results in figure 6.10.

The third point is less straightforward, and it will depend on the activation function we use. Since our weights and biases are so small, every output of a layer, pending the activation function, is magnitudes smaller than its input was. For example, if we have a fully connected layer of 100 neurons a single input, the output will be, in the best case scenario, where all the weights are the maximum of $8e-5$:

$$input * 100 * 8e-5 = 8e-3 * input$$

In a more realistic scenario, we would have hundreds of inputs and they would also be both positive and negative, the same as the weights, which would further cancel each other out.

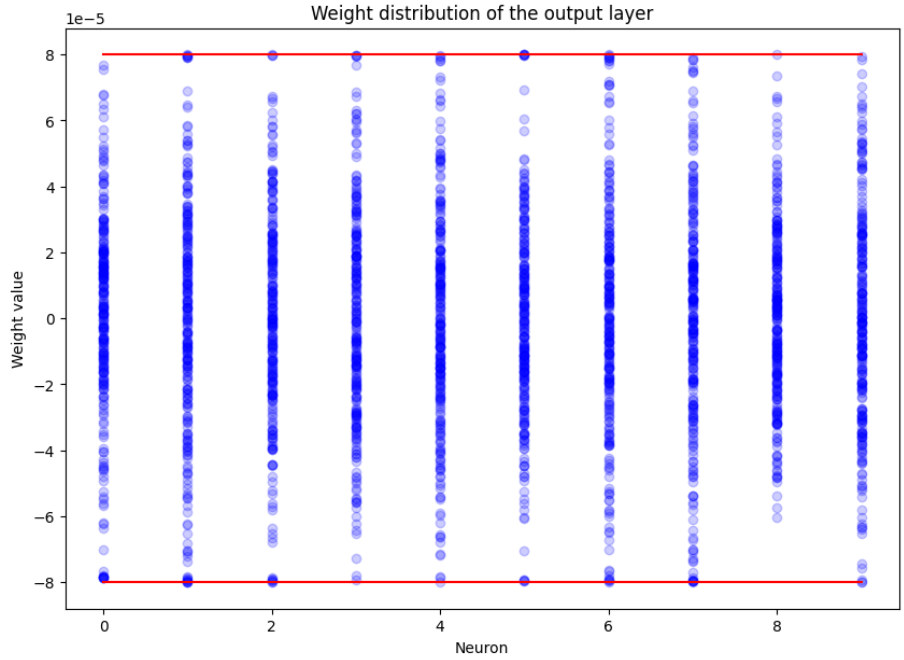


Figure 6.10: Weight distribution of the output layer of a neural network trained with parameter range restrictions

If we are using RELU as the activation function, the values will continue to get smaller and smaller with each layer, which causes a problem similar to the vanishing gradient problem, and the network struggles to learn anything. Sigmoid presents a different set of problems that we will tackle later.

To solve this, we used inspiration from our physical design. In subsection 6.2.1 we explained how the operational amplifier multiplied the resulting current by the value of a resistor R_f. We will use the same concept here, multiplying by a fixed value the output of every layer. We will then use the same value for R_f in the Netlist builder to ensure that the voltages don't vanish either, so we will have to make sure that it does not produce voltages outside the interval $(-0.1V, 0.25V)$ or else it would change the state of the memristors in the following layer.

Finding the perfect value for this purpose is complicated, since we can have very different configurations of weights, biases and inputs. We found through experimentation that $1e4$ seemed to work well in most cases, particularly in fully connected layers. In convolutional layers, it sometimes produced outputs outside the desired intervals; we will explain our solution for that in section 6.6.

We were also able to implement sigmoid activation functions, and it's worth explaining the challenges it presented. Firstly, the output of a sigmoid is in the interval $(0, 1)$ and $sigmoid(0) = 0.5$ which is well above our allowed values. For this reason, we need to change the sigmoid for a $sigmoid/10$, making the output $(0, 0.1)$. Another new problem with the sigmoid function is that, if the inputs are very close to 0, it will behave like a linear function. To avoid this, we do something similar to what we did before; we multiply the outputs before the activation function by a constant. For sigmoids, $1e5$ worked the best.

Having found a way to successfully train neural networks that meet the restrictions imposed by our physical design, we trained the following configurations to later simulate and compare results:

1. Simple fully connected network, no hidden layers, for input sizes $12x12$ and $16x16$.

2. Fully connected network with a 20 neuron hidden layer, for input sizes $12x12$ and $16x16$. Using RELU as the activation function.

3. Simple convolutional network with a single convolutional layer, with kernel 3, stride 2, and a fully connected output layer, for input sizes $12x12$ and $16x16$. Using RELU as the activation function.

4. Convolutional network with two convolutional layers, with kernel 3, stride 2, and a fully connected output layer, for input sizes $12x12$ and $16x16$. Using RELU as the activation function.

5. Fully connected network with a 20 neuron hidden layer, for input sizes $12x12$. Using Sigmoid as the activation function.

6.6. Simulation

Once we have the Netlist builders for our circuits, the parameters for the neural networks and a way of converting them to H0 values. We can start the simulations.

Our goal was to do inference on the first 1000 images from the test set, both for the digital network and for the simulated circuit. For the same reasons explained in section 5.1, we could not parallelize all simulations, one input at a time, because that way the variability parameters always take the same values. But we still wanted to parallelize as much as possible because the simulations take a considerable amount of time.

With $32GB$ of memory, we can do up to 10 simulations at a time of the biggest circuits among the ones tested, which speeds up the total execution time by around a factor of $5X$.

To be able to have both the parallelization and the steps through the variability parameters of the physical model, we settled on doing 10 threats, each of which will process 10 different inputs using steps. This way, a batch size of 100 images is divided into sets of 10, which will be processed by different simulations. We do 10 steps instead of 100 because for some reason unknown to us, beyond 20 or so steps, it starts to take considerably longer per step.

This is the reason why the Netlists of our circuits are sub-circuits, they need to receive the input as a parameter that will change with every step up to 10 times.

The function that performs the simulation does the following:

1. Calls the appropriate Netlist builder for the circuit, which in turn writes to the file specified the design of the neural network.

2. Includes the sub-circuits needed, i.e, the memristor, the operational amplifiers and the circuit just built.

3. Creates an instance of the neural network, giving it variables as the inputs.

4. Steps through the input variables with the values of the different input images.

An example of 4 would be the following:

.step param Vx list 1 2 3 4 5 6 7 8 9 10

.param paramin000 table(Vx, 1, -0.1, 2, -0.1, 3, -0.1, 4, -0.1, 5, -0.1, 6, -0.1, 7, -0.1, 8, -0.1, 9, -0.1, 10, -0.1)

In this case, since it's a pixel on the corner of the image, it will always take the same value -0.1 which is black.

6.7. Results and Variability

We can now compare how our hardware implementations perform against their digital counterparts on inference precision. We include the results in accuracy in table 6.1. Where the column *Ratio* is the ratio of the simulation accuracy by the digital accuracy and the *type* column follows the following name conventions:

- FC Simple: a fully connected network without hidden layers.
- FC Double: a fully connected network with a 20 neuron hidden layer. Using RELU.
- CV Simple: one convolutional layer followed by the output layer. Kernel size 3, stride 2, output channels 3. Using RELU.
- CV Double: two convolutional layers followed by the output layer. Kernel size 3, stride 2, output channels: first layer 3, second layer 6. Using RELU.
- FC Double Sigmoid: a fully connected network with a 20 neuron hidden layer. Using Sigmoid.

The results are very promising, always around a few percentage points of the digital network. In some cases, even surpassing the original; an unexpected behavior for which we might be able to offer a reasonable explanation.

The mistakes the simulated analog circuit makes are mostly due to the variability it was purposely built with. This variability will not affect the results for most inputs, i.e., the maximum value will correspond to the same output neuron even though its magnitude may vary. It will make mistakes where the digital network won't if the input makes the two output neurons with the highest activation values have very close values to each other.

On the other hand, when the digital network makes a mistake, the activation of an incorrect output neuron will have been the largest, but it is reasonable to assume that, if it is a high-quality neural network, the value of the output neuron with the correct answer was a close second. Now, given that our physical model has

Type	Input size	Digital Acc	Simulation Acc	Ratio
FC Simple	12x12	90.2	89.3	0.990
FC Simple	16x16	91.5	91.7	1.002
FC Double	12x12	91.9	90.3	0.982
FC Double	16x16	93.6	92.8	0.991
CV Simple	12x12	91.3	89.2	0.976
CV Simple	16x16	94.7	93.0	0.982
CV Double	12x12	91.5	88.0	0.961
CV Double	16x16	94.7	93.2	0.984
FC Double Sigmoid	12x12	86.9	90.0	1.035

Table 6.1: Accuracy results comparison between the digital network and the analog simulation

been built with an intrinsic variability by design, if we have a very close activation value on two outputs, the maximum of them might change with the variability. This would make the physical model sometimes able to make the right prediction where its digital counterpart will always fail.

To illustrate this phenomenon, we can select an example where the digital failed and the analog simulation succeeded,and see how the variability is responsible for this effect. to do this, we step 20 times for the same input. The results are in figure 6.11

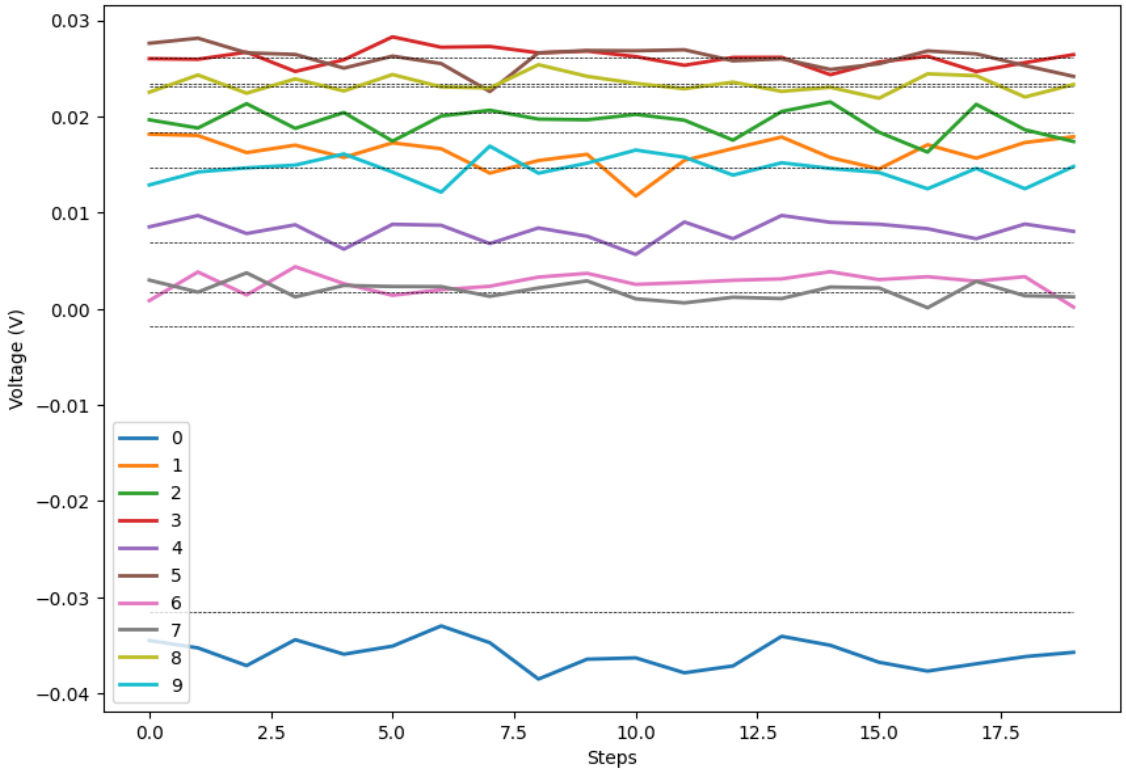


Figure 6.11: Values of the output neurons. Colored lines represent the outputs of the analog simulation, dashed lines the outputs of the digital neural network.

The right answer was 3, but the digital predicted 5. We can clearly see that the maximum between the neurons that represent 3 and 5 are constantly changing with the variability parameters.

Looking at the overall results, it is reasonable to assume that, in most cases, variability will lead to errors rather than the other way around.

Another interesting observation is that the performance of physical circuits improves relatively as the input size increases. In table 6.1 we can see that, for every architecture with two input size variants, the larger one has a slighter higher ratio. We would require more data to be certain, but this seems to suggest that the wider the network, the lesser the negative effects of the variability.

The depth of the network does not appear to affect the comparisons in either way, but again, more tests would be required. This would be an interesting finding, since it would mean we can scale these implementations with no considerable penalty to the performance.

Chapter 7

Conclusions and Future Work

We have successfully designed and simulated analog circuits based on memristors to perform inference on neural networks. For the simulations, we have used a state of the art physical model of the memristor that we were able to adjust to very reasonably match the behavior and empirical measurements of our real commercial device. We also managed to identify and overcome the restrictions our circuit design imposed over the parameters of the neural networks.

The results obtained are very satisfactory, as we have demonstrated that these analog circuits can obtain similar levels of performance.

It appears that there is limited literature available on neural networks built with Knowm devices that take into account variability. It's also noted that while some researchers Florini et al. (2022) use the same devices but with different models and learning rules, they tend to use Spike-Timing-Dependent Plasticity (STDP) networks rather than convolutional networks. This could be due to the inherent properties of Knowm devices that may be more compatible with the principles of STDP, which is a biological process. However, the application of Knowm devices in convolutional networks could be an interesting area for future research, given the potential of these devices in mimicking synaptic behavior.

The part of this work pertaining to memristor modeling has been published in the 14th Spanish Conference on Electron Devices (CDE 2023) Jiménez-Gallo et al. (2023)

Personal Contributions

Pablo Alex Lázaro

In the early days, my task was to familiarize myself with neural networks. At the beginning of the final year project, I hadn't taken any courses related to this topic yet, and my knowledge was limited to what I had been learning on my own out of personal interest. Therefore, it was a necessary step.

Once I understood the basic concepts of a neural network, I started researching memristors by referring to various documentation provided by my supervisor, Guillermo. After grasping the concept and understanding how they work in broad terms, we were provided with the necessary hardware to further explore memristors. Over the following weeks, together with my partner Ignacio, we explored the different options offered by the Memristor Discovery software and conducted the experiments mentioned in chapter 4.

Meanwhile, every Friday, we had meetings with our project supervisors to report the progress we were making. During these meetings, our supervisors, Guillermo and Francisco, provided detailed explanations about the behavior of memristors, which gave us a better understanding of the steps to follow or the results obtained.

After completing the experimentation phase together, we proceeded with modifying the software used for experimentation to create our own simple neural network using the available memristors with Memristor Discovery.

Next, I collaborated with Ignacio to extract the necessary data for characterization from the application and made the required adaptations to the software to enable the execution of 20 pulses.

We get ready the data needed for the device characterization as explained in Section 4.2. Also, we need to create a Python script to convert the data to be positive and be separated with commas to work with them easily.

Then I create the memristor figures that appears in this doccument using the data extracted and adapted as explain before, with the software Origin.

The next stage involved creating the physical model in LTspice. During this time, we had several meetings with our project supervisors, who explained how LTspice worked and how the created model behaved.

I participate in the creation of the genetic algorithms used for result adjustment. I worked in pararelle with Ignacio to devolve the genetic algorithm, finally getting

the Ignacio‘s one because its performance. Also I research alternatives and possible changes to them, which ultimately were not implemented, because the great results obtained.

Regarding the creation of neural networks, I contributed to adapting the script that created the desired neural network in LTspice. I also, investigated ways to optimize the created networks to minimize execution times, which initially were quite costly.

While researching on the operation and implementation of the developed neural networks, I took a more supportive stance in this phase regarding the creation of neural networks. However, I followed their development constantly, and I helped with the difficulties that arose.

I also performed parallel executions of the neural networks on my device, but due to their extensive duration and the limitations of my laptop, they couldn't be completed on time. Therefore, only the data obtained by Ignacio was used since they were executed considerably faster on his machine.

Finally, the development of this report was a joint effort between Ignacio and me. Although we divided the sections of the report between us, we reviewed each other's work, so it cannot be said that any section was written by only one person.

In summary, the project has been developed collaboratively for most of the time, so as you can see, the work has been primarily collaborative, with only a few parts carried out individually.

Ignacio Jiménez Gallo

In the early days, I dedicated myself to familiarizing myself with the term memristor by investigating both its behavior and its application in neural networks.

Once we received the logic analyzer and the memristors, we began the experimental phase. During the first weeks of work, we focused on exploring the different options provided by the Memristor Discovery software. Together with Pablo, we performed the calibration and all the experiments explained in Chapter 4.

In collaboration with my partner, we extracted the necessary data from the DC experiment and adapted the Memristor Discovery code to obtain 20-pulse runs in order to expedite the extraction and processing of the data.

After this initial period of adaptation to this technology, we started the device characterization phase. In this phase, using the results obtained from the DC experiment, together with Pablo we prepared the code for the device characterization as explained in Section 4.2. I also created a Python script to obtain the values in the required format since we needed the data to be positive and separated by commas. Furthermore, I wrote the script we used to interpolate the values of the current at the reading voltages we were interested in.

Once the characterization was completed, we began the physical model design phase. I searched for ways to integrate Spice simulations directly from python, and experimented with them until I figured out how to use it for our purposes.

In parallel with Pablo, we worked on designing a genetic algorithm that could adapt the parameters used in the model as faithfully as possible. I worked out the

way to calculate the loss function of the simulated curves after much trial and error. Ultimately, we ended up using the algorithm I created as it yielded the best results. I then took care of adding variability to the genetic algorithm I had created and parallelized the execution of individuals in LTspice to speed up the process, which meant more researching from my part.

Afterwards, we proceeded with the development of neural networks using the created model. Together, we created a script capable of building the circuit we wanted instead of creating it manually, which would have been impossible. My partner worked on designing the Netlist builders of the different neural networks that we planned to test.

I was in charge of obtaining the weights and biases to be used in the simulations. For this purpose, I trained many digital neural networks with different architectures. I had to find a way around the limited range of values our weights, biases and inputs had to be, which was a very experimental process, since there isn't much literature on neural network training with this sort of restrictions.

For the last part, since I have the faster machine with the most amount of RAM, I run all the final simulations. This was a tedious process since every different architecture took a considerable amount of time. To mitigate this a little, I found ways to parallelize and reduce the precision of the simulations and tested whether it affected the results.

Finally, this report was written by both of us with revisions from each other.

In conclusion, this work required linear development, so we worked collaboratively almost all the time and there were not many opportunities to parallelize processes.

Bibliography

Albelwi, S. y Mahmood, A. A framework for designing the architectures of deep convolutional neural networks. *Entropy*, vol. 19(6), 2017. ISSN 1099-4300.

Bishop, C. M. *Pattern recognition and machine learning*. Springer, 2006.

Botella, G., García, A., Rodríguez-Álvarez, M., Ros, E., Meyer-Baese, U. y Molina, M. C. Robust bioinspired architecture for optical-flow computation. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 18(4), páginas 616–629, 2009.

Chua, L. Memristor-the missing circuit element. *IEEE Transactions on Circuit Theory*, vol. 18(5), páginas 507–519, 1971.

F.L. Aguirre, E. M., J. Suñé. Spice implementation of the dynamic memdiode model for bipolar resistive switching devices. *Micromachines*, vol. 13, página 14, 2022.

Florini, D., Gandolfi, D., Mapelli, J., Benatti, L., Pavan, P. y Puglisi, F. M. A hybrid cmos-memristor spiking neural network supporting multiple learning rules. *IEEE Transactions on Neural Networks and Learning Systems*, páginas 1–1, 2022. Epub ahead of print.

Goodfellow, I., Bengio, Y. y Courville, A. Deep learning. *MIT Press*, 2016.

Hasan, R., Taha, T. M. y Yakopcic, C. On-chip training of memristor crossbar based multi-layer neural networks. *Microelectronics Journal*, vol. 66, páginas 31–40, 2017. ISSN 0026-2692.

Haykin, S. S. Neural networks: a comprehensive foundation. *Prentice Hall*, 1999.

Hu, M., Graves, C. E., Li, C., Li, Y., Ge, N., Montgomery, E., Davila, N., Jiang, H., Williams, R. S., Yang, J. J., Xia, Q. y Strachan, J. P. Memristor-based analog computation and neural network classification with a dot product engine. *Advanced Materials*, páginas 1705914–, 2018.

Indiveri, G. y Liu, S.-C. Memory and information processing in neuromorphic systems. *Proceedings of the IEEE*, vol. 103(8), páginas 1379–1397, 2015.

Jiménez-Gallo, I., Alex-Lázaro, P., Botella, G., del Barrio, A., Maldonado, D., Roldán, J. y Jiménez-Molinos, F. Characterization and modeling of variability in commercial self-directed channel memristors. *IEEE Proceedings on 15th Spanish Conference on Electron Devices (CDE)*, 2023.

Maldonado, D., Aldana, S., González, M., Jiménez-Molinos, F., Campabadal, F. y Roldán, J. Parameter extraction techniques for the analysis and modeling of resistive memories. *Microelectronic Engineering*, vol. 265, página 111876, 2022. ISSN 0167-9317.

Moreno, D. G., Del Barrio, A. A., Botella, G. y Hasler, J. A cluster of fpaas to recognize images using neural networks. *IEEE Transactions on Circuits and Systems II: Express Briefs*, vol. 68(11), páginas 3391–3395, 2021.

Nielsen, M. A. *Neural Networks and Deep Learning*. Determination Press, 2015.

Pershin, Y. V. y Di Ventra, M. Memory effects in complex materials and nanoscale systems. *Advances in Physics*, vol. 60(2), páginas 145–227, 2011.

Roldán, J. B. Variability in resistive memories. página 33, 2023.

Strukov, D. B., Snider, G. S., Stewart, D. R. y Williams, R. S. The missing memristor found. *Nature*, vol. 453(7191), páginas 80–83, 2008.

Yamashita, R., Nishio, M., Do, R. K. G. y Togashi, K. Convolutional neural networks: an overview and application in radiology. *Insights into Imaging*, vol. 9(4), páginas 611–629, 2018. ISSN 1869-4101.

Yang, J. J. y Strukov, D. B. The science and applications of resistive random access memory. *Annual Review of Materials Research*, vol. 43, páginas 195–224, 2013.

Yang, X., Taylor, B., Wu, A., Chen, Y. y Chua, L. O. Research progress on memristor: From synapses to computing systems. *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 69(5), páginas 1845–1857, 2022.