university of groningen / faculty of science and engineering

Designing a Hopfield Neural Network Using Memristors

Bachelor's Project Thesis

Jory Klaverstijn, s3764826, j.klaverstijn@student.rug.nl,
Supervisors: Dr. J.P. Borst & T.F. Tiotto, MSc

Abstract: In this paper it is explained how Nb-doped $SrTiO_3$ memristors could be integrated into classical Hopfield neural networks (HNN), and how a working simulated model could be designed. The performance of this model was tested and compared to the performance of a linearised version, non-memristor version and a modern continuous HNN. The performances of the models are measured on the MNIST data-set by classifying the converged patterns with a secondary feed forward network. From the experiment, it can be concluded that a memristor HNN model can have an adequate accuracy. The accuracy of the (linearised) memristor HNN ranged from 63.6% to 89.4%, dependent on the amount of noise added to the images. It is shown that the linearised version of the memristor based HNN performs slightly better and that the memristor based HNNs perform worse than the continuous HNN model for higher magnitudes of noise.

1 Introduction

Hopfield networks are a classic example of a neural network. In this paper, the classical Hopfield networks, as described by John Hopfield, are going to be explained. More modern versions of this type of neural network are going to be touched upon as well, including a type of Hopfield network able to accept continuous input patterns. A relatively new type of electrical circuit component, called a "memristor", is going to be discussed after that. A memristor, or memristive device, is a kind of electrical component, of which the resistance can be modified by applying voltage pulses. This innovative device may allow for advances in the field of neuromorphic computing. In this project it is going to be investigated how memristors might be integrated into Hopfield networks.

1.1 Classical Hopfield network

"Neurons that fire together, wire together" [1] has become a familiar phrase in the field of neuropsychology. After research by Little in 1974 [2], a researcher named John Hopfield used this idea to create a model of associative memory [3], later known as the Hopfield neural network (HNN). A classical HNN is a single layer recurrent neural network able to store multiple binary patterns, which can later be individually retrieved by presenting only part of the pattern.

The first notable attribute of a HNN, is that the single layer of neurons is both the input and the output of the network. A classical HNN typically has symmetric weights and does not have any self connections. Eq. 1.1 shows a simple example of weight calculations for a HNN.

$$W_{ij} = W_{ji} = \begin{cases} \sum_{s=1}^{N} (2V_i^s - 1)(2V_j^s - 1) & \text{if i} \neq \text{j} \\ 0 & \text{if i} = \text{j} \end{cases} \tag{1.1}$$

Where W_{ij} is the weight from node i to j, W_{ji} is the weight from node j to i and N is the number of patterns to be stored. $\mathbf{V}^s$ is a specific binary pattern vector that is stored inside the network representing pattern s, and V_i^s is the activation level (0/1) of node i in pattern s. This vector could be interpreted as a set of neurons either firing (=1) or not firing (=0).

Eq. 1.1 is used to calculate the final weights according to the "neurons that fire together, wire together" principle. This can be seen in Fig. 1.1, which shows an example of some weight calculations of a HNN.

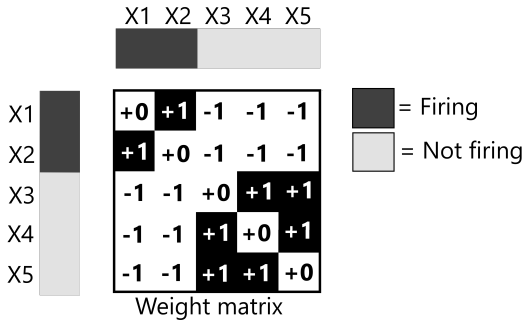


Figure 1.1: An example showing how the weights are updated. In this example the weights are updated with a given binary pattern of size 5: [1, 1, 0, 0, 0].

The storing of the patterns does not necessarily have to be done using a set formula, but can also be achieved by using online neuron learning methods. One could for example use Hebbian learning [1] or Oja learning [4] to obtain weights that allow the user to, for example, restore images [5].

When one wants to converge from a query pattern to a stored pattern, the start value of every neuron unit of the network is set to the value of the query pattern. The units can then be updated one unit at a time using Eq. 1.2.

$$s_i = \begin{cases} 1 & \text{if} \sum_{j=0}^{m} W_{ji} s_j \geq 0 \\ 0 & \text{Otherwise} \end{cases} \tag{1.2}$$

Where s_i is the chosen unit to be updated, m is the total number of of neurons in the HNN and $\sum_{j=0}^{m} W_{ji} s_j$ is the weighted sum of the activation values of other neurons afferent to node s_i. The update rule for converging from a query pattern to a stored pattern/attractor can be done sequentially, changing one bit at a time, or in parallel, where multiple bits get changed at each update iteration, as is tested in [6].

1.2 Modern and continuous Hopfield networks

An important problem of the classical HNN is that the storage capacity is linear with respect to the number of pixels per pattern/ dimensions (around 0.14 patterns per extra dimension can be stored [7]). This problem was largely solved by the development of modern Hopfield networks by Krotov and Hopfield [8], also known as dense associative memories. The way classical Hopfield networks converged, was by considering an energy for each state of the network, as is shown in Eq. 1.3 [3].

$$E = -\frac{1}{2} \sum_{i=0}^{m} \sum_{j=0, j \neq i}^{m} W_{ij} s_i s_j \tag{1.3}$$

When using the update rule given by Eq. 1.2, this energy will monotonically decrease for each update. The energy landscape of a trained classical Hopfield network has local minima to which the query pattern will converge when updated using Eq. 1.2. To increase the storage capacity, a new energy function was introduced by Krotov and Hopfield [8]:

$$E = -\sum_{\mu}^{N} F(\xi_i^{\mu} \sigma_i) \tag{1.4}$$

Where N is again the number of patterns stored, ξ denotes a stored pattern, σ denotes the current configuration of the neurons (either 0 or 1), and F(x) is some smooth function (summation over index i is assumed). This is further extended by Demircigil and colleagues [9], where the function exp(x) is chosen for F(x). This new energy function could be rewritten to:

$$E = exp(lse(1, \mathbf{X}^T \cdot \boldsymbol{\xi})) \tag{1.5}$$

Where $\mathbf{X}$ is a matrix of stored pattern column vectors concatenated to each other $\{\mathbf{x_1}, \mathbf{x_2}, ..., \mathbf{x_n}\}$, and lse is the log-sum-exp function:

$$lse(\beta, \mathbf{z}) = \frac{1}{\beta} log \left(\sum_{i=1}^{d} exp(\beta \cdot z_i) \right) \tag{1.6}$$

Where d is the number of dimensions (or rows in 2D) of $\mathbf{z}$ and β is a chosen hyper parameter. β can be seen as a temperature parameter: When a lower β value is used, the resulting pattern will be more likely to be an average of multiple patterns close to the query pattern, whereas a higher β value will cause the HNN to converge to a single pattern closest to the query pattern.

This energy function, and a found update rule that minimises this energy function for every step, allows the network to have a storage capacity of around $2^{d/2}$, which is far better than a linear storage capacity with regards to dimensions as is the case with the classical Hopfield network.

Another problem of the classical HNN, but also the modern HNN, is that they only accept binary/discrete patterns. A HNN that could accept continuous patterns would therefore be useful to,

for example, restore non-binary images [10, 11]. A recent design for a continuous HNN has been published in the paper "Hopfield Networks is All You Need" [12]. In this paper a new energy function and update rule is described, having many parallels to the attention mechanisms described in "Attention is All You Need" [13]. These attention mechanisms were originally used for language translation tasks, but seem to have useful properties for HNN purposes. The energy function from [12] is as follows:

$$E = -lse\left(\beta, \mathbf{X}^T \cdot \boldsymbol{\xi}\right) + \frac{1}{\beta} logN + \frac{1}{2}(M^2 + \boldsymbol{\xi}^T \cdot \boldsymbol{\xi}) \tag{1.7}$$

Where N is the number of stored continuous patterns and M is the largest norm of all patterns.

This energy function is dependent on the state pattern $\boldsymbol{\xi}$ and has two components: a concave term and a convex term. A method was found to derive an update rule that could minimise a concave-convex function such as this energy function [14], which results in the following update rule [12]:

$$\boldsymbol{\xi}^{new} = \boldsymbol{X}\text{softmax}\left(\beta \boldsymbol{X}^T \boldsymbol{\xi}\right) \tag{1.8}$$

This update rule allows the energy of the network to always decrease with every update and keep the exponential storage capacity and often single update convergence, similar to using the energy function supplied by Demircigil and colleagues [9] shown in Eq. 1.5. A visualisation of this algorithm is shown in Fig. 1.2.

In Fig. 1.3 an example is shown for the stored patterns, query pattern and converged pattern of a continuous HNN, using different values for β. In the example, 8 RGB images are stored inside the network. One of the 8 images is taken and distorted using Gaussian noise. The continuous network is initialised with this distorted image and Eq. 1.8 is used to restore the image to its original state. Higher β values could be used to converge to a single image, whereas lower β values could be used to converge to a prototype of an image type.

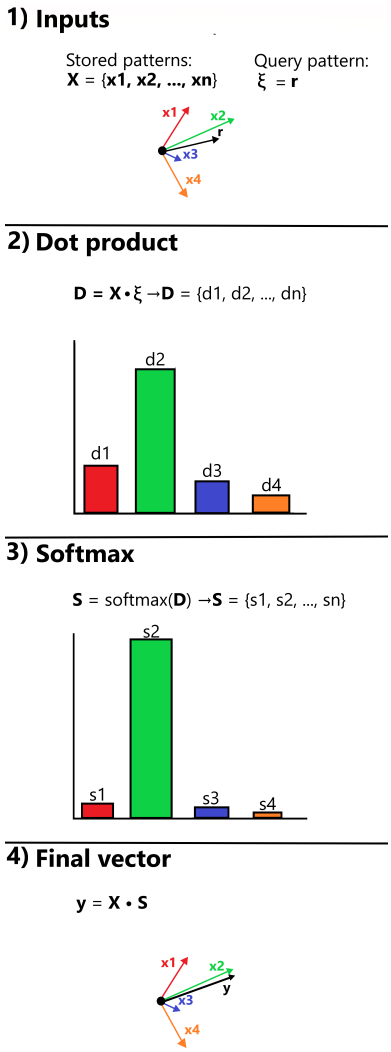


Figure 1.2: First the input pattern matrix $\mathbf{X}$ is created, and the network state pattern $\boldsymbol{\xi}$ is initialised to the query pattern $\mathbf{r}$. The second step is to calculate the dot product of $\mathbf{X}$ with $\boldsymbol{\xi}$. The third step is to calculate the softmax of the result of step 2. Finally, the converged pattern $\mathbf{y}$ is then the dot product of $\mathbf{X}$ with the result of step 3.

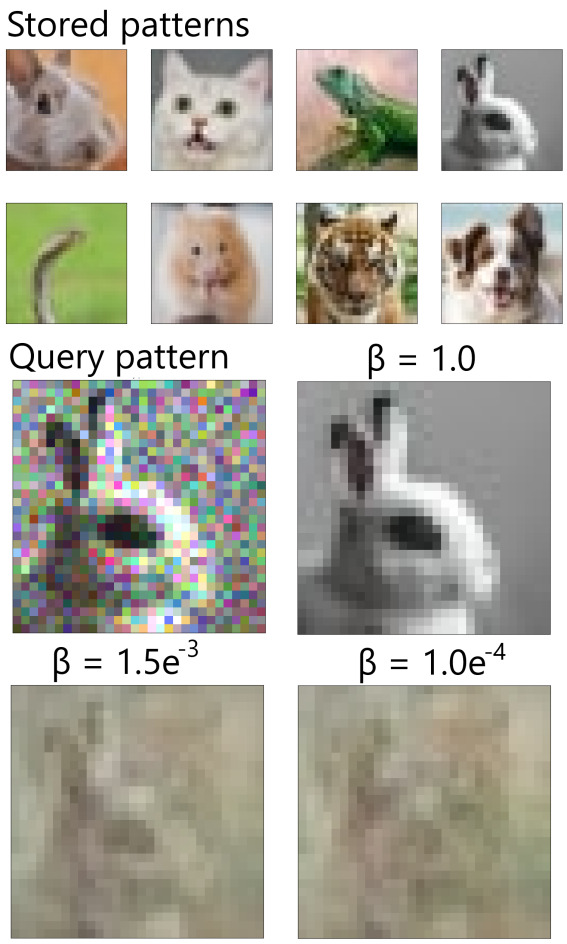


Figure 1.3: Example of how images converge using a continuous HNN with different values for β. Images are taken from [15] and compressed to 30x30 pixels.

1.3 Memristors

In 2008, members of an HP lab realised an electrical component [16] whose measured properties could be explained by the memristor theory of Chua, published in 1971 [17]. The resistance of a memristor can be altered in a continuous way by applying analogue voltage pulses [18]. Changing the resistance of such a memristive device, modifies the voltage drop across the device according to Ohm's law [19]:

$$R_{memristor} = \frac{V}{I} \tag{1.9}$$

If such a device were to be used in an artificial neural network, its resistance state could represent the weight between the neurons and, moreover, directly implement changes in this without any external control. One could configure a larger set of memristors in such a way that a dot product between a vector (input voltage pulses) and a weight matrix (resistance values of a set of memristors) produces an output vector (output currents). Dot products are important in artificial neural networks, because they are used in the forward pass of feed-forward networks (FFNs) and can also be used in the convergence process of a HNN. This configuration has already been realised and is called a "hybrid CMOS-memristor crossbar array", which could heavily speed up computation times of dot products [20, 21]. The memristors could be premodified with some learning algorithm, and could later be used to represent weights, allowing for efficient computation of dot products [22]. This architecture for a HNN using memristors has already been used to demonstrate some useful applications such as object recognition [23, 24].

1.4 $SrTiO_3$ Memristors

An example memristor is the Niobium-doped strontium titanate (Nb-doped $SrTiO_3$) memristor. The applicability of this memristor type has been described in [25]. The memristor has a maximum and a minimum resistance, and the decrease of the resistance after repeated application of positive voltages is not linear, but exponential [26]. These $SrTiO_3$ memristors are well-behaved at room temperature and do not require electroforming before being used [26, 27]. The base mechanism of this memristor, and other memristors, is "resistive switching". This is a phenomenon where the resistance of the material changes under the influence of a sufficiently high magnitude voltage pulse. This process is reversible, which means the memristor can be re-used after resistance has already been modified. The process is also non-volatile, which means that even when no current flows through it, the resistive state remains the same. This could allow for energy efficient applications such as a memory device, as only the reading and writing costs energy [28]. A voltage can be applied to the memristor in forward or reverse bias. For this project, the most important detail is that a positive voltage pulse (forward bias) decreases the resistance of the memristor, whereas a negative voltage pulse (reverse bias) increases the resistance of the memristor. More details about the behaviour of these memristors is given in the method section

of this paper (Section 2).

1.5 Aim of this paper

Hopfield networks are shown to be a useful tool in restoring images [5] and classifying objects [24], and because memristors provide a way to efficiently calculate dot-products, which are very useful in neural network applications, it would be useful to find out how we could combine memristors and Hopfield networks.

This paper aims to design a Hopfield network using simulated spiking neurons and memristors to represent the weights of the recurrent connections of the HNN. The main research question of this paper is: *"Is it possible to design and simulate an accurate classical Hopfield network that utilises Nb-doped $SrTiO_3$ memristors?"*.

Considering the benefits of a modern HNN, or even a continous HNN, one might wonder why a classical HNN is chosen. According to John Hopfield and Dimitri Krotov, the continuous HNN described in [12] would be biologically implausible [29] as it would require more than two neurons participating in a synaptic connection. Because this project focuses on simulating the HNN using neurons, the more logical choice would then be a classical HNN.

The HNN will be modeled and simulated using a Python package called "Nengo" [30], which is a framework used for simulating large scale spiking neural networks. The memristors simulated in this paper are Nb-doped $SrTiO_3$ memristors, which are the same as the memristors used in [26].

First the performance of the network will be analysed, with the standard memristor resistance decrease behaviour. Next the natural exponential behaviour of the memristors are compared to linearised behaviour. Third, the network will be compared to a similar model that does not utilise memristors. Finally the network will be compared to a continuous HNN similar to the one described in [12]. The networks will be trained on the MNIST data-set [31].

Section 2 will very generally explain how the Nb-doped $SrTiO_3$ memristors behave, how the performance of a HNN is evaluated, and will show the experimental setups. Section 3 will show the results of the experiments. Section 4 will explain the meaning and relevance of the results. Finally section 5 will give a conclusion to this research.

2 Method

2.1 $SrTiO_3$ Memristors

As stated in Section 1, the type of memristors used for this project are Nb-doped $SrTiO_3$ memristors. The way these memristors can be created, is explained in [27]. In that same paper an experiment has been performed to study the behaviour of these specific memristors. Voltage pulses of +0.1V with a duration of 1 second were applied to the memristor after which the resistance was measured. The resistance decrease of the memristor from this experiment can be seen in Fig. 2.1. The decline of the resistance of a memristor over the pulses can be described using a formula of the following form:

$$R(n, V) = R_0 + R_1 \cdot n^{a+bV} \tag{2.1}$$

Here, $R(n, V)$ is the resistance value of the memristor after n voltage pulses with a voltage magnitude of V, R_0 is the lowest resistance value the memristor can reach, and $R_0 + R_1$ is the highest possible resistance value of the memristor. "a" gives the decline of the memristor's resistance independent of the voltage, whereas "b" signifies the decrease of the memristor's resistance dependent on the voltage magnitude of the pulse. The values for these constants where found in [26] by applying multiple voltage pulses to the memristor with different magnitudes. This gives rise to the following estimate formula for the resistance of the $SrTiO_3$ memristors:

$$R(n, V) = 200 + 2.3 \cdot 10^8 \cdot n^{-0.093-0.53V} \tag{2.2}$$

As can be seen in Eq. 2.1, the voltage depends on the number of pulses already applied to the memristor. The number of voltage pulses already applied to the memristor can be calculated with the following formula from [26]:

$$n = \left(\frac{R(n, V) - R_0}{R_1}\right)^{\frac{1}{a+bV}} \tag{2.3}$$

Where $R(n, V)$ is the current voltage and R_0, R_1, a and b are all the same value as in Eq. 2.2, which gives the final formula for calculating the pulse number:

$$n = \left(\frac{R(n, V) - 200}{2.3 \cdot 10^8}\right)^{\frac{1}{-0.093-0.53V}} \tag{2.4}$$

The simulated decrease of the resistance of memristors, based on these equations, for multiple different voltage pulse magnitudes, can be seen in Fig. 2.1. As mentioned before, this figure also shows the measured values of the resistance after each pulse from [26].

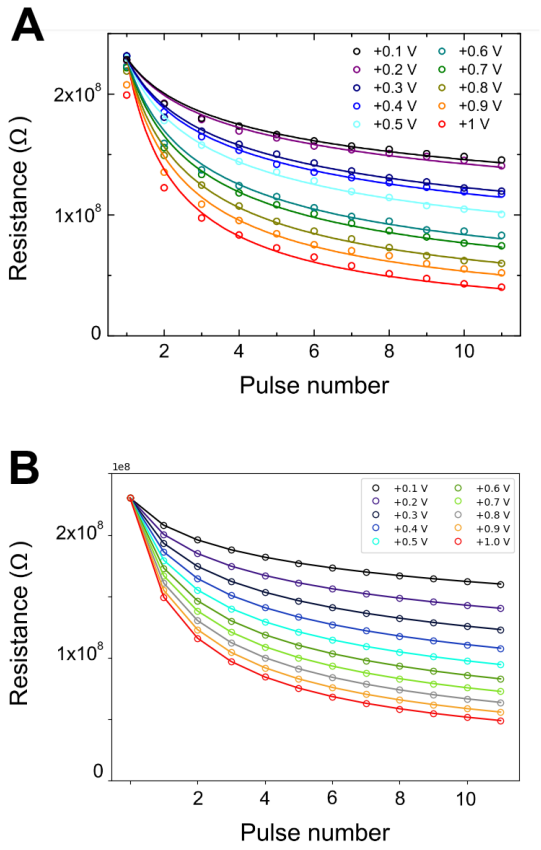


Figure 2.1: Decrease of resistance over voltage pulses for multiple voltage magnitudes. Figure A shows the resistance measured after each pulse from the paper by Tiotto and colleagues [26]. Figure B shows the decrease of the resistance for a simulated memristor using Eq. 2.2.

One can gain more control over the memristors when it is known what voltage is necessary to bring the memristor to a desired resistance. It is possible to iteratively find the voltage necessary to decrease the current resistance of a memristor to a desired resistance using a method called "Halley's method", which is explained in appendix A. This allows the user to control the decrease of the memristor to, for example, linearise the decrease of the memristance over time. Linearising would be useful here, because linear decrease makes it so that the order of presenting the pattern does not matter as much anymore, because each pattern will have an about equal effect on the weights.

Because multiple different magnitudes of voltage pulses can be applied to the memristor, the idea of the pulse number "n" has to be generalised. The pulse number is needed when one wants to calculate the effect of a single pulse on the memristor's resistance. It should be viewed as the number of pulses of a single voltage necessary to get from the starting resistance to the current resistance.

It is also important to note that in our simulations, at every step, the resistance of the memristor can be measured without modifying the actual resistance value. This can be done in practice as well by applying a "read" voltage pulse that has a magnitude low enough to not modify the resistance of the memristor. The resistance can then be calculated with the use of Ohm's law, and the measured currents through the memristor.

To stay true to this concept of a threshold below which voltage does not affect the memristor, a threshold of 0.1 V inclusive has been arbitrarily chosen to be the lowest voltage at which the resistance of the memristor will still be modified according to Eq. 2.2. This limits the control of the memristor at higher resistances as we can not modify the resistance with high precision anymore, which makes the simulation more realistic.

2.2 Hopfield network performance

There are multiple ways to measure the performance of a Hopfield network. For this project, the MNIST data set [31] is used as training and testing data. This data set consists of around 60,000 gray-scale images of handwritten digits. The images are compressed from 28x28 pixels (px) to 14x14 px by taking the average value of every 2x2 px square. Each pixel of every 14x14 px image is rounded up to be a white pixel (1.0) if its gray-scale value is above 60/255, otherwise it becomes black (0.0). Here 60/255 is arbitrarily chosen to make the images still look like digits. These processed images are then stored in the HNN network. A visualisation of this process is shown in Fig. 2.2.

The function of the network is to take a digit image with a percentage of the bits flipped, and make

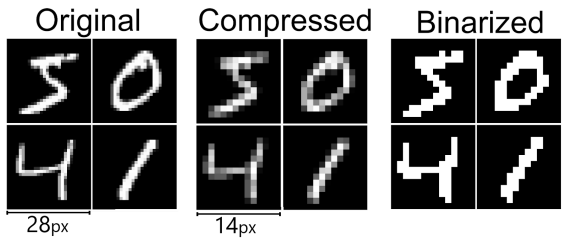


Figure 2.2: The processing of the MNIST images. The images are first compressed, and then binarised.

it converge to a digit without noise that is classified with the same label as the given digit. Because of some storage capacity limitations of the classical HNN that were discussed in Section 1, only a subset of digit images are stored in the HNN at a time. After storage of 10 random images from {1, 2, 3, 5, 8, 10} different labels into the network, a random processed input digit image is retrieved with the same label as a stored image, from which {0, 5, 10} percent of the bits are flipped. The noisy image is then presented to the network and the network will converge to a final image using a dot product and a threshold.

To evaluate the performance of the network on its task of converging the given noisy digit image to a correct representation of the digit, another type of network will be used. A feed-forward network (FFN) will first be trained on the MNIST data set, and can then be used to classify the converged digit image. This FFN is trained on 60,000 processed images and has an accuracy of around 97.5%.

The FFN classifies the converged image and gives a value between 0 and 1 for each label, based on its confidence that it is the correct label. If the FFN classifies the converged image to have the same label as the provided label of the input image, it means the HNN has converged to an image close to the original digit, and thus has a good performance. The performance of the HNN is then the accuracy, given by the proportion of correctly classified digit images.

For this project, the model will be tested for 100 iterations. For each iteration, the network is trained only once using a set of digit images. After the network is trained, in each iteration, 100 query images will be presented to the network and they will converge using the network's weights. The accuracy of the iteration is then the proportion of correctly classified converged images. The final accuracy of the model is then the average of the accuracy of each iteration. The standard deviation of iteration accuracies will also be calculated.

2.3 Neuron based HNN

As is stated in Section 1, the HNN's are simulated in Nengo [30]. With this framework, multiple neurons can be simulated over time. To start making a Nengo model for this project, an input node is created with 196 dimensions. Nodes provide non-neural input to other objects in Nengo, and these will be used to represent the input patterns/ digit images. An input node of 196 dimensions will be necessary, because there are 14x14 (=196) pixels in each image. Each node dimension will provide a constant signal to a single neuron, thus there are also 196 neurons. This signal provided by the input node will have a value that is either -2 for a black pixel or 2 for a white pixel. The neurons are used to temporarily represent the pixels. While a neuron is fed a signal of 2, it will keep spiking. While the neuron is fed a signal of -2 it will not spike at all. Because there are multiple images that must be stored in the network, the value of the input node will change over time, and thus so will the neuron spiking. All the images to be stored inside the network are shuffled, and will be shown in cycles of one second each, for 10 cycles. An example of a single cycle of three input nodes feeding pixel information of three images to the set of neurons is shown in Fig. 2.3.

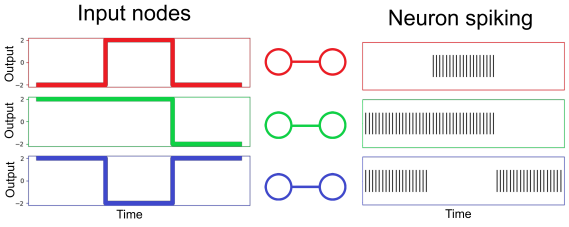


Figure 2.3: The input nodes' values and the neuron spiking as result of the input nodes feeding their signal to the neurons. The images represented by the input nodes from top to bottom are as follows: [0, 1, 1], [1, 1, 0], and [0, 0, 1].

Now that there are neurons that fire when they represent a white pixel or do not fire when they represent a black pixel, a recurrent connection must be added for the HNN to function. Every neuron

in the ensemble is connected to every other neuron in the ensemble and to itself. To learn the weight values that can represent the set of images that must be stored in the HNN, some kind of learning rule must be used. The connections between neurons are represented by the conductance values of a set of 196^2 positive memristors and 196^2 negative memristors. Each weight is represented by a single positive-negative memristor pair. Because only forward bias is used to influence a memristor's resistance, two memristors are used, so the weight can be increased or decreased by pulsing the positive or negative memristor respectively. How the weights are calculated, is shown in Eq. 2.5:

$$W = \gamma \left[\left(\frac{\frac{1}{R_{pos}} - \frac{1}{R0}}{\frac{1}{R1} - \frac{1}{R0}} \right) - \left(\frac{\frac{1}{R_{neg}} - \frac{1}{R0}}{\frac{1}{R1} - \frac{1}{R0}} \right) \right] \quad (2.5)$$

Where R_{pos} and R_{neg} are the current resistance value of the positive memristor and negative memristors respectively, $R0$ and $R1$ are the minimum and maximum possible resistance values of the memristor, and γ is the gain. The resistance of the positive and negative memristors are initialised to be 10^8 Ω.

An often used learning rule is Hebbian learning [1]. The weight change is simply the outer product of the post- and pre-synaptic activity:

$$\Delta \mathbf{W} = \mathbf{y}\mathbf{x}^T \quad (2.6)$$

Where $\mathbf{y}$ is the post-synaptic activity vector, $\mathbf{x}$ is the pre-synaptic activity vector and $\mathbf{W}$ is the weight matrix.

An improved version of the Hebbian learning rule, is the Oja learning rule [4], which is the learning rule used in this project. This learning rule is very similar to Hebbian learning, but adds a "forgetting" component, which ensures that the weights converge to a stable state. The formula for Oja learning is as follows:

$$\Delta \mathbf{W} = \mathbf{y}\mathbf{x}^T - \zeta \mathbf{W}\mathbf{y}\mathbf{y} \quad (2.7)$$

Where ζ is a parameter that controls the amount of forgetting.

The weights are represented by memristor pairs, which means they can not simply be set to a desired value. Therefore a matrix must be obtained that allows us to modify the memristors such that they converge to some desired weight values. This is done by creating a matrix that consists of voltage values that must be applied to the memristor pairs. The Oja learning rule using memristors will be called mOja from here on out.

From Eq. 2.5 it can be concluded that the weight value of a positive-negative memristor pair is increased when the positive memristor resistance is decreased. It can also be concluded from Eq. 2.5 that the weight is decreased when the resistance of the negative memristor is decreased. Using this information, a voltage matrix can be created:

$$\mathbf{V} = sgn(\Delta \mathbf{W}) \cdot 0.1 \quad (2.8)$$

Where $\mathbf{V}$ is the matrix holding the voltage values that must be applied to the memristors, and sgn(x) is a function that takes the sign of the given parameter. This matrix represents the voltage pulses that must be applied to the positive-negative memristor pairs. A value of 0.1 means that a voltage pulse of +0.1 V must be applied to the positive memristor of the corresponding weight. A value of -0.1 means that a voltage of +0.1 V must be applied to the negative memristor of the corresponding weight. After these voltages are applied to the correct memristors, the weights will be increased or decreased, approximating the Oja learning rule.

2.4 Experiment: linearised memristors

In this experiment the decrease of the memristors' resistances will be linearised to try and negate the bias of patterns shown earlier in the training cycle, because the resistance of a memristor decreases more when its resistance is still high. The linearisation of the memristor resistance decrease can be realised by storing a desired resistance for each memristor, and each time when the memristor would receive a voltage pulse, this desired resistance value would be decreased by a set amount. To get the memristor to this desired resistance value, the calculations for the required voltage in appendix A could be used. Because this method for calculating the correct voltage requires a lot of computations, which takes very long to run, a simplified method will be used to linearise the memristors decrease of resistance (see Fig. 2.5).

In Fig. 2.5, pseudo-code is shown for linearising the memristor resistance decrease. Boolean *pulse* is whether normally a pulse would be applied to the memristor. *R_cur* is the current resistance of the memristor, *R0* is the minimum resistance, *R1* is the maximum resistance and *R_des* is the desired

resistance. a, b are -0.093 and -0.53 respectively, and V is the magnitude of the voltage.

To start this simplified method, a desired resistance value for the memristor will be initialised to be the current resistance of the memristor. At each time step, the memristor's resistance value will be compared to the desired resistance value, and only when the memristor resistance is above the desired resistance, it will receive a pulse of +0.1V. Finally, at each pulse, the desired resistance is decreased by a constant c. This simple rule controls the decrease of the memristor resistance value, such that it approximates a linear decrease as can be seen in Fig. 2.4. Because the minimum magnitude of a voltage pulse is +0.1V, using Halley's method (see appendix A) with high resistance values will not give any different results to the simplified method shown in Fig. 2.5. When the resistance is lower, Halley's method will allow for better linearisation, as higher magnitudes could be applied to decrease the resistance by a constant amount each time step.

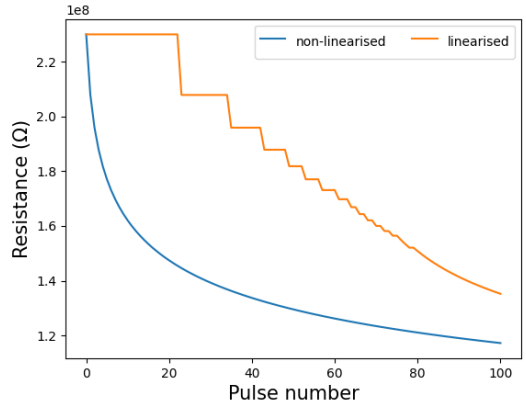


Figure 2.4: Example of standard memristor resistance decrease versus linearised resistance decrease when applying +0.1 V pulses.

2.5 Experiment: Non-memristor Oja learning rule

The Nengo package [30] already provides standard learning rules which can be easily implemented into a Nengo model. In this project a network is proposed that uses mOja learning [4], where weights are represented by positive-negative memristor pairs. The performance of this network can be compared to the performance of a network using Oja learning (without the use of memristors). This non-memristor Oja network is using the standard Oja implementation of Nengo.

Algorithm 2.1 Linearising memristor

Result: updated resistance of memristor
if *pulse* **then**
  $n \leftarrow calc\_n(R_{cur}, R0, R1, a, b)$
  $R\_des \leftarrow R\_des - c$
  $R\_next \leftarrow calc\_R(n+1, R0, R1, a, b)$
  if $R\_next > R\_des$ **then**
    $R\_cur = R\_next$
  end
end

Figure 2.5: Pseudo-code for linearising memristor. The *calc_R* function is given by Eq. 2.1, and *calc_n* is given by Eq. 2.3.

2.6 Experiment: Continuous HNN

In Section 1, a new type of continuous HNN is discussed from [12]. The performance of this network will also be tested on the MNIST data set. A small modification will be made to the update rule of the continuous HNN to increase the performance. After some testing with the original continuous HNN update rule, the network seemed to often converge to an image with a large norm. To compensate for this, the original update rule will be modified to the following:

$$\boldsymbol{\xi}^{new} = \boldsymbol{X}\text{softmax}\left(\mathbf{N} \odot (\beta \boldsymbol{X}^T \cdot \boldsymbol{\xi})\right) \tag{2.9}$$

Where $\mathbf{X}$ is a matrix of stored patterns $\{\mathbf{x_1}, \mathbf{x_2}, ..., \mathbf{x_n}\}$, $\mathbf{N}$ is a vector of the inverse norms of all stored patterns $\{\frac{1}{\|\mathbf{x_1}\|}\frac{1}{\|\mathbf{x_2}\|}, ..., \frac{1}{\|\mathbf{x_n}\|}\}$, $\boldsymbol{\xi}$ is the current state of the network, and β is a hyper-parameter that determines the interpolation strength between stored patterns. $\odot$ is an element-wise product. This new update rule is more similar to comparing the cosine similarities between the input pattern and the stored patterns, whereas the original equation (Eq. 1.8) compares the dot product between the input pattern and stored patterns.

3 Results

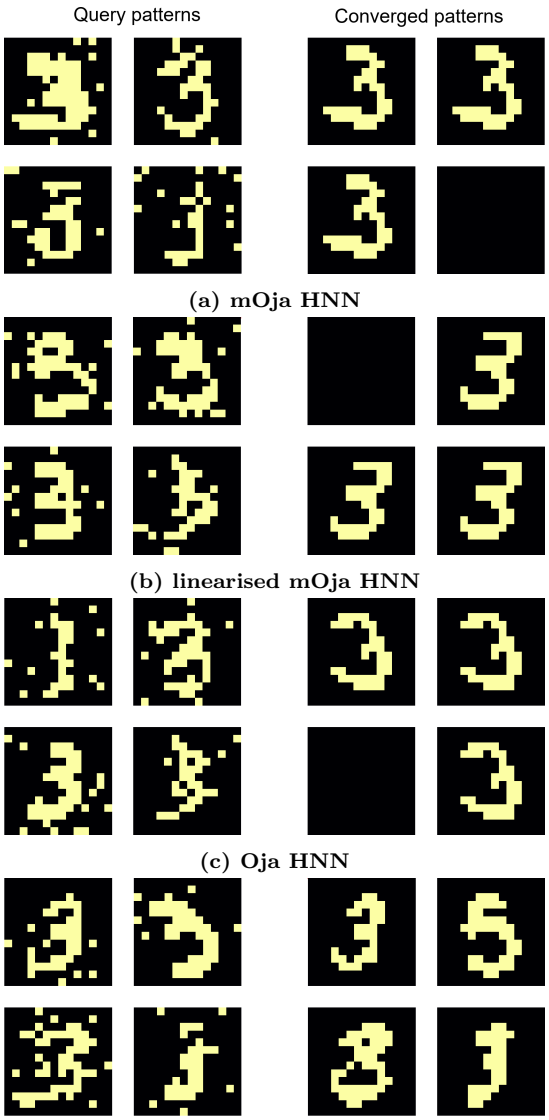
3.1 Model performances

From Table 3.1a it can be concluded that the accuracy of the mOja model does not seem to be affected by the number of unique digits stored inside the network. The noise, however, does seem to have a negative effect on the performance of the network, with 86.1% accuracy for 0.00 noise, 77.1% accuracy for 0.05 noise, and 64.7% accuracy for 0.10 noise. This seems logical, as more noise would make the query image less similar to any of the stored images with the same label, and it would thus be harder to retrieve an image that will be classified as the correct label.

Table 3.1b shows the results of the linearised mOja model, and gives very similar results to the non-linearised mOja results, as the accuracy also does not seem to be affected by the number of unique digits stored, but is negatively affected by noise. When comparing the accuracies of the linearised mOja model to the accuracies of the standard mOja model, the performance of the linearised model seems slightly better, with 88.4% accuracy for 0.00 noise, 80.5% accuracy for 0.05 noise, and 69.2% accuracy for 0.10 noise.

Table 3.1c shows the results of the regular Oja model, which again gives very similar results: The model does not seem to be affected by the number of unique digits stored, but does seem negatively affected by noise. The performance seems to be similar to both the standard mOja and the linearised mOja, but the standard deviation is a bit higher overall, with a standard deviation of 0.222 for 0.00 noise, 0.198 for 0.05 noise, and 0.227 for 0.10 noise.

Finally, Table 3.1d shows the results for the modern continuous HNN. The results for this model seem to differ quite a bit from the other three models: The performance here actually does seem to be affected by the number of unique digits stored inside the network, as more digits gives a worse performance. It also does not seem to be negatively affected by noise as much as the other three models, especially for lower number of digits. This model has 85.0% accuracy for 0.00 noise, 83.1% accuracy for 0.05 noise, and 79.9% accuracy for 0.10 noise.



(a) mOja HNN

(b) linearised mOja HNN

(c) Oja HNN

(d) continuous HNN

Figure 3.1: Query patterns (left) and the query patterns after convergence by the HNN (right). The examples are chosen on basis of having three correctly classified converged images and one incorrect classified image. Each network is trained using 60 images total, 30 of label "3", and 30 of label "5".

3.2 Examples

In Fig. 3.1 we can see some examples of query patterns before and after convergence. In Figures 3.1a, 3.1b and 3.1c some clear similarities can be seen: All of the correctly classified images have converged to

(a) Accuracy of mOja HNN

Noise	0.00		0.05		0.10	
Digits	avg	SD	avg	SD	avg	SD
1	0.851	0.143	0.785	0.172	0.641	0.213
2	0.853	0.165	0.773	0.197	0.637	0.179
3	0.874	0.166	0.775	0.179	0.659	0.197
5	0.856	0.146	0.762	0.164	0.654	0.193
8	0.862	0.149	0.781	0.171	0.636	0.222
10	0.868	0.141	0.748	0.192	0.657	0.186
total	0.861	0.151	0.771	0.179	0.647	0.198

(b) Accuracy of linearised mOja HNN

Noise	0.00		0.05		0.10	
Digits	avg	SD	avg	SD	avg	SD
1	0.890	0.135	0.793	0.177	0.684	0.185
2	0.878	0.135	0.825	0.141	0.716	0.190
3	0.893	0.116	0.810	0.156	0.698	0.220
5	0.867	0.139	0.788	0.169	0.683	0.235
8	0.894	0.145	0.805	0.152	0.701	0.205
10	0.880	0.126	0.807	0.185	0.670	0.187
total	0.884	0.133	0.805	0.162	0.692	0.204

(c) Accuracy of Oja HNN

Noise	0.00		0.05		0.10	
Digits	avg	SD	avg	SD	avg	SD
1	0.834	0.225	0.821	0.211	0.718	0.211
2	0.871	0.207	0.794	0.226	0.698	0.220
3	0.896	0.194	0.802	0.202	0.740	0.204
5	0.892	0.201	0.818	0.181	0.706	0.256
8	0.841	0.256	0.834	0.157	0.704	0.252
10	0.851	0.249	0.813	0.212	0.722	0.218
total	0.864	0.222	0.814	0.198	0.715	0.227

(d) Accuracy of Continuous HNN

Noise	0.00		0.05		0.10	
Digits	avg	SD	avg	SD	avg	SD
1	0.991	0.027	0.987	0.030	0.995	0.021
2	0.921	0.095	0.916	0.093	0.905	0.103
3	0.876	0.101	0.876	0.111	0.838	0.116
5	0.842	0.126	0.774	0.145	0.748	0.134
8	0.741	0.142	0.732	0.152	0.680	0.161
10	0.732	0.163	0.700	0.161	0.626	0.164
total	0.850	0.109	0.831	0.115	0.799	0.117

Table 3.1: These tables show the average accuracy and standard deviation over the iterations of the four models for different values of noise and amount of unique digits stored.

the same image for each of the separate networks, and the incorrectly classified images have converged to a fully black image. The convergence of the continuous HNN can be seen in Fig. 3.1d, and shows very different results. Each of the correctly classified images shows a different version of an image that resembles a three. The incorrectly classified image here is not a fully black image, but an image resembling a five.

After looking at more examples, an observation can be made about the difference between Oja based HNN's and the continuous HNN. First, Oja based HNNs do not seem to store more than one prototype per label, as they often converge to the exact same pattern. This suggest a low storage capacity in contrary to the continuous HNN. Second, When an image is incorrectly classified, an Oja based HNN seems to converge to a black image, and does not converge to an image resembling a digit at all, whereas a continuous HNN almost never converges to a fully black image, but to an image resembling a stored pattern.

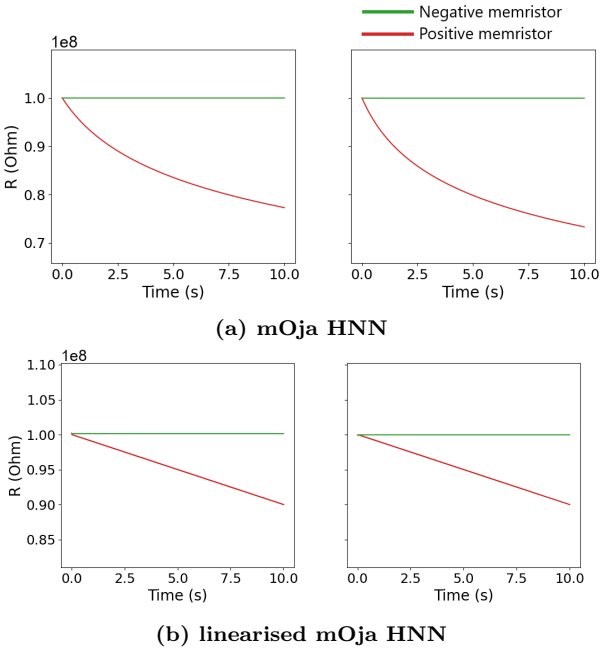
In Fig. 3.2 an example of two synapse pairs is shown. As is explained in the method section, the patterns to be stored inside the network are first shuffled, and then presented to the network in cycles of one second each. A problem that may arise from the standard decrease of the resistance of a memristor, is that the first pattern that is presented to the network has a larger effect on the weights, as the decrease of resistance is lower for lower values of current resistance (see Fig. 3.2a).

This means the model might be biased towards the first pattern shown in each cycle and especially the first pattern shown in the first cycle. This could explain the improved performance of the linearised mOja HNN as this decreases an approximately equal amount for each pattern shown (see Fig. 3.2b). If one wants to solve this bias without linearising the resistance decrease of the memristor, it could be partially solved by shuffling the patterns each cycle, but the effect of the first pattern shown

might still pose a problem.

Another problem that can noticed in both the standard and the linearised mOja HNN, is that the negative memristor does not receive pulses. The idea of a memristor synapse is that the weight can be increased and decreased, but in our case the weight is never decreased, which means the negative memristor of each pair seems useless. The reason why the weight is never decreased, is because the forgetting component of the Oja formulas is never larger than the Hebbian part of the formula (see Eq. 2.7). This could be solved by removing the negative memristor, or finding a use for the unused memristor.

The final problem that will be noted in this subsection is that the effect of a voltage pulse on a memristor's resistance is minimal when its resistance is already low. Because in this project only the forward bias is used, and thus the resistance of a memristor only decreases, it will lead to the memristor's resistance to saturate. This problem was also pointed out in [26], and a resetting scheme was suggested. A resetting scheme would mean that when the resistance of one of the memristor of a synapse pair is below a arbitrary threshold, both memristors should be reset. The memristors will be reset to their maximum resistance with negative voltage pulses, and then voltage pulses will be applied to one of the memristors such that the weight represented by the memristor pair remains the same as it was before the reset. This way the weight value remains the same, but the memristors can have a higher resistance than before the reset.



(a) mOja HNN

(b) linearised mOja HNN

Figure 3.2: Resistance decrease of memristor-synapse pairs for standard mOja (a) and linearised mOja (b). The left image is the self-connection of neuron 90, and the right image is the connection between neuron 90 and 91. The network is trained, again, using 60 images total, 30 of label "3", and 30 of label "5".

4 Discussion

In this discussion section, the relevance of the results will first be analysed. Limitations of the project will be discussed next and finally future research topics will be discussed.

4.1 Relevance of the results

To give an answer to the research question: It is indeed possible to use Nb-doped $SrTiO_3$ memristors to make an accurate classical HNN. Although the accuracy seems adequate, it is difficult to compare the performance of this network with models from other papers, for a number of reason reasons. The primary reason is the method of testing the performance of the HNN on the recalling of patterns differs between papers, such as measuring the recall rate [32], correct transition probability [33] and ROC-area under the curve [12]. There would be little value in comparing the results found in this project to the results found in other papers as the method of measuring performance differs. Another important difference between this project and some similar projects using the MNIST data-set, is that this project pre-processes the images from the MNIST data-set to be 14x14 instead of 28x28 and makes the images binary instead of gray-scale, which would make the comparison between performances even less meaningful.

4.2 Limitations of this project

In this project there were also some limitations. First, the computational power available for running the network is limited. Methods such as the Halley method discussed in the method section could have been used if there was more computational power available and might improve the pre-

dictability of the memristors, which could benefit the performance of the network. This method could also be used to pre-calculate the necessary weights, and set the resistance of each memristor in a single step such that they correspond to the pre-calculated weights.

Another limitation of this project is that the formula for memristor resistance decrease (Eq. 2.2) is not completely accurate. Because there were only a set amount of test done in [26], the formula does not describe the actual resistance decrease of a Nb-doped $SrTiO_3$ memristors. This means that we cannot accurately simulate the resistance decrease of such a memristor yet. This formula also only describes the decrease of the resistance in forward bias, whereas there might be interesting interactions with the reverse bias that could prove useful for artificial neural network applications.

4.3 Future research

An important next step for memristors in HNN would be to implement memristors in a modern HNN. Modern HNNs have an exponential storage capacity [8] in contrary to the linear storage capacity of classical HNNs [7]. Integrating memristors into a continuous HNN like the modern Hopfield network from [12] is complicated, as it does not have any weights, but it does use dot products to converge to an answer. The stored patterns matrix could be represented by a set of memristors, similar to how it was done in this project, but instead of using Oja to learn these memristor resistance values, they could be set to pre-calculated values (Using, for example, the method described in Appendix A).

One of the largest disadvantages of the continuous HNN that can be seen in the results, is that the accuracy of the continuous HNN decreases when more different labels are used. This seems logical as there is more information to be stored inside of the HNN, but the other networks do not seem to perform much worse when more labels are added. One possible reason for the worse performance compared to the classical HNN could be that the continuous HNN stores many separate prototypes, as the storage capacity is much higher than the classical HNN. There could be prototypes that are on the edge of being classified as one of multiple labels, but multiple query images could be converged to this image. This could cause for some missclassifications. This might be prevented by using a lower β value, to instead converge to an average of multiple stored patterns, instead of converging to a single stored pattern. This might mitigate the effect of outliers on the performance. Testing what effect the β value has on associative memory problems could be a topic for future research.

The linearisation seems to increase the performance of the Oja HNN, but it only works to a certain degree. As can be seen in Fig. 2.4, the decrease is very abrupt when the resistance is still high. The linearisation only works for a certain section of the resistance decrease, after which the decrease seems to follow the original curve again. An interesting topic for future research could be to test what other ways there are to remove the bias of patterns presented to the network when the memristor resistances are still high.

Currently the mOja model only works on binary patterns, which might be able to be improved to discrete, or even continuous patterns. The algorithm could also be implemented in a more computationally efficient way, which could allow for using the MNIST 28x28 px images, instead of having to compress them to 14x14 px. With more information available to the model, it should also be easier to extract more useful information, and allow for better performance.

Other weight representations could be tested as well. In this paper a positive-negative memristor synapse pair was used to represent the weights, but there are other representation options as well. There are multiple options for arranging the memristors, and one could use one or more memristors to represent a single weight, as is done in [34, 35, 36].

Another experiment that could be performed, is testing if there actually is a bias for the first pattern show in each cycle. Other methods of presenting the patterns to the model could be found and the difference in performance could be analysed.

5 Conclusion

This research aimed to find out if it was possible to design and simulate an accurate classical HNN utilising Nb-doped $SrTiO_3$ memristors. It was found that it is indeed possible to design such a network and maintain an adequate accuracy for noise magnitudes of around 0-5%. The accuracy of the (linearised) mOja HNN ranges from 63.6% to 89.4%

dependent on the amount of noise added to the images. It was found that a HNN using weights represented by memristor pairs was about equally accurate to a non-memristor based Oja HNN. It was also found that linearising the memristor decrease may have a beneficial effect on the accuracy of the network. Finally, to move forward with using memristors inside of HNNs, modern HNNs might prove more useful in practice, as they have a much higher storage capacity compared to classical HNNs.

Code Availability

All code used in this study is publicly available on GitHub at `https://github.com/JoryKlaverstijn/public_bachelor_project`

References

[1] Donald O. Hebb. *The organization of behavior: A neuropsychological theory*. Wiley, New York, 06 1949. ISBN 0-8058-4300-0.

[2] W.A. Little. The existence of persistent states in the brain. *Mathematical Biosciences*, 19(1): 101–120, 1974. ISSN 0025-5564. doi: https://doi.org/10.1016/0025-5564(74)90031-5. URL `https://www.sciencedirect.com/science/article/pii/0025556474900315`.

[3] John Hopfield. Neural networks and physical systems with emergent collective computational abilities. *Proceedings of the National Academy of Sciences of the United States of America*, 79:2554–8, 05 1982. doi: 10.1073/pnas.79.8.2554.

[4] E. Oja. Simplified neuron model as a principal component analyzer. *Journal of Mathematical Biology*, 15:267–273, 1982.

[5] Tomasz Szandała. Comparison of different learning algorithms for pattern recognition with hopfield's neural network. *Procedia Computer Science*, 71:68–75, 12 2015. doi: 10.1016/j.procs.2015.12.205.

[6] Joonki Paik and Aggelos Katsaggelos. Image restoration using a modified hopfield network. *IEEE transactions on image processing : a publication of the IEEE Signal Processing Society*, 1:49–63, 02 1992. doi: 10.1109/83.128030.

[7] Daniel J. Amit, Hanoch Gutfreund, and H. Sompolinsky. Storing infinite numbers of patterns in a spin-glass model of neural networks. *Phys. Rev. Lett.*, 55:1530–1533, 09 1985. doi: 10.1103/PhysRevLett.55.1530. URL `https://link.aps.org/doi/10.1103/PhysRevLett.55.1530`.

[8] Dmitry Krotov and John J Hopfield. Dense associative memory for pattern recognition, 2016.

[9] Mete Demircigil, Judith Heusel, Matthias Löwe, Sven Upgang, and Franck Vermet. On a model of associative memory with huge storage capacity. *Journal of Statistical Physics*, 168(2):288–299, 05 2017. ISSN 1572-9613. doi: 10.1007/s10955-017-1806-y. URL `http://dx.doi.org/10.1007/s10955-017-1806-y`.

[10] nour-eddine Joudar, Karim Moutaouakil, Ettaouil Mohamed, Sidi Farssi, and Ben Morocco. An original continuous hopfield network for optimal images restoration. *WSEAS Transactions on Computers*, 14:10, 03 2015.

[11] Qinghui Hong, Ya Li, and Xiaoping Wang. Memristive continuous hopfield neural network circuit for image restoration. *Neural Computing and Applications*, 32, 06 2020. doi: 10.1007/s00521-019-04305-7.

[12] Hubert Ramsauer, Bernhard Schäfl, Johannes Lehner, Philipp Seidl, Michael Widrich, Thomas Adler, Lukas Gruber, Markus Holzleitner, Milena Pavlović, Geir Kjetil Sandve, Victor Greiff, David Kreil, Michael Kopp, Günter Klambauer, Johannes Brandstetter, and Sepp Hochreiter. Hopfield networks is all you need, 2020.

[13] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need, 2017.

[14] Alan L Yuille and Anand Rangarajan. The concave-convex procedure (cccp). In T. Dietterich, S. Becker, and Z. Ghahramani, editors, *Advances in Neural Information Processing Systems*, volume 14. MIT Press, 2002. URL `https://proceedings.`

neurips.cc/paper/2001/file/a012869311d64a44b5a0d567cd20de04-Paper.pdf.

[15] Mikael Cho. Free images & pictures. URL https://unsplash.com/.

[16] Dmitri B Strukov, Gregory S Snider, Duncan R Stewart, and R Stanley Williams. The missing memristor found. *Nature*, 453(7191): 80–83, 2008.

[17] L. Chua. Memristor-the missing circuit element. *IEEE Transactions on Circuit Theory*, 18(5):507–519, 1971. doi: 10.1109/TCT.1971.1083337.

[18] Sung Jo, Ting Chang, Idongesit Ebong, Bhavitavya Bhadviya, P. Mazumder, and Wei Lu. Nanoscale memristor device as synapse in neuromorphic systems. *Nano letters*, 10:1297–301, 03 2010. doi: 10.1021/nl904092h.

[19] Georg Simon Ohm. *Die galvanische Kette [microform] : mathematisch bearbeitet / von G.S. Ohm.* T.H. Riemann Berlin, 1827.

[20] Alex James. A hybrid memristor–cmos chip for ai. *Nature Electronics*, 2:1, 07 2019. doi: 10.1038/s41928-019-0274-6.

[21] Chenchen Liu and Fuqiang Liu. Brain-inspired computing accelerated by memristor technology. pages 1–6, 09 2017. doi: 10.1145/3109453.3123960.

[22] Miao Hu, Catherine Graves, Can Li, Yunning Li, Ning Ge, Eric Montgomery, Noraica Dávila, Hao Jiang, Stan Williams, Jianhua Joshua Yang, Qiangfei Xia, and John William Strachan. Memristor-based analog computation and neural network classification with a dot product engine. *Advanced Materials*, 30, 01 2018. doi: 10.1002/adma.201705914.

[23] Shukai Duan, Zhekang Dong, Xiaofang Hu, Lidan Wang, and Hai Li. Small-world hopfield neural networks with weight salience priority and memristor synapses for digit recognition. *Neural Computing and Applications*, 27, 04 2015. doi: 10.1007/s00521-015-1899-7.

[24] N. M. Nasrabadi and W. Li. Object recognition by a hopfield neural network. *IEEE Transactions on Systems, Man, and Cybernetics*, 21(6):1523–1535, 1991. doi: 10.1109/21.135694.

[25] T. van Dalfsen and J. van Dam. The applicability of srtio3 in memristive devices : a preliminary investigation, 11 2013. URL http://essay.utwente.nl/65060/.

[26] Thomas Tiotto, Anouk Goossens, J Borst, T. Banerjee, and Niels Taatgen. Learning to approximate functions using nb-doped srtio3 memristors. 12 2020.

[27] A. S. Goossens, A. Das, and T. Banerjee. Electric field driven memristive behavior at the schottky interface of nb-doped srtio3. *Journal of Applied Physics*, 124(15):152102, 2018. doi: 10.1063/1.5037965. URL https://doi.org/10.1063/1.5037965.

[28] R. S. Williams. How we found the missing memristor. *IEEE Spectrum*, 45(12):28–35, 2008. doi: 10.1109/MSPEC.2008.4687366.

[29] Dmitry Krotov and John J. Hopfield. Large associative memory problem in neurobiology and machine learning. In *International Conference on Learning Representations*, 2021. URL https://openreview.net/forum?id=X4y_10OX-hX.

[30] Trevor Bekolay, James Bergstra, Eric Hunsberger, Travis DeWolf, Terrence Stewart, Daniel Rasmussen, Xuan Choo, Aaron Voelker, and Chris Eliasmith. Nengo: a Python tool for building large-scale functional brain models. *Frontiers in Neuroinformatics*, 7(48):1–13, 2014. ISSN 1662-5196. doi: 10.3389/fninf.2013.00048.

[31] Yann LeCun and Corinna Cortes. MNIST handwritten digit database. 2010. URL http://yann.lecun.com/exdb/mnist/.

[32] T. Singh. Performance analysis of hopfield model of neural network with evolutionary approach for pattern recalling. 2010.

[33] Yi Sun. Hopfield neural network based algorithms for image restoration and reconstruction. ii. performance analysis. *IEEE Transactions on Signal Processing*, 48(7):2119–2131, 2000. doi: 10.1109/78.847795.

[34] Qinghui Hong, Liang Zhao, and Xiaoping Wang. Novel circuit designs of memristor synapse and neuron. *Neurocomputing*, 330: 11–16, 2019. ISSN 0925-2312. doi: https://doi.org/10.1016/j.neucom.2018.11.043. URL `https://www.sciencedirect.com/science/article/pii/S0925231218313742`.

[35] Hyongsuk Kim, Maheshwar Pd. Sah, Changju Yang, Tamás Roska, and Leon O. Chua. Memristor bridge synapses. *Proceedings of the IEEE*, 100(6):2061–2070, 2012. doi: 10.1109/JPROC.2011.2166749.

[36] Peng Yao, Huaqiang Wu, Bin Gao, S. Eryilmaz, Xueyao Huang, W. Zhang, Qingtian Zhang, Ning Deng, Luping Shi, H. Wong, and H. Qian. Face classification using electronic synapses. *Nature Communications*, 8, 2017.

[37] J. M. Ortega and W. C. Rheinboldt. *Iterative Solution of Nonlinear Equations in Several Variables*. Society for Industrial and Applied Mathematics, 2000. doi: 10.1137/1.9780898719468. URL `https://epubs.siam.org/doi/abs/10.1137/1.9780898719468`.

[38] G. Alefeld. On the convergence of halley's method. *The American Mathematical Monthly*, 88(7):530–536, 1981. ISSN 00029890, 19300972. URL `http://www.jstor.org/stable/2321760`.

A Appendix

A.1 Voltage magnitude to decrease Resistance by a specified amount

To decrease the resistance of a memristor by a specific amount, a certain voltage is needed. To calculate the voltage necessary to get the memristor's resistance from the current resistance to a desired resistance, a method called 'Halley's method will be used [37]. This is a method that can iteratively find the value of a variable needed to obtain the root of an equation. To be able to apply this method, an equation must exist such that an expression equals 0 in the equation and the expression is differentiable twice. This equation must include both the desired resistance and the voltage magnitude of the pulse so that the necessary voltage can be calculated to get the resistance to a desired value after a single pulse. We use formula 2.1 that is repeated below:

$$R(n, V) = R_0 + R_1 \cdot n^{a+bV} \tag{2.1 revisited}$$

To find the formula for the resistance after an extra voltage pulse is applied, n of formula 2.1 is replaced by $(n + 1)$ as follows:

$$R_{des} = R_0 + R_1 \cdot (n + 1)^{a+bV} \tag{A.1}$$

Where n is given by formula 2.3:

$$n = \left(\frac{R_{cur} - R_0}{R_1}\right)^{\frac{1}{a+bV}} \tag{2.3 revisited}$$

And can thus can be filled into equation A.1:

$$R_{des} = R_0 + R_1 \cdot \left(\left(\frac{R_{cur} - R_0}{R_1}\right)^{\frac{1}{a+bV}} + 1\right)^{a+bV} \tag{A.2}$$

To make this equation more easy to differentiate, it will be rewritten as follows:

$$\left(\frac{R_{cur} - R_0}{R_1}\right)^{\frac{1}{a+bV}} - \left(\frac{R_{des} - R_0}{R_1}\right)^{\frac{1}{a+bV}} + 1 = 0 \tag{A.3}$$

Now that we have an equation in which the expression equals 0, we can write the expression as a function $f(V)$:

$$f(V) = Q^{\frac{1}{P}} - W^{\frac{1}{P}} + 1 \tag{A.4}$$

Where Q $= \frac{R_{cur}-R_0}{R1}$, W $= \frac{R_{des}-R_0}{R1}$ and P $= a + bV$.
For Halley's method we also need the first and second derivative functions:

$$f'(V) = -\frac{b \cdot \ln(Q) \cdot e^{\frac{\ln(Q)}{P}}}{(P)^2} + \frac{-b \cdot \ln(W) \cdot e^{\frac{\ln(W)}{P}}}{(P)^2} \tag{A.5}$$

$$f''(V) = \frac{b^2 \cdot Q^{\frac{1}{P}} \cdot \ln(Q)^2}{P^4} - \frac{b^2 \cdot W^{\frac{1}{P}} \cdot \ln(W)^2}{P^4} \tag{A.6}$$

Now that the function is known together with its first and second derivative, Halley's method can be used to iteratively find the magnitude of the voltage pulse necessary to decrease the memristor from its current resistance to the desired resistance:

$$V_{n+1} = V_n - \frac{2 \cdot f(V_n) \cdot f'(V_n)}{2 \cdot [f'(V_n)]^2 - f(V_n) \cdot f''(V_n)} \tag{A.7}$$

Where V_0 can be chosen as a an initial estimate of the voltage. For each iteration of this method, the error between the current calculated voltage estimate and the actual voltage pulse magnitude needed to decrease the memristor to the desired voltage will decrease. This rate of convergence to the root is cubic [38].