university of groningen / faculty of science and engineering / bernoulli institute

Improving the Learning Performance of a Simulated Memristor Neural Network Using Optimisation Algorithms

Bachelor's Project Thesis

Luuk van Keeken, s3512290, l.van.keeken@student.rug.nl
Supervisors: MSc T.F. Tiotto & Dr J.P. Borst

Abstract: A memristor is a relatively novel electronic component which has the property that its resistance can be adjusted, and that it can retain that new resistance. In contrast to the von Neumann architecture used in most modern computers, this opens up the possibility of co-locating memory and computation, as such mimicking the way in which the brain works. This allows for a decrease of computation time, an increase of energy efficiency, and a more brain-like computational substrate for artificial neural networks. Previous research has explored the use of Nb-doped $SrTiO_3$ memristors in differential pairs as weights of a simulated spiking neural network. The simulated model was capable of showing adequate performance in learning transformations of periodically time-varying input signals. The current research builds on that work by exploring the use of optimisation techniques based on Simulated Annealing and Stochastic Gradient Descent with Momentum. The implemented optimisation methods were found to significantly increase the learning performance of the simulated model.

1 Introduction

1.1 Problems of modern computing

For more than half a century, the standard in computing has been the "von Neumann architecture", first developed by John von Neumann (Von Neumann, 1945/1993). The two pillars of this architecture are the memory (where data and program instructions are stored) and the central processing unit or CPU (where data manipulation occurs). Even though there have been major advancements in developing these components, as the demand for the use of artificial neural networks (as well as computing power and data storage in general) is increasing rapidly we are starting to run into the limitations of this architecture (Sangwan and Hersam, 2020).

One obstacle inherent to the von Neumann architecture is that memory and the CPU are not *co-located*. As a result there is a limit to how much data can be transferred between these components at a time, and a limit to how much time one transfer takes. This has been known for a few decades already, and has been called the "von Neumann bottleneck" (Backus, 1978). An important consequence is that developments of individual components will have no use anymore, as the overall performance is always limited by the von Neumann bottleneck as the weakest link.

Another downside of the von Neumann architecture and the use of complementary metal-oxide-semiconductor (CMOS) materials is energy-inefficiency. Jeong et al. (2016) calculated that, looking at the current trends in computation demands, in 2040 the energy required to satisfy to computation demand will be about 10^{27} Joule. This is higher than the prospected worldwide energy production at that time.

These obstacles have stimulated a search for a new architecture that takes inspiration from properties of the brain. A major field of research associated with this search is *neuromorphic computing* (Schuman et al., 2017). Next to increasing computation speeds and energy-efficiency, an additional benefit specific to artificial neural networks (ANNs) is that such an architecture could remove the current incongruence between simulating "brain-like" ANNs using hardware that in itself is very different from the brain.

1.2 Memristors

One example of techniques and materials being developed in the field of neuromorphic computing are *memristors*. A memristor is an electronic component which (among others) has the property that its resistance can be adjusted, and that it can retain that new resistance (which is why its name is a contraction of "memory" and "resistor").

It was already theorised in the '70s (Chua, 1971) and brought into reality for the first time in 2008 (Strukov et al., 2008).

Memristors can be used in the construction of networks that are similar to neuronal structures in the brain. Just like synapses (with their adjustable connection strengths), memristors allow for memory and computation to be co-located. That is, because resistances can be adjusted and retained, values can be stored at the same location as they are processed. One consequence of this co-location is that there is no von Neumann bottleneck. Additionally, hardware implementations of ANNs can be made, as such allowing for a computational substrate that is much more brain-like (Adhikari et al., 2012; Yao et al., 2020).

Various possible applications of memristors have been and are being explored. One such application is to arrange memristors in a so-called *crossbar array* architecture. One property of this architecture is that it allows for doing matrix multiplication, arguably the most important operation in neural network learning algorithms. Additionally, analogue sensor signals can be applied directly, which means the energy costly conversion between analogue and digital signals (and vice versa) does not have to occur (Xia and Yang, 2019).

There are various techniques, materials, and principles that underlie the various types of memristors. Each type shows resistive switching behaviour, but each also has its own characteristics and properties that make it suitable for different applications. Some types, such as the ones based on ionic movement between electrodes (Hu et al., 2018), on phase switching between amorphous/non-crystalline (high resistance) and crystalline states (low resistance) (Kuzum et al., 2011), or on electric field control of ferroeletric materials (Kim and Lee, 2019), have been studied quite extensively and have been explored in various applications. Memristors based on adjusting properties of the interface between a metal and a semi-conductor have also been studied (Sawa, 2008; Goossens et al., 2018), but have not been explored in actual applications as much.

Tiotto et al. (2021) have used such interface-type memristors. Specifically they performed measurements on, and subsequently modelled, Nb-doped $SrTiO_3$ memristors, which have the property of good performance at room temperature, as well as requiring no forming processes to show resistive switching behaviour. Using the Nengo Brain Maker framework, they simulated a spiking neural network where differential pairs of memristors were organised in a crossbar array architecture, representing a fully-connected network between pre- and post-synaptic neurons. They developed a learning algorithm/rule based on the biologically plausible Prescribed Error Sensitivity learning rule, adjusted to account for the discrete resistance updates of memristors. They showed that the model was capable of showing adequate performance in learning transformations of periodically time-varying signals, even when incorporating noise in initial resistance states and resistance updates (representing real-life hardware constraints).

The current research builds on the work by Tiotto et al. (2021). Although they showed sufficient learning performance was attainable, optimal learning was seldom reached. Here, optimisation techniques based on simulated annealing and Stochastic Gradient Descent with Momentum are implemented to explore their possible benefits on improving learning performance.

2 Methods and Materials

2.1 The Neural Engineering Framework

The Neural Engineering Framework (NEF) is a mathematical theory of how cognitive functions are executed by a neural system such as the brain (Eliasmith, 2013; Stewart, 2012). By representing neurons at an adequate level of detail, and by adhering to biological constraints, it can be used to create large-scale, biologically plausible models of brain functions. The three principles that the NEF is based on are "representation", "transformation", and "dynamics", of which the first two are especially relevant to this research.

The principle of representation is about how information is encoded in a neural system and how it can be decoded. It is proposed that populations of neurons together represent values. Each neuron has its own non-linear "tuning curve", defining which input values it is most responsive to in terms of firing rate. The set of spike trains that the neurons produce then represents the input signal. In the process of decoding this representation, the post-synaptic currents (PSCs) that would be generated by each spike train are calculated. A weighted average of all PSCs is then calculated, which results in an estimate of the input signal. The best weights are calculated by minimising the squared difference between the input signal and the decoded estimate. By this method of weighting each decoding neuron with a constant and adding them, there is linear decoding of non-linearly encoded representations.

Transformation in the NEF is essentially a generalisation of how representations are decoded. Instead of finding weights to minimise the squared difference between the input signal and the decoded estimate, weights can also be found to minimise the squared difference between any function of the input signal (e.g. x^2) and the decoded estimate of that function of the input signal. This allows for incorporating complex, non-linear transformations in models, while still using linear decoding.

The Nengo Brain Maker framework is a Python package that can be used to build software models of neuronal systems, according to the NEF (Bekolay et al., 2014). Here, "Ensembles" are populations of neurons, that represent vectors of values. Between entire Ensembles, "Connections" can be made, which also can transform the vector. Relevant to the model discussed in this paper, direct connections can also be made between individual neurons of two Ensembles. The user can define learning rules which adjust the connection weights during a simulation, as opposed to Nengo pre-calculating the connection weights.

The higher the complexity of a network, and the larger its size, the longer it takes to simulate it on a CPU. NengoDL is a framework based on Nengo and the deep learning framework Tensorflow, which allows running simulations on a GPU (Rasmussen, 2019). Using this framework, deep learning techniques can be applied to the creation of neural system models, and deep learning networks can be created in a neural context. Especially relevant to this research is that it can significantly decrease the amount of time needed to simulate models with large numbers of neurons.

2.2 Nb-Doped $SrTiO_3$ memristors

Tiotto et al. (2021) performed various experiments on Nb-doped $SrTiO_3$ memristors in order to incorporate their behaviour in the simulated model. By applying positive voltage pulses (SET pulses) the resistance of a memristor is decreased, and conversely by applying negative voltage pulses (RESET pulses) the resistance is increased. The larger the amplitude of the pulse, the larger the resistance update. It became clear that the update behaviour followed the shape of a power law (see Figure 2.1), and thus it could be modelled by an equation of the following form:

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

where $R(n, V)$ is the resistance, V is the voltage of the SET pulse, n is the number of applied pulses, R_0 is the lowest possible resistance, and $R_0 + R_1$ is the highest possible resistance. By rewriting this equation, the number of applied pulses n can be calculated from the resistance $R(n, V)$:

$$n = \left(\frac{R(n, V) - R_0}{R_1}\right)^{\frac{1}{a+b\cdot V}} \tag{2.2}$$

For each of the pulse sizes, the resistance values found after applying the SET pulses were fitted to Equation 2.1, resulting in an exponent value for

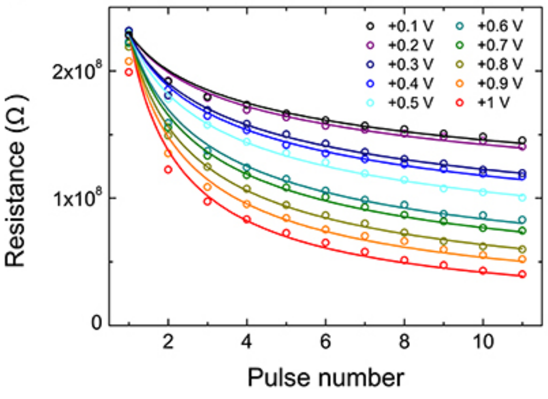


Figure 2.1: Resistance values after applying sequences of SET pulses, for various pulse sizes. Source: Tiotto et al. (2021)

each pulse size. Linear regression was then used to find fitting values for the parameters a and b of the exponent, as can be seen in Figure 2.2. This resulted in the following update equation:

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

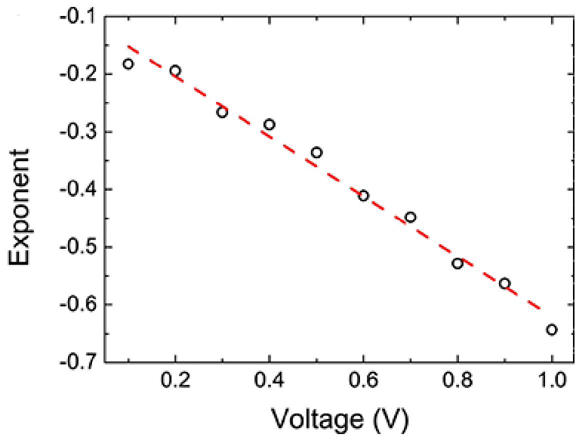


Figure 2.2: Exponents extracted from fitting resistance update sequences to Equation 2.4, as a function of the pulse size. In red, the linear regression fit for the parameters a and b of the exponent. Source: Tiotto et al. (2021)

In the simulated network, the weights of the connections between pre-synaptic ensemble neurons and post-synaptic ensemble neurons are based on the conductance values of simulated memristors. Each connection has a corresponding differential pair of memristors, one "positive" (M^+) and one "negative" (M^-). The reason for using differential pairs of memristors, and not just single memristors, is to allow for negative weights. Their conductance values (the inverse of their resistance values), normalised with regard to the memristors' minimal and maximal conductance values, then determine the value of the corresponding weight as follows:

$$\omega = \gamma \left[\left(\frac{\frac{1}{R^+} - \frac{1}{R_1}}{\frac{1}{R_0} - \frac{1}{R_1}} \right) - \left(\frac{\frac{1}{R^-} - \frac{1}{R_1}}{\frac{1}{R_0} - \frac{1}{R_1}} \right) \right] \tag{2.4}$$

$$= \gamma \left[\left(\frac{G^+ - G_0}{G_1 - G_0} \right) - \left(\frac{G^- - G_0}{G_1 - G_0} \right) \right] \tag{2.5}$$

where γ is a gain factor. By applying a pulse of a positive voltage to one of the memristors in the pair, that memristor's conductance is increased, and the corresponding weight changes accordingly. Pulsing M^+ increases the weight, while pulsing M^- decreases it. The learning rule identifies how a weight should change based on the error signal, and thus which memristor should be pulsed.

2.3 The Simulated Model

The model (Tiotto et al., 2021) that was used to explore the learning performance of a memristor-based spiking neural network is made up of three main parts, namely a pre-synaptic ensemble, a post-synaptic ensemble, and an ensemble that is used to calculate the error signal $\boldsymbol{E}$ (see Figure 2.3). The pre-synaptic ensemble takes in a certain input signal $\boldsymbol{x}$, and the learning goal of the model is to represent a certain desired transformation $f(\boldsymbol{x})$ in the post-synaptic ensemble.

Each neuron of the pre-synaptic ensemble is directly connected to each neuron of the post-synaptic ensemble. The connections represent memristors organised in a crossbar array neural network, in which the memristors' conductance values represent the weights (Xia and Yang, 2019). A learning rule is used to adjust the connection weights, by changing the conductance values of simulated memristors (see Subsection 2.4).

The learning rule requires as input the error $\boldsymbol{E}$ between the post-synaptic representation $\boldsymbol{y}$ and the ideal transformation of the input signal $f(\boldsymbol{x})$. This error is calculated in the third ensemble, which has connections coming in from the pre-sypaptic in post-synaptic ensembles. The signal $\boldsymbol{y}$ is not transformed on the connection between the post-synaptic ensemble and the error ensemble, thus it stays the same. $\boldsymbol{x}$ however is transformed on the connection between the pre-synaptic ensemble and the error ensemble, according to the desired transformation $f(\boldsymbol{x})$ specified before the simulation. (Note that Nengo itself automatically calculates the best weights for this connection before the simulation, as described in Section 2.1.) The error signal then is calculated as $\boldsymbol{E}$ = $\boldsymbol{y}$ - $f(\boldsymbol{x})$, where $\boldsymbol{y}$ $\sim f(\boldsymbol{x})$.
The input signal $\boldsymbol{x}$ is a vector that has d dimensions. Each dimension x_d is a sine wave of equal period, and all waves are uniformly phase-shifted relative

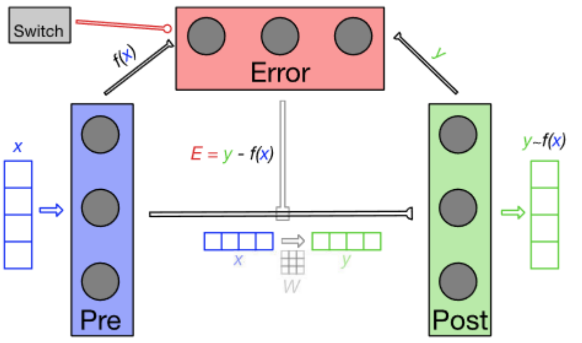


Figure 2.3: The model's structure. Source: Tiotto et al. (2021)

to each other:

$$x_d = sin(\frac{1}{4}2\pi t + i\frac{2\pi}{d}), i \in [0, d) \tag{2.6}$$

To represent the fact that not all memristors would be identical in their behaviour and initial resistance values, and to account for hardware noise, randomness was incorporated in two ways. Firstly, the initial resistance values of the memristors were sampled from a Normal distribution with as the mean 1.8 MΩ, and as the standard deviation 15% of that mean. This mean was chosen based on experiments that were done with the physical memristors that are modelled here, in which they received a series of negative voltage pulses that let the resistance values increase and approach this value. Secondly, the exponents used in the equations that model the resistance updates after a pulse were also sampled from a Normal distribution with 15% noise.

In the performed experiments, the simulation time is 30 seconds for each run. During the first 22 seconds, the model is learning as it adjusts the connections weights based on the error signal. After that time, a fourth ensemble ("Switch" in Figure 2.3) is activated, which inhibits the error ensemble. As a result, weights are no longer adjusted, and thus in the last 8 seconds the performance of the model can be tested. For $f(\boldsymbol{x})$ and $\boldsymbol{y}$ the mean squared error (MSE) and the Spearman correlation coefficient ρ are calculated. These are combined into one performance metric as $\frac{\rho}{MSE}$, reflecting that with a good fit MSE is small and ρ is close to 1. For Tiotto et al. (2021), simulations using 10 neurons lead to an average learning performance of 6.8, and simulations using 100 neurons resulted in an average learning performance of 7.9.

2.4 Learning Rules

What follows is a description of the PES learning rule, and how it was adjusted by Tiotto et al. (2021) to work with memristive synapses.

2.4.1 PES

Prescribed Error Sensitivity (PES) is a biologically plausible method for fine-tuning synaptic connection weights (MacNeil and Eliasmith, 2011; Bekolay et al., 2013). An important property of this learning rule is that weights are adjusted not based on the global error but on local errors. That is, their individual contributions to the global error. Additionally, it enables online learning (i.e. adjusting of weights during the simulation).

In the NEF, the activity a_i of a neuron i is calculated based on the neural non-linearity G (which depends on the used neural model), a gain parameter α, the neuron's encoding vector $\boldsymbol{e}$ (i.e. its "tuning curve"), the represented value vector $\boldsymbol{x}$, and a bias term b (representing the neuron's background current), as follows:

$$a = G[\alpha \boldsymbol{e} \cdot \boldsymbol{x} + b] \tag{2.7}$$

$\boldsymbol{d}$ is the decoder that specifies how the spike trains from a pre-synaptic ensemble should be averaged to best estimate the represented vector in a post-synaptic ensemble. The connection weights $\boldsymbol{\omega}_{ij}$ are then calculated as:

$$\boldsymbol{\omega}_{ij} = \alpha_j \boldsymbol{e}_j \boldsymbol{d}_i \tag{2.8}$$

for each pre-synaptic neuron i and post-synaptic neuron j.

Central to PES are two equations which can be used to calculate at each time step how the decoder values, and with that the connection weights, should be adjusted:

$$\Delta \boldsymbol{d}_i = k \boldsymbol{E} a_i \tag{2.9}$$

$$\Delta \boldsymbol{\omega}_{ij} = k \alpha_j \boldsymbol{e}_j \cdot \boldsymbol{E} a_i \tag{2.10}$$

where k is the learning rate, and $\boldsymbol{E}$ is the global error between the transformation that should be represented in the post-synaptic neuron, and that neuron's estimate. Note that in essence, the weight adjustment is only dependent on the local error $\boldsymbol{\epsilon} = \alpha_j \boldsymbol{e}_j \cdot \boldsymbol{E}$. If the local error is positive, the weight should increase; if it is negative, the weight should decrease. Each individual weight's update optimises only the corresponding "portion" of the global error.

2.4.2 mPES

mPES is an adjustment of PES that accounts for the more discrete way in which weights are updated in a network of memristors. This discreteness is the result of how the conductances of the memristors are always adjusted by applying a pulse of +0.1 V. As such, the weight updates can only have certain sizes (depending on the current conductances of the memristors in each pair). Additionally, a network of memristors is different in that there is uncertainty in the initial conductances and in the exact change of conductance after a pulse.

mPES contains a gain parameter γ in the transformation from memristor conductances to a weight value. It resembles a learning rate, but changing the size of γ only affects the influence of a pulse on the weight update sizes, not on the conductance update sizes.

It also contains an error threshold hyperparameter θ_ϵ. If all local errors are smaller than θ_ϵ, the memristors are not updated. The main reason for doing this is connected to the discrete resistance/weight updates of the memristors. If a memristor pair that corresponded to a very small, non-zero local error was to be pulsed, it would be very likely that the new local error would be bigger, as the resistance/weight update is not fine-grained enough. A second reason is that the inhibition of the error ensemble by the fourth "switch" ensemble (at the end of learning) does not bring $\boldsymbol{E}$ completely down to 0. As a result, without the use of θ_ϵ (which is set at 10^{-5}) the learning rule would continue to adjust weights. In other words, the learning phase would not be stopped while the validation phase should commence at that point.

One iteration of mPES is then as follows:

1. The global error $\boldsymbol{E}$ is projected on each neuron's tuning vector to calculate each neuron's local error $\boldsymbol{\epsilon}$: $\boldsymbol{\epsilon} = -\boldsymbol{e}_j \boldsymbol{E}$.

2. Only if at least one local error is larger than θ_ϵ are the memristors updated.

3. From $\boldsymbol{\epsilon}$ and the pre-synaptic activity $\boldsymbol{a}^{pre}$ the Δ matrix is calculated, the entries of which are analogous to $\Delta \boldsymbol{\omega}_{ij}$ in PES: Δ = -$\boldsymbol{\epsilon} \otimes \boldsymbol{a}^{pre}$. $\boldsymbol{a}^{pre}$ is filtered such that any spike of a pre-synaptic neuron is represented as a value of 1 (as opposed to 0), independent of the intensity.

4. Based on the sign of each Δ matrix entry it is determined whether the positive or negative memristor of each corresponding pair should be pulsed. If the update direction V_{ij} = $\mathrm{sgn}(\Delta_{ij})$ is larger than 0, then M^+ is pulsed. If $\mathrm{V}_{ij} < 0$, then M^- is pulsed. As such, the synapses that contribute to $\boldsymbol{E}$ in a positive way are strengthened, while the ones that contribute to $\boldsymbol{E}$ negatively are inhibited.

5. The new weights are calculated following Equation 2.5.

2.5 Optimisation Algorithms

What follows is an overview of the optimisation techniques that were implemented, and how they were based on existing optimisation methods.

2.5.1 Simulated Annealing

When one performs a search through a parameter space to find the settings that lead to the best performance, it is easy to get stuck on a local maximum when only neighbouring settings are considered. Simulated annealing is one example of an optimisation method that tries to balance constantly moving towards better neighbouring settings and avoiding to get stuck on local maxima. The essence of simulated annealing is that it sometimes allows for moves towards worse parameter settings in order to move away from a local maximum. As the search progresses, such moves happen less and less.

Kirkpatrick et al. (1983) introduced the algorithm as an analogy to the process of "annealing" in metallurgy, which is performed to make, for example, a piece of metal more workable. The material is first heated up and then slowly cooled. It is important that the temperature is decreased as slowly as possible, such that large crystals are formed (the global maximum in simulated annealing). If the material cools too quickly, the material solidifies into a structure of small crystals, making the material more brittle and less ductile (a local maximum).

In simulated annealing, there is an analogous "temperature" parameter T that in part controls whether a worse parameter setting is accepted. Each iteration a neighbouring parameter setting is picked randomly. The performance difference with the previous setting is the difference in "energy" ΔE. If ΔE is larger than 0 (i.e. the new setting is better), then the new setting is accepted. However, if the new setting is worse in terms of performance, it is accepted only with probability $e^{\frac{\Delta E}{T}}$. The temperature follows a non-increasing schedule. As a result, the probability that a similar decrease in performance will be accepted gets smaller and smaller as the iterations go on.

In this implementation, simulated annealing's temperature schedule was an inspiration for improving the way in which the exponents used in the resistance update equations are calculated. In Tiotto et al. (2021) it was already found that performance was the best when these exponents were sampled from an normal distribution with a mean equal to the value corresponding to a pulse of 0.1 V and a standard deviation of 15% ("noise") of that mean. The sampling was done once at the start of each simulation run. Here it is explored whether letting the noise percentage change according to a certain schedule, and re-sampling at each time step leads to a higher performance. The idea is that, similar to simulated annealing, a higher noise level at the start allows the weights to sometimes change to a relatively worse setting in order to escape local maxima and get closer to the global maximum. As the noise decreases, the weights should ideally settle at the global maximum.

Two schedules were considered in this implementation, namely a linear one (Algorithm 2.1) and an exponential one (Algorithm 2.2). In both cases an initial noise level and a final noise level are set, and the noise gradually changes from one to the other according to the schedule. For the linear schedule, how quickly the noise level changes depends on the total amount of time steps in the simulation. For the exponential schedule, this depends on the size of the base number of the exponentiation.

Algorithm 2.1 Linear schedule

```
$initnoise \Leftarrow$ noise level at start of simulation
$finnoise \Leftarrow$ noise level at end of simulation
$t \Leftarrow$ current time step
$total \Leftarrow$ total amount of time steps
$b \Leftarrow initnoise$
$a \Leftarrow (finnoise - b)/total$
$noise \Leftarrow a \cdot t + b$
```

Algorithm 2.2 Exponential schedule

```
$initnoise \Leftarrow$ noise level at start of simulation
$finnoise \Leftarrow$ noise level at end of simulation
$t \Leftarrow$ current time step
$base \Leftarrow$ base number of the exponentiation
$noise \Leftarrow (initnoise - finnoise) \cdot base^t - finnoise$
```

2.5.2 Adaptive Pulsing

In the implementation of Tiotto et al. (2021), whether the positive or negative memristor of a memristor pair is pulsed is based on the sign of the corresponding entry in the Δ matrix, as this sign represents whether the corresponding weight should be increased or decreased. What this does not make use of is the fact that the size of a Δ matrix entry reflects how much the corresponding weight contributes to the global error.

Thomas Tiotto (unpublished) introduced a method that does make use of this property. In this method, a certain maximum number of pulses is set. The number of pulses that a memristor pair receives stands in approximately the same proportion to that maximum number as the corresponding Δ matrix entry stands in proportion to the smallest and largest Δ values observed up to and including the current time step. A larger maximum number of pulses allows for both larger and more fine-grained weight adjustments. Whether the positive or the negative memristor receives the pulse(s) still depends on the sign of the corresponding Δ matrix entry.

2.5.3 Stochastic Gradient Descent with Momentum

The weights being updated in relation to their contributions to the global error opens up the possibility of implementing an optimisation algorithm that is inspired by Stochastic Gradient Descent (SGD) with Momentum. In SGD, a model's weights $\boldsymbol{\omega}$ are updated based on the partial derivatives of the used loss function $F(\boldsymbol{\omega})$ (i.e. a function that quantifies how well the model's predictions fit the data) with regard to each weight (Amari, 1993). Taken together, the partial derivatives form a vector pointing towards the steepest ascent of the loss function (i.e. the gradient $\nabla F(\boldsymbol{\omega})$). The negative of this gradient then designates how the weights should change to move towards the minimum of the loss function. The weight update in iteration i then is as follows:

$$\boldsymbol{\omega}_i = \boldsymbol{\omega}_{i-1} - \alpha \cdot \nabla F(\boldsymbol{\omega}_{i-1}) \tag{2.11}$$

where α is the learning rate.

Stochastic Gradient Descent differs from Gradient Descent in that at each iteration only one training sample is considered when estimating the gradient, instead of the complete data set. This is done to lower computational cost. As a result, SGD's individual weight updates can be quite variable (Bottou, 2012). This can be beneficial in terms of possibly escaping local minima of the loss function, but can also cause problems with convergence (i.e. actually finding a minimum and terminating the search) (Ruder, 2016).

One particular convergence problem in SGD occurs when the curvature of (a part of) the loss function is much steeper in one dimension than it is in the perpendicular dimension. What results is a path of weight updates that has a high degree of oscillation due to the weight updates making large jumps in the direction perpendicular to the minimum, and less so in the direction of the minimum.

Momentum aims to decrease the degree of oscillation. It does this by adding a fraction of the previous update vectors to the current update vector (where an update vector equals the learning rate time the gradient). The weights are thus updated as follows:

$$v_i = \mu v_{i-1} + \alpha \cdot \nabla F(\boldsymbol{\omega}_{i-1}) \tag{2.12}$$

$$\boldsymbol{\omega}_i = \boldsymbol{\omega}_{i-1} - v_i \tag{2.13}$$

where μ is the Momentum term with a value on the range [0, 1]. If μ is smaller than 1, past update vectors have increasingly less influence as iterations go on.

As a result of using Momentum, directions that the update vectors have in common (i.e. towards the minimum) are amplified, while moves in perpendicular directions are averaged out (illustrated in Figure 2.4). This allows for more rapid convergence.

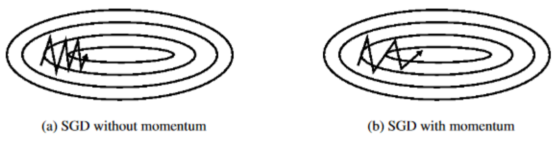


Figure 2.4: The difference between SGD with and without Momentum when converging on a minimum. Source: Ruder, 2016.

In the simulated memristor network, the Δ matrix is analogous to the gradient, in that it reflects by how much the weights should change. The adaptive pulsing method already makes use of this property by basing the numbers of applied pulses on the Δ values. In the optimisation method inspired by SGD with Momentum, before adaptive pulsing is done, a fraction of the previous Δ matrices is added to the Δ matrix calculated in the current time step. Ideally, opposing Δ values would average each other out, and similar Δ values would strengthen each other.

2.6 Experiments

Several experiments were carried out to compare the learning performances when using one of the optimisation methods to the performances in Tiotto et al. (2021). In each experiment the input was a three-dimensional sine wave, and the model had to learn the function $f(x) = x$, just as in Tiotto et al. (2021). Any other parameter values that are not mentioned here were also kept the same. Simulations using Nengo were performed with an Intel Core i7-7500U CPU, and simulations using NengoDL were performed with an Nvidia Titan Xp.

2.6.1 Simulated Annealing

For both the linear and exponential noise schedule types, simulations were performed using various combinations of initial and final noise levels, all between 0% and 40%. This range was chosen because initial tests revealed that combinations containing higher noise levels never resulted in adequate learning performance. For the same reason, not all combinations on this range were considered. The tested combinations also included cases where the final noise level was higher than the initial noise level. For each combination of noise levels the learning performance was averaged over 50 runs. Each time the network consisted of 10 memristor pairs. These simulations were performed using the Nengo framework.

The reason for also exploring combinations where the noise level increased is because of how the resistance of a memristor changes as it receives more pulses. Namely, instead of changing linearly, the resistance follows a power law, which means that the more pulses are applied, the smaller the resistance change after each pulse. In a sense, some simulated annealing properties are already present in the nature of the memristor. It was thought that actually increasing the noise over time might counteract the decreasing resistance update size in a positive way.

After these simulations it was found that there was quite some variability in the results, making it difficult to find exact relations between noise level and noise schedule settings, and learning performance. For this reason, the simulations were repeated, now using 100 memristor pairs. It was theorised that using more memristor pairs in the network would possibly decrease the variability in the results, as 100 memristor pairs/weights would have a higher representational power. These simulations were performed in NengoDL, as that framework is more suitable when doing much larger amounts of computations (as is the case when using 100 memristor pairs instead of 10). To make sure that any found difference between the 10 and 100 memristor pair simulations would not be because of the difference in the used framework, the 10 memristor pair simulations were also repeated using NengoDL.

2.6.2 Adaptive Pulsing

Simulations were performed using several maximum numbers of pulses, ranging from 10 to 800. This was done for 10 and 100 memristor pairs. Again, the learning performance was averaged over 50 runs for each setting. Note that simulated annealing was not used here. That is, the exponents used in the memristor update equations were set once at the start of each run (as in Tiotto et al. (2021)). The NengoDL framework was used.

2.6.3 Stochastic Gradient Descent with Momentum

As the SGD with Momentum method is built on top of the adaptive pulsing method, in these simulations the best maximum pulse number from the adaptive pulsing simulations was used. Simulations were performed using several Momentum term values, ranging from 0.05 to 0.95. This was done for 10 and 100 memristor pairs. The learning performance was averaged over 50 runs for each setting. The NengoDL framework was used.

3 Results

Presented here are the results of the simulations when using the implemented optimisation methods.

3.1 Simulated Annealing

Figures 3.1 and 3.2 show the results for the 10 neuron Nengo simulations and the 100 neuron NengoDL simulations, respectively. Each coloured square shows the performance of a certain combination of starting noise level and ending noise level. The higher the performance metric, the lighter the colour of the square, and the better the learning performance.

What stands out for the 10 neuron simulations (Figure 3.1) is that the best performance is reached when using combinations of low-to-intermediate noise levels. When higher noise levels are used, performance decreases (although higher starting noise levels tend to lead to better performance than ending noise levels of the same size).

Notably, for the 100 neuron simulations (Figure 3.2) the relation between noise levels and performance seems to have flipped with regard to the 10 neuron simulations. Now, combinations of intermediate-to-high noise levels result in the best performance, instead of combinations of low-to-intermediate noise levels. Performance decreases for the highest noise levels here as well, and again higher starting noise levels tends to lead to better performance than the same ending noise levels.

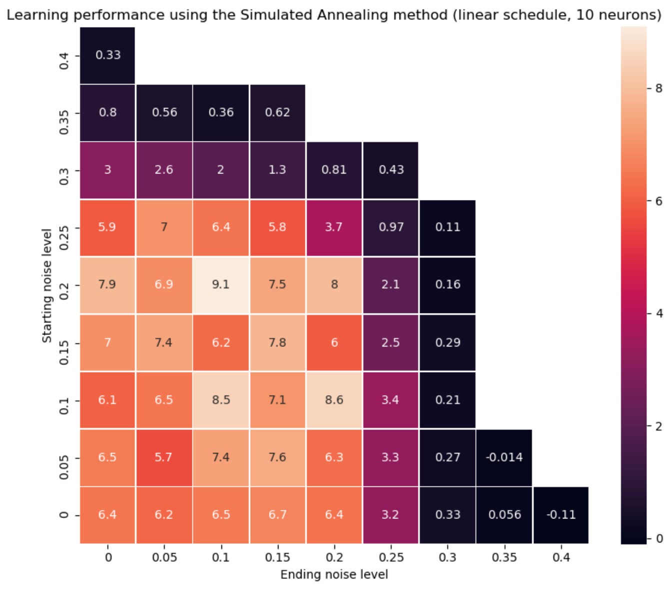
Figure A.1 shows the results of the simulations with 10 neurons and a linear schedule performed in the NengoDL framework. The patterns of which noise level combinations lead to the best and the worst performance are very similar to the 10 neuron simulations performed in the Nengo framework.

3.2 Adaptive Pulsing and Momentum

Figure 3.3 shows the performance when using adaptive pulsing and 10 neurons. Although there is quite some variability, the performance seems to increase with the number of pulse levels. Note that for all pulse levels that were explored, the performance was better than the 6.8 of Tiotto et al. (2021).

Figure 3.4 shows the performance when using both Momentum and adaptive pulsing with the number of pulse levels that resulted in the best performance, and 10 neurons. What can be seen is that performance decreases with larger Momentum terms. A significant range of the explored Momentum terms result in better than baseline performance. It should be noted that for almost all Momentum terms, the performance is worse than when only using the best adaptive pulsing parameters.

Figures 3.5 and 3.6 show the results of simulations with the same optimisation methods, but then performed with 100 neurons. Patterns similar to the ones visible in Figures 3.3 and 3.4 arise.



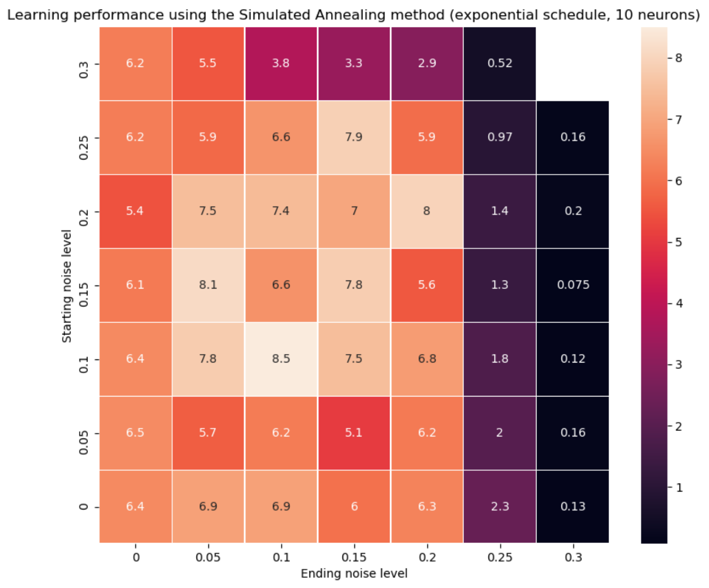
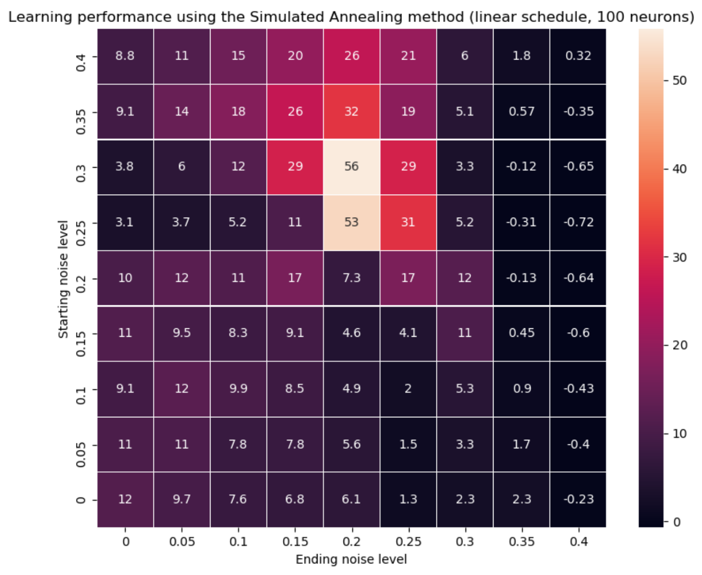


Figure 3.1: Learning performance for various combinations of noise levels, using the simulated annealing method and 10 neurons, in the Nengo framework. Top: linear schedule. Bottom: exponential schedule.

4 Discussion

This research set out to explore the use optimisation methods in improving the learning performance of a memristor-based spiking neural network. The results indicate that, at least with the right parameter settings, learning performance can indeed be increased above the baseline performances of 6.8 (10 neurons) and 7.9 (100 neurons) with the use of the implemented optimisation methods. Although this was the case for all of the methods, it should be noted that the use of Momentum seems to more often diminish the positive effect of adaptive pulsing, rather than improve upon it. It is not entirely clear why the use of Momentum does not seem to add much in terms of improving the learning performance. One possibility is that perhaps the use of Momentum is only beneficial when the search space actually contains many areas where the curvature is much steeper in one dimension than it is in the perpendicular dimension. It could be interesting to look more into the properties of this learning



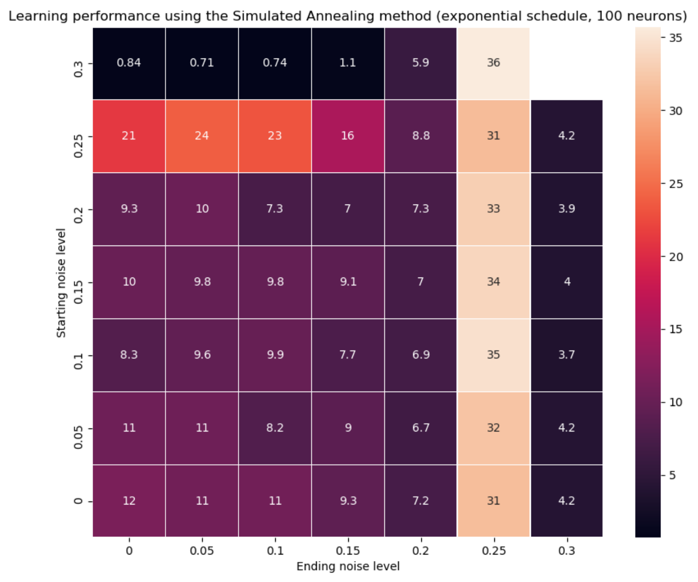


Figure 3.2: Learning performance for various combinations of noise levels, using the simulated annealing method and 100 neurons, in the NengoDL framework. Top: linear schedule. Bottom: exponential schedule.

problem's search space and implement optimisation methods based on those properties. Alternatively, one could explore the use of more general optimisation methods that are less dependent on specific search space properties.

For the simulated annealing method, arguably the most surprising result was that increasing the number of neurons affected the relation between the noise levels and the performance. It was expected that using 100 neurons instead of 10 neurons would merely decrease the variability in performance scores between similar parameter settings, as generally larger ensembles can more accurately represent values. For the 10 neuron simulations the Nengo framework was used, while for the 100 neuron simulations the NengoDL framework was used. The fact that the results for the Nengo and NengoDL 10 neuron simulations are very similar (as can be seen in Figures 3.1 and A.1) indicates that the change in best performing noise level combinations is purely the result of using different numbers of neurons. It is not clear why this difference arises.

To further improve the simulated annealing method, it could be interesting to research more

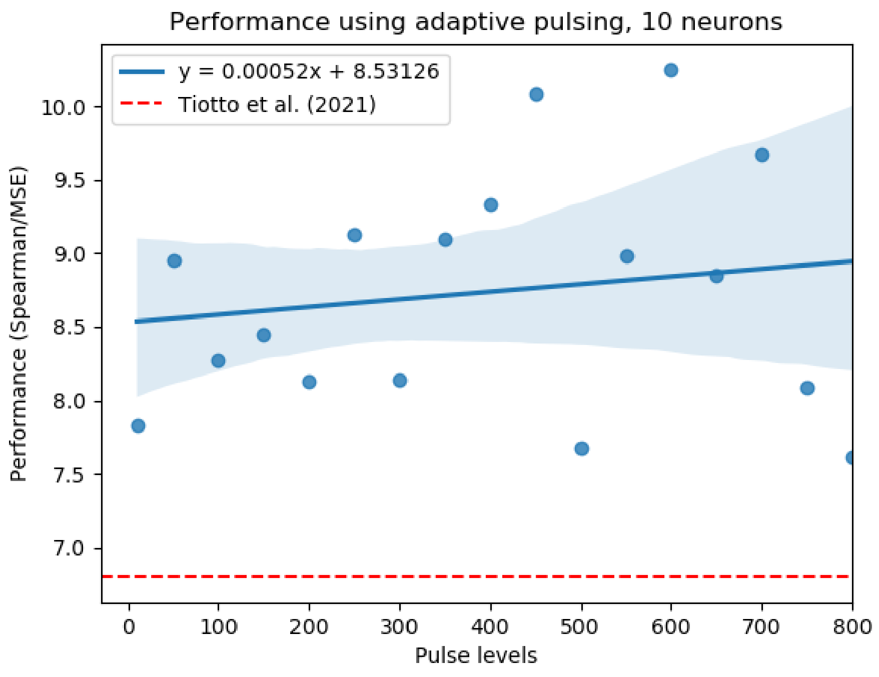


Figure 3.3: Learning performance using adaptive pulsing and 10 neurons

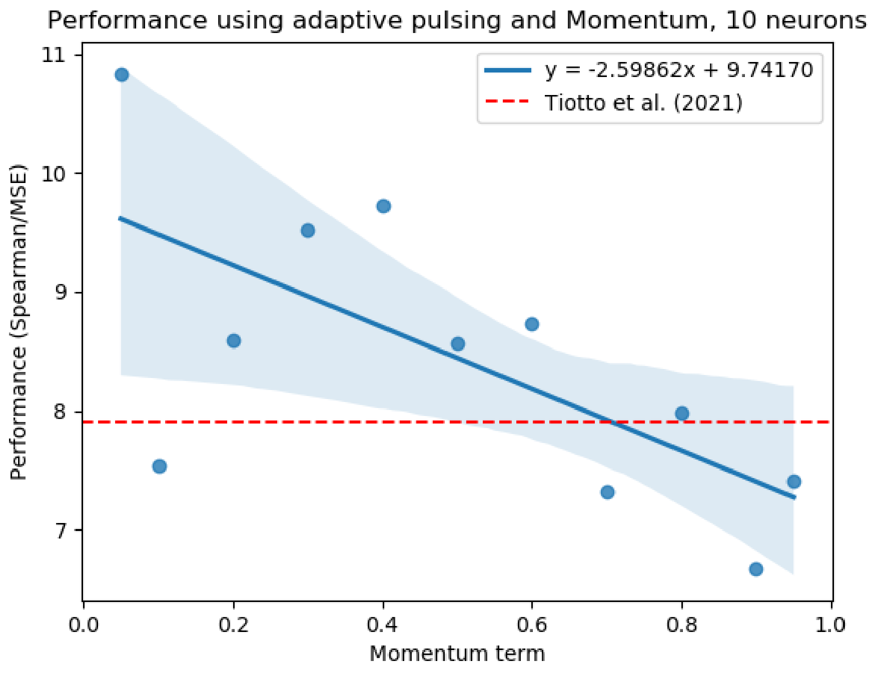


Figure 3.4: Learning performance using adaptive pulsing and Momentum, and 10 neurons

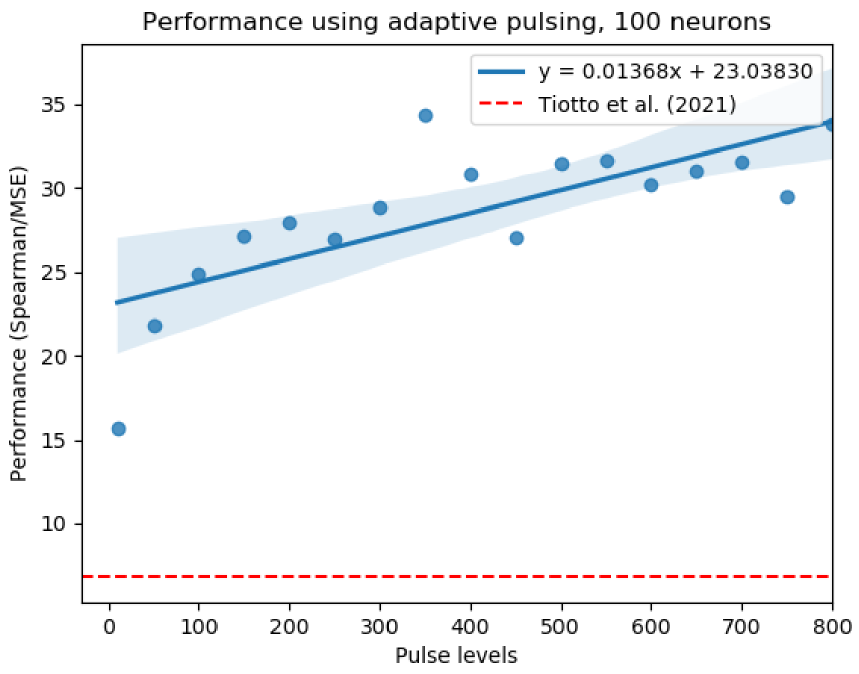


Figure 3.5: Learning performance using adaptive pulsing and 100 neurons

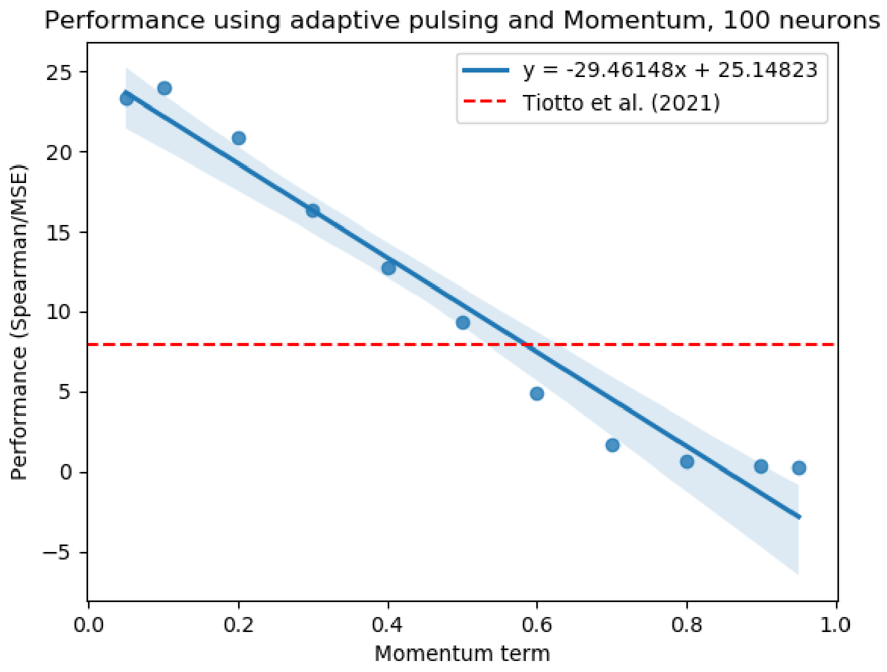


Figure 3.6: Learning performance using adaptive pulsing and Momentum, and 100 neurons

adaptive noise schedules. That is, instead of letting the noise follow a fixed schedule, it could be made to be more responsive to how the model is improving in its learning.

Regarding adaptive pulsing and Momentum, it should be noted that even though linear regression of the adaptive pulsing data points indicated that 800 pulse levels might lead to the best performance, still the numbers of pulse levels that actually resulted in the best performances in the simulations were used in the Momentum simulations (600 for the 10 neuron simulations, 350 for the 100 neuron simulations). It was not taken into account whether those best parameter settings were outliers. It would be interesting to explore whether using 800 pulse levels in the Momentum simulations would lead to different results. Above that, the linear regression results seem to indicate that numbers of pulse levels larger than 800 could lead to even higher performance when using adaptive pulsing. It is not entirely surprising that larger maximum pulse numbers result in higher performance, as they allow for applying required weight adjustments in less time steps and with more detail. More simulations should be done to explore which number of pulse levels results in the very best performance.

5 Conclusion

It was shown that the use of optimisation methods based on simulated annealing and Stochastic Gradient Descent with Momentum can significantly increase the learning performance of a simulated spiking neural network. The results of this research underscore the potential of memristors as a more energy-efficient computational device, and can function as a starting point of more exploration, in order to attain even better learning performance and to decrease computation time.

6 Code Availability

The code used in this study is available at https://github.com/LuukvanKeeken/Improving-the-learning-performance-of-a-memristor-neural-network-using-optimisation-techniques

References

Adhikari, S. P., Yang, C., Kim, H., and Chua, L. O. (2012). Memristor bridge synapse-based neural network and its learning. *IEEE Transactions on Neural Networks and Learning Systems*, 23(9):1426–1435.

Amari, S.-i. (1993). Backpropagation and stochastic gradient descent method. *Neurocomputing*, 5(4-5):185–196.

Backus, J. (1978). Can programming be liberated from the von Neumann style? A functional style and its algebra of programs. *Communications of the ACM*, 21(8):613–641.

Bekolay, T., Bergstra, J., Hunsberger, E., DeWolf, T., Stewart, T. C., Rasmussen, D., Choo, X., Voelker, A., and Eliasmith, C. (2014). Nengo: a python tool for building large-scale functional brain models. *Frontiers in Neuroinformatics*, 7:48.

Bekolay, T., Kolbeck, C., and Eliasmith, C. (2013). Simultaneous unsupervised and supervised learning of cognitive functions in biologically plausible spiking neural networks. In *Proceedings of the Annual Meeting of the Cognitive Science Society*, volume 35.

Bottou, L. (2012). Stochastic gradient descent tricks. In *Neural networks: Tricks of the trade*, pages 421–436. Springer.

Chua, L. (1971). Memristor-the missing circuit element. *IEEE Transactions on circuit theory*, 18(5):507–519.

Eliasmith, C. (2013). *How to build a brain: A neural architecture for biological cognition*. Oxford University Press.

Goossens, A. S., Das, A., and Banerjee, T. (2018). Electric field driven memristive behavior at the Schottky interface of Nb-doped SrTiO3. *Journal of Applied Physics*, 124(15):152102.

Hu, P., Wu, S., and Li, S. (2018). Synaptic behavior in metal oxide-based memristors. In *Advances in Memristor Neural Networks - Modeling and Applications*. InTech.

Jeong, D. S., Kim, K. M., Kim, S., Choi, B. J., and Hwang, C. S. (2016). Memristors for energy-efficient new computing paradigms. *Advanced Electronic Materials*, 2(9):1600090.

Kim, M.-K. and Lee, J.-S. (2019). Ferroelectric analog synaptic transistors. *Nano Letters*, 19(3):2044–2050.

Kirkpatrick, S., Gelatt, C. D., and Vecchi, M. P. (1983). Optimization by simulated annealing. *Science*, 220(4598):671–680.

Kuzum, D., Jeyasingh, R. G. D., Lee, B., and Wong, H.-S. P. (2011). Nanoelectronic programmable synapses based on phase change materials for brain-inspired computing. *Nano Letters*, 12(5):2179–2186.

MacNeil, D. and Eliasmith, C. (2011). Fine-tuning and the stability of recurrent neural networks. *PloS one*, 6(9):e22885.

Rasmussen, D. (2019). Nengodl: Combining deep learning and neuromorphic modelling methods. *Neuroinformatics*, 17(4):611–628.

Ruder, S. (2016). An overview of gradient descent optimization algorithms. *arXiv preprint arXiv:1609.04747*.

Sangwan, V. K. and Hersam, M. C. (2020). Neuromorphic nanoelectronic materials. *Nature Nanotechnology*, 15(7):517–528.

Sawa, A. (2008). Resistive switching in transition metal oxides. *Materials Today*, 11(6):28–36.

Schuman, C. D., Potok, T. E., Patton, R. M., Birdwell, J. D., Dean, M. E., Rose, G. S., and Plank, J. S. (2017). A survey of neuromorphic computing and neural networks in hardware. *arXiv preprint arXiv:1705.06963*.

Stewart, T. C. (2012). A technical overview of the neural engineering framework. *University of Waterloo*.

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

Tiotto, T., Goossens, A., Borst, J., Banerjee, T., and Taatgen, N. (2021). Learning to approximate functions using Nb-doped $SrTiO_3$ memristors. *Frontiers in Neuroscience*, 14:1456–1472.

Von Neumann, J. (1993). First draft of a report on the EDVAC. *IEEE Annals of the History of Computing*, 15(4):27–75. Original work published 1945.

Xia, Q. and Yang, J. J. (2019). Memristive crossbar arrays for brain-inspired computing. *Nature Materials*, 18(4):309–323.

Yao, P., Wu, H., Gao, B., Tang, J., Zhang, Q., Zhang, W., Yang, J. J., and Qian, H. (2020). Fully hardware-implemented memristor convolutional neural network. *Nature*, 577(7792):641–646.

A Appendix

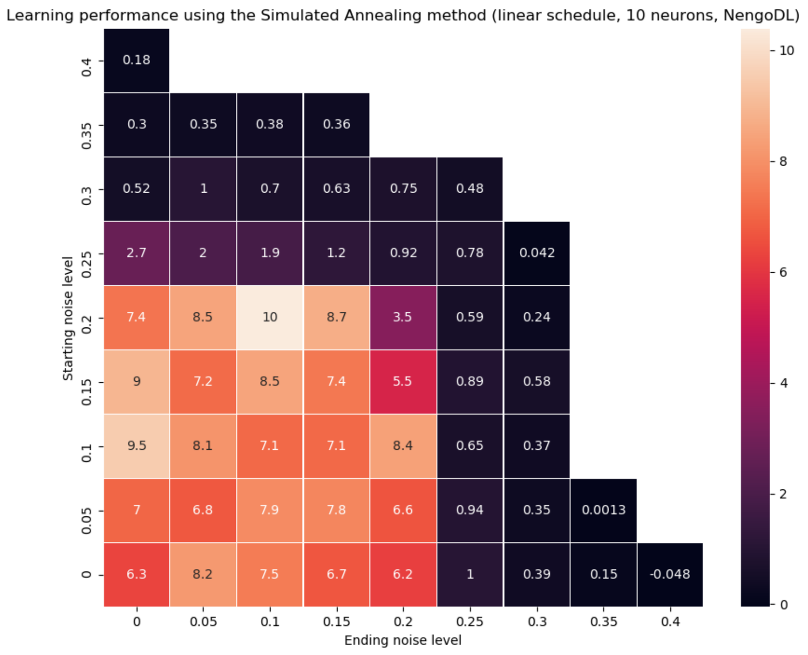


Figure A.1: Learning performance for various combinations of noise levels, using the simulated annealing method and 10 neurons, in the NengoDL framework.