Aid Ahmetovic

Deep Learning in Spiking Neural Networks with Memristive Synapses

Master's Thesis

to achieve the university degree of

Diplom-Ingenieur

Master's degree programme: Information and Computer Engineering

submitted to

Graz University of Technology

Supervisor:
Assoc.Prof. Dipl.-Ing. Dr.techn. Robert Legenstein
Institute of Theoretical Computer Science

Graz, December 2018

Affidavit

I declare that I have authored this thesis independently, that I have not used other than the declared sources/resources, and that I have explicitly indicated all material which has been quoted either literally or by content from the sources used. The text document uploaded to TUGRAZonline is identical to the present master's thesis.

SIGNATURE INFORMATION	**Signatory**	Aid Ahmetovic
	Date/Time-UTC	2018-11-30T10:28:30+01:00
	Verification	Information about the verification of the electronic signature can be found at: https://www.signaturpruefung.gv.at
Note	This document is signed with a qualified electronic signature. According to Art. 25 para. 2 of the Regulation (EU) No 910/2014 of 23. July 2014 ("eIDAS-Regulation") it shall have the equivalent legal effect of a handwritten signature.	

Date

Signature

Abstract

Recent advances in the field of nanotechnology led to the physical realization of memristors, whose dynamics is suitable for implementing the synaptic weights in both spiking and artificial neural network structures. In this thesis, we define a novel concept for deep spiking neural networks containing custom neuron models with memristive synapses. Spiking neural networks (SNNs) have become increasingly popular since achieving the same or even better results compared to the artificial neural networks (ANNs). We demonstrate that the output spike times of our neuron model are differentiable, thus standard supervised learning algorithms can be applied directly to networks of such neurons. We validate that small networks with standard non-memristive synapses can be trained to solve simple logical functions. In a deeper SNN setup, our model obtains a performance similar to ANN when solving more complex classification tasks, such as Iris and MNIST. Similar results are obtained when these SNN are trained on logical functions with modeled memristive synapses. Finally, we demonstrate the learning capabilities of single neuron with real hardware memristive synapses. In such a framework, the logical NAND function was learned by a single neuron. In general, we get slightly degraded results with real memristive synapses as compared to simualated synapses.

Zusammenfassung

Forschung im Bereich der Nanotechnologie führte zur physikalischen Realisierung von Memristoren, deren Dynamik geeignet ist, die synaptischen Gewichte sowohl in spikenden als auch künstlichen neuronalen Netzwerkstrukturen zu implementieren. In dieser Arbeit definieren wir ein neuartiges spikendes neuronales Netzwerk, welches spezielle Neuronenmodelle mit memristiven Synapsen enthält. Spikende neuronale Netzwerke (SNNs) sind zunehmend populär geworden, da dies ähnlich gute oder sogar bessere Ergebnisse im Vergleich zu künstlichen neuronalen Netzwerken (KNNs) erzielen. Wir zeigen, dass die Spikezeiten unseres neuronalen Modells differenzierbar sind, sodass die standardisierten überwachten Lernalgorithmen in den Netzwerken unserer neuronalen Modelle direkt angewendet werden können. Wir validieren, dass ein Neuron mit standarden nicht-memristiven Synapsen trainiert werden kann, um einfache logische Funktionen zu lösen. In einer tieferen SNN-Konfiguration hat unser Modell eine ähnliche Performanz wie KNNs, wenn komplexere Klassifizierungsaufgaben gelöst werden, wie Iris und MNIST. Ähnliche Ergebnisse werden erhalten, wenn diese SNN auf logischen Funktionen mit modellierten memristiven Synapsen trainiert werden. Schließlich demonstrieren wir die Lernenfähigkeiten eines Neurons mit echten memristiven Synapsen. In einem solchen Rahmen war die logische NAND-Funktion von einem einzelnen Neuron gelernt. Im Allgemeinen erhalten wir etwas verschlechterte Ergebnisse mit echten memristiven Synapsen im Vergleich zu simulierten Synapsen.

Acknowledgment

I would like to express my sincerest gratitude to:

My supervisor, Professor Robert Legenstein, who provided me support and valuable feedbacks in every stage of my thesis. This motivated me to pursue research in this interesting research area.

My wife Emina, for providing me the most substantial support and encouragement,

My family, especially my parents, sister, grandmother and grandfather, for supporting me not only in academic, but also in every aspect of my life,

Professor Themis Prodomakis, Alexandru Serb and Spyros Stathopoulos for generosity, help and assistance during my visit at the University of Southampton.

Michael Müller for providing me this great template.

Contents

1 Introduction

Deep learning with traditional artificial neural networks (ANNs) has been proven a great tool that achieves above human level performances in many application areas [1]. In deep neural network structures, time-consuming training is imperative to obtain state of the art results. In the case of supervised learning tasks, where for each input we have a corresponding true label value, ANNs are trained using stochastic optimization algorithms (in the form of the gradient descent) in combination with the backpropagation algorithm [2]. The main idea behind the backpropagation algorithm is that the gradients are calculated through backpropagating errors from the final layer to the first layer. However, brain-inspired processing is not captured with ANNs. One can argue that ANNs, due to their static model, cannot capture processes present in the brain. For this reason, another form of the neural networks, called spiking neural networks (SNNs), was invented and proven to be more suitable for modeling brain-like computations [3]. Briefly, spiking neural networks contain neuron models interconnected with synapses (with changeable weights). The presence and timing of individual spikes are essential for that type of networks. Neuron models in SNNs are based on real neurons in the brain, where the presynaptic action potentials (spikes) change the membrane potential of the postsynaptic neuron, until the voltage reaches the threshold, after which the postsynaptic neuron fires. This means that spiking neurons communicate via discrete events, called spikes, while artificial neurons communicate via analog values. It was also proven that the computational capabilities of SNNs are larger than those of ANNs [3] and compared to ANNs, learning in SNNs takes an entirely different form. Unsupervised learning is suitable in SNNs in the form of spike-timing-dependent plasticity rules (STDP), where the time difference between the presynaptic and postsynaptic spikes plays a crucial role. The original idea behind supervised learning cannot be applied to SNNs because of the non-differentiability of spikes, which are discrete events and therefore the original form of the backpropagation algorithm cannot be used. With several simplifications of SNN models it is possible to compute gradients, and we summarize those approaches in Section 3. We define a new neuron model in Section 6.1, in which the membrane potential is a superposition of linear functions in time. Our spiking neuron also enables an exact gradient computation, thus, supervised learning in its original form can be applied. In Section 6.2, we demonstrate the learning capabilities of single neuron as well as of a network of neurons, by training them on the AND, OR, XOR and NAND logical functions.

In this thesis, not only a new neuronal model is defined, but we also include the concept of the memristive synapses, where our synaptic weights between neurons are modeled with memristive devices. The general intro-

duction to memristive devices and modeling of such devices is summarized in Section 4. Memristors are non-volatile memory devices that have low power consumption and dynamics suitable for non-Von Neumman architectures. We include supervised learning results of the AND, OR, XOR and NAND logical functions in the SNN setup containing real memristive hardware devices. This is similar to the prior work described in Section 5 , where the real memristive hardware devices have been used in an ANN supervised learning setup, and in a SNN unsupervised learning setup.

Results on the Iris and MNIST classification tasks are presented in Sections 6.5 and 6.6, respectively, where we used deeper SNNs to solve the tasks. We conclude that results of networks containing our custom spiking neuron models are close to results of ANNs when both network types have the same number of learnable parameters.

2 Neuron models and spiking neural networks

In this chapter, spiking neural networks (SNNs) are introduced in Section 2.1. The Leaky-Integrate and Fire (LIF) neuron model is summarized in Section 2.2.1, while synaptic plasticity is addressed in Section 2.3.

2.1 Spiking neural networks (SNNs)

Traditional artificial neural networks (ANNs) are a fully-connected feed-forward or recurrent structures of units called artificial neurons. Connections between artificial neurons typically have a weight that is adjusted during the learning process. Fully-connected feed-forward ANNs consist of multiple layers: an input layer, several hidden layers, and an output layer.

The first and the simplest artificial neuron model was the McCulloch-Pitts model [4]. The concept behind the McCulloch-Pitts model is straightforward: if a weighted sum of incoming signals is larger than the threshold value, the output of the neuron model is 1, otherwise, the output is 0. The second generation of the artificial neuron models does not use the threshold function to compute the output, but a continuous activation function such as sigmoid or hyperbolic tangent. ANNs, with one hidden layer and the continuous activation function, can approximate any continuous function on a bounded interval (universal approximation theorem) [5].

Spiking neural networks (SNNs) as a new generation of neural networks are highly inspired by the architecture of the brain. SNNs consist of the biologically inspired mathematical models of neurons which communicate in continuous time with discrete events - action potentials or spikes. Incoming action potentials change the membrane potential of the neuron until it reaches the threshold value and produces a spike itself. Neurons communicate to other neurons through synapses. SNNs can be perceived as a fully-connected feed-forward or recurrent structures of spiking neuron models. Synaptic connections between those neuron models have changeable synaptic weights associated to them. Synaptic weights in SNNs are updated during the learning process.

In traditional artificial neural networks (ANNs) analog values form the input and output of the single neuron, and those values can be understood as the frequency of the neuron spike firing. In SNNs, the primary mean of communication and computation is the presence and timing of individual spikes.

Because of that, in SNNs we need new notations to define a meaning of the presence and timing of individual spikes. Those notations are called coding schemes. One example of such a coding scheme is temporal coding. The main idea behind the temporal coding scheme is that the input vector

of real numbers is converted into a spike train based on the biological idea: the more intensive the input, the earlier the spike transmits. This biological inspiration can be found in our visual system [6].

To encode n analog numbers as spike times relative to T_{in} we need to consider a SNN structure with n input neurons N_i. The input is fed by n-dimensional analog values $\mathbf{x} = (x_1, ..., x_n)$ with all x_i values inside a bounded interval $[a, b]$ of $\mathbb{R}$. Those values are converted into spike trains relative to the timing of an external stimulus: if an external stimulus comes at time T_{in}, spike emission of neuron N_i is coded at time $t_i = T_{in} + (b - x_i)$ [7]. In temporal coding, time of the first spike for each neuron contains all the information about the new stimulus: a neuron which fires shortly before the external stimulus indicates a strong stimulation, while earlier firing encodes a weaker stimulation. An illustration of the temporal coding scheme is presented in Figure 1.

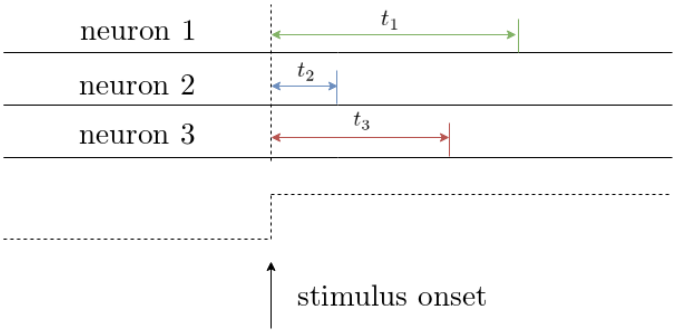


Figure 1: Temporal coding where the spike train of three neurons is shown. The second neuron is the one that fires first after the stimulus onset (arrow).

In this coding principle, we face the restriction that one neuron can fire only once. That means that temporal coding cannot be applied to more continues computing where we can observe spike trains: neurons can fire multiple spikes.

2.2 Spiking neuron models

The general idea of a neuron model in SNNs is to handle the incoming action potentials (spikes) by modifying its membrane potential. When the neuron's membrane potential reaches the predefined threshold value the neuron produces an action potential itself.

2.2.1 Integrate-and-fire (I&F) neuron models

In integrate-and-fire (I&F) neuron models, every spike is a discrete event defined only by the precise timing. Hence, an action potential in integrate-and-fire models is described by the Dirac-delta function.

One important neuron model that we will examine is a leaky-integrate-and-fire (LIF) model [8]. The electrical circuit of the LIF neuron model contains capacitor C in parallel to the resistance R, driven by the external current $I(t)$ (shown in Figure 2). The mathematical model of the LIF neuron is defined with the first order linear differential equation:

$$\tau_m \frac{du(t)}{dt} = u_{rest} - u(t) + RI(t) \tag{1}$$

The neuron spikes when:

$$u(t^f) = \vartheta \quad \text{with} \quad u^{'}(t^f) > 0 \quad : \quad u(t^f) \leftarrow u_{reset} \tag{2}$$

where:

- $\tau_m = RC$ is the time constant of the neuron membrane that models the leakage of the voltage,
- t^f is the threshold crossing time,
- u_{rest} is the resting potential, and,
- u_{reset} is the reset potential to which the membrane potential is set immediately at time t^f.

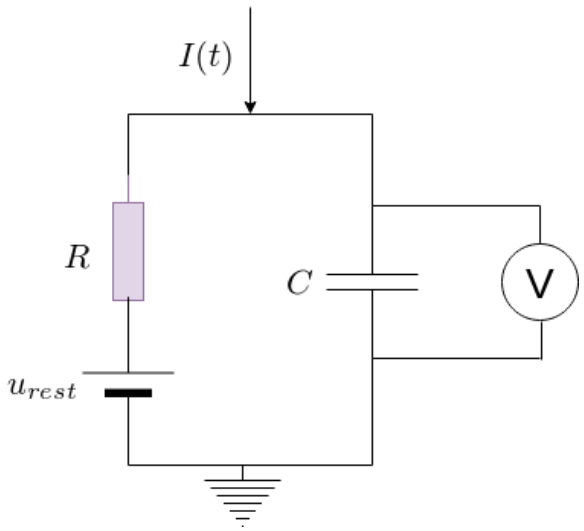


Figure 2: Electrical circuit of the Leaky-Integrate-and-Fire neuron model.

The LIF model is used often in spiking neural network structures because of its simplicity and possibility to find the analytical solution of equation (1).

However, the external reset mechanism is needed for the threshold crossing condition, and is defined in equation (2).

In the LIF neuron with current based synapses, $I(t)$ is defined as a weighted sum of incoming action potentials:

$$I(t) = \sum_i w_i \sum_f \epsilon(t - t_i^{(f)}) \quad \text{with} \quad \epsilon(s) = \exp\left(-\frac{s}{\tau_{\text{fall}}}\right)\Theta(s) \tag{3}$$

where $t_i^{(f)}$ represents the time of the $f-th$ spike of $i-th$ presynaptic neuron, w_i is the $i-th$ synaptic weight, τ_{fall} is a decaying time constant, and $\Theta(x)$ represents the Heaviside function.

In the LIF neuron with conductance based synapses, $I(t)$ is defined as:

$$I(t) = -(u(t)-E_{syn}) \sum_i g_i(t) \sum_f \epsilon(t-t_i^{(f)}) \quad \text{with} \quad \epsilon(s) = \exp\left(-\frac{s}{\tau_{\text{fall}}}\right)\Theta(s) \tag{4}$$

where $t_i^{(f)}$ represents the time of the $f-th$ spike of $i-th$ presynaptic neuron, $u(t)$ is the current membrane potential, E_{syn} is the reversal potential, $g_i(t)$ is the $i-th$ synaptic weight, τ_{fall} is a decaying time constant, and $\Theta(x)$ represents the Heaviside function. Reversal synaptic potential E_{syn} determines the type of a synapse, thus the synapse is excitatory if $E_{syn} > \vartheta$, or inhibitory if $E_{syn} < \vartheta$.

2.3 Synaptic plasticity and spike-timing-dependent-plasticity (STDP)

Opposed to the constant parameters in neuron models, connections between neurons are dynamic which forms the basis for learning and adaptation. Synaptic plasticity is a term referring to the modification, creation, and removal of the synaptic connections between neurons in the brain. From the current biological knowledge of the brain, several processes take place: long-term potentiation (LTP) related to increasing changes of synaptic weights that last several hours to days, and long-term depression (LTD) describing the decrease of weights over a long time interval. On the other hand, processes that change the synaptic weights on the time scale of seconds to minutes are called: short-term potentiation (STP) if weights are increased, or short-term depression (STD) if weights are decreased.

Spike-timing-dependent plasticity (STDP) is a form of synaptic plasticity which is highly sensitive to the precise spike timing of the presynaptic and postsynaptic neurons [9]. The main idea is the following: the maximal increase of the synaptic weight occurs on a connection where the presynaptic neuron fires shortly before the postsynaptic neuron, while the decrease

of synaptic weights happen if the presynaptic neuron fires shortly after the postsynaptic neuron. If the time difference between the presynaptic and postsynaptic neurons is large, there is no change in the synaptic weights.

To use STDP in SNNs, learning windows for LTP and LTD are created/derived from the neurobiological experiments [10]. It is assumed that the weight changes Δw are proportional to:

$$\Delta w = \begin{cases} A_+ e^{\left(\frac{-\Delta t}{\tau_+}\right)}, & \text{if} \quad \Delta t > 0 \\ -A_- e^{\left(\frac{-\Delta t}{\tau_-}\right)}, & \text{if} \quad \Delta t \leq 0 \end{cases} \tag{5}$$

where A_+, A_-, τ_+, τ_- are constants and $\Delta t = t_{post}$ - t_{pre}. The plot of equation (5) is called a learning window. The x-axis of the learning window corresponds to the time difference $\Delta t = t_{post}$ - t_{pre} of the postsynaptic and presynaptic spike times, while the y-axis corresponds to the weight change that is positive if $\Delta t > 0$ and close to zero, otherwise the weight change is negative. One example of the learning window function is presented in Figure 3.

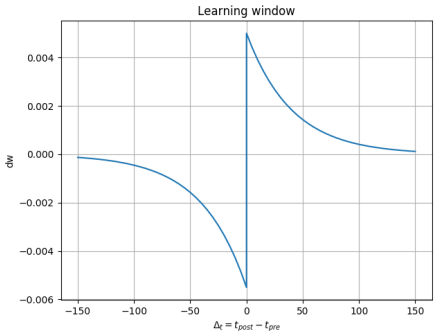


Figure 3: One example of a STDP learning window function.

General changes in the synaptic weights are done based on the following equation:

$$w \leftarrow w + \Delta w \tag{6}$$

Even though STDP opens many exciting ideas regarding learning in SNNs, there are also several drawbacks and problems that might occur. For instance, with STDP, one cannot apply constant potentiation or depression without fixing bounds for the values of weights (etc. in the range [0, w_{max}]). Even with bounded weights, it is possible to have a restrained network when all weights are small or to have an unstable network when all weights are

large. Those cases directly limit the network ability for further adaptation.

3 Supervised learning in spiking neural networks

In this section, several supervised learning approaches for SNNs are presented. The general idea for all those methods is to find the optimal synaptic weights w_{ij}, given the input vector of spike times $T^{in}(t)$ and a vector of target spike times $T^{target}(t)$, such that the spike times of neurons in the output layer $T^{out}(t)$ are equal to the wanted spike times $T^{target}(t)$.

This section consists of four subsections, where in the first subsection 3.1 the SpikeProp method is explained. The introduction of the ReSuMe method is explained in the subsection 3.2. The novel approach of Gradient Descent in SNNs will be presented in subsection 3.3. Supervised learning ideas in the recurrent neural networks (LSNNs) are discussed in subsection 3.4 and a supervised learning method for SNNs with temporal coding is discussed in subsection 3.5.

3.1 SpikeProp

SpikeProp [11] is a gradient-based method for supervised learning in SNNs. The main idea behind SpikeProp originates from the computation in traditional ANNs where the backpropagation algorithm [2] is used. Gradient computation in SNNs is infeasible due to the discrete output spikes of spiking neurons. To tackle this problem, several simplifications should be defined such as the one used in the SpikeProp method where one neuron can fire only once during the time-course of the simulation. That means that the neuron's membrane potential must reach the threshold value exactly once to be sure that the gradients can be evaluated. Otherwise, gradients are not defined. Another assumption that solves the discontinuity problem is that the membrane voltage is linearly approximated in the region around the spike time so that gradients can be evaluated.

In the SpikeProp method, it is assumed that several weight connections w_{ij}^k containing different delay d_{ij}^k are present between the presynaptic neuron i and postsynaptic neuron j. Initially, the SpikeProp method has been defined using the Spike Response Model (SRM) [12], where the membrane potential of the neuron j is defined as:

$$V_j(t) = \sum_{i \in \Gamma_j} \sum_k w_{ij}^k \epsilon(t - t_i^{out} - d_{ij}^k) \tag{7}$$

where:

- Γ_j is a set representing the presynaptic neurons connecting to neuron j,

- w_{ij}^k is the weight of the synaptic terminal k between neurons i and j,
- $\epsilon(t) = \frac{t}{\tau}\exp(1 - \frac{t}{\tau})$ with some time constant τ,
- t_i^{out} is the firing time of the neuron i, and,
- d_{ij}^k is the synaptic terminal delay.

The objective function that is minimized is the mean-squared error between the actual and wanted output spike times:

$$E = \frac{1}{2}\sum_j (t_j^{out} - t_j^{target})^2 \tag{8}$$

With the previously noted objective function it is straightforward to compute the gradients of the loss function E concerning weights w_{ij}^k:

$$\frac{\partial E}{\partial w_{ij}^k} = \frac{\partial E}{\partial t_j}(t_{j^{out}})\frac{t_j}{w_{ij}^k}(t_{j^{out}}) = \frac{\partial E}{\partial t_j}(t_{j^{out}})\frac{\partial t_j}{\partial V_j}(t_{j^{out}})\frac{\partial V_j}{\partial w_{ij}^k}(t_{j^{out}}). \tag{9}$$

To compute the gradients from (9), it is assumed that we have linear approximated function V_j^m in the small region around $t = t_j^{out}$. This is directly connected with having the constant partial derivative. Linear approximation of the membrane voltage in the SpikeProp method is necessary due to the Spike Response Model that is used. That is one of the disadvantages of the SpikeProp method compared to our method described in Section 6.1, because in our model the membrane voltage is differentiable and no linear approximation is needed.

Results of a SNN optimized with the SpikeProp method were presented on the non-linear logical XOR task and several classification tasks (Iris, Wisconsin breast cancer, and the Statlog Landsat), where supervised learning in SNNs obtains comparable performance to the traditional ANNs in [11] . In the original SpikeProp definition, authors argued that all weights must be positive and only in that case the algorithm guarantees successful convergence. Later, the convergence of the SpikeProp algorithm with negative weights was also proved, and the algorithm was thoroughly investigated and improved over the time [13].

3.2 ReSuMe

The ReSuMe method [14] was initially motivated by the examination of real-time movement control techniques for disabled persons. It is a method that is substantially different to other supervised learning methods such as SpikeProp presented in the previous section. While most of the supervised learning methods in SNNs include some simplifications for computing gradients, ReSuMe method is a combination of the learning windows (such as ones included in the spike-timing dependent plasticity) and a novel concept called **remote supervision**.

In [14], the authors defined the ReSuMe learning procedure as following: each neuron that updates its synaptic connections based on the ReSuMe method in the SNN, has an additional reference (teacher) signal that represents the desired output time of that neuron. However, those teacher signals are not directly connected to the learning neurons, but they rather supervise the synaptic weights that connect to the learning neuron itself. That is schematically illustrated in Figure 4.

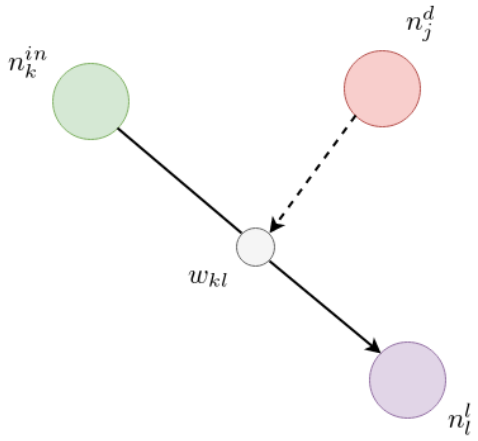


Figure 4: Input n_k^{in}, learning n_l^l and teacher n_j^d neurons in ReSuMe method.

Synaptic weights are updated according to two learning rules. The first rule $W^d(s^d)$ is taking into account the difference between presynaptic and reference spike times ($s^d = t^{d,(f)} - t^{in,(f)}$), whereas for the second rule $W^l(s^l)$ the timing difference between the presynaptic and postsynaptic connections ($s^l = t^{l,(f)} - t^{in,(f)}$) is important. From the first rule we have that the excitatory/inhibitory synapse is facilitated/depressed if the presynaptic neuron fires shortly before the reference spike time, while in the second rule is defined in a way that synapses are depressed/facilitated when the presynaptic neuron fires directly before the postsynaptic neuron. The second rule is sometimes referred to as an anti-STDP rule.

If the time t_m^f defines the $f-th$ spike of neuron m, the spike train of the $m-th$ neuron is:

$$S_m(t) = \sum_f \delta(t - t_m^f) \tag{10}$$

where $\delta(x)$ denotes the Dirac-delta function.

Synaptic weights w_{ki} are updated based on the following equation:

$$\frac{d}{dt} w_{ki}(t) = [S^d(t) - S^l(t)] \left[a + \int_0^\infty W(s) S^{in}(t-s) ds \right] \tag{11}$$

where the $S^d(t)$, $S^{in}(t)$ and $S^l(t)$ are spike trains of target, input and learning neurons respectively. The integral part of the previous equation defines the correlation change of the weights that is dependent on the window function $W(s)$ and input spike train $S^{in}(t)$. The constant a defines a non-correlation weight change contribution. If the synapses are excitatory, a constant is positive and $W(s)$ is similar to the STDP learning curve. For inhibitory synapses, a is negative, and the shape of the $W(s)$ looks like learning windows in anti-STDP rules.

We can conclude that the difference $[S^d(t) - S^l(t)]$ plays the main role regarding the weight change modifications and that the weight change is zero if and only if $S^d(t) = S^l(t)$.

One of the advantages of the ReSuMe method compared to our method described in Section 6.1 is neuron's ability to spike multiple times which makes it more convenient for modeling spike trains. The disadvantage is that each neuron is individually supervised using teacher signals and we do not have a global supervision as in our method.

In [14] it has been shown that ReSuMe method converged fast when learning desired output spike trains, given the input spike trains.

3.3 Gradient descent in spiking neural networks

The authors of [15] introduced a novel idea for supervised learning in SNNs, by defining the differential network model with differential current-based synapses. They modified the standard current based synaptic current dynamics that is usually modeled as a linear filter:

$$\tau \dot{s} = -s + \sum_i \delta(t - t_i) \tag{12}$$

where s is the state variable and models the current, t_i is the time of the voltage threshold crossing and $\delta(t)$ is the Dirac-delta function. From the previous equation, the current is activated when the membrane voltage of the presynaptic neuron v reaches the voltage threshold value. With equation

(12) two types of synaptic currents are possible: zero current if the membrane voltage does not reach the threshold, or discrete current if it does. In both cases the current is non-differentiable.

Modified synaptic current dynamics includes the change of the spike train part (modeled with the Dirac-delta sum in equation (12)), with the non-negative function $g(v) \geq 0$ with the unit integral ($\int g(v)dv = 1$):

$$\tau \dot{s} = -s + g(v)\dot{v} \tag{13}$$

with $\dot{v}$ being the time derivative of the presynaptic voltage.

With the modified equation, threshold crossing responses are differentiable and in all cases we have a constant charge $\int sdt = 1$, while we have differentiable graded responses and smaller amount of charge $0 \leq \int sdt < 1$ if the voltage is in the active region. Neural dynamics should be also differentiable and is defined as:

$$\dot{v} = f(v, I) \tag{14}$$

Then the dynamics of the fully-connected network can be defined as:

$$\begin{aligned} \vec{I} &= W\vec{s} + U\vec{i} + \vec{I_o} \\ \vec{o} &= O\vec{s} \end{aligned} \tag{15}$$

where W is the recurrent connectivity matrix, U is the input weight matrix, i is the input signal of the network, I_o models ionic current, o is network output, and O is the readout matrix.

The cost function that is minimized is based on the penalization of the readout error and the synaptic activity:

$$l = \frac{||o - o_d||^2 + \lambda||s||^2}{2} \tag{16}$$

where o_d is the desired output and λ is the regularization parameter. SNNs defined in this way can be optimized via gradient descent, where the gradients are calculated through backpropagation-through time (BPTT) [16].

This method is using a differentiable neuron model which can spike multiple times. That is the advantage compared to our model, where the neuron has a limitation of spiking only once.

Authors in [15] state that the previously defined method is the first method that can capture millisecond time-scale interaction, as well as second time-scale interaction between neurons. Those time-scales are usually carried out in the brain. That is demonstrated by two tasks: predictive

coding task where the information processing is in the milliseconds-scale, and delayed XOR task, where the time scale is in seconds.

3.4 Supervised learning in long-short-term memory networks (LSNNs)

In this chapter, supervised learning in long short-term memory networks of spiking neurons (LSNNs) will be discussed. The authors in [17] defined the ĹSNN structure and showed that is has similar learning capabilities as standard artificial LSTM networks. More importantly, LSNNs can be trained with the backpropagation-through time algorithm [16] for supervised learning tasks. The LSNN structure consists of a population of leaky integrate and fire (LIF) neurons that can be excitatory and inhibitory (population R), and an excitatory population of LIF neurons that are adapting (population A). Input population X of neurons provides input to the R and A populations. The network output is provided by linear readout neurons Y.

Since output spikes of LIF neurons in the LSNN structure are non-differentiable, the estimated (pseudo) gradient is used for optimization. Even with the approximation of gradients, good results have been achieved with this method. However, approximation of the gradient is the disadvantage compared to our neuron model since output spike times of our neuron model are differentiable. On the other hand, LIF and adaptive LIF neurons can spike multiple times during a simulation, and this is clearly the advantage compared to our model.

It has been demonstrated that the previous setup obtained the final test accuracy of 96% in the sequential MNIST task and this accuracy is the same as the one of artificial LSTM networks. Furthermore, the LSNN network achieved comparable performance to standard LSTM networks, in the case of the TIMIT speech recognition task.

3.5 Supervised learning in spiking neural networks using temporal coding

A new supervised learning approach in spiking neural networks using temporal coding has been presented very recently [18]. The supervised learning method in [18] has strong similarities to SpikeProp and our method described in Section 6.1: input is a temporally coded stimulus, neurons are allowed to fire only once and output spike times of neurons are differentiable. A non-leaky integrate-and-fire neuron model is used in the background:

$$\frac{du^j(t)}{dt} = \sum_i w_{ji} \sum_r \exp\left(-\frac{t - t_i^r}{\tau}\right) \Theta(t - t_i^r) \tag{17}$$

where $u^j(t)$ is the membrane potential of the $j-th$ neuron, w_{ji} is the synaptic weight from neuron i to neuron j, t_i^r represents the $r-th$ spike from neuron i and $\Theta(x)$ is the Heaviside step function.

It is interesting to observe that a membrane potential shape in this model is similar to the one described in our neuron model. One of the differences is the linear increase of voltage in our model compared to this model, where the voltage increases exponentially. Moreover, in our model weights are always positive, while here input weights can be positive and negative, and with negative weights, an inhibition is modeled.

A fully-connected feed-forward network of spiking neurons has been trained using the standard version of the gradient descent method in combination with the backpropagation algorithm. In this paper a cost function is defined as:

$$Cost = -\ln \frac{\exp(-t_g)}{\sum_i \exp(-t_i)} \tag{18}$$

where t_g is the target output spike time of the neuron that should spike first, while t_i are output spike times of all output neurons. Minimizing the cost defined in the previous equation will encourage the correct neuron in the output layer to spike first.

Since weights in this model can be both positive and negative, a weight cost term has been added to the cost function, that will highly penalize negative weights of the neuron. The neuron will spike if the sum of weights is larger than 1, thus weight sum cost is crucial for successful learning. They defined the weight sum cost in the following way:

$$WeightSumCost = K \sum_j \max(0, 1 - \sum_i w_{ji}) \tag{19}$$

where index j goes over the all neurons, and index i represents the neurons that form input to neuron j. Weight penalization constant is K.

With this model, promising results are obtained on the nonlinear XOR task and the MNIST image classification task. Nonlinear XOR task has been completely solvable using a SNN with one hidden layer containing 4 neurons. That is a similar setup to ours presented in Section 6.2.1. On the other hand, a high test classification accuracy of 97.2 % has been obtained using the network structure containing 800 neurons in the first hidden layer (784-800-10). A more important conclusion is that using the previously described SNN structure and learning procedure, output spike times of a trained model are sparse. For instance, on the MNIST classification task, only 3 % of hidden neurons have spiked on average before the first output neuron, whose time is used for the correct classification on the test set.

4 Memristors

4.1 General introduction

Chua et al. [19] defined the memristor (or memory resistor) as the fourth fundamental circuit element. From the mathematical point of view, in circuit systems there are four fundamental circuit variables: current i, voltage v, charge q and magnetic flux ϕ. In the Faraday's law of induction, we have the relation that induced voltage is equal to the time derivative of the magnetic flux $d\phi = vdt$. The current i and charge q are related by $dq = idt$. By looking more closely, Chua included new mathematical relations between the remaining variables to define a memristor: $d\phi = Mdq$, with M denoting the memristance.

The mathematical formulation of the memristor has interesting properties when the memristance M is not a constant value, but a function of q. In that case, the memristor is the non-linear element. The $i - v$ characteristics of such a memristive device should look like a hysteresis where one would have possibilities to reach different resistance values based on the different values of current or voltage, depending on the memristor type. Two memristor types are introduced, current controlled memristor (20) and voltage controlled memristor (21):

$$\begin{aligned} v(t) &= R(w)i(t) \\ \frac{dw}{dt} &= i(t) \end{aligned} \tag{20}$$

$$\begin{aligned} i(t) &= G(w)v(t) \\ \frac{dw}{dt} &= v(t) \end{aligned} \tag{21}$$

where w is a state variable, $R(w)$ and $G(w)$ are resistance and conductance values that depend on that state variable.

Later on a more generic version of the memristive systems has been introduced [20]. Mathematical definition of the current controlled memristive system is:

$$\begin{aligned} v(t) &= R(w,i)i(t) \\ \frac{dw}{dt} &= f(w,i) \end{aligned} \tag{22}$$

while for the voltage controlled memristive system we have:

$$\begin{aligned} i(t) &= G(w, v)v(t) \\ \frac{dw}{dt} &= f(v, w) \end{aligned} \tag{23}$$

where w can be set of the state variables, and functions f, R and G are general functions that depend not only on the set of state variables w, but also on the current i or voltage v.

To conclude, memristors are non-volatile memory devices. The term non-volatile means that the memristor remembers its last resistive state.

4.2 Memristor modeling

Mathematical modeling of memristors is important for simulations of experiments in both hardware and software. The formulation of the memristor was only a theoretical idea until 2008, when researchers in HP lab made a breakthrough and developed the first practical device [21]. The mathematical model of the HP memristor will be described in the Section 4.2.1. An improved HP mathematical model of the memristor is defined by Biolek [22]. Finally, the two mathematical models that are used throughout this thesis as background models for the memristive synapses will be discussed in Sections 4.2.3 and 4.2.4.

All novel mathematical models of memristors include a concept of the window function that should relate to the non-linear dopant drift effect in the physical devices. More detailed explanation of this effect will be presented in the following sections.

4.2.1 HP memristor

The first practical memristor [21] had the top and bottom electrode made of Platina Pt, while the middle area consisted of different Titanium Dioxide layers: TiO_{2-x} with smaller amount of the oxygen and pure TiO_2. In a TiO_{2-x} layer, oxygen vacancies are donors to electrons and they are positively charged. Applying a positive voltage on the top electrode will move the oxygen vacancies to the pure TiO_2 layer and that will increase the width of the TiO_{2-x} and decrease the width of the TiO_2 layer. On the other hand, applying a negative voltage has the opposite effect. Different widths of TiO_{2-x} and TiO_2 layers will lead to different resistive states of the device. HP memristor is schematically illustrated in Figure 5. From the previous definition, researches from HP created the following mathematical model, with $w(t)$ being the state variable:

$$M(t) = R_{on}\frac{w(t)}{D} + R_{off}\left(1 - \frac{w(t)}{D}\right)$$
$$\frac{dw}{dt} = \frac{\mu_v R_{on}}{D} i(t) \tag{24}$$

where R_{on} is the resistance of the TiO_{2-x} region, while R_{off} is the resistance of the TiO_2 region. D is the width of the device, $w(t)$ is the width of the TiO_{2-x} layer and the μ_w is the average mobility of ions.

The non-linear dopant drift effect causes the ionic speed of the boundary between the TiO_2 and TiO_{2-x} to decrease to zero [22]. To model nonlinear dopant drift behavior the appropriate window function should be included. For the HP memristor model, zero ionic transport should take place when the $w(t) = 0$ or $w(t) = D$. Zero ionic transport is equal to the transport that causes no change in the resistance. First window function that is proposed for the HP memristor [23] is following:

$$f(x) = 1 - (2x - 1)^{2p} \tag{25}$$

where $x = \frac{w(t)}{D}$ and p is a positive integer. The shape of the function is shown in Figure 6.

The window function (25) guarantees zero speed when approaching both boundaries, but once the boundary is reached, no external voltage can change its state again. In other words, the model remembers the state infinitely long.

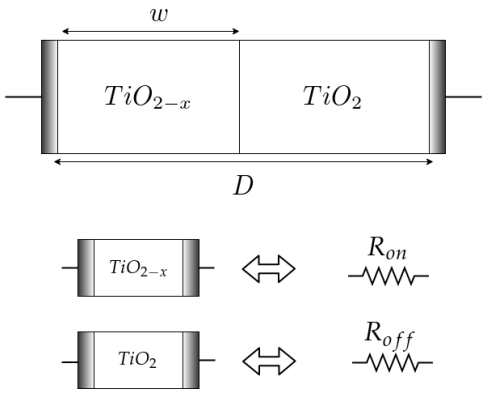


Figure 5: HP memristor structure, where w is the width of TiO_{2-x} layer and D is the width of the device.

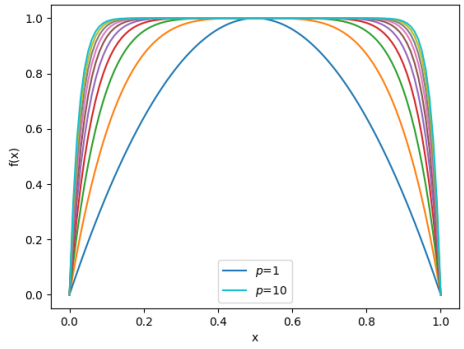


Figure 6: Window function for different values of integer p, introduced in [23].

4.2.2 Biolek model

Biolek et al. [22] proposed a new window function that correctly models the nonlinear dopant drift for the HP memristor. The main idea is that the window function should not only depend on the state variable x, but also on the current i. The sign of a current plays an essential role when going back from the boundary condition to the working range again. The new window function defined by Biolek is:

$$f(x, i) = 1 - (x - stp(-i))^{2p} \tag{26}$$

where the $x = \frac{w(t)}{D}$, and $stp(i)$ is the Heaviside step function defined as $stp(i) = 1 \quad \text{if} \quad i > 0$, otherwise $stp(i) = 0$.

Biolek's window function is shown in the following figure:

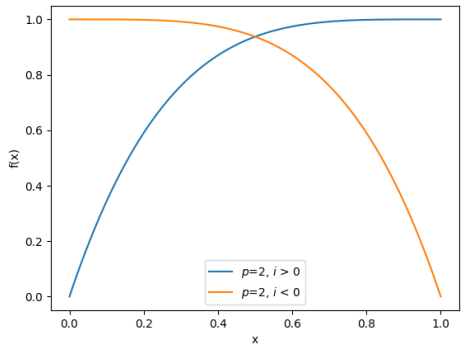


Figure 7: Biolek's window function for positive and negative current and $p = 2$.

4.2.3 A data driven verilog-A ReRAM model

The memristor model defined in this section [24] corresponds to the voltage-controlled model that exhibits bipolar switching, that is, positive and negative resistive changes. The state variable in this model is the resistance R. Authors define the model by using the sensitivity $s(v)$ and window $f(R, v)$ functions, that both have terms that depend exponentially on the voltage. The window function $f(R, v)$ bounds the state variable of the memristor to the working range $[R_{min}, R_{max}]$ based on the applied voltage v and also models the nonlinear dopant drift effect. Usually, two voltage levels make sense in memristors of a similar type. A readout voltage that is used for reading the current resistance value and is typically small, etc. $V_{readout} = 0.2$ V, and a threshold voltage that is applied to change the current resistance to smaller or larger value. In this model, all those voltage levels are captured. One of the advantages of this model is that we do not need the time derivative of the resistance to be calculated $\frac{dR}{dt}$, because it is possible to find the analytical solution of the differential equation for a constant voltage. The $I - V$ characteristics of the voltage-driven memristor model is represented with the following set of equations:

$$i(R, v) = \begin{cases} a_p \frac{1}{R} \sinh(b_p v), & \text{for} \quad v > 0 \\ a_n \frac{1}{R} \sinh(b_n v), & \text{for} \quad v < 0 \end{cases} \tag{27}$$

A sample switching rate surface which represents the time derivative of the state variable R is:

$$\frac{dR}{dt} = g(R, v) = s(v) f(R, v) \tag{28}$$

with, $s(v)$ describing the sensitivity function:

$$s(v) = \begin{cases} A_p(-1 + e^{t_p|v|}), & v > 0 \\ A_n(-1 + e^{t_n|v|}), & v < 0 \\ else \quad 0 \end{cases} \tag{29}$$

and $f(R, v)$ is the window function:

$$f(R, v) = \begin{cases} -1 + e^{\eta k_p (r_p(v) - R)}, & R < \eta r_p(v), \quad v > 0 \\ -1 + e^{\eta k_n (R - r_n(v))}, & R > \eta r_n(v), \quad v < 0 \\ else \quad 0 \end{cases} \tag{30}$$

where $a_{p,n}$, $b_{p,n}$, $A_{p,n}$, $t_{p,n}$ and $k_{p,n}$ are parameters that can be fit. Resistance values depend on the voltage polarity:

$$\begin{aligned} r_p(v) &= r_{p0} + r_{p1} v \quad \text{if} \quad v > 0 \\ r_n(v) &= r_{n0} + r_{n1} v \quad \text{if} \quad v \leq 0 \end{aligned} \tag{31}$$

with $r_{p0}, r_{p1}, r_{n0}, r_{n1}$ being additional fitting parameters. The above equations bound the resistance change of the memristor to $[R_{min}, R_{max}]$. The parameter η is equal to 1, if the positive voltage $V_b > 0$ induces a positive resistance change $\Delta R(V_b) > 0$, while the η is -1, if the positive voltage $V_b > 0$ induces the negative resistance change $\Delta R(V_b) < 0$. The analytical solution of the equation (28) is:

$$R(t)|_{V_b} = \frac{ln(e^{\eta k_p r_p(V_b)} + e^{-\eta k_p s_p}(e^{\eta k_p R_0} - e^{\eta k_p r_p(V_b) t}))}{k_p}, \quad \text{for} \quad V_b > 0, R < \eta r_p(V_b) \tag{32}$$

$$R(t)|_{V_b} = \frac{ln(-e^{\eta k_n R_0 + \eta k_n s_n(V_b) t} - e^{-\eta k_n r_n(V_b)}(-1 + e^{\eta k_n s_n(V_b) t})}{k_n}, \quad \text{for} \quad V_b < 0, R > \eta r_n(V_b) \tag{33}$$

Parameters of one memristor model [24] fitted to real measurement data are shown in Table 1.

Parameter	Value
A_p	0.12
A_n	-79.03
t_p	0.59
t_n	1.12
k_p	$8.10 \cdot 10^{-3}$
k_n	$9.43 \cdot 10^{-3}$
r_{p0}	3085
r_{p1}	1862
r_{p2}	0
r_{n0}	5193
r_{n1}	378
r_{n2}	0
a_p	0.24
b_p	2.81

Table 1: Memristor model parameters taken from [24].

From parameters in the previous table we can plot resistance changes and IV characteristics of the memristor model for different voltage values:

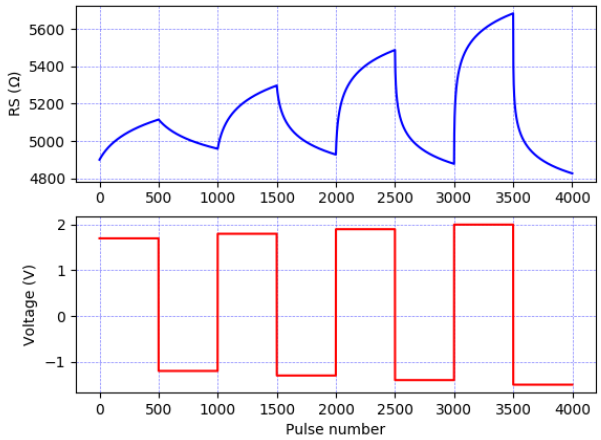


Figure 8: Changes in a resistance value for the memristor model given in Table 1. The initial resistance value was set to $R_0 = 4.9k\Omega$. We applied 500 pulses of positive and negative voltages to the memristor. The amplitude of voltage pulses was increased/decreased every 500 pulses, causing the resistance to further increase or decrease. Starting values of positive and negative voltage levels were V_{startp}=1.7V and V_{startn} = -1.2V, while the increase/decrease step was V_{step} =0.1V.

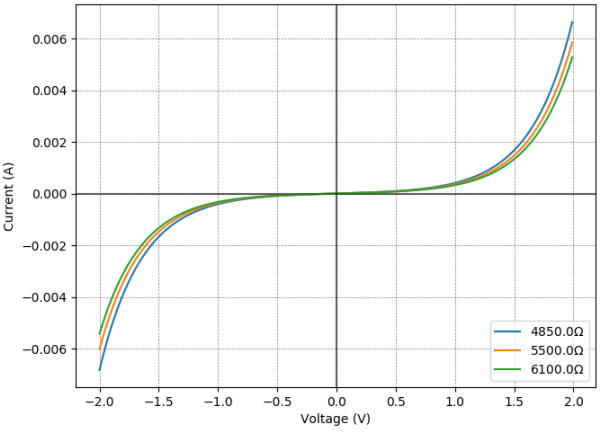


Figure 9: IV characteristics for the memristor model given in Table 1 for different resistance values.

4.2.4 A compact verilog-A ReRAM model

The model presented in this paragraph is similar to the one described in the section above. Authors in [25] modeled the voltage controlled memristor and defined the experimental procedures of fitting the data from a real memristor to a modeled one. However, it differs in the formulation of the mathematical model. The switching rate function is defined as following:

$$\begin{aligned} \frac{dR}{dt} = g(R,v) = s(v)f(R,r(v)) = \\ A_p(-1+e^{\frac{|v|}{t_p}})(r_p(v)-R)^2 stp(r_p(v)-R)stp(v)+ \\ A_n(-1+e^{\frac{|v|}{t_n}})(R-r_n(v))^2 stp(R-r_n(v))stp(-v) \end{aligned} \tag{34}$$

with t_p, t_n being the fitting parameters, and $stp(v)$ being the Heaviside step function.

The sensitivity function $s(v)$ for the positive and negative voltages v is defined as:

$$s(v) = A_p(-1+e^{\frac{|v|}{t_p}})stp(v) + A_n(-1+e^{\frac{|v|}{t_n}})stp(-v) \tag{35}$$

Subsequently, the expression that limits the working range of the memristor to $[R_{min}, R_{max}]$ is modeled as:

$$r(v) = \begin{cases} r_p(v) = a_0 + a_1 v, & v > 0 \\ r_n(v) = b_0 + b_1 v, & v \leq 0 \end{cases} \tag{36}$$

with a_0, a_1, b_0, b_1 being fitting parameters.

The voltage threshold values which are used to change the current resistive state are also taken into account in this model and amount to:

$$V_t(R) = \begin{cases} V_{tp}(R) = (R-a_0)/a_1, & v > 0 \\ V_{tn}(R) = (R-b_0)/b_1, & v < 0 \end{cases} \tag{37}$$

The model described in this section is used for the experiments with real memristive synaptic weights as explained in Section 6.3. Parameters of one memristor model fitted to real measurement data during the preparation of experiments with real memristive synaptic weights are shown in Table 2.

Parameter	Value
A_p	0.197
A_n	-0.126
t_p	1.731
t_n	1.731
a_{0p}	2731.854
a_{0n}	6568.330
a_{1p}	3393.513
a_{1n}	636.491

Table 2: Memristor model parameters which were extracted in the preparation phase of experiments with real memristive synaptic weights (summarized in Section 6.3).

From parameters in the previous table we can plot resistance changes of the memristor model for different voltage values:

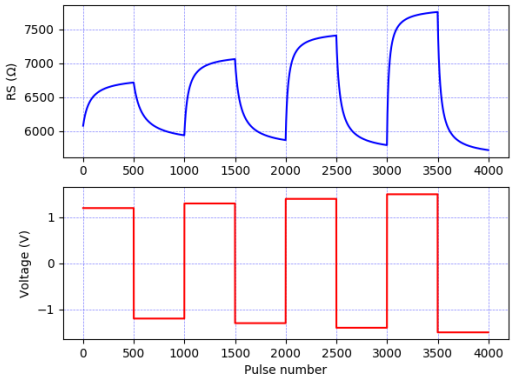


Figure 10: Changes in a resistance value for the memristor model given in Table 2. The initial resistance value was set to $R_0 = 6k\Omega$. We applied 500 pulses of positive and negative voltages to the memristor. The amplitude of voltage pulses was increased/decreased every 500 pulses, causing the resistance to further increase or decrease. Starting values of positive and negative voltage levels were V_{startp}=1.2V and V_{startn} = -1.2V, while the increase/decrease step was V_{step} =0.1V.

5 ANNs/SNNs with memristors

After the memristor was developed in 2008, researchers from many application areas have discussed and proven its potential in several fields. The memristor is an analog nano-device that has non-volatile memory. Also, it is a device with low power consumption, which makes it suitable for neuromorphic hardware applications. In neuromorphic applications, SNNs containing a vast amount of neurons and synaptic connections are implemented directly on hardware with power consumption as low as possible. Even though the memristor concept has a lot of advantages, there are also several disadvantages and complications. For example, a physical process behind the memristive devices is not yet completely understood, leading to fabricated devices that have entirely different characteristics and short life. Currently, using ANNs/SNNs with memristive weights is usually done by mapping the real weight value to conductance/resistance of the memristor. That concept, although straightforward, is limited because conductance/resistance values can saturate and lead to no changes in weights afterward. Another problem is that each memristive device must be controlled individually because of the variability of voltage threshold levels and resistance levels. Further discussion of problems in the ANNs with memristive synapses are discussed in Section 5.1. Unsupervised learning in SNN with STDP and memristive synapses is depicted in Section 5.2.

5.1 Supervised learning with hardware memristive synapses

Researchers from IBM analyzed in [26] the training of a deep ANN with 165.000 hardware memristive synapses on the MNIST classification task. In their setup, a 3-layer ANN was used, and each synaptic weight was modeled with two hardware memristive devices ($w_{ij} = G_+ - G_-$). They considered ANNs as a non-Von Neumann architecture where data and processing are not separated, thus, an ANN is created as the crossbar array of memristive devices each containing one selector device, through which neurons can communicate. The network is trained using a modified and hardware friendly backpropagation algorithm. In crossbar architectures like this, the current I is proportional to the input signal x_i (voltage in our case) and conductance value G. Because of that the forward pass is parallel and requires the comparison of the positive and negative currents.

The main problems that cause the degraded performance of such a network are following: non-linearity of the conductance change, varying working range of conductance values, the asymmetry between the increase/decrease of weights and low or high conductance values that result in non-responsive devices [26]. It is also noted that bounding conductance values in some

range degrades the performance slightly. In the previous setup, individual memristors are controlled by software. To correctly derive the backpropagation algorithm in the software, mathematical models of memristors are used in the background and gradients are mapped to the respective conductance changes. Therefore, learning is performed by applying the parallel pulse trains to reach the new conductance values suggested by the optimization algorithm.

In our learning setup with real memristive synapses, discussed in Section 6.3, we are not using a crossbar array structure, but the learning process is structured similarly like in this work.

Final test accuracy obtained for such a network on the MNIST task was 82.9%. Results derived in the IBM lab are promising, since they not only tackle a lot of problems related to memristive synapses, but also because results of the more extensive and scalable memristive architecture are presented. All results published before the IBM work contained less than 100 hardware memristive synapses.

5.2 Unsupervised learning with STDP

Significant results have been also achieved in unsupervised learning experiments with hardware memristive synapses [27]. The memristive devices are suitable for the spike-timing-dependent plasticity (STDP) learning since their dynamic takes similar form. If the synaptic weight is encoded as the memristive conductance, long-term potentiation (LTP) and long-term depression (LTD) can be obtained easily using the superposition of the different voltage shapes [28].

The memristive dynamics is suitable for the unsupervised learning tasks since a change in conductance for both LTD and LTP events converge to the unique equilibrium points: large conductance and small conductance values for LTP and LTD respectively. Those values represent the upper boundary conductance and the lower boundary conductance of the memristor's operating range. If LTP and LTD events are modeled with probabilities p and $1-p$ respectively and represent a input to one memristive device, a conductance will converge to the unique equilibrium point. For example, if we have 95% of LTP events and only 5% of the LTD events, the conductance would converge close to the upper conductance boundary. These ideas are experimentally demonstrated in [27], where it has been shown that the memristive device as the dynamical system can encode the conditional probabilities $p(PRE|POST = 1)$. Conditional probability $p(PRE|POST = 1)$ represent a probability that the presynaptic neuron fires given that the postsynaptic neuron has fired.

It was also demonstrated that a binary clustering task could be solved with a probabilistic neural network in the form of a winner-take-all (WTA) network [8]. The WTA network structure consisted of two spiking neurons

that both had four inputs. Inputs of the one neuron were fully given by four hardware memristive synapses, while for the other neuron, four software based memristive synapses were used. Software based memristive synapses had similar properties as hardware ones. Hardware and software synaptic weights were mapped to the conductance values of the memristive devices using the equation $w_{ij} = \alpha(G_{ij} - G_c)$ where α is the scaling term, G_{ij} is the current conductance value of a device and G_c is a bias term. That is exactly the same principle as used in our SNN with hardware memristive synapses in 6.3.

As an input, 1200 four-bit patterns $\mathbf{y} = (y_0, y_1, y_2, y_3)$ were presented. Base parts of input patterns were 1001 and 0110, and with a probability of 10% one bit in the input is switched. In the WTA networks, the probability $p_i(\mathbf{y}, t)$ that the neuron i wins the competition and spikes at event t is:

$$p_i(\mathbf{y}, t) = \frac{e^{U_i(\mathbf{y},t)}}{\sum_j e^{U_j(\mathbf{y},t)}} \tag{38}$$

The neuron's membrane potential values were updated using:

$$U_i(\mathbf{y}, t) = \theta_i(t) + \mathbf{w}_i(t) \cdot \mathbf{y}(t) \tag{39}$$

where $U_i(\mathbf{y}, t)$ is the membrane potential of neuron i, $\theta_i(t)$ is a homeostatic constant (bias term) that regulates the firing activity of the neuron and $\mathbf{w}_i$ is a weight vector from inputs $\mathbf{y}$ to neuron i. The dot indicates the dot product. The homeostatic plasticity term $\theta_i(t)$ makes sure that both neurons compete equally in the WTA network and fire similarly often on average. That is highly important for robust learning.

It was shown that the WTA network completely solved the binary clustering task, with one neuron specializing to one pattern (for instance 1001), whereas the other neuron concentrated on the opposite one (for example 0110). Also, the network robustly handled the noise that was included in the input data. A forgetting and re-learning possibility was also demonstrated with such a WTA network, where weights of one neuron have been intentionally changed so that neuron must re-learn the specific pattern.

6 Experimental results

This chapter describes a new neuron model that enables exact gradient computation. Our neuron model and algorithm have similarities with the SpikeProp method [11] and one very recent method for supervised learning in SNNs which uses temporal coding [18]: input to our neuron is a temporally coded stimulus and one neuron can fire exactly once during the time course of the SNN simulation.

We will show that our spiking neuron model can be trained to solve logical functions such as AND, OR and NAND. With a network of such spiking neurons, the non-linear XOR function can be learned. Similar results are also obtained for spiking neuron models and memristive synapses in simulation and those results can be found in Section 6.2. The learning capabilities of our spiking neuron with real memristive synapses is discussed in Section 6.3. In that setup, a degraded performance is observed due to the variability and noise in memristive devices. In Sections 6.5 and 6.6 it is shown that a deep network of our spiking neurons achieved comparable performance to artificial neural networks of the similar structure, when solving the Iris and MNIST classification tasks.

6.1 New neuron model

Our neuron model exploits spike timing for the computation. Temporally coded excitatory and inhibitory spike times are forming an input to our neuron. Each excitatory input has a corresponding excitatory weight, and the same applies to the inhibitory input. The combination of input spiking times and weights will give us a membrane "voltage" value $V(t)$ of the neuron. As each neuron spikes only once, we define the output spike time t_{sp} of the neuron as time when $V(t)$ reaches some predefined threshold value ϑ. We define our neuron model with the following equations:

$$E(t) = \sum_{i:t_i^E<t} w_i^E \tag{40}$$

$$I(t) = \sum_{j:t_j^I<t} w_j^I \tag{41}$$

$$\dot{V}(t) = \frac{E(t)}{I(t)+1} + b \tag{42}$$

where:

- t_i^E - represents the input spike time of an excitatory neuron i,
- t_j^I - represents the input spike time of an inhibitory neuron j,

- w_i^E - represents the weight from an excitatory neuron i,
- w_j^I - represents the weight from an inhibitory neuron j, and
- b - represents the bias.

Graphical representation of our neuron model together with changes of $E(t)$, $I(t)$ and $V(t)$ values is shown in Figure 11.

All weights in our neuron model are always positive, and they cannot change the timing of an input spike. From equations (40), (41) and (42) it is clear that V value increases linearly between two input spikes. That can also be observed in Figure 12, where we can see different changes in membrane voltage for different excitatory and inhibitory weights and input spike times.

We can compute a voltage value $V(t_j)$ at spike time t_j from the following equation:

$$V(t_j) = \sum_{i=1}^{j} (t_i - t_{i-1}) \dot{V}(t_i^-) \tag{43}$$

where:

- $\dot{V}(t_i^-)$ - represents the $\dot{V}$ value just before the $i-th$ input spike.

Then, the V value in the ϵ surrounding of t_j (for small ϵ) is:

$$V(t_j + \epsilon) = V(t_j) + \dot{V}(t_j^+)\epsilon \tag{44}$$

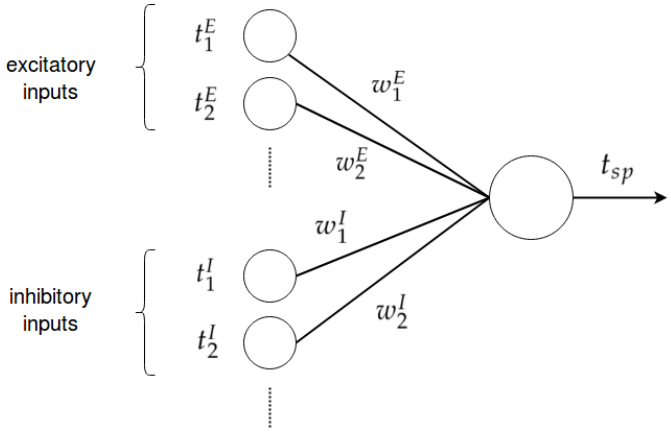
Using the previous equation we can investigate the threshold condition and compute a time of crossing the threshold t_{sp}:

$$\begin{aligned} V(\hat{t} + \Delta t) = V(\hat{t}) + \dot{V}(\hat{t}^+)\Delta t \Rightarrow \\ V(\hat{t}) + \dot{V}(\hat{t}^+)(t_{sp} - \hat{t}) \stackrel{!}{=} \vartheta \end{aligned} \tag{45}$$

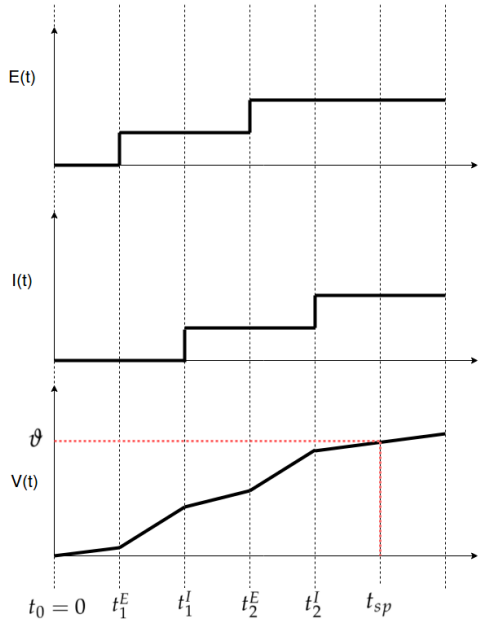
$$t_{sp} = \hat{t} + \frac{\vartheta - V(\hat{t})}{\dot{V}(\hat{t}^+)} \tag{46}$$

where:

- $\hat{t}$ - represents the last input spike time before the threshold is crossed, and
- t_{sp} - represents the output spiking time of the neuron (time of threshold crossing).

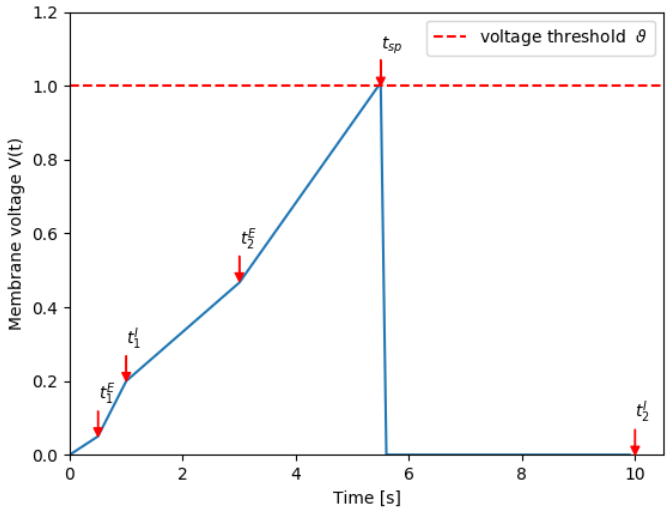


(a) Graphical representation of the neuron model. The neuron receives excitatory (t_1^E, t_2^E, ...) and inhibitory (t_1^I, t_2^I, ...) spike times from input neurons. Additionally, every excitatory input has an excitatory (w_1^E, w_2^E, ...) weight and every inhibitory input has an inhibitory (w_1^I, w_2^I, ...) weight. Each neuron spikes only once and this time represents the output spike time of the neuron t_{sp}.

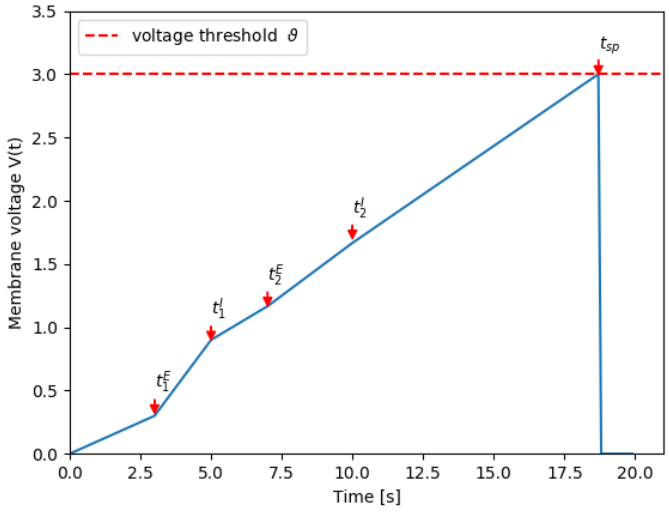


(b) One possible example of $E(t)$, $I(t)$ and $V(t)$ changes over time given input excitatory spike times t_1^E, t_2^E, input inhibitory spike times t_1^I, t_2^I and threshold value ϑ. As it can be seen from the figure, spiking time of the neuron t_{sp} is defined as the time when voltage V(t) crosses the threshold ϑ.

Figure 11: Graphical representation of the neuron model and one example of evolution of $E(t)$, $I(t)$ and $V(t)$ values over time.



(a) Change in membrane voltage $V(t)$ over time for one neuron with two excitatory inputs ($t^E \in \{0.5, 3\}$) with $w^E \in \{0.2, 0.5\}$) and two inhibitory inputs ($t^E \in \{1, 10\}$) with $w^I \in \{5, 0.5\}$). Neuron threshold is set to $\vartheta = 1$. Neuron spikes at $t_{sp} = 5.5$. The last inhibitory spike t_2^I comes much later, and does not have any contribution to the spike condition.



(b) Change in membrane voltage $V(t)$ over time for one neuron with two excitatory ($t^E \in \{3, 7\}$ with $w^E \in \{0.2, 0.2\}$) and two inhibitory ($t^I \in \{5, 10\}$ with $w^I \in \{5, 1.5\}$) inputs. Neuron threshold is set to $\vartheta = 3$. The neuron spikes at $t_{sp} = 18.2$ and all input spike times contribute to the spike generation.

Figure 12: Changes in membrane voltage $V(t)$ over time.

Derivatives of the output spike time t_{sp} of the $i-th$ neuron w.r.t excitatory/inhibitory weights w_i^E/w_i^I are:

$$\frac{\partial t_{sp}}{\partial w_i^*} = \frac{-\frac{\partial V(\hat{t})}{\partial w_i^*}\dot{V}(\hat{t}^+) - \frac{\partial \dot{V}(\hat{t}^+)}{\partial w_i^*}(\vartheta - V(\hat{t}))}{\dot{V}(\hat{t}^+)^2} \tag{47}$$

where $*$ describes excitatory E or inhibitory I weights and:

$$\frac{\partial \dot{V}(\hat{t}^+)}{\partial w_i^E} = \frac{1}{I(\hat{t}^+)+1} \qquad t_i^E \leq \hat{t} \tag{48}$$

$$\frac{\partial \dot{V}(\hat{t}^+)}{\partial w_i^I} = -\frac{E(\hat{t}^+)}{(I(\hat{t}^+)+1)^2} \qquad t_i^I \leq \hat{t} \tag{49}$$

and:

$$V(\hat{t}) = \sum_{i=1}^{idx_{\hat{t}}} (t_i - t_{i-1})\dot{V}(t_i^-) \tag{50}$$

$$\frac{\partial V(\hat{t})}{\partial w_i^*} = \sum_{j=1}^{idx_{\hat{t}}} (t_j - t_{j-1})\frac{\partial \dot{V}(t_j^-)}{\partial w_i^*} \tag{51}$$

where $idx_{\hat{t}}$ represents the index of last input spike time before the threshold is crossed. By inserting equations (48) and (49) into equation (51), we get:

$$\frac{\partial V(\hat{t})}{\partial w_i^E} = \sum_{j=idx_{t_i^E}}^{idx_{\hat{t}}} (t_j - t_{j-1})\frac{1}{I(\hat{t_j}^-)+1} \tag{52}$$

$$\frac{\partial V(\hat{t})}{\partial w_i^I} = -\sum_{j=idx_{t_i^I}}^{idx_{\hat{t}}} (t_j - t_{j-1})\frac{E(\hat{t_j}^-)}{(I(\hat{t_j}^-)+1)^2} \tag{53}$$

Derivatives of the output spike time t_{sp} of $i-th$ neuron w.r.t the neuron bias b_i are:

$$\frac{\partial t_{out}}{\partial b_i} = \frac{-\frac{\partial V(\hat{t})}{\partial b_i}\dot{V}(\hat{t}^+) - \frac{\partial \dot{V}(\hat{t}^+)}{\partial b_i}(\vartheta - V(\hat{t}))}{\dot{V}(\hat{t}^+)^2} \tag{54}$$

where from equation (42) we have:

$$\frac{\partial \dot{V}(\hat{t}^+)}{\partial b_i} = 1 \tag{55}$$

and:

$$V(\hat{t}) = \sum_{i=1}^{idx_{\hat{t}}} (t_i - t_{i-1}) \dot{V}(t_i^-) \tag{56}$$

$$\frac{\partial V(\hat{t})}{\partial b_i} = \sum_{j=1}^{idx_{\hat{t}}} (t_j - t_{j-1}) \frac{\partial \dot{V}(t_j^-)}{\partial b_i} \tag{57}$$

By inserting equation (55) into equation (57) we get:

$$\frac{\partial V(\hat{t})}{\partial b_i} = \sum_{j=idx_{t_i}}^{idx_{\hat{t}}} (t_j - t_{j-1}) \tag{58}$$

Finally, derivatives of the output spike time t_{sp} of the $i-th$ neuron w.r.t input spike times of the same neuron t_k are:

$$\frac{\partial t_{sp}}{\partial t_k} = \frac{\partial t_{sp}}{\partial V(\hat{t})} \frac{\partial V(\hat{t})}{\partial t_k} \tag{59}$$

with:

$$\frac{\partial t_{sp}}{\partial V(\hat{t})} = -\frac{1}{\dot{V}(t_{sp})} \tag{60}$$

and using equation (56) we have that:

$$\frac{\partial V(\hat{t})}{\partial t_k} = \dot{V}(t_k^-) - \dot{V}(t_{k+1}^-) = \dot{V}(t_k^-) - \dot{V}(t_k^+) \tag{61}$$

which means that:

$$\frac{\partial t_{sp}}{\partial t_k} = \frac{\dot{V}(t_k^+) - \dot{V}(t_k^-)}{\dot{V}(t_{sp})} \tag{62}$$

Notations used for the backpropagation algorithm in a fully-connected spiking neural network with one hidden layer are shown in the Figure 13.

Derivatives of the chosen loss function w.r.t weights and bias values in hidden and output layers are computed using chain rules:

$$\frac{\partial Loss}{\partial w_{jk}^{o*}} = \frac{\partial Loss}{\partial t_k} \frac{\partial t_k}{\partial w_{jk}^{o*}} \tag{63}$$

$$\frac{\partial Loss}{\partial b_{jk}^{o*}} = \frac{\partial Loss}{\partial t_k} \frac{\partial t_k}{\partial b_{jk}^{o*}} \tag{64}$$

$$\frac{\partial Loss}{\partial w_{ij}^{1*}} = \frac{\partial Loss}{\partial t_k} \frac{\partial t_k}{\partial t_j} \frac{\partial t_j}{\partial w_{ij}^{1*}} \tag{65}$$

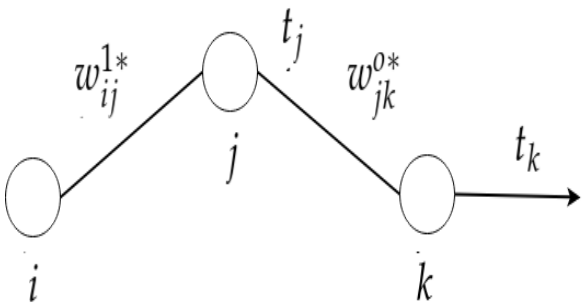


Figure 13: Part of the spiking neural network setup with one hidden layer (i - neuron in the input layer, j - neuron in the hidden layer and k - neuron in the output layer).

$$\frac{\partial Loss}{\partial b_{ij}^{1*}} = \frac{\partial Loss}{\partial t_k}\frac{\partial t_k}{\partial t_j}\frac{\partial t_j}{\partial b_{ij}^{1*}} \tag{66}$$

6.2 Learning logical functions with non-memristive and modeled memristive synapses

Learning logical functions (AND, OR, XOR and NAND) with our neuron model required encoding binary inputs and outputs into input and output spike times. We used a temporal coding principle: the more intensive an input is, the earlier spike transmission happens. Hence, input binary values are encoded with early and late input spike times, where the 0 input binary value is defined as the late input spike time (t_{late}), and the 1 input binary value is defined as the early input spike time (t_{early}). Output binary values are encoded based on the same principle, but different values are chosen for early and late output spike times. Late output spike time ($t_{sp_{late}}$) corresponds again to the output binary value of 0, and the early output spike time ($t_{sp_{early}}$) corresponds to the output binary value of 1.

Chosen early and late input spike times (t_{early} and t_{late}), early and late output spike times ($t_{sp_{early}}$ and $t_{sp_{late}}$) and threshold value (ϑ) for each neuron are presented in the following table:

t_{early} [s]	t_{late} [s]	$t_{sp_{early}}$ [s]	$t_{sp_{late}}$ [s]	ϑ
1.5	3	4	5	1

Table 3: Chosen initial input and output spike time values.

One of the challenges was the presentation of input spike times to the neuron. The initial idea was to present input spike times simply as excitatory inputs (as depicted in Figure 14). However, this cannot handle more complex logical mappings (for instance in the XOR and NAND functions). In case of NAND and its logical table, a late spike in both inputs should give the result of an early output spike time ($t_{sp_{early}}$). On the other hand, early input spike times should give the result of a late output spike time ($t_{sp_{late}}$). If we consider only excitatory inputs and the mathematical model of the neuron, we can conclude that it is not possible to obtain those two conditions defined by the NAND logical function. For this reason, more complex input representations are defined in Figure 15. That improved the final result by including both excitatory, inhibitory, inverted excitatory ($invx_i^E$ which is defined as a time-inverted version of the x_i^E) and inverted inhibitory inputs ($invx_i^I$ which is defined as a time-inverted version of the x_i^I).

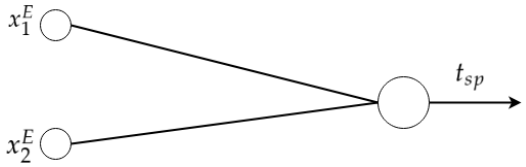
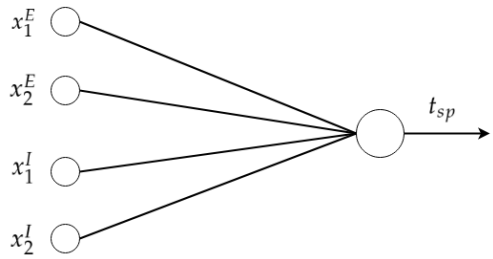


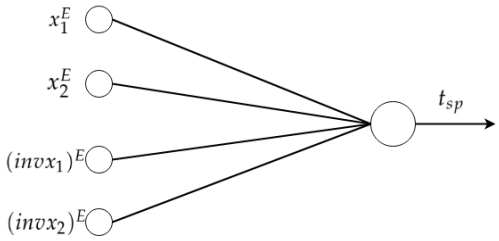
Figure 14: Input representation containing two excitatory values (in further text - **simple**).

We created regression tasks to obtain exact early and late output spike times based on input spike times with a certain precision, for each logical function. As objective function, we introduced the modified mean-squared error (MMSE) loss function as a slight modification of the mean-squared error (MSE) loss. The main difference between the MSE and MMSE is that in the MMSE loss function we include regions that will have zero loss and gradients to improve our final result further. Those regions are based on desired output spike times (t_i) and actual output spike times (tsp_i):

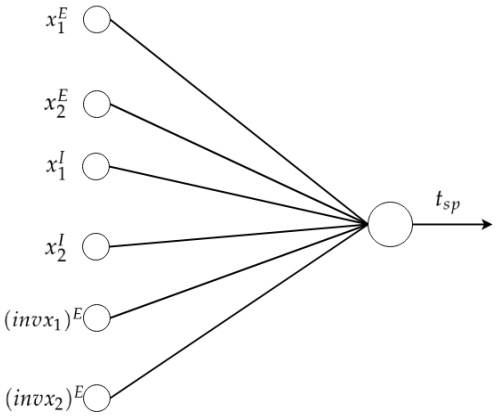
$$MMSE = \begin{cases} 0 & \text{if } t_i = t_{sp_{late}} \quad \text{and} \quad tsp_i \geq t_i \\ 0 & \text{if } t_i = t_{sp_{early}} \quad \text{and} \quad tsp_i \leq t_i \\ \frac{1}{n}(t_i - tsp_i)^2 & \text{otherwise} \end{cases} \tag{67}$$



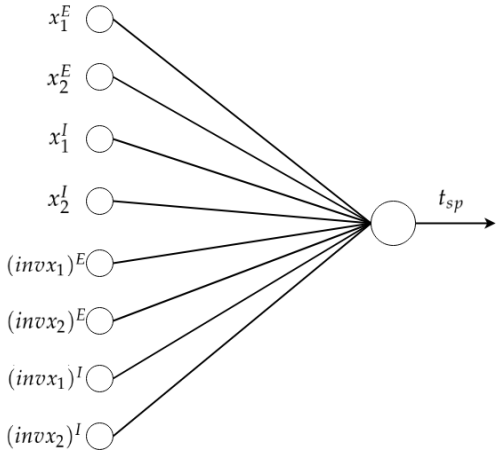
(a) Input representation containing two excitatory and two inhibitory values (in further text - **basic**).



(b) Input representation containing two excitatory and two inverted excitatory values (in further text - **time-inverted**).



(c) Input representation containing two excitatory, two inhibitory and two inverted excitatory values (in further text - **excitatory - inhibitory**).



(d) Most complex input representation, containing two excitatory, two inhibitory, two inverted excitatory and two inverted inhibitory inputs (in further text - **full expansion**).

Figure 15: More complex input representations.

6.2.1 Learning logical functions with non-memristive synapses

Single neuron learning. The modified-mean-squared error [MMSE] loss function was minimized for learning simple logical functions with single neuron. All results were obtained with the Gradient Descent Optimizer and learning rate $\alpha = 0.001$. Batch learning was applied with a batch size of four. Input dimensionality varied from only two inputs (basic setup) to twelve inputs (full expansion setup). The number of epochs was set to 5000, and weight clipping to positive weights was also applied.

Final results of the single neuron learning are depicted in the following table:

Task	Input encoding	Loss	Misclassification
AND	simple	0.0217	0
	basic	0.0362	0
	time-inverted	0.0217	0
	excitatory-inhibitory	0.0495	0
	full expansion	$8.86 \cdot 10^{-7}$	0
OR	simple	0.0338	0
	basic	0.0075	0
	time-inverted	0.0179	0
	excitatory-inhibitory	0.0014	0
	full expansion	0.0083	0
XOR	simple	0.2554	2
	basic	0.2371	1
	time-inverted	0.2500	2
	excitatory-inhibitory	0.2220	2
	full expansion	0.2267	1
NAND	simple	0.1890	1
	basic	0.1890	1
	time-inverted	0.0179	0
	excitatory-inhibitory	0.0024	0
	full expansion	$5.02 \cdot 10^{-5}$	0

Table 4: Results of the single neuron learning.

In the previous table, final loss and misclassification results are shown for each logical task (AND, OR, XOR and NAND) and for each input encoding

that is defined (simple, basic, time-inverted, excitatory-inhibitory and full expansion). Final loss values represent final modified-mean squared error values for all tasks. To count the number of misclassified examples, we included a threshold value of $\Theta = 4.5$s and based on that value we classified results into binary values (0's and 1's). In one simulation run, actual output spike times for the AND logical function were $[5.023, 4.751, 4.751, 3.861]$s. With the consideration of the previously defined threshold value $\Theta = 4.5$s, we will map each value from the final result to 0 if that value is larger than Θ. Otherwise, the value will be mapped to 1. In this example, the final mapped result will be $[0, 0, 0, 1]$. The number of misclassified examples from the mapped result are counted, and that result is depicted in the last column. The previous result is the same one we would get for the AND logical function, so the number of misclassified examples is 0.

From results shown in Table 4, we can conclude that simple logical functions such as AND and OR were solvable for every input encoding. That was not the case for more complex logical functions such as XOR and NAND. XOR logical function was not solvable in any case, while the NAND logical function was solvable for **time-inverted**, **excitatory-inhibitory** and **full expansion** encodings.

Spiking neural network learning. A spiking neural network was created with the following structure:

- number of input neurons: varied based on a setup
- number of hidden neurons: 4
- number of output neurons: 1

The SNN was implemented as a feed-forward fully-connected network, using the PyTorch [29] library. The forward pass through the network was done, where all output spiking times in all layers were computed. In this step, all the dynamic computational graphs needed for the gradient computation were constructed. Gradients of the MMSE loss w.r.t weights were computed in the subsequent step. In the end, we updated weights using the Gradient Descent Algorithm and learning rate $\alpha = 0.001$. In this setup, batch learning with a batch size of four was performed.

The SNN was trained for 15000 epochs, and weight clipping to allow only positive weights was applied. The same number of excitatory (two in our case) and inhibitory (two in our case) output spike times of the hidden layer formed the input to neurons in the output layer.

Final results of the SNN applied to the logical functions are presented in the following table:

Task	Input encoding	Loss	Misclassification
AND	simple	0.0217	0
	basic	0.0131	0
	time-inverted	0.0049	0
	excitatory-inhibitory	$2.57 \cdot 10^{-5}$	0
	full expansion	$2.37 \cdot 10^{-23}$	0
OR	simple	0.0033	0
	basic	0.0005	0
	time-inverted	0.00164	0
	excitatory-inhibitory	0.00445	0
	full expansion	$5.41 \cdot 10^{-15}$	0
XOR	simple	0.0478	0
	basic	$7.72 \cdot 10^{-5}$	0
	time-inverted	0.0794	0
	excitatory-inhibitory	0.0120	0
	full expansion	$4.08 \cdot 10^{-10}$	0
NAND	simple	$5.66 \cdot 10^{-5}$	0
	basic	0.0009	0
	time-inverted	0.00015	0
	excitatory-inhibitory	$4.9 \cdot 10^{-11}$	0
	full expansion	$1.57 \cdot 10^{-29}$	0

Table 5: Results of the spiking neural network.

If we look into the results obtained from the spiking neural network, we can conclude that all logical functions were solvable for every input encoding. As expected, results of the SNN are in general better compared to the results obtained with single neuron and yield to smaller final loss values. The lowest loss value is obtained for all logical functions with the **full expansion** input representation.

6.2.2 Learning logical functions with modeled memristive synapses

We have also defined a neuron model with modeled memristive synapses. In that case, we defined excitatory and inhibitory weights as:

$$w_{ex,inh} = \alpha_a a_p (G_{ex,inh} - G_c) \tag{68}$$

where:

- α_a - represents the scaling parameter of memristive weights,
- a_p - represents the parameter defined in the mathematical model of the memristor (given in Table 6),
- $G_{ex,inh}$ - represents the conductance of the memristor, and,
- G_c - represents the bias conductance value. We chose G_c such that our memristive weights were positive and in reasonable range.

Parameter	Value
A_p	0.12
A_n	-79.03
t_p	0.59
t_n	1.12
k_p	$8.10 \cdot 10^{-3}$
k_n	$9.43 \cdot 10^{-3}$
r_{p0}	3085
r_{p1}	1862
r_{p2}	0
r_{n0}	5193
r_{n1}	378
r_{n2}	0
a_p	0.24
b_p	2.81

Table 6: Memristor model parameters taken from [24].

Initial memristive setup. Parameters defined in Table 6 were used for all our memristive weights in different setups outlined below. Those parameters are defined in [24] and represent the experimental fit of the memristor model defined in Section 4.2.3. The resistance of each memristive weight was defined randomly, and the G_c was always defined such that initial weights are positive. For the objective function, MMSE was chosen.

Single neuron learning. In single neuron learning with memristive synapses, batch learning with a batch size of four and weight clipping to positive weights were applied. Chosen hyperparameters for learning simple logical functions with such a setup are shown in Table 7.

α_a	G_{init} (ex. or inh.)	G_c	Optimizer	Learning rate	Epochs
80000	$\frac{1}{rand(4800,5000)}$	$\frac{1}{5500}$	Gradient Descent	500	10000

Table 7: Hyperparameters of the single neuron learning with modeled memristive synapses.

Final results of the single neuron learning with modeled memristive synapses can be found in Table 8.

Task	Input encoding	Loss	Misclassification
AND	simple	0.0217	0
	basic	0.0312	0
	time-inverted	0.0217	0
	excitatory-inhibitory	0.0689	0
	full expansion	$2.99 \cdot 10^{-12}$	0
OR	simple	0.0286	0
	basic	0.0044	0
	time-inverted	0.0179	0
	excitatory-inhibitory	0.00011	0
	full expansion	0.0005	0
XOR	simple	0.2554	2
	basic	0.207	1
	time-inverted	0.2500	2
	excitatory-inhibitory	0.210	2
	full expansion	0.246	2
NAND	simple	0.1889	1
	basic	0.357	2
	time-inverted	0.0179	0
	excitatory-inhibitory	0.00019	0
	full expansion	0.00124	0

Table 8: Results of the single neuron learning with memristive synapses.

If we look into the results obtained with single neuron containing modeled memristive synapses, we get the same conclusion as in the case with single neuron with non-memristive synapses. Results of the single neuron with memristive and with non-memristive synapses converge in most cases to the similar optimal solution.

Memristive spiking neural network learning. In this section, a memristive spiking neural network, consisting of neurons with memristive synapses was created in a feed-forward fully-connected manner. The simple memristive SNN structure used for solving logical functions was:

- number of input neurons: varied based on a setup
- number of hidden neurons: 4
- number of output neurons: 1

The same number of excitatory (two in our case) and inhibitory (two in our case) output spike times of the hidden layer formed the input to neurons in the output layer.

Algorithm. To train the SNN with memristive synapses on logical functions, the algorithm was altered compared to the standard backpropagation where gradients of the loss function are always evaluated with respect to weights:

1. A spiking neural network structure with the combination of neurons described above was created.

2. a. Excitatory and inhibitory weights were created using the formula depicted in equation (68).
   b. The feed-forward pass of the network was done, where spike times of neurons in all layers were computed, including spike times of the output layer. For this step, the PyTorch library [29] was used, and dynamic computational graphs were constructed.
   c. The PyTorch built-in *autograd* functionality was used to calculate the gradients and minimize the $MMSE$ loss function. Gradients of the $MMSE$ loss function w.r.t to the resistance were computed to obtain the resistance change. From gradients, we update the resistance:

$$R_{new} = R_{old} - \eta \frac{\partial MMSE}{\partial R} \tag{69}$$

   where η is a learning rate.

3. From the mathematical models, the pulse train length needed to reach the new resistance value (R_{new}) based on the old resistance value (R_{old}) was calculated. That pulse train was applied to the memristor and after that step, it was possible to calculate new conductance G_{new} and weight w_{new} from the new resistance R_{new}:

$$\begin{aligned} G_{new} &= \frac{1}{R_{new}} \\ w_{new} &= a_p \alpha_a (G_{new} - G_c) \end{aligned} \tag{70}$$

4. Steps 2 and 3 were repeated for the predefined number of epochs.

Again we applied batch learning with a batch size of four and weight clipping to positive weights. Chosen hyper-parameters for learning simple logical functions with memristive SNN are shown in the following table:

α_a	G_{init} (ex. or inh.)	G_c	Optimizer	Learning rate	Epochs
320000	$\frac{1}{rand(4800,5000)}$	$\frac{1}{5100}$	Gradient Descent	1000	15000

Table 9: Memristive spiking neural network hyperparameters.

Final results of the memristive SNN can be found in Table 10.

Task	Input encoding	Loss	Misclassification
AND	simple	0.0217	0
	basic	$2.91 \cdot 10^{-21}$	0
	time-inverted	$1.77 \cdot 10^{-28}$	0
	excitatory-inhibitory	$1.23 \cdot 10^{-28}$	0
	full expansion	$1.65 \cdot 10^{-28}$	0
OR	simple	0.0049	0
	basic	0.0025	0
	time-inverted	$2.52 \cdot 10^{-29}$	0
	excitatory-inhibitory	$4.90 \cdot 10^{-26}$	0
	full expansion	$1.85 \cdot 10^{-29}$	0
XOR	simple	0.0119	0
	basic	$2.29 \cdot 10^{-28}$	0
	time-inverted	0.0433	0
	excitatory-inhibitory	$1.81 \cdot 10^{-9}$	0
	full expansion	$8.58 \cdot 10^{-19}$	0
NAND	simple	$5.68 \cdot 10^{-26}$	0
	basic	$6.38 \cdot 10^{-14}$	0
	time-inverted	$4.48 \cdot 10^{-06}$	0
	excitatory-inhibitory	$1.71 \cdot 10^{-25}$	0
	full expansion	$3.48 \cdot 10^{-12}$	0

Table 10: Results of the spiking neural network with memristive synapses.

From results in Table 10, we can see that all logical functions were solvable for every input encoding. Results of the memristive spiking neural network are in general better compared to the results obtained with single neuron with memristive synapses. Also, the final results of spiking neural network and memristive spiking neural network are comparable.

6.3 Learning logical functions with real memristive synapses

We tried our memristive spiking neuron model using real memristive nanodevices as synaptic weights throughout our cooperation with the Electronics and Computer Science Institute, Electronic Materials and Devices Research group of the University of Southampton. For that purpose, our code was ported to the framework called ArcOne NeuroPack that enables communication and control of real hardware devices. Before applying the actual learning on real memristive devices, parameters from devices were extracted, and mathematical models were created. The mathematical model described in the paper [25] was used in the background for the calculation of a pulse length for resistance change.

Since memristors are devices with a lot of variability and noise, the double learning idea was tested to minimize the noise effect on the final result:

Double learning. All computations (time changes, resistance changes) were done on mathematical models of memristors that are kept in a memory, while actual updates were done on real devices. When applying this idea to real devices, we anticipatedly observed a degraded performance because of:

- readout noise while reading the current resistance value.
- noise that is present in the resistance change of the real devices,
- limitation of voltage pulse train length based on the time step dt. With positive and negative voltage pulse trains we obtain resistance/-conductance changes in real memristive devices.

6.3.1 Setup

Single neuron learning with the time-inverted input encoding was tested in a simulation setup and with real hardware devices. Working ranges for defined voltage levels of memristors were extracted before each experiment with the real memristive devices. Afterwards, model fitting parameters and initial resistance values were collected. That allowed us to have full mathematical models of memristors in the background, with similar initial values as the real ones. Synaptic weights were mapped to conductance values of the memristive devices. This is similar to the setup described in Section 6.2.2. Before applying each pulse train on real devices, a voltage pulse time needed to reach the new resistive state was calculated from the mathematical model. After that, the time was converted to a pulse train with the constant time step $dt = 10^{-6}$ s. It is worth noting that at each learning step the length of a pulse train was limited to 1000 pulses. The single neuron

learning setup with time-inverted input encoding is presented in Figure 15.

Hyperparameters and mathematical models (given in Tables 21-22) used for different logical learning tasks are shown in the following tables:

Weight	Model	R_{init} [Ω]	R_{min} [Ω]	R_{max} [Ω]	R_c [Ω]	α_a
$w_{1,5}$	Model 3	15246	14500	16500	16500	119625
$w_{2,5}$	Model 4	53879	52000	56000	56000	728000
$w_{3,5}$	Model 5	5859	5700	7000	7000	37546
$w_{4,5}$	Model 6	10764	10000	11500	11500	76666

Table 11: Hyperparameters and mathematical models used in the AND task.

Weight	Model	R_{init} [Ω]	R_{min} [Ω]	R_{max} [Ω]	R_c [Ω]	α_a
$w_{1,5}$	Model 3	15845	14500	16500	16500	119625
$w_{2,5}$	Model 4	53634	52000	56000	56000	728000
$w_{3,5}$	Model 5	5763	5700	7000	7000	37546
$w_{4,5}$	Model 6	11059	10000	11500	11500	76666

Table 12: Hyperparameters and mathematical models used in the OR task.

Weight	Model	R_{init} [Ω]	R_{min} [Ω]	R_{max} [Ω]	R_c [Ω]	α_a
$w_{1,5}$	Model 3	15434	14500	16500	16500	119625
$w_{2,5}$	Model 4	53298	52000	56000	56000	728000
$w_{3,5}$	Model 5	6069	5700	7000	7000	37546
$w_{4,5}$	Model 7	9143	9000	11000	11500	49500

Table 13: Hyperparameters and mathematical models used in the XOR task.

Weight	Model	R_{init} [Ω]	R_{min} [Ω]	R_{max} [Ω]	R_c [Ω]	α_a
$w_{1,5}$	Model 3	15790	14500	16500	16500	119625
$w_{2,5}$	Model 4	53822	52000	56000	56000	728000
$w_{3,5}$	Model 5	5754	5700	7000	7000	37546
$w_{4,5}$	Model 6	10877	10000	11500	11500	76666

Table 14: Hyperparameters and mathematical models used in the NAND task.

We defined memristive weights as:

$$w_{i,j} = \alpha_a(G_{i,j} - G_c) \tag{71}$$

where:

- α_a - represents the scaling parameter defined for each memristive synapse,
- $G_{i,j}$ - represents the current conductance ($G_{i,j} = \frac{1}{R_{i,j}}$) of the memristor, and,
- G_c - represents the baseline parameter ($G_c = \frac{1}{R_c}$) defined differently for each memristive synapse.

Hyperparameters from previous tables are chosen such that all weights are scaled using the equation (71), to the range $w_{i,j} \in [0, 1]$.

Algorithm. The algorithm used for training real memristive devices is slightly different than the one explained with modeled memristive synapses:

1. Using initial resistance values from real memristive devices, weights were created using the formula depicted in equation (71). Also, background mathematical models of memristors were created with similar initial values as real ones,

2. In the feed-forward pass, we computed the spikes of output neurons. For that step, the PyTorch library [29] was used, and dynamic computational graphs were constructed. The feed-forward pass was done using weights created from real memristive devices.

3. Using the PyTorch built-in *autograd* functionality gradients were automatically collected for the minimization of the $MMSE$ loss function. Gradients of the $MMSE$ loss function w.r.t to the resistance are computed to obtain the resistance change. From gradients, the updated resistance is:
$$R_{new_{sim}} = R_{old_{sim}} - \eta \frac{\partial MMSE}{\partial R} \tag{72}$$
where:
   - η is the learning rate,
   - $R_{old_{sim}}$ is the old/previous resistance of the simulated model, and,
   - $R_{new_{sim}}$ is the new resistance that we would obtain with the simulated model.

4. From the mathematical models, the pulse train length needed to reach the new resistance value ($R_{new_{sim}}$) based on the old resistance value ($R_{old_{sim}}$) was calculated. That pulse train was applied on both simulated and real memristive devices. After that, we read out the new resistance value from the real memristive devices to calculate $G_{new_{real}} = \frac{1}{R_{new_{real}}}$. Finally, new weights were calculated based on the conductance of real memristive devices:
$$w_{new_{real}} = \alpha_a (G_{new_{real}} - G_c) \tag{73}$$

5. Steps 2, 3 and 4 were repeated for the predefined number of epochs.

6.3.2 Results

In this section, simulation results of single neuron learning with modeled and real memristive synapses are presented. All results were obtained with the Gradient Descent Optimizer with a learning rate of $\alpha = 10000$, which was used for the minimization of the MMSE function. Batch learning was

performed with a batch size of four, the number of epochs was set to 1500, and weight clipping was used to limit our weights to only positive values. To define initial weights of our simulated model, the same mathematical models and initial resistance values from Tables 11-14 were used.

Task	Input encoding	Loss	Misclassification
AND	time-inverted	0.0233	0
OR	time-inverted	0.0374	0
XOR	time-inverted	0.2500	2
NAND	time-inverted	0.0181	0

Table 15: Results of single neuron learning with modeled memristive synapses.

Simulation with modeled memristive synapses. We can conclude that AND, OR and NAND functions were solvable in the simulation setup when modeled memristive synapses were used in our neuron model. On the other hand, the XOR function was not solvable, and for this logical function, two output bits were not correctly classified.

Task	Input encoding	Loss	Misclassification
AND	time-inverted	0.1400	1
OR	time-inverted	0.1274	1
XOR	time-inverted	0.7641	2
NAND	time-inverted	0.0475	0

Table 16: Results of single neuron learning with real memristive synapses.

Real memristive synapses. It is interesting to see that the NAND function was utterly solvable in the setup with real memristive synapses. We can state that this result is comparable to results obtained with the modeled memristive synapses. AND and OR functions were not completely solvable, and for them, the final misclassification rate was one. We get the misclassification rate of two (that is the same result as in the simulation setup) for the XOR function.

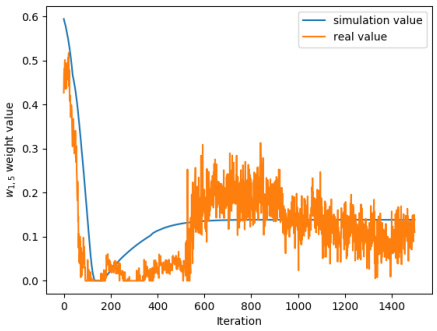
In the following part, we will compare "optimal" simulation results and results obtained from real devices for all weights, resistances, and loss function values.

AND. From Figure 16, we can see that all real weight values are noisy, but the most significant noise is observable in weights $w_{1,5}$ and $w_{3,5}$ approximately after the iteration number 600. Weights $w_{1,5}$ and $w_{4,5}$ follow the "optimal" simulated value to some extent, but $w_{2,5}$ deviates from the simulated value very quickly and reaches 0, while the optimal value amounts to roughly 0.2. The reason behind the steep decrease concerning the weight $w_{2,5}$ to the value of 0 is because the real resistance quickly reaches the maximum resistance bound of $R_c = 56k\Omega$. That means that the memristive device changed its working range compared to what it was initially measured. The $w_{3,5}$ value nicely follows the simulated value up to epoch 100 when we can see the divergence from the optimal value followed by the large noise until the end of the learning process. Simulated and real loss values are similar in the first 100 epochs where we can see the nice minimization progress. The final loss value in this setup is 0.140, and that is larger than the optimal one of 0.0233.

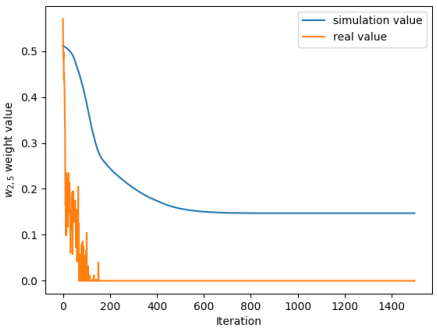
The plot of applied pulse trains and actual resistance values of all weights is displayed in Figure 17. This plot is used to investigate the noise in weights $w_{1,5}$ and $w_{3,5}$, which is significant after the iteration number 600. From resistance changes of $R_{1,5}$ and $R_{3,5}$ values, we can see that roughly after the pulse number 1200, our algorithm applied the positive and negative mix of pulse trains because of the positive and negative gradients in our stimuli. Then, the noise in resistance changes takes the bigger part in the case of resistance $R_{1,5}$. Noise in resistance $R_{3,5}$ is even more significant, and we can argue that the memristor started working in the switching mode making big changes with the small number of pulses.

OR. From Figure 18, the significant noise in weights $w_{1,5}$ and $w_{3,5}$ roughly after the iteration number 200 can be seen. Weights $w_{3,5}$ and $w_{4,5}$ follow the "optimal" simulated value, while $w_{2,5}$ deviates from the simulated value very quickly and reaches 0 even though the optimal value amounts to roughly 0.4. The reason behind the deviation of the real weight to 0 value $w_{2,5}$ is the same as in the AND experiment. We can also see that weight $w_{1,5}$ starts initially at a quite different value from the estimated one. Approximately at iteration 200, we can observe that the weight $w_{1,5}$ goes into another direction compared to the simulated value. That happened because the weight $w_{1,5}$ compensates changes introduced by the weight clipping of value $w_{2,5}$. The real loss value follows the simulated one, but the noise in loss value increases between the iteration number 170 and 600. That is due to the noise that is present in other weights as well. The real loss reaches the value of 0.127 that is larger than the optimal one of 0.0374.

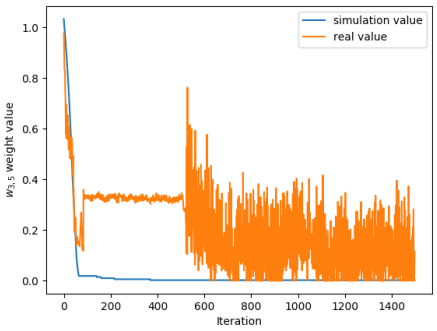
Regarding the OR task and weight changes over time, we can observe



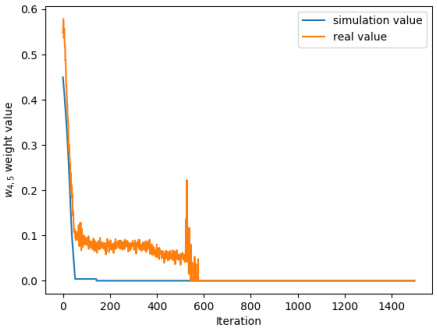
(a) Simulated and real weight $w_{1,5}$ value



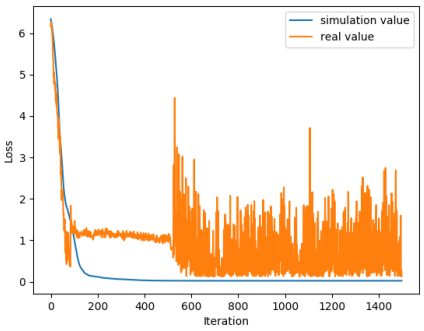
(b) Simulated and real weight $w_{2,5}$ value



(c) Simulated and real weight $w_{3,5}$ value

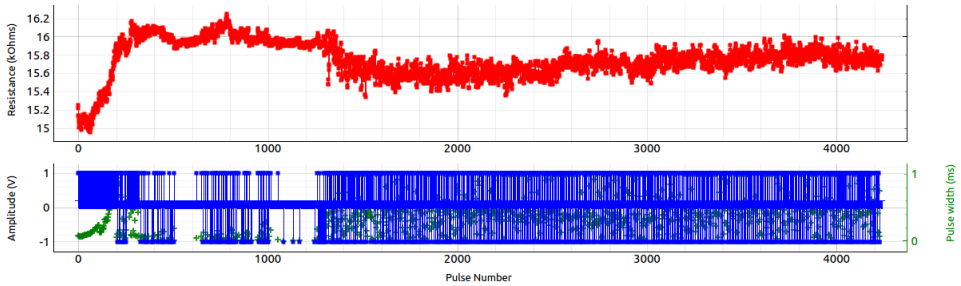


(d) Simulated and real weight $w_{4,5}$ value

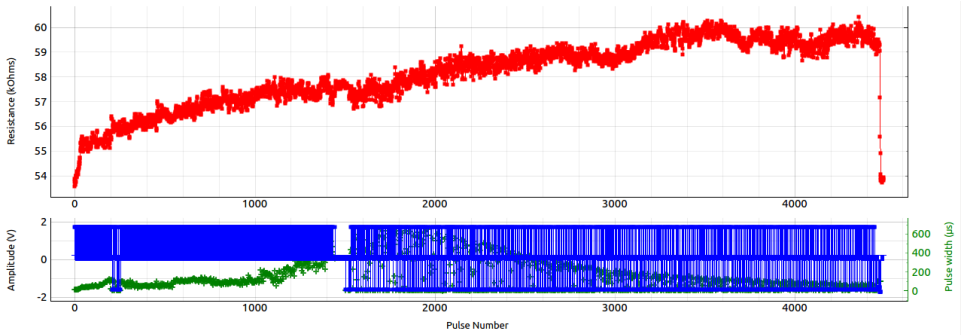


(e) Simulated and real loss values

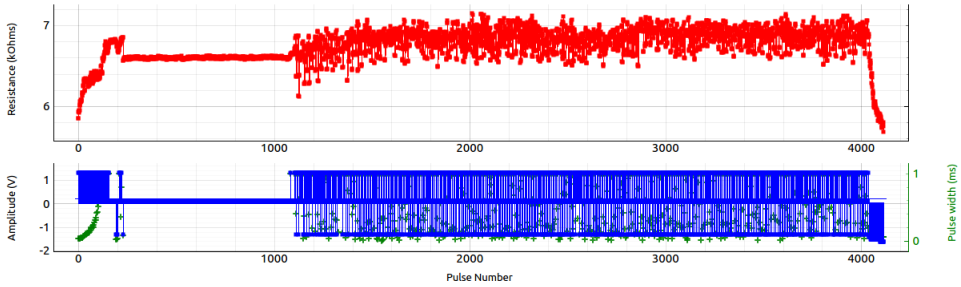
Figure 16: Comparison of simulation results and results obtained for real memristive devices for the AND task.



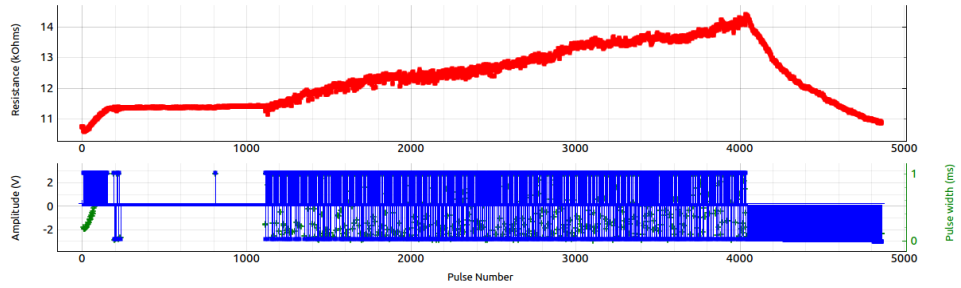
(a) $R_{1,5}$ change and applied pulse trains



(b) $R_{2,5}$ change and applied pulse trains



(c) $R_{3,5}$ change and applied pulse trains

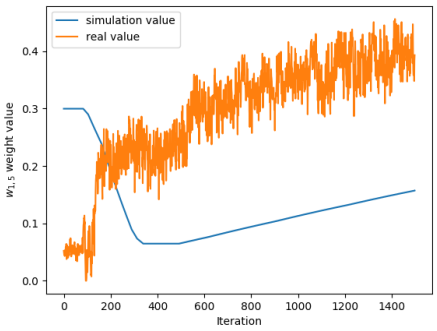


(d) $R_{4,5}$ change and applied pulse trains

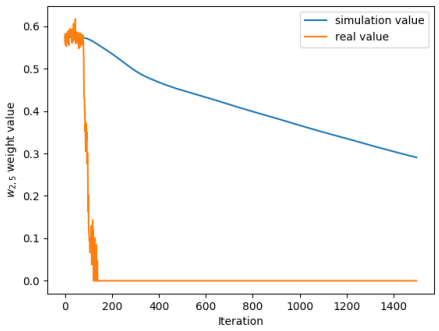
Figure 17: Actual resistance values and applied pulse trains for the AND task.

that noise takes a larger effect in weights $w_{1,5}$ and $w_{3,5}$ after the iteration number 200. From Figure 19 it can be seen that our algorithm applied positive and negative pulse trains roughly after iteration 200.

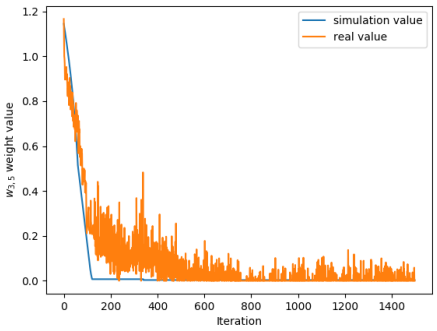
XOR. From Figure 20, we can see that only the weight value $w_{4,5}$ follows the simulated one. Weights $w_{1,5}$ and $w_{3,5}$ nicely follow the optimal value up to iteration number 400. After that, they diverge because of the significant noise that is present later. The $w_{2,5}$ weight value does not capture needed resistance changes and quickly goes to the value of 0, while the optimal value is around 0.2. The reason behind the deviation of real weight value $w_{2,5}$ is the same as in the previous experiments. Regarding the loss value, we can see that real values follow optimal ones up to the iteration number 400. After that period, the noise has a tremendous effect on the final loss value until the end of the learning process. The final loss value obtained in the setup with real memristors was 0.74, while in the simulation setup we obtained 0.25.



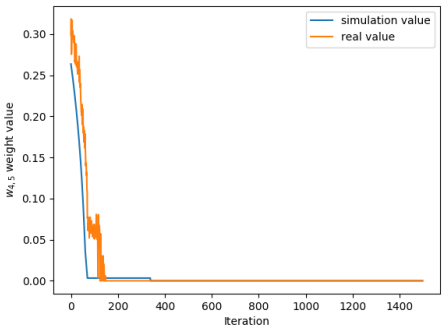
(a) Simulated and real weight $w_{1,5}$ value



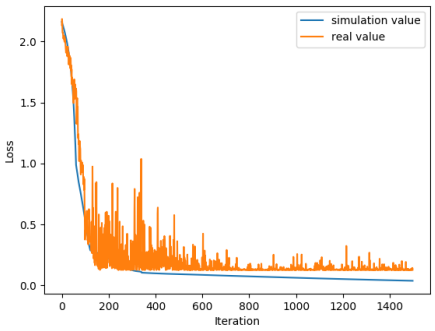
(b) Simulated and real weight $w_{2,5}$ value



(c) Simulated and real weight $w_{3,5}$ value

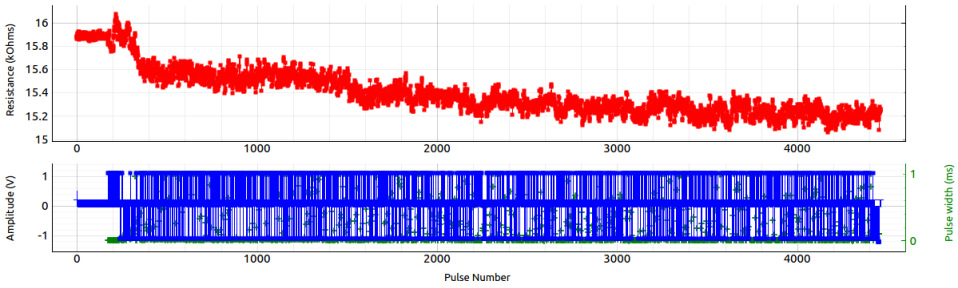


(d) Simulated and real weight $w_{4,5}$ value

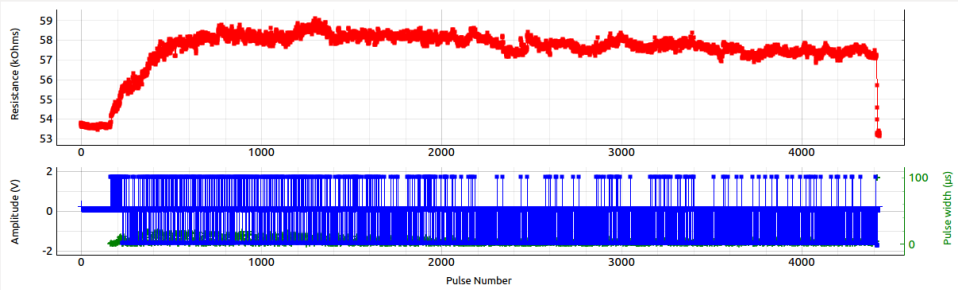


(e) Simulated and real loss comparison

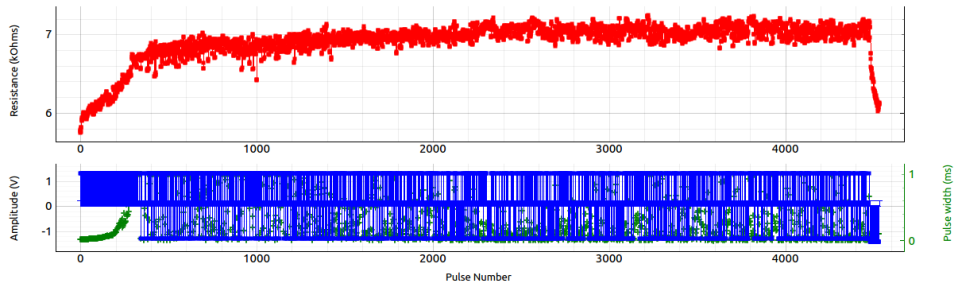
Figure 18: Comparison of simulation results and results obtained for real memristive devices for the OR task.



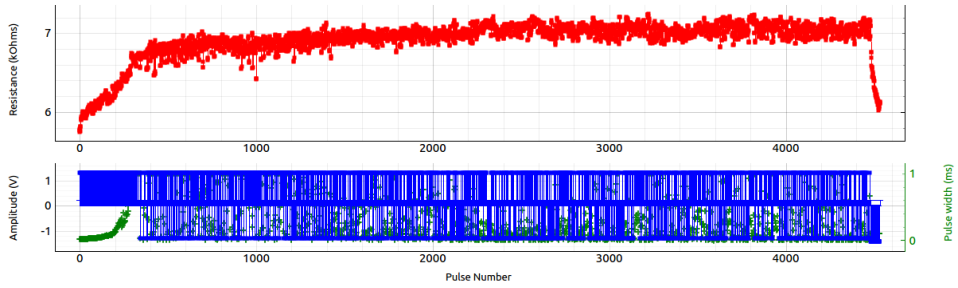
(a) $R_{1,5}$ change and applied pulse trains



(b) $R_{2,5}$ change and applied pulse trains

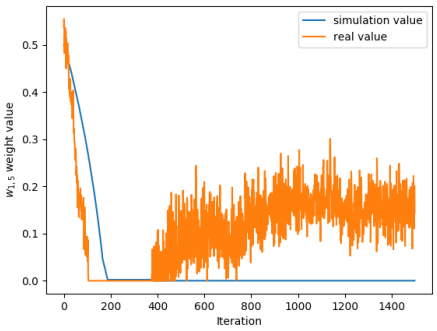


(c) $R_{3,5}$ change and applied pulse trains

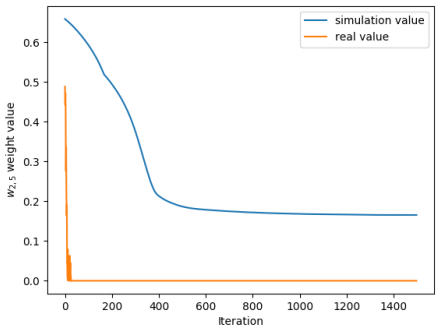


(d) $R_{4,5}$ change and applied pulse trains

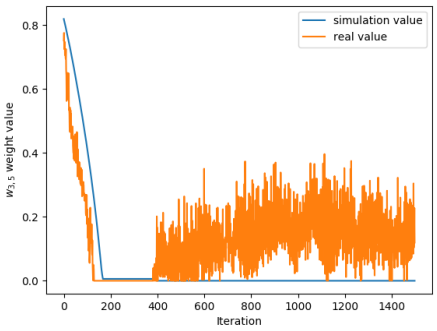
Figure 19: Actual resistance values and applied pulse trains for the OR task.



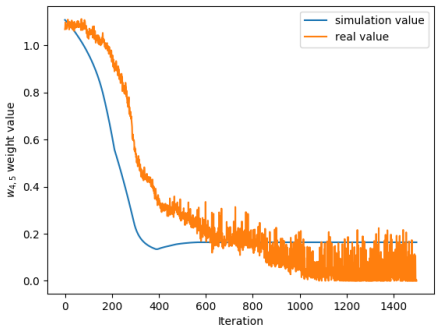
(a) Simulated and real weight $w_{1,5}$ value



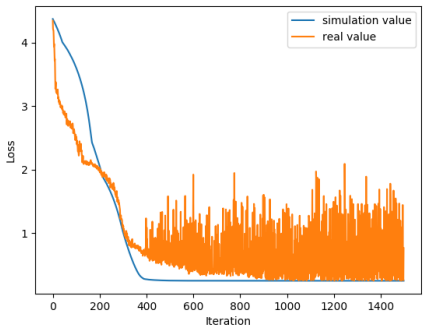
(b) Simulated and real weight $w_{2,5}$ value



(c) Simulated and real weight $w_{3,5}$ value



(d) Simulated and real weight $w_{4,5}$ value



(e) Simulated and real loss comparison

Figure 20: Comparison of simulation results and results obtained for real memristive devices for the XOR task.

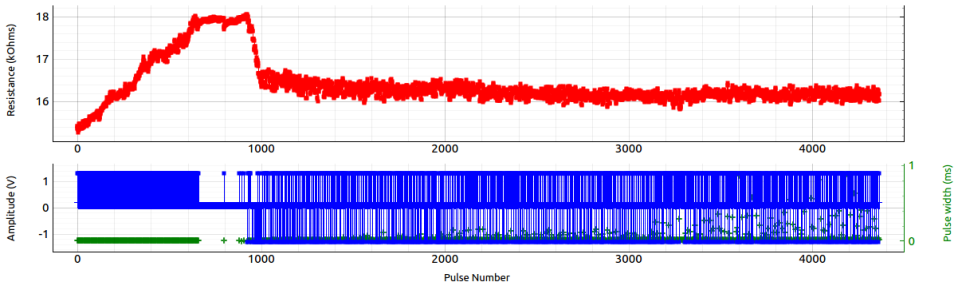
The plot of applied pulse trains and actual resistance values of all weights for the XOR task is displayed in Figure 21. From Figure 21, we can see that the switching behavior of memristors causes large resistance changes after the pulse number 900. This can be observable mainly in resistance values $R_{1,5}$ and $R_{3,5}$.

NAND. If we look at weight changes shown in Figure 22, we can see that real weights and simulated weights are similar at the end of the learning process. Real weights $w_{1,5}$ and $w_{2,5}$ reach 0 much faster than optimal ones. The most significant deviation between real and optimal weight value can be observed in the weight change $w_{4,5}$. The loss value obtained with actual devices is like a delayed version of the simulated loss value. Nevertheless, real and simulated loss values are similar, and they converge to values of 0.0475 and 0.0264 respectively. With this final results, we can say that the NAND logical task was completely solvable with the configuration containing real memristive devices. We can also see that the effect of noise on all weights and loss value was insignificant compared to the previous tests.

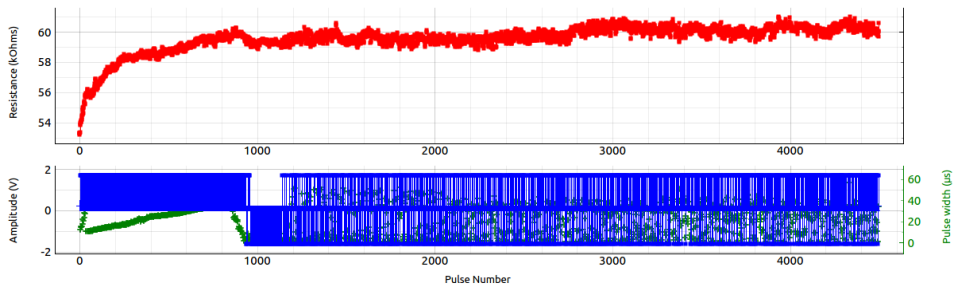
From Figure 23, we can see nice positive resistance changes on all memristive weights. If we look at applied pulse trains, we can argue that our algorithm applied mostly positive pulse trains in all cases. Compared to the previous examples, we can see that we do not experience enormous resistive jumps in weights and this led to a smaller noise and a possibility to solve the NAND logical function.

In the above applied algorithm, resistance values from real devices are not copied back to our simulated model. That implies that we are always calculating a pulse train length based on our simulated model and if there is a significant difference between simulated and real weights the time step computation is not entirely correct.

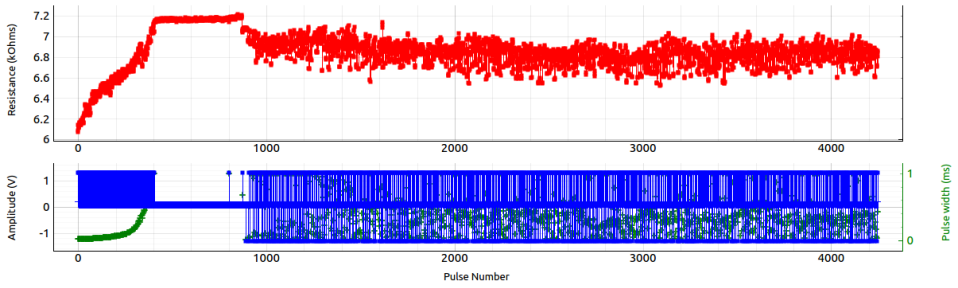
To solve this problem, copying real/noisy resistance values from real memristive devices to the simulated ones each n iterations is needed. This approach was only tried in simulation, where respective noise levels were modeled. More detailed discussion of the noisy memristive setup can be found in the next chapter.



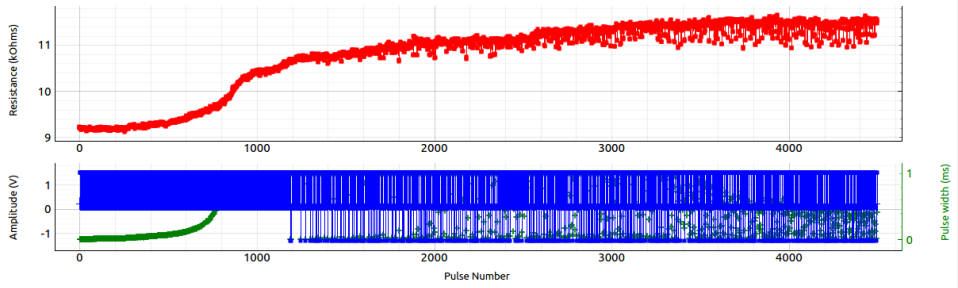
(a) $R_{1,5}$ change and applied pulse trains



(b) $R_{2,5}$ change and applied pulse trains

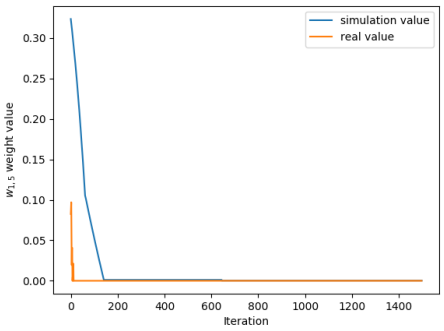


(c) $R_{3,5}$ change and applied pulse trains

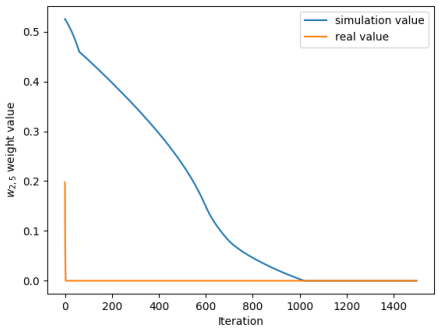


(d) $R_{4,5}$ change and applied pulse trains

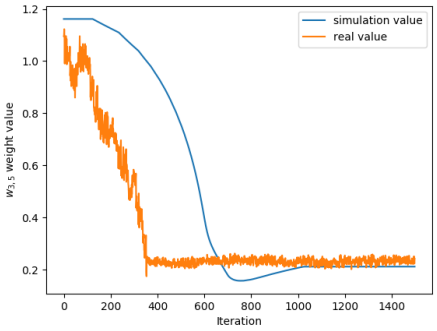
Figure 21: Actual resistance values and applied pulse trains for the XOR task.



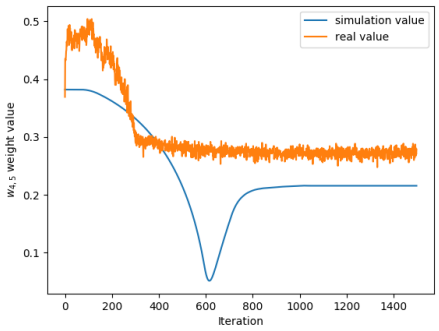
(a) Simulated and real weight $w_{1,5}$ value



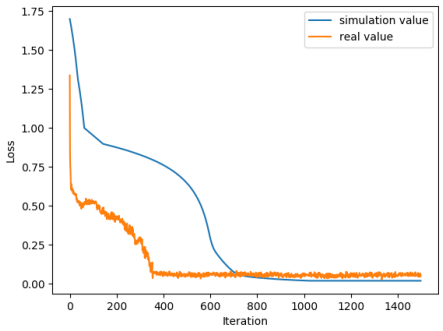
(b) Simulated and real weight $w_{2,5}$ value



(c) Simulated and real weight $w_{3,5}$ value

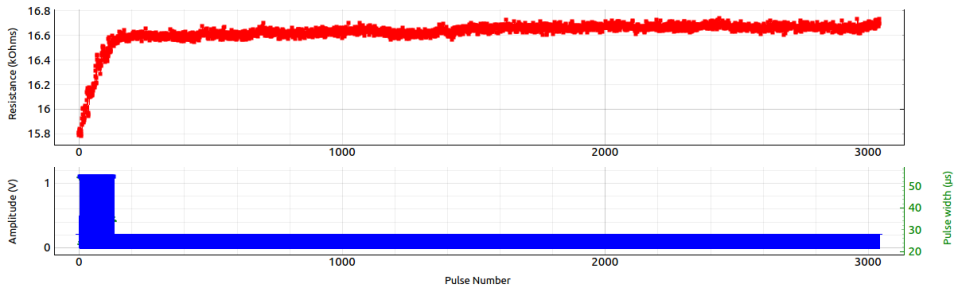


(d) Simulated and real weight $w_{4,5}$ value

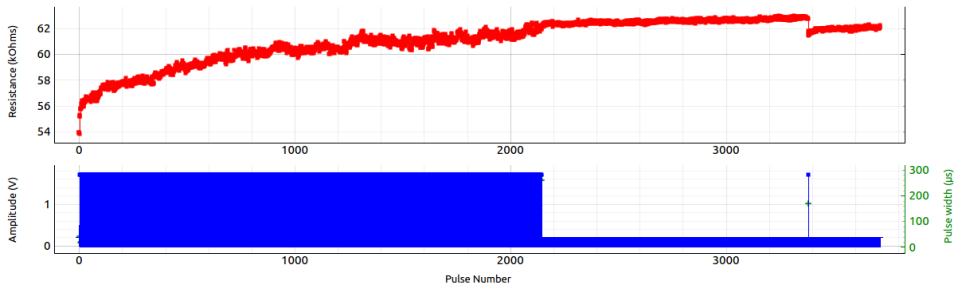


(e) Simulated and real loss comparison

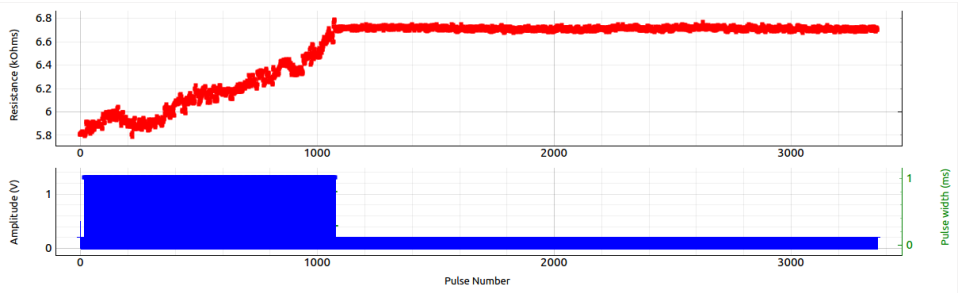
Figure 22: Comparison of simulation results and results obtained for real memristive devices for the NAND task.



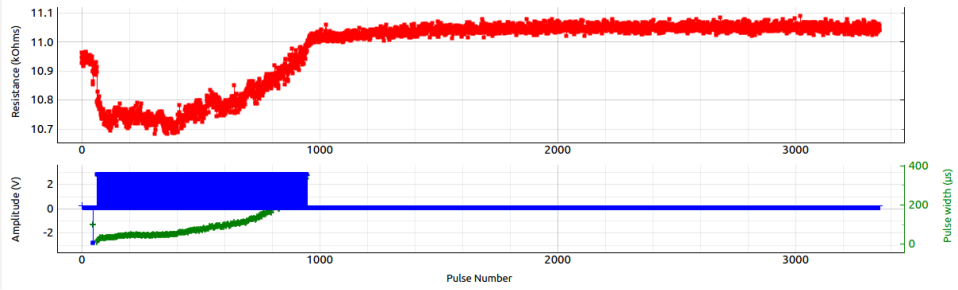
(a) $R_{1,5}$ change and applied pulse trains



(b) $R_{2,5}$ change and applied pulse trains



(c) $R_{3,5}$ change and applied pulse trains



(d) $R_{4,5}$ change and applied pulse trains

Figure 23: Actual resistance values and applied pulse trains for the NAND task.

6.4 Learning logical functions with modeled noisy memristive synapses

In this subsection, we discuss results on AND, OR, XOR and NAND functions of the single neuron learning with time-inverted input encoding, but with modeled noisy memristive synapses. Readout noise was modeled with the uniform distribution using $\gamma \cdot R \cdot (2 \cdot uniform(0,1) - 1)$ where $\gamma = 0.004$ and R was the current resistance value. The noise in the resistance change for all memristors was drawn from the uniform distribution $\delta \cdot R \cdot (2 \cdot uniform(0,1) - 1)$ where $\delta = 0.001$. Noise models are taken from our partners at the University of Southampton and parameters γ and δ were chosen to match noise level observed in experiments with real memristive synapses. In simulations with noise, we had two networks: one containing mathematical models without noise (modeling the simulated memristors) and one containing mathematical models with noise (modeling the real memristors). We calculated a pulse width needed to reach the new resistance value from the model without noise, and then we applied this pulse train on both models. The noisy model diverged from the model without noise, and therefore, we copied weights and resistance values from the noisy model back to the model without noise every 300 iterations. Memristive weights have been created based on the mathematical models and initial values defined in the previous chapter, as depicted in Tables 11 - 14. Also in this setup, memristive weights were bounded to range $w_{i,j} \in [0,1]$ and batch learning was applied with a batch size of four. The MMSE loss was minimized using the standard Gradient Descent optimizer with a learning rate of 10000, and the number of training epochs was set to 2100. Final results are shown in the following table:

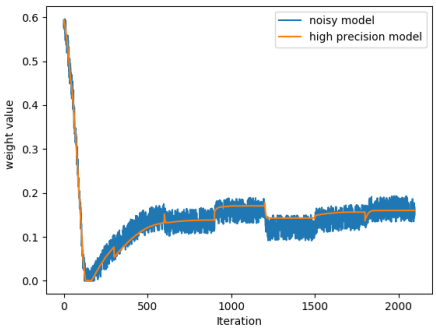
Task	Input encoding	Loss	Misclassification
AND	time-inverted	0.0559	0
OR	time-inverted	0.0863	0
XOR	time-inverted	0.2713	2
NAND	time-inverted	0.0254	0

Table 17: Results of the single neuron learning with noisy memristive synapses.

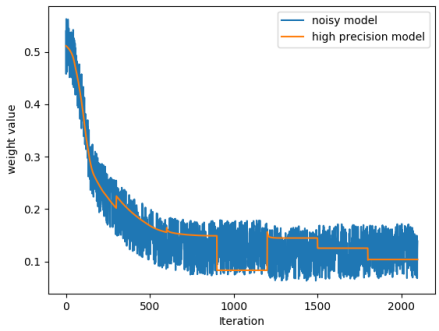
From the results above, we can see that it was possible to solve AND, OR and NAND logical functions with noisy memristive synapses, while the XOR task was not solvable. Final results are similar to simulation results obtained with memristive synapses without noise (shown in Table 15).

In the following figures, the evolution of high precision (noiseless) and noisy memristive weights can be seen for AND, OR, XOR and NAND tasks.

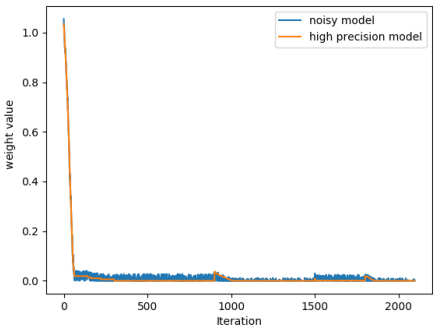
From all figures presented below, it can be seen that every 300 iterations weight values of our high precision model are set to the noisy weight values. Deviations between the noisy model and high precision model can also be observed. Since we copy weights and resistance values from the noisy model back to the high precision model every 300 iterations, our noisy model tends to go towards the "optimal" result.



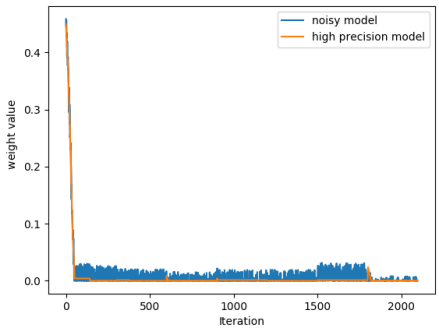
(a) High precision and noisy weight $w_{1,5}$ value



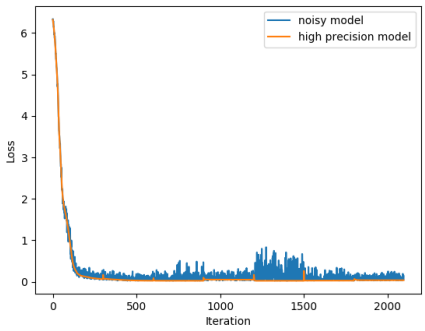
(b) High precision and noisy weight $w_{2,5}$ value



(c) High precision and noisy weight $w_{3,5}$ value

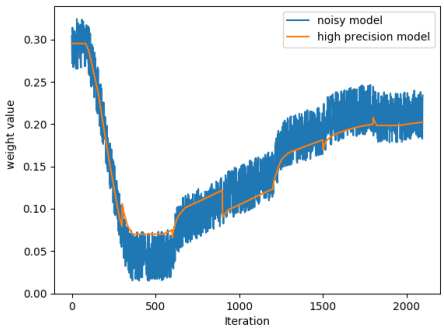


(d) High precision and noisy weight $w_{4,5}$ value

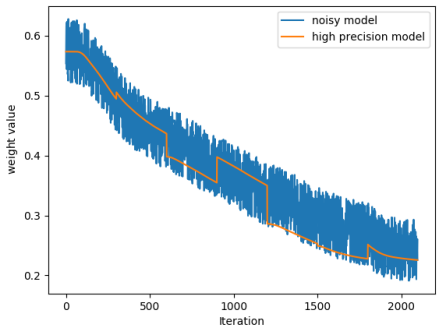


(e) Loss comparison of high precision and noisy single neuron learning

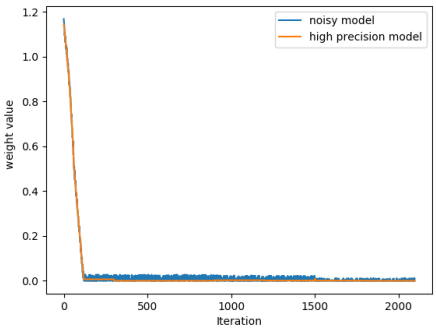
Figure 24: Comparison of noisy weights and high precision weights (without noise) for the AND task in the single neuron learning setup with time-inverted input encoding.



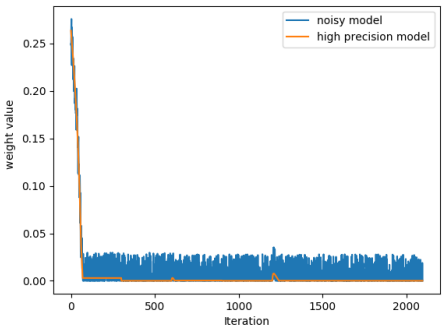
(a) High precision and noisy weight $w_{1,5}$ value



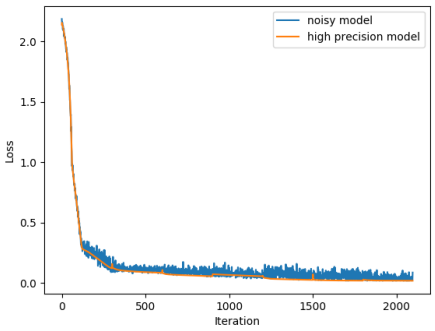
(b) High precision and noisy weight $w_{2,5}$ value



(c) High precision and noisy weight $w_{3,5}$ value

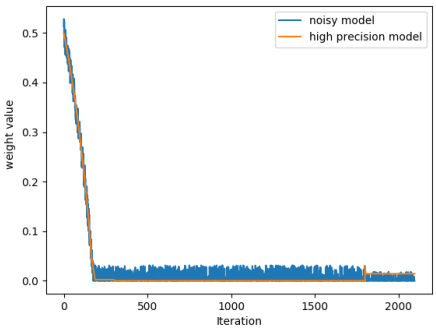


(d) High precision and noisy weight $w_{4,5}$ value

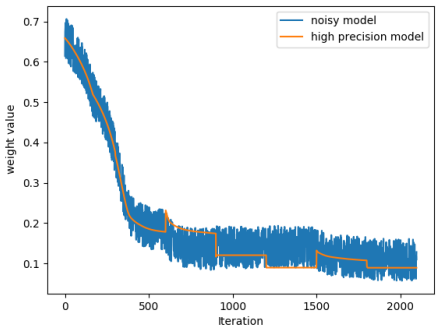


(e) Loss comparison of high precision and noisy single neuron learning

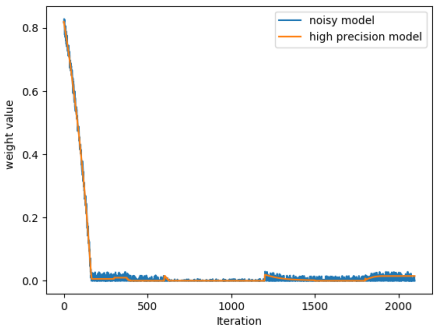
Figure 25: Comparison of noisy weights and high precision weights (without noise) for the OR task in the single neuron learning setup with time-inverted input encoding.



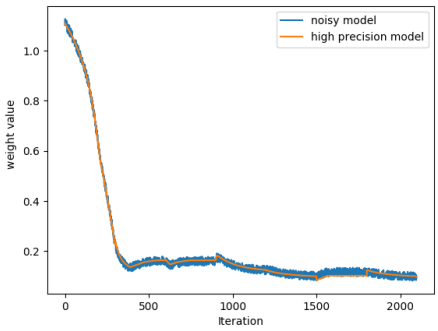
(a) High precision and noisy weight $w_{1,5}$ value



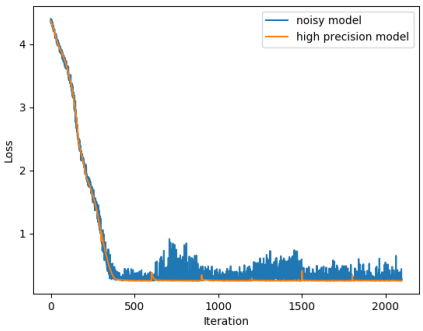
(b) High precision and noisy weight $w_{2,5}$ value



(c) High precision and noisy weight $w_{3,5}$ value

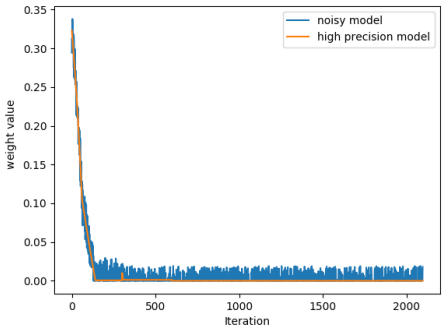


(d) High precision and noisy weight $w_{4,5}$ value

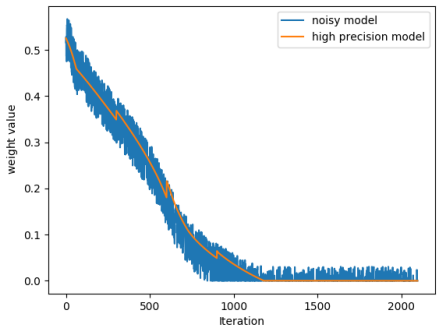


(e) Loss comparison of the high precision and noisy single neuron learning

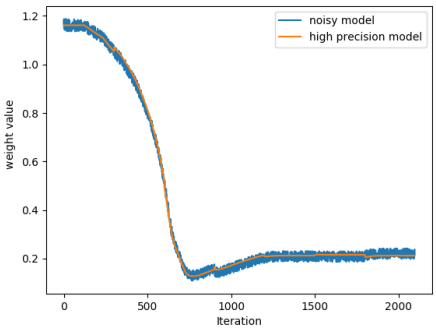
Figure 26: Comparison of noisy weights and high precision weights (without noise) for the XOR task in the single neuron learning setup with time-inverted input encoding.



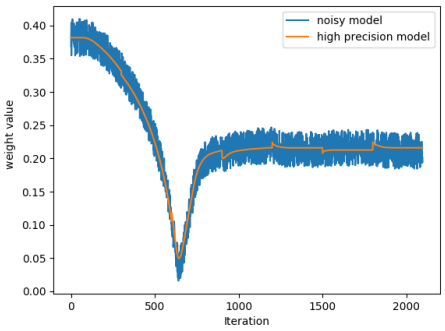
(a) High precision and noisy weight $w_{1,5}$ value



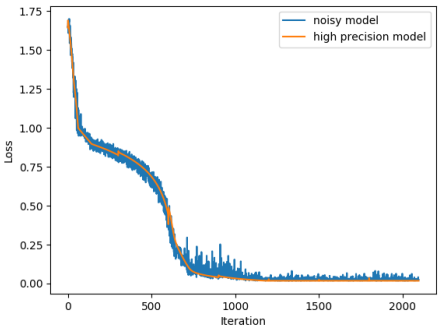
(b) High precision and noisy weight $w_{2,5}$ value



(c) High precision and noisy weight $w_{3,5}$ value



(d) High precision and noisy weight $w_{4,5}$ value



(e) Loss comparison of the high precision and noisy single neuron learning

Figure 27: Comparison of noisy weights and high precision weights (without noise) for the NAND task in the single neuron learning setup with time-inverted input encoding.

6.5 Learning the Iris task using SNN with non-memristive synapses

We compared results achieved by ANNs and SNNs on the Iris task to check the generalization capabilities of our SNN model, i.e., to investigate how the network performs on unseen data. The Iris dataset [30] consists of 150 samples from three different species of Iris flower (Iris setosa, Iris virginica, and Iris versicolor). The original dataset was split into a training set (containing 120 samples) and test set (containing 30 samples).

6.5.1 Artificial neural network setup

Regarding the artificial neural network hyperparameters for the Iris task, we used one hidden layer containing 30 neurons and the Adam optimization algorithm [31] with a learning rate of 0.005. The number of training iterations was set to 500 and training was performed on a full dataset of 120 examples. Furthermore, the cross-entropy cost function was optimized.

6.5.2 Spiking neural network setup

Regarding the spiking neural network hyper-parameters for the Iris task, we used one hidden layer containing 30 neurons and Adam optimization algorithm [31] with a learning rate of 0.005. The number of training iterations was set to 500, and the learning rate was decayed with a factor of 0.5 every 150 iterations. Like in the previous setup, training was performed on a full dataset of 120 examples. Each training example in the Iris dataset contains four unique features, and original features (without any transformation) were presented as excitatory inputs to the spiking neural network. Each neuron in the hidden layer was created with the threshold of $\vartheta = 1.5$, while neurons in the output layer had the threshold of $\vartheta = 3$. All spikes of the hidden layer were split into half excitatory and half inhibitory inputs to neurons in the output layer.

We optimized the cost function that will encourage the $i-th$ output neuron to spike first if the input image should be classified to $i-th$ class:

$$Loss = \sum_{\substack{i \\ i \neq c}} \sigma\left(t_c - t_i\right) \tag{74}$$

where:

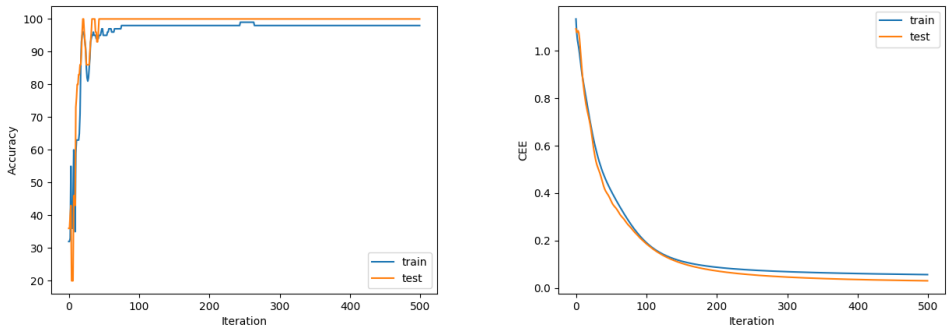
- $\sigma(x)$ - is the sigmoid function,
- t_c - represents the output spike time of the neuron that spikes first, and,
- t_i - represent output spike times of all other neurons.

Final results are shown in the following table:

Setup	Test set accuracy [%]	Mean firing rate [%]
ANN	100	-
SNN	100	71

Table 18: Final Iris results.

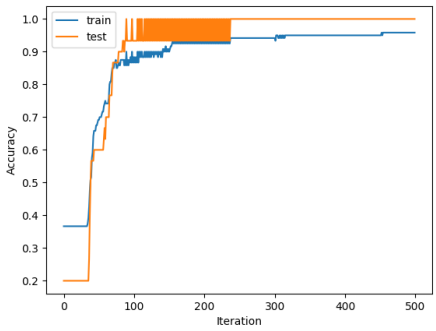
From Table 18, we can see that both networks obtained 100% accuracy on the test set with almost the same hyperparameters and number of training epochs. Train and test accuracy and loss function trends during learning for both ANN and SNN are shown in Figures 28 and 29. From these figures, we can conclude that the ANN generalized faster and obtained the test accuracy of 100% roughly at epoch 100, compared to the SNN, where the test accuracy of 100% was reached after approximately 250 epochs. In both cases, a smooth and decreasing trend of the loss function was observed, explaining the stable learning process. In test accuracy plots at the beginning of learning one can see slight jumps in the SNN training, but the learning process stabilizes once the task is mastered. The third column in the previous table depicts a final mean rate of all neurons on the test set. The mean rate is calculated by looking at the first spike time in the output layer and calculating the percentage of all neurons that spiked before that time. From mean rate results it is clear that only a subset of our spiking neurons (on average) is needed to solve the Iris task, thus the sparse output spike times in SNN are the one advantage compared to ANN.



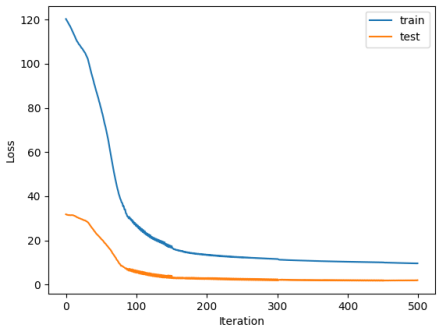
(a) Train and test accuracy comparison

(b) Train and test loss comparison

Figure 28: Train and test accuracy and loss trend of the ANN on the Iris task.



(a) Train and test accuracy comparison



(b) Train and test loss comparison

Figure 29: Train and test accuracy and loss trend of the SNN on the Iris task.

6.6 Learning the MNIST task using SNNs with non-memristive synapses

In this section, a comparison of results achieved by ANNs and SNNs on the MNIST handwritten digits classification task are presented. The MNIST dataset [32] consists of 55000 training and 10000 testing images (each with a dimension of 28x28 pixels) of 10 handwritten digits.

6.6.1 Artificial neural network setup

Regarding the artificial neural network hyper-parameters for the MNIST task, we used one hidden layer containing 100 and 200 neurons. Both Adam [31] and RMSProp [33] optimization algorithms are tested with a learning rate of 0.005, and batch learning is performed with a batch size of 100. The number of training iterations was set to 2750. The cross-entropy cost function was optimized.

6.6.2 Spiking neural network setup

Regarding the spiking neural network hyper-parameters for the MNIST task, spiking neural network setups with one hidden layer containing 100 and 200 neurons were used. Optimization with the Adam algorithm [31] and learning rates of 0.01 and 0.02 was tested for the setup with 100 and 200 neurons respectively. We have also tried the RMSProp optimization algorithm [33] with the learning rate of 0.0005 on setups containing 100 and 200 neurons in the first hidden layer. In all cases, batch learning was performed with a batch size of 100 and the initial learning rate was decayed with the factor of 0.5 every 500 iterations. The number of training iterations was set to 2750. Pixels in input images were normalized between values 0 and 1, and based on that we created our input encoding (input spike times) in the range $[a, b]$ by using the following formula:

$$input\_encoding = a + (1 - pixel\_value)(b - a) \tag{75}$$

In our setup, a and b values are set to 1 and 3 respectively. Equation (75) represents another form of the temporal coding we use, where again the stronger stimuli (larger input pixel) is encoded to the earlier spike time, and vice versa. Basic input representation is used where input spike times are fed to the same number excitatory and inhibitory input neurons.

Both neurons in the hidden and output layer had a firing threshold set to $\vartheta = 20$. All spike times obtained in the hidden layer output are split into one half being excitatory and the other half inhibitory inputs to the neurons in the output layer. Furthermore, we optimized the cost function defined in equation (74).

Final results are shown in the following tables:

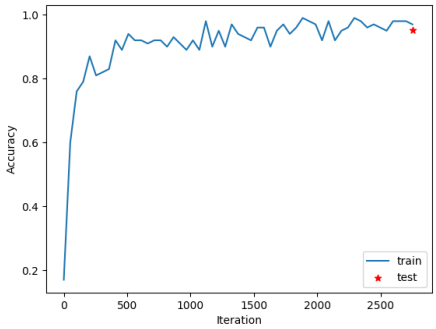
Network	Opt. algo	lr	Hidden count	Test acc. [%]	Mean rate [%]
ANN	Adam	0.005	100	95.08	-
	Adam	0.005	200	94.88	-
	RMSProp	0.005	100	95.65	-
	RMSProp	0.005	200	95.83	-

Table 19: Final results of different ANNs on the MNIST task.

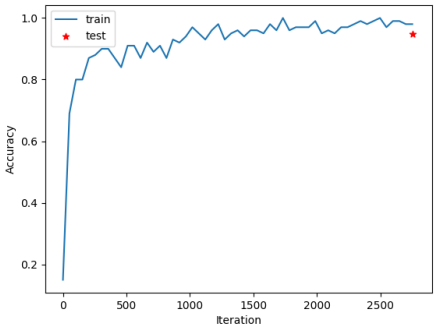
Network	Opt. algo.	lr	Hidden count	Test acc. [%]	Mean rate [%]
SNN	Adam	0.01	100	93.29	75.67
	Adam	0.02	200	94.01	72.31
	RMSProp	0.0005	100	93.64	82.52
	RMSProp	0.0005	200	93.95	81.01

Table 20: Final results of different SNNs on the MNIST task.

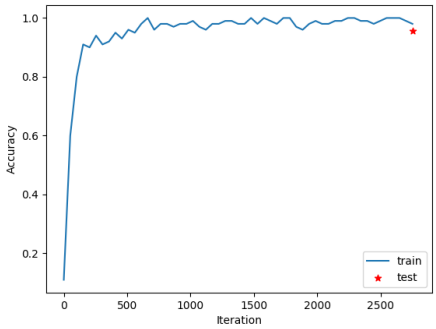
From Tables 19 and 20, we can see that ANNs obtained slightly better results compared to our SNN setup for the same number of training epochs. From average mean firing rate values in SNNs, it is clear that only a subset of neurons spiked before the time used to classify results on the test set. From accuracy trends shown in 30 and 31 we can observe similar generalization power of both networks. However, our best SNN result of 94.01 % is smaller than the best ANN result of 95.83 %.



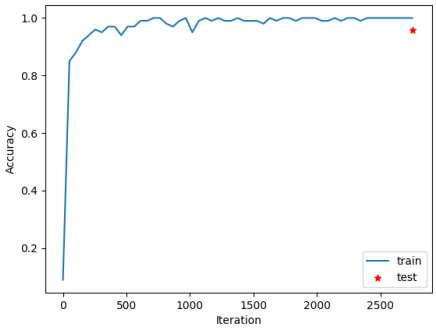
(a) Train and test accuracy in the ANN setup with 100 hidden neurons and Adam optimization algorithm



(b) Train and test accuracy in the ANN setup with 200 hidden neurons and Adam optimization algorithm

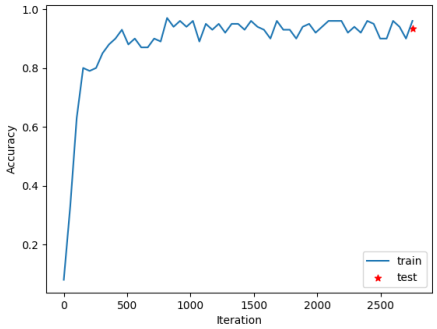


(c) Train and test accuracy in the ANN setup with 100 hidden neurons and RMSProp optimization algorithm

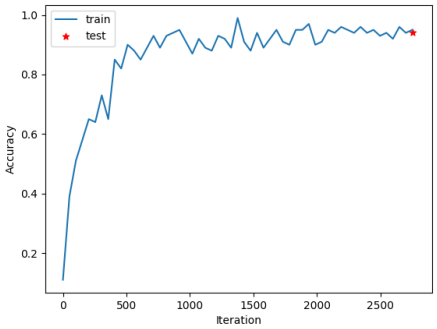


(d) Train and test accuracy in the ANN setup with 200 hidden neurons and RMSprop optimization algorithm

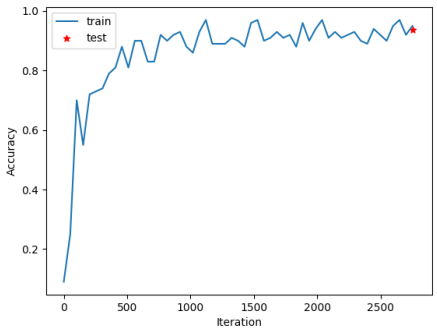
Figure 30: Train and test accuracy trends of different ANN setups on the MNIST task.



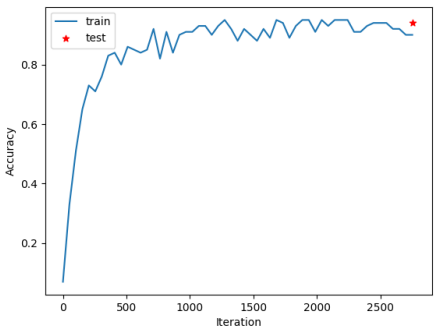
(a) Train and test accuracy in the SNN setup with 100 hidden neurons and Adam optimization algorithm



(b) Train and test accuracy in the SNN setup with 200 hidden neurons and Adam optimization algorithm



(c) Train and test accuracy in the SNN setup with 100 hidden neurons and RMSProp optimization algorithm



(d) Train and test accuracy in the SNN setup with 200 hidden neurons and RMSprop optimization algorithm

Figure 31: Train and test accuracy trends of different SNN setups on the MNIST task.

7 Conclusion

In this thesis, we introduced a novel spiking neural network containing custom neuron models with memristive synapses. We showed that output spike times of our neuron model are differentiable, thus the backpropagation algorithm can be applied directly to networks of such neurons. We applied our SNN with non-memristive synapses to simple and more complex tasks and achieved results comparable with ANNs. Similar results are obtained when SNNs with modeled memristive synapses are trained on simple tasks. On the other hand, we experienced problems during training of SNNs with real memristive synapses. Those problems were related to the variability and noise of different devices, thus each device was controlled individually with the respective threshold voltages. This is similar to the discussion addressed in Section 5. Real memristive devices were sensitive when we applied the mix of a small number of positive and negative pulse trains. In that case, they started working in the switching regime making large positive and negative resistance changes for the small number of pulses. In the future work, one possibility to solve this problem would be to include a smaller time step dt to refine the resistance change. The time step used in all experiments before was $dt = 10^{-6}$ s. With ArcOne it will be possible to decrease this time step to $dt = 50 \cdot 10^{-9}$ s, and this might lead to the decrease of switching effect influence and smaller noise in our weight changes.

Using the copying of real (noisy) weights to our simulated (not-noisy) weights every n iterations, we can use the correct gradient information which can improve final results in the setup with real hardware devices. We demonstrated the results of that approach only through our simulations in Section 6.4.

Appendices

A Gradients of different loss functions

Gradients of the MMSE (67) loss with respect to a neuron output spike time tsp_i are:

$$\frac{\partial MMSE}{\partial tsp_i} = \begin{cases} 0 & \text{if } t_i = t_{sp_{late}} \quad \text{and} \quad tsp_i \geq t_i \\ 0 & \text{if } t_i = t_{sp_{early}} \quad \text{and} \quad tsp_i \leq t_i \\ -\frac{2}{n}(t_i - tsp_i) & \text{otherwise} \end{cases} \tag{76}$$

Gradients of the loss function (74) used in Iris and MNIST tasks w.r.t spike times of neurons in the output layer t_k are:

$$\frac{\partial Loss}{\partial t_k} = \begin{cases} \sum_{\substack{i \\ i \neq c}} \sigma\left(t_c - t_i\right)\left(1 - \sigma(t_c - t_i)\right) & \text{if } t_k = t_c \\ -\sigma\left(t_c - t_k\right)\left(1 - \sigma(t_c - t_k)\right) & \text{if } t_k \neq t_c \end{cases} \tag{77}$$

B Experimental memristor mathematical models

During multiple experiments with real memristors, several memristor model parameters were extracted. All models are presented in the following tables:

Model	Model 1	Model 2	Model 3	Model 4	Model 5	Model 6
A_p	0.057	1.958	0.043	0.0116	0.197	0.0365
A_n	-15.734	-3.038	-0.405	-0.059	-0.126	-0.648
t_p	2.596	1.875	1.442	2.452	1.731	4.039
t_n	2.596	1.875	1.442	2.308	1.731	4.039
a_{0p}	-54210.50	1752.045	5848.479	16367.18	2731.854	519.336
a_{0n}	34965.853	10275.769	14903.227	72784.951	6568.330	8376.799
a_{1p}	63549.984	10743.670	10731.767	23896.231	3393.513	4100.118
a_{1n}	-544.459	-228.823	-329.116	15913.471	636.491	-884.598
V_p	1.8	1.3	0.9	1.5	1.3	2.8
V_n	-1.8	-1.3	-0.85	-1.45	-1.3	-2.8

Table 21: Various memristor model parameters.

Model	Model 7	Model 8	Model 9	Model 10	Model 11	Model 12
A_p	0.0713	0.299	-0.161	-7.154	1.357	1.1438
A_n	-0.197	-0.163	0.0306	1.995	-4.681	-1.1483
t_p	2.452	3.318	1.586	2.596	5.049	1.731
t_n	2.164	3.173	1.586	2.452	5.049	1.731
a_{0p}	-458.574	7800.857	15872.892	8710.499	5809.417	9000
a_{0n}	15399.756	11637.933	9876.268	7932.314	6662.820	5000
a_{1p}	7822.382	1911.918	-5196.629	-770.313	111.667	500
a_{1n}	4752.090	49.856	-2975.463	13.757	256.923	500
V_p	1.5	2.2	-1	1.6	3.3	1.3
V_n	-1.3	-2.1	1	-1.6	-3.3	-1.3

Table 22: Various memristor model parameters.

C References

[1] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep learning. *nature*, 521(7553):436, 2015.

[2] David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. Learning representations by back-propagating errors. *nature*, 323(6088):533, 1986.

[3] Wolfgang Maass. Networks of spiking neurons: the third generation of neural network models. *Neural networks*, 10(9):1659–1671, 1997.

[4] Warren S McCulloch and Walter Pitts. A logical calculus of the ideas immanent in nervous activity. *The bulletin of mathematical biophysics*, 5(4):115–133, 1943.

[5] Kurt Hornik, Maxwell Stinchcombe, and Halbert White. Multilayer feedforward networks are universal approximators. *Neural networks*, 2(5):359–366, 1989.

[6] Timothy J Gawne, TROELS W Kjaer, and BARRY J Richmond. Latency: another potential code for feature binding in striate cortex. *Journal of neurophysiology*, 76(2):1356–1360, 1996.

[7] Hélene Paugam-Moisy and Sander Bohte. Computing with spiking neuron networks. In *Handbook of natural computing*, pages 335–376. Springer, 2012.

[8] Wulfram Gerstner, Werner M Kistler, Richard Naud, and Liam Paninski. *Neuronal dynamics: From single neurons to networks and models of cognition*. Cambridge University Press, 2014.

[9] Jesper Sjöström and Wulfram Gerstner. Spike-timing dependent plasticity. *Spike-timing dependent plasticity*, 35(0):0–0, 2010.

[10] Larry F Abbott and Sacha B Nelson. Synaptic plasticity: taming the beast. *Nature neuroscience*, 3(11s):1178, 2000.

[11] Sander M Bohte, Joost N Kok, and Han La Poutre. Error-backpropagation in temporally encoded networks of spiking neurons. *Neurocomputing*, 48(1-4):17–37, 2002.

[12] Wulfram Gerstner. Spike-response model. *Scholarpedia*, 3(12):1343, 2008.

[13] Andrzej Kasiński and Filip Ponulak. Comparison of supervised learning methods for spike time coding in spiking neural networks. *International Journal of Applied Mathematics and Computer Science*, 16:101–113, 2006.

[14] Filip Ponulak. Resume-new supervised learning method for spiking neural networks. *Institute of Control and Information Engineering, Poznan University of Technology*, 42, 2005.

[15] Dongsung Huh and Terrence J Sejnowski. Gradient descent for spiking neural networks. *arXiv preprint arXiv:1706.04698*, 2017.

[16] Paul J Werbos. Backpropagation through time: what it does and how to do it. *Proceedings of the IEEE*, 78(10):1550–1560, 1990.

[17] Guillaume Bellec, Darjan Salaj, Anand Subramoney, Robert Legenstein, and Wolfgang Maass. Long short-term memory and learning-to-learn in networks of spiking neurons. *arXiv preprint arXiv:1803.09574*, 2018.

[18] Hesham Mostafa. Supervised learning based on temporal coding in spiking neural networks. *IEEE transactions on neural networks and learning systems*, 29(7), 2018.

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

[20] Leon O Chua and Sung Mo Kang. Memristive devices and systems. *Proceedings of the IEEE*, 64(2):209–223, 1976.

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

[22] Zdeněk Biolek, Dalibor Biolek, and Viera Biolkova. Spice model of memristor with nonlinear dopant drift. *Radioengineering*, 18(2), 2009.

[23] Yogesh N Joglekar and Stephen J Wolf. The elusive memristor: properties of basic electrical circuits. *European Journal of Physics*, 30(4):661, 2009.

[24] Ioannis Messaris, Alexander Serb, Spyros Stathopoulos, Ali Khiat, Spyridon Nikolaidis, and Themistoklis Prodromakis. A data-driven verilog-a reram model. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2018.

[25] Ioannis Messaris, Alexander Serb, Ali Khiat, Spyridon Nikolaidis, and Themistoklis Prodromakis. A compact verilog-a reram switching model. *arXiv preprint arXiv:1703.01167*, 2017.

[26] Geoffrey W Burr, Robert M Shelby, Severin Sidler, Carmelo Di Nolfo, Junwoo Jang, Irem Boybat, Rohit S Shenoy, Pritish Narayanan, Kumar Virwani, Emanuele U Giacometti, et al. Experimental demonstration and tolerancing of a large-scale neural network (165 000 synapses) using

phase-change memory as the synaptic weight element. *IEEE Transactions on Electron Devices*, 62(11):3498–3507, 2015.

[27] Alexander Serb, Johannes Bill, Ali Khiat, Radu Berdan, Robert Legenstein, and Themis Prodromakis. Unsupervised learning in probabilistic neural networks with multi-state metal-oxide memristive synapses. *Nature communications*, 7:12611, 2016.

[28] Johannes Bill and Robert Legenstein. A compound memristive synapse model for statistical learning through stdp in spiking neural networks. *Frontiers in neuroscience*, 8:412, 2014.

[29] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. Automatic differentiation in pytorch. In *NIPS-W*, 2017.

[30] Ronald A Fisher and Michael Marshall. Iris data set. *RA Fisher, UC Irvine Machine Learning Repository*, 440, 1936.

[31] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[32] Yann LeCun, Corinna Cortes, and CJC Burges. The mnist dataset of handwritten digits. 1998.

[33] Tijmen Tieleman and Geoffrey Hinton. Lecture 6.5-rmsprop: Divide the gradient by a running average of its recent magnitude. *COURSERA: Neural networks for machine learning*, 4(2):26–31, 2012.