RUPRECHT-KARLS-UNIVERSITÄT HEIDELBERG

Julian Göltz

Training Deep Networks with Time-to-First-Spike Coding on the BrainScaleS Wafer-Scale System

Master Thesis

KIRCHHOFF-INSTITUT FÜR PHYSIK

Department of Physics and Astronomy
University of Heidelberg

Master Thesis
in Physics
submitted by
Julian Göltz
born in Crailsheim

2019

Training Deep Networks with Time-to-First-Spike Coding on the BrainScaleS Wafer-Scale System

This master thesis has been carried out by Julian Göltz
at the

Kirchhoff Institute for Physics

Ruprecht-Karls-Universität Heidelberg

under the supervision of

Dr. Johannes Schemmel

and

Prof. Dr. Karlheinz Meier

Training Deep Networks with Time-to-First-Spike Coding on the BrainScaleS Wafer-Scale System

*Artificial neural network*s (ANNs) achieve impressive results in pattern recognition tasks. Recently, a way of implementing ANN algorithms in *spiking neural network*s (SNNs) was derived with the help of time-to-first-spike coding. This coding enables and even encourages usage of established methods like error backpropagation on neuromorphic hardware. The goal of this thesis is to realise learning with time-to-first-spike coding on the *BrainScaleS* (BSS) wafer-scale system, harnessing its power of fast and energy-efficient analogue emulation. However, analogue circuits impair full control of neuron parameters and introduce noise on spike times, both detrimental to the use of exact and differentiable relations, as in the original approach. This work establishes a framework that significantly adapts the original idea to the neuron model used on BSS. The aforementioned challenges are cautiously addressed by investigations into the reproducibility of spike times on hardware, extensive software simulations, and implementation of training with time-to-first-spike coding on hardware. This enables the demonstration of high-speed visual data classification using fully hardware-emulated networks with extremely sparse response properties, thus paving the way towards larger-scale setups for complex pattern recognition that may challenge the state of the art in terms of speed and energy efficiency.

Training von Tiefen Netzwerken mithilfe von Time-to-First-Spike-Kodierung auf dem BrainScaleS Wafer-Scale System

Künstliche neuronale Netzwerke (KNN) erzielen beeindruckende Ergebnisse bei Mustererkennungsaufgaben. Kürzlich wurde mit Hilfe der Time-to-First-Spike-Kodierung eine Möglichkeit zur Implementierung von KNN-Algorithmen in spikenden neuronalen Netzen abgeleitet. Diese Kodierung ermöglicht auf neuromorpher Hardware die Verwendung etablierter Methoden, wie z.B. "Backpropagation of Error". Das Ziel dieser Arbeit ist es, das Lernen mit Time-to-First-Spike-Kodierung auf dem BrainScaleS (BSS) Wafer-Scale System zu realisieren und dabei die Fähigkeit der schnellen und energieeffizienten analogen Emulation zu nutzen. Analoge Schaltkreise beeinträchtigen jedoch die vollständige Kontrolle der Neuronenparameter und verursachen das Rauschen von Spikezeiten, was beides der Verwendung exakter und differenzierbarer Gleichungen, wie im ursprünglichen Ansatz, abträglich ist. Diese Arbeit schafft einen Rahmen, der die ursprüngliche Idee signifikant an das Neuronenmodell von BSS anpasst. Die genannten Herausforderungen werden durch Untersuchungen zur Reproduzierbarkeit von Spikezeiten auf Hardware, umfangreichen Softwaresimulationen und der Durchführung von Lernen mit Time-to-First-Spike-Kodierung auf Hardware sorgfältig angegangen. Dies ermöglicht die Demonstration der visuellen Datenklassifizierung bei hohen Geschwindigkeiten unter Verwendung vollständig hardwareemulierter Netzwerke mit extrem spärlichen Antworteigenschaften und ebnet so den Weg zu hochskalierten Umsetzungen für eine komplexe Mustererkennung, die den Stand der Technik hinsichtlich Geschwindigkeit und Energieeffizienz in Frage stellen kann.

Contents

1 Introduction

Possessing a realistic model of one's surrounding is of critical importance for survival and reproduction. The resulting evolutionary pressure has consequently moulded our brains into pattern recognition engines of unparalleled efficiency. It is thus only natural to turn to the brain for inspiration on how to build artificial machines of comparable capability.

In the field of machine learning, aspects of cortical circuitry and dynamics serve as inspiration for building *artificial neural network*s (ANNs) [*Cireşan et al.*, 2010; *Krizhevsky et al.*, 2012]. Over the last decade, these have become the predominant class of algorithms for solving pattern recognition tasks [*LeCun et al.*, 2015]. Interestingly, neither the idea of ANNs [*Rosenblatt*, 1958] nor the algorithms used for training these networks [*Rumelhart et al.*, 1986] are new. It has much rather been the availability of improved computing devices, especially *graphics processing unit*s (GPUs) for parallel matrix multiplications, that has accelerated the simulation of such networks by orders of magnitude, thus enabling their study and deployment at the necessary scale for becoming competitive against alternative solutions.

Compared to ANNs, *spiking neural network*s (SNNs) lie significantly closer to their biological archetypes. Unlike ANNs, SNNs are dynamical systems operating in continuous time. Furthermore, information exchange in SNNs is mediated by their name-giving, all-or-none, singular events called spikes, in contrast to the floating-point precision arithmetic employed by ANNs.

Such spiking networks form the computational architecture instantiated by most so-called neuromorphic systems [*Mead*, 1990]. Over the past decades, many neuromorphic platforms have emerged [*Indiveri et al.*, 2011], and in particular, the *BrainScaleS* (BSS) wafer-scale system developed in Heidelberg as part of [*FACETS*, 2010], [*Schemmel et al.*, 2010], and [*Human Brain Project*, 2013] stand out as particularly interesting candidates, given their unique characteristics and capabilities. The BSS system uses a clever combination of analogue circuits for emulation of neuron dynamics and high-bandwidth digital communication between neurons [*Klähn*, 2017], while at the same being energy-efficient compared to traditional simulations [*Müller*, 2014]. The analogue emulation results in a speed-up factor of 10^4, meaning that 10 s in biological time can be simulated in 1 ms wall-clock time. Crucially, this acceleration factor does not depend on network size, unlike for simulation times on classical CPUs, which typically scale quadratically with the number of neurons in the simulated network.

The BrainScaleS system has previously been used for emulating networks that use single-spike encoding schemes, such as in the case of neural sampling [*Kungl*, 2016; *Dold et al.*, 2017]. However, the emulation of deep, feed-forward, backpropagation-trained

networks has so far relied on the use of firing rates for encoding information [*Schmitt et al.*, 2017; *Petrovici et al.*, 2017].

Rather than average firing rates, the timing of single spikes can also be used to transmit information in feed-forward networks, thereby abolishing the wait time needed to gather spike statistics as well as reducing energy consumption by reducing the number of processed spikes. While other ideas for training spiking networks exist [e.g. *Bohte et al.*, 2002; *Zenke and Ganguli*, 2018; *Gütig and Sompolinsky*, 2006; *Neftci et al.*, 2019], the approach presented in [*Mostafa*, 2017] is particularly compelling for its use of deep networks on big data sets that can make use of the size of BSS. In [*Mostafa*, 2017] all information is embedded in spike times and the category assigned to a particular input is determined by the neuron in the label layer that spikes first. The networks in [*Mostafa*, 2017] are trained on the logical `XOR` and the *Modified National Institute of Standards and Technology* (MNIST) data set [*LeCun et al.*, 1998] and achieve respectable classification results. The crucial ingredient for training the networks is an exact and differentiable relation for the time-to-first-spike which enables the usage of traditional machine learning algorithms such as error backpropagation [*Rumelhart et al.*, 1986] for training these networks. A key result of [*Mostafa*, 2017] is that classification is finished even before most neurons in hidden layers get to spike.

Prior to the implementation of time-to-first-spike coded learning on BSS, however, a number of specific challenges needed to be met. First, the neuron model used in [*Mostafa*, 2017] is different from the one instantiated on BSS, thus requiring significant adaptation of the framework. Second, [*Mostafa*, 2017] notices a high dependence on individual components and states explicitly that no neuron is redundant. While the analogue nature of BSS enables high-speed and energy-efficient computation, this comes at a price in the form of noise, partial unreliability and only incomplete control over neuron parameters. Part of the aim of this thesis was to verify the suitability of Mostafa's approach for neuromorphic hardware with analogue components.

In this thesis, the capability of BSS to train SNNs with a time-to-first-spike coding is demonstrated. The underlying theory is derived and tested extensively in software simulations. Sufficiency of the spike time accuracy of the hardware is examined. Starting from random initialisations, different networks are trained on BSS, including a proof-of-principle classification of a reduced version of the MNIST data set.

The thesis is structured as follows. In Chapter 2, the training of an SNN as in [*Mostafa*, 2017] is introduced. In the process, I define the parameters and methods necessary for training networks on hardware.

In Chapter 3, BSS is described. The focus will be on the constraints imposed by the hardware that are in contrast to the model in [*Mostafa*, 2017], as well as potential sources of error for the approach with time-to-first-spike coding.

In Chapter 4, I describe a model that answers the challenges detailed in the previous

Chapter. Furthermore, an analysis of the expected behaviour of relevant quantities is discussed.

In Chapter 5, the software implementation of the models is discussed. The data sets used in training are described as well.

In Chapter 6, the models described in Chapter 4 are put to the test for their predictive power for the time-to-first-spike, both for simulations in software and emulations on BSS.

In Chapter 7, trainings of SNNs in software are examined. Here the learning process is described in detail and underlying concepts of the training are determined, with a focus on the stability of learning. An important part of this Chapter is to incorporate hardware-realistic features into the software simulations.

In Chapter 8, spiking networks are trained on BSS. The investigation contains sanity checks of the training to ensure proper functionality and includes training to classify a reduced MNIST data set. All networks are initialised with random weights and trained exclusively on hardware.

2 Training Integrate-and-Fire Neurons with Time-to-First-Spike Coding

The main finding in [*Mostafa*, 2017] is that pattern recognition for *integrate-and-fire* (IF) neurons with time-to-first-spike coding is possible and allows for fast classification. The concepts needed to train a *spiking neural network* (SNN) with this coding are introduced in this Chapter.

2.1 Time-to-First-Spike Coding

Mostafa uses layered networks with an input layer, one or more hidden layers, and the label layer, see Fig. 2.1 for an example. A layer here is a population of neurons that receive input from the previous layer and pass their spikes to the next layer. Patterns (Fig. 2.2) are passed as spikes to the input layer, and the neurons propagate the information modulated by the strength of their connections, called the synaptic weights. In the label layer, each neuron is assigned one class. Information is encoded in the time it takes a neuron to spike, measured from the start of the inputs. This coding is called time-to-first-spike coding.

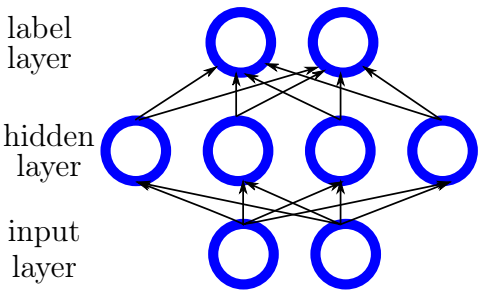


Figure 2.1: Network layout taken from [*Mostafa*, 2017].

The goal of training is to make that neuron spike first that is associated with the correct class. An input is deemed misclassified if another neuron from the label layer, one not assigned the correct class, spikes first. Thus, the training task is to learn to spike with the correct class when presented with input.

Training means optimisation of degrees of freedom for best possible classification rate or, equivalently, lowest error rate. For SNNs in [*Mostafa*, 2017] and in this thesis, the degrees of freedom that are updated are synaptic weights.

Figure 2.2: 100 example patterns from the MNIST [*LeCun et al.*, 1998] test set, see further Section 5.2.

2.2 Energy and Gradient Descent

The energy is chosen so that smaller energy corresponds to a higher rate of classification. With this choice of energy, the highest possible classification rate correlates with a local minimum of the energy in the high-dimensional space of parameters. With an energy E that fulfils this requirement (see Section 4.4), it is possible to train the network via gradient descent.

Gradient descent calculates the updates Δw to the synaptic weights w in direction of steepest descent

$$\Delta w \propto -\frac{\partial E}{\partial w}. \tag{2.1}$$

Because the energy is a function of each label neuron's time-to-first-spike, a differentiable relation for the time-to-first-spike is necessary. In a deep network, error backpropagation [*Rumelhart et al.*, 1986] is used to calculate the updates for deeper layers.

The proportionality constant for Eq. (2.1) is called the learning rate η. The value of η is important as it is responsible for the speed of learning. It plays a crucial role in whether the learning succeeds, too. If η is too large, the weight updates are too large. This may impede learning by making jumps that are too big. If η is smaller, learning takes more steps. If the learning rate is too small, learning may be too slow to see the learning success. In [*Mostafa*, 2017], an exponentially decaying learning rate is used to aid the learning.

2.3 The Integrate-and-Fire Neuron Model

Computational units in the SNNs in [*Mostafa*, 2017] are *integrate-and-fire* (IF) neurons with *current-based* (CuBa) synapses and an exponential synapse kernel. For a general introduction into theoretical neuroscience and different neuron models, refer to [*Petrovici*, 2015; *Dayan and Abbott*, 2001; *Gerstner and Kistler*, 2002]. The IF neuron model

describes neurons as simple units that add up (integrate) their input current I onto their membrane V and emit a spike (fire) when the membrane crosses a threshold V_{th}. The input is weighted by synaptic weights w.

The time evolution of the membrane voltage V^k of IF neurons indexed with k is governed by the differential equation

$$C_{\text{m}} \frac{\partial V^k(t)}{\partial t} = I^k_{\text{syn}}(t), \tag{2.2}$$

with the membrane capacitance C_{m}.

Inputs are received in form of spikes that induce a synaptic current. The full synaptic current is given by a weighted sum over the synapses from neurons i to the neuron k with the respective weight w_{ki}:

$$I^k_{\text{syn}}(t) = \sum_i w_{ki} \kappa(t - t_i). \tag{2.3}$$

Using the Heaviside θ function and the synaptic time constant τ_{syn}, the exponential synapse kernel has the form

$$\kappa(t) = \theta(t) \exp\left(-\frac{t}{\tau_{\text{syn}}}\right). \tag{2.4}$$

For the IF neurons, the capacitance is included in the weights for simplicity.

With the convenient initial condition of an inactive neuron prior to any input spikes

$$V(\bar{t}) = 0 \text{ for } \bar{t} \leq t_k \forall k \tag{2.5}$$

the differential equation (Eq. (2.2)) has the solution

$$V^k(t) = \sum_i w_{ki} \theta(t - t_i) \left(1 - \exp\left(-\frac{t - t_i}{\tau_{\text{syn}}}\right)\right). \tag{2.6}$$

From the time evolution of the membrane voltage, the time-to-first-spike can be determined as a function of the input times and weights.

For an IF neuron, the membrane voltage is fixed for the refractory time τ_{ref} after an emitted spike. This refractory time is set to infinity in [*Mostafa*, 2017] in order to prevent additional spikes and to stick with single spike coding.

2.4 The Time-to-First-Spike and its Derivative

The neuron elicits a spike when the membrane crosses the threshold, implicitly defining the time-to-first-spike of neuron k as

$$t_k \text{ s.t. } V^k(t_k) = V_{\text{th}} \tag{2.7}$$

With the set

$$C_k = \{i : t_i < t_k\} \tag{2.8}$$

the implicit equation can be written as

$$V_{\text{th}} = \sum_{i \in C_k} w_{ki} \left(1 - \exp\left(-\frac{t_k - t_i}{\tau_{\text{syn}}}\right)\right). \tag{2.9}$$

Solving for the time of the outgoing spike, the result is

$$e^{\frac{t_k}{\tau_{\text{syn}}}} = \frac{\sum_{i \in C_k} w_{ki} e^{\frac{t_i}{\tau_{\text{syn}}}}}{\sum_{i \in C_k} w_{ki} - V_{\text{th}}}. \tag{2.10}$$

With the variable transformation

$$z = \exp\left(\frac{t}{\tau_{\text{syn}}}\right) \tag{2.11}$$

the result can be simplified to

$$z_k = \frac{\sum_{i \in C_k} w_{ki} z_i}{\sum_{i \in C_k} w_{ki} - V_{\text{th}}}. \tag{2.12}$$

For this time-to-first-spike, the derivatives are

$$\frac{\partial z_k}{\partial w_{ki}} = \frac{z_i - z_k}{\sum_j w_{kj} - V_{\text{th}}} \mathbb{1}_{i \in C_k} \qquad \frac{\partial z_k}{\partial z_i} = \frac{w_{ki}}{\sum_j w_{kj} - V_{\text{th}}} \mathbb{1}_{i \in C_k}, \tag{2.13}$$

where $\mathbb{1}_{i \in C_k}$ is equal to 1 if $i \in C_k$, i.e. $t_i < t_k$, and 0 otherwise.

2.5 Ensure Sufficient Spiking

Additional mechanisms might be needed to ensure training success. One mechanism used in [*Mostafa*, 2017] is increasing the weight of the synapses to ensure spiking of each neuron. A neuron or whole network without a spikes is called quiescent. Preventing quiescent networks is important for approaches with time-to-first-spike coding, as the spikes are needed to optimise the weights.

Because the neurons have infinite memory, from Eq. (2.6) it can be concluded that for exactly one spike per presynaptic neuron, the following is a sufficient condition for an output spike

$$\sum_i w_{ki} > V_{\text{th}}. \tag{2.14}$$

This is enforced by adding to the energy the term

$$\sum_k \max\left[0, V_{\text{th}} - \sum_i w_{ki}\right]. \tag{2.15}$$

This term, which is called weight sum cost in [*Mostafa*, 2017], checks Eq. (2.14) for every neuron and potentially increases the weights of that neuron.

A further mechanism is to add a L2 norm of the weights to the energy. The goal is to prevent the network from being dependent on single inputs due to too large weights.

An additional step is to normalise the gradients before one uses them with the back propagation for calculating the update. In [*Mostafa*, 2017], this normalisation is described as ensuring the Frobenius norm of the gradients is not larger than some value.

2.6 Results in [*Mostafa*, 2017]

In [*Mostafa*, 2017], the approach is used on two problems, classifying the logical `XOR` and classifying handwritten digits from the MNIST database (Section 5.2).

No training progress is shown in [*Mostafa*, 2017].

2.6.1 XOR

A setup like in Fig. 2.1 learns reliably. [*Mostafa*, 2017] reports that the network is able to learn the non linear problem because for different input patterns the causal sets C_k are different.

2.6.2 MNIST

Mostafa uses two network architectures for MNIST classification. One architecture has one hidden layer with 800 neurons, the other architecture has two hidden layers with 400 neurons each. The networks are trained for 100 epochs, i.e. the complete training set is gone through 100 times. A neuron that fires a starting spike to all neurons in all layers at the start of an input improves accuracy. To increase the testing performance and combat overfitting of the training set, [*Mostafa*, 2017] trains with both non-noisy and noisy input.

The test error rate for MNIST is between 2.45% and 3.08% depending on the architecture and noise. This is several percent above a linear classifier[1], but below the rates of elaborate artificial neural networks [e.g. *Cireşan et al.*, 2010].

[*Mostafa*, 2017] shows that the classification happens before the majority of hidden neurons spike. On a system as *BrainScaleS* (BSS) this is intriguing, as the speed-up (Chapter 3) accelerates classification further.

[1] Linear classification accuracy is tested with the class `LogisticRegression` of the module `sklearn.linear_model`, see [*Pedregosa et al.*, 2011].

The classification speed is enforced by favouring early spikes. According to [*Mostafa*, 2017], reducing the number of spikes nudges the network to use every spike in the best possible way.

All of this makes the design suitable to implement on neuromorphic hardware, as is noted in [*Mostafa*, 2017]. On a neuromorphic system, the parallel nature can be fully exploited to have fast classification.

3 The BrainScaleS Neuromorphic Platform

The BSS platform is a mixed-signal accelerated neuromorphic system using a *very-large-scale integration* (VLSI) design. Development started originally in *Fast Analog Computing with Emerging Transient States* (FACETS) and BSS and continues currently in the *Human Brain Project* (HBP).

In this Chapter, the relevant structures of the hardware and the physical modelling approach [*Schemmel et al.*, 2010] are described, starting with its smallest components and ending with the full wafer module. The overview given here claims in no way to be exhausting, and for more features as well as more comprehensive descriptions refer to [*HBP SP9 partners*, 2014; *Schemmel et al.*, 2010; *Millner*, 2012; *Jeltsch*, 2014; *Müller*, 2014; *Koke*, 2017].

3.1 The Analogue Circuits

The idea of physical modelling used on BSS is that neurons are implemented as analogue circuits that emulate the behaviour of neurons. This is in contrast to traditional neuron simulators on von Neumann architectures solving differential equations. Like biological neurons, in both approaches the neurons react to input in the form of spikes and, depending on the given input, elicit spikes themselves.

The circuits show the same behaviour as neurons because its components are chosen in order for the circuit to be governed by the same differential equations. Due to the electronic rather than biological nature, the time constants in the circuits are orders of magnitude smaller, making the emulations faster by a speed-up of factor 10^4, compared to biology. This means that 10 ms in the biological time domain can be emulated in 1 μs in the real-time domain[1].

The neuron model implemented on the hardware are *leaky integrate-and-fire* (LIF) neurons with *conductance-based* (CoBa) synapses. In fact, the neuron model on BSS is the *adaptive exponential integrate-and-fire* (AdEx) model [*Gerstner and Brette*, 2009; *Brette and Gerstner*, 2005], however this neuron model can be configured as LIF neurons [*Koke*, 2017] as is done here. The LIF neuron and CoBa synapses are described in Chapter 4.

The neural circuits are subject to variations in the production steps, causing circuit-to-circuit variations in the parameters. These variations are called fixed-pattern noise.

[1] In this thesis, units are always implied to be in the biological domain if not stated otherwise.

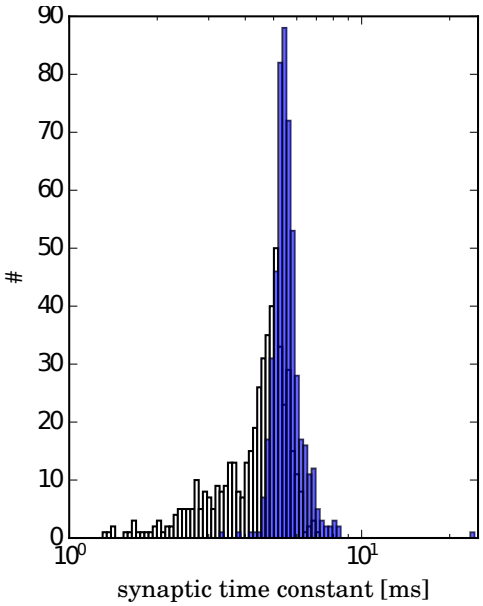


Figure 3.1: Effect of calibration of the synaptic time constant. While the measurements of the synaptic time constant of all neurons on a HICANN without calibration (white) are spread out, the distribution of the calibrated measurement (blue) is narrowed down. Plot taken from [*Schmitt et al.*, 2017].

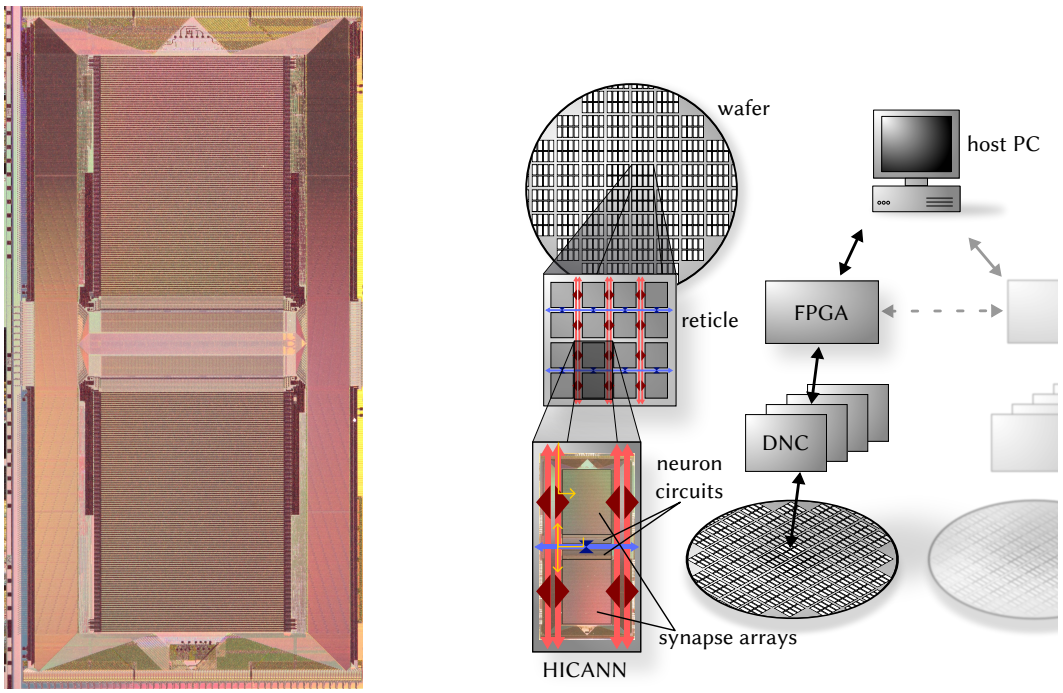
The circuits can be adapted by settable analogue parameters which are stored on-chip using analogue storage units called *floating gate*s (FGs). The process of setting the parameters introduces trial-to-trial variations [*Koke*, 2017; *Kungl*, 2016]. These noise sources imply neuron-to-neuron variations as well as trial-to-trial variations for the same neuron after rewriting the FGs. By calibration [*Koke*, 2017] this noise can be reduced but not eliminated (Fig. 3.1).

All neuron parameters are set with analogue parameters, this includes the time constants. In [*Mostafa*, 2017] single spikes were enforced by setting an infinite refractory time τ_{ref}. In simulations and emulations, this is equivalent to τ_{ref} equal to the observed time span. I choose this time span heuristically as $10\tau_{\text{syn}} = 100\,\text{ms}$. On hardware, however, this value is out of scope for τ_{ref}. For smaller values, multiple spikes can occur for a neuron (e.g. Fig. 7.18b) but with a large enough value of τ_{ref} the relevant dynamics happen before neurons are able to spike more than once. In this thesis, the parameter is set to $\tau_{\text{ref}} = 20\,\text{ms}$.

3.2 The HICANN Chip

The relevance of BSS is in part due to its size, as it combines a very large number of neuronal circuits. The circuits are arranged in a structured fashion, and the next larger structure above the analogue circuits is the *high input count analog neural network* (HICANN) chip (Fig. 3.2a). One HICANN contains 512 neuron circuit, also termed *dendrite membrane*s (DenMems) in this context. The number of possible inputs per circuit is limited by 224, but DenMems can be combined forming multi-compartment neurons, up to 64 can be short circuited to allow for more inputs [*Millner*, 2012].

The strength of synaptic inputs is determined among others by digital 4-bit weights set independently for each synapse, and an analogue parameter g_{max} scaling the effective



(a) Detailed view of a single HICANN chip. The two large synaptic arrays at the top and bottom are most prominent, the neuron circuits are located in between. Photo taken from [*Klähn*, 2017].

(b) Schematic view of the communication structure of HICANNs with a host computer. The red and blue lines are used by the L1 communication. The yellow lines is an example for a route a spike could travel on the chip. For routes in my network see the visualisation Fig. 8.1. Taken from [*Petrovici et al.*, 2014].

Figure 3.2: Photo of a HICANN chip and scheme of its communication structure on- and off-wafer.

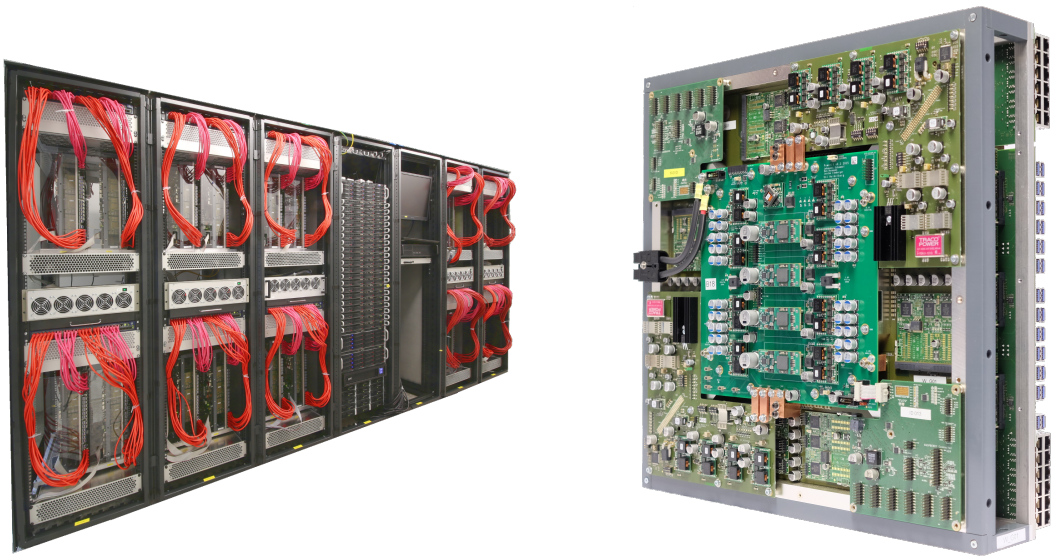
conductance. For training, only the 4-bit weights are rewritten.

3.3 The Wafer Module

The next structure is the wafer module. It is partitioned into 48 reticles that each consist of 8 HICANNs and each have an associated *field-programmable gate array* (FPGA) responsible for input and output, i.e. configuration and data extraction. In total, up to 180 000 neural circuits and 40 000 000 synapses can be emulated on a wafer module [*Schemmel et al.*, 2010].

All of the HICANNs on one wafer module are not single chips but produced on the exact same material. One reticle is produced with a single photolithographic mask. Communication between reticles is enabled by a post-processing step connecting the reticles [*Zoschke et al.*, 2017].

All communication between neurons is solely through spikes. The spikes are conveyed



(a) A number of wafer modules make up the BSS system.

(b) View of a single assembled module. For operating the wafer, power supply is needed as well as FPGAs and additional supporting infrastructure to interact with a host computer.

Figure 3.3: The assembled wafer modules, photos taken from [*Schmitt et al.*, 2017].

digitally but translated to an analogue signal prior to injection into the analogue neural circuits, hence the name mixed-signal hardware for BSS.

HICANN-to-HICANN communications happen via the asynchronous *layer-1* (L1). For off-wafer communication, time-stamped spikes are transported via the synchronous *layer-2* (L2). The capabilities of the communication structures are detailed in [*Klähn*, 2017] and potentially spike loss can occur in each layer. In this work no spike loss is observed due to the single-spike approach.

Two analogue readouts per HICANN can extract membrane voltage traces via *analog-to-digital converter*s (ADCs) to the host computer.

The interface allows high-level neuron descriptions while at the same time providing access to calibration and mapping, the process of writing a network to the hardware.

The user facing interface for the hardware is PyHMF [*Jeltsch*, 2014], which is an implementation of the *python neural networks* (PyNN) [*Davison et al.*, 2008] *application programming interface* (API). This interface allows a network-level description of the system, abstracting away the lower-level parts of the configuration and hardware handling. In order to handcraft the neuron placement on hardware, marocco is used manually, everything else is left to the software.

4 Derive, Define and Describe

In the previous chapter, the requirement of modifications of the approach in [*Mostafa*, 2017] for an implementation on BSS was shown. In particular, the algorithm presented in Chapter 2 has to be adapted to LIF neurons, CoBa synapses have to be treated correctly, and disturbances to an ideal model like spike-timing jitter and fixed-pattern noise need to be addressed.

First, the problem of finding the analytic expression for LIF neurons with CuBa synapses is described. Then, solutions in two special cases are derived. Afterwards, an approximation for CoBa synapses is introduced. The following Section elaborates on the choice of the energy. The Chapter ends with a substitution for the weight sum cost of [*Mostafa*, 2017].

4.1 The Leaky-Integrate-and-Fire Neuron Model

The BSS wafer-scale system does not feature IF neurons but LIF neurons. Thus, an analytical expression for the time-to-first-spike in LIF neurons is derived.

The main difference between a leaky IF neuron from a non-leaky IF neuron is the leak term that pulls the membrane voltage towards the leak potential E_L. Together with the threshold voltage V_th this defines a fixed scale, that, without loss of generality, can be chosen as $V_\mathrm{th} = 1$ and $E_\mathrm{L} = 0$ for simplicity.

The decay back to the leak potential happens in an exponential fashion, determined by the membrane time constant τ_m. This decay is combined with the synaptic current I_syn, and thus the membrane voltage of a LIF neuron is governed by the differential equation

$$\tau_\mathrm{m} \frac{\partial V}{\partial t} = -V + \frac{I_\mathrm{syn}}{g}. \tag{4.1}$$

The membrane time constant is given by the capacitance C_m and conductance g as $\tau_\mathrm{m} = \frac{C_\mathrm{m}}{g}$. The input current is a weighted superposition of the spike kernel as before

$$I^k_\mathrm{syn}(t) = \sum_i w_{ki} \kappa(t - t_i) \tag{4.2}$$

$$= \sum_i w_{ki} \theta(t - t_i) e^{-\frac{t-t_i}{\tau_\mathrm{syn}}}. \tag{4.3}$$

Plugging the synaptic current into Eq. (4.1) gives the full CuBa differential equation

$$\tau_{\mathrm{m}} \frac{\partial V^k}{\partial t} = -V^k + \frac{1}{g} \sum_i w_{ki} \theta(t - t_i) e^{-\frac{t-t_{i,r}}{\tau_{\mathrm{syn}}}}. \tag{4.4}$$

The homogeneous part of the equation is a simple declining exponential, thus the full solution is given by solving

$$V^k(t) = e^{-\frac{t}{\tau_{\mathrm{m}}}} \int^t \mathrm{d}s \, I^k_{\mathrm{syn}}(s) \frac{e^{\frac{s}{\tau_{\mathrm{m}}}}}{g\tau_{\mathrm{m}}}. \tag{4.5}$$

The integral is carried out to arrive at

$$V^k(t) = \frac{1}{g\tau_{\mathrm{m}}} e^{-\frac{t}{\tau_{\mathrm{m}}}} \sum_i w_{ki} \theta(t - t_i) e^{\frac{t_i}{\tau_{\mathrm{syn}}}} \int_{t_i}^t \mathrm{d}s \, e^{-s\left(\frac{1}{\tau_{\mathrm{syn}}} - \frac{1}{\tau_{\mathrm{m}}}\right)} \tag{4.6}$$

$$= \frac{1}{C_{\mathrm{m}}} \frac{\tau_{\mathrm{m}}\tau_{\mathrm{syn}}}{\tau_{\mathrm{m}} - \tau_{\mathrm{syn}}} \sum_i w_{ki} \theta(t - t_i) \left(e^{-\frac{t-t_i}{\tau_{\mathrm{m}}}} - e^{-\frac{t-t_i}{\tau_{\mathrm{syn}}}} \right) \tag{4.7}$$

$$= \frac{1}{C_{\mathrm{m}}} \frac{\tau_{\mathrm{m}}\tau_{\mathrm{syn}}}{\tau_{\mathrm{m}} - \tau_{\mathrm{syn}}} \sum_{i \in C_k} w_{ki} \theta(t - t_i) \left(e^{-\frac{t-t_i}{\tau_{\mathrm{m}}}} - e^{-\frac{t-t_i}{\tau_{\mathrm{syn}}}} \right). \tag{4.8}$$

For simplicity I take a vanishing leak term and initial conditions as above, $V_k(0) = 0$. Furthermore, I use the set C_k from above again, and single spike times. One can check that for a slow membrane, the limit $\lim_{\tau_{\mathrm{m}} \to \infty} = \lim_{g \to 0}$, one recovers both the differential equation Eq. (2.2) and the membrane voltage Eq. (2.6) of the IF model.

The spike time t_k defined by Eq. (2.7) is given by

$$V_{\mathrm{th}} = V^k(t_k) = \frac{1}{C_{\mathrm{m}}} \frac{\tau_{\mathrm{m}}\tau_{\mathrm{syn}}}{\tau_{\mathrm{m}} - \tau_{\mathrm{syn}}} \sum_{i \in C_k} w_{ki} \theta(t_k - t_{i,r}) \left(e^{-\frac{t_k - t_i}{\tau_{\mathrm{m}}}} - e^{-\frac{t_k - t_i}{\tau_{\mathrm{syn}}}} \right). \tag{4.9}$$

Because the spike time t_k appears in two exponents with different coefficients, no explicit formula for t_k exists without loss of generality. Two special cases are chosen to derive formulas for the spike time.

4.2 The Time-to-First-Spike and its Derivatives

It is instructive to show how the membrane voltage behaves for different values of τ_{m}. Figure 4.1 shows the time evolution of the voltage for example values of τ_{m}. Time evolutions of the membrane voltage are also called membrane or voltage trace.

The key to finding a relation for t_k is to have a relation between the coefficients in the two exponents, a relation between τ_{syn} and τ_{m}. With the definition of the ratio

$$\rho := \tau_{\mathrm{m}} / \tau_{\mathrm{syn}}, \tag{4.10}$$

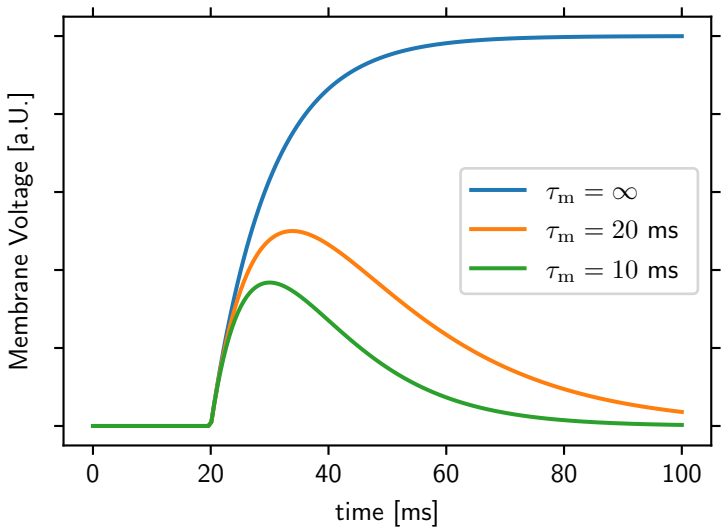


Figure 4.1: Comparison of different membrane time constants.
Membrane traces for different values of $\tau_m \in \{\infty, 20\text{ms}, 10\text{ms}\}$ (blue, orange and green) and corresponding g, keeping all other parameters fixed, especially $\tau_{syn} = 10\,\text{ms}$. Apart from returning to the leak potential faster, the height of the trace is reduced.

the two solvable cases are the ratios $\rho = 1$ and $\rho = 2$. Because Eq. (4.8) is symmetric in τ_{syn} and τ_m, a ratio of $\rho = 2$ is equivalent to the ratio $\rho = 0.5$ in this theoretical analysis. The same is true for the design in [*Mostafa*, 2017], for which the two ratios are $\rho = 0$ and $\rho = \infty$.

The formula for the time-to-first-spike for $\rho = 1$ with $\tau_m = \tau_{syn}$ is called *equal-time formula*, and this name will be used to refer to it. In the other case with $\rho = 2$ and $\tau_m = 2\tau_{syn}$, the relation is called *double-time formula*.

4.2.1 $\rho = 1$, $\tau_m = \tau_{syn}$ and the Equal-Time Formula

Preparations

In the limit $\tau_m \to \tau_{syn}$, L'Hôpital's rule gives

$$\lim_{\tau_m \to \tau_{syn}} V^k(t) = \sum_{i \in C_k} \frac{w_{ik}}{C_m} \theta(t - t_i) \lim_{\tau_m \to \tau_{syn}} \tau_m \tau_{syn} \frac{e^{-\frac{t-t_i}{\tau_m}} - e^{-\frac{t-t_i}{\tau_{syn}}}}{\tau_m - \tau_{syn}} \tag{4.11}$$

$$\overset{\text{L'Hôpital}}{=} \sum_{i \in C_k} \frac{w_{ik}}{C_m} \theta(t - t_i) e^{-\frac{t-t_i}{\tau_{syn}}} (t - t_i). \tag{4.12}$$

This is solved for the spike time t_k of the neuron

$$V_{th} = V^k(t_k) = \sum_{i \in C_k} \frac{w_{ki}}{C_m} e^{-\frac{t_k - t_i}{\tau}} (t_k - t_i) \tag{4.13}$$

by reordering and scaling accordingly, with $x = \frac{t_k}{\tau}$, to get

$$0 = \sum_{i \in C_k} \frac{w_{ki}}{C_\mathrm{m}} e^{\frac{t_i}{\tau}} t_i - \sum_{i \in C_k} \frac{w_{ki}}{C_\mathrm{m}} e^{\frac{t_i}{\tau}} \tau \frac{t_k}{\tau} + V_\mathrm{th} e^{\frac{t_k}{\tau}} \tag{4.14}$$

$$= a + bx + ce^x, \tag{4.15}$$

employing the following definitions

$$a := \sum_{i \in C_k} \frac{w_{ki}}{C_\mathrm{m}} e^{\frac{t_i}{\tau}} t_i \qquad b := -\sum_{i \in C_k} \frac{w_{ki}}{C_\mathrm{m}} e^{\frac{t_i}{\tau}} \tau \qquad c := V_\mathrm{th}. \tag{4.16}$$

With the transformation $y = x + \frac{a}{b}$, Eq. (4.15) is rearranged to

$$0 = a + bx + ce^x \tag{4.17}$$

$$= by + ce^y e^{-\frac{a}{b}} \tag{4.18}$$

$$\frac{c}{b} e^{-\frac{a}{b}} = -ye^{-y} \tag{4.19}$$

The equation

$$z = h \cdot e^h \tag{4.20}$$

is solved by $h = W(z)$ with the differentiable Lambert W function. For more on this function, see below on Page 19. Using the Lambert function for $z = \frac{c}{b} e^{-\frac{a}{b}}$ and $h = -y$ one gets

$$W\left(\frac{c}{b} e^{-\frac{a}{b}}\right) = -y = -x - \frac{a}{b}. \tag{4.21}$$

Through rearranging, a formula for the time-to-first-spike is recovered.

$$t_k = \tau x = \tau \left(-W\left(\frac{c}{b} e^{-\frac{a}{b}}\right) - \frac{a}{b}\right). \tag{4.22}$$

The derivative of the LambertW function is determined by deriving both sides of Eq. (4.20) wrt. z, resulting in

$$W'(z) := \frac{\mathrm{d}W}{\mathrm{d}z}(z) = \frac{1}{e^{W(z)} + z}. \tag{4.23}$$

Altogether, an analytical and differentiable relation for the time-to-first-spike was obtained. This relation can now be differentiated.

Derivatives

The derivatives of this function are more complicated than in the design in [*Mostafa*, 2017]. To keep the equations simple, I use the chain rule and write down the derivatives separately.

For the derivatives, a naming scheme has to be fixed. The presynaptic neurons are indexed by i, the postsynaptic neuron by k. The time-to-first-spike of the postsynaptic neuron is

$$t_k = \tau \left(-W \left(\frac{c_k}{b_k} e^{-\frac{a_k}{b_k}} \right) - \frac{a_k}{b_k} \right) \tag{4.24}$$

with specific variables a_k, b_k, and c_k each depending on the weights w_{ki}, the presynaptic spike times t_i and the causal set C_k like

$$a_k = \frac{1}{C_\mathrm{m}} \sum_{i \in C_k} w_{ki} e^{\frac{t_i}{\tau}} t_i \qquad b_k = -\frac{\tau}{C_\mathrm{m}} \sum_{i \in C_k} w_{ki} e^{\frac{t_i}{\tau}} \qquad c_k = V_\mathrm{th}. \tag{4.25}$$

For each, I need the derivative wrt. to the weights w_{ki} and for the backpropagation also wrt. t_i. I arrive at

$$\frac{\partial a_k}{\partial t_i} = +\frac{1}{C_\mathrm{m}} w_{ki} e^{\frac{t_i}{\tau}} \left(1 + \frac{t_i}{\tau} \right) \mathbb{1}_{i \in C_k} \qquad \frac{\partial a_k}{\partial w_{ki}} = \frac{1}{C_\mathrm{m}} t_i e^{\frac{t_i}{\tau}} \mathbb{1}_{i \in C_k} \tag{4.26}$$

$$\frac{\partial b_k}{\partial t_i} = -\frac{1}{C_\mathrm{m}} w_{ki} e^{\frac{t_i}{\tau}} \mathbb{1}_{i \in C_k} \qquad \frac{\partial b_k}{\partial w_{ki}} = -\frac{\tau}{C_\mathrm{m}} e^{\frac{t_i}{\tau}} \mathbb{1}_{i \in C_k} \tag{4.27}$$

The derivatives of the spike time is best expressed by its differential. Using the definition $z_k = \frac{c_k}{b_k} e^{-\frac{a_k}{b_k}}$ this can be written as

$$\mathrm{d}t_k = \left[\frac{\tau}{b_k} \left(W'(z_k) z_k \left(1 - \frac{a_k}{b_k} \right) + \frac{a_k}{b_k} \right) \right] \mathrm{d}b_k \tag{4.28}$$

$$+ \left[\frac{\tau}{b_k} \Big(W'(z_k) z_k - 1 \Big) \right] \mathrm{d}a_k. \tag{4.29}$$

The LambertW Function

In the framework (Section 5.1), the implementation of the LambertW function from `scipy.special.lambertw` is used. That code itself is based on [*Corless et al.*, 1996].

The defining equation (Eq. (4.20)) is multivalued. Because the spike times are real, as are the weights and other parameters, narrowed down for real-valued arguments there are 2 branches. Those two branches correspond to being left or right of the minimum in Fig. 4.2a. Following the naming in [*SciPy Documentation*], the 0 branch is to the right and real-valued for $-1/e < z$. The other branch, called -1 branch, is to the left of the minimum and real-valued for $-1/e < z < 0$.

In Fig. 4.2b the function and its derivative are shown.

In the calculations above, I implied using the 0 branch. For the reasoning I first look at the value of $h = -y$ for a single input spike and then generalise to multiple spikes.

In the setting of one input spike, there is no sum in the fraction $\frac{a}{b}$ and I get

$$h = -y = -x - \frac{a}{b} = -\frac{t_k}{\tau} + \frac{t_i}{\tau}. \tag{4.30}$$

The interpretation is the negative difference between the input spike and the output spike in units of τ. This difference is maximal when the membrane voltage barely touches the threshold. The maximum of Eq. (4.12) for one input spike happens at $t_i + \tau$. Thus, the maximal difference is τ, and the minimal value for the negative difference is $h = -1$. If the weight is larger, the difference is shorter, and therefore $h > -1$. This means, for one input spike the used branch is always $k = 0$.

This result is valid for more than one spike. Assuming multiple input spikes, first, I take an auxiliary spike with high enough weight at a short time before the output spike time t_k. For this setup, without any of the actual input spike, I know that I have $h > -1$. Second, I now add all the excitatory spikes, reducing the time difference. Now I remove the auxiliary spike. Because I know there is a spike happening, the excitatory inputs must suffice to reach the threshold. In the end, I add all inhibitory spike. The inhibitory spikes will increase the time difference, but not above τ or no spike will happen. All of these changes can be made infinitesimally, and because of continuity following from differentiability the solution will not leave the branch. The correct branch for all calculations is 0.

4.2.2 $\rho = 2$, $\tau_\mathrm{m} = 2\tau_\mathrm{syn}$ and the Double-Time Formula

Preparations

Given Eq. (4.8), setting

$$\tau_\mathrm{syn} = \frac{\tau_\mathrm{m}}{2} \tag{4.31}$$

in the relation for $V(t_k) = V_\mathrm{th}$ one recovers

$$0 = -C_\mathrm{m}\frac{\tau_\mathrm{m} - \tau_\mathrm{syn}}{\tau_\mathrm{m}\tau_\mathrm{syn}}V_\mathrm{th} + \sum_{i\in C_k} w_{ki} e^{\frac{t_i}{\tau_\mathrm{m}}} e^{-\frac{t_k}{\tau_\mathrm{m}}} - \sum_{i\in C_k} w_{ki} e^{2\frac{t_i}{\tau_\mathrm{m}}} e^{-2\frac{t_k}{\tau_\mathrm{m}}} \tag{4.32}$$

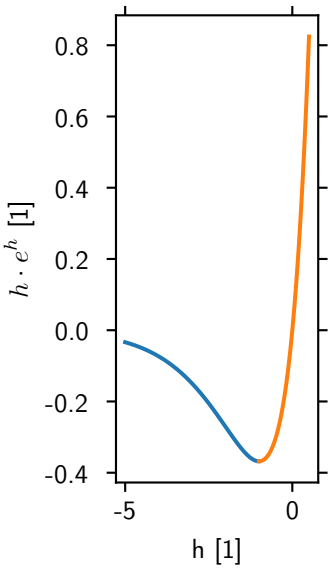
$$= c + by + ay^2, \tag{4.33}$$

with the definitions

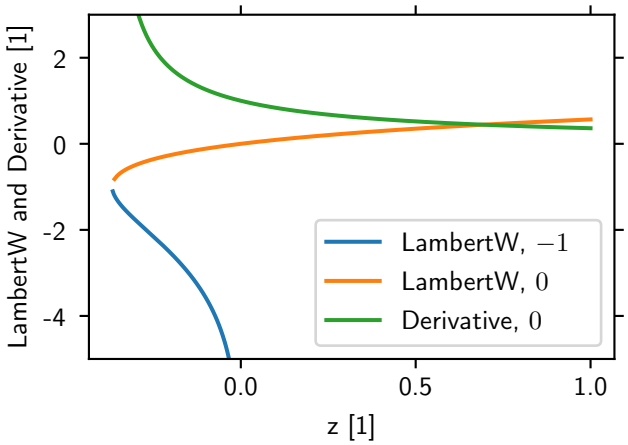
$$a = -\sum_{i\in C_k} w_{ki} e^{2\frac{t_i}{\tau_\mathrm{m}}} \qquad b = \sum_{i\in C_k} w_{ki} e^{\frac{t_i}{\tau_\mathrm{m}}} \qquad c = -C_\mathrm{m}\frac{\tau_\mathrm{m} - \tau_\mathrm{syn}}{\tau_\mathrm{m}\tau_\mathrm{syn}}V_\mathrm{th} = -\frac{C_\mathrm{m}V_\mathrm{th}}{\tau_\mathrm{m}}. \tag{4.34}$$

The shorthand $y := e^{-\frac{t_k}{\tau_\mathrm{m}}}$ was introduced to make it clear that it is a quadratic equation that is solved by

$$y = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}, \tag{4.35}$$



(a) The defining relation of the LambertW function.



(b) The LambertW function $W(z)$ for its real branches and the derivative of the used branch 0 given as in Eq. (4.23). The 0 branch is defined for values larger than $z > -1/e \approx -0.4$. The -1 branch is defined for $0 > z > -1/e$. The LambertW is injective for real z with $z > 0$.

Figure 4.2: Plot of the LambertW function. Because the defining relation is solved by the Lambert W function, the two plots are inverses of each other, as made clear by the colours. To the right of the minimum in (a) is the same branch as in the top half in (b).

giving the relation for the spike time as

$$t_k = \tau_\mathrm{m} \log\left[\frac{2a}{-b \pm \sqrt{b^2 - 4ac}}\right]. \tag{4.36}$$

Again, a differentiable relation for the spike time was obtained, and can be used to calculate the derivatives.

Derivatives

With the same assumptions regarding the indexing of pre- and postsynaptic neurons, the parameters are

$$a_k = -\sum_{i \in C_k} w_{ki} e^{2\frac{t_i}{\tau_\mathrm{m}}} \qquad b_k = \sum_{i \in C_k} w_{ki} e^{\frac{t_i}{\tau_\mathrm{m}}} \qquad c_k = -\frac{C_\mathrm{m} V_\mathrm{th}}{\tau_\mathrm{m}}. \tag{4.37}$$

There are two branches in Eq. (4.36). After a threshold crossing, the leak voltage will lead to a second crossing at some later time. The crossing with the smaller time is the one that should be used in Eq. (4.36).

Due to the positive weights needed for a spike, $b_k > 0$, and $a_k < 0$ is a reasonable assumption. The logarithm in Eq. (4.36) is monotonic. The earlier solution will be the one with the larger denominator. Therefore the $-$ branch is the correct one.

Putting everything together, the time-to-first-spike can be calculated by

$$t_k = \tau_\mathrm{m} \log\left[\frac{2a_k}{-b_k - \sqrt{b_k^2 - 4a_k c_k}}\right]. \tag{4.38}$$

The derivatives of a and b are calculated to be

$$\frac{\partial a_k}{\partial t_i} = -\frac{w_{ki}}{\tau_\mathrm{syn}} e^{\frac{t_i}{\tau_\mathrm{syn}}} \mathbb{1}_{i \in C_k} \qquad \frac{\partial a_k}{\partial w_{ki}} = -e^{\frac{t_i}{\tau_\mathrm{syn}}} \mathbb{1}_{i \in C_k} \tag{4.39}$$

$$\frac{\partial b_k}{\partial t_i} = +\frac{w_{ki}}{\tau_m} e^{\frac{t_i}{\tau_m}} \mathbb{1}_{i \in C_k} \qquad \frac{\partial b_k}{\partial w_{ki}} = +e^{\frac{t_i}{\tau_m}} \mathbb{1}_{i \in C_k}, \tag{4.40}$$

and the differential of the spike time can be written, with the help of $x_k := \sqrt{b_k^2 - 4a_k c_k}$, as

$$\mathrm{d}t_k = \left[\frac{\tau_\mathrm{m}}{a_k} + \frac{2\tau_\mathrm{m} c_k}{(b_k + x_k) x_k}\right] \mathrm{d}a_k + \left[-\frac{\tau_\mathrm{m}}{b_k + x_k}\left(1 + \frac{b_k}{x_k}\right)\right] \mathrm{d}b_k. \tag{4.41}$$

4.3 Conductance-Based Synapses and the Weight Scale Factor

Motivation

For a detailed motivation and history of CoBa synapses for neurons refer to the literature [*Petrovici*, 2015; *Dayan and Abbott*, 2001; *Gerstner and Kistler*, 2002]. Here it is enough to show the different model, highlight the differences and explain the used approximation.

For CoBa synapses, the synaptic current has a direct dependence on the membrane voltage and the reversal potentials E_{rev}:

$$\begin{aligned} I_{\text{syn}}^k(t) = &\sum_{i\in\text{Exc}} w_{ki}(E_{\text{rev,exc}} - V^k(t))\kappa(t-t_i) + \\ &\sum_{i\in\text{Inh}} w_{ki}(E_{\text{rev,inh}} - V^k(t))\kappa(t-t_i). \end{aligned} \tag{4.42}$$

The weights have dimension of a conductance, nS. They modulate the effect of the reversal potentials E_{rev}. The reversal potentials are separated in excitatory and inhibitory reversal potentials, and incoming spikes are associated with one of the two.

Plugging I_{syn} into Eq. (4.1) and grouping all voltage terms together, an equation similar to the CuBa differential equation can be obtained. Dividing by the total conductance

$$g_{\text{tot}}(t) := g_{\text{m}} + \sum_{i\in\text{Exc}} w_{ki} \sum_r \kappa(t-t_{i,r}) + \sum_{i\in\text{Inh}} w_{ki} \sum_r \kappa(t-t_{i,r}) \tag{4.43}$$

results in

$$\begin{aligned} \frac{C_{\text{m}}}{g_{\text{tot}}(t)}\frac{\partial V^k}{\partial t} = -V^k + &\frac{E_{\text{rev,exc}}}{g_{\text{tot}}(t)} \sum_{i\in\text{Exc}} w_{ki} \sum_r \theta(t-t_{i,r}) e^{-\frac{t-t_{i,r}}{\tau_{\text{syn}}}} + \\ &\frac{E_{\text{rev,inh}}}{g_{\text{tot}}(t)} \sum_{i\in\text{Inh}} w_{ki} \sum_r \theta(t-t_{i,r}) e^{-\frac{t-t_{i,r}}{\tau_{\text{syn}}}}. \end{aligned} \tag{4.44}$$

There exists no closed form solution to the membrane voltage of a LIF neuron with CoBa synapses, to my knowledge, much less for the time-to-first-spike. Therefore, an approximation is needed. Approximations are common for training of spiking neural networks, see e.g. [*Neftci et al.*, 2019].

Equation (4.44) looks a lot like the full differential equation for the CuBa model Eq. (4.4), if it were not for the time dependent conductance and the separated inhibitory and excitatory weights.

The assumption is now that the spike times in the CoBa model can be predicted to a reasonable accuracy by using a CuBa model. In Section 6.2 the assumption for the prediction is verified and in Section 7.5 the possibility of learning with a CoBa network is shown.

The approximation is done by scaling the CuBa weights with the *weight scale factor* (WSF) α. The WSF has nothing to do with the weight sum cost in the design in [*Mostafa*, 2017]. Formally, the assumption is that α can be chosen to satisfy

$$\frac{E_{\text{rev,exc}}}{g_{\text{tot}}(t)} \approx \frac{\alpha}{g} \approx \frac{E_{\text{rev,inh}}}{g_{\text{tot}}(t)} \text{ and } \frac{C_{\text{m}}}{g_{\text{tot}}(t)} \approx \tau_{\text{m}} = \frac{C_{\text{m}}}{g}. \tag{4.45}$$

This is equivalent to the transformation in the CuBa weights

$$w \to \tilde{w} = \alpha w. \tag{4.46}$$

The WSF has the unit of a voltage $[\alpha] = \left[\frac{w_{\text{CuBa}}}{w_{\text{CoBa}}}\right] = V$. In experiments, the factor will be measured in a similar scenario as its usage. In this way, a ratio of the conductances can be incorporated.

Calculating the WSF

For all three cases of ρ, can be determined from recorded spikes.

Starting with infinite τ_{m}, $\rho = 0$, the relevant equation is Eq. (2.12):

$$z_k = \frac{\sum_{i \in C_k} w_{ki} z_i}{\sum_{i \in C_k} w_{ki} - V_{\text{th}}}. \tag{4.47}$$

With the transformation Eq. (4.46) the following is obtained

$$\alpha_k = \frac{V z_k}{\sum_{i \in C_k} w_{ki}(z_k - z_i)}. \tag{4.48}$$

In the $\rho = 1$ scenario, performing Eq. (4.46) on Eq. (4.15)

$$0 = a + bx + ce^x \tag{4.49}$$

results in

$$0 = \alpha_k a_k + \alpha_k b_k x_k + c_k e^{x_k} \tag{4.50}$$

$$\alpha_k = -\frac{c_k e^{x_k}}{a_k + b_k x_k}, \tag{4.51}$$

with the definitions for a_k, b_k, and c_k from Eq. (4.25).

For $\rho = 2$, a similar calculation starting from Eq. (4.33)

$$0 = c + by + ay^2, \tag{4.52}$$

results in

$$\alpha_k = -\frac{c_k}{b_k y_k + a_k y_k^2}. \tag{4.53}$$

In practise, the WSF is determined by simulating or emulating a network and registering all spikes. Given these input and output spike times, the WSF can be calculated with one of Eqs. (4.48), (4.51) and (4.53). The mean of the resulting WSF is the basis for the training.

4.4 Improving Optimisation Based on a Superior Energy

The networks are trained with gradient descent, which is based on the energy. Changing the energy therefore likely changes the training, and, potentially, can both improve or worsen the training in terms of convergence or stability. In this section, the energy is investigated in order to find possible improvements to stabilise training.

The energy is composed of different parts. I will distinguish between the energy and the total energy. The latter contains the former and additional terms intended to improve training. In the design in [*Mostafa*, 2017] the energy is a cross entropy given by

$$E[\mathbf{z}, j] = -\log\left[\frac{\exp(-z_j)}{\sum_i \exp(-z_i)}\right]. \tag{4.54}$$

Adding to that the L2 norm of the weights and the weight sum cost term gives the total energy.

The difference between energy and total energy is important. In my plots I will show the energy because the quantification of the classification is encoded in it.

Even for the ideal case, this energy will not decrease monotonically. First, because the update uses a fixed step size given by the learning rate η. Second, the *total* energy is optimised, and there is a trade off between the different parts of the energy.

The role the energy and total energy play has many names. Among those names are energy, objective, and loss. They have in common that they are minimised.

4.4.1 Deducing the Energy from Classifying Entropy

In the present setting, the purpose of the energy is to quantify the difference between a target distribution and a measured distribution. The former is determined by the correct label of the input, the latter is the result of the network simulation. A possibility for comparison is to define a target spike sequence, and use a L2 norm of the spike time differences, see e.g. [*Zenke and Ganguli*, 2018].

Following [*Mostafa*, 2017], I measure the entropic difference of two probability distributions given by the output values of the neurons. The output values of the neurons are taken as an a priori undefined function f applied to the spike times t_k.

Given a vector $\mathbf{t}$ with components t_k and a function f, the kth component of a vector

q after a softmax transformation is

$$q_k = \frac{\exp f(t_k)}{\sum_i \exp f(t_i)}. \tag{4.55}$$

q has the defining properties of a probability distribution, i.e. it is non-negative and sums to 1. The correct probability distribution is determined by a one-hot encoding of the label. A one-hot encoding of a label i is a vector with 0 in all components except at the ith component, where it is 1.

Given an estimated probability distribution Q, I can use the cross entropy to quantify the difference to the correct distribution P. The formula for a calculation is given by

$$H(P,Q) = \langle -\log Q \rangle_P = -\sum_i P_i \log Q_i \tag{4.56}$$

The $\langle \dots \rangle_P$ notation means the expectation value wrt. the probability distribution P.

Bringing the formulas together, the softmaxed cross entropy of network with label layer t_k and correct label j is

$$E[\mathbf{t}, j] = -\log\left(\frac{\exp f(t_j)}{\sum_i \exp f(t_i)}\right) = \log\left(\sum_i \exp(f(t_i) - f(t_j))\right) \tag{4.57}$$

In the previous equation, the function f of the spike time is not defined yet. Mostafa used the softmaxed cross entropy of the negative of the exponential times

$$f(t) = -\exp\left(\frac{t}{\tau_{\text{syn}}}\right) = -z. \tag{4.58}$$

Putting this into Eq. (4.57), one recovers the energy from Eq. (4.54)

$$E[\mathbf{z}, j] = -\log\left[\frac{\exp(-z_j)}{\sum_i \exp(-z_i)}\right]. \tag{4.59}$$

Mostafa calculates exclusively with the exponential times z to have simpler equations. Using the actual times t, the energy is

$$E[\mathbf{t}, j] = -\log\left[\frac{\exp\left(-\exp\left(\frac{t_j}{\tau_{\text{syn}}}\right)\right)}{\sum_i \exp\left(-\exp\left(\frac{t_i}{\tau_{\text{syn}}}\right)\right)}\right]. \tag{4.60}$$

It is important to keep in mind, that an energy is *chosen*, and there is freedom in doing so. As an example, for dimensional reasons a time constant is needed with every occurrence of a spike time. This constant is not fixed, and for $\tau_{\text{syn}} \neq \tau_{\text{m}}$ two natural time scales exist, requiring a choice.

There is no obvious reason to use the second exponentiation, and instead a change

from exponential times to linear times with a constant factor ξ is possible

$$f(t) = -\exp\left(\frac{t}{\tau_{\text{syn}}}\right) \to f(t) = -\xi\frac{t}{\tau_{\text{syn}}}. \tag{4.61}$$

This leaves me with the energy

$$E[\mathbf{t}, j] = -\log\left[\frac{\exp\left(-\xi\frac{t_j}{\tau_{\text{syn}}}\right)}{\sum_i \exp\left(-\xi\frac{t_i}{\tau_{\text{syn}}}\right)}\right]. \tag{4.62}$$

This will not vastly change the extrema of the loss function. exp() is a strictly monotonic function. Optimising one or the other function can lead to similar results. However, whether the training succeeds and how fast the training proceeds depends on the details. Given other terms in the loss function like weight regularisation, choosing one energy over the other can make a difference.

4.4.2 Analysis of the Energy

The analysis of the behaviour of the energy is done in two ways to improve intuition. First I look at the fraction as two separate terms, and later as one term. In Eq. (4.57), I assume a generic $f(t)$ that is strictly monotonically decreasing as both examples above are.

The numerator can be simplified to $-f(t_j)$. This is minimised by decreasing t_j, the spike time of the labelled class because of the monotonicity of f. This is why Mostafa uses the *negative* spike times as output values. Due to the monotonicity of the logarithm, the denominator $\log\left[\sum_i \exp f(t_i)\right]$ is minimised my minimising the sum. As all the summands are strictly positive, each term should be minimal. Therefore, the denominator is minimised by increased spike times for all spikes. Together, the energy is minimised by decreasing the spike time for the correct neuron, and increasing all others, just as intended.

The other way to think about it is in terms of

$$E[f, \mathbf{t}, j] = \log\left[\sum_i \exp(f(t_i) - f(t_j))\right]. \tag{4.63}$$

Following the logic above, each difference $f(t_i) - f(t_j) \forall i \neq j$ is maximised, i.e. the correct spike time is pushed forward, and the other spike times backward.

Consider the following concrete example to better understand the dynamics of this energy. Assuming just two classes, the correct one being t_0 and the wrong one t_1. The energy is

$$E[f, t_0, t_1] = \log\left[1 + \exp\left(f(t_1) - f(t_0)\right)\right] \tag{4.64}$$

$$= \log\left[1 + \exp\left(h(t_0, t_1)\right)\right]. \tag{4.65}$$

The two cases of linear time and exponential time amount to

$$h(t_0, t_1) = -\xi \frac{t_1 - t_0}{\tau} \tag{4.66}$$

and

$$\begin{aligned} h(t_0, t_1) &= -\exp(t_1/\tau) + \exp(t_0/\tau) && (4.67)\\ &= \exp(t_0/\tau)\left(1 - \exp\left(\frac{t_1 - t_0}{\tau}\right)\right) && (4.68)\\ &\approx \exp(t_0/\tau)\left(-\frac{t_1 - t_0}{\tau} + \mathcal{O}\left(\frac{t_1 - t_0}{\tau}^2\right)\right). && (4.69) \end{aligned}$$

I did a Taylor approximation in the last step, which is accurate for spike times near each other relative to τ_{syn}.

Spike times close together is a valid assumption for a randomly initialised network. From this calculation one can see that the two functions have similarities, see Fig. 4.3a. However, the factor depending on the first spike time makes a big difference.

Furthermore, different behaviour for bigger time differences is expected. For this, I look at the energy with exponential times, with j being the correct class.

$$\begin{aligned} E[\mathbf{t}, j] &= \log\left[\sum_i \exp\left[\left(-\exp\frac{t_i}{\tau}\right) - \left(-\exp\frac{t_j}{\tau}\right)\right]\right] && (4.70)\\ &= \log\left[1 + \sum_{i \neq j} \exp\left(-\left(\exp\frac{t_i}{\tau} - \exp\frac{t_j}{\tau}\right)\right)\right] && (4.71) \end{aligned}$$

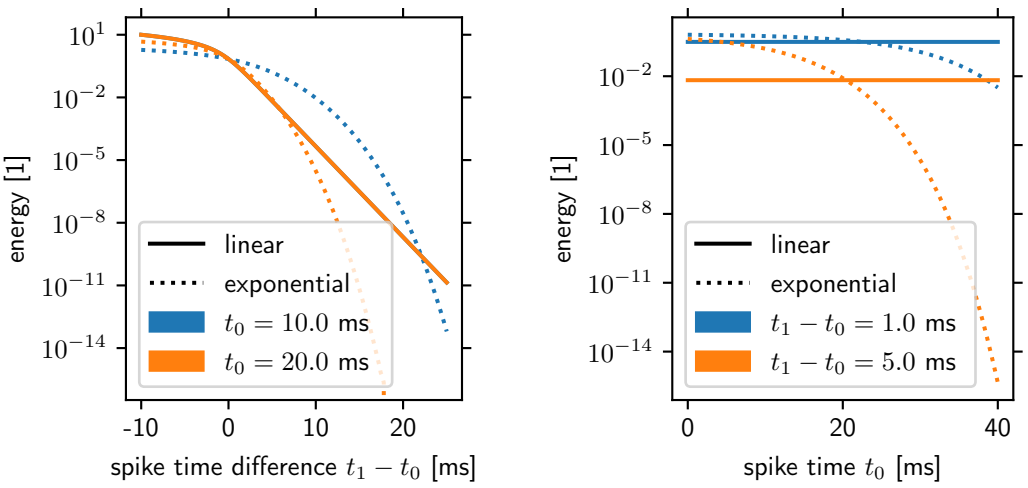
Now I assume an advanced stage in the training where there already is good classification for all inputs. Good classification implies good separation of the patterns, i.e. $t_i \gg t_j \forall i \neq j$. The relevant time constant for $\gg$ is τ_{syn}. From this follows $\exp\frac{t_i}{\tau} \gg \exp\frac{t_j}{\tau}$ and thus

$$\exp\frac{t_i}{\tau} - \exp\frac{t_j}{\tau} \approx \exp\frac{t_i}{\tau}. \tag{4.72}$$

For the energy follows

$$E[\mathbf{t}, j] \approx \log\left[1 + \sum_{i \neq j} \exp\left(-\exp\frac{t_i}{\tau}\right)\right]. \tag{4.73}$$

Minimising this term is done by increasing each t_i. Due to the batching (Section 5.1), i.e. averaging updates for different inputs at each training step, optimising this energy will only reduce all weights. This effect takes place once the network has learned. It has to be cancelled before the whole network becomes quiet. The effect can be also seen in Fig. 4.3b. There, the energy is minimised solely by increasing the initial spike time. The same energy can be achieved by a vastly smaller spike time difference when the initial



(a) The energy is plotted against the spike time difference, for two different t_0. For all setups increasing time difference reduces the energy. The energies with linear f are identical for different t_0. This is not the case for an exponential f.

(b) To emphasis the last point of Fig. (a), the energy is plotted against the spike time t_0 for two spike time differences $t_1 - t_0$. Again, for linear f the energy does not depend on t_0. For exponential f, the energy is reduced by keeping the same difference and only increasing t_0. Based on the intersection with the solid orange line, the energy for a difference of 5 ms with $t_0 \approx 20$ ms is equal to the energy for a difference of 1 ms at $t_0 \approx 40$ ms.

Figure 4.3: Energy of a two class system as in Eq. (4.64), plotted against the spike times t_0 and t_1. Parameters are $\tau_{\text{syn}} = 10$ ms and $\xi = 10$ from Eq. (4.61).

spike is later.

4.5 Ensure Sufficient Spiking

In [*Mostafa*, 2017], the weight sum cost (Eq. (2.14)) ensures a spike for each neuron. With a finite τ_{m}, this is not as simple. Even if the neuron was close to spiking at some point, the membrane is pulled back to the leak voltage. An input after some time may not elicit a spike. Without knowledge of the spike times, it is not possible to predict from the weights whether a neuron will spike. Thus I need to replace the ad hoc weight sum cost mechanism of Mostafa with a different ad hoc mechanism.

The most straight forward way is to check if neurons spike during training. If a quiet neuron is detected one can then increase all the weights leading to that neuron. This is as is done in [*Mostafa*, 2017], as there, too, only the weights of one specific neuron are

increased. However, this would spoil the dynamics. As was just (Section 4.4.2) shown, the spike times tend to slowly shift to later times for training based on the energy of exponential times-to-first-spike. During this shift, one neuron will be the first to not spike at all. Increasing the weights of only that neuron will likely skew the dynamics and reduce the accuracy.

Instead, increasing all the weights in that layer will make the neuron fire again while keeping the dynamics more stable. On hardware, there might be some neurons in hidden layers that might not be used in each pattern, thus it is possible to only apply the message when the number of quiet neurons is larger than some threshold.

The value by which the weights are increased is also important. If it is too small, the mechanism has no effect. If it is too high, the weights will become too large and spoil the dynamics. I am using a small starting value that increases exponentially in case the process had no effect. This is done by recording layerwise whether the mechanism was applied in the previous step and multiplying the current rate by a factor where necessary.

5 Framework

This Chapter introduces the software and training data used in this thesis. The framework enables sharing the same training code for emulations and simulations. All features implemented for simulations are also available for emulations thereby reducing the time needed to implement new features.

I took over the code basis from Oliver J. Breitwieser at the beginning of my master thesis. The code is located in the group repository server and accessible at `https://openproject.bioai.eu/projects/model-tempodrom`. It can also be reached via git with `git@brainscales-r.kip.uni-heidelberg.de:model-tempodrom.git`.

5.1 The Software Framework

Training Algorithm

Prior to training, the network is set up according to the configuration in a parameter file and the network is, if applicable, mapped to hardware. The actual training proceeds in steps.

At the start of each step, random samples of the data are selected to create a mini-batch. The inputs of those samples are concatenated with an idle period as a separator, together forming a long input spike train. This input is fed into the computational substrate which returns the output spikes. The elicited spikes are divided up to the input patterns in the mini-batch they belong to.

For each pattern and its recorded spikes, the update is calculated based on the energy and gradient formula given in the parameter file. The updates are conditionally normalised so that, for each layer, the L1 norm of that layers does not exceed a value given in the parameter file. Conditionally means that values are reduced if they are too big, but small values are left untouched. Afterwards, the updates in the mini-batch are averaged, and for the batch the drive weight mechanism (Section 4.5) is calculated. The resulting total update is applied to the weights.

At regular intervals the accuracy is checked to document the progress.

The gradients are calculated with Eqs. (2.13), (4.28) and (4.41). For an output spike at t_k, the set of causally influencing spikes C_k (Eq. (2.8)) is defined as the set of all input spikes earlier than t_k. Because the spike times are given to the algorithm by the substrate, this set does not have to be calculated, unlike in [*Mostafa*, 2017] where a dedicated algorithm determines C_k.

Simulations and Emulations

The training algorithm is independent of the computational substrate. The substrate returns spike times for given inputs and passes them to the training algorithm where gradients and updates are calculated.

In this work, both simulations (Chapters 6 and 7) and emulations (Chapters 6 and 8) are performed. The simulations use the *NEural Simulation Tool* (NEST) as a simulator [*Diesmann and Gewaltig*, 2002], while the emulations on BSS are controlled with PyHMF [*Jeltsch*, 2014].

PyHMF is an implementation of the PyNN API of version `0.7.5`. The NEST implementation of that PyNN version is `2.2.2` and was two orders of magnitude slower in my experiments (no data shown) compared to a more recent version of NEST. The used version of NEST is `2.14.0`. To enable emulations on hardware with PyHMF as well as fast simulations in NEST this framework uses a wrapper for those two applications. The wrapper makes changing the substrate possible with just one change in the parameter file.

Additionally, a patch set [*NEST Pull 1112*] was used to solve an issue for $\tau_{\mathrm{m}} = \tau_{\mathrm{syn}}$ simulations that was discovered in the thesis [*NEST Issue 1087*].

Auxiliary Tools

Parameter Files The software expects parameter files to be *YAML Ain't Markup Language* (YAML) files. For this thesis, the YAML files were often generated by *YAML with pre-Computed Common Parameters* (YCCP) for parameter sweeps [*YCCP repository*; *Breitwieser*, 2015].

Network Generation Naturally, the network construction is automated depending on the parameter file. In the parameter file the type of data (Section 5.2) is given along with the configuration of the hidden layers in form of a list. Each list item defines the number of neurons in that hidden layer, e.g. an empty list is translated to a shallow network. The input and label layer are determined by the size of the input data and number of classes in the data, respectively.

Parameter Noise To investigate the effect of fixed-pattern noise on hardware, noise was introduced to the neuron parameters in the software simulations. This noise (Section 7.5.3) is set when creating the networks. Noise is only added to those parameters that have a noise level set in the parameter file. The value set for the parameter is its noiseless value multiplied with a random factor sampled from a normal distribution with mean 1 and standard deviation equal to the noise level from the file.

ADC Channel Double Use PyHMF allows recording voltage traces of neurons on hardware (Chapter 3). The analogue voltage trace of the circuits is converted to a digital signal by an ADC.

While examining voltage traces, an unpredictable double-usage of the same channel of an ADC was discovered when recording more than one voltage at a time. This bug produced the same trace for two different neurons. To prevent the problem, [*Changeset 3720*] was added for review. With this changeset, the occupied ADC channels are logged and double-use throws an exception. The problem can be avoided by taking the voltage trace of only one neuron at a time, and repeating the measurement for all neurons. This is done automatically when recording voltages in the framework, including the repetition for averaging (Fig. 6.8).

Randomly Initialised Weights In the network initialisation process, the weights are usually set randomly. For each layer a mean and standard deviation σ are defined in the parameter file, and from those the weights are sampled from a Gaussian truncated at 2σ distance. Truncation of the Gaussian ensures that the values are within a specific range. Values outside the range are resampled until they lie inside.

Noisy Inputs When training a network with *Modified National Institute of Standards and Technology* (MNIST) data, there are 10 000 independent images that can be used to quantify the test error. In contrast, the simple pattern (Section 5.2) data set includes only four different patterns. To get a feeling for the stability against spike time noise, for classification each pattern is shown several times with noise added to the input spike times. This noise is again a truncated Gaussian with the standard deviation given in the parameter file.

The same process is employed to produces a larger number of training images that prepare the network for noisy test inputs.

Fixed-Precision Weights During learning with fixed-precision weights (Fig. 7.20) the weights are saved as floating point values, but prior to writing them to a network they are rounded to a precision determined from the parameter file.

There is no maximal weight fixed, therefore the number of used weights can vary during training. To quantify the number of possibly used weights, the cardinality is defined as the maximum weight divided by the weight precision.

5.2 The Data Framework

In this work, two main data sets are used, simple patterns and MNIST data. For reproducibility, the data generation is described here.

Implementing new data sets in the framework is easy. After defining a function that returns training and test data and setting some parameters, like the input layout used for plotting example patterns in confusion matrices (e.g. Fig. 7.15), the data can be used in any network as well as for automated generation of plots.

The data is presented to the network in form of early or late input spikes. Binarised images can be transformed into spikes by equating black pixels with early spikes and white pixels with late spikes for example. This choice is arbitrary, and in Section 8.1.3

it is verified that training succeeds for inverted patterns with the opposite choice. The effect of the time separation between early and late spike is investigated in Section 7.2.4.

XOR

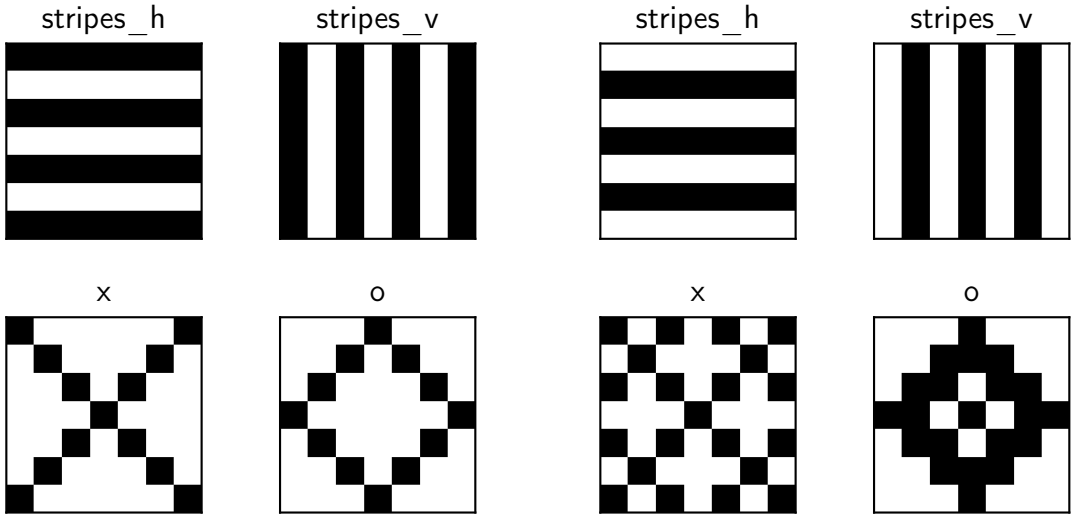
In [*Mostafa*, 2017], the first results are presented for a network (Fig. 2.1) trained on the logical `XOR`. Two boolean variables are given to the network and it should respond with `True` if exactly one input is `True`, and `False` otherwise. With two input sources, but four different patterns, this is a problem that is not linearly separable, and needs a hidden layer to be solved.

In this work no networks were trained to classify `XOR`. The network is presented here for completeness as it was used in [*Mostafa*, 2017]. During the thesis, some work was done with XOR data, but it was decided early on that patterns are a more promising test set for the BSS system.

Simple Patterns

The data set used in most simulations in this thesis is shown in Fig. 5.1b. It consists of images of 7×7 pixels and 4 classes of displayed patterns. The names of the classes, *stripes_h*, *stripes_v*, *x*, and *o*, derive from a previous set shown in Fig. 5.1a.

The difference between the two sets is the number of black pixels in each image. The *balanced* set has the same number of 21 black pixels in each image, while the number of black pixels for the *unbalanced* set varies between 12 and 28. The same number of black pixels implies the same number of early input spikes and prevents classification solely by counting input spikes.



(a) Unbalanced (b) Balanced

Figure 5.1: Plots of the simple patterns set. The unbalanced version is only mentioned in Chapter 8 and shown here to explain the names of the classes as displayed above the corresponding image.

MNIST

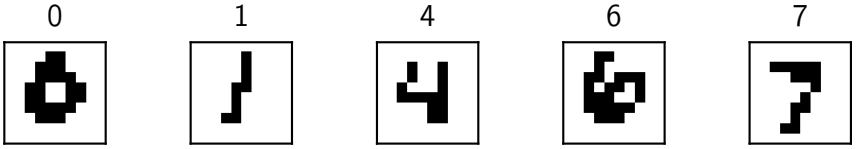


Figure 5.2: Example digits of the downscaled MNIST data set inspired by [*Schmitt et al.*, 2017] used in Section 7.4. The labels corresponding to the displayed image are shown above the images.

The *Modified National Institute of Standards and Technology* (MNIST) data set [*LeCun et al.*, 1998] consists of 28×28 pixel images of handwritten digits. The data is split up in a training set with 60 000 images and a test set with 10 000 images. This data is often used to test pattern recognition algorithms and the best algorithms get below 0.5 % test error rate, see e.g. [*Cireşan et al.*, 2010]. Example images of the full MNIST data set can be seen in Fig. 2.2.

In this work, reduced versions of MNIST are used.

First, in Section 7.4 networks are trained on a version inspired by [*Schmitt et al.*, 2017]. The images are averaged to 10×10 pixels and binarised, and the subset of the digits 0, 1, 4, 6, and 7 are used. Examples are shown in Fig. 5.2.

Then, MNIST was reduced further to fit the mapping currently in place in my code,

i.e. to only have 49 inputs. To this end, the images were averaged and binarised to 9×9 pixels, and the centre 7×7 was selected, discarding a one-pixel border that possesses only little information. In this first try only the subset of the 0, 1, and 4 digits was used for training because due to the subsampling (49 instead of 784 pixels) only a fraction of the information is available. Due to the still little information in the images, only the subset of the 0, 1, and 4 images was used. Example digits can be seen in Fig. 5.3.

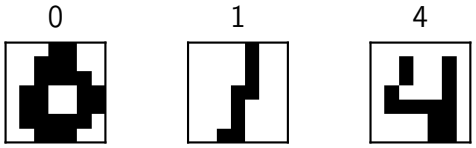


Figure 5.3: Example digits of the most downscaled MNIST data set used in Section 8.2. The labels corresponding to the displayed image are shown above the images.

6 Predictability of Spike Times

In this Chapter, the validity of the formulas for the time-to-first-spike (Eqs. (4.24) and (4.38)) as derived in Chapter 4 is shown by comparison with simulations and emulations. Predicting the correct time-to-first-spike with a differentiable function suggest the possibility of optimisation of an energy of the spike time like the energy (Section 4.4).

First, the precision of the equal-time (Eq. (4.24)) and double-time formula (Eq. (4.38)) for the ideal LIF neurons with CuBa synapses and with $\rho = 1$ and $\rho = 2$ respectively is demonstrated. Afterwards, it is shown that the *weight scale factor* (WSF) approximation (Section 4.3) for CoBa synapses has predictive power for software simulations. In this process, I determine the WSF for the equal-time and double-time formulas. For the hardware, the reproducibility of voltages and spike times is investigated first. In the end, the WSF for the BSS is determined.

I need to choose parameters for the neurons to do simulations. In the derivation, I used that most of the neuron parameters can be scaled away, and the results are independent of those parameters. This independence holds to some degree for the software simulations. On the hardware, however, the precision of the emulation varies for different parameter settings. To be consistent with the parameters throughout the thesis, little change between the parameters is preferred. Thus for all experiments, I choose parameters close to the ones used in the [*online Guidebook*] for the BSS system. The exact neuron parameters in this Chapter can be found in Listing 1.

6.1 Predicting the Time-to-First-Spike for LIF Neurons in Software Simulations

The equal-time and double-time formulas were derived for CuBa synapses in Chapter 4 and the coincidence of the calculation and simulation is shown for different scenarios and both $\rho := \tau_\mathrm{m}/\tau_\mathrm{syn} = 1$ and $\rho = 2$.

In the figures in this Chapter, voltages are shown alongside the spikes for the simulation and the calculation. The voltages provide plausibility to the calculated spike time. However, the voltage is not needed for the calculation and is not calculated during training. The spike times are calculated by the Eqs. (4.24) and (4.38) in Chapter 4.

There is no reset mechanism for the calculated voltage trace. For the calculation of the voltage, the threshold has no special meaning and the dynamics continue if there is a spike.

The calculation and simulation are compared for three different scenarios. Using multiple scenarios helps detecting mistakes, especially when using a factor to fit the

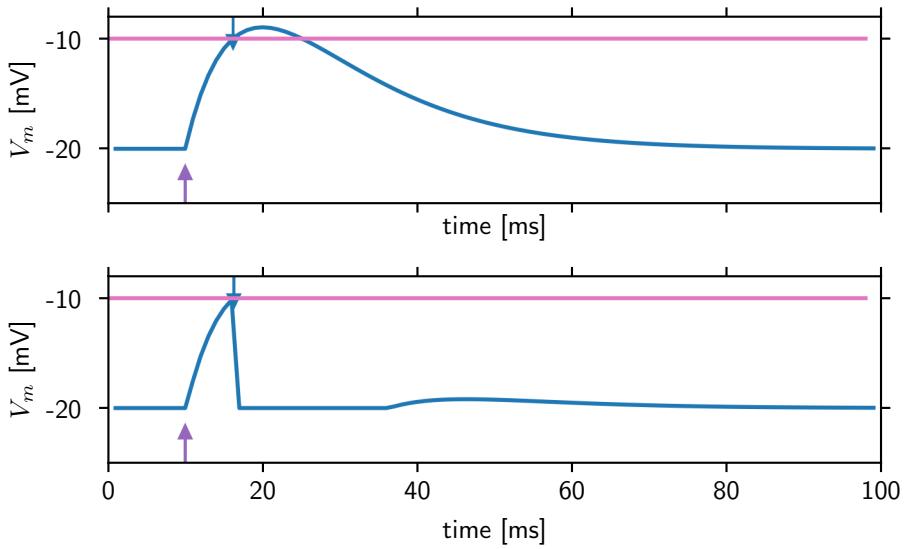


Figure 6.1: Comparison of simulation and calculation for $\rho = 1$ and the equal-time formula

In the top panel traces and output spikes are calculated according to Eqs. (4.12) and (4.24), the bottom panel shows a software simulation (Chapter 5) for the same parameters. In each panel, the membrane voltage is shown over time, with the incoming spikes as arrows at the bottom and the output spikes as arrows on top. The voltage and spikes are colour coded, so same colours belong together, while different colours are different neurons (only 10 different colours are used, neurons after that are shown in black). The threshold voltage V_{th} is shown in pink in this Chapter. There is no reset mechanism after a spike in the calculation, thus the membrane voltage continues after the output spike. On the other hand, in the software simulation the neuron membrane is reset after a spike and clamped at the reset voltage during the refractory time $\tau_{\text{ref}} = 20\,\text{ms}$. After τ_{ref} when the voltage is released the input from the spike has not subsided completely as a slight raise of the voltage is visible.

In this scenario, one pre-synaptic neuron (violet) is strong enough to elicit a spike in the post synaptic neuron (blue). The calculated spike time agrees with the simulated one for several decimals.

results as the WSF in the other Sections. The different patterns are introduced in the following.

The first pattern has one strong enough input spike to elicit an outgoing spike, see Figs. 6.1 and 6.2. With only one spike in the set of causal spikes C_k (Eq. (2.8)), there is no sum in the equal-time formula Eq. (4.24) and double-time formula Eq. (4.38).

The second pattern consists of multiple excitatory spikes. Each of the spikes is too weak to elicit an outgoing spike individually, but the combines input produces an output spike. The results are shown in Fig. 6.3.

The last pattern includes an inhibitory spike. Inhibitory connections are expected in the networks, therefore it is important that the prediction works in this scenario as well.

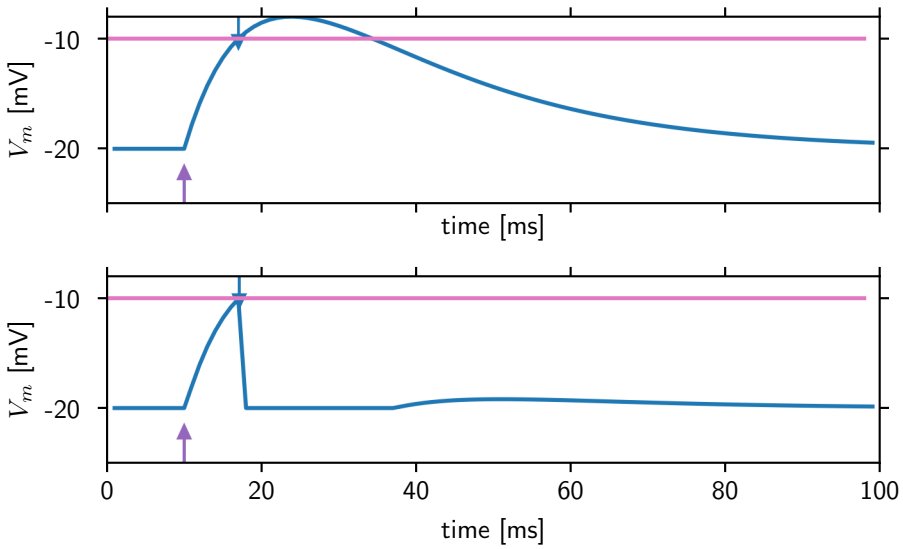
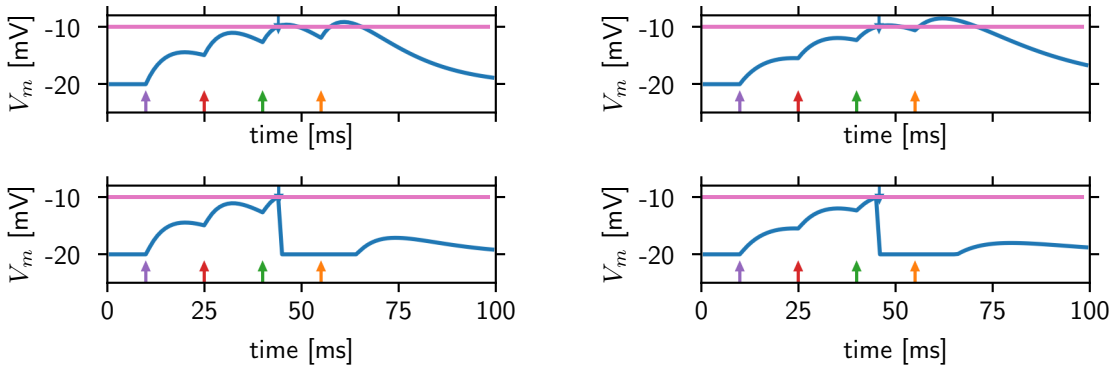


Figure 6.2: Similar to Fig. 6.1 but for $\rho = 2$ and the double-time formula.
The neuron parameters and input spike times are the same as in Fig. 6.1 except for τ_{m}. The weights are adapted to create a similar setup. The difference in τ_{m} can be seen in the slower decay in the upper panel, compared to Fig. 6.1. The calculated spike is identical to the simulated.

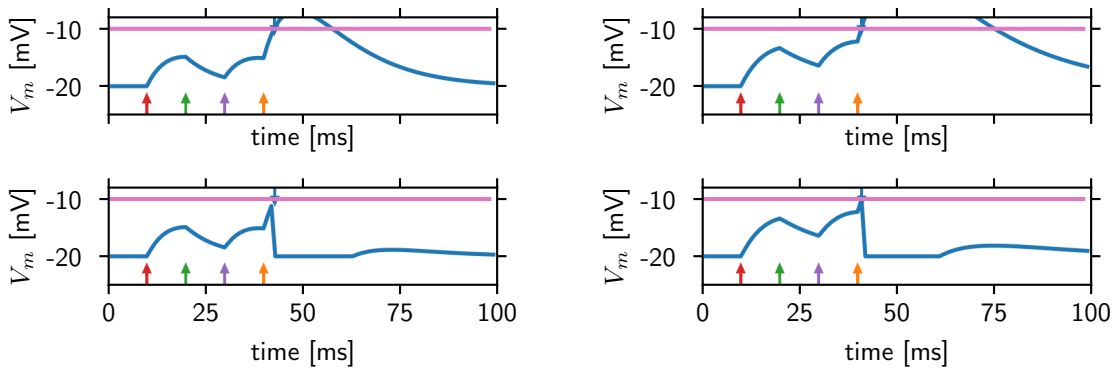


(a) $\rho = 1$, equal-time formula (b) $\rho = 2$, double-time formula

Figure 6.3: Similar to Figs. 6.1 and 6.2, but for different spike inputs.
Comparison of simulation and calculation for different scenarios shows the quality of the calculation. Here four input neurons (violet, red, green and orange) are used, each connection has the came excitatory weight in each plot. The calculated and simulated spike times agree.

Fig. 6.4 shows the results.

The figures in this Chapter show that the prediction works. In fact, the simulated and calculated spike times agree for many decimals. Because the formulas are differentiable and the implementation is stable, optimisation of the energy based on the spike times is



(a) $\rho = 1$, equal-time formula (b) $\rho = 2$, double-time formula

Figure 6.4: Comparison for mixed inhibitory and excitatory weights.
Setup like Fig. 6.3. The input neurons have different weights, including one inhibitory connection (green). Inhibitory connections are expected in the training, thus the formulas have to predict correctly as well. The spike times from calculation and simulation agree.

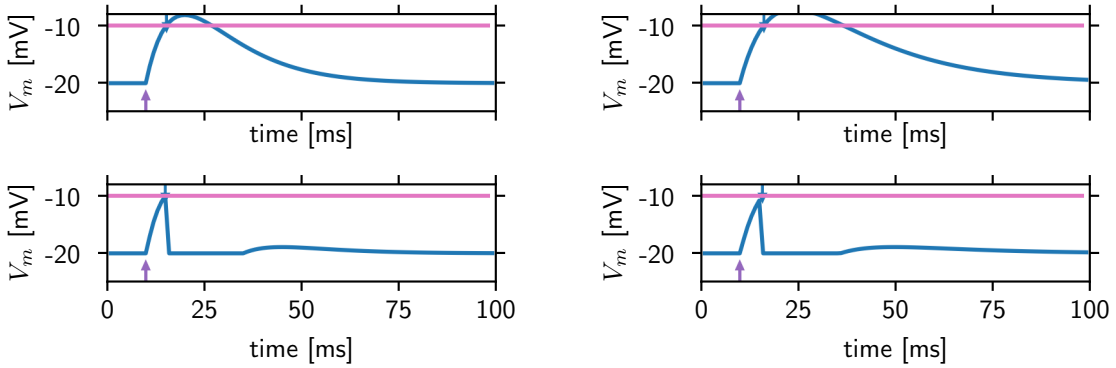
likely (Chapter 7).

6.2 Finding a Weight Scale Factor for Conductance-Based Software Simulations

The synapses used on the hardware are conductance-based synapses. Because learning on the hardware is my goal, my algorithm has to handle a CoBa network. In Chapter 4, I assumed that the *weight scale factor* (WSF) allows to approximate the CoBa spike times with CuBa equations. Here, the approximation is validated and I conclude that the WSF approximation to CoBa neurons is useful to train a network. The actual training is shown in Section 7.5.2.

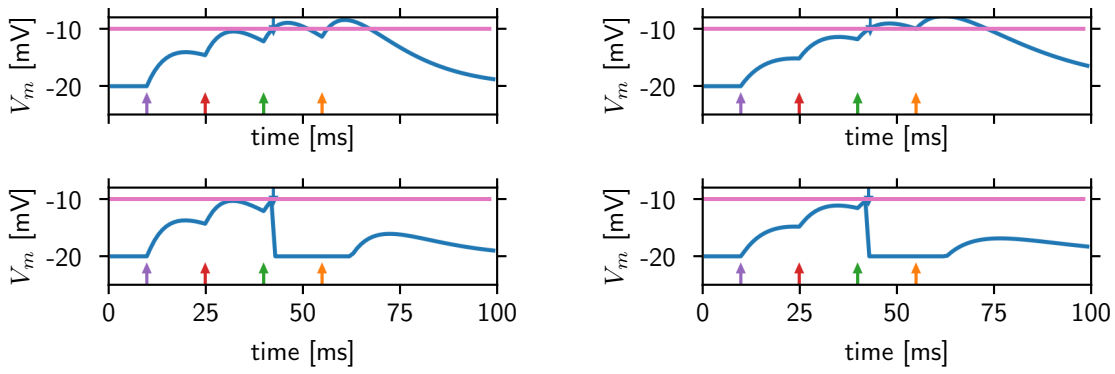
The defining relation of the WSF (Eq. (4.45)) is a ratio of quantities changing for different simulations. While introducing WSF, it was already mentioned (Section 4.3) that the WSF should be calculated from simulations similar to those used in training. This fits the factor better to the situations in which it is used. For given input and output spikes, the WSF can be calculated by one of Eqs. (4.48), (4.51) and (4.53), depending on ρ. I used the scenario (Fig. 6.7a) of mixed input spikes for $\rho = 1$ to determine the WSF that is used for both $\rho = 1$ and $\rho = 2$ in this chapter. The calculated and simulated spikes in this particular scenario align by definition due to the choice of WSF.

With the stated choice of WSF the calculated and simulated spike times in Figs. 6.5 to 6.7 differ by less than 0.4 ms for both ρ. Depending on the scenario, the perfect WSF would be larger or smaller. This suggests that the chosen WSF is in the correct range for many scenarios. Even for the fast spike in Fig. 6.5, the observed variation is less than 3%.



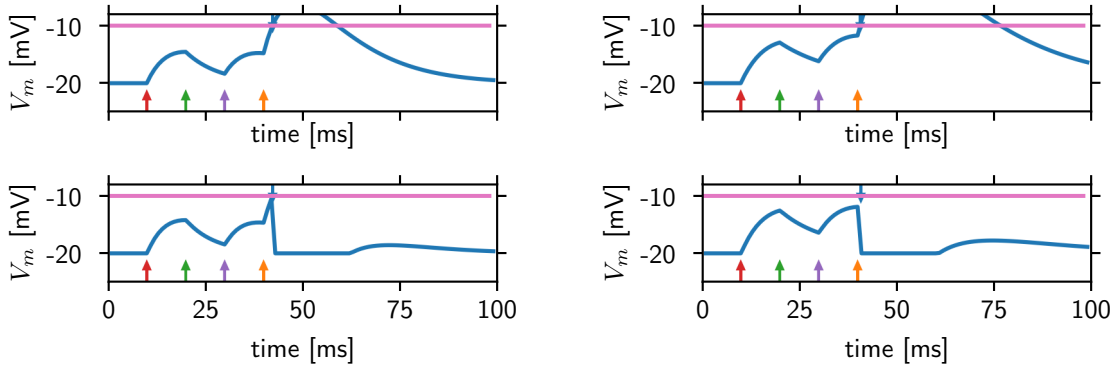
(a) $\rho = 1$ (b) $\rho = 2$

Figure 6.5: Prediction for neuron with CoBa synapses and one input spike.
Setup as in Figs. 6.1 and 6.2, but for neurons with CoBa synapses. The WSF determined here and in the other plots from this Chapter can be found in Table 6.1. With this WSF, the calculated and simulated spike time agree reasonably. The accuracy is discussed in the text.



(a) $\rho = 1$ (b) $\rho = 2$

Figure 6.6: As in Fig. 6.3 but for neurons with CoBa synapses. The same WSF as in Figs. 6.5 and 6.7 is used. The spike times agree reasonably between calculation and simulation as discussed in the text.



(a) $\rho = 1$ (b) $\rho = 2$

Figure 6.7: Setup as in Fig. 6.4 but with CoBa synapses. The inhibitory and excitatory inputs appear in separate terms for CoBa synapses, see Eqs. (4.44) and (4.45). The calculated and simulated spike times agree reasonably as discussed in the text.

6.3 Predictability on Hardware

The hardware was described in its general structure in Chapter 3. It was noted that the analogue nature introduces different kinds of noise, thus an investigation into the variability of spike times is warranted. In this Section I check the reproducibility of voltage traces and spike times.

For small networks, the voltage evolution gives a basic understanding of the dynamics. While the spike times carry a lot of information themselves, the general behaviour is often easier understood with voltages traces. For the voltages to carry information they

have to be reproducible, this is verified first.

It is important that the spike times are consistent as the training is jeopardised by large variations. Thus, the noise of the analogue system (Chapter 3) could produce problems, but here it is shown that the noise in manageable.

Once the reproducibility of voltages and spikes is established, the WSF for BSS can be determined analogous to Section 6.2.

As discussed in Chapter 3, there are different sources of noise, for example the trial-to-trial variability when setting the *floating gate* (FG) values. To distinguish noise due to rewritten FGs from other sources like electronic jitter or delays, the two terms 'job' and 'run' are defined here to have specific meaning. With this I want to make clear the life time of a set of FG values.

Definition 1. *A run is defined as one execution of the* PyNN *command* `pynn.run()`. *During the run, the response of the network to some input spike train defined beforehand is recorded. Because the run is executed at once in one piece, the* FG *values stay the same during it.*

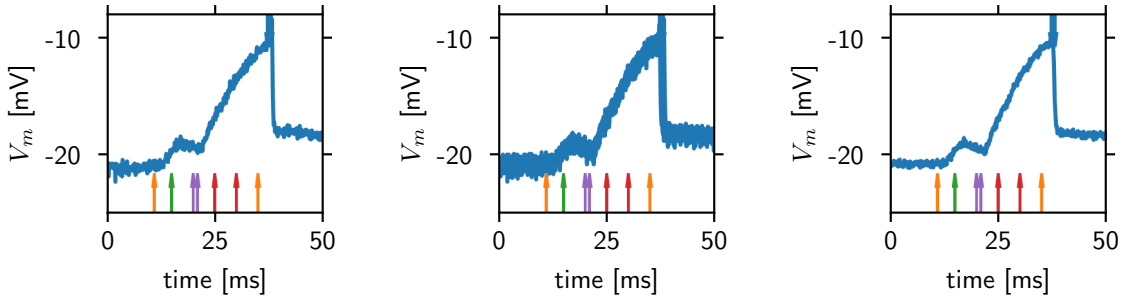
Definition 2. *A job is defined as one executed script on the hardware, e.g. an entire training process. At the start of a job, the* FG*s are set. Therefore different jobs have different* FG *values. Within one job, the* FG *values are not rewritten. A job usually includes multiple runs, which all share the same* FG *values. Between runs, digital settings like the digital weights can be reconfigured without changing the* FG *values.*

6.3.1 Reproducibility of Voltage Traces

Voltage traces will be most helpful at the beginning of the training right after the initialisation, or at the end after the training. At those points they provide a benefit to understanding the dynamics of both the training and the neurons itself.

In Fig. 6.8a a single recorded voltage trace is shown. There is variation visible that is due to noise in the analogue circuit. For comparison, the input is repeated 20 times within one run. All traces are shown overlain in Fig. 6.8b. Due to the noise the line appears thick. Averaging as seen in Fig. 6.8c leads to a clear line. Especially the dynamic after the early inhibitory input is displayed well.

For all plots of voltages the traces are the average over 20 repetitions. However, all recorded spikes are displayed to see outliers and get an estimate of spike time distribution. The distribution is barely visible in Fig. 6.8, this is discussed in more detail in the next Section.



(a) Only one trace is shown. (b) All 20 traces are shown. (c) The average is shown.

Figure 6.8: Voltage traces on the BSS.
Due to electric noise there are variations when recording a voltage trace on hardware. Here, 20 repetitions of the same setup are recorded after each other in one run. The traces are shown in different ways but with the same colour coding as in Fig. 6.1. The noise is reduced by averaging and the dynamics are seen crisper. In all coming plots the averaged traces are shown. There is no averaging for the output spikes, all recorded spikes are shown. Thus they can be spread out, and outliers might be visible.

6.3.2 Trial-to-Trial Variation of Spike Times

The classification as well as the training in my model is based on spike times. For a classification to be reproducible and the training to work the spike times on hardware must not vary too much. The variation of the spike times is examined in this Subsection.

In Fig. 6.8 very little variation in the outgoing spikes was visible. This is one of two edge cases, also shown with statistics for multiple jobs in Fig. 6.9. Variations in the FGs influence, among others, the difference $V_{\text{th}} - E_{\text{L}}$ that has to be crossed before a spike happens. A larger difference increases the spike time for the same weights, because it takes longer to increase the voltage to the larger difference.

Once the FGs are set, noise, delays and random variations cause only small variations of the spike time in this case. The slope of the trace at the crossing is large and relatively stable for variations. Because the slope decreases over time, variations in the input spikes lead to larger variations in the output spike for later spike times. This explains the positive correlation between mean spike time μ_t and standard deviation of the times σ_t seen in Fig. 6.9b.

The other edge case happens when the crossing of the threshold is less steep at the possible spike times. For the following argument I assume that FG variations mainly change the difference $V_{\text{th}} - E_{\text{L}}$ between the threshold and leak voltage. With FGs affecting also other values than the difference $V_{\text{th}} - E_L$ the argument stays the same. The neuron needs to get over this voltage difference before a spike happens. In Fig. 6.10, such a theoretical scenario and three qualitatively different thresholds are shown.

First, for the solid green line there is a large slope at the crossing. The spike happens right after a strong input spike. On the hardware, little variation is expected for the

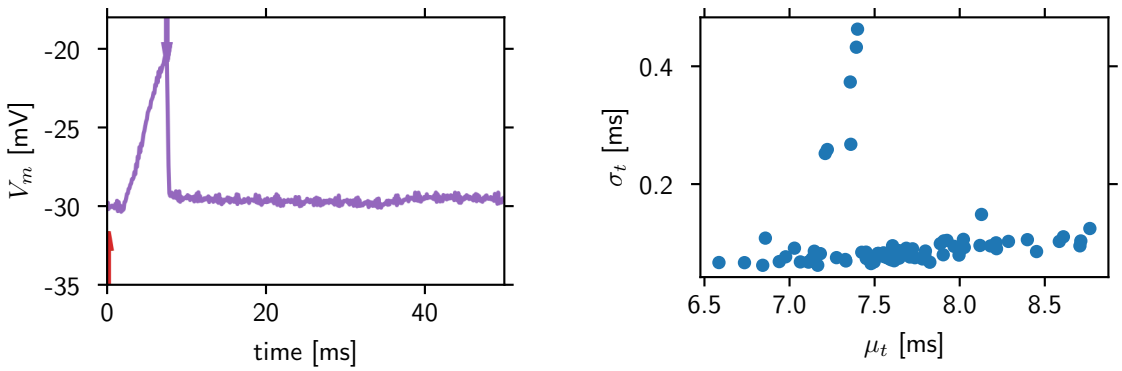
runs in a job with these FG values.

The dotted brown threshold is similar. The crossing happens right after a strong input spike, and the crossing is steep. The variation should be small in this case, too.

The middle case of the dashed grey threshold is different. Relative delays of spikes or noise have a larger influence on the spike time as the slope at the crossing is very flat. Larger variation is expected for the runs in a job with this set of FG values.

Combining these three cases, with increasing average spike time an increase of the variation of the spike times for the runs in a job is followed by a decrease. Fig. 6.11d shows data from the hardware of such a setting.

For the samples shown in Section 6.2, the largest deviation of the CoBa calculation from the simulation is slightly smaller than the *interquartile range* (IQR) of the mean spike times μ_t for the best-case scenario on hardware (Fig. 6.9b).



(a) Voltage of an example job. Spikes come in at the start that are strong enough to immediately elicit a spike. The recorded output spikes in this job are not visibly spread out.

(b) The standard deviation σ of the spike times within one job is shown over the mean μ of those spike times. Some outliers exist, but a small positive correlation between μ and σ is visible. The median of the mean spike times $\mu_t \pm$ IQR is $7.59\,\text{ms}^{+0.31\,\text{ms}}_{-0.24\,\text{ms}}$ showing the positive correlation.

Figure 6.9: Example trace and spike time distribution for a setup that allows direct spiking.
The data comes from 80 jobs with 100 runs each. In 10 jobs the number of recorded spikes was only between 28 and 38, but the average-variance combination of those are not outliers. To repeat my definition, within a job all runs share the same FG values. Between the jobs, the FGs get rewritten.

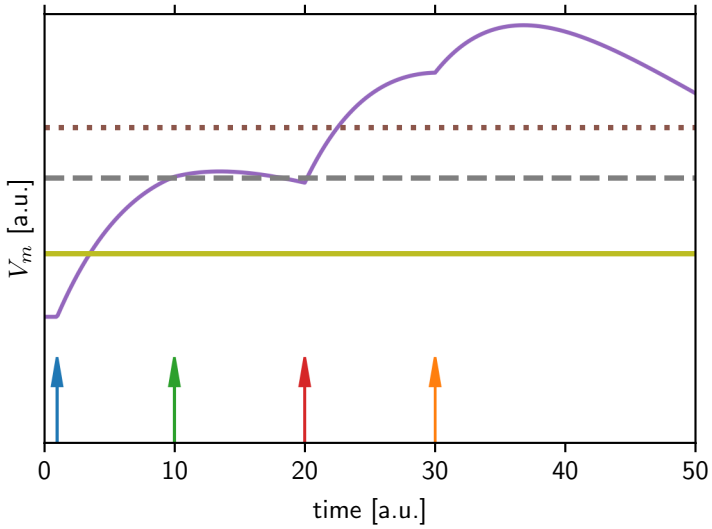
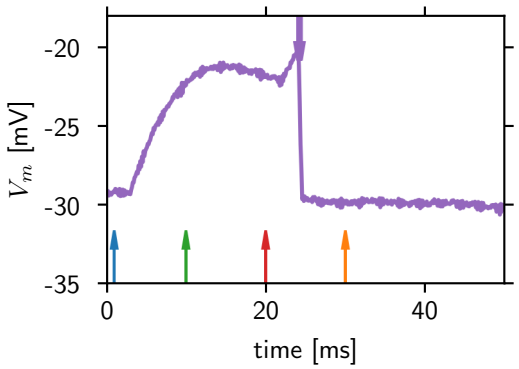
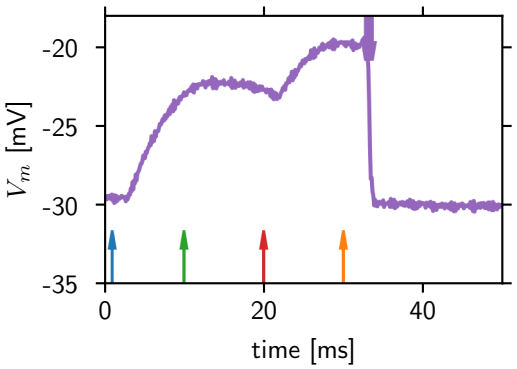


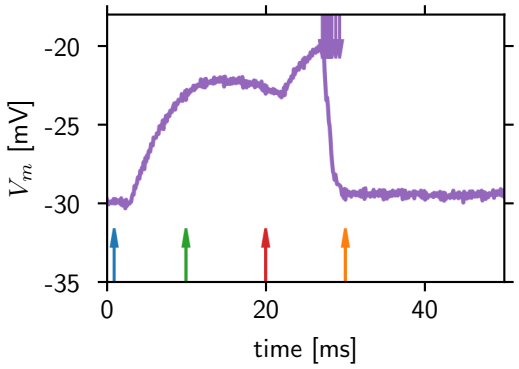
Figure 6.10: Generic voltage trace to understand the dynamics when an output spike is not fired directly.
For this simplified explanation, I assume the FGs predominantly affect the difference between threshold and leak voltage. Due to variance in the FG values, the threshold can be in different parts of the ideal voltage trace (violet). Three possible thresholds are shown. The solid yellow and dotted brown threshold are crossed right after a strong excitatory input spike. The slope at those crossings is large, so a small jitter or input delay leads to small variation in the output spike time. For the dashed grey line however, a small variation will lead to a larger variation in the output spike time. Therefore, the correlation between the mean and standard deviation of the spike time is not expected to be only positive for a setup like this.



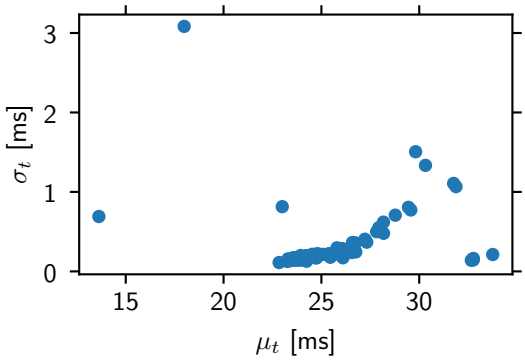
(a) Output spikes happen early, directly after an incoming spike. Little variation of output spike time is visible.



(b) Output spikes happen directly after last input spike. Little variation of the output spikes can be seen.



(c) Output spikes happen between those of a and b. The output spikes are visibly spread out.



(d) The standard deviation σ is shown over the mean μ of the output spike times. While there is a positive correlation at first, σ reduces again as seen in a to c. This is explained in Fig. 6.10. The median of the mean spike times $\mu_t \pm$ IQR is $24.9\,\mathrm{ms}^{+2.1\,\mathrm{ms}}_{-0.8\,\mathrm{ms}}$.

Figure 6.11: Analysis of spike time variation as in Fig. 6.9 but for a setup as in Fig. 6.10. Three example traces are shown to verify the theoretical consideration in Fig. 6.10. The data comes from 71 jobs with each 100 runs.

6.3.3 Finding a Weight Scale Factor for Neurons on Hardware

The procedure here is equivalent to the one in Section 6.2 but for emulations instead of simulations. The WSF is calculated from the recorded output spike via one of Eqs. (4.48), (4.51) and (4.53) depending on ρ.

The parameters are the same as before, see Listing 1. An additional parameter that is set on the hardware is $g_{\max}$ (Chapter 3). This parameter scales the weights, i.e. increasing $g_{\max}$ increases the weights. As the WSF is determined for a fixed set of parameters, $g_{\max}$ has to be chosen beforehand.

The network I want to train on hardware has 49 inputs, see Chapter 5, and approximately 20 of them spike at a time. With 16 available weights, the value of $g_{\max}$ has to be set carefully to not limit the dynamic range of the neurons. $g_{\max} = 400$ was chosen heuristically to allow the neurons to use information from many sources, and not spike based on only one input spike.

In Figs. 6.12 and 6.13 the comparison of calculation and emulation are shown for $\rho = 1$ with the equal-time formula and $\rho = 2$ with the double-time formula. Running the scenarios again with rewritten FGs results in comparable predictions.

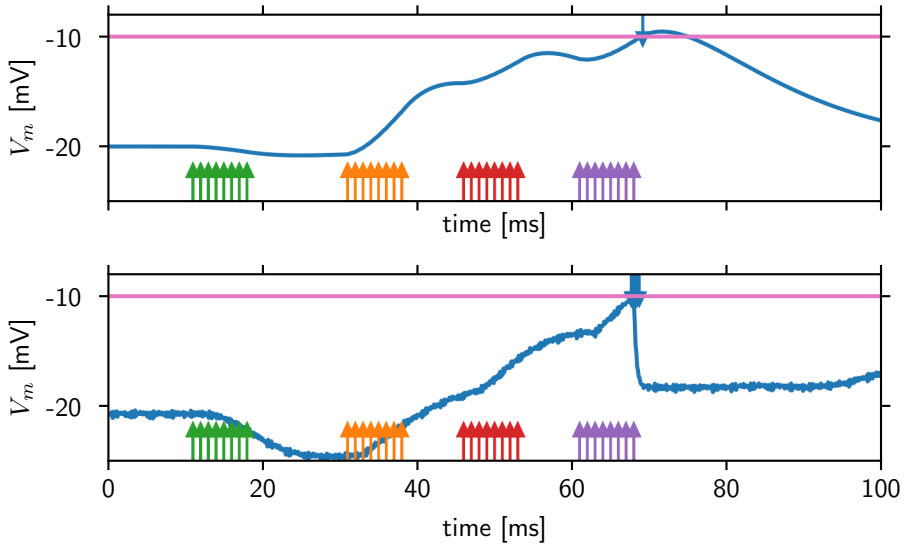


Figure 6.12: Comparison of calculation and emulation on hardware for $\rho = 1$ and $g_{\max} = 400$.
Figure setup as in Fig. 6.1. For the used $g_{\max}$ more spikes are necessary to elicit an output spike. The first inputs (green) come with a negative weight. The average of 20 runs is shown, as discussed in Fig. 6.8. Rewriting the FGs leads to results where the spike times agree similarly.

It is of course possible to redo the simulations for other parameters, and exemplary I chose a second value for $g_{\max} = 1023$, the maximal setting. This can be a suitable choice for a smaller network that uses less spikes, for example. Figs. 6.14 and 6.15 show the calculation and prediction for this setting.

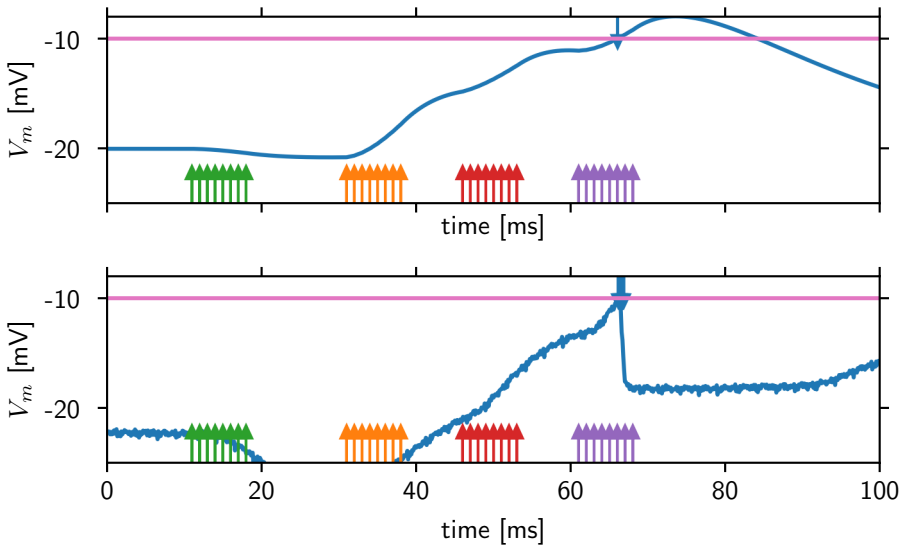


Figure 6.13: Comparison as in Fig. 6.12 but for $\rho = 2$.
The longer τ_{m} can be seen from the slower decay in the calculated voltage trace.

Table 6.1 shows the WSFs that were found in this Chapter. More than those WSFs were used during the thesis. Thus, in experiments in Chapters 7 and 8 other WSFs are used, see Appendices A.2 and A.3 for the exact parameters in each case.

For simulations (Section 6.2), the WSF could be chosen equal while on hardware, the WSFs were chosen different. The WSF was introduced as a heuristic approximation, and as long as the simulations work, this is no cause for concern.

I have shown the accuracy of the equal-time and double-time formula in this Chapter. Furthermore, it was shown that introducing the WSF is a suitable method to approximate the time-to-first-spike of CoBa neurons. Choosing the WSF in a correct range allows for prediction of time-to-first-spike on hardware.

	Simulated CoBa	HW CoBa	
$\rho = 1$	71.1	$g_{\max} = 400$	0.00281
		$g_{\max} = 1023$	0.0125
$\rho = 2$	71.1	$g_{\max} = 400$	0.00197
		$g_{\max} = 1023$	0.00836

Table 6.1: Table of WSF determined in this Chapter.

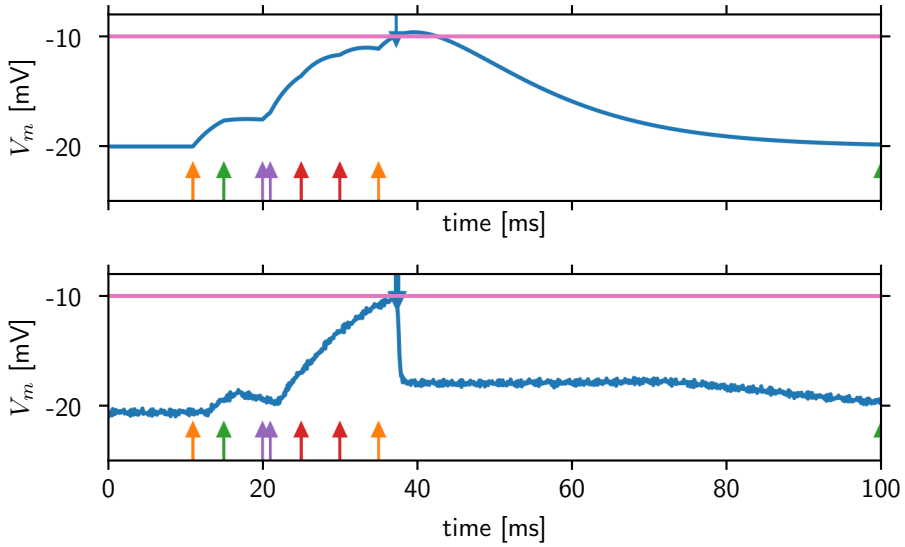


Figure 6.14: Determining the WSF for $\rho = 1$ and $g_{\max} = 1023$, the maximal setting. Less spikes are needed to elicit the spike compared to Fig. 6.12. The green input is inhibitory again.

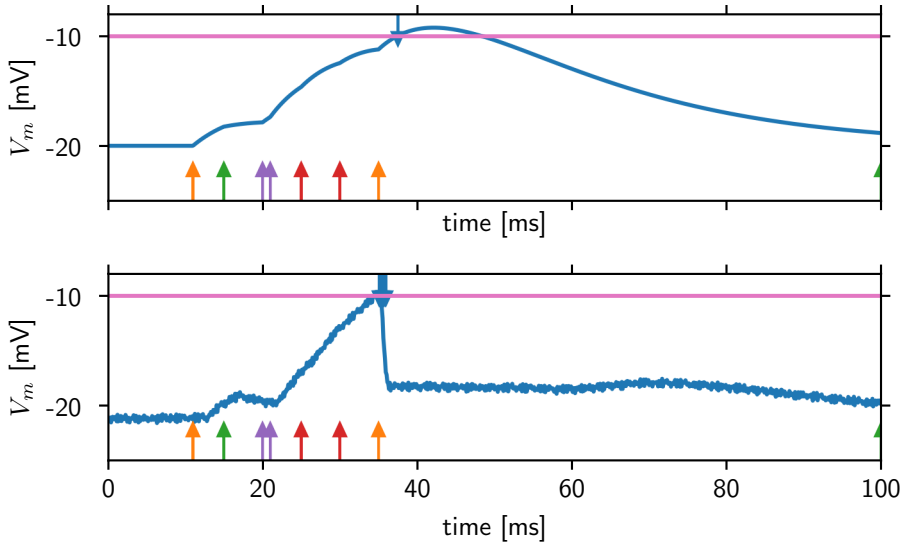


Figure 6.15: As Fig. 6.14 but for $\rho = 2$. The longer τ_{m} leads to a slower decay of the voltage.

7 Simulations of Spiking Networks

In this Chapter, I present results of the training of spiking networks of LIF neurons with the equal-time (Eq. (4.24)) and double-time formula (Eq. (4.38)) derived in Section 4.2. The predictive power of the formulas was shown in the previous chapter (Section 6.1), the stated goal is to train network of LIF neurons. This training is investigated here, with a focus on initialisation stability of the training.

The networks are trained to classify patterns (Section 5.2). These patterns consist of black and white pixels. The black and white pixels are translated to early and late input spikes respectively. Given those inputs, the networks learn to clasify the inputs.

Initialisation stability is tested by varying the random initialisation of weights. Weights are randomly sampled from a truncated Gaussian with mean and standard deviation (Chapter 5).

The *random number generator* (RNG) is initialised by a seed. Changing the seed leads to different random weights. By varying the seed I can test dependence on the initialisation. Unless otherwise specified, results in this Chapter are presented as sweeps over multiple seeds. The evolution of energy and accuracy is then displayed as 0, 25, 50, 75 and 100 percentiles of the results of the individual trainings. For each sweep, the parameter file for one seed is shown in Appendix A.2.

First, I explain the learning of the network step by step. This is helpful to understand all training processes throughout this thesis.

Furthermore, an understanding of the setup is gained by changing individual parameters and analysing the results. The experiments are done for both equal-time and double-time formulas.

The two formulas, as well as the formula from [*Mostafa*, 2017] in the present framework, are then compared while varying the membrane time constant τ_{m}. A short excursion shows the general capability of the framework to classify the MNIST dataset (Section 5.2).

The balanced pattern data set used in the other section is discussed in Section 5.2 as well. In Section 7.1, it is shown that a shallow network is able to classify this data set. Thus, it is strictly speaking not necessary to classify this data set in a deep layer. This is still done, as evidence is obtained for the capability of the framework to train deep networks of LIF neurons.

In the end I advance from the ideal CuBa setup with floating point precision weights towards networks more similar to the hardware.

7.1 Learning in a Spiking Network

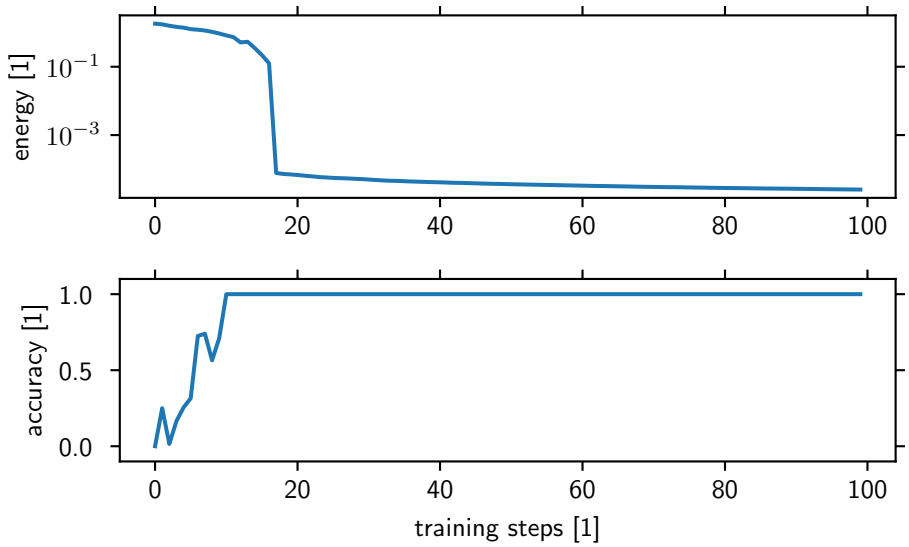
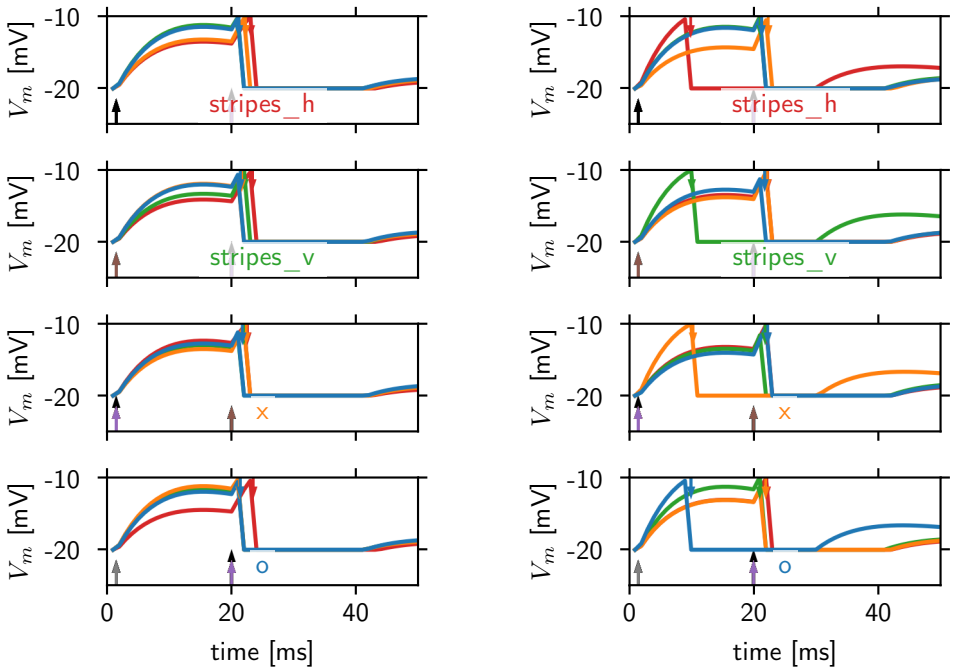


Figure 7.1: Evolution of energy of the linear spike times (top panel, logarithmic scale) and accuracy (lower panel) during training.
A shallow network with $\rho = 2$ is trained via the double-time formula on balanced patterns (Chapter 5). Both accuracy and energy are averaged over noised samples. 100% accuracy is achieved after around 15 steps while the decrease of the energy slows down after 18 steps. The training converges in a stable manner, i.e. neither a decrease in accuracy nor an increase in energy is seen. Both fast advance to a high accuracy, low energy state and stability of learning are desired. The non-monotonic changes in both the energy and accuracy stem from of the learning rate η being finite. The membrane voltages during the training are plotted in Figs. 7.2a, 7.2b and 7.3. For the parameters see Listing 3.

The training process is explained step by step to show how the network dynamics change throughout the learning. The step by step explanation is done for an example network with $\rho := \tau_{\mathrm{m}}/\tau_{\mathrm{syn}} = 2$ (Section 4.2) that is trained via the double-time formula. For simplicity the network is shallow, i.e. there is no hidden layer, to make the training as transparent as possible.

The training process is shown in Fig. 7.1. The training succeeds, as can be seen from the high accuracy and low energy at the end. The network learns in a few steps and converges in a stable manner. Convergence stability means that the accuracy stays high and the energy low once classification is correct.

The large decrease in the energy happens only once the accuracy is at 100%. As long as one class is misclassified or has a low separation, the energy for that class is on the order of 1. Even when the other classes have much smaller energy, the averaging procedure does not change the order of magnitude due to the exponential definition of the energy (Section 4.4.2). Once all classes are suitably separated, the energy reflects the quality of the classification.



(a) Before training
The weights are initialised randomly, see Section 5.1. The random weights lead to only slightly different voltage traces for the different neurons. No output spikes happen before the second input spikes.

(b) After training
By my definition that the first spiking neuron determines the class of the input, all inputs are classified correctly. The separation of the spikes is clear, more than one synaptic time constant $\tau_{\text{syn}} = 10\,\text{ms}$. The classifying spike is elicited approximately one τ_{syn} after the first input spikes, well before the second input spikes.

Figure 7.2: Voltage traces for different inputs in a shallow network, see Fig. 7.1.
Each panel has incoming spikes as arrows from below, outgoing spikes as arrows from above added to the voltage traces in a colour coded fashion, cf. Fig. 6.1. Each row shows the dynamics of the network for an input of the class that is named in the panel (*stripes_h, stripes_v, x, o*). The colour of the class corresponds to the neuron coding for that class. In later plots for larger networks, a hidden layer will be shown as another column.
Listing 3 The transition of the voltage traces from (a) to (b) as seen in the training Fig. 7.1 can be followed in Fig. 7.3.

Instead of the mean energy, using the median energy of the noisy samples could display the process better. However, the median energy can be very low even if one class is completely misclassified. Thus, the mean energy over different samples is used

to display the training evolution.

The voltage trace at the end of training is seen in Fig. 7.2b. It shows correct and fast classification. The separation of the input spikes is approximately τ_{syn} and each neuron spikes exactly once.

At initialisation there is no correct classification (Fig. 7.2a). The spikes happen late, in the wrong order and at similar times. The voltage traces are similar to each other as well. The process from the initialisation to the correct classification is displayed step by step in Fig. 7.3.

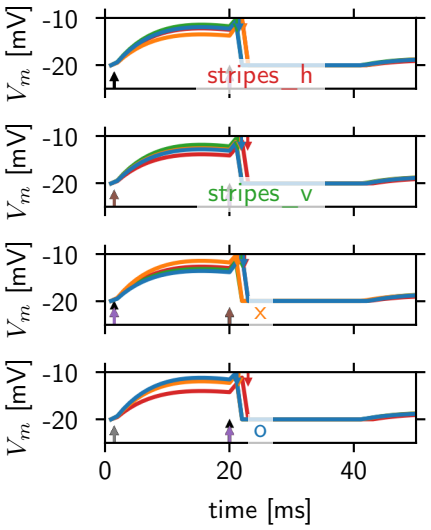
During the learning, for each input class the membrane of the correct neuron is lifted up, and at some point the time of spike jumps forward. Compare those jumps with the energy evolution. The large decrease in the energy happens when the last spike has jumped and has a good separation. Once that has happened, the neuron dynamics are very similar to the final traces in Fig. 7.2b. This is also seen in the evolution of the weights.

The weight evolution is displayed in Fig. 7.4 for individual weights. The weights reach an equilibrium state and change little at the end of training. At the beginning there is much change in the weights.

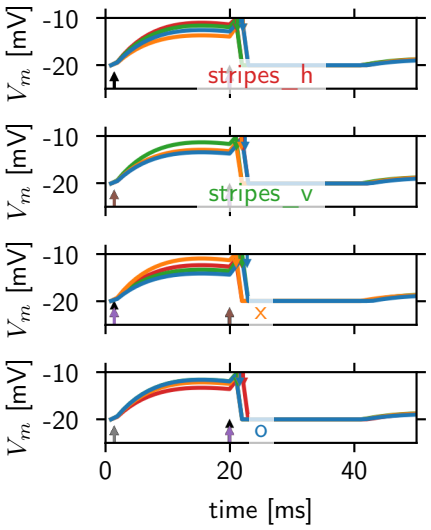
Both at the beginning and in the end there are many more excitatory than inhibitory weights. There also are weights changing from inhibitory to excitatory state, and vice versa.

The weight distribution does not have a clear trend This is the criterion I used for initialising the networks. I checked whether the weights changed the order of magnitude during training. If a clear trend was visible, I adapted the mean or standard deviation. In general, the framework is stable towards initialisation and training succeeds without fine tuning the initialisation.

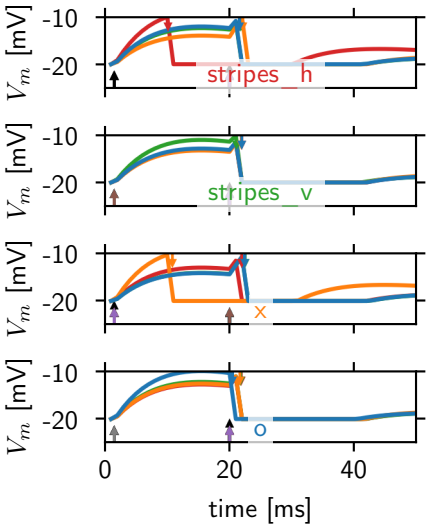
In this Section, the learning process in a spiking network was explained.



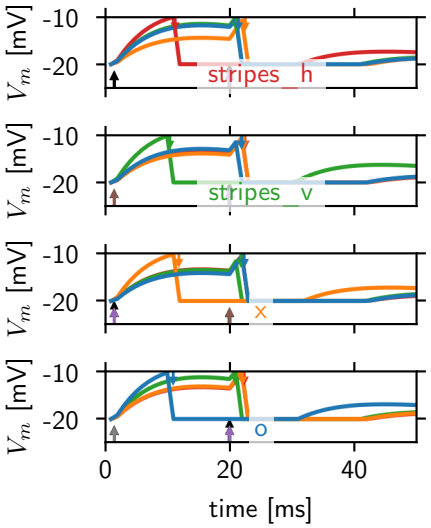
(a) Step 8: while the spike times are similar to those before training, the voltage traces have changed. Especially for the x and o patterns, the correct neuron has the highest voltage already.



(b) Step 10: the spike times are still similar, but now for all cases the correct neuron has the highest potential between the two input spike times. This shows that the correct weights are increased.



(c) Step 15: two neurons spike early with a good separation. The other two neurons are close to spiking earlier.



(d) Step 18: all four inputs are classified correctly, and the output spikes are early and with good separation. These traces are close to those of the final result in Fig. 7.2b.

Figure 7.3: Changes of the voltage traces during the training at different steps during the training, from Fig. 7.2a to Fig. 7.2b. Listing 3

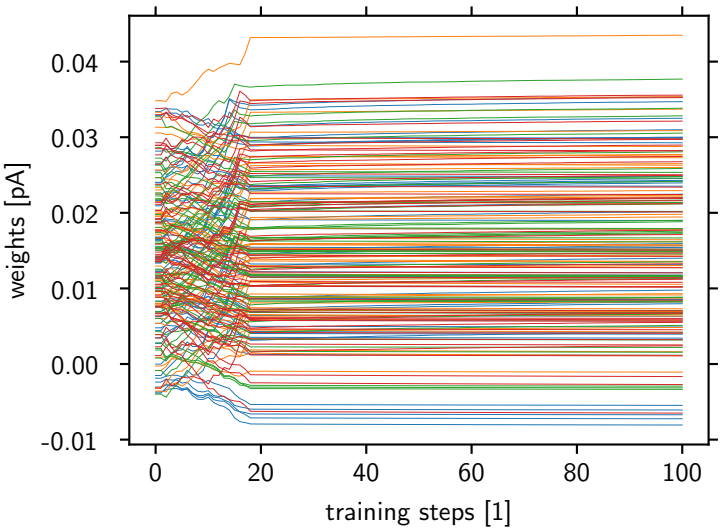
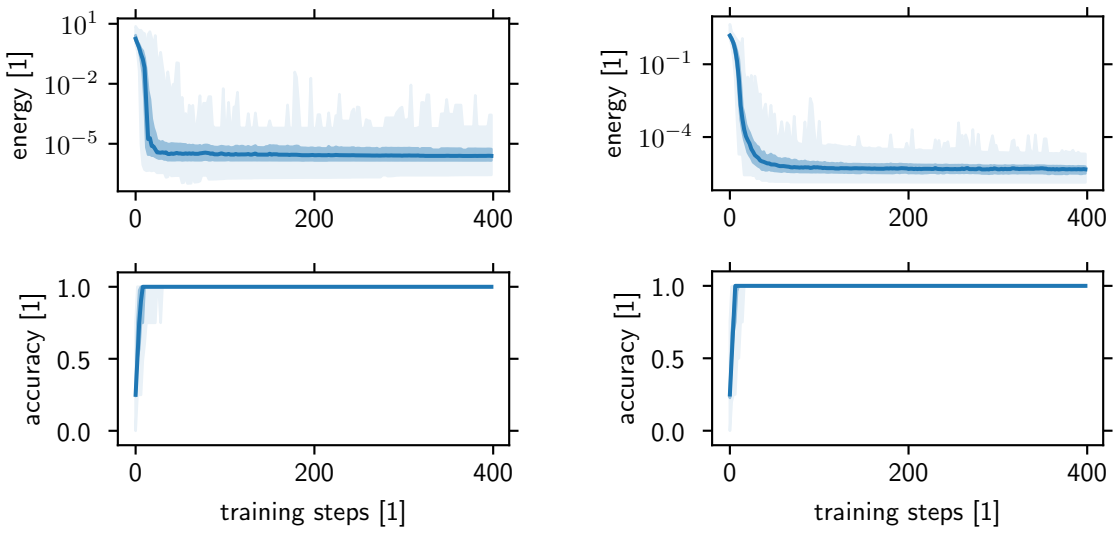


Figure 7.4: Weight evolution during the training, Fig. 7.1.
Individual weights are plotted, starting from the random initial state. The weights approach equilibrium values. There are inhibitory and excitatory weights. Because the network is shallow there is only one panel for the weights from input to output. The lines are colour coded for the neuron their synapses lead to, compare Figs. 7.2a, 7.2b and 7.3.

7.2 Variations in the Learning Setting

It is shown that the training success is independent of the initialisation. I show the benefit of training networks based on the energy of linear spike times as opposed to the energy of exponential spike times as discussed in Section 4.4. The effects of variations in the coding of the input spikes are investigated as well.

7.2.1 Dependence on Initialisation



(a) $\rho = 1$, equal-time formula

(b) $\rho = 2$, double-time formula

Figure 7.5: A network with 10 hidden units was trained for 100 different initialisations. The networks were trained on the pattern data (Section 5.2). The random weights are determined by a seed given in the parameter file. The 0, 25, 50, 75, and 100 percentiles are shown for both the energy and accuracy for both cases of ρ. One example of the voltage traces for $\rho = 2$ can be seen in Fig. 7.6, and the corresponding parameter file is Listing 7.

A dependence of the training success on the initialisation can be ruled out by varying said initialisation. A general training algorithm should succeed independent from initial weights.

Varying the seed of the RNG results in different random weights. A sweep of 100 seeds is shown in Fig. 7.5. The trained network is a deep network with 49 inputs, 10 hidden and 4 output neurons (49-10-4). It is trained to classify the patterns introduced in Section 5.2. Both the equal-time and double-time algorithm learn fast and stable for different initialisations.

An example voltage traces for one seed of the double-time sweep can be seen in Fig. 7.6. The classification is correct and exhibits good separation. This is the first example of classification with a deep network in this thesis.

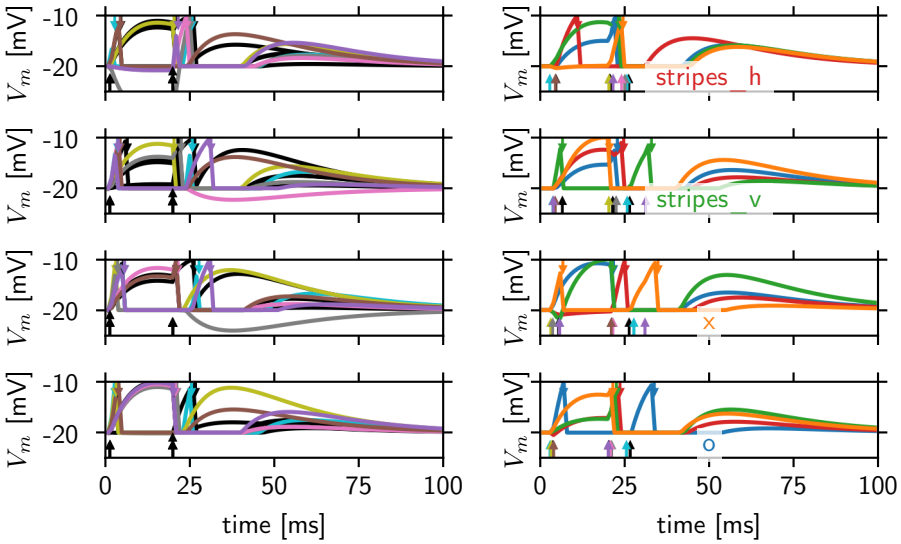


Figure 7.6: The membrane dynamics for an example of Fig. 7.5.
The parameters can be found in Listing 7, but particularly $\rho = 2$. The plot setup is like Fig. 7.2 but for a two layer network. In the left column, the dynamics of the hidden layer is shown. In the right column, the label layer, the correct neurons spike first as can be seen from the colour scheme.

7.2.2 Comparison of Driving Weight Mechanisms

Apart from the spike time formulas I made other changes to the algorithm, especially the driving weights mechanism (Section 4.5) and the energy (Section 4.4).

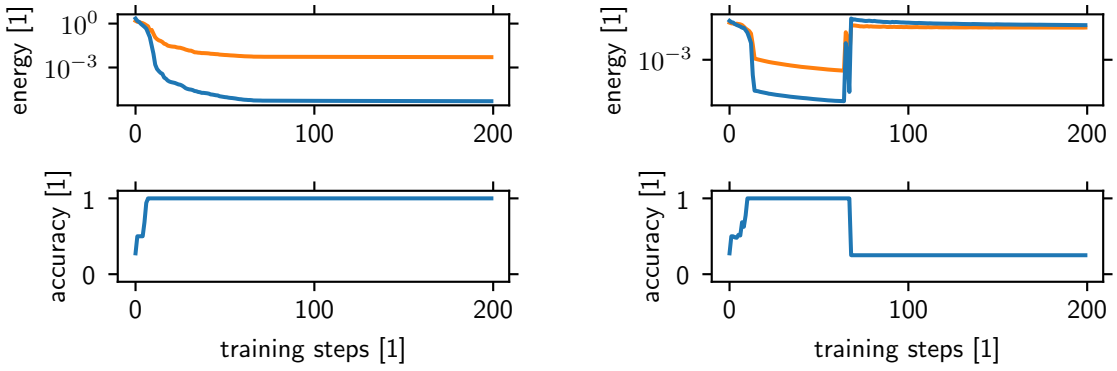
In Section 4.5 it was argued why the weight sum cost from [*Mostafa*, 2017] can not be used for finite τ_{m} and has to be replaced.

The weight sum cost increases the weights of one neuron when its weights become too small for a spike. The direct adaption is to increase the weights of neurons that do not spike. I call this the *old drive weight mechanism*.

In the experiments I noticed that this mechanism can be improved. Increasing all weights in one layer when a percentage of neurons does not spike shows enhanced stability, both convergence and initialisation stability. This mechanism is used throughout the thesis to drive weights.

With the old mechanism, in many networks the accuracy collapsed after it had already been trained. That is due to the other terms in the total energy. As explained in Section 4.4, only the energy term depending on the spike times (Eq. (4.62)) optimises the accuracy. Other mechanisms like the drive weights mechanism or the weight sum cost can reduce the accuracy with lasting effects.

The behaviour of collapsing accuracies produced a large dependence on initials and hyperparameters. This in turn was a complication for doing meaningful sweeps of parameters. The instability is solved in my framework and anecdotal evidence for the problem is shown in Fig. 7.7.



(a) Starting from the same initials the new mechanism leads to 100% accuracy and low energy.

(b) The combination of the gradient function and energy from [*Mostafa*, 2017] and the old drive weight mechanism does not learn in a stable way. Both right before the sharp rise and before the drop of the accuracy weights are increased with the mechanism.

Figure 7.7: Direct comparison of the two drive weight methods, linear and exponential energy and different gradient functions for an otherwise identical setup. Both the energy for linear (blue) and exponential (orange) spike times is given. The setup of the left plot is given by Listing 4, and the difference between the two is Listing 5.

7.2.3 Energy of Exponential and Linear Time

The energy of exponential spike times (Eq. (4.60)) shows problematic behaviour (Section 4.4.2). Optimising that energy results in shifting spikes to later times. For finite τ_m spikes can only happen close to input spikes (Section 4.2.1). Thus shifting spikes to later times is not always possible. The energy of linear spike times (Eq. (4.62)) does not have this problem of shifting spikes back in time.

This convergence instability of the exponential energy can be seen in Fig. 7.8. While the median of the seeds is trained well, overall there is a lack of stability compared to Fig. 7.5. The learning is faster for linear energy as well (data not shown). The situation is identical for $\rho = 1$ and the equal-time formula (not shown).

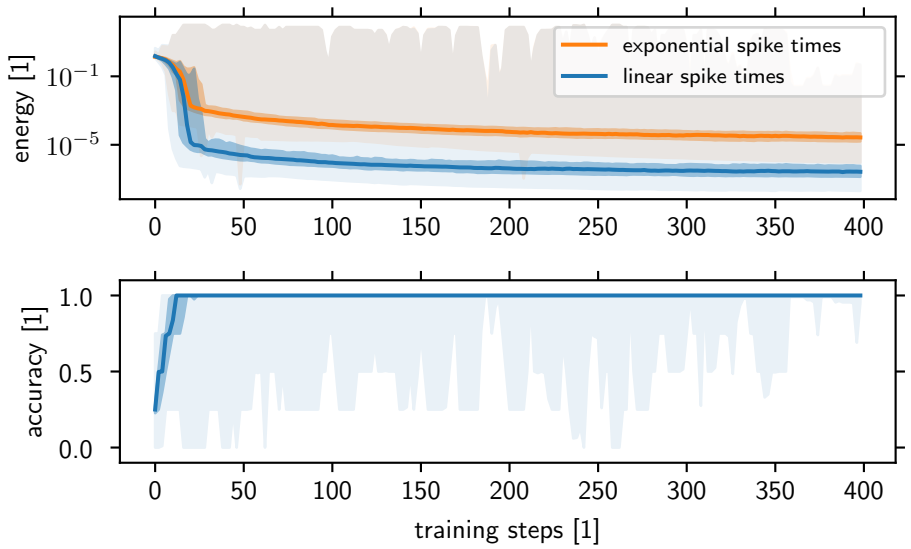


Figure 7.8: Direct comparison with Fig. 7.5b for a training with the energy of exponential spike times.
Apart from the energy, the setup is identical and again training progress for 100 different seeds is shown. The majority of the 100 seeds learn, but less stable and reliant. Both the energy for linear (blue) and exponential (orange) spike times are shown. An example parameter is Listing 8.

7.2.4 Time Separation of the Input

In this Section I discuss the effects of changing the separation between the early and late spikes.

There are two extreme cases. In the limit of $\frac{t_{\text{late}}-t_{\text{early}}}{\tau_{\text{syn}}} \to 0$ there is no distinction between late and early spikes. Thus no classification is possible. For $\frac{t_{\text{late}}-t_{\text{early}}}{\tau_{\text{syn}}} \to \infty$, i.e. no late spikes, only information from the early spikes is available for the classification. This does not prevent classification, as it often takes place before the second input spike (Figs. 7.2b and 7.6).

Experiments show that the learning is more stable when there is a second input spike shortly after the first (Figs. 7.9 and 7.10). The pattern networks are trained and tested with noisy samples (Chapter 5), with a truncated spread of 1 ms for each spike. Therefore, if the non-noisy spikes have a separation of 6.5 ms, the separation of the actual spikes is between 4.5 ms and 8.5 ms.

In the figures only the accuracy is displayed because the energy has no additional relevance as I explain now. With finite τ_{m}, a spike can happen only close after an input spike (Section 4.4.2). For large input separation, the separation of the output spikes is thus either small when the output spikes happen after the same input, or the separation is large if different inputs elicit the output spikes, compare Fig. 7.12. Due to its exponential definition (Eq. (4.62)) the energy is thus either on the order of 1 or several orders of magnitude smaller. The range in between is not accessible.

Example voltages for $\rho = 2$ and the double-time formula for small and large separation are shown in Figs. 7.11 and 7.12. The difference in the input separation is visible, also compare to the usual separation in Fig. 7.6. The classifications are correct.

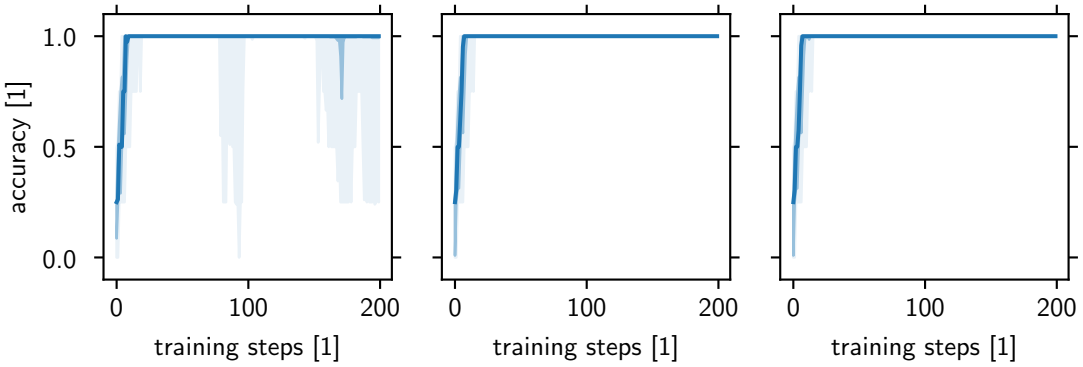


Figure 7.9: Comparison of different separation of input spikes.
The training process for different separations is shown for 10 seeds and the double-time algorithm. From left to right, the separation in terms of τ_{syn} is approximately 0.65, 1.5 and 9.0. The red vertical lines are the times the drive weights algorithm was active. The network learns most reliably for the separation slightly larger than τ_{syn}. For very small separations the learning becomes unstable. The smallest separation is chosen as the first one where instabilities become visible, here for 0.65. An example parameter file can be found in Listing 9.

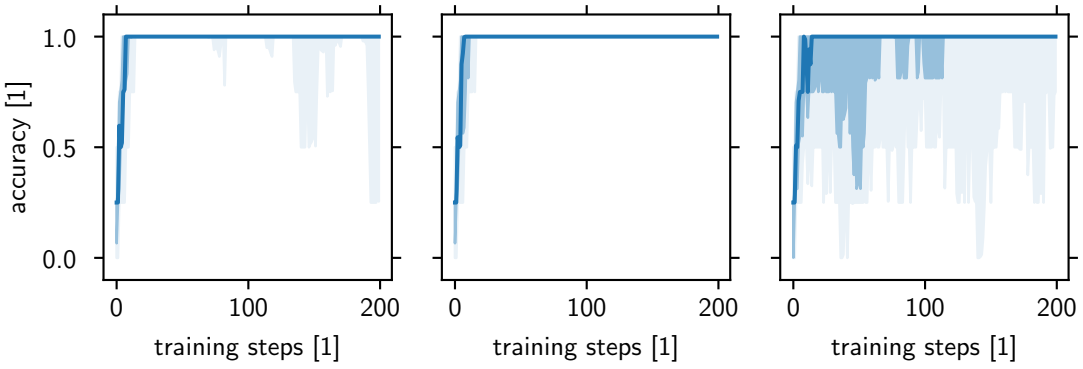


Figure 7.10: For the equal-time algorithm and $\rho = 1$ the training is shown as in Fig. 7.9.
The separations in terms of τ_{syn} are approximately 0.85, 1.5 and 9.0. For small separation the learning is unstable as for $\rho = 2$. Here the training is also unstable for large separation. The smallest separation is chosen as the first one where instabilities become visible, here for 0.85.

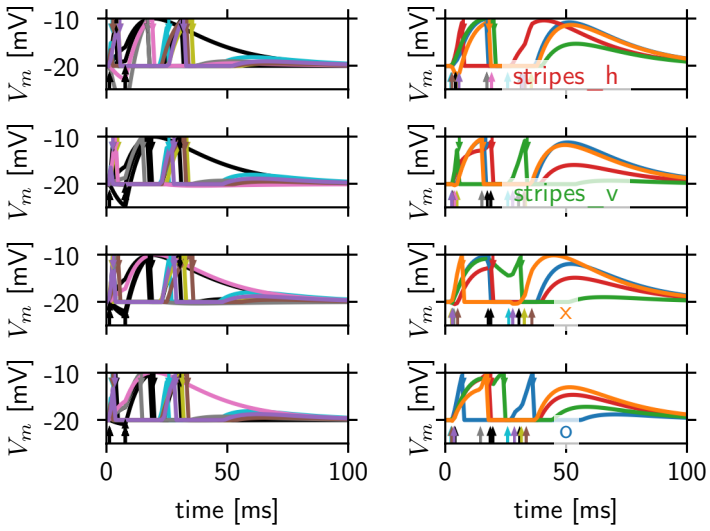


Figure 7.11: Response of the network for a separation of the input spikes of 6.5 ms. With the double-time algorithm and $\rho = 2$ the network is able to correctly classify the inputs. The classification is fast and the separation of the output spikes is large. The parameter file of the setup is Listing 9.

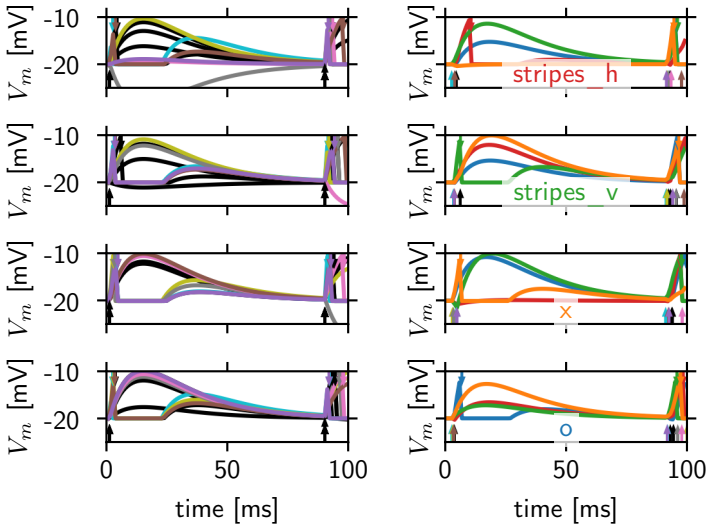


Figure 7.12: For a large separation of the input spikes close to 90 ms the voltage traces are shown as in Fig. 7.11.
At the time the late spikes of the input are received the membrane voltage has already decayed close to the leak voltage. Still, the network classifies the inputs correctly. The parameter file is Listing 10.

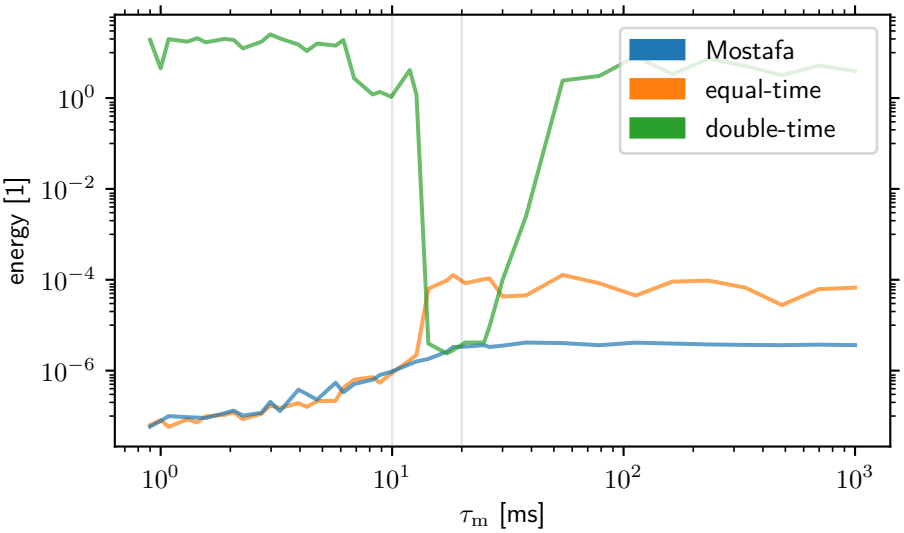


Figure 7.13: Fitness of different formulas for varying τ_m.
The average energy after training 1000 steps with the gradient function is displayed over τ_m in a log-log plot. Each data point is the average over the last 5 saved steps of the median energy for a sweep of 10 seeds. For the median energy of a sweep look at e.g. Fig. 7.5. A low energy implies high accuracy and good separation of the spikes. High energy implies that training was not fully achieved or collapsed again. $\tau_\mathrm{syn} = 10\,\mathrm{ms}$, i.e. $\tau_\mathrm{m} = 10^1\,\mathrm{ms}$ corresponds to $\rho = 1$. $\rho = 1$ and $\rho = 2$ are marked by vertical lines, $\rho = \infty$ is approached for large τ_m. An example parameter file is found in Listing 6.
The double-time formula learns correctly around its ideal case $\rho \approx 2$. The equal-time and Mostafa formula learn successfully throughout the whole range of τ_m. At each of the three regarded values of ρ, the formula derived for that ratio has among the best result (orange at the left vertical line, green at the right verticle line, and blue towards the far right).

7.3 Equal-Time, Double-Time and Mostafa Formula for Varying τ_m

The three formulas, equal-time, double-time, and Mostafa, are derived for one exact ratio $\rho := \tau_\mathrm{m}/\tau_\mathrm{syn}$ (Chapters 2 and 4). An interesting question is what happens when the ratio in a simulation is changed. The expectation is that each formula works at least in a range around its ideal ratio. The result is seen in Fig. 7.13.

The double-time formula shows the expected behaviour. However, the equal-time and Mostafa formula in my framework show training success at all τ_m. Around every ideal case, $\rho = 1$, $\rho = 2$, and $\rho = \infty$, the respective gradient formula has among the optimal success.

In case of correct classification and a separation of $1\,\tau_\mathrm{syn}$ between the output spikes, the energy is approximately $6 \cdot 10^{-5}$. An energy around this values thus shows sufficient classification. Energies below this order of magnitude do not point to better classification

per se.

7.4 Classifying Reduced MNIST in a Deep Network

I trained my algorithms on the reduced MNIST data set (Section 5.2). The aim of the thesis is to train spiking networks with time-to-first-spike coding on analogue hardware (Chapter 8). Thus, this part of the thesis is proof-of-principle and not aimed towards parameter tuning and perfect classification rates.

Furthermore, as a simulator I used NEST, as opposed to an event-driven simulator as in [*Mostafa*, 2017]. An event driven simulator calculates the spike times directly, NEST simulates the membrane voltage at every step, which is slower. Simulations took a long time, and therefore no parameter sweeps were done. It is very likely that the results can be improved significantly by finding better parameters.

I present results for a reduced MNIST data set inspired by [*Schmitt et al.*, 2017] and described in Chapter 5. The network has a hidden layer with 15 neurons. Training was done for the double-time formula (Fig. 7.14) and the equal-time formula (Fig. 7.16a. Learning improves the accuracy and after training it is at 90% for the double-time formula.

The confusion matrix after training (Fig. 7.15) indicates the individual accuracy of the classes. For this training, the diagonal is prominent, suggesting good classification. This is furthered when looking at example voltage traces, Fig. 7.17.

The voltage plot is already crowded, the relevant information is extracted by only showing the spikes in a raster plot. A comparison of spike times before and after training is Fig. 7.18. While the spikes happen close to each other prior to training, the correct neurons spike first after training. There is a clear separation, furthermore some neurons spike twice.

The experiments in this Section show applicability of my framework for the task of classifying MNIST digits. In [*Schmitt et al.*, 2017], the accuracy was 97% for the pure software model and $95^{+1}_{-2}\,\%$ for the model on hardware at the end of the in-the-loop training.

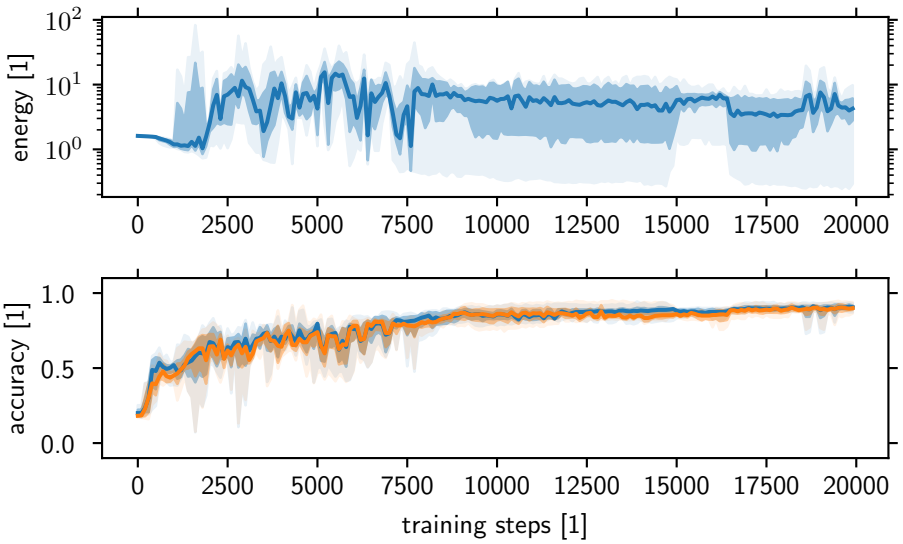


Figure 7.14: Training results for reduced MNIST in a deep network.
Training results for a 100-15-5 network learning with the double-time algorithm is shown for 4 seeds like in Fig. 7.5. Because the MNIST data is separated in training and test data, accuracy for both test data (blue) and training data (orange) is displayed. The energy is calculated from the test data.
At the end the accuracy for both training and test data is at 90%. An example parameter file is given in Listing 16.

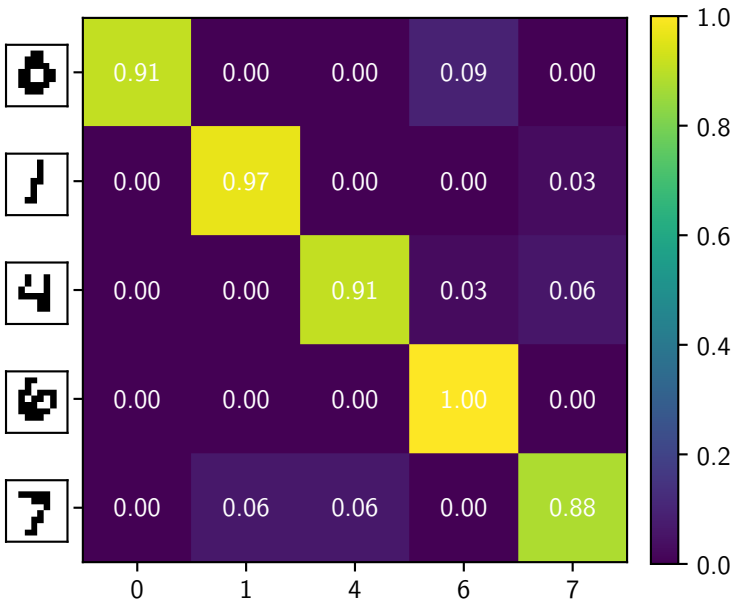
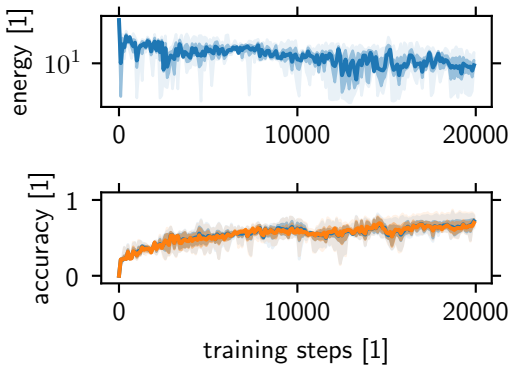
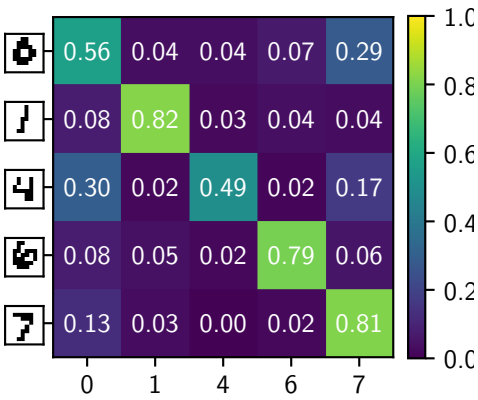


Figure 7.15: Confusion matrix of the double-time network for reduced MNIST. Classification is displayed for Listing 16. A confusion matrix is a breakdown of the accuracy. The matrix displays the classification results (columns) of the inputs (rows) in a colour map. A perfect classification would show as an identity matrix. Example images of the inputs are shown in front of the rows. This plot allows finding potential systematics in misclassification. The diagonal is prominent, which corresponds to good classification (between 0.88 and 1.00). There is no systematic in the off-diagonal elements.



(a) Training process for 4 seeds is shown. At the end both accuracies are at 70%.



(b) The confusion matrix has a less prominent diagonal compared to the result for $\rho = 2$. The correct classifications are between 0.49 and 0.82.

Figure 7.16: Results for the equal-time algorithm and $\rho = 1$ like Figs. 7.14 and 7.15. Example parameters can be found in Listing 17.

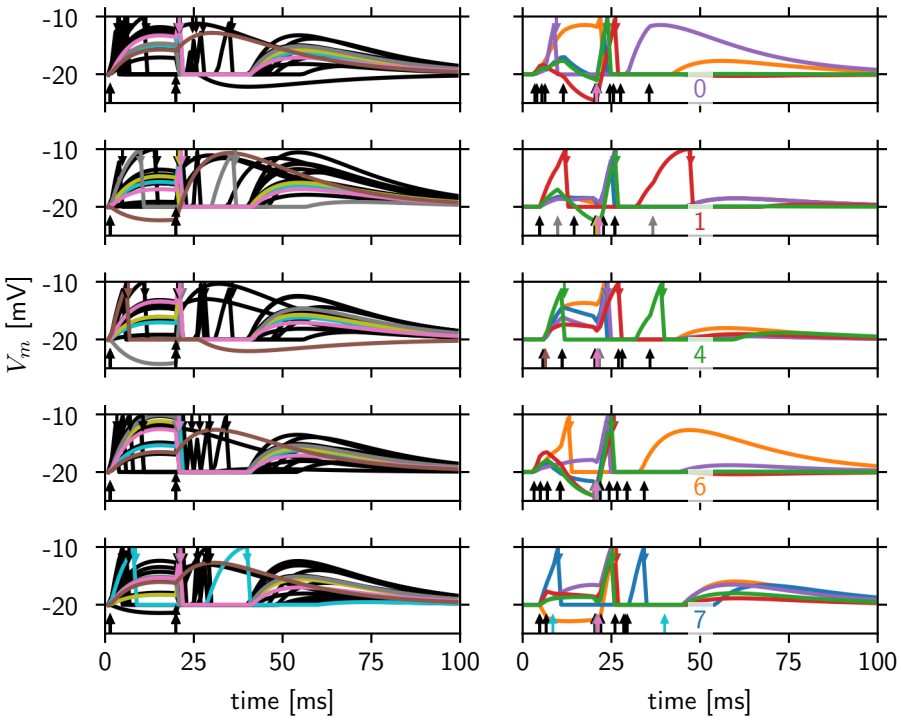
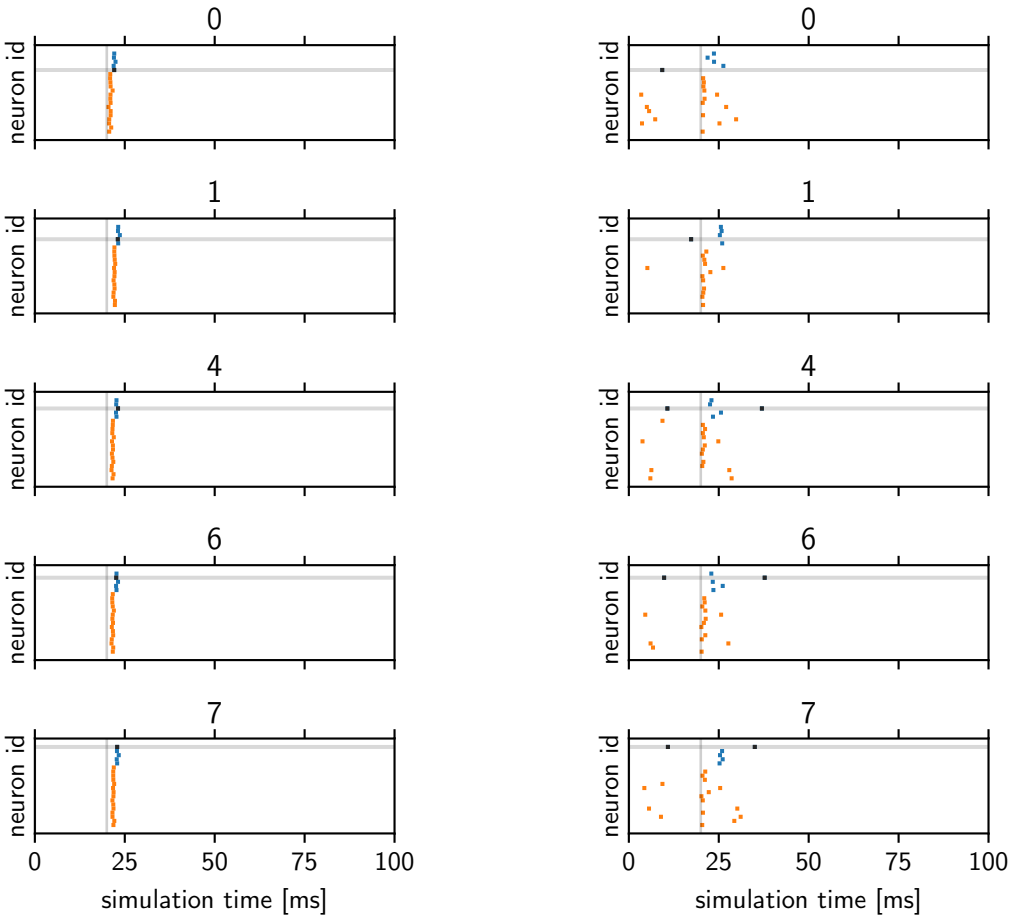


Figure 7.17: Voltages of the double-time network for reduced MNIST after training. Plot like Fig. 7.2 for Listing 16 that was also shown in Fig. 7.15. The classifications for the shown examples are all correct and the separation is large.



(a) Spike times of the randomly initialised network. All neurons spike right after the late input spikes.

(b) Spike times after training. The correct output neuron spikes first in the shown examples. Some neurons spike twice, seen best when it is the highlighted neuron.

Figure 7.18: Raster plot of the double-time network for reduced MNIST. For examples from the input classes the spike times of both layers are shown in a raster plot before and after training. The hidden layer is shown as orange dots, the label layer as blue dots. The neuron corresponding to the correct class is highlighted by the grey bar. Its spikes are shown as black dots. For comparison, the times of the input spikes are 1.5 ms for early spikes (black pixels of the input patterns; very left of each plot) and 20 ms for late spikes (white pixels of the input patterns; vertical grey line).

7.5 Integrating Features of the Hardware into the Learning

The simulations up to now are done for ideal neurons with CuBa synapses and floating precision weights. Before studying training on the hardware, I investigate training networks with features of the hardware.

I begin by discretising weights and show training success for as few as 7 available weight values. Next, using CoBa synapses with an appropriate WSF instead of CuBa synapses allows networks to learn nonetheless. At the end, networks of neurons with fixed-pattern noise are trained to success.

7.5.1 Training Neurons with Limited Weight Precision

On the hardware the weights are set digitally with 4 bit values allowing for 16 different weights. My framework allows to round the weights to a specified precision before writing them to the network (Section 5.1).

In Fig. 7.19 a sweep is shown where the precision was set to 0.1 pA. The learning happens fast and stable. For the different seeds, the cardinality (the number of used weights, cf. Chapter 5) of the weights between input and hidden layer was 2 or 3. The cardinality of the weights between hidden and label layer was between 3 and 7.

The discrete nature of the weights can be seen in the energy where plateaus are visible. The weight evolution of one example seed (Fig. 7.20) shows the reduced number of available weights as well. For $\rho = 1$ and the equal-time algorithm the results are similar.

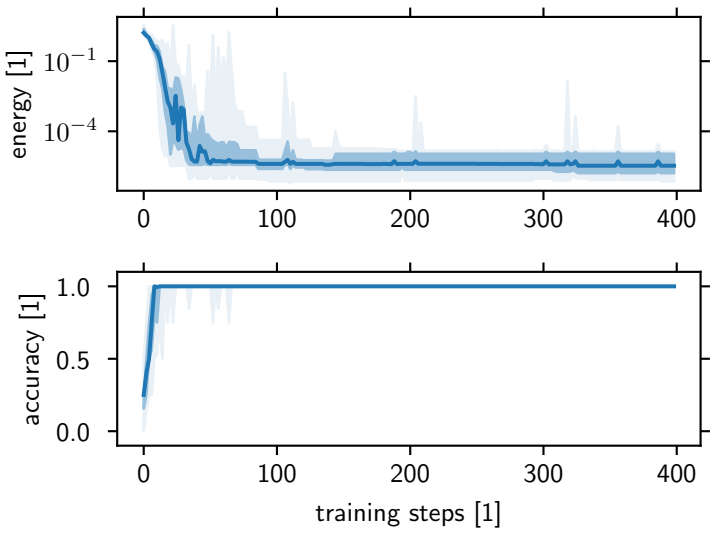
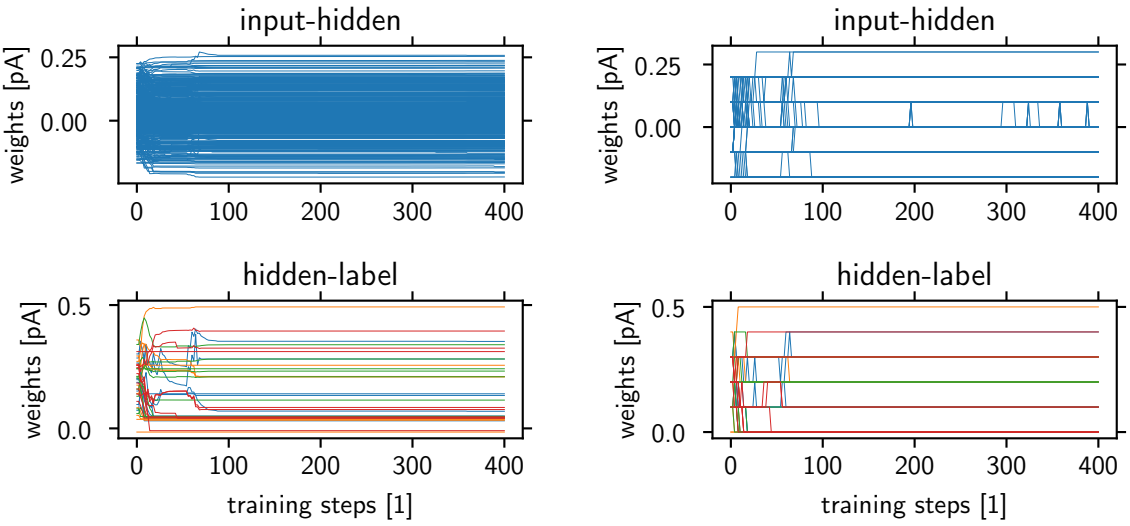


Figure 7.19: Training of a network with digital weights for $\rho = 2$ and 10 different seeds. The weights are rounded to a precision of 0.1 pA. Between the first layers the highest weight is 0.3 pA, 3 different weights are used. The numbers are 0.7 pA and 7 between the other layers. The learning proceeds fast and stable. The faint red lines are the times the drive weights mechanism is used. The transparency is scaled with the number of seeds. An example parameter is in Listing 11.



(a) float precision weights

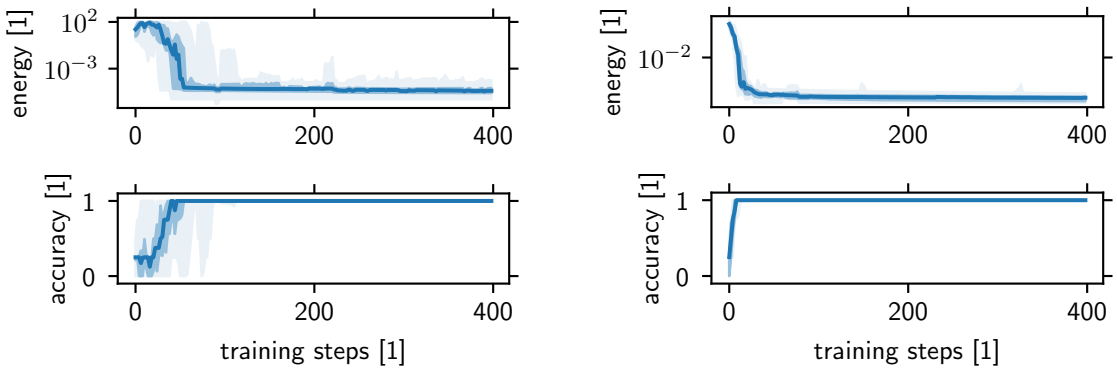
(b) rounded weights

Figure 7.20: The weight evolution (see Fig. 7.4) of both the float precision and rounded weights is shown for one seed, Listing 11.
The float precision is kept during training. Before updating the synapses, the weights are rounded. The precision of the weights in this case is 0.1 pA. The synapses to the hidden layer have 4 different weights (absolute value). The synapses to the label layer have 6 used weights.

7.5.2 Training Neurons with Conductance-Based Synapses

The formula for the spike time for neurons with CuBa synapses can be derived because there is a closed-from solution for the membrane voltage. For neurons with CoBa synapses that are used on the hardware there is no closed-from expression. In the derivation (Chapter 4) I introduced an approximation using the CuBa formulas and the *weight scale factor* (WSF). I showed the quality of the approximation for spike time in Chapter 6. In this Section I show that the training succeeds as well.

For $\rho = 2$ and the double-time formula the training process (Fig. 7.21) is very similar to CuBa synapses. The equal-time formula learns slower but just as stable. Towards the end of training there is little change, pointing to an equilibrium state. This is enforced in the weight evolution (Fig. 7.22), where little change is visible at the end. From these plots it can be concluded that networks with CoBa synapses can be trained.



(a) equal-time

(b) double-time

Figure 7.21: Training results for neurons with CoBa synapses.
The results shown are for both algorithms and 10 seeds each. Compared to Fig. 7.5, the double-time algorithm learns just as good in the CoBa case. The equal-time algorithm learns slower with CoBa synapses, but reaches a similar result. Example parameter are in Listings 12 and 13.

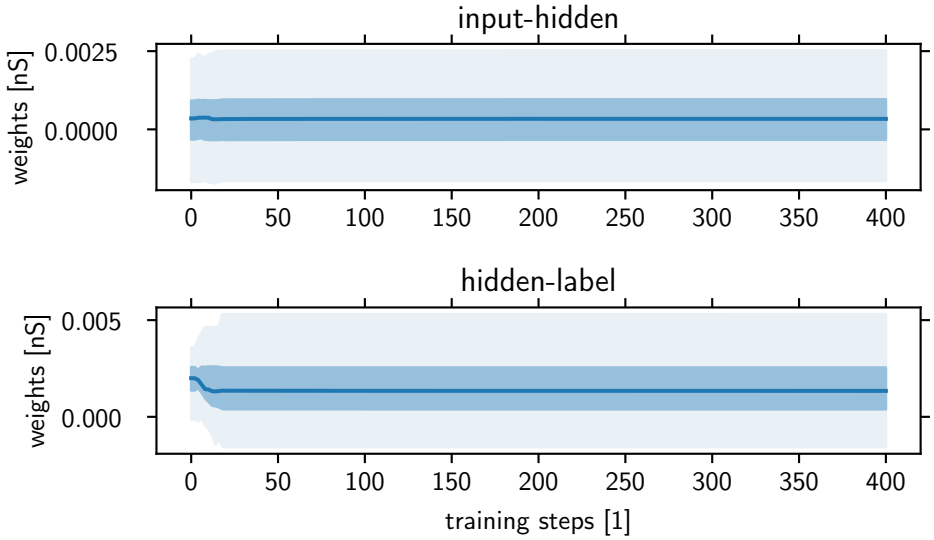


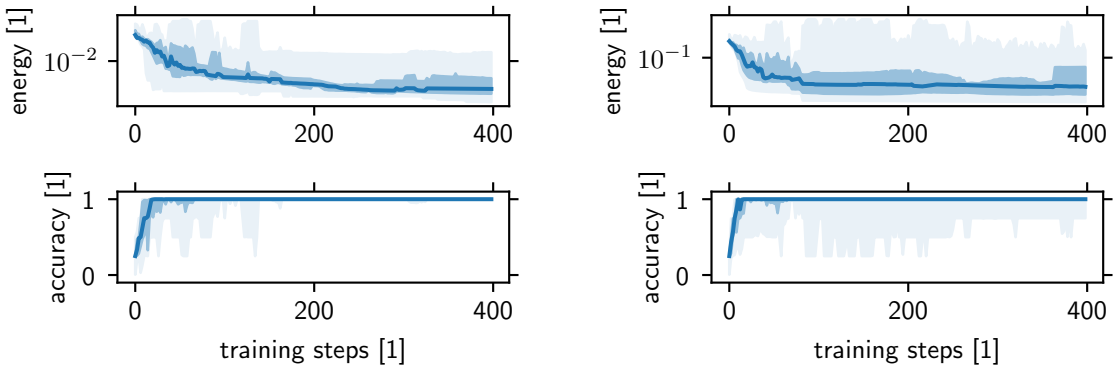
Figure 7.22: Weight evolution for learning with CoBa synapses and $\rho = 2$, see Listing 12. Only the evolution of the statistics of the weights is shown to provide a clearer plot. In both panels the 0, 25, 50, 75, and 100 percentiles of the weights are shown. The learning is stable and the weights approach an equilibrium. The weights here have a different unit, sievert instead of ampere, and are much smaller numerically.

7.5.3 Training Neurons with Fixed-Pattern Noise

The neuron parameters on the hardware are set with analogue values (Chapter 3). These values are not set to an exact value but in a range with some random deviation, creating so called fixed-pattern noise. The formulas, however, are derived for exact neuron parameters. Here I show the tolerance to fixed-pattern noise.

The equal-time formula has a higher tolerance than the double-time formula (Fig. 7.23). The noise is applied neuron-wise to the threshold V_{th}, leak voltage E_{L} and the two time constants τ_{m} and τ_{syn}, and leads to inter-neuron differences of those variables (see Fig. 7.24). For the equal-time formula, 20% noise is compensated and the training succeeds. Example voltage traces can be seen in Fig. 7.24. The double-time formula is less capable to train the noisy neurons. Because the time constants are affected as well, this was suspected from Fig. 7.13.

For $\rho = 2$ and the double-time formula noise below 15% is not a problem, and the training succeeds. However, I chose the plot for 15% noise. At least 7 of the 10 seeds successfully train, the faint line is only the 0 percentile. This just means that at some noise level some combinations will be too far away from the ideal values to allow training, while other combinations still work. This is also the expectation for the hardware (Chapter 8). Some jobs will work and some will not, because sometimes the FG values will be too far off.



(a) equal-time with 20% noise

(b) double-time with 15% noise

Figure 7.23: Training networks with fixed-pattern noise.
10 seeds are used for both formulas. The noise affects the time constants and potentials of the neurons, cf. Chapter 5 and Listings 14 and 15. For both algorithms the training is slower and less stable. The double-time formula can train with 10% and less noise.

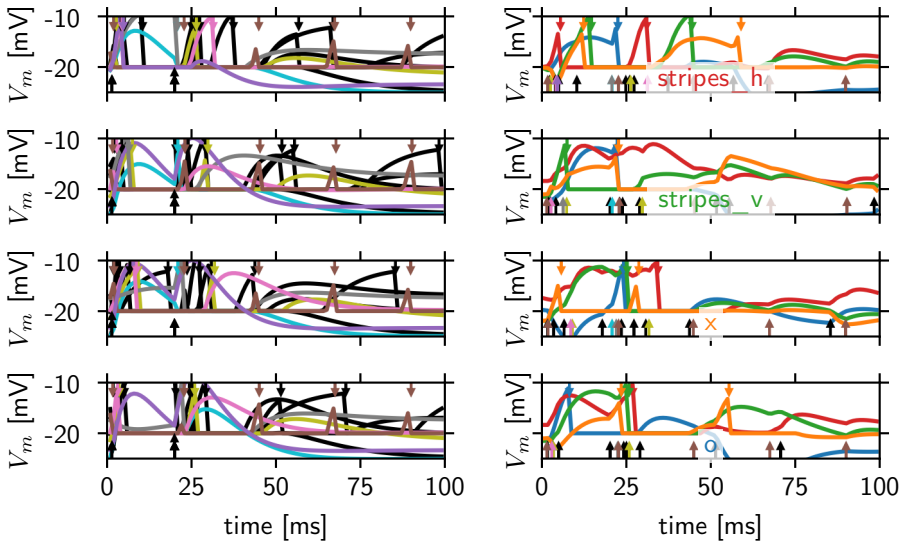


Figure 7.24: Effect of noise on membrane voltages.
Voltage evolution like in Fig. 7.2 for equal-time with 20% noise, see Listing 14, is shown. The noise affects the time constants, leakage and threshold voltage. The inter-neuron differences of the leakage is seen in the asymptotic voltages of the hidden neurons for example. The variations in the threshold can be seen in the difference between the potentials of the yellow and green neurons at the time of spikes.

8 Emulations on Hardware

In Sections 6.3.2 and 6.3.3, I showed predictability of spike times on hardware. In Chapter 7, I showed I can train spiking networks of leaky integrate-and-fire neurons. I also incorporated specifics of the hardware like fixed-pattern noise, CoBa synapses and digital weights into training (Section 7.5). In this Chapter, I train networks on hardware. All results in this Chapter are trained exclusively on hardware, i.e. starting from with a random weights initialisation.

Writing floating gates on hardware makes each job different (see Chapter 3 and Definitions 1 and 2). A number of jobs did not train to 100% accuracy. Due to fixed-pattern noise on neuron parameters this is expected as explained in Section 7.5.3, but needs to be investigated further.

The Weight Scale Factor, used wafer rack, and other settings can be found in Appendix A.3. Note that WSFs presented in Chapter 6 are newer values, with more data backing them. Experiments presented in this Chapter were executed before the detailed measurements in Section 6.3.3 were conducted and therefore use older WSFs. The values are detailed in Appendix A.3.

Except for Section 8.1.1, all data is from deep networks with one hidden layer. The deep networks have 20 hidden neurons and a network layout 49-20-4, and 49-20-3 for Section 8.2. For the latter, a visualisation of the mapping with the help of [*Boell*, 2018] is shown in Fig. 8.1.

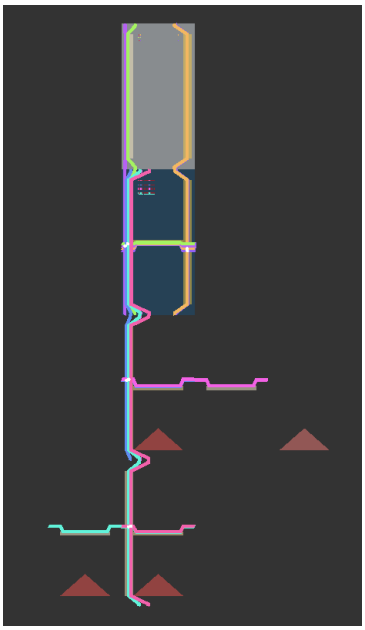


Figure 8.1: Visualisation of the mapping of the network in Section 8.2 according to [*Boell*, 2018]. The network is of shape 49-20-3 with the 20 hidden neurons on the HICANN shown in dark blue (HICANN 271). The label layer is the topmost area, HICANN 239. The inputs are distributed over 4 HICANNs (322, 323, 299, 301), indicated by the red triangles. The routes (see Fig. 3.2b) between the HICANNs are shown as coloured lines.

The networks in this Chapter were optimised based on the energy of exponential spike times (Section 7.2.3). Accordingly, that energy is shown in the training plots.

Improvement by optimising the linear energy is expected and planned. General suitability of the linear energy was proven (data not shown).

In addition to the equal-time and double-time formulas, I used the Mostafa formula in my framework, i.e. with the drive weights mechanism from Section 4.5. Section 7.3 showed that the Mostafa formula worked well for finite τ_{m} as well. To provide transparency, I provide a tuple of (ρ, formula, parameter File) for every figure in this Chapter. A quantitative analysis of the effectiveness of the different formulas should be done, but wasn't due to time constraints.

8.1 Training Patterns on Hardware

Patterns are used as an example data set that the network can solve. The high number of inputs allows for compensation for single faulty neurons. The model is defined in Chapter 5 and is the same as trained in the majority of Chapter 7.

The patterns used for training in this Chapter are balanced. This means the different classes have the same number of black pixels, equivalent to the same number of early spikes (Chapter 5). Unbalanced patterns (Chapter 5) can also be trained but training is less stable (data not shown).

8.1.1 Patterns in a Shallow Network

As in Chapter 7, learning in a shallow network is presented first. The training evolution in Fig. 8.2 shows a positive trend for the accuracy and a negative trend for the energy. The accuracy stays close to 100% and recovers after small drops.

An example voltage trace (Fig. 8.3) shows early and correct classification, and distinct separation. Inter-neuron variations due to fixed-pattern noise is visible.

This result shows that learning on hardware is possible in general and 100% accuracy can be reached for this data set. The next step is to train a deep network.

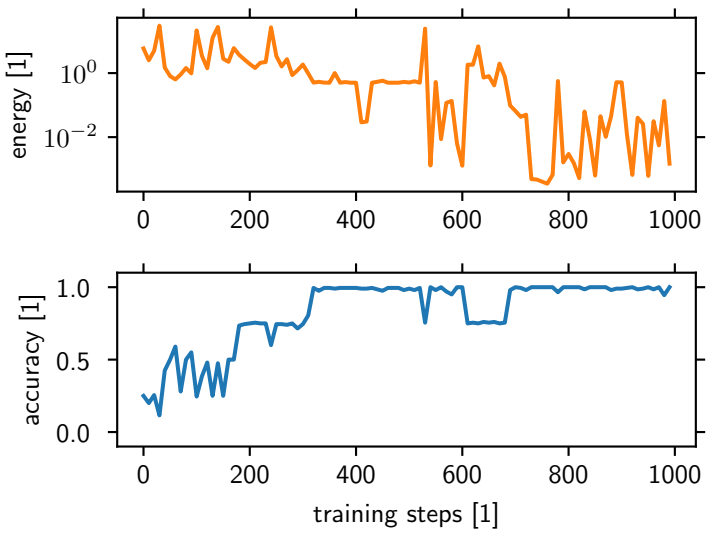


Figure 8.2: Training a shallow network on hardware.
($\rho = 1$, equal-time formula, Listing 19), the meaning of this tuple is explained in the text. The training is shown as in Fig. 7.1, but with the exponential energy (Section 4.4). The accuracy rises to close to 100%. Two times the accuracy is reduced, but quickly rises again. There is a negative trend for the energy, but with distinct fluctuations. The fluctuations also happen when the accuracy is nearly constant.

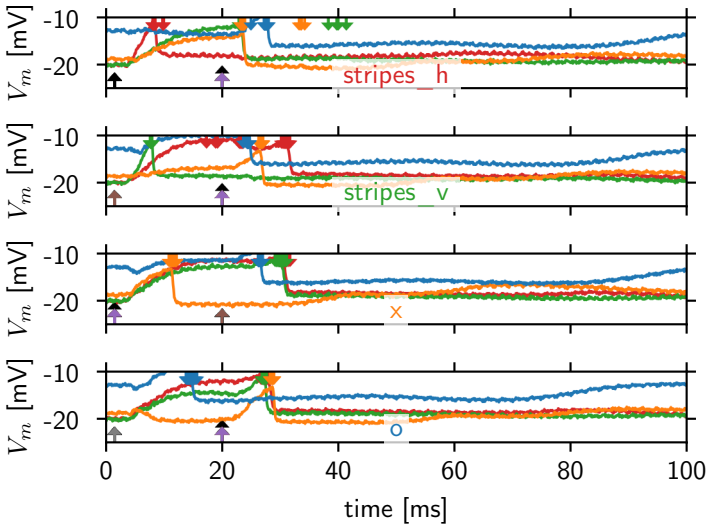


Figure 8.3: Voltage traces after training (Fig. 8.2), shown as in Figs. 6.8 and 7.2.
As explained in Chapter 5 the traces for different neurons are taken concurrently, for the accuracy thus look at the training evolution or potentially a confusion matrix. The separation of spikes can be extracted from a raster plot.
The inputs are classified correctly, and there is a large separation. The inter-neuron variation of the leakage (compare values for late times) and threshold (compare membrane voltage at time of spike, especially yellow and blue) are visible.

8.1.2 Patterns in a Deep Network

Training evolution for a deep network is shown in Fig. 8.4. Compared to the shallow network (Fig. 8.2) the process is slower with more fluctuations, and the energy at the end of training is larger.

A larger energy points to less separation. This is confirmed by Fig. 8.5. There is little separation between the spikes. Yet the confusion matrix (Fig. 8.6) shows the same high accuracy as seen in the training evolution (Fig. 8.4).

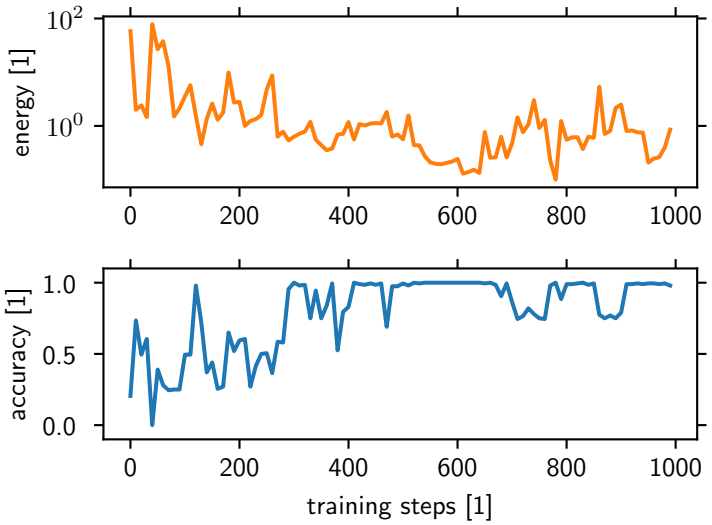


Figure 8.4: Training a deep network on hardware.
($\rho = 1$, equal-time formula, Listing 18). The network achieves 100%, but slower than before. During the recorded period collapses in the accuracy are reversed. Thus the classification at the end of the training is high, see Figs. 8.5 and 8.6. The energy fluctuates, with a negative trend.

In Section 7.2.1 sweeps for different seeds were done to show initialisation stability of my algorithm. The same seed results in different initialisations on hardware for different jobs due to the FGs (Section 3.1). Figure 8.7 shows a sweep for five different initialisations from five consecutive jobs. Consecutive means the data comes from a sequence of jobs, and the results are not hand-picked to only good data.

The accuracy rises fast and stays at 100% for the rest of training. The mean energy at the end suggests distinct separation

An example one of the five jobs confirms this suggestion (Figs. 8.8 and 8.9). Classification spikes happen around the time of the second input spike and there is indeed clear separation, even for the voltages. The confusion matrix is diagonal with entirely vanished off-diagonals (not shown).

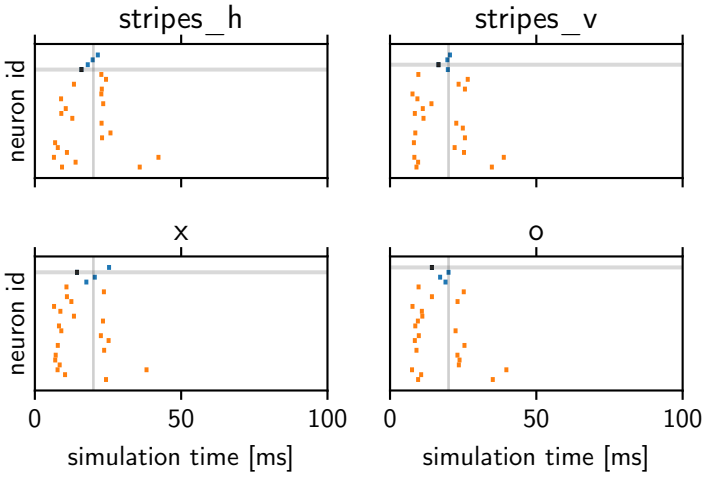


Figure 8.5: Example inputs of Fig. 8.4 are shown in a raster plot like Fig. 7.18. For each input, the correct output neuron spikes first with small but visible separation.

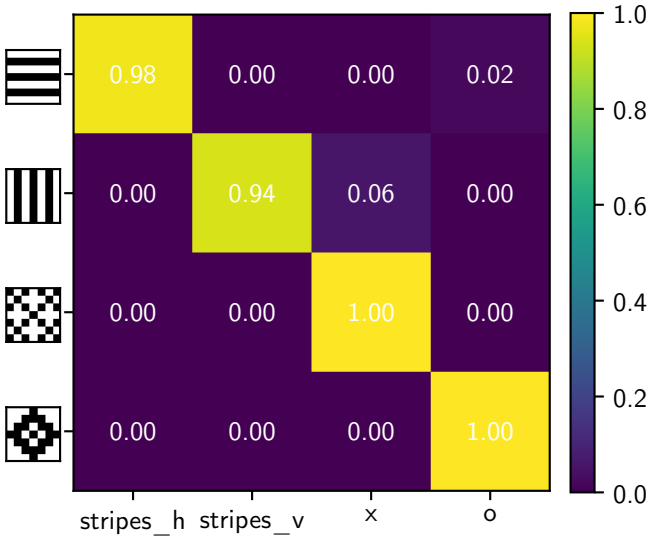


Figure 8.6: Confusion matrix of a deep network on hardware. Plotted like Fig. 7.15 is the job from Fig. 8.4. The confusion matrix is close to a unity matrix, i.e. close to perfect classification. With 50 examples for each class, see Listing 18, there are only four misclassified inputs.

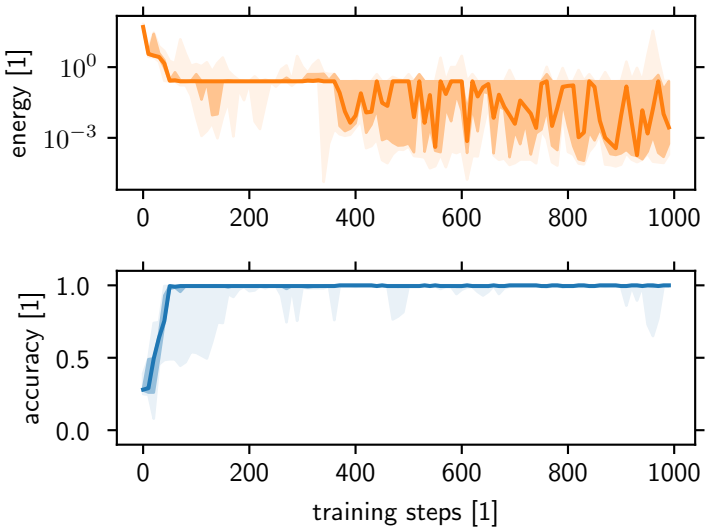


Figure 8.7: Training results of a deep network for five consecutive runs.
($\rho = 2$, Mostafa formula, Listing 20). The results are displayed like the sweeps in the earlier plots, cf. Fig. 7.5. There is no need to set different seeds, as setting the FGs provides a random setup. The accuracy rises to 100% fast and stays there. In fact, all but the 0 percentile are close to 100%, i.e. after 100 training steps four of the five results are at perfect classification at all times. The energy is smaller compared to Fig. 8.4. There is a negative trend in the energy.

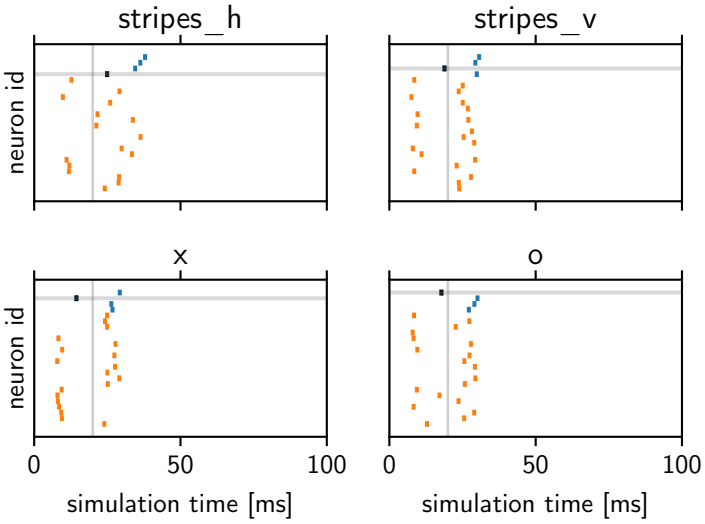


Figure 8.8: Raster plot after training of a deep network.
The classification spike is correct for each input and has distinct separation separation. The training process is shown for five jobs with the same setup in Fig. 8.7.

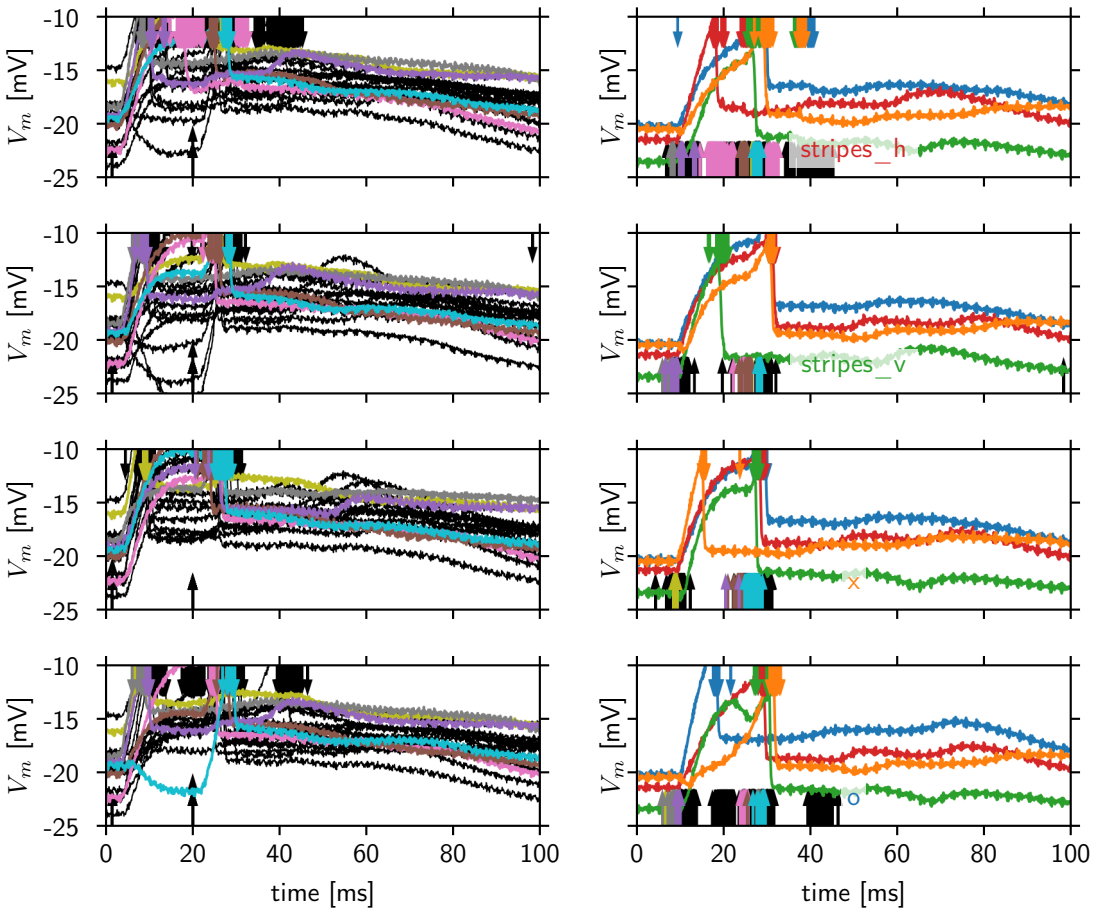


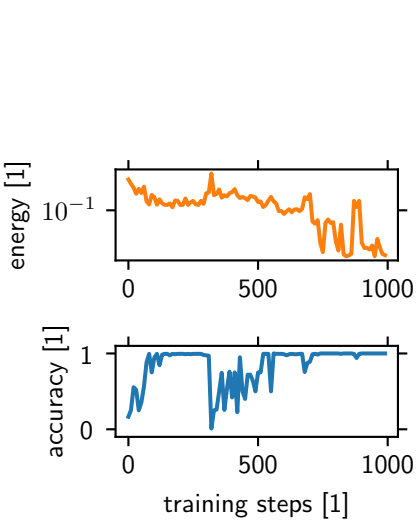
Figure 8.9: Voltage traces for one of job of Fig. 8.7 plotted as Fig. 8.3. Classification is early, correct and with a good separation. The sequence of spikes is seen better in Fig. 8.8.

8.1.3 Inverting the Patterns

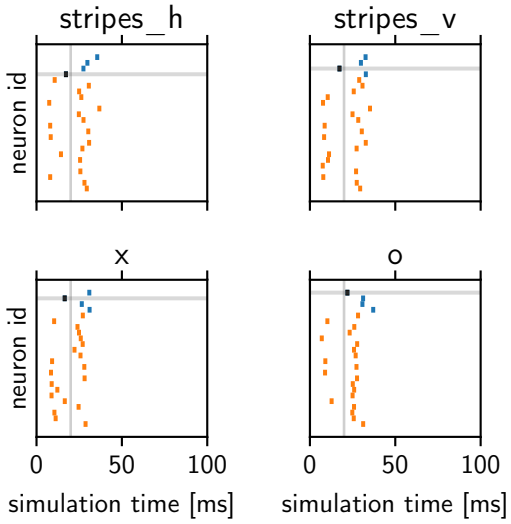
In Chapter 2 the input time coding was introduced. This coding translates black pixels to early input spikes and white pixels to late input spikes. This definition is arbitrary and the training should work for inverted attribution of spike times.

The training of a network with inverted patterns can be seen in Fig. 8.10. For clarity I interchanged black and white pixels and left the spike times unchanged, compare Fig. 8.11 with Fig. 8.6.

Training succeeds and the classification is distinct (Fig. 8.10b). While the accuracy collapses to 0% around 300 steps during training the network recovers to 100%. For the last 200 steps the accuracy stays close to 100%.



(a) The accuracy rises to 100% and the network recovers from a full collapse of the accuracy.



(b) The classification spikes (black) are correct and early, with a good separation from the subsequent spikes (blue).

Figure 8.10: Training inverted pattern on hardware.
($\rho = 2$, Mostafa formula, Listing 21). Inverting the patterns exchanges the black and white pixels in the patterns (see the example patterns in Fig. 8.11). For the network it amounts to exchanging the early and late input spikes.

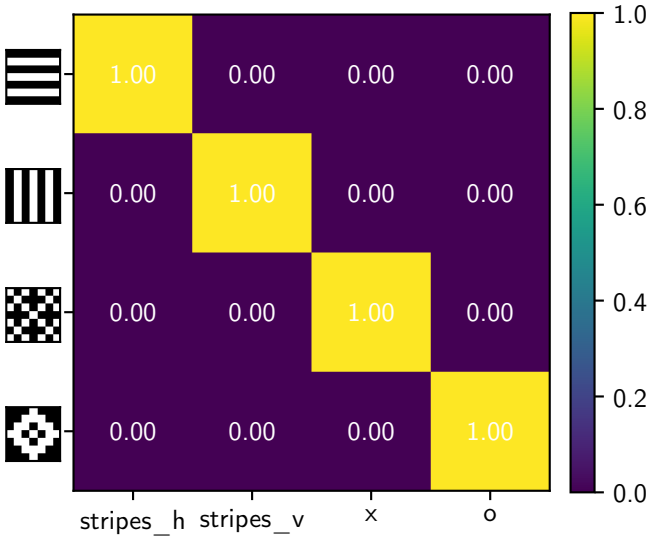


Figure 8.11: Confusion matrix after training for inverted patterns.

8.2 Reduced 7×7 MNIST on Hardware

The patterns used before are a good example data to establish the fundamentals of the algorithm. The MNIST data set (Chapter 5) is an established benchmark where the images are not derived from one perfect image. In MNIST the training and test images are different data.

The 28×28 original images are adapted in size to fit the already mapped networks and thus averaged to 7×7 pixels (Chapter 5). To counter reduced level of detail in the images, only the subset of the 0, 1, and 4 digits are used. This is simplified from the original data set, but the network still classifies data it has not been trained with.

The training increases the accuracy fast and proceeds in a convergent stable manner (Fig. 8.12). The energy reduces promptly at the beginning and fluctuates around a plateau during training.

The confusion matrix after training (Fig. 8.13) is nearly perfect, for 150 tested images only two images were misclassified. In the example voltage traces (Fig. 8.14) correct classification and clear separation is visible.

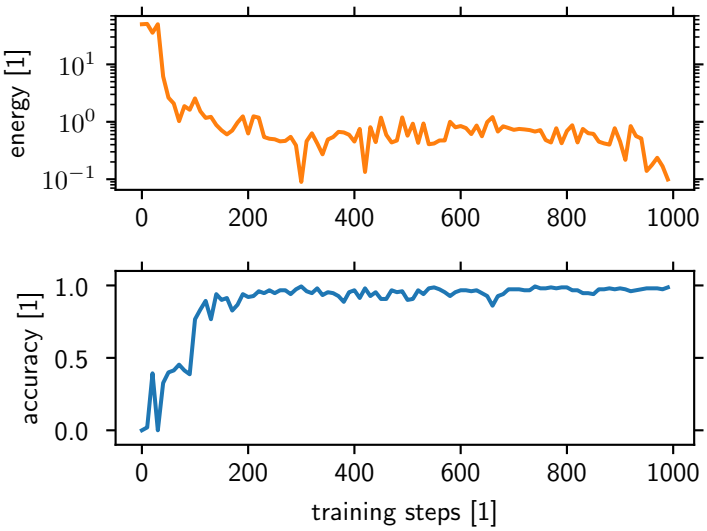


Figure 8.12: Training of a deep network to classify 7×7 digits.
($\rho = 2$, Mostafa formula, Listing 22). The accuracy rises fast and stays close to 100%. Only the accuracy of the test data is shown. The energy shows a negative trend with a lot of fluctuations.

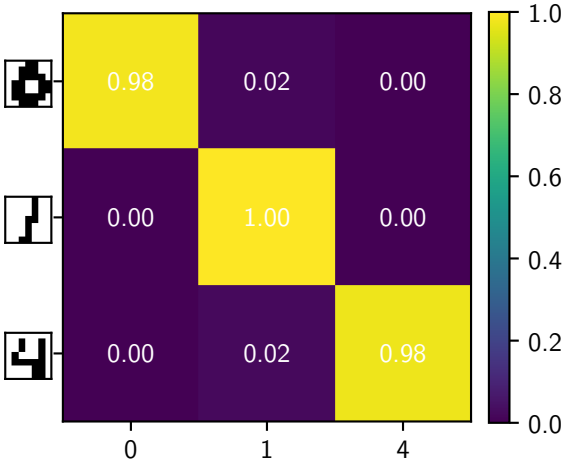


Figure 8.13: The confusion matrix of Fig. 8.12 as in Fig. 7.15.
As 50 images per class were used for testing (Listing 22) only 2 images of 150 are misclassified.

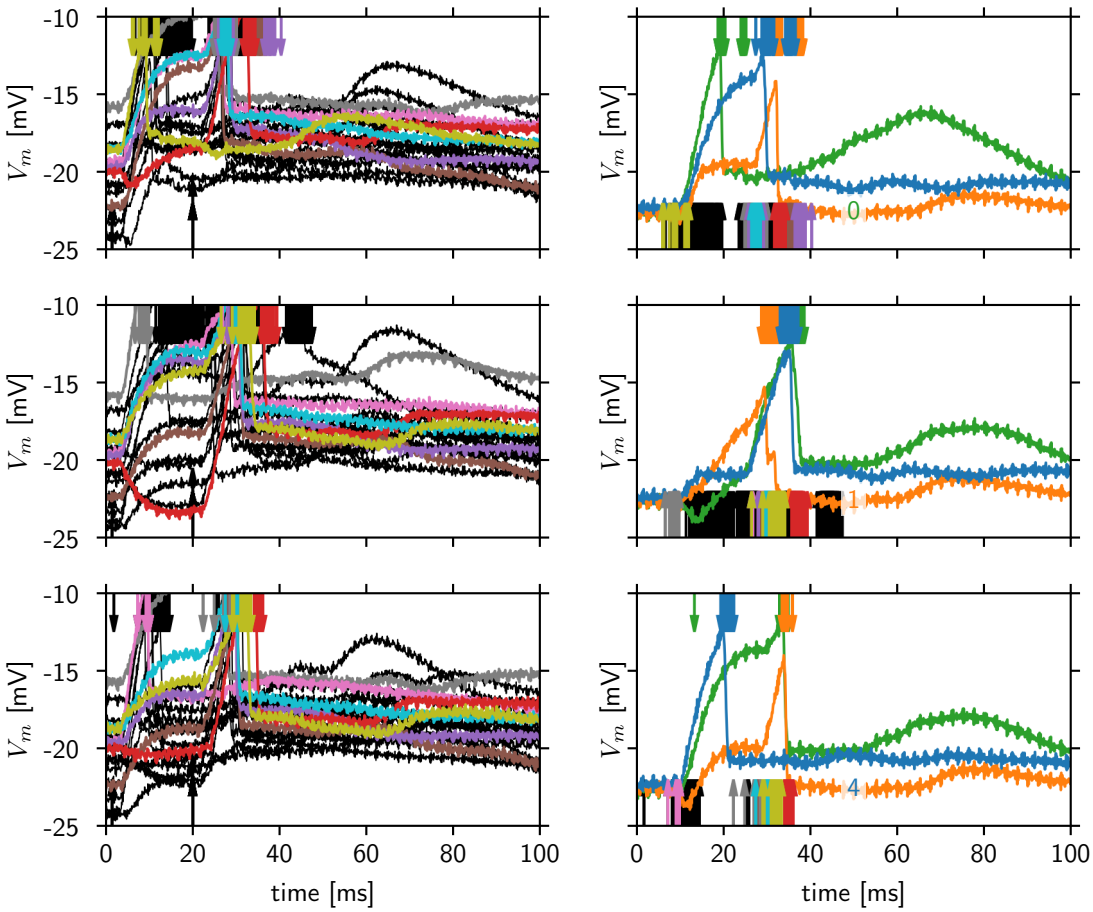


Figure 8.14: Voltages of Fig. 8.12 at the end of training, like in Fig. 8.3. Classification is correct with visible separation. The inter-neuron variations of the parameters is also visible.

9 Summary

In this thesis, deep networks were trained on the *BrainScaleS* (BSS) wafer-scale system with time-to-first-spike coding inspired by [*Mostafa*, 2017]. This presented a challenge due to the analogue nature of the BSS system and significant enhancements of the initial ansatz were needed to achieve learning on hardware.

The networks were set up in a layered fashion (Chapter 2) and trained to recognise patterns. The patterns were binarised images of black and white pixels, that were translated to early and late input spikes for the first layer, respectively. The weights were updated through gradient descent of the energy based on the spike times in the label layer (Section 4.4). The goal was to have the neuron in the label layer associated with the given class spike first. For error backpropagation in deep networks, gradient descent required differentiable relations for the spike times, which were extended in this thesis to work for LIF neurons as well.

At first, the theoretical foundation were looked at. Differentiable formulas for the time-to-first-spike of neurons with finite membrane time constant were derived under particular parametrisations, termed the equal-time ($\rho := \tau_\mathrm{m}/\tau_\mathrm{syn} = 1$) and double-time ($\rho = 2$) formula (Section 4.2). This analytical derivation was followed by an approximation of *conductance-based* (CoBa) synapses to *current-based* (CuBa) synapses by means of the *weight scale factor* (WSF) (Section 4.3). It became apparent that the energy used in [*Mostafa*, 2017] for gradient descent could be improved to further stabilise training (Sections 4.4 and 7.2.3). The approximation and energy analysis included predictions that were then proven in the subsequent Chapters 6 and 7.

In Chapter 6, a quantitative analysis of the prediction for neurons with CuBa synapses was done (Section 6.1) alongside a quantitative analysis of the reproducibility of spike times on hardware (Section 6.3.2). The analysis for the prediction with CoBa synapses including the WSF was done in a qualitative manner (Section 6.2). This sufficed because the prediction error even in the worst case was lower than the variation of spike times on hardware for the best case.

Then, example networks were trained with the formulas derived in this thesis (Section 7.1). It was shown that the weights get updated such that the correct label neuron spikes distinctly ahead of all others, and the weights approach an equilibrium (Section 7.1). Independence of the training results on the initial weight configuration was confirmed (Section 7.2.1) and increased stability due to the use of linear spike times in the energy (Section 4.4) was substantiated (Section 7.2.3). This benefit was predicted in Chapter 4 and, additionally, increased convergence speed could be seen. Although

after training classification was often independent of late input spikes (Figs. 7.2b, 7.6, 7.11 and 7.12), they played an essential role during training.

In Section 7.3 it was shown that the double-time formula relied on the ratio $\rho = \tau_{\mathrm{m}}/\tau_{\mathrm{syn}}$ of time constants (Section 4.2) to be close to its ideal ratio of $\rho = 2$ while the equal-time formula and the formula from [*Mostafa*, 2017] work in a broad range of ρ. It is surprising that the framework established in this work enables the formula from [*Mostafa*, 2017] to reliably train networks with finite membrane constant. The effectiveness of the Mostafa formula in spite of limited predictive capability (data not shown) could not be explained and needs to be investigated further.

Preceding the learning on hardware, it was shown that the algorithm can incorporate specific features of the hardware such as limited precision weights, CoBa synapses and fixed-pattern noise on neuron parameters (Section 7.5).

The general capability to train networks to classify the MNIST data set was shown in a reduced setting (Chapter 5) inspired by [*Schmitt et al.*, 2017], but the classification rate was below state-of-the-art algorithms on the full data set [e.g. *Cireşan et al.*, 2010] and also below comparable designs on the full data set [*Mostafa*, 2017]. However, there was no time for an intensive investigation of this subject.

In [*Mostafa*, 2017] ideas for improving the classification for MNIST are given, like an exponentially decaying learning rate, optimisation of the input separation, a lot more training steps (more than one order of magnitude) and an additional input that signals the start of a pattern to all layers. These mechanisms seem to make a big difference for the MNIST data set but they are not necessary to train the patterns introduced in Chapter 5, hence they were not implemented in the framework due to time constraints.

In [*Mostafa*, 2017] it is noted that no training with dropout was possible, as every single hidden layer neuron was important for classification and removing any neuron affected the output spike times. With its trial-to-trial variability, a robust implementation on analogue neuromorphic hardware is difficult. Thus a framework that is able to train a network starting from random initials is remarkable. In Section 8.1, it was shown that training a network to classify patterns succeeds for both shallow and deep networks with robustness to the initialisation.

Furthermore, a network can be trained to classify a reduced subset of MNIST digits presented with 7×7 pixels (Section 8.2). The classification happens in $4\,\mu$s wall-clock time (40 ms biotime) and each neuron spikes at most once. Fewer spikes improve the energy efficiency of the hardware and the classification speed is noteworthy as well. In [*Schmitt et al.*, 2017] patterns were presented for $90\,\mu$s wall-clock time (900 ms biotime), thus the time-to-first-spike coding is more than 20 times faster.

Findings from other Chapters like increased stability due to the different energy (Section 7.2.3) and an improved WSF (Section 6.3.3) have not been included in experiments of Chapter 8 yet and potentially improve the results on hardware. Moreover, the size of the current network does in no way use the hardware to the full extent and scaling up the network is already in progress.

10 Outlook

Training spiking networks with time-to-first-spike coding on analogue hardware is a new approach. Questions regarding improvements in learning on hardware that could not be addressed in the time scope of this thesis are discussed here.

While an analysis of the quality of prediction was done for different scenarios, an additional quantitative investigation with randomly generated input spike patterns would provide a solid basis for comparison of different hardware neurons. The gained statistics of prediction and *weight scale factor* (WSF) for neurons on different *high input count analog neural network* (HICANN) chips can explain potential systematic training differences for changing hardware setups.

With a working framework on hardware that achieves similar results as the simulation, it becomes more important to get closer to the classification results in software from [*Mostafa*, 2017]. Implementing the addenda from [*Mostafa*, 2017], a decaying learning rate, a start spike to all layers and optimised separation (Chapter 9), is the next step. New simulations with these features can improve classification, that is, reduce the error-rate for classifications on the test set as well as increase the training stability. The presented framework is set up in such a way (Chapter 5) that optimising mechanisms for simulations can directly be used for emulations on hardware, too.

An investigation and more thorough comparison of the three gradient formulas is in order. In the thesis (Section 7.3), the ratio $\rho = 2$ is identified as a point where all three gradient formulas work. Evaluating the training process in detail allows for adaptions of the gradient formulas to stabilise training.

On hardware, an immediately available action is to repeat training with improved energy (Section 7.2.3) and a well-founded WSF (Section 6.3.3). Optimising hyperparameters like batch size (Chapter 5), learning rate, and hardware parameters will improve convergence speed and robustness. This is easiest done by sweeps over the relevant parameters. Sweeping, in turn, will benefit from analysis and potential optimisation of the runtime of training on hardware. This includes acceleration of both the python code used for training and the sophisticated software stack developed by the group with ongoing progress [*Meehan*, 2019].

This thesis lays the groundwork for a quantification of the success rate of training on hardware. As pointed out in Section 7.5.3, a fraction of networks not succeeding to train on hardware is expected. However, the suspected reason, i.e. variations in *floating gate* (FG) values, for limiting the success of training must be investigated. If a failure-predictor, e.g. a lack of precision in the prediction of spike times, can be determined, irredeemable emulations can be restarted or better yet reconfigured to work properly.

Executing several consecutive emulations without rewriting the FGs makes it possible to reuse the same FG values. This allows for investigations of trial-to-trial variability with unchanged FG values and initial weights as well as stability with respect to different initial weights. FG stability over time has been investigated for different tasks [*Millner*, 2012; *Kononov*, 2011; *Klähn*, 2017] and is sufficient for repeated emulations.

By scaling up the network and training on larger data sets, more complex tasks can be tackled. The first step is to scale up to [*Schmitt et al.*, 2017], serving as a fitting comparison due to its implementation on *BrainScaleS* (BSS). The network therein is approximately twice as large as the biggest network realised on hardware in this thesis, both in terms of inputs and neurons. The next achievable goal is to obtain good classification results on the full MNIST [*LeCun et al.*, 1998] dataset on hardware.

My extension of the formula in [*Mostafa*, 2017] realises energy-efficient and fast pattern recognition on neuromorphic hardware. In fact, new possibilities have been opened up for use cases that were not pursued because error backpropagation was not available for spiking neural networks.

This thesis is fundamental research, but using improved realisations of the framework on neuromorphic chips allows a principled deployment of fast and energy-efficient spike-based solutions to pattern recognition problems. Notably, the high speed of the classification makes this technology interesting for detectors in particle accelerators. At the very high luminosity regimes of modern colliders, detector setups routinely generate several orders of magnitude more information than can be saved to disk, rendering the efficient identification of interesting collision data one of the most mission-critical aspects of detector readout control. Consequently, fast event classification in early trigger systems is a topic of active development, to which the presented setup might deliver important contributions.

Appendix

A Parameters

For transparency I disclose all parameter sets used for the simulations in Chapters 6 to 8. For a detailed understanding, looking at the code[1] online at `https://openproject.bioai.eu/projects/model-tempodrom` or at the git repository `git@brainscales-r.kip.uni-heidelberg.de:model-tempodrom.git` can be helpful.

The parameters are either dimensionless or in the *NEural Simulation Tool* (NEST) default units.

A.1 Parameters for Chapter 6

All data in Chapter 6 including the data from hardware (except Section 6.3.2) was acquired anew with code from the commit `248da05fec83c08b3e5ac324b9b205890418cd3c`.

Section 6.3.2 was done with code from the commit `d8c3f337f1380d6db2c4d9c16fb1987037354606`.

```
params = {'V_rest': -20.,
          'V_diff': 10.,
          'E_ex_factor': 8.,
          'E_in_factor': -8.,
          'tau_syn': 10.}
params['tau'], params['time_per_step'], params['tau_ref']\
    = np.array((1., 10., 2.)) * params['tau_syn']
params['C_m'] = 0.2
params['V_th'] = params['V_rest'] + params['V_diff']
params['E_ex'], params['E_in'] = params['V_rest'] + \
    np.array((params['E_ex_factor'],
              params['E_in_factor'])) * \
    params['V_diff']
neuronparams = {
    'C_m': params['C_m'],
    'E_L': params['V_rest'],
    'I_e': 0.0,
    'V_m': params['V_rest'],
    'V_reset': params['V_rest'],
    'V_th': params['V_th'],
    't_ref': params['tau_ref'],
    'g_L': params['C_m'] / params['tau_m'],
    'tau_m': params['tau_m'],
    'tau_syn_ex': params['tau_syn'],
    'tau_syn_in': params['tau_syn'],
    'E_ex': params['E_ex'],
    'E_in': params['E_in'],
    }
```

Listing 1: Parameter used in Chapter 6. τ_m and g_max are different for the different scenarios, as are the input weights and spike times.

A.2 Parameters for Chapter 7

I tried to use similar initials for the simulations. Changing only few parameters aids understanding of those changes. The master file is given in Listing 2, the other parameters are shown as `diff`s to this file.

[1] An account is needed to view the code this way. If you do not have an account but are interested, contact the group or me directly.

All simulations shown in Chapter 7 were redone with the current code, and can thus be repeated with the commit
`fe4245b406b4fae86f38d77d2f501cf116d64fed`.

```
cache:
    E_ex_factor: 8.0
    E_in_factor: -8.0
    V_diff: 10.0
    V_rest: -20.0
    batch_number: 8
    cm: 0.2
    digitalise_weights_digits: 6
    num_snapshots: 200
    save_every_x_batches: 1
    tau_m: 20.0
    tau_ref: 20.0
    tau_syn: 10.0
    time_per_step: 100.0
    tmp_factor_for_rounding: 1
config:
    data: patterns_bal
    eval_level:
    - save_process
    - plot_process
    - save_voltages
    - plot_voltages
    gradient_function: 2
    input_popsize: 1
    network_layout:
    - 10
    network_layout_depth: 1
    neurons: iaf_psc_exp_ps
    objective: time
    sim_infinite_tau_m: true
    simulator: nest
eval_data:
    early: 1.5
    late: 20.0
    noisy_samples: true
    num_samples: 50
    truncated_spread: 0.5
input_data:
    early: 1.5
    late: 20.0
    noisy_samples: true
    num_samples: 50
    trainingOrTesting: training
    truncated_spread: 0.5
net:
    initials:
    - random
    params:
        C_m: 0.2
        E_L: -20.0
        E_ex: 60.0
        E_in: -100.0
        I_e: 0.0
        V_m: -20.0
        V_reset: -20.0
        V_th: -10.0
        g_L: 0.01
        t_ref: 20.0
        tau_m: 20.0
        tau_syn_ex: 10.0
        tau_syn_in: 10.0
    weights_mean:
    - 0.03
    - 0.175
    weights_std:
    - 0.1
    - 0.1
rates:
    absolute_regulizer: 0.0
    drive_weights: 0.001
    drive_weights_factor: 1.5
    drive_weights_old: 0.0
    drive_weights_thresh: 0.1
    l2_regulizer: 0.0
    learning: 0.1
    norm_grad_frob: 0.0
    norm_grad_l1: 0.0
    norm_grad_l2_colwise: 0.0
    norm_grad_lsup: 0.0
    norm_update_l1: 10.0
    update_wsc: 0.0
simulation:
    loglevel: M_ERROR
    numpy_seed: 85412
    pattern_separation_time: 100.0
    resolution: 0.01
    resolution_decimals: 2
```

Listing 2: Parameter file used as a master file for the simulations. The meaning of some of the parameters is given in Chapter 5.

```
@@ -8,3 +8,3 @@
     digitalise_weights_digits: 6
-    num_snapshots: 200
+    num_snapshots: 100
     save_every_x_batches: 1
@@ -25,3 +25,3 @@
     network_layout:
-    - 10
+    - null
     network_layout_depth: 1
@@ -46,3 +46,3 @@
     initials:
-    - random
+    - rand
     params:
@@ -62,6 +62,6 @@
     weights_mean:
-    - 0.03
+    - 0.015
     - 0.175
     weights_std:
-    - 0.1
+    - 0.01
     - 0.1
@@ -72,5 +72,5 @@
     drive_weights_old: 0.0
-    drive_weights_thresh: 0.1
+    drive_weights_thresh: 0.35
     l2_regulizer: 0.0
-    learning: 0.1
+    learning: 0.02
     norm_grad_frob: 0.0
```

Listing 3: Parameter file used for Figs. 7.1, 7.2a, 7.2b, 7.3 and 7.4, given as a difference to Listing 2.

```
@@ -46,3 +46,3 @@
     initials:
-    - random
+    - rand
     params:
@@ -72,3 +72,3 @@
     drive_weights_old: 0.0
-    drive_weights_thresh: 0.1
+    drive_weights_thresh: 0.35
     l2_regulizer: 0.0
```

Listing 4: Parameters of Fig. 7.7a compared to Listing 2.

```
@@ -22,3 +22,3 @@
     - plot_voltages
-    gradient_function: 2
+    gradient_function: 0
     input_popsize: 1
@@ -28,3 +28,3 @@
     neurons: iaf_psc_exp_ps
-    objective: time
+    objective: time_exp
     sim_infinite_tau_m: true
@@ -69,5 +69,5 @@
     absolute_regulizer: 0.0
-    drive_weights: 0.001
+    drive_weights: 0.0
     drive_weights_factor: 1.5
-    drive_weights_old: 0.0
+    drive_weights_old: 1.5
     drive_weights_thresh: 0.35
```

Listing 5: Parameter difference between Figs. 7.7a and 7.7b.

```
@@ -8,5 +8,5 @@
    digitalise_weights_digits: 6
-   num_snapshots: 200
-   save_every_x_batches: 1
-   tau_m: 20.0
+   num_snapshots: 100
+   save_every_x_batches: 10
+   tau_m: 30.000000000000032
    tau_ref: 20.0
@@ -19,5 +19,3 @@
    - save_process
-   - plot_process
-   - save_voltages
-   - plot_voltages
+   - discard_output
    gradient_function: 2
@@ -47,2 +45,3 @@
    - random
+   initials_nonRandFactor: 1.0
    params:
@@ -56,5 +55,5 @@
        V_th: -10.0
-       g_L: 0.01
+       g_L: 0.006666666666666666
        t_ref: 20.0
-       tau_m: 20.0
+       tau_m: 30.000000000000032
        tau_syn_ex: 10.0
@@ -83,3 +82,3 @@
    loglevel: M_ERROR
-   numpy_seed: 85412
+   numpy_seed: 2177
    pattern_separation_time: 100.0
```

Listing 6: Example parameter file for the comparison of the gradient formulas, see further Fig. 7.13

```
@@ -9,3 +9,3 @@
    num_snapshots: 200
-   save_every_x_batches: 1
+   save_every_x_batches: 2
    tau_m: 20.0
@@ -19,5 +19,2 @@
    - save_process
-   - plot_process
-   - save_voltages
-   - plot_voltages
    gradient_function: 2
@@ -47,2 +44,3 @@
    - random
+   initials_nonRandFactor: 1.4
    params:
@@ -83,3 +81,3 @@
    loglevel: M_ERROR
-   numpy_seed: 85412
+   numpy_seed: 2177
    pattern_separation_time: 100.0
```

Listing 7: Example parameter file for a sweep with 100 seeds, see Fig. 7.5.

```
@@ -9,3 +9,3 @@
    num_snapshots: 200
-   save_every_x_batches: 1
+   save_every_x_batches: 2
    tau_m: 20.0
@@ -19,5 +19,2 @@
    - save_process
-   - plot_process
-   - save_voltages
-   - plot_voltages
    gradient_function: 2
@@ -28,3 +25,3 @@
    neurons: iaf_psc_exp_ps
-   objective: time
+   objective: time_exp
    sim_infinite_tau_m: true
@@ -47,2 +44,3 @@
    - random
+   initials_nonRandFactor: 1.4
    params:
@@ -83,3 +81,3 @@
    loglevel: M_ERROR
-   numpy_seed: 85412
+   numpy_seed: 2177
    pattern_separation_time: 100.0
```

Listing 8: Example parameter file for comparison of the energies, see Fig. 7.8.

```
@@ -19,5 +19,2 @@
    - save_process
-   - plot_process
-   - save_voltages
-   - plot_voltages
    gradient_function: 2
@@ -33,3 +30,3 @@
    early: 1.5
-   late: 20.0
+   late: 8.0
    noisy_samples: true
@@ -39,3 +36,3 @@
    early: 1.5
-   late: 20.0
+   late: 8.0
    noisy_samples: true
@@ -47,2 +44,3 @@
    - random
+   initials_nonRandFactor: 1.4
    params:
@@ -83,3 +81,3 @@
    loglevel: M_ERROR
-   numpy_seed: 85412
+   numpy_seed: 2177
    pattern_separation_time: 100.0
```

Listing 9: Example parameter file for variations of the time of the later spike time, see Figs. 7.9 and 7.11.

```
@@ -19,5 +19,2 @@
     - save_process
-    - plot_process
-    - save_voltages
-    - plot_voltages
     gradient_function: 2
@@ -33,3 +30,3 @@
     early: 1.5
-    late: 20.0
+    late: 90.0
     noisy_samples: true
@@ -39,3 +36,3 @@
     early: 1.5
-    late: 20.0
+    late: 90.0
     noisy_samples: true
@@ -47,2 +44,3 @@
     - random
+    initials_nonRandFactor: 1.4
     params:
@@ -83,3 +81,3 @@
     loglevel: M_ERROR
-    numpy_seed: 85412
+    numpy_seed: 2177
     pattern_separation_time: 100.0
```

Listing 10: Example parameter file for variations of the time of the later spike time between Fig. 7.11 and Fig. 7.12.

```
@@ -7,5 +7,5 @@
     cm: 0.2
-    digitalise_weights_digits: 6
+    digitalise_weights_digits: 1
     num_snapshots: 200
-    save_every_x_batches: 1
+    save_every_x_batches: 2
     tau_m: 20.0
@@ -19,5 +19,2 @@
     - save_process
-    - plot_process
-    - save_voltages
-    - plot_voltages
     gradient_function: 2
@@ -47,2 +44,3 @@
     - random
+    initials_nonRandFactor: 1.4
     params:
@@ -83,3 +81,3 @@
     loglevel: M_ERROR
-    numpy_seed: 85412
+    numpy_seed: 2177
     pattern_separation_time: 100.0
```

Listing 11: Example parameter file for training with digital weights, compared to Listing 2. For the sweeped training see Fig. 7.19, for the weight evolution see Fig. 7.20.

```
@@ -9,3 +9,3 @@
     num_snapshots: 200
-    save_every_x_batches: 1
+    save_every_x_batches: 2
     tau_m: 20.0
@@ -19,5 +19,2 @@
     - save_process
-    - plot_process
-    - save_voltages
-    - plot_voltages
     gradient_function: 2
@@ -27,3 +24,3 @@
     network_layout_depth: 1
-    neurons: iaf_psc_exp_ps
+    neurons: iaf_cond_exp
     objective: time
@@ -47,2 +44,3 @@
     - random
+    initials_nonRandFactor: 1.4
     params:
@@ -62,7 +60,7 @@
     weights_mean:
-    - 0.03
-    - 0.175
+    - 0.0003
+    - 0.00175
     weights_std:
-    - 0.1
-    - 0.1
+    - 0.001
+    - 0.001
 rates:
@@ -83,3 +81,3 @@
     loglevel: M_ERROR
-    numpy_seed: 85412
+    numpy_seed: 2177
     pattern_separation_time: 100.0
```

Listing 12: Example parameter file with $\rho = 2$ for training CoBa synapses, compared to Listing 2. For the sweeped training see Fig. 7.21, for the weight evolution see Fig. 7.22. The WSF used in the training is 73.59.

```
@@ -10,3 +10,3 @@
     save_every_x_batches: 2
-    tau_m: 20.0
+    tau_m: 10.0
     tau_ref: 20.0
@@ -19,3 +19,3 @@
     - save_process
-    gradient_function: 2
+    gradient_function: 1
     input_popsize: 1
@@ -44,3 +44,3 @@
     - random
-    initials_nonRandFactor: 1.4
+    initials_nonRandFactor: 2.5
     params:
@@ -54,5 +54,5 @@
         V_th: -10.0
-        g_L: 0.01
+        g_L: 0.02
         t_ref: 20.0
-        tau_m: 20.0
+        tau_m: 10.0
         tau_syn_ex: 10.0
```

Listing 13: Example parameter file with $\rho = 1$ ms for training CoBa synapses, compared to Listing 12. For the sweeped training see Fig. 7.21. The WSF used in the training is 69.18.

```
@@ -9,4 +9,4 @@
     num_snapshots: 200
-    save_every_x_batches: 1
-    tau_m: 20.0
+    save_every_x_batches: 2
+    tau_m: 10.0
     tau_ref: 20.0
@@ -19,6 +19,3 @@
     - save_process
-    - plot_process
-    - save_voltages
-    - plot_voltages
-    gradient_function: 2
+    gradient_function: 1
     input_popsize: 1
@@ -47,2 +44,3 @@
     - random
+    initials_nonRandFactor: 2.5
     params:
@@ -56,7 +54,13 @@
         V_th: -10.0
-        g_L: 0.01
+        g_L: 0.02
         t_ref: 20.0
-        tau_m: 20.0
+        tau_m: 10.0
         tau_syn_ex: 10.0
         tau_syn_in: 10.0
+    params_noise:
+        E_L: 0.2
+        V_th: 0.2
+        tau_m: 0.2
+        tau_syn_ex: 0.2
+        tau_syn_in: 0.2
     weights_mean:
@@ -83,3 +87,3 @@
     loglevel: M_ERROR
-    numpy_seed: 85412
+    numpy_seed: 2177
     pattern_separation_time: 100.0
```

Listing 14: Example parameter file for $\rho = 1$ with fixed-pattern noise, compared to Listing 2. The noise is applied to the time constants, the voltage and threshold. For the sweeped training see Fig. 7.23, for the membrane voltage see Fig. 7.24.

```
@@ -9,3 +9,3 @@
     num_snapshots: 200
-    save_every_x_batches: 1
+    save_every_x_batches: 2
     tau_m: 20.0
@@ -19,5 +19,2 @@
     - save_process
-    - plot_process
-    - save_voltages
-    - plot_voltages
     gradient_function: 2
@@ -47,2 +44,3 @@
     - random
+    initials_nonRandFactor: 1.4
     params:
@@ -61,2 +59,8 @@
         tau_syn_in: 10.0
+    params_noise:
+        E_L: 0.15
+        V_th: 0.15
+        tau_m: 0.15
+        tau_syn_ex: 0.15
+        tau_syn_in: 0.15
     weights_mean:
@@ -83,3 +87,3 @@
     loglevel: M_ERROR
-    numpy_seed: 85412
+    numpy_seed: 2177
     pattern_separation_time: 100.0
```

Listing 15: Example parameter file for $\rho = 2$ with parameter noise, compared to Listing 2. The noise is applied to the time constants, the voltage and threshold. For the sweeped training see Fig. 7.23.

```
@@ -5,3 +5,3 @@
     V_rest: -20.0
-    batch_number: 8
+    batch_number: 25
     cm: 0.2
@@ -9,3 +9,3 @@
     num_snapshots: 200
-    save_every_x_batches: 1
+    save_every_x_batches: 100
     tau_m: 20.0
@@ -16,3 +16,3 @@
  config:
-    data: patterns_bal
+    data: mnist_small
     eval_level:
@@ -20,4 +20,2 @@
     - plot_process
-    - save_voltages
-    - plot_voltages
     gradient_function: 2
@@ -25,3 +23,3 @@
     network_layout:
-    - 10
+    - 15
     network_layout_depth: 1
@@ -36,2 +34,3 @@
     truncated_spread: 0.5
+    trainingOrTesting: training
@@ -41,3 +40,3 @@
     noisy_samples: true
-    num_samples: 50
+    num_samples: 0
     trainingOrTesting: training
@@ -47,2 +46,3 @@
     - random
+    initials_nonRandFactor: 1.4
     params:
@@ -62,7 +62,7 @@
     weights_mean:
-    - 0.03
-    - 0.175
+    - 0.01
+    - 0.15
     weights_std:
-    - 0.1
-    - 0.1
+    - 0.005
+    - 0.075
  rates:
@@ -72,5 +72,5 @@
     drive_weights_old: 0.0
-    drive_weights_thresh: 0.1
+    drive_weights_thresh: 0.25
     l2_regulizer: 0.0
-    learning: 0.1
+    learning: 0.005
     norm_grad_frob: 0.0
@@ -83,3 +83,3 @@
     loglevel: M_ERROR
-    numpy_seed: 85412
+    numpy_seed: 2177
     pattern_separation_time: 100.0
```

Listing 16: Example parameter file for reduced MNIST training with $\rho = 2$, compared to Listing 2.

```
@@ -10,3 +10,3 @@
     save_every_x_batches: 100
-    tau_m: 20.0
+    tau_m: 10.0
     tau_ref: 20.0
@@ -20,3 +20,5 @@
     - plot_process
-    gradient_function: 2
+    - save_voltages
+    - plot_voltages
+    gradient_function: 1
     input_popsize: 1
@@ -33,3 +35,3 @@
     noisy_samples: true
-    num_samples: 50
+    num_samples: 200
     trainingOrTesting: testing
@@ -46,3 +48,3 @@
     - random
-    initials_nonRandFactor: 1.4
+    initials_nonRandFactor: 2.5
```

```
    params:
@@ -56,5 +58,5 @@
         V_th: -10.0
-        g_L: 0.01
+        g_L: 0.02
         t_ref: 20.0
-        tau_m: 20.0
+        tau_m: 10.0
         tau_syn_ex: 10.0
@@ -62,7 +64,7 @@
     weights_mean:
-    - 0.01
-    - 0.15
+    - 0.007
+    - 0.025
     weights_std:
+    - 0.003
     - 0.005
-    - 0.075
     rates:
```

Listing 17: Example parameter file for reduced MNIST training with $\rho = 1$, compared to Listing 16.

A.3 Parameters for Chapter 8

Possible formulas for the training are the one from [*Mostafa*, 2017], called Mostafa formula, and my equal-time and double-time formulas, derived in Chapter 4.
The WSFs used in this Chapter are not the same ones I presented data for in Chapter 6. This is due to older settings, and because I acquired new data for Chapter 6. For $\rho = 1$, the WSF that was used is 0.00332064 V compared to 0.00281 V in Chapter 6. For $\rho = 2$, the used WSF is 0.00353232 V compared to 0.00197 V earlier.
All emulations are done on the *BrainScaleS* (BSS) system in rack 37, except for Listing 19 which was done on the system in rack 33.
In the experiments, neurons on hardware are large neurons comprising 10 *dendrite membrane*s (DenMems) (Chapter 3).
The experiments in this Chapter use different versions of my code. The used version is given as the git commit SHA. Because experiments use software to access hardware as described in Chapters 3 and 5, the `nmpm_software` module used for executing the code is given as well.

```yaml
cache:
    E_ex_factor: 8.0
    E_in_factor: -8.0
    V_diff: 10.0
    V_rest: -20.0
    batch_number: 8
    cm: 0.2
    digitalise_weights_digits: 3
    num_snapshots: 100
    save_every_x_batches: 10
    tau_m: 10.0
    tau_ref: 20.0
    tau_syn: 10.0
    time_per_step: 100.0
    tmp_factor_for_rounding: 1
config:
    data: patterns_bal
    eval_level:
    - save_process
    - plot_process
    - save_voltages
    - plot_voltages
    gradient_function: 1
    input_popsize: 1
    network_layout:
    - 20
    network_layout_depth: 1
    neurons: iaf_cond_exp
    objective: time_exp
    sim_infinite_tau_m: true
    simulator: pyhmf
eval_data:
    early: 1.5
    late: 20.0
    noisy_samples: true
    num_samples: 50
    truncated_spread: 0.5
input_data:
    early: 1.5
    late: 20.0
    noisy_samples: true
    num_samples: 50
    truncated_spread: 0.5
net:
    initials:
    - rand
    params:
        C_m: 0.2
        E_L: -20.0
        E_ex: 60.0
        E_in: -100.0
        I_e: 0.0
        V_m: -20.0
        V_reset: -20.0
        V_th: -10.0
        g_L: 0.02
        t_ref: 20.0
        tau_m: 10.0
        tau_syn_ex: 10.0
        tau_syn_in: 10.0
    weights_mean:
    - 3.0
    - 3.0
    weights_std:
    - 2.0
    - 2.0
rates:
    absolute_regulizer: 0.0
    drive_weights: 0.0001
    drive_weights_thresh: 0.5
    l1_norm_update: 10.0
    l1_norm_update_final: 0.0
    l2_regulizer: -0.003
    learning: 0.05
    threshold_frob: 0.0
    threshold_frob_final: 0.0
    update_wsc: 0.0
simulation:
    loglevel: M_ERROR
    numpy_seed: 85412
    pattern_separation_time: 100.0
    resolution: 0.01
    resolution_decimals: 2
```

Listing 18: Parameter file with $\rho = 1$ and equal-time formula, see Figs. 8.4 and 8.6.
Code of git commit `8c25fbe0491bb87efcd9393714aa824961c0cd03`
`nmpm_software/2019-01-14-1`

```
@@ -25,3 +25,3 @@
     network_layout:
-    - 20
+    - null
     network_layout_depth: 1
@@ -62,6 +62,4 @@
     - 3.0
-    - 3.0
     weights_std:
-    - 2.0
     - 2.0
 rates:
@@ -69,3 +67,3 @@
     drive_weights: 0.0001
-    drive_weights_thresh: 0.5
+    drive_weights_thresh: 0.35
     l1_norm_update: 10.0
```

Listing 19: Parameter file with $\rho = 1$ and equal-time formula, given as difference to Listing 18.
Results shown in Figs. 8.2 and 8.3. BSS in rack 33 was used here.
Code of git commit `88138ed67b20ced35be7c1984104af87fb6552d6`
`nmpm_software/2019-01-07-1`

```
@@ -10,3 +10,3 @@
     save_every_x_batches: 10
-    tau_m: 10.0
+    tau_m: 20.0
     tau_ref: 20.0
@@ -22,3 +22,3 @@
     - plot_voltages
-    gradient_function: 1
+    gradient_function: 0
     input_popsize: 1
@@ -55,5 +55,5 @@
         V_th: -10.0
-        g_L: 0.02
+        g_L: 0.01
         t_ref: 20.0
-        tau_m: 10.0
+        tau_m: 20.0
         tau_syn_ex: 10.0
@@ -69,3 +69,3 @@
     drive_weights: 0.0001
-    drive_weights_thresh: 0.5
+    drive_weights_thresh: 0.35
     l1_norm_update: 10.0
```

Listing 20: Parameter file with $\rho = 2$ and Mostafa formula, given as difference to Listing 18.
Results shown in Figs. 8.8 and 8.9, and for an average of five consecutive runs in Fig. 8.7.
Code of git commit `9b36f63198f833924ec74cdb797a09a317a3e322`
`nmpm_software/2019-01-14-1`

```
@@ -10,3 +10,3 @@
     save_every_x_batches: 10
-    tau_m: 10.0
+    tau_m: 20.0
     tau_ref: 20.0
@@ -16,3 +16,3 @@
 config:
-    data: patterns_bal
+    data: patterns_bal_inv
     eval_level:
@@ -22,3 +22,3 @@
     - plot_voltages
-    gradient_function: 1
+    gradient_function: 0
     input_popsize: 1
@@ -55,5 +55,5 @@
         V_th: -10.0
-        g_L: 0.02
+        g_L: 0.01
         t_ref: 20.0
-        tau_m: 10.0
+        tau_m: 20.0
         tau_syn_ex: 10.0
@@ -69,3 +69,3 @@
     drive_weights: 0.0001
-    drive_weights_thresh: 0.5
+    drive_weights_thresh: 0.35
     l1_norm_update: 10.0
```

Listing 21: Parameter file with $\rho = 2$ and Mostafa formula, given as difference to Listing 18. Results shown in Figs. 8.10a, 8.10b and 8.11.
Code of git commit `28bac7762106ad7b5229581f371a4ea3dca0294c` `nmpm_software/2019-01-24-1`

```
@@ -10,3 +10,3 @@
     save_every_x_batches: 10
-    tau_m: 10.0
+    tau_m: 20.0
     tau_ref: 20.0
@@ -16,3 +16,3 @@
 config:
-    data: patterns_bal
+    data: mnist_smallest
     eval_level:
@@ -22,3 +22,3 @@
     - plot_voltages
-    gradient_function: 1
+    gradient_function: 0
     input_popsize: 1
@@ -36,2 +36,3 @@
     num_samples: 50
+    trainingOrTesting: testing
     truncated_spread: 0.5
@@ -42,2 +43,3 @@
     num_samples: 50
+    trainingOrTesting: training
     truncated_spread: 0.5
@@ -55,5 +57,5 @@
         V_th: -10.0
-        g_L: 0.02
+        g_L: 0.01
         t_ref: 20.0
-        tau_m: 10.0
+        tau_m: 20.0
         tau_syn_ex: 10.0
@@ -69,9 +71,11 @@
     drive_weights: 0.0001
-    drive_weights_thresh: 0.5
-    l1_norm_update: 10.0
-    l1_norm_update_final: 0.0
+    drive_weights_factor: 1.5
+    drive_weights_thresh: 0.35
     l2_regulizer: -0.003
     learning: 0.05
-    threshold_frob: 0.0
-    threshold_frob_final: 0.0
+    norm_grad_frob: 0.0
+    norm_grad_l1: 0.0
+    norm_grad_l2_colwise: 0.0
+    norm_grad_lsup: 0.0
+    norm_update_l1: 10.0
     update_wsc: 0.0
```

Listing 22: Parameter file with $\rho = 2$ and Mostafa formula, given as difference to Listing 18. Results shown in Figs. 8.12 to 8.14.
Code of git commit `127f0d9460733debb489ea29dbbdfa5c645b6496` `nmpm_software/2019-01-24-1`

List of Figures

Bibliography

Boell, R., Visualization of mapping and routing of the brainscales system, Bachelorarbeit, Universität Heidelberg, 2018.

Bohte, S. M., J. N. Kok, and H. La Poutré, Error-backpropagation in temporally encoded networks of spiking neurons, *Neurocomputing*, *48*(1-4), 17–37, doi:10.1016/S0925-2312(01)00658-0, 2002.

Breitwieser, O., Towards a neuromorphic implementation of spike-based expectation maximization, Master thesis, Ruprecht-Karls-Universität Heidelberg, 2015.

Brette, R., and W. Gerstner, Adaptive Exponential Integrate-and-Fire Model as an Effective Description of Neuronal Activity, *Journal of Neurophysiology*, *94*(5), 3637–3642, doi:10.1152/jn.00686.2005, 2005.

Changeset 3720, on Gerrit, [Online; accessed: 2019-03-16], `https://gerrit.bioai.eu/c/marocco/+/3720`.

Cireşan, D. C., U. Meier, L. M. Gambardella, and J. Schmidhuber, Deep, big, simple neural nets for handwritten digit recognition, *Neural Computation*, *22*(12), 3207–3220, doi:10.1162/NECO_a_00052, 2010.

Corless, R. M., G. H. Gonnet, D. E. G. Hare, D. J. Jeffrey, and D. E. Knuth, On the LambertW function, *Advances in Computational Mathematics*, doi:10.1007/BF02124750, 1996.

Davison, A. P., D. Brüderle, J. Eppler, J. Kremkow, E. Muller, D. Pecevski, L. Perrinet, and P. Yger, PyNN: a common interface for neuronal network simulators, *Front. Neuroinform.*, *2*(11), 2008.

Dayan, P., and L. F. Abbott, *Theoretical Neuroscience: Computational and Mathematical Modeling of Neural Systems*, The MIT press, Cambride, Massachusetts, 2001.

Diesmann, M., and M.-O. Gewaltig, NEST: An environment for neural systems simulations, in *Forschung und wisschenschaftliches Rechnen, Beiträge zum Heinz-Billing-Preis 2001*, *GWDG-Bericht*, vol. 58, edited by T. Plesser and V. Macho, pp. 43–70, Ges. für Wiss. Datenverarbeitung, Göttingen, 2002.

Dold, D., et al., Stochastic computation on spiking neuromorphic hardware, in *Bernstein Conference 2017*, 10.12751/nncn.bc2017.0075, 2017.

FACETS, Research project, `http://http://facets.kip.uni-heidelberg.de/`, 2010.

Gerstner, W., and R. Brette, Adaptive exponential integrate-and-fire model, *Scholarpedia*, *4*(6), 8427, doi:10.4249/scholarpedia.8427, 2009.

Gerstner, W., and W. Kistler, *Spiking Neuron Models: Single Neurons, Populations, Plasticity*, Cambridge University Press, 2002.

Gütig, R., and H. Sompolinsky, The tempotron: a neuron that learns spike timing–based decisions, *Nature Neuroscience*, *9*(3), 420–428, doi:10.1038/nn1643, 2006.

HBP SP9 partners, *Neuromorphic Platform Specification*, Human Brain Project, 2014.

Human Brain Project, Research project, `https://www.humanbrainproject.eu/`, 2013.

Indiveri, G., et al., Neuromorphic silicon neuron circuits, *Frontiers in Neuroscience*, *5*(0), doi:10.3389/fnins.2011.00073, 2011.

Jeltsch, S., A scalable workflow for a configurable neuromorphic platform, Ph.D. thesis, Universität Heidelberg, 2014.

Klähn, J., Training functional networks on large-scale neuromorphic hardware, Master, Universität Heidelberg, 2017.

Koke, C., Device Variability in Synapses of Neuromorphic Circuits, Ph.D. thesis, doi:10.11588/heidok.00022742, 2017.

Kononov, A., Testing of an analog neuromorphic network chip, Diploma thesis, Ruprecht-Karls-Universität Heidelberg, hD-KIP-11-83, 2011.

Krizhevsky, A., I. Sutskever, and G. E. Hinton, Imagenet classification with deep convolutional neural networks, in *Advances in Neural Information Processing Systems 25*, edited by F. Pereira, C. Burges, L. Bottou, and K. Weinberger, pp. 1097–1105, Curran Associates, Inc., 2012.

Kungl, A., Sampling with leaky integrate-and-fire neurons on the hicannv4 neuromorphic chip, Masterarbeit, Universität Heidelberg, 2016.

LeCun, Y., L. Bottou, Y. Bengio, and P. Haffner, Gradient-based learning applied to document recognition, *Proceedings of the IEEE*, doi:10.1109/5.726791, 1998.

LeCun, Y., Y. Bengio, and G. Hinton, NatureDeepReview, *Nature*, doi:10.1038/nature14539, 2015.

Mead, C. A., Neuromorphic electronic systems, *Proceedings of the IEEE*, *78*, 1629–1636, 1990.

Meehan, P., Analyzing and optimizing the reconfiguration time of the brainscales-1 system by implementing differential configuration, Bachelor thesis, Ruprecht-Karls-Universität Heidelberg, 2019.

Millner, S., Development of a multi-compartment neuron model emulation, Ph.D. thesis, Ruprecht-Karls University Heidelberg, 2012.

Mostafa, H., Supervised Learning Based on Temporal Coding in Spiking Neural Networks, *IEEE Transactions on Neural Networks and Learning Systems*, (Nips), 1–9, doi:10.1109/TNNLS.2017.2726060, 2017.

Müller, E. C., Novel operation modes of accelerated neuromorphic hardware, Ph.D. thesis, Ruprecht-Karls-Universität Heidelberg, hD-KIP 14-98, 2014.

Neftci, E. O., H. Mostafa, and F. Zenke, Surrogate Gradient Learning in Spiking Neural Networks, 2019.

NEST Issue 1087, NEST simulator on github, [Online; accessed: 2019-03-29], `https://github.com/nest/nest-simulator/issues/1087`.

NEST Pull 1112, NEST simulator on github, [Online; accessed: 2019-03-29], `https://github.com/nest/nest-simulator/pull/1112`.

online Guidebook, Electronic Vision(s) on GitHub, [Online; accessed: 2019-03-08], `https://electronicvisions.github.io/hbp-sp9-guidebook/pm/using_pm_newflow.html`.

Pedregosa, F., et al., Scikit-learn: Machine Learning in Python, *Journal of Machine Learning Research*, *12*, 2825–2830, 2011.

Petrovici, M. A., Function vs. substrate: Theory and models for neuromorphic hardware, Ph.D. thesis, 2015.

Petrovici, M. A., et al., Characterization and compensation of network-level anomalies in mixed-signal neuromorphic modeling platforms, *PLOS ONE*, *9*(10), e108,590, 2014.

Petrovici, M. A., et al., Pattern representation and recognition with accelerated analog neuromorphic systems, in *Proceedings - IEEE International Symposium on Circuits and Systems*, doi:10.1109/ISCAS.2017.8050530, 2017.

Rosenblatt, F., The perceptron: a probabilistic model for information storage and organization in the brain, *Psychological Review*, *65*, 386–408, 1958.

Rumelhart, D. E., G. E. Hinton, and W. R.J., Learning internal representations by error propagation, *Parallel Distributed Processing: Explorations in the Microstructures of Cognition*, *I*, 318–362, 1986.

Schemmel, J., D. Brüderle, A. Grübl, M. Hock, K. Meier, and S. Millner, A wafer-scale neuromorphic hardware system for large-scale neural modeling, in *Proceedings of the 2010 IEEE International Symposium on Circuits and Systems (ISCAS)*, pp. 1947–1950, 2010.

Schmitt, S., et al., Neuromorphic Hardware In The Loop: Training a Deep Spiking Network on the BrainScaleS Wafer-Scale System, doi:10.1109/IJCNN.2017.7966125, 2017.

SciPy Documentation, *SciPy: Open source scientific tools for Python*, Jones, Eric and Oliphant, Travis and Peterson, Pearu and Others, [Online; accessed: 2019-02-10], `http://www.scipy.org/`.

YCCP repository, *YAML with pre-Computed Common Parameters*, Breitwieser, Oliver, [Online; accessed: 2019-03-29], `https://github.com/obreitwi/yccp`.

Zenke, F., and S. Ganguli, SuperSpike: Supervised learning in multilayer spiking neural networks, *Neural Computation*, *30*(6), 1514–1541, doi:10.1162/neco_a_01086, 2018.

Zoschke, K., M. Guttler, L. Bottcher, A. Grubl, D. Husmann, J. Schemmel, K. Meier, and O. Ehrmann, Full wafer redistribution and wafer embedding as key technologies for a multi-scale neuromorphic hardware cluster, in *2017 IEEE 19th Electronics Packaging Technology Conference (EPTC)*, pp. 1–8, IEEE, doi:10.1109/EPTC.2017.8277579, 2017.

Acronyms and Technical Terms

ADC . . . . . analog-to-digital converter . . . . . (pp. 14, 32, 33)
AdEx . . . . adaptive exponential integrate-and-fire . . . . . (p. 11)
ANN . . . . . artificial neural network . . . . . (pp. I, 1)
API . . . . . application programming interface . . . . . (pp. 14, 32)
BSS . . . . . BrainScaleS . . . . . (pp. I, 1–3, 9, 11, 12, 14, 15, 32, 34, 37, 43, 44, 87, 90, 96, 97)
CoBa . . . . . conductance-based (pp. 11, 15, 23, 37, 41, 42, 45, 49, 69, 71, 72, 75, 87, 88, 94, 99–101)
CuBa . . . . current-based . . . . . (pp. 6, 15, 16, 23, 24, 37, 41, 51, 69, 71, 87)
DenMem dendrite membrane . . . . . (pp. 12, 96)
FACETS . Fast Analog Computing with Emerging Transient States . . . . . (p. 11)
FG . . . . . floating gate . . . . . (pp. 12, 43–46, 48, 72, 75, 78, 80, 89, 90)
FPGA . . . field-programmable gate array . . . . . (p. 13)
GPU . . . . . graphics processing unit . . . . . (p. 1)
HBP . . . . . Human Brain Project . . . . . (p. 11)
HICANN high input count analog neural network . . . . . (pp. 12–14, 75, 89, 99)
IF . . . . . integrate-and-fire . . . . . (pp. 5–7, 15, 16)
IQR . . . . . interquartile range . . . . . (pp. 45, 47)
L1 . . . . . layer-1 . . . . . (p. 14)
L2 . . . . . layer-2 . . . . . (p. 14)
LIF . . . . . leaky integrate-and-fire . . . . . (pp. 11, 15, 23, 37, 51, 75)
MNIST . . Modified National Institute of Standards and Technology . (pp. 2, 3, 33, 35, 36, 99)
NEST . . . . NEural Simulation Tool . . . . . (pp. 32, 64, 91)
PyNN . . . . python neural networks . . . . . (pp. 14, 32, 43)
RNG . . . . . random number generator . . . . . (pp. 51, 57)
SNN . . . . . spiking neural network . . . . . (pp. I, 1–3, 5, 6)
VLSI . . . . . very-large-scale integration . . . . . (p. 11)
WSC . . . . . weight sum cost . . . . . (pp. 9, 15, 24, 25, 29, 58)
WSF . . . . . weight scale factor . (pp. 24, 25, 37, 38, 41–43, 48–50, 69, 71, 75, 87–89, 94, 96, 100)
YAML . . . YAML Ain't Markup Language . . . . . (p. 32)
YCCP . . . YAML with pre-Computed Common Parameters . . . . . (p. 32)

Statement of Originality (Erklärung):

I certify that this thesis, and the research to which it refers, are the product of my own work. Any ideas or quotations from the work of other people, published or otherwise, are fully acknowledged in accordance with the standard referencing practices of the discipline.

Ich versichere, dass ich diese Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.

Heidelberg, April 1, 2019

_____

(signature)