Department of Physics and Astronomy
University of Heidelberg

Bachelor Thesis in Physics
submitted by

Philipp Spilger

born in Heidelberg (Germany)

2018

Spike-based Expectation Maximization on the HICANN-DLSv2 Neuromorphic Chip

This Bachelor Thesis has been carried out by Philipp Spilger at the
Kirchhoff Institute for Physics in Heidelberg
under the supervision of
Prof. Meier

Abstract Neuromorphic spike-based expectation maximization (NSEM) is an adaption of the unsupervised learning method of spike-based expectation maximization (SEM) tailored to implementation on neuromorphic hardware. A cause layer is able to infer salient features in spike-trains presented by an input layer. The network combines a local STDP learning rule with homeostatically stabilized stochastic neurons. Whereas SEM was introduced with abstract stochastic neurons, NSEM extends the model to include LIF neurons with exponential synapses, exponential STDP curves and stochastic weight updates to overcome a possibly limited weight resolution in hardware. This thesis presents the implementation of NSEM on the hybrid neuromorphic HICANN-DLS version 2 prototype chip developed in the Human Brain Project. Correct functioning of all distinct building blocks of the network is verified individually and in combination. The learning rules are fully implemented on the plasticity processing unit (PPU), a general purpose processor embedded in the chip, using a newly developed fractional number type for stochastic weight updates while keeping encapsulation and parameterization in mind. Implementation of this type is described and its performance evaluated. In addition, improvements to the process of providing parameters to the PPU from the experiment-executing host computer are elaborated.

Zusammenfassung Neuromorphe Spike-basierte Erwartungsmaximierung (NSEM) ist eine speziell auf die Implementierung in neuromorpher Hardware angepasste Variante der unüberwachten Lernmethode Spike-basierte Erwartungsmaximierung (SEM). Das Netzwerk ist in der Lage, in, von einer Eingangsschicht, präsentierten Spikeabfolgen unbeaufsichtigt typische Muster zu erkennen. Dabei wird eine lokale STDP Lernregel mit homöostatisch stabilisierten stochastischen Neuronen kombiniert. Während SEM mit abstrakten stochastischen Neuronen formuliert wurde, erweitert NSEM das Modell auf LIF Neuronen mit exponentiellen Synapsen, exponentiellem STDP und stochastischen Gewichtsanpassungen, um deren mögliche limitierte Auflösung auf Hardware zu überwinden. Diese Arbeit präsentiert die Implementierung von NSEM auf dem zweiten Prototypen der hybriden neuromorphen Plattform HICANN-DLS, die im Rahmen des Human Brain Projects entwickelt wird. Die korrekte Funktionsweise von allen eigenständigen Bestandteilen des Netzwerks wird einzeln und in Kombination überprüft. Die Lernregeln sind vollständig auf dem Plastizitätsprozessor (PPU), einem Mehrzweck-Prozessor eingebettet auf dem Chip, unter Ermöglichung von Parametrisierung und Kapselung, implementiert. Ein neu entwickelter Datentyp zum Speichern von Bruchzahlen wird zur Implementation von stochastischen Gewichtsanpassungen verwendet. Die Implementation ebendiesen Datentyps wird beschrieben und die Leistungsfähigkeit untersucht. Darüber hinaus werden Verbesserungen der Parameterversorgung der PPU vom Experiment-ausführenden Computer detailliert beschrieben.

Contents

1. Introduction

While conventional computers are geared towards fast and correct sequential calculation of prewritten arithmetic operations, they are highly limited in parallel computational tasks. The opposite is observed in spiking neural networks such as the brain, where lots of highly interconnected units (neurons) form a massively parallelized network which is able to learn by altering its connectivity. Neural networks are governed through differential equations describing the time evolution of their neurons' membrane potentials. Although numeric simulation of these differential equations is possible, the computational resources scale with the network size and depending on the network size can lead to a simulation time larger than the equivalent biological time.

Hybrid neuromorphic hardware can solve this problem by instead of numerically calculating the time evolution of neural networks physically emulating neural components in analogue electronic circuitry obeying the same differential equations. This greatly improves scaling of computational power, as the computational resources, i.e. number of emulating units scales linearly with the number of neurons to be emulated. In addition, the emulation can be modified such that the differential equations' characteristic time constants are far shorter than their biological counterparts. This speeds up the time of emulation and thereby allows for rapid experiment throughput and for investigation of experiments otherwise not possible due to too long run-time.

The *High Input Count Analogue Neural Network - Digital Learning System* (HICANN-DLS) (S. Friedmann and J. Schemmel et al. 2017; S. A. Aamir and Y. Stradmann et al. 2018) version 2 prototype, developed in the scope of the Human Brain Project, is a hybrid neuromorphic system with analogue neuron circuits and an embedded general purpose processor designed for implementation of learning, i.e. alteration of network parameter. Featuring spike-timing information measurement for each synapse, it allows for spike-timing dependent plasticity (STDP). The analogue circuits time constants are 1000 times faster than their biological counterparts.

Neuromorphic spike-based expectation maximization (NSEM) (Breitwieser 2015) is an unsupervised learning scheme geared towards implementation on neuromorphic hardware. Emerging from Neural Sampling (Buesing et al. 2011), it assumes the network operates on samples of an underlying probability distribution. An online version of *expectation maxim-*

ization (EM) is performed, which enables the neurons to find hidden causes in their input fields. It relies on local STDP and bias stabilization. A cause layer of stochastic LIF neurons, applying the theory of LIF Sampling (Petrovici et al. 2016), receives structured input from an input layer and is able to classify input pattern. Input pattern can either have fixed labels (such as distinct images) or smoothly transition into each other (images of an arbitrarily rotated bar). Each cause layer neuron will attain its receptive fields towards one distinct label in the first case whereas the whole cause layer will equipartition the smooth input space in the latter case. It describes a network comprised of two plasticity rules and additional static parts, which are to exist side by side.

Each plasticity rule therefore is to be applied only to a subset of all synapses available on the chip, which requires masking of execution of each algorithm. In addition, fulfilling differing timing constraints while preventing interference requires scheduling of algorithm execution on the sequential processor.

The main goal of this thesis is to implement NSEM on the HICANN-DLS version 2 prototype and to improve masking, scheduling and parameter distribution to the embedded processor.

Thesis Outline

This thesis is structured in the following way: Methods used and methods developed during this thesis are presented in Chapter 2. First, the theoretical background is shortly presented. Then, the neuromorphic platform is described to a detail used in the following sections and the NSEM adjustments are listed and their implementations explained. Subsequent, software used, improvements and new developments emerged in the time course of this thesis are explained in detail.

Chapter 3 collects calibration, experiment and PPU software performance results of measurements conducted in this thesis. First, automated calibration of the correlation read-out chain developed, building upon given instructions on how to conduct manual calibration, is shown. In addition, neuron activation calibration and evaluation is displayed in this section.

Afterwards the network implementation is evaluated individually for all parts of the network. Starting with an implementation of the NSEM plasticity rule applied on a single synapse, evaluation is continued with examination of the homeostasis rule's ability to stabilize neuron firing rates. The combination of these two parts is then first used for a single-synapse and afterwards extended to an 5×5-pixel input layer with one homeostatically stabilized cause layer neuron. Next, the winner-take-all network is considered for three neurons and its ability to inhibit a neuron from firing in the refractory period of another neuron's spike is evaluated. All network parts evaluated isolated are finally combined to form a NSEM network of three cause layer neurons and 25 input layer units. This network is examined using a set of three separated input pattern to show that it is capable of classifying randomly presented pattern without supervision.

Attaching, the software developments' resource demands of PPU software developments is investigated both for time as well as memory consumption. Additionally, the repository structure of all experiment software developed and the continuous integration set in place is explained.

All experiments in this thesis are performed with the HICANN-DLS version 2 prototype neuromorphic system. Experiment control and result evaluation is executed on conventional computing devices.

2. Methods

In this chapter, the theoretical network structure is outlined. Furthermore, the DLSv2 neuromorphic platform is described and afterwards, adaptions of the theoretical model necessary for implementation on the neuromorphic hardware are outlined. Afterwards, software parts developed are described in detail.

2.1. Theory

Spike-based expectation maximization (SEM) is an unsupervised method for learning spike-based patterns. The theory of Neural Sampling and Expectation Maximization as well as the combined network structure are shortly explained in the following as basis of the subsequent sections. Each section leads to literature explaining the specific topic in more detail.

2.1.1. Boltzmann machine

A *Boltzmann machine* (BM) (Hinton 2007) is a probabilistic model over binary random variables (RV) $\mathbf{Z} = (Z_i)$, called units. Being binary means, that each RV can be either active ($Z_i = 1$) or inactive ($Z_i = 0$) with an intrinsic bias b_i. Units are pairwise connected via weights W_{ij}, yielding the probability distribution

$$p(\mathbf{z}) = \frac{1}{Z} \cdot \exp\left(-E(\mathbf{z})\right) \tag{2.1}$$

with a normalization constant Z, such that $\sum_{\mathbf{z}} p(\mathbf{z}) = 1$ and the Energy E as

$$E(\mathbf{z}) = -\sum_{i,j} \frac{1}{2} W_{ij} z_i z_j - \sum_k b_k z_k. \tag{2.2}$$

2.1.2. Generative model

A generative model is a probabilistic model with probability distribution $p(\mathbf{y}|\theta)$ quantifying how likely it is to generate an external data sample $\mathbf{y}$ given parameters θ (Bishop 2006, Chapter 8). In the case of a BM, the parameters are the weights W_{ij}. The samples $\mathbf{y}$ are drawn from an unknown probability distribution $p^*(\mathbf{y})$. The goal is to approximate $p^*(\mathbf{y})$ by $p(\mathbf{y}|\theta)$ by finding θ that minimizes the Kulback-Leibner divergence ($\mathrm{D_{KL}}$) between the two distributions:

$$\hat{\theta} = \arg\min_{\theta} \mathrm{D_{KL}}\left(p^*(\mathbf{y})||p(\mathbf{y}|\theta)\right) \tag{2.3}$$

As shown in (Breitwieser 2015, Chapter 2), inserting the definition of the $\mathrm{D_{KL}}$ leads to

$$\hat{\theta} = \arg\max_{\theta} \left\langle \ln p(\mathbf{y}|\theta) \right\rangle_{p^*(\mathbf{y})} \tag{2.4}$$

where $\ln p(\mathbf{y}|\theta)$ is the log-likelihood and $\hat{\theta}$ the maximum likelihood estimate.

2.1.3. Expectation Maximization

Expectation Maximization (EM) is a technique to find a (local) maximum likelihood solution for a probabilistic model with latent variables (Bishop 2006, Chapter 9). Latent variables z_k are not directly observed in the data but are merely part of the model and therefore can only be inferred. An example for a latent variable is the angle of a rotated stick in an image. In order to find the solution, two steps are alternatingly repeated. Given a set of parameters θ^{old}, first evaluate the a-posteriori distribution

$$p(\mathbf{z}|\mathbf{y}, \theta^{\mathrm{old}}), \tag{2.5}$$

called the expectation step (E-step). Then find a new set of parameters θ^{new} by maximizing Equation (2.6), therefore called maximization step (M-step).

$$\theta^{\mathrm{new}} = \arg\max_{\theta} \left\langle \ln p(\mathbf{y}|\mathbf{z}|\theta) \right\rangle_{p^*(\mathbf{y})p(\mathbf{z}|\mathbf{y},\theta^{\mathrm{old}})}. \tag{2.6}$$

The steps are then to be repeated with $p(\mathbf{z}|\mathbf{y}, \theta^{\mathrm{new}})$. For further information see (Bishop 2006, Chapter 9) and (Dempster, Laird and Rubin 1977).

2.1.4. Neural Sampling

A BM with N RVs has 2^N states. Exact inference is therefore impossible for large N and approximation by sampling from the probability distribution is to be employed (Bishop

2006, Chapter 11). In Neural Sampling (Buesing et al. 2011), binary RVs are represented by neurons with the relation that $z_k = 1$ if the corresponding neuron spiked and is in the refractory period and $z_k = 0$ otherwise. If the neuron's membrane potential u_k suffices the *neural computability condition*, that is it encodes the log-odds of the RV z_k being active or inactive given the state of all other RVs $\mathbf{z_k}$, the spiking activity corresponds to samples from the underlying probability distribution $p^*(\mathbf{z})$:

$$u_k(t) = \log \frac{p^*(z = 1|\mathbf{z}_k(t))}{p^*(z = 0|\mathbf{z}_k(t))} \tag{2.7}$$

Therefore, each neuron has a logistic activation function.

2.1.5. LIF Neuron

The membrane potential dynamics of a leaky-integrate and fire (LIF) neuron follow the differential equation depicted in Equation (2.8), adapted from (Gerstner and Kistler 2002, Chapter 4), with the membrane capacitance C_m, the membrane time constant $\tau_m = C_m/g_L$, the membrane potential of neuron k, u_k, the leak conductance g_L, the leak potential E_L and the input current $I(t)$.

$$\begin{aligned} C_m \frac{\mathrm{d}u_k}{\mathrm{d}t} &= -g_L \left(u_k - E_L\right) + I(t) \\ \Leftrightarrow \tau_m \frac{\mathrm{d}u_k}{\mathrm{d}t} &= -\left(u_k - E_L\right) + \frac{I(t)}{g_L} \end{aligned} \tag{2.8}$$

A neuron is set to spike, if the membrane voltage is exceeds the threshold potential, $u_k > V_{\text{thresh}}$ and in that case the membrane voltage resides at V_{reset} for the refractory time τ_{ref}. For current-based synapses, the input current $I(t)$ as described in (Breitwieser 2015, Chapter 2) is composed of

$$I(t) = \sum_i w_{ik} \sum_{t_i^s} \exp\left(-\frac{t - t_i^s}{\tau_{\text{syn}}}\right) \Theta(t - t_i^s) \tag{2.9}$$

with the weight w_{ik} and the spike arrival times t_i^s of the i-th synapse of neuron k, the synaptic time constant τ_{syn} and the Heaviside function Θ. Each pre-synaptic spike adds a linearly weighted exponentially decaying input current.

2.1.6. LIF Sampling

The theory of Neural Sampling can be extended from stochastic neurons to deterministic leaky-integrate and fire (LIF) neurons (Petrovici et al. 2016). A LIF neuron subject to high

excitatory and inhibitory noise features a stochastic spike behavior with a logistic activation function (Petrovici et al. 2016), that is the probability $p(z_k = 1|\mathbf{z}_k)$ of the neuron k to spike relative to its maximally possible activity, is described in (Breitwieser 2015, Chapter 2) as

$$p(z_k = 1|\mathbf{z}_k) = \sigma(\bar{u}_k) = \sigma\left(\frac{\bar{u}_k - \bar{u}_k^0}{\alpha}\right) \tag{2.10}$$

with the sigmoid function σ, the average membrane potential $\bar{u}_k$ of neuron k, the offset voltage $\bar{u}_k^0$, where the activity is $\frac{1}{2}$ and the slope α^{-1}.

2.1.7. Spike-based expectation maximization

Expectation maximization applied to Neural Sampling is called spike-based expectation maximization (SEM). A cause layer consisting of stochastic neurons receives input as Poisson spike-trains from an input layer. Using local *spike timing dependent plasticity* (STDP) and homeostatic excitability modification, it can be shown, that the network performs EM to find hidden causes in the input patterns, see (Habenschuss, Puhr and Maass 2013). Figure 2.1 shows the network topology. The cause layer forms a BM and is composed of

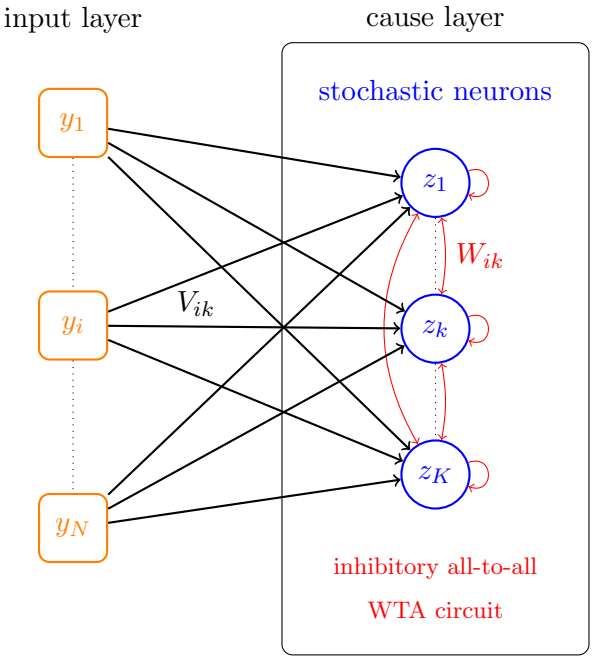


Figure 2.1: The SEM network topology, adapted from (Breitwieser 2015). A cause layer consisting of stochastic neurons z_k receives structured input as Poisson spike-trains from input layer neurons y_i. The two layers are connected via weights V_{ik}. The cause layer forms a winner-take-all circuit by employing strong inhibitory weights W_{kl} between cause layer neurons.

stochastic neurons z_k interconnected via strong inhibitory weights W_{kl} forming a winner-take-all (WTA) circuit, where only one neuron fires at a time. The E-step is described by evaluating the cause layer spikes to a given input, whereas the M-step is implemented as local STDP updates. It can be shown then that the cause layer spikes sample from the posterior distribution $p(\mathbf{z}|\mathbf{y}, \theta)$ with $\theta = \mathbf{b}', \mathbf{W}, \mathbf{V}$ consisting of the biases $\mathbf{b}'$, the WTA

weights $\mathbf{W}$ and the input layer to cause layer weights $\mathbf{V}$. For a proof see (Bill et al. 2015) and for a detailed explanation see (Breitwieser 2015, Chapter 2). The variables y_i describe the eligibility trace of the input neuron i, that is, each time the input neuron fires, y_i is increased by 1 for the time interval τ_{syn} equivalent to a rectangular PSP. (Bill et al. 2015) then show, that the learning rule for V_{ik} results in

$$\frac{\mathrm{d}V_{ik}}{\mathrm{d}t} = \eta \cdot z_k(t) \cdot \left(y_i(t) e^{-(V_{ik}+V_{i0})} - 1 \right) \tag{2.11}$$

with the learning rate η and the default hypothesis $V_{i0} = \log(\lambda_{i0})$ describing the *null cause*, the case if no cause layer neuron is active and all input neurons spike with rate $\nu_{i0} = \lambda_{i0}/\tau_{\text{syn}}$. The effective bias $\mathbf{b}'$ changes on V_{ik} updates. To account for that and for possibly differing overall input pattern strength, homeostasis is applied on the cause layer neurons so that each cause layer neuron's activity is stabilized on a target activity m_k with $\sum_k m_k \leq 1$ (Habenschuss, Puhr and Maass 2013). The update rule for the homeostatic bias is derived in (Bill et al. 2015) to be

$$\frac{\mathrm{d}b_k^{\text{hom}}}{\mathrm{d}t} = \eta_b \left(m_k - \langle z_k \rangle (t) \right). \tag{2.12}$$

2.2. Neuromorphic Hardware

This section describes the neuromorphic platform used for experiment conduction and adaptions of the theoretical model outlined in Section 2.1 for implementation on that platform.

2.2.1. The DLS neuromorphic platform

The HICANN-DLS (S. A. Aamir and Y. Stradmann et al. 2018) (High Input Count Analogue Neural Network - Digital Learning System) version 2 prototype chip (DLSv2) is a hybrid neuromorphic system. The neuron membrane and synapses are emulated as analogue circuits, whereas the system is digitally configurable and spikes are distributed digitally. In addition, the Plasticity Processing Unit (PPU) (Friedmann 2013; S. Friedmann and J. Schemmel et al. 2017), a general-purpose processor featuring the Power instruction set architecture (IBM 2010) is embedded in the chip, enabling learning and alteration of network parameters during run-time of experiments.

In the following, properties of the chip and its surrounding are collected. Further information can be found in (S. A. Aamir and Y. Stradmann et al. 2018; S. Friedmann and J. Schemmel et al. 2017).

The chip (and digital-to-analogue converters on the base board) can be configured from a host computer via an Field-Programmable Gate Array (FPGA).

The prototype features 32 leaky integrate and fire (LIF) neurons with 32 current based synapses each. The synapses have a digital strength resolution of 6 bit. Synapses are driven by one synapse driver for each synapse row. Synapses can synapse driver wise be set to be either excitatory or inhibitory. Figure 2.2 shows the logical layout of the synapses, neurons and synapse drivers. Pre-synaptic spike events can be sent to a subset of synapse drivers. Each spike carries a 6 bit label as does each synapse. Both are compared and a weighted synaptic current pulse is generated if they match. This allows for arbitrary spike routing to neurons. Pre-synaptic spike events can either be externally emitted or routed from neurons by the FPGA to a subset of synapse drivers with an adjustable label. The latter allows for recurrent networks.

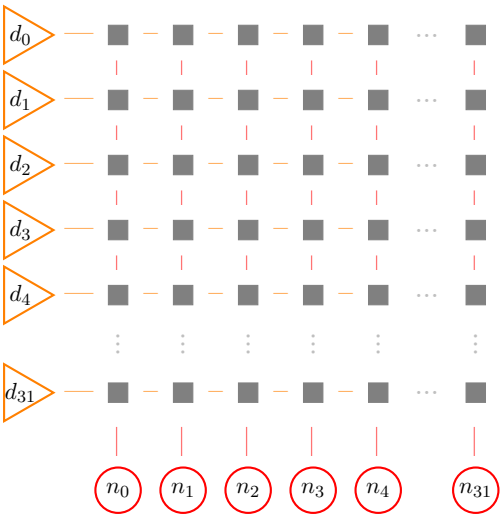


Figure 2.2: Schematic logical layout of the synapses as gray rectangles, the neurons n_k in red below and the synapse drivers d_i in orange on the left of the synapses. On DLSv2 each neuron has 32 synapses in its column. Each synapse row is driven by a synapse driver which can be set to be either excitatory or inhibitory for the whole row. A synapse carries a 6 bit weight w_{ik} and a 6 bit label used for spike routing.

Each synapse has a causal and an anti-causal correlation measure. The eligibility trace measurement is digitized by a ramp-compare 8 bit analogue-to-digital converter (ADC), the CADC (Correlation ADC). The CADC has 32 causal and 32 anti-causal channels for parallel readout. Synapse rows are then multiplexed onto these channels. The causal and anti-causal channels can be read-out simultaneously per row. The analogue eligibility trace circuit features an accumulated nearest neighbor pair-based measurement. For each pre- (causal) or post-synaptic (anti-causal), i.e. neuron spike, an exponential decay with time constant $\tau_{\text{cor}}^{\{c,a\}}$ is started and read-out on the next post- (causal) or pre-synaptic (anti causal) spike. Each read-out value is then added to the accumulated eligibility trace capacitor.

Every neuron has a digital 10 bit output rate counter, which is optionally automatically reset-able on reads and counts the neuron's spikes since the last reset.

For experiment conduction, playback programs for the FPGA are constructed in a host program. The playback program is then transferred to and executed from the FPGA. It consists of a sequence of instructions and can be constructed from either wait instructions for temporal structuring of the experiment, spike fire events or reading or writing configuration containers created using the hardware abstraction layer software framework `haldls` (Electronic Vision(s) 2018). The `haldls` containers abstract chip and FPGA configuration. To enable for example a neuron's digital spike output, a write instruction for the `NeuronDigitalConfig` is to be added to the playback program with the option `enable_digital_out` set to `true`.

The PPU can access the whole chip configuration and features a weakly coupled 16×8 bit and 8×16 bit vector unit with direct access especially to synaptic weights and correlation measurements row-wise for the first and second 16 synapses. The vector unit can perform (unsigned) integer and saturating fractional arithmetic operations.

Programs for the PPU are written in assembler, the `C` or `C++` programming languages. They are compiled and linked using a modified version of the *gnu compiler collection* (*gcc*) and the binary utilities collection *binutils* with support for the custom vector unit. For more information on the tool chain, see (Heimbrecht 2017) and (Spilger 2018). The processor is equipped with 16 kB of memory, which can be read from and written to from the host computer via the FPGA. Precompiled PPU programs can thereby be loaded, executed and altered during playback program execution.

2.2.2. Neuromorphic SEM adjustments

The network setup depicted in Figure 2.1 is not directly transferable to the neuromorphic system, since it implements key elements differently, e.g. the eligibility trace measurement or the neuron model. Therefore the network is to be adjusted for hardware implementation in order to account for those differences. In the following sections, the necessary adaptions formulated in (Breitwieser 2015) are displayed.

2.2.2.1. Neurons

The neuron circuits on DLSv2 model LIF neurons. LIF neurons can be adjusted to behave stochastically with a logistic activation function, see Section 2.1.6. Each neuron is therefore connected to a strong inhibitory and an excitatory Poisson background source via two synapses and the reset potential is set near the threshold potential in order for the noise to introduce the maximal amount of stochasticity (Breitwieser 2015).

2.2.2.2. Synaptic weight translation

The theory assumes rectangular post-synaptic potentials (PSPs), whereas the emulation features exponentially decaying PSPs. In order to translate the hardware weights w_{ik} back to theoretical weights W_{ik}, the area under the PSP is demanded to be equal to a rectangular PSP with duration τ_{on} for the duration τ_{on} (Breitwieser 2015). The conversion then follows Equation (2.13) on insertion of Equations (2.8) and (2.9) and is adapted from the derivation for conductance based synapses presented in (Breitwieser 2015).

$$\begin{aligned} W_{ik}\tau_{\text{on}} &\stackrel{!}{=} \frac{1}{\alpha}\int_0^{\tau_{\text{on}}} u_{\text{PSP}}(t) \\ \text{(2.9), (2.8)} \Rightarrow W_{ik}\tau_{\text{on}} &= w_{ik}\frac{1}{\alpha g_L}\tau_{\text{syn}}\left(1 - e^{-\frac{\tau_{\text{on}}}{\tau_{\text{syn}}}}\right) \\ \tau_{\text{on}} = \tau_{\text{syn}} \Rightarrow W_{ik} &= w_{ik}\frac{1}{\alpha g_L}\left(1 - e^{-1}\right) =: w_{ik}\frac{1}{f_{\text{theo}\to\text{bio}}} \end{aligned} \tag{2.13}$$

$f_{\text{theo}\to\text{bio}}$ describes the constant conversion factor from theoretical weights to biological or hardware weights. The scale factor α is the inverse slope of the activation function as outlined in Equation (2.10).

2.2.2.3. Homeostasis

Instead of directly adjusting the bias from Equation (2.12) by adjusting the leak potential, (Breitwieser 2015) show a conversion to a homeostasis with background sources, enabling activity adjustment through employing current onto the membrane. The weight update rule for the neuron k for each spike then follows as derived in (Breitwieser 2015) to be

$$\Delta w_k^{\text{hom}} = \eta_b \begin{cases} m_k \cdot \frac{\nu^{\text{net}}}{\nu^{\text{pre}}}, & \text{background source spike} \\ -1, & \text{cause layer neuron spike} \end{cases}. \tag{2.14}$$

with the learning rate η_b, the synaptic weight w_k^{hom}, average rate $\nu_k^{\text{target}} = m_k\frac{\nu^{\text{net}}}{\nu^{\text{pre}}}$ the neuron is stabilized to, the background source mean rate ν^{pre} and the fraction m_k of the k-th neuron of the total network rate ν^{net}.

2.2.2.4. Synaptic inter-layer update rule

The synaptic SEM weight update rule, see Equation (2.11) is to be adjusted to fit the observables present on the hardware. The presented adaptions are adopted from (Breitwieser 2015).

Homeostatically controlled neurons First, due to the multiplicative binary variable $z_k(t)$, the original learning rule updates the weights only every cause layer spike. Since homeostasis is employed on the cause layer neurons, the average activation $\langle z_k \rangle = m_k$ is restricted to be constant leading to a simplification of Equation (2.11):

$$\frac{\mathrm{d}V_{ik}}{\mathrm{d}t} = \eta \cdot \left(z_k(t) y_i(t) e^{-(V_{ik}+V_{i0})} - m_k \nu^{\mathrm{net}} \right) \tag{2.15}$$

For a derivation see (Breitwieser 2015).

Pair-based nearest-neighbor correlation measurement The theoretical model assumes box shaped correlation signals for each pre-synaptic spike and no dependence of the post-synaptic activity. On the DLSv2 however, each pre-synaptic spike starts an exponential decay which is stopped on occurrence of a post-synaptic spike and the remaining amplitude is added to the causal correlation measurement. Therefore $y_i(t)$ and $z_k(t)$ are not independent. In addition, instead of adding an exponential decay for every pre-spike, the decay is reset on the occurrence of a post-synaptic spike. This leads to a systematic drift of the measured correlation to lower values as expected with indefinite decay. (Breitwieser 2015) shows, that the average nearest-neighbor eligibility trace corresponds to the theoretical eligibility trace via

$$\frac{\langle y_i \rangle_k^{nn}}{\langle y_i \rangle_k} = \frac{1}{1 + \tau_{\mathrm{syn}} \nu_{ik}} \left[1 - \exp\left\{ - \left(1 + \frac{1}{\tau_{\mathrm{syn}} \nu_{ik}} \right) \nu_{ik} T_{ISI} \right\} \right] \tag{2.16}$$

with the average inter-spike interval (ISI) of the active cause layer neuron T_{ISI}, the actual pre-synaptic spike rate ν_{ik} and the synaptic time constant of the exponential eligibility trace decay τ_{syn}. Using Equation (2.11), this leads to a nearest-neighbor null cause hypothesis rate derived in (Breitwieser 2015) of

$$\lambda_{i0}^{nn} = \frac{1}{1 + \frac{1}{\tau_{\mathrm{syn}} \nu_{ik}}} \left[1 - \exp\left\{ - \left(1 + \frac{1}{\tau_{\mathrm{syn}} \nu_{ik}} \right) \nu_{ik} T_{ISI} \right\} \right] \tag{2.17}$$

Analogously finding λ_{ik}^{nn} leads to a weight to an inferred rate conversion of

$$V_{ik} = \log \frac{\lambda_{ik}^{nn}(\nu_{ik})}{\lambda_{i0}^{nn}} \tag{2.18}$$

The adjusted update rule adopted from (Breitwieser 2015) is displayed in Equation (2.19).

$$\frac{\mathrm{d}V_{ik}}{\mathrm{d}t} = \eta \cdot \left((z_k y_i)(t) \frac{1}{\lambda_{i0}^{nn}} e^{-V_{ik}} - m_k \nu^{\mathrm{net}} \right) \tag{2.19}$$

Accumulated update A weight update for every cause layer spike is not possible, because the (mean) inter-spike interval is lower than the minimal duration of a weight update. Moreover in the case of the homeostasis Equation (2.14), there's no hardware event accessible for the PPU for pre-synaptic spikes disallowing updates at each spike event of the background source.

Therefore, the weights are updated for an accumulation of spikes in a defined update interval. In order to precalculate the target rate, updates with constant update period T_{update} are chosen, leading to the accumulated weight update rule given in Equation (2.20).

$$\begin{aligned} \Delta w_{k,\text{acc}}^{\text{hom}} &= \eta_b \left(m_k \frac{\nu^{\text{net}}}{\nu^{\text{pre}}} T_{\text{update}} \nu^{\text{pre}} - n_k^{\text{post}} \right) \\ \Leftrightarrow \Delta w_{k,\text{acc}}^{\text{hom}} &= \eta_b \left(m_k \nu^{\text{net}} T_{\text{update}} - n_k^{\text{post}} \right) \end{aligned} \tag{2.20}$$

The number of cause layer spikes during the last period is denoted as n_k^{post}. The rate counter of the neurons is used in the implementation, developed as part of this thesis, to count the number of cause layer neuron spikes individually for each neuron. The target average number of spikes $n_k^{\text{target}} = m_k \nu^{\text{net}} T_{\text{update}}$ can be precalculated as it stays constant during an experiment, leaving a rate counter lookup, a subtraction and a multiplication to be performed for each weight update. Similarly, the SEM learning rule is adjusted to be performed periodically with an update period of τ_{update} leading to Equation (2.21), derived in (Breitwieser 2015).

$$\Delta V_{ik} = \eta \cdot \left(\sum_l (z_k y_i)_l \frac{1}{\lambda_{i0}^{nn}} e^{-V_{ik}} - \tau_{\text{update}} m_k \nu^{\text{net}} \right) \tag{2.21}$$

The accumulated correlation, $\sum_l (z_k y_i)_l$, corresponds to a readout and reset of the accumulating causal hardware measurement. Dynamically changing variables in this equation are the correlation readout and the weight. All other values stay constant and are precomputed as a constant in the case of the regulatory ($\eta \tau_{\text{update}} m_k \nu^{\text{net}}$)-term and as a lookup table for the 64 hardware weights in case of the exponential term, as proposed in (Breitwieser 2015). Each update therefore consists of readout of correlation and weight, weight lookup for the exponential term, multiplication with the correlation value and subtraction of the regulatory constant (Breitwieser 2015).

Stochastic weight updates The discretized synaptic hardware weights are mapped to theoretical weights via a constant factor w_{step}:

$$V_{ik} = w_{\text{step}} w_{ik} \tag{2.22}$$

The weight updates Δw_{ik} and Δw_k^{hom} will mostly be a fraction of a hardware weight unit, since the weight updates converge towards target weights. These small weight updates will be performed as described in (Breitwieser 2015) by stochastically deciding to update a hardware weight step depending on the fractional part of the weight update. This leads to, on average, first order correct mean weights and especially correct learning rates as opposed to choosing to round up all weight updates smaller than a hardware weight unit.

2.2.3. Neuromorphic SEM network implementation

Applying the adjustments from (Breitwieser 2015), outlined in the previous sections, the theoretical network topology, depicted in Figure 2.1, is to be adapted. Figure 2.3 shows the adjusted network topology as is implemented on the DLSv2 neuromorphic hardware as part of this thesis. The adjusted network is called neuromorphic SEM (NSEM) in the following and first introduced in (Breitwieser 2015). The neuromorphic SEM learning rule refers to the learning rule for input- to cause-layer neurons, i.e. Equation (2.21).

In order to be able to lessen and increase the neuron activity, a homeostasis background source is connected to an excitatory and an inhibitory synapse of each neuron, as the hardware only allows for one type at a time for each synapse row. As alternative implementation, a homeostasis altering the leak potential directly additionally is developed. This implementation however leads to unresolved hang-ups of chip or FPGA and is therefore not further evaluated in the following.

The homeostatic and NSEM synaptic connection strength updates are fully performed on the PPU. Opposed to the FPGA playback program, it allows for online learning including decisions based on dynamic observables.

Input pattern and background spike sources are pregenerated and played back in the FPGA playback program.

The WTA network behavior is implemented using the spike routing capability of the FPGA. The cause layer neurons are all-to-all inhibitory connected strong enough so that only one neuron is active at a time. This provides all neurons with the same probability to spike again after the last active neuron's refractory period.

Static configuration of the network topology, the spike-router's routing as well as static parameter calibration data such as neuron parameters are set via the FPGA in advance to learning and image pattern presentation.

Stochastic weight updates, used in the homeostasis and the NSEM update rule implementation, need data types capable of storing fractional values and computing arithmetic operations efficiently. The implementation developed is described in detail in Section 2.3.2.

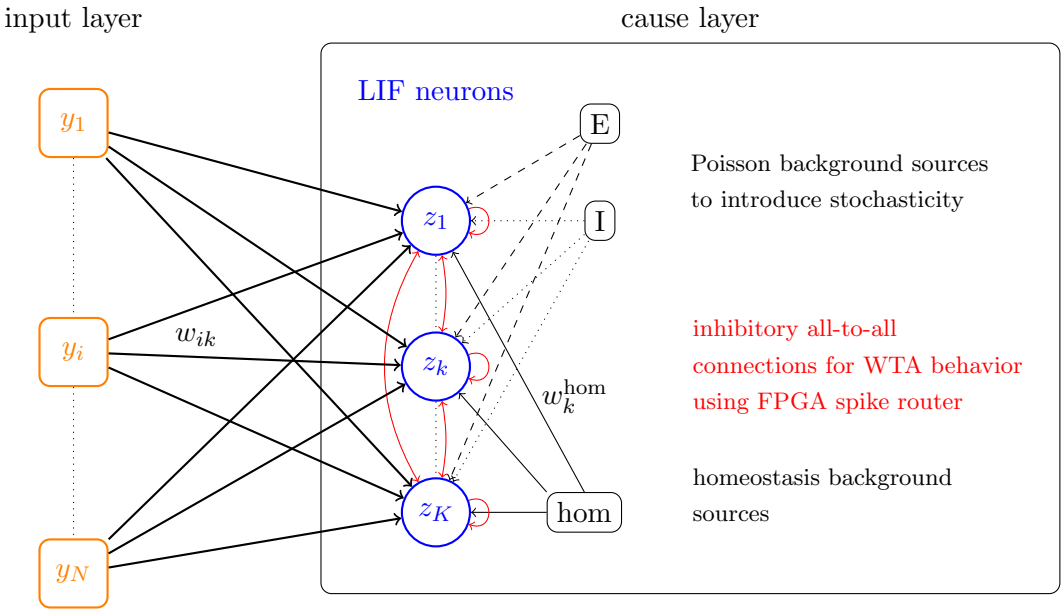


Figure 2.3.: Neuromorphic SEM network topology adjusted from the topology in Figure 2.1 for implementation on the DLSv2 hardware. The input pattern y_i is presented by Poisson spike trains to cause layer neurons z_k with weights w_{ik}. The cause layer LIF neurons are brought into stochastic regime by excitatory and inhibitory background spike trains. The effective bias of the cause layer neurons is adjusted by homeostasis through adjusting weights w_k^{hom} of an excitatory and an inhibitory connection to background sources. The cause layer implements a WTA behavior by strong inhibitory all-to-all spike routing.

2.2.3.1. Time convention

The DLSv2 neuromorphic platform features time constants such as the neuron refractory time constant in the order of 1 µs to 30 µs hardware time. This is in the order of $O(1 \times 10^3)$ faster than the biological domain. The digital clock frequency is 97.5 MHz for the setup used. Therefore, a speedup factor of 975 is defined to convert the time constants to a biologically comparable range. A FPGA or PPU digital cycle thereby corresponds to 10 µs biological time. In the following, most times are presented in biological time using the speedup factor in the following and are denoted by $_{\text{bio}}$.

2.3. Software and Experiment Control

This section describes software developed during this thesis and used in experiment conduction. Host software is developed in the framework of the hardware abstraction layer `haldls` and the hardware coordinates `halco`. PPU software is developed using `libnux`, an auxiliary library supplying helper functions and hardware access information to e.g. read and write weights.

First, communication of learning rule parameters from the host to the PPU is described. Then the implementation of a new datatype for storage of and computation with fractional numbers on the PPU is explained. In addition, scheduling of tasks on the PPU and masking of plasticity rules is outlined. Finally, an encapsulation of experiments in the host software, extensions to ease processing of multiple spike trains and serialization of hardware configuration are presented.

2.3.1. Host-PPU communication

The NSEM learning rule and the homeostasis rule are implemented as algorithms in PPU executables. The program is precompiled and loaded into PPU RAM (random access memory) via the FPGA. In order to prevent the need to recompile the PPU program at every change in rule parameters, e.g. the update rate, the parameters are to be supplied to a precompiled program. To achieve that, means to simplify Host-PPU communication are developed as part of this thesis.

The parameters are supplied to the PPU program during run-time via write access of PPU memory by the FPGA. The PPU binaries are linked to be in the `ELF` *(Executable* and *Linkable Format)* binary file format. This format, in addition to the executable code, stores information about position and size of global symbols, e.g. global variables or structures enabling direct access to the objects. As described in more detail in (Spilger 2018), accessing the positional information in the `ELF` file allows for accessing objects in PPU programs by name. In addition, converter functions from basic types to 32 bit word based `PPUMemoryBlock` intervals are implemented allowing for direct external access to (unsigned) integer, floating point and vector variables in running PPU programs.

The PPU program is set to wait on parameters to be supplied by the FPGA via write locks implemented as infinite loop `while(!signal) {}` to be exited on a change of the global variable `signal`, that is set to be 0 at program start.

As the PPU memory is 16 kB in size, precomputing parts of update rule parameters to be used in the update loop will not be possible in the same program as the actual experiment

conduction because of memory limits. Because the learning rule's parameters are real numbers, this problem is enlarged by the need to use floating point parameter variables, as the PPU does not feature hardware floating point support. Thereby, every floating point operation is computed in software as opposed to call a dedicated instruction, which leads to a large fraction (about 4 kB for all additive and multiplicative operations) of the totally available memory consumed by software implementations of these operations. The pregeneration of update rule internal parameters is therefore split out of the experiment conduction program. The parameters are then directly inserted in the update rule's symbol area encoded in its internally used format (e.g. unsigned integer vector for the homeostasis' learning rate) at the memory location, the linker decided to place the variable to.

To precompute the parameters, two approaches are possible. Either, the parameters are precomputed on the host or the parameters are precomputed in a different PPU program by supplying the rule-external parameters as e.g. floating point numbers and reading back the internal representation to the Host after calculation on the PPU. While the latter demands hardware usage and additional code to build and execute a parameter-generation-only playback program, the rule-internal representation is obtained without additional investment of resources on understanding the structure internal placement of the compiler. Therefore, the parameter generation is split out into another PPU program to be executed and evaluated before the actual experiment, in which the generated parameters are inserted. The experiment program then is free of parameter translation functionality, which leads to small enough programs to be executed.

Care needs to be taken about which parts of parameter structures to insert into the experiment program. Especially values of variables such as pointers linking to other data structures will (need to) differ from the parameter generation program to the experiment conduction program. The reason for this is, that executable code portions are placed differently by the linker, if code is added or removed. Replacing pointer values might therefore lead to malfunctioning code. An example of malfunctioning replacement would be changing a pointer to a random seed, now referring to a zeroed portion of the memory, leaving the corresponding random number generator to only produce zero as random number.

The structure-internal placement of variables to be excluded from insertion is to be found out manually as there is currently no full understanding of the placement rule applied by the compiler available. The approach used for the homeostasis and NSEM learning rule structure is printing the structures memory area and linking portions to variables based on known size and value, given they are beforehand set to distinguishable values.

2.3.2. Fractional arithmetics with stochastic evaluation

The weight update rules in Equations (2.20) and (2.21) lead to sub-hardware-resolution weight updates as mentioned in Sections 2.2.2.3 and 2.2.2.4. Therefore, an efficient sub-integer representation and back conversion to integer values is developed.

The correlation measurements and weights are accessed from the PPU via its vector unit, yielding 16×8 bit vectors. Each vector entry corresponds to a weight or correlation value.

The vector unit of the PPU supplies builtin fractional arithmetics for 8 bit and 16 bit wide values. These arithmetics however perform lossy arithmetic operations and don't provide means to adjust the resolution on demand. To overcome these limitations, a digit-based fixed point fractional representation is developed. A digit-based fractional number is composed of integer digits and fractional (i.e. sub-integer) digits, just as in the decimal system, see Equation (2.23).

$$27.123 = \underbrace{27}_{\text{two integer digits}} \; . \; \underbrace{123}_{\text{three fractional digits}} \tag{2.23}$$

Instead of the decimal base 10, the base 128 is chosen for the implementation. It can be represented in an `uint8_t` byte and at the expense of loosing only one bit, carryover detection for arithmetic operations is possible due to the unused upper bit.

The choice of the `uint8_t` type allows implementation using the vector unit's 16×8 bit unsigned integer vectors, which parallelizes arithmetic operations and speeds up computation. In the following, a fractional vector denotes the implementation storing 16 fractional numbers using a 128 base.

Analogously to decimal fractional numbers, the fractional vector variables can be created with an arbitrary number of digits. This allows for arbitrary precision to be adjusted in 7 bit steps. As the fractional vector is used in the homeostasis and NSEM learning rule as sub-integer extension for 8 bit integer values, the implementation is restricted to allow only one integer digit. This way, the fractional vector can be seen as an integer vector with additional arbitrary sub-integer precision.

Using vectors as base digit type enables parallelized fractional computation. Each array of 16 digits is represented by a 16×8 bit vector. To allow for carryover detection, this limits the digit base to 7 bit for fractional digits.

Figure 2.4 shows the fractional representation of numbers with three fractional and one integer digit.

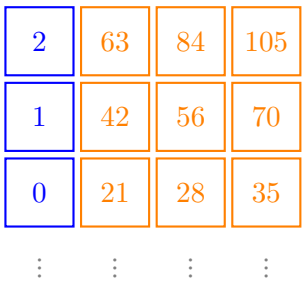


Figure 2.4: The representation of fractional numbers using three fractional digits, depicted in orange and one integer digit, depicted in blue. Each row represents a fractional number. There are 16 numbers stored in parallel using the builtin `vector` type to speed up arithmetic operations.

2.3.2.1. `vector_fractional`-API

The application programming interface (API) of this new `vector_fractional` type is displayed in Table 2.1. The `C++` operators are used so that the type behaves like an internal type from the user's perspective. In addition to addition and subtraction operators, a lossless multiplication with an integer vector is implemented, which divides the result by 128 and adds a sub-integer digit.

2.3.2.2. Stochastic down-conversion

Since the hardware weights can nonetheless only be set to 6 bit integer values, a down-conversion is to be performed. The fractional representation could be converted to integer by discarding the fractional part. This however leads to resolution loss and especially on small weight updates wrong learning, i.e. the weight would be stuck at a value, if all weight updates are smaller than 1, as discussed in Section 2.2.2.4. (Breitwieser 2015) propose stochastically evaluating the fractional part of a weight update, as this leads to on average correct learning rates and weight updates. This stochastic conversion to integer is implemented for the `vector_fractional` type as a free function, as a random number generator instance reference is to be supplied and would lead to an additional 16 byte memory consumption of the type because of structure-internal alignment. The integer value is computed by adding to each entry's integer part a 1 with probability of the fractional part of the entry. For parallel computation, an efficient random number generator using the `xorshift128` algorithm (xorshift implementation for 128 bit wide values) is implemented with a down-conversion to 7 bit per entry.

2.3.2.3. Exponential precalculation

As described in Section 2.2.2.4 and proposed by (Breitwieser 2015), the exponential part of equation (2.21) should be precomputed for all 64 possible weight values in order to speed up the weight update computation.

member function	specification
`vector_fractional()`	Default constructor, no zero initialization for increased creation speed
`void set_entry(size_t entry, float)`	Set entry in vector to float number, automatically convert to fractional representation
`vector uint8_t get_digit_integer()`	Get integer digits of all entries
`vector uint8_t get_digit_fractional(size_t pos)`	Get fractional digits at `pos` of all entries
`void set_digit_integer(vector uint8_t)`	Set integer digits of all entries
`void set_digit_fractional(size_t pos, vector uint8_t)`	Set fractional digits at `pos` of all entries
`vector_fractional<p> operator+(vector_fractional)`	Addition operator
`vector_fractional<p> operator-(vector_fractional)`	Subtraction operator
`vector_fractional<p>& operator+=(vector_fractional)`	Addition to operator
`vector_fractional<p>& operator-=(vector_fractional)`	Subtraction to operator
`vector_fractional<p> operator-()`	Sign change operator
`vector_fractional<p+1> operator*(vector uint8_t)`	Multiplication with integer vector operator, operation divides through 128 and increases precision by one digit to counteract resolution loss
`operator vector uint8_t ()`	Convert to integer vector by discarding fractional part, i.e. rounding down

Table 2.1.: API of the `vector_fractional` type. The type is templated on the number of fractional digits, i.e. its precision, `p`.

Therefore, a parallelized lookup table for the `vector_fractional` type using `uint8_t` type as index is developed. Table 2.2 shows the API.

member function	specification
`vector_fractional_lookup()`	Default constructor
`vector_fractional<p> lookup(vector uint8_t)`	Lookup all entries of the supplied index vector and create a `vector_fractional` instance with the looked-up values
`void set_entry(uint8_t index, float)`	Set lookup value at specified `index`

Table 2.2.: API of the `vector_fractional_lookup` lookup table. Lookup is parallelized by providing a vector of indices as type `vector uint8_t`. In the case of the NSEM rule, this would be the weights as directly read using the vector unit of the PPU. The lookup table is templated on the number of entries (256 at max.) and the fractional precision, i.e. number of digits, of its entries.

2.3.2.4. Usage example

The use of the fractional vector, the lookup table and the stochastic down-conversion is exemplified for the implementation of the NSEM learning rule. Here, a NSEM plasticity rule instance holds a lookup table of precision p for the exponential part of equation (2.21) and a fractional vector instance of precision $p+1$ for the regulatory constant. Each update, the lookup table is queried for the weights. The resulting fractional vector instance is multiplied with the correlation measurement vector and the regulatory constant is subtracted. This temporary fractional vector of precision $p+1$ is then stochastically down-converted to integer values, which are added to the old weights and set to the synapses. Listing 1 shows a simplified excerpt of the NSEM plasticity rule implementation. The full implementation is displayed in Listing 4.

2.3.3. Plasticity rule encapsulation and masked execution

The NSEM network features two plasticity rules, the NSEM rule and the homeostasis. To allow reusing of the rule implementations, the rules are encapsulated in an object-oriented manner. Each rule is comprised of a structure capable of initialization, setting parameters and executing the update algorithm.

Additionally, an execution mask for the synapses is to be supplied for the learning rule. The NSEM rule implemented as part of this thesis uses the masking and encapsulation

```
template <class rng, std::size_t precision>
class StochasticSEM {
        rng* m_random;
        vector_fractional_lookup<num_hw_weights, precision> m_exp_lookup;
        vector_fractional<precision + 1> m_regulatory;

public:
        vector uint8_t kernel(
            vector uint8_t const& weights, vector uint8_t const& causal)
        {
                return weights + draw(
                    (m_exp_lookup.lookup(weights) * causal) - m_regulatory,
                     m_random);
        }
};
```

Listing 1: Usage of the `vector_fractional` type, the lookup table and stochastic down-conversion with the `draw` function in the NSEM plasticity rule implementation. The plasticity rule instance holds a pointer to a random number generator instance used for the stochastic down-conversion and a lookup table for the exponential part as well as a constant fractional vector for the regulatory part of equation (2.21). The `kernel` member function implements the update algorithm by first looking up the exponential part, the multiplying with the correlation measurement, afterwards subtracting the regulatory constant and finally performing a stochastic down-conversion to an integer vector.

framework already presented in (Spilger 2018). The rule is constructed of an algorithm operating on vectors of synapses and a mask supplying information as to which part of the synapse vectors to operate on. This is possible, as the NSEM rule is fully local, i.e. only the weight and correlation measurement of a specific synapse is needed in order to update this synapse's weight.

To the contrary, the homeostasis rule logically operates on the neuron scale and has an excitatory and an inhibitory synapse associated to every neuron to update. Masking of execution therefore is implemented by supplying a neuron mask in the form of two 16-entry binary vectors and information about the excitatory and inhibitory row to operate on.

2.3.4. Update scheduling

The plasticity rule algorithms displayed in Equations (2.20) and (2.21) are to be performed periodically with precise timing. Wrong timing implies wrong interpretation of the rates measured in the homeostasis rule and the correlation measurement in the NSEM rule. To guarantee precise timing of single rule execution and execution of tasks in parallel, the earliest-deadline-first real-time scheduler developed in (Spilger 2018) is used. It allows specification of timing requirements independent of the task to be performed and the scheduling process it is executed in. In addition, requested timing can be evaluated after execution to verify correct behavior.

2.3.5. Experiment Encapsulation

Each experiment consists of supplying parameters, executing the experiment, evaluating and eventually saving results. To provide a common code-base for all single-playback-program experiments, an `ExperimentBase`-class providing hardware access and experiment functionality used in every experiment is developed. It provides a playback program builder, helper functions such as for loading a PPU program from a file and inserting a load and start instruction to a program builder or for applying calibration data. Every experiment inheriting from the base class encapsulates instructions to create an experiment as a single playback program. As from a users point of view, there's no difference to higher order experiments, i.e. experiments consisting of multiple single-playback-program experiments, they share their interface with the single-playback-program experiments. An abstract experiment interface is outlined in Listing 2. Encapsulation of experiments with a common interface eases re-usability and combination of experiments. A real world example would e.g. be specific calibration with an experiment afterwards dependent on the previous results, which are inserted as calibration parameter there.

```
class ExperimentA {

public:
        /// Provide initial parameters for experiment creation.
        ExperimentA(parameters...);

        /// Additional parameter settings.
        void set_parameter(value_t value);
        void get_parameter(value_t value);

        /// Calibration settings.
        void set_capmem(CapMem capmem);
        void set_board(Board board);
        void get_capmem(CapMem capmem);
        void get_board(Board board);

        /// Experiment execution, e.g.\ on hardware.
        void execute();

        /// Evaluate and get some experiment result.
        result_t get_result();

        /// Save results, parameters, calibration, etc.
        void save(std::string filename);
};
```

Listing 2: Common experiment interface featuring parameter setting, calibration supply, experiment execution, evaluation and saving of results. This allows for seamless combination of experiments, e.g. sequentially dependent on the previous experiment's results.

2.3.6. Spike-train generation and combination

During the experiments performed in Chapter 3, multiple spike trains are to be processed interleaved in the FPGA playback program. During playback program generation, instructions can only be added sequentially. Because the spikes correspond to multiple interleaved spike-trains with different targets, efficient generation and combination is necessary. In addition to pre-synaptic spikes, also post-synaptic correlation signal trigger events are treated as spikes and are to be combined. Therefore, an `AbstractSpike` type is developed that can hold a variant from the set of `PreSpike` and `PostSpike` events. As an abstract spike initially does not hold a spike target description but only a time, abstract spike-train generation is possible by only specifying timing information and supplying the target information afterwards. This allows for pregeneration and reuse of generated spike-trains.

Using `C++` standard library `std::vector<AbstractSpike>` and `std::sort`, spike-trains with different type and target can then be concatenated or interleaved after individual generation. The two spike sources used in the experiments carried out of this thesis are regular and Poisson spike-trains. A spike source thereby refers to a spike-train processed from the playback program in the following. For these two spike-trains generator functions are implemented yielding a vector of abstract spikes without target, enabling its specification afterwards.

2.3.7. Serialization of `haldls`-Containers

Every hardware configuration is abstracted as a container in the `haldls` hardware abstraction software layer. Each container corresponds to a configurable feature of the neuromorphic system. To allow storage and reuse of configuration present during run-time, serialization of all containers present in `haldls` using the serialization library `cereal` (Grant and Voorhies 2017) is implemented in this thesis. This directly enables loading and saving of configuration in human-readable `JSON` (Crockford 2018) or more space saving `binary` format, depending on the application. Possible applications include calibration data insertion or transfer of static configuration between encapsulated experiments, described in Section 2.3.5.

3. Results

First, calibration of the correlation measurement and the neuron activation necessary for experiment conduction is explained and results are presented.

In the following, the implementation of the parts of the network described in figure 2.3 will be examined individually and in conjunction with each other. First, the NSEM learning rule, see equation (2.21), will be studied for a single synapse with artificially generated pre- and post-synaptic spikes. Then, the homeostasis implementation, outlined in section 2.2.2.3, is tested. Connecting these two parts, the single synapse learning rule is then combined with homeostatically controlling the post-synaptic neuron's firing rate. Subsequently, the combination of 5×5 synapses equipped with the NSEM learning rule connected to a homeostatically stabilized post-synaptic neuron will be evaluated on inferring the input rates of presented images. Afterwards, the winner-take-all network is inspected isolated in combination with the homeostasis. Finally, the full model is implemented as a set of three neurons in a winner-take-all network with 5×5 learning NSEM synapses. A set of three separated input pattern is presented to the network and its capability of performing classification is investigated.

Additionally, the performance both in time and in memory consumption of the PPU-based plasticity rules and the `vector_fractional` type is evaluated.

All following measurements are conducted using the same setup, i.e. combination of chip, FPGA and base board, which are documented in table A.1 in the appendix A.1.

3.1. Calibration

The analogue circuits are subject to fixed pattern variation and temporal noise. To calibrate the analogy neuron parameters, they are set through digitally configurable Capacitive memory (Capmem) cells providing analogy voltages or currents (Hock et al. 2013). The calibration data to each chip is stored in the database `dls2calib` and can be accessed via a `python` interface. The database is used for neuron calibration and yields bias current and neuron time constant calibration settings given a chip ID and requested target values in wall-clock-time. Conversion to biological time therefore is done as described in 2.2.3.1.

More information on the database structure, calibration procedure and results is found in (Stradmann 2016). The calibration data used is obtained from (Billaudelle 2018) using an automated calibration procedure[1].

As described in Section 2.2.2.1, in order to bring the neurons to a stochastic firing regime, the leak potential is to be adjusted to an activity of optimally $p = \frac{1}{2}$, i.e. the neuron fires with half its maximal firing rate when stimulated with excitatory and inhibitory noise.

In addition to neuron calibration, the correlation measurement is to be calibrated. The pair-based nearest-neighbor adaption described Section 2.2.2.4 assumes exponentially decaying correlation signals with time constant τ_{syn}. To transform measured correlation to the theoretical regime, the amplitude and time constant are to be measured and adjusted.

In the following, the additionally needed calibration steps are discussed in detail.

3.1.1. Correlation

The correlation measurement is read out by the correlation ADC (CADC). The voltage of the correlation storage capacitor is read out via a source follower in between the CADC and the capacitor (S. Friedmann and J. Schemmel et al. 2017). Both the CADC as well as the source follower have mismatch and need to be calibrated. To calibrate the correlation measurement, first the CADC is therefore to be calibrated to a desired voltage range. Then, the source follower is to be adjusted for linearity and desired range. Last, the correlation amplitude and time constant are to be measured and adjusted. The calibration procedure conducted follows (Wunderlich 2016).

3.1.1.1. CADC and Source follower

The CADC is a ramp-compare ADC. A periodic analogue linear ramp with offset is compared to the to be measured signal. Each period, a digital counter is started and stopped on comparator flip. The counter value thereby linearly digitizes the analogue value.

On DLSv2, the ramp amplitude (`ramp_slope`) and offset (`ramp_01`) are adjustable via DACs on the base board (Wunderlich 2016). They are however not independent from each other, so the amplitude and offset value are to be adjusted alternatingly approaching the desired bounds (Schreiber 2018).

Additionally, there are two bias parameters `ramp_bias` and `v_bias` adjusting the slope and the spread of channels (Schreiber 2018).

[1] This calibration is under code review at the time of writing this thesis and is found under the change set number 4419 in the repository `dls2calib-routines`.

Given an input interval $[U_l, U_u]$, the optimization goal for the ramp amplitude and offset is, that all CADC channels should not saturate at either end of the interval, be linear in between and use the maximally possible digital value range.

For the process of finding suitable ramp and offset parameters for given interval boundaries, an automated calibration program is developed as part of this thesis. This allows for potentially aligning different board-chip setups to perform the experiments conducted on another setup without the need to manually calibrate the CADCs. Although there also have been developed automated optimization programs for the bias parameters, the values are tuned manually, as the automated optimization did not converge reliably.

The reference input for the measurements is generated connecting an unused 2.5 V 10 bit DAC on the base board via external cables to debug pins connected to all CADC channels, prohibiting values above 1.25 V because of presumed damage to the CADCs (Schreiber 2018).

Figure 3.1 shows the resulting automated calibration for the CADC for given boundary voltages of $U_l = 0.21\,\mathrm{V}$ and $U_u = 0.92\,\mathrm{V}$. This calibration is used in the following for all correlation measurements. Smaller lower and larger upper bounds (not displayed) lead to increased spread towards the boundaries and especially imply saturation of some channels. As this again reduces the usable dynamic range, these bounds were discarded.

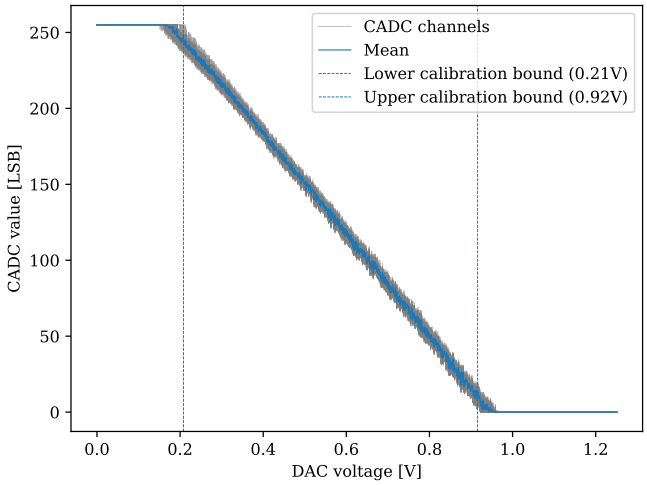


Figure 3.1: CADC input sweep over all 64 channels after calibration of `ramp_01` and `ramp_slope` for specified boundary voltages. The parameters found are documented in Table A.2.

With the calibrated CADC, the source follower linearity can be measured by adjusting the reset potential of the correlation circuit, as shown in (Wunderlich 2016), which is the potential to which the correlation curve is drawn at a post-synaptic spike. The reset potential is provided by a 1.2 V board DAC. The optimization goals are linearity and large input range. The source follower can be adjusted by changing `v_coroutbias`. Manual

calibration yields the CADC digital value to source follower input dependence depicted in Figure 3.2. For experiments, the reset potential is set towards minimal digital null-offset, around 1.1 V in Figure 3.2, to maximize the usable digital range without losing low-correlation information.

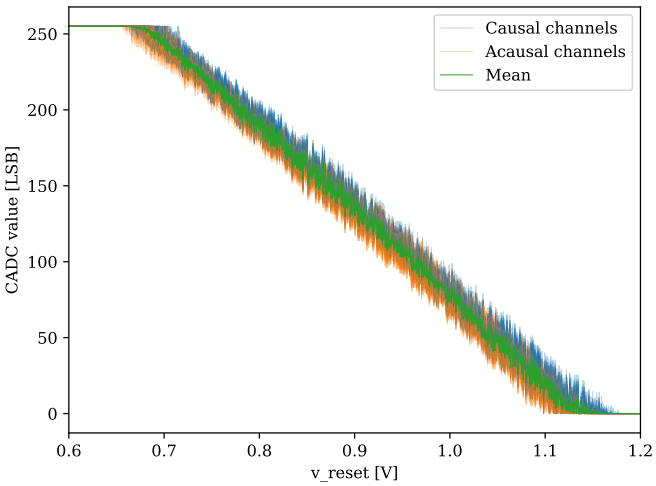


Figure 3.2: `v_reset` sweep with optimized `v_coroutbias`. The fluctuations are suspected to come from crosstalk of switching regulators on the FPGA board residing on the base board. The parameters found are documented in Table A.2.

Both the CADC response in Figure 3.1 and the source follower response in Figure 3.2 show noise, that is determined to vary in time for the setup used in this thesis, see Table A.1. On all setups with this crosstalk with $f_{\text{noise}} \approx 600\,\text{kHz}$ (wall-clock time) from the switching regulators (ABB Ltd 2013) on the FPGA board, is measured at the output side of the DAC used for sweeping, whereas for the one setup (Flyspi ID B291698), that does not feature the noise in the CADC measurements also no crosstalk is measured, suggesting a connection between the crosstalk and the noise in CADC measurements. The cause for the crosstalk however remains to be identified.

3.1.1.2. Causal Correlation

In order to translate correlation measurements on hardware to the theoretical measurements in Equation (2.21), the exponential decay amplitude and time constant are to be measured.

The causal decay is measured as described in (Wunderlich 2016) by sending a pre-synaptic spike and triggering the post-synaptic spike after a time Δt. This yields the value of the decay Δt after start. By sweeping the time difference, the course of the decay is recorded.

The post-synaptic spike is triggered by a pre-synaptic spike to an excitatory synapse driver, while the synapse driver of the synapse to be measured is disabled to not trigger a neuron,

i.e. post-synaptic, spike. The neuron is set to `bypass_exc`-mode, which bypasses the neuron circuit and is used to ensure each pre-synaptic spike on the excitatory synapse driver triggers a post-synaptic spike of the neuron.

This recording scheme allows to measure correlation of a single synapse or of multiple synapses by simultaneous stimulation, as at least only one synapse driver is needed to be excitatory to trigger neuron spikes and all other 31 synapse drivers' synapses can be measured, which greatly accelerates the measurement. Figure 3.3 shows the exponential decay for a single causal correlation sensor measurement. An exponential function of the

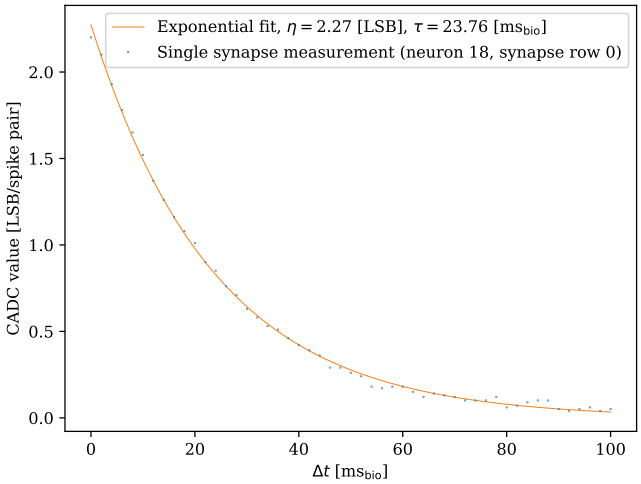


Figure 3.3: Causal correlation exponential decay of one synapse. The measurement is averaged 10 times and each run 100 spike pairs are emitted. The parameters used are shown in Table A.3.

form

$$f(\Delta t) = \eta \cdot \exp\left(-\frac{\Delta t}{\tau}\right) + o \tag{3.1}$$

is fitted to the data for each synapse, where η is the amplitude, τ the time constant of the decay and o the constant offset.

The time constant does not differ significantly between the simultaneous-stimulation and the single-synapse measurement (not displayed). However, the amplitude increases systematically with the number of synapse drivers used for stimulation. The reason for this is unclear. Figure 3.4 shows a violin plot of the individual dependencies from amplitudes of 32 synapses of one row to the number of simultaneously stimulated synapse drivers. The setup for different numbers of parallel measured synapse drivers thereby only differs in the number of rows receiving pre-synaptic spikes.

For single-synapse experiments, the single-synapse calibration is used for best accuracy, whereas for multi-synapse experiments, the calibration measuring half the synapse drivers is used to include the systematic amplitude drift and prevent overflow in the correlation measurement. Figure 3.5 shows the amplitude and time constant distribution for all 1024

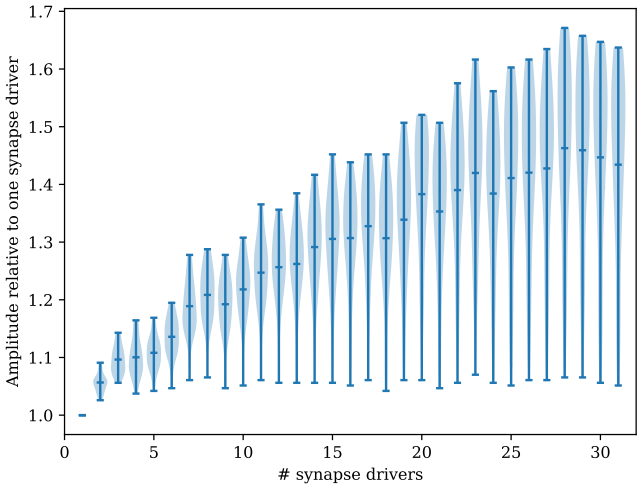


Figure 3.4: Violin plot, displaying the minimal, maximal and mean value in combination with the value distribution, of the amplitudes of 32 synapses of the second synapse driver in dependence of the number of parallel measured (i.e. fed with spikes) synapse driver's synapses relative individually to the measurement of only one synapse driver. The amplitude relative to only one synapse driver stimulated increases with the number of synapse drivers to about 50 % for 31 synapse drivers.

synapses measuring half of the synapse rows in parallel. In experiments involving several NSEM synapses, the mean of the specific subset of synapses used is used as amplitude and time constant measurement.

The amplitude and the time constant can be adjusted globally via the parameters `syn_v_store` and `syn_v_ramp` respectively, see (Wunderlich 2016). Additionally, there are two bit calibration adjustments available locally per synapse (S. Friedmann and J. Schemmel et al. 2017). (Wunderlich 2016) shows, that the time constant spread can be reduced by about a factor of 1.7 to 2.1 (mean absolute deviation), whereas the amplitude spread can be reduced by a factor of around 1.4 to 1.7 by optimizing the calibration bits.

For the experiments conducted however, the calibration bit setting is left untouched at time calibration 0 and amplitude calibration 1 for all synapses, because there was no directly usable optimization solver available for the parameter optimization. The highest amplitude setting of 0 is by design not intended to be used (Wunderlich 2016), which is the reason for choosing the second highest setting of 1.

Instead or in addition to reducing the fixed pattern noise by using the calibration bits, the author proposes a different countermeasure to align the correlation measurements as a future improvement of the NSEM learning rule implementation. Equation (2.21) shows, that both the time measurement and amplitude measurement can be combined in a single

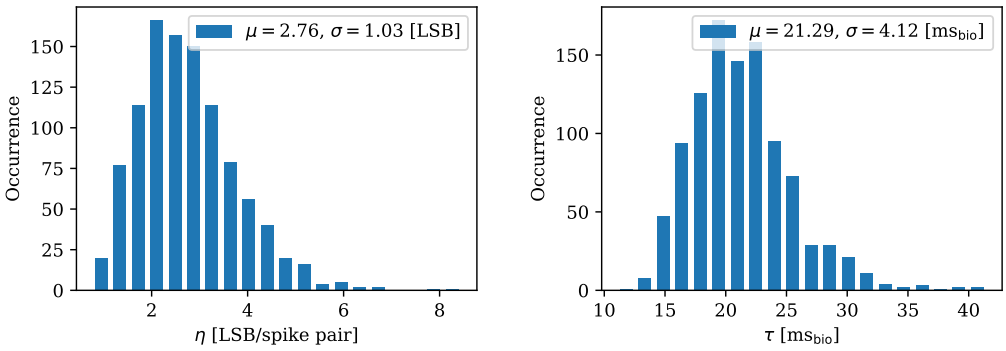


Figure 3.5.: Amplitude and time constant distribution for all 1024 synapses. The correlation parameter used are displayed in Table A.3.

multiplicative factor of the correlation read-out. Therefore, a multiplicative 8 bit calibration factor is suggested for each synapse, leading to an additional PPU memory consumption of maximally 1 kB, if all synapses are to be used and an additional multiplication to be performed at each weight update. Therefore, the precision of the correction is expected to yield smaller spread than is achievable by using only the calibration bit settings. This method's memory consumption however increases linearly with the number of synapses on a chip. The succeeding chip version features 512×256 synapses and the same PPU main memory size of 16 kB but can access the significantly larger FPGA memory at the expense of a high access time (Müller 2018). As the learning rule described in Equation (2.21) can be updated with a period of several seconds biological time, accessing the calibration coefficients stored on the FPGA memory with partial prefetching might be possible without significantly enlarging the learning rule's update period.

3.1.2. Neuron Activation

The neuron's threshold, reset and leak potential is not covered by the calibration described in (S. A. Aamir and Y. Stradmann et al. 2018). The calibration database however yields an approximate voltage to digital Capmem-value conversion.

The reset potential is global for all neurons and is set to 0.6 V (340 LSB), the threshold is adjusted to be at 0.8 V (458 LSB). These settings are a trade off between signal to noise ratio and a reset potential near the threshold, as demanded by Section 2.1.6. The reset and threshold settings are fixed for all measurements involving background noise to bring the neurons in a stochastic firing regime.

Then, the activation, the fraction of the neuron's spiking activity relative to the maximal neuron spiking activity, is measured by applying excitatory and inhibitory Poisson

background noise and measuring the dependency of the activation on the leak potential.

Figure 3.6 shows the activation in dependence of the leak potential for a background noise rate of $\nu_{\text{bg}} = 100\,\text{Hz}_{\text{bio}}$ and a weight of $w_{\text{bg}} = 45\,\text{LSB}$ of one excitatory and inhibitory noise synapse. The inverse slope α of the logistic fit corresponds to the inverse slope in

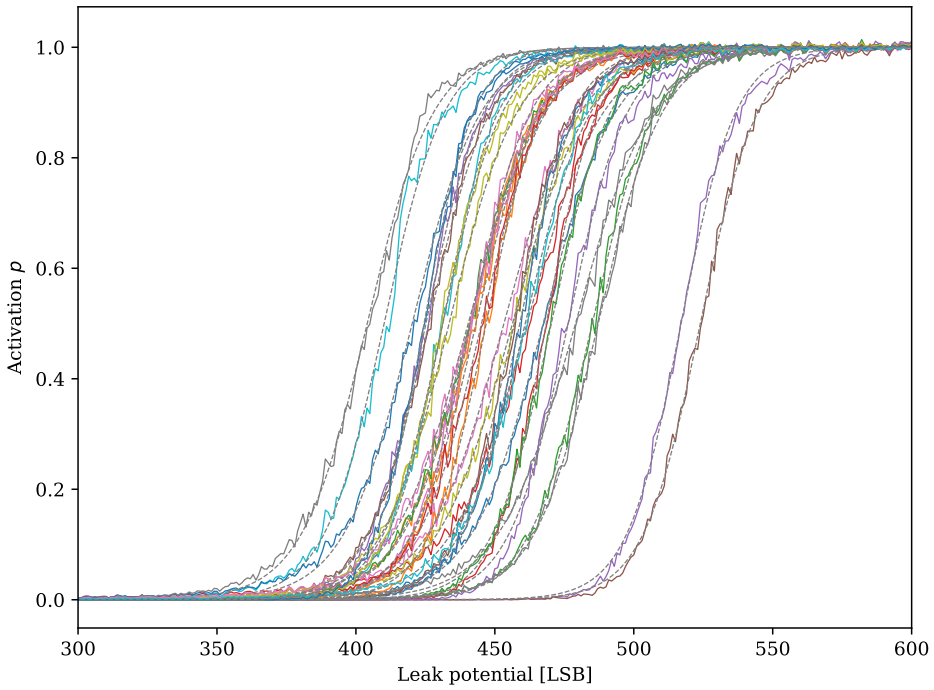


Figure 3.6.: Activation of all neurons except neuron 25 for chip 30 listed in Table A.1, which showed erratic behavior. The Poisson background noise rate is $100\,\text{Hz}_{\text{bio}}$, the background noise synapses have a weight of 45 LSB. For each activation, a logistic fit is performed, displayed as gray dashed line. The individual activations are normalized on the amplitude of their fit.

Equation (2.10) and linearly influences the hardware weight to theory weight conversion, described in Equation (2.13). For experiments involving three neurons, neurons with similar slope and maximal spike rate are chosen and the leak potential is individually set to reside at $p = \frac{1}{3}$ to be near the target mean activity in a cause layer consisting of three neurons, where each neuron is set to spike with an equal fraction of the maximal network rate $\nu_{\text{net}}^{\text{max}}$.

3.2. Single synapse NSEM

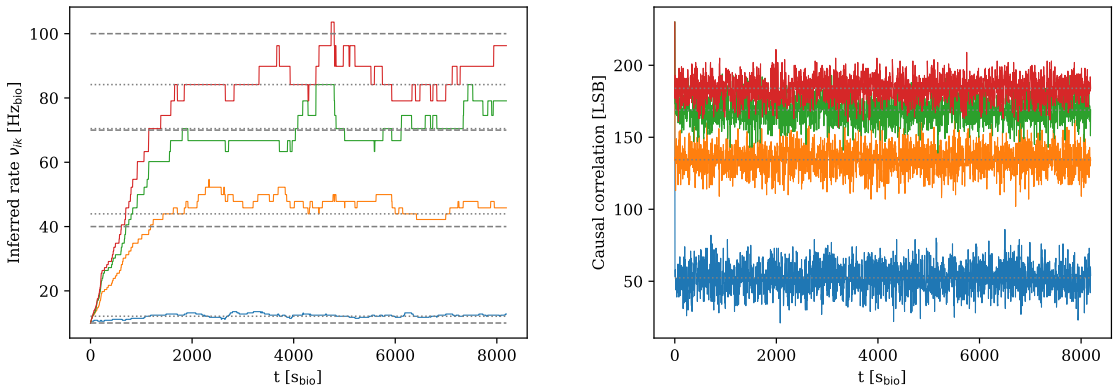
To test the PPU-based NSEM learning rule, a single synapse is investigated in an artificial environment. The pre- and post-synaptic spike trains are provided similarly to the single synapse correlation measurement in Section 3.1.1.2. The synapse under investigation is located in a synapse row with disabled synapse driver, to prevent it from triggering post-synaptic neuron spikes. The post-synaptic spike-train is inserted by an additional excitatory synapse leading to a on-to-one spike-response of the neuron set to `bypass_exc`-mode.

The post-synaptic spike train is set to be regular with a firing rate of $\nu_k = 30\,\mathrm{Hz_{bio}}$ to mimic the response of a neuron with $\tau_{\mathrm{ref}} = \tau_{\mathrm{syn}} = 30\,\mathrm{ms_{bio}}$ that is firing with almost (90 %) of it's maximum firing rate as will be the case in the WTA circuit if said neuron "wins". The pre-synaptic spike train is set to be Poisson with mean firing rates ranging from $10\,\mathrm{Hz_{bio}}$ to $100\,\mathrm{Hz_{bio}}$ to mimic the input-layer spike trains of images presented later.

To maximize the weight resolution in the according rate range under investigation, the null-cause rate is set to be $10\,\mathrm{Hz_{bio}}$, whereas the weight conversion factor in Equation (2.22) is chosen such that the maximally possible hardware weight of 63 LSB corresponds to a weight value, an input neuron firing with $100\,\mathrm{Hz_{bio}}$ would induce. The learning rate is set to 5×10^{-5}, the update time period is fixed to be $\tau_{\mathrm{update}} = 4\,\mathrm{s_{bio}}$.

Figure 3.7 shows the inferred input rates, calculated from the learned hardware weights via Equations (2.18) and (2.22) for an experiment duration of $T_{\mathrm{exp}} = 8000\,\mathrm{s_{bio}}$ for input rates ν_{ik} of $10\,\mathrm{Hz_{bio}}$, $40\,\mathrm{Hz_{bio}}$, $70\,\mathrm{Hz_{bio}}$ and $100\,\mathrm{Hz_{bio}}$ together with the causal correlation measurement. In addition to the expected inferred input rates from the inserted pre-synaptic spike trains, the target rate given the average measured correlation is depicted.

While the inferred rates deviate slightly from the input rates, they clearly wiggle around the target rate given from the measured mean correlation. This shows, that the learning rule algorithm given a certain correlation infers the correct input rates. Several measurement (not displayed) of the same setup using slightly different calibration values for the correlation measurement, namely the offset, amplitude and time constant, showed large changes of the inferred input rates around the inserted ones, especially for high input rates, as small changes in the correlation measurement lead to high changes in the inferred rates there. It is therefore suspected, that given a better correlation calibration, the inferred input rates would match the inserted input rates even closer.



(a) Inferred input rate. The gray dashed line refer to the actual input rates. The gray dotted lines represent the expected inferred input rates given the mean correlation measurements from Figure 3.7b.

(b) Correlation measurement. The mean value for each input rate is depicted as gray dotted line.

Figure 3.7.: Inferred rates (Figure 3.7a) and causal correlation (Figure 3.7b) trace. The four different input rates 10 Hz$_{\text{bio}}$, 40 Hz$_{\text{bio}}$, 70 Hz$_{\text{bio}}$ and 100 Hz$_{\text{bio}}$, displayed as gray dashed lines, are measured and displayed in blue, orange, green and red respectively. The weight trace is translated to inferred input rates in Figure 3.7a using Equations (2.18) and (2.22). The stochastic weight updates are clearly visible especially for the two higher input rates. Here, as the target rate is approached, the weight updates become smaller and the weight changes only after several update cycles. The correlation and learning rule parameters used are collected in Table A.4.

3.3. Homeostasis

The homeostasis is tested isolated by controlling a neuron's target rate, that is initially non-spiking. A regular and a Poisson homeostasis background source are compared in terms of spread and accuracy in reaching the target rate. Additionally, the effect of applying background noise with differing rate is evaluated to investigate the homeostasis' capability to adapt to changes in the neuron's excitability.

To mimic the setting used later for learning, the refractory period and synaptic time constant are set to $30\,\mathrm{ms_{bio}}$, the membrane time constant is set to $1\,\mathrm{ms_{bio}}$. The target rate is set to $10\,\mathrm{Hz_{bio}}$, which is slightly smaller than $\frac{1}{3}\nu_{\mathrm{net}}^{\mathrm{max}}$ given the time constants used for a network consisting of three neurons. The update time period is set to $4\,\mathrm{s_{bio}}$, the background source rate to $200\,\mathrm{Hz_{bio}}$ and the learning rate to $\eta_b = 0.0156$.

Figure 3.8 shows the rate and weight time course for neuron 1 using synapse driver 0 as excitatory and synapse driver 1 as inhibitory row. Information on the setup are found in Table A.1. As expected, the excitatory weight is increased until the rate reaches the target value. The rate then is stabilized through small changes in the excitatory weight.

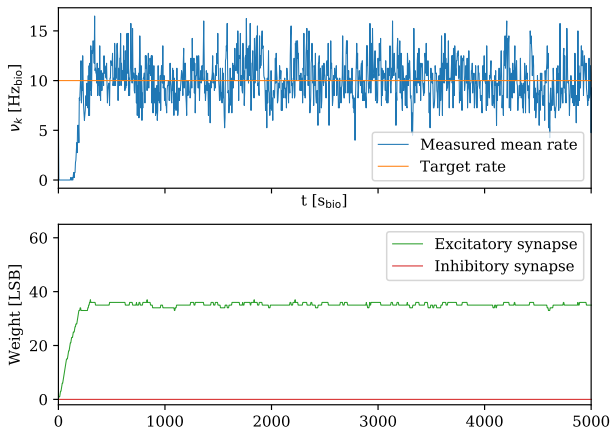


Figure 3.8: Neuron spike-rate and hardware weight time course of homeostasis applied on neuron 1 of chip 30 using a Poisson background source. The target rate is set to be $10\,\mathrm{Hz_{bio}}$. The rate is obtained by storing the rate counter readout. As expected, the target rate is reached by only applying excitatory input to the membrane.

To see the regulatory effect of the homeostasis on varying neuron excitability, excitatory background is applied using 10 synapses with hardware weight $w_{ik} = 10\,\mathrm{LSB}$ subject to the same spike-source. Three different Poisson spike-trains with the rates $50\,\mathrm{Hz_{bio}}$, $200\,\mathrm{Hz_{bio}}$ and $25\,\mathrm{Hz_{bio}}$ are applied continuously for a third of the experiment time each. The learning rule parameter are set as before. Figure 3.9 shows the neuron's spike-rate and homeostasis weight time course. The neuron is stabilized around the target rate for all three excitability settings. On positive change in the excitability, i.e. from $50\,\mathrm{Hz_{bio}}$ to $200\,\mathrm{Hz_{bio}}$ background

rate, the neuron's rate instantaneously increases and is pulled back down to the target rate by the homeostasis by changing the homeostasis weight to negative, i.e. inhibitory values. Analogously, on decreasing excitability at the transition from 200 $\mathrm{Hz_{bio}}$ background to 25 $\mathrm{Hz_{bio}}$ background rate, the neuron's rate drops and the homeostasis again stabilizes its rate through now applying excitatory input to the membrane.

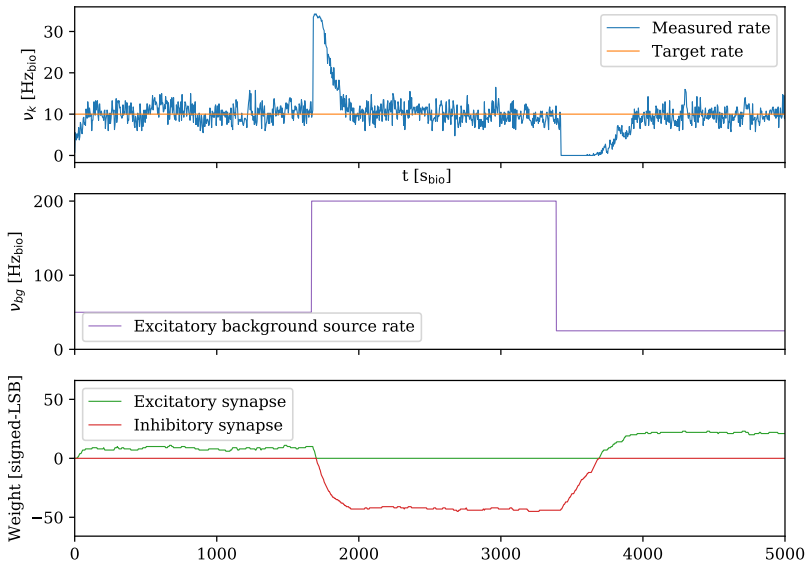


Figure 3.9.: Neuron spike-rate and hardware weight time course of homeostasis applied on neuron 3 of chip 30 using a Poisson background source with varying background excitation. The target rate is set to be 10 $\mathrm{Hz_{bio}}$. The rate is obtained by storing the rate counter readout. The background excitation is applied through 10 excitatory synapses with fixed weight $w_{ik} = 10\,\mathrm{LSB}$ subject to the same spike-source. At background excitation change, the rate jumps towards higher values on a change to a higher rate and to lower values on a change to a lower rate. The homeostasis is able to again stabilize the rate after the excitation changes. The weight trace shows the seamless switch from the excitatory to the inhibitory homeostasis synapse in the case, the neuron exceeds the target rate because of high background excitation.

In order to compare Poisson and regular homeostasis background sources, the rate distribution for both sources is examined after stabilization. Figure 3.10 shows the rate distribution after stabilization of a homeostatically controlled neuron with either Poisson or regular homeostasis background source. While both sources yield an average rate not significantly differing from the target rate of 10 $\mathrm{Hz_{bio}}$, the regular source's rate distribution is far broader than the Poisson source's distribution, that is clearly centered around the

target rate. Therefore, the Poisson background source is used for all following measurements. This finding is contrary to the result found in (Breitwieser 2015, Chapter 5), where in simulation the regular background source outperforms the Poisson source in learning. The different behavior is to be investigated in the future.

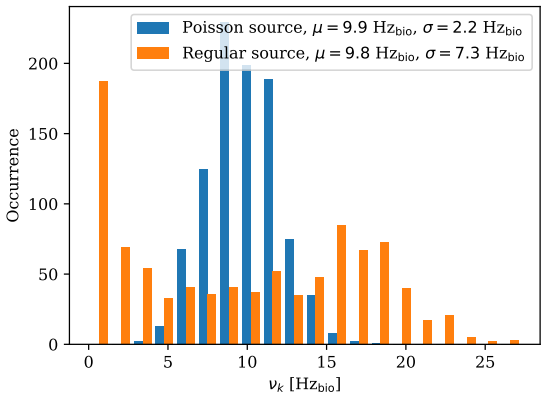


Figure 3.10: Neuron spike-rate distribution of a neuron stabilized by homeostasis to a target rate of $10\,\text{Hz}_{\text{bio}}$ for a regular and a Poisson homeostasis background source. The rates are obtained by storing the rate counter readout. While for both background sources, the mean rate of $9.8\,\text{Hz}_{\text{bio}}$ for regular and $9.9\,\text{Hz}_{\text{bio}}$ for Poisson is near the target rate, the regular source rate distribution is more than three times wider than the Poisson source distribution and does not feature a clear maximum around the target rate.

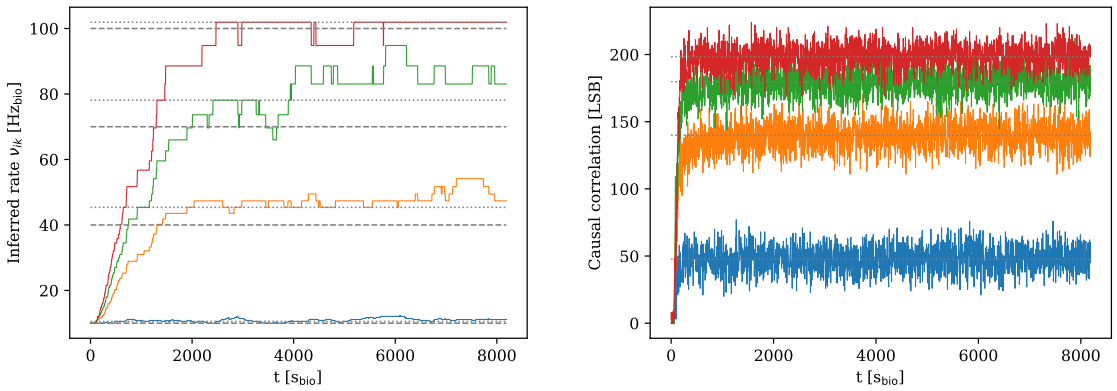
3.4. Homeostatically controlled neuron with NSEM synapse

Similar to the single synapse NSEM learning rule experiment described in Section 3.2, a single synapse equipped with the NSEM learning rule is examined. Instead of artificially providing the post-synaptic neuron's spike-train, the neuron is now stabilized using the homeostasis inspected in Section 3.3.

The neuron target spike rate is set to $28\,\text{Hz}_{\text{bio}}$, to align the spiking behavior to the case, where several neurons are connected via a winner-take-all network with a similar ν_k^{net}. The refractory time period as well as the synaptic time constant are set to $30\,\text{ms}_{\text{bio}}$, the membrane time constant is set to $1\,\text{ms}_{\text{bio}}$. The update time period for both the homeostasis and the NSEM rule is set to $4\,\text{s}_{\text{bio}}$ to arrange for sufficient averaging in the correlation as well as the rate counter measurement. The homeostasis background rate is set to a $200\,\text{Hz}_{\text{bio}}$ Poisson source.

Figure 3.11b shows the correlation time course. The inferred input rates are depicted in Figure 3.11a.

Comparing the correlation measurements to the ones depicted in Figure 3.7b, the correlation, especially for the higher values, is systematically higher for the homeostatically



(a) Inferred input rate. The gray dotted lines represent the expected inferred input rates given the mean correlation measurements (mean from $1024\,\mathrm{s_{bio}}$ to $8092\,\mathrm{s_{bio}}$) from Figure 3.7b.

(b) Correlation measurement. The mean value for each input rate is depicted as gray dotted line.

Figure 3.11.: Inferred rates (Figure 3.7a) and causal correlation (Figure 3.7b) trace. The four different input rates $10\,\mathrm{Hz_{bio}}$, $40\,\mathrm{Hz_{bio}}$, $70\,\mathrm{Hz_{bio}}$ and $100\,\mathrm{Hz_{bio}}$, displayed as gray dashed lines, are measured and displayed in blue, orange, green and red respectively. The weight trace is translated to inferred input rates in Figure 3.7a using Equations (2.18) and (2.22). The correlation and learning rule parameters used are collected in Table A.6 in the appendix.

controlled case than for the artificial post-synaptic spike train case. In the homeostatically controlled case, three synapse drivers experience spike input, while in the artificial case, only two synapse drivers do. This can be explained by taking into account the observations made in Section 3.1.1.2, that the amplitude of the correlation measurements increases with the number of synapse drivers receiving spike input. In order to account for that, the correlation calibration used for the NSEM learning rule is scaled accordingly by a factor of 1.1 read out of Figure 3.4.

It should be noted, that neuron target rates lower than the $28\,\mathrm{Hz_{bio}}$ set lead to a higher post-synaptic rate variation which largely affects the correlation measurement spread (not displayed). This leads to vanishing differences of the mean correlations between the higher three rates examined and thereby destroys the separation between the different rates. Since in the winner-take-all circuit, one neuron's activity however will typically dominate per presented pattern, post-synaptic firing rates are close to the maximum activity $\nu_{\mathrm{net}}^{\mathrm{max}} = \frac{1}{\tau_{\mathrm{ref}}}$.

3.5. Homeostatically controlled neuron inferring 5×5 pixel images

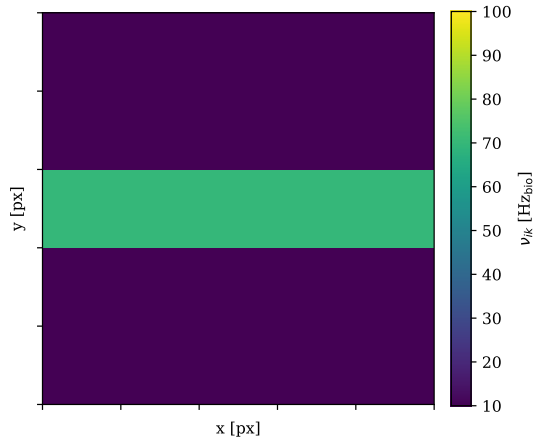
As described in Section 2.2.1, each neuron on the DLSv2 has 32 synapses associated via which it can receive synaptic input. Therefore the quadratic input image with the maximal resolution is a $5\,\mathrm{pixel} \times 5\,\mathrm{pixel}$ image. The NSEM learning rule is therefore tested for input pattern on 25 synapses with a homeostatically controlled neuron with a target rate of $33\,\mathrm{Hz_{bio}}$. As described in Section 3.1.1.2, the average time constant and amplitude of the correlation measurement is used in computation of the λ_{i0}^{nn} constant in Equation (2.21) and the correlation scale factor.

The images presented are chosen to be a one pixel wide stick, that is rotated to 0° and 90°. The stick is seen as a pixel value of 1, corresponding to a rate of $70\,\mathrm{Hz_{bio}}$ of the input spike train. Background pixels are set to a input rate of $10\,\mathrm{Hz_{bio}}$, the null cause rate of the NSEM rule is set to $15\,\mathrm{Hz_{bio}}$. The background pixel rate being below the null-cause rate increases the contrast in inferred weights, as the background pixels are drawn more towards zero weight. For further information about the contrast enhancing by increasing the null-cause rate, see (Breitwieser 2015, Section 5.4.3).

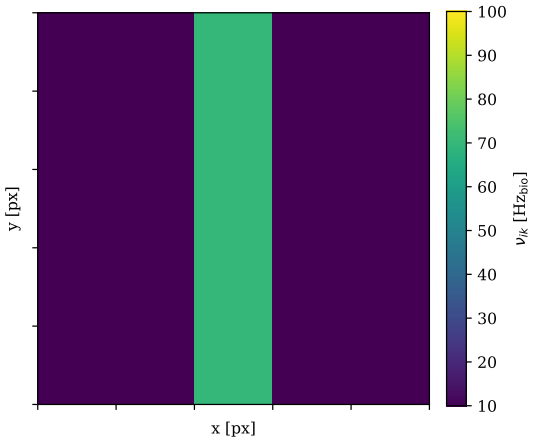
The refractory time period as well as the synaptic time constant are set to $30\,\mathrm{ms_{bio}}$, the membrane time constant is set to $1\,\mathrm{ms_{bio}}$. The update time period for the NSEM rule is set to $4\,\mathrm{s_{bio}}$ and for the homeostasis set to $1\,\mathrm{s_{bio}}$. The homeostasis background rate is set to a $250\,\mathrm{Hz_{bio}}$ Poisson source.

Figure 3.12 shows the inferred rates after $5000\,\mathrm{s_{bio}}$ experiment execution for a stick of rotation angle 0° and 90° in comparison with the actual input rates.

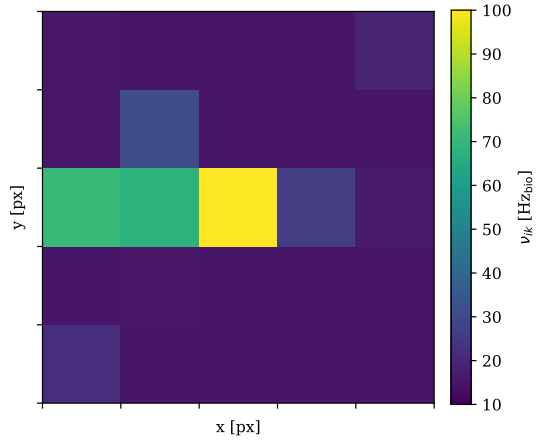
While the input pattern are clearly visible in the inferred rates, the absolute rates deviate from the actual input rates. This is suspected to arise from the correlation fixed pattern noise, because the average correlation time constant and amplitude are used for both the calculation of plasticity rule parameters and the back conversion of hardware weight values to inferred rates.



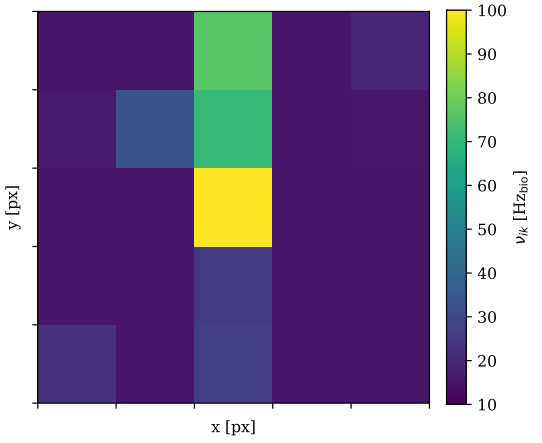
(a) Input rate of horizontal stick.



(b) Input rate of vertical stick.



(c) Inferred rates for horizontal stick input pattern.



(d) Inferred rates for vertical stick input pattern.

Figure 3.12.: Actual and inferred input rates for a horizontal and a vertical stick input pattern. The stick corresponds to 70 Hz_{bio} input rate, the background to 10 Hz_{bio}. The supplied pattern is clearly visible in the inferred rates although the absolute rates inferred deviate from the actual input rates. This is suspected to originate from the variation in correlation amplitude and time constant around the mean used for the plasticity rule parameters and the back conversion from weight values to inferred rates. The parameters used are collected in Table A.7.

3.6. Winner-take-all network

As outlined in Section 2.2.3, the cause layer winner-take-all behavior is established through strong recurrent inhibitory all-to-all connections. Each cause layer neuron's spikes are routed to a number n of synapse drivers with a constant spike label. Each cause layer neuron's synapse in these synapse rows share the same weight, allowing thereby to adjust the inhibitory strength through setting the weight or changing the number of synapse drivers used.

To evaluate the performance of the network, three cause layer neurons are homeostatically stabilized at $7\,\mathrm{Hz_{bio}}$, which in sum is roughly $70\,\%$ of the maximally possible network rate of $\nu_{\mathrm{net}}^{\mathrm{max}} = 33\,\mathrm{Hz_{bio}}$ given the refractory time period and synaptic time constant are set to $30\,\mathrm{ms_{bio}}$. A full list of parameters used is found in Table A.8. The spikes of all cause layer neurons are recorded and added to a time-sorted list, agnostic of the neuron, the spike belongs to. Using this sorted list of spikes, the inter-spike interval (ISI), the time between consecutive spikes, is computed. It is expected, that the winner-take-all circuit leads to inhibition of spikes of all neurons in the network during the refractory time period of the previous spike's neuron.

Figure 3.13 shows the inter-spike interval distribution of $2500\,\mathrm{s_{bio}}$ measurement duration using $n = 3$ synapse drivers for the inhibition. Figure 3.14 shows a raster plot of the three neurons' spikes together with notion of each spike's refractory period and synaptic time constant. It is expected, that refractory periods don't overlap.

The inter-spike interval distribution shows inhibition for times below the refractory and synaptic time constant, as expected from the inhibitory connections. The raster plot shows, that for most spikes, no other spike occurs within the refractory period. When performing the experiment multiple times, the inter-spike interval distribution however sometimes does not feature this distinct minimum in the refractory and synaptic time constant range. The reason for this is not fully understood.

To verify, that the spike-routers delay does not have a significant effect, it is measured using two neurons, A and B, with external simultaneous excitatory stimulus. Both neurons are set to `bypass_exc`-mode, in order to have a one-to-one relation between stimulus and neuron's spikes. A's spikes are routed excitatory B. In this setting, B then spikes twice for each external excitatory stimulus, once via the external stimulus and once via the routed stimulus from A. The time difference between the successive spikes then is the delay of the spike router. Figure 3.15 shows the distribution of the spike routers delay for all combinations of emitting and receiving neuron for an average of 10 pre-synaptic spikes for each combination.

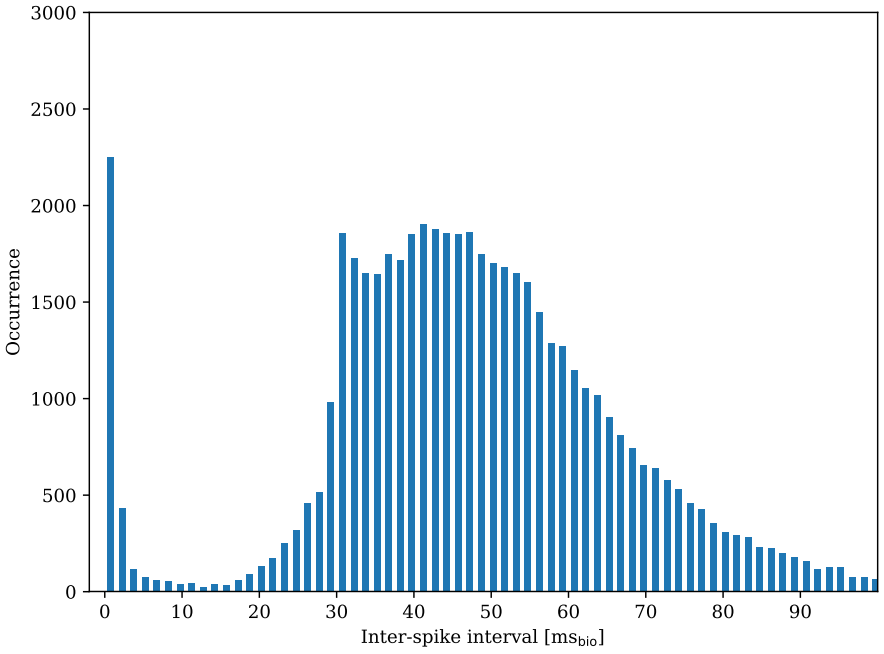


Figure 3.13.: Inter-spike interval distribution for three homeostatically controlled neurons at $\nu_k = 7\,\mathrm{Hz_{bio}}$ with refractory and synaptic time constant of $30\,\mathrm{ms_{bio}}$ and $n = 3$ inhibitory all-to-all connection synapses with weight $w_{kl} = 63\,\mathrm{LSB}$ each. As expected, the occurrence of inter-spike-intervals between $t_{\mathrm{ISI}} = 0\,\mathrm{ms_{bio}}$ and $t_{\mathrm{ISI}} = 30\,\mathrm{ms_{bio}}$ is significantly lower than for larger t_{ISI}.

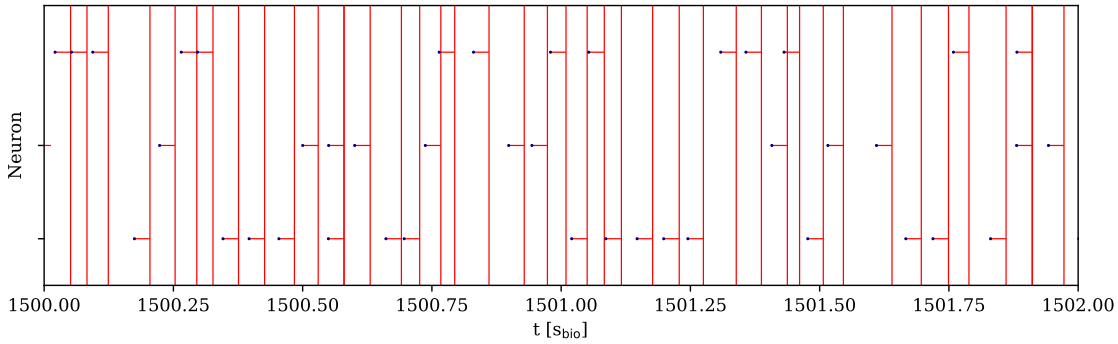


Figure 3.14.: Raster plot of the three neurons at $1500\,\mathrm{s_{bio}}$. Each spike is depicted as a blue dot. For each spike, the refractory period is depicted as red horizontal bar. At the end of each refractory period, a vertical red line is placed to better compare with the occurrence of subsequent spikes from other neurons. As expected, the occurrence of spikes within another spikes refractory period is low compared to the occurrence of larger inter-spike-intervals.

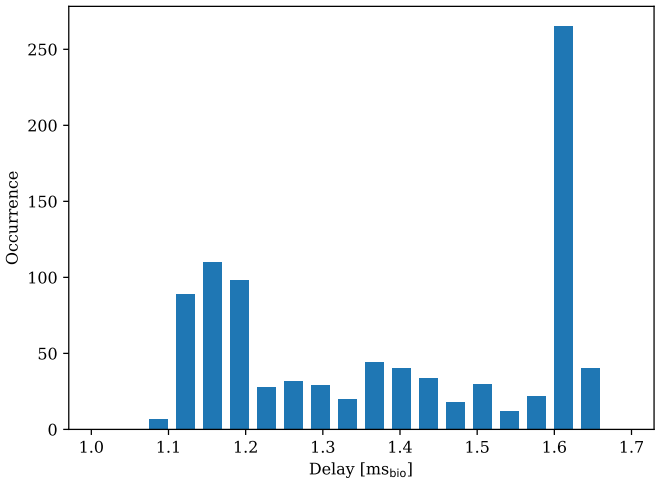


Figure 3.15: Averaged spike router delay distribution for 10 pre-synaptic spikes for all combinations of emitting and receiving neuron. The distribution shows a maximal delay of 1.65 $\mathrm{ms_{bio}}$.

The spike router delay ranges between 1.1 $\mathrm{ms_{bio}}$ and 1.65 $\mathrm{ms_{bio}}$. The high peak for low values in the ISI distribution in Figure 3.13 therefore can be explained through the spike router delay. During the delay period of a routed spike, other neurons are not inhibited by that spike and can therefore also spike.

3.7. NSEM network separated bar classification

The setup described in Section 3.5 is used and combined with the WTA spike routing of Section 3.6 for three cause layer neurons. The network therefore implements the full network described in Section 2.2.3. All times are presented in biological time using the conversion from 2.2.3.1.

To evaluate, whether the network can find hidden causes in input pattern presented, i.e. distinguish between them, a set of three exclusive input pattern, non-overlapping bars is used. The input pattern are displayed in Figure 3.16.

The experiment duration is 10 000 $\mathrm{s_{bio}}$. Because this duration with the amount of spikes to be sent in the network yields playback programs too large for execution as a whole, the experiment is split into parts of 1000 $\mathrm{s_{bio}}$ duration. The size limit of a FPGA playback program is determined to be around 91 MB, but smaller than 92.4 MB. After each part, the weights of the homeostasis and NSEM synapses are read out and used as starting point for the next part. This allows arbitrary learning durations.

There exists a known issue[2] with enabling the spike-router and read requests in the same playback program. Routed spikes corrupt answers to read requests. Therefore the weights

[2]This issue #2375 is to be found in `https://brainscales-r.kip.uni-heidelberg.de/projects/fpga-flyspi-boilerplate/work_packages`.

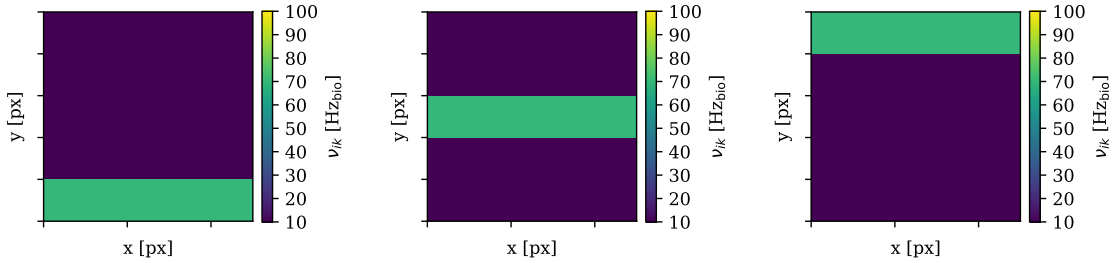


Figure 3.16.: Input pattern of non-overlapping bars. The bar corresponds to 70 Hz_{bio} input rate, the background to 10 Hz_{bio} to enhance the contrast.

are read out with an additional playback program after each experiment part, in which the spike-router is disabled to read uncorrupted weight values.

The update time period is set to 1 s_{bio} for the homeostasis and to 4 s_{bio} for the NSEM rule. The homeostasis background rate is set to a 250 Hz_{bio} Poisson source, the target rate is 10 Hz_{bio}. To enhance contrast, the null-cause rate is set to 15 Hz_{bio}. Images to be presented are drawn randomly from the three images in Figure 3.16. Each image is presented for 0.5 s without pause. A full list of parameters used is found in Table A.9.

Figure 3.17 shows the inferred rates after 10 000 s_{bio} combined experiment duration for the three cause layer neurons. The receptive fields show nearly exclusive classification of input pattern, i.e. a one-to-one relation between input pattern and specialized neuron. The receptive fields are sorted the same way as the input pattern in Figure 3.16. As the neurons decide randomly, which is going to specialize on which presented input, this order is not fixed. For further information see (Breitwieser 2015, Chapter 5). Figure 3.18 shows the corresponding raster-plot of the cause layer neurons' spikes after 9500 s_{bio} of learning. During presentation of each input pattern, one cause layer neuron spikes with a significantly higher rate than the others.

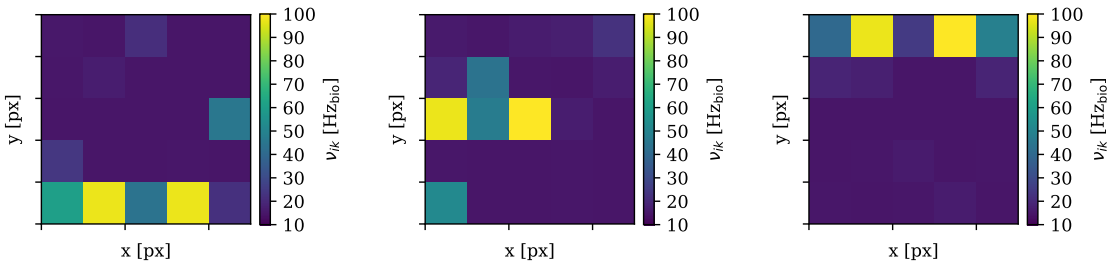


Figure 3.17.: Receptive fields of the three cause layer neurons after learning for 10 000 s_{bio}. The presented images are the horizontal bars depicted in Figure 3.16. Each receptive field shows strong specialization of each neuron for one input pattern.

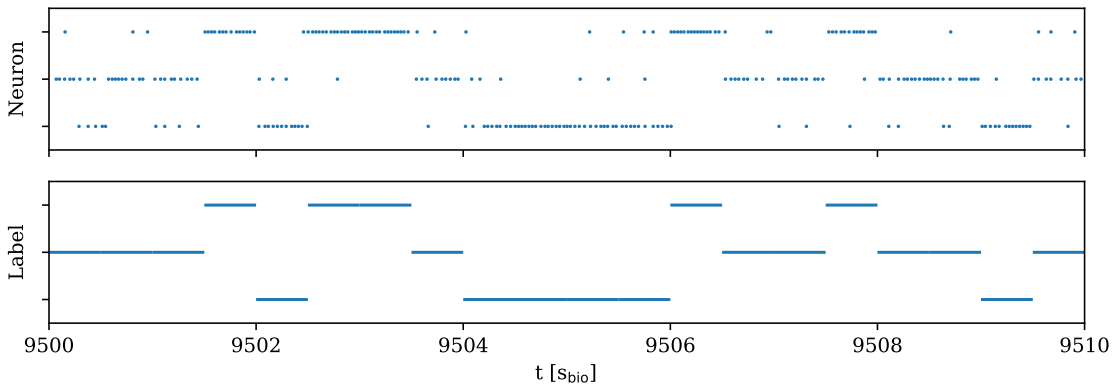


Figure 3.18.: Raster-plot of the cause layer neurons' spikes after 9500 s_{bio} of learning (top) together with the image labels presented (bottom). The presented images are the horizontal bars depicted in Figure 3.16. For each presented pattern, one neuron spikes with a significantly higher rate than the others.

3.8. Performance

This section presents time measurements of the homeostasis and NSEM rule implementation on the PPU as well as of the `vector_fractional` implementation. All measurements are conducted 10 times successively using the time register of the PPU, which counts clock cycles. The time difference varies between the consecutive measurements due to branch prediction of the processor. The maximal and minimal time measured are presented in the following and can be seen as a worst-case and best-case approximation for use in actual experiments. The measurements presented are carried out with PPU programs compiled with the `gcc` compile-time optimization flag `-O2`.

3.8.1. `vector_fractional` arithmetics

As described in Section 2.3.2, the arithmetics for the fractional vector type are implemented analogously to decimal arithmetic operations. Is is therefore expected, that the computation time increases with the precision of the fractional vectors, i.e. the number of fractional digits. Table 3.1 shows the time consumption for the arithmetic operations for a precision range of 0 to 4 7 bit fractional digits. In the case of no fractional digits, the algorithm transforms to the integer vector operations and can therefore be used as reference.

Number of fractional digits	0	1	2	3	4
Operation	Min. (max.) time consumption [ms_{bio}]				
`operator+(vector_fractional)`	0.9 (1.2)	2.0 (3.0)	3.5 (5.2)	6.5 (7.7)	8.8 (10.2)
`operator-(vector_fractional)`	0.9 (1.2)	2.0 (3.1)	3.5 (4.6)	6.8 (8.2)	8.9 (10.3)
`operator+=(vector_fractional)`	0.9 (1.1)	2.2 (3.4)	3.0 (4.1)	5.2 (6.4)	6.4 (7.7)
`operator-=(vector_fractional)`	0.9 (1.1)	2.2 (3.4)	3.0 (4.1)	5.2 (6.4)	6.5 (7.7)
`operator-()`	0.7 (1.0)	2.0 (2.3)	3.3 (3.7)	6.3 (7.9)	8.0 (9.7)
`operator*(vector uint8_t)`	1.3 (1.5)	2.8 (3.4)	6.0 (6.9)	8.0 (8.8)	9.9 (10.8)

Table 3.1.: Time consumption measurements of `vector_fractional` arithmetic operations for 0 to 4 fractional digits precision.

It can be clearly seen from comparing e.g. the `+` with the `+=` operation, that saving a `vector_fractional` instance from registers to memory significantly consumes time at higher precision. Therefore plasticity rule algorithms are to be optimized to the least amount of temporarily created objects. The operators `+=` and `-=` show a monotonically increasing dependency of the time consumption to the number of fractional digits as expected, because each additional digit adds two arithmetic operations, one at the digit's position and a carryover operation. The sign change, `+`, `-` and multiplication operator are also expected to show a monotonically increasing dependency of the time consumption to

the number of fractional digits. Opposed to the `+=` and `-=` operators, temporary objects need to be created during this operations and therefore the absolute time consumption is higher.

3.8.2. `vector_fractional_lookup`

The time for a lookup of a vector of indices in `vector_fractional_lookup` is displayed in Table 3.2. As expected, the time consumption increases with the number of fractional digits, as for each digit, the 16 entries of an additional vector are to be looked up. The time consumption for 2 and 3 fractional digits results in the same duration. The reason for this is unclear, but may be explained through differing compile-time optimization between the two cases.

Number of fractional digits	0	1	2	3	4
Operation	Min. (max.) time consumption [ms_{bio}]				
`lookup(vector uint8_t)`	2.5 (2.7)	3.0 (3.1)	3.8 (4.0)	3.8 (4.0)	9.0 (9.3)

Table 3.2.: Time consumption measurements of the `vector_fractional_lookup` lookup operation for 0 to 4 fractional digits precision.

3.8.3. Random number generator for stochastic down-conversion

The stochastic conversion from a `vector_fractional` instance to an integer vector, described in Section 2.3.2.2 needs a random number generator for digits, i.e. 16-entry 7 bit integer vectors. The time consumption of the implementation using the `xorshift128` algorithm is 0.8 ms_{bio} (1.3 ms_{bio}) per vector of 7 bit wide random numbers. For each fractional digit in a `vector_fractional` instance, one random number vector from the random number generator is needed. The time consumption for random number generation therefore scales linearly with the number of fractional digits. An alternative implementation using four calls of `xorshift32` needs 1.4 ms_{bio} (1.8 ms_{bio}) per vector of 7 bit wide random numbers. The `xorshift128` implementation is therefore more than 20 % faster than the `xorshift32` implementation.

3.8.4. Homeostasis

The homeostasis implementation uses fractional vectors of precision 2 to store the target rate and of precision 1 to fit the 10 bit wide measured rate from the rate counters. The internal structure and the API of the homeostasis plasticity rule is displayed in Listing 3.

The rate counters are read individually per neuron. Table 3.3 shows the time consumption of rate acquisition from the rate counters and conversion to `vector_fractional` values, the time consumption of the update algorithm, i.e. Equation (2.20) and the combination of the two parts, i.e. the time consumption for a full update cycle. All 32 neurons are processed at once.

Operation	Min. (max.) time consumption [$\mathrm{ms_{bio}}$]
read-out rates	9.3 (9.9)
weight update	22.3 (25.5)
read-out rates and do weight update	31.6 (32.0)

Table 3.3.: Time consumption measurements of the homeostasis update algorithm for all 32 neurons.

The measured time consumption shows a maximally possible update rate of about $\frac{1}{32\,\mathrm{ms_{bio}}} \approx 30\,\mathrm{Hz_{bio}}$ for the homeostasis. Because the rate counters measure the number of occurred spikes since the last reset, the update period should be significantly higher than the average firing rate of the neurons in order to average over the mean period of several spikes. In the experiments conducted, rates of about 40 spikes during an update cycle show sufficient averaging, see Section 3.3. A fast update execution is nonetheless important as it allows combination with other more time consuming plasticity rules to be executed alternatingly without significant processing time consumption. The combined maximal duration of reading rate counters and updating weights is faster than adding the isolated measurements. This may be explained by different branch prediction of the processor in the combined case.

3.8.5. NSEM plasticity rule

The NSEM learning rule is templated over the fractional precision of the calculations. Its internal structure, API and implementation is displayed in Listing 4. It uses the masking mentioned in Section 2.3.3. As shown in (Spilger 2018), the time consumption is therefore expected to scale linearly with the number of synapse vectors to update, because for each synapse vector in the mask, the same fully local update algorithm is called. For DLSv2, at most 64 synapse weight vectors are to be updated. Therefore, the time consumption for updates of all 64 vectors as worst-case approximation are depicted in Table 3.4 for a fractional precision of 1 to 3 digits. For comparison with the single arithmetic operation time consumption depicted in Table 3.1, the time consumption is additionally measured for one synapse vector. Aside from that, the table shows the time consumption of reading and writing a weight vector as well as reading and resetting a correlation measurement as

reference, i.e. lower bound of the possible execution time. This measurement is scaled up to 64 synapse vectors for comparison.

Operation	Min. (max.) time consumption [ms_{bio}]	
	64 vectors	1 vector
NSEM update (1 digit fractional precision)	1110 (1112)	17.4 (19.5)
NSEM update (2 digits fractional precision)	1597 (1600)	25.1 (27.0)
NSEM update (2 digits fractional precision)	1747 (1749)	26.6 (28.1)
weight and correlation read and write / reset	257 (273)	4.0 (4.3)

Table 3.4.: Time consumption measurements (parts of) of the NSEM update algorithm for all 1024 synapses and one synapse vector in comparison with the necessary hardware access time.

Comparison between the time consumption for the update of one synapse vector to 64 synapse vectors shows a linear scaling with the number of synapse vectors updated. (Breitwieser 2015) shows in simulation, that update periods of up to $10\,s_{bio}$ don't affect the capability to correctly infer input rates. The experiments conducted in Chapter 3 use an update period of $4\,s_{bio}$ and confirm on hardware, that the maximal time consumption of a full chip measurement of about $1.75\,s_{bio}$ for three fractional digits is low enough to be usable.

3.9. Storage requirements

As explained in Section 2.2.1, the PPU has 16 kB of memory available for both executable code and data storage. Therefore, parts of PPU programs are to be optimized for small size requirements. Knowing the memory consumption of objects also allows formulating memory-limit constraints. This section therefore describes storage consumption of the homeostasis and NSEM plasticity rules and the `vector_fractional` implementation.

3.9.1. `vector_fractional`

The `vector_fractional` type stores 16 fractional numbers in parallel. Each number is composed as described in Section 2.3.2 of an 8 bit integer digit and an arbitrary number of 7 bit fractional digits. The memory consumption of a `vector_fractional` instance with precision p in byte is therefore given as

$$\text{memory}(p) = 16 \cdot (1 + p) \cdot 8\,\text{bit} \tag{3.2}$$

Each fractional digit thereby only uses 7 bit of 8 bit because of carryover detection.

3.9.2. `vector_fractional_lookup`

The lookup table needs one byte per entry and digit precision. Memory consumption of a lookup table with n entries and fractional precision p is therefore given as

$$\text{memory}(n, p) = n \cdot (1 + p) \text{ byte}. \tag{3.3}$$

Because of 32 bit addressing on the PPU, this is to be padded to full 4 byte.

3.9.3. Homeostasis

The homeostasis plasticity rule stores the learning rate individually as 8 bit value per neuron. Measured rates are stored in a fractional vector with one fractional digit, the target rate is stored in a fractional vector with two fractional digits. Additionally a pointer to a random number generator instance is stored, which adds 4 byte. Because the vector instances need to be aligned to 16 byte to be accessible from the vector unit, this adds up to 16 byte to the memory consumption. Memory consumption for a homeostasis plasticity rule therefore is 208 bytes. The plasticity rule API and internal structure can be found in Listing 3. The measured rates could in principle be hold on the stack, but were chosen to be a member variable, because this easily allows the update algorithm to be performed for several excitatory and inhibitory rows with independently measuring the rates only once.

3.9.4. NSEM plasticity rule

The NSEM plasticity rule internal structure, API and implementation are shown in Listing 4. The NSEM plasticity rule is templated on the number p of fractional digits used in computation of the exponential factor in Equation (2.21). This factor is then multiplied with an 8 bit integer correlation measurement yielding a $p + 1$ fractional vector. The rule therefore holds a lookup table of precision p with 64 entries, one for each possible weight value. The regulatory factor is then of precision $p + 1$ and is to be subtracted from the temporary variable of the exponential factor multiplied with the correlation. The NSEM plasticity rule in addition holds a pointer to a random number generator instance, a vector mask and a container storing correlation offset values, adding 12 byte, which are again to be aligned to 16 byte because of the vectors stored. The memory consumption of the plasticity

rule therefore is given by

$$\text{memory}(p) = \underbrace{64 \cdot (1+p)}_{\text{lookup table}} + \underbrace{16 \cdot ((p+1)+1)}_{\text{regulatory term}} + \underbrace{16}_{\text{pointers}} \quad \text{byte} \tag{3.4}$$

The correlation container in addition holds a 8 bit offset value per synapse, i.e. 1024 byte for the whole chip. As described in (Spilger 2018), the mask information consumes 1 byte for the synapse vector address and 16 byte of binary mask information per synapse vector. For the whole chip, this amounts to 64 byte + 1024 byte in the worst case. If full synapse vectors are to be updated however, the binary mask vector is not stored, reducing to 1 byte per full synapse vector.

Although the masks memory consumption scales linearly with the number of synapses, this is assumed to pose no up-scaling problem, as the FPGA-memory access via the PPU in future chips allows preloading parts of the mask information during the remaining time of the update period. In addition, the plasticity rule scheme developed in (Spilger 2018) is agnostic of the mask implementation, which allows to improve the mask implementation without changing the plasticity rule's implementation.

The `vector_fractional` member variable storing the regulatory term holds the same value in every entry. In the future, this unnecessary overhead could be reduced to only hold the digit values and creating a temporary `vector_fractional` instance splatted with these values every update cycle.

The memory consumption evaluation presented show, that the memory limit allows for multiple plasticity rules implemented to be executed concurrently.

3.10. Repository and Continuous Integration

The source code to all experiments presented in this thesis can be found in the *git* repository `model-hw-nsem`[3]. The build flow is set up using a customized version of the *Waf* (Nagy 2018) build tool, `symwaf2ic`, which is used for all software projects developed in the group. In addition to building both the host and the PPU executables, *Doxygen* (Heesch 2018) source code documentation generation is implemented in the build flow.

To enable continuous integration (CI) of both the program build step as well as experiment

[3]To directly clone the repository, follow `ssh://git@gitviz.kip.uni-heidelberg.de/model-hw-nsem.git`. The source code can also be viewed in a web browser under `https://brainscales-r.kip.uni-heidelberg.de/projects/model-hw-nsem/repository`. The commit reproducing all experiments described in this thesis is `ec3961da7cebb00bd24f1ff94f06f94c92664a61`. Additionally used software is found in table B.1.

execution and result evaluation, a daily executed CI job[4] is implemented as part of this thesis using the group's *Jenkins* (*Jenkins* 2018) CI server. The job builds the executables, generates documentation, executes hardware tests, static code analysis for `C++`-code and the experiments (with larger step-size in sweeps in order to reduce hardware occupation time) described in this thesis. The code documentation as well as the experiment results are automatically published in the Web-GUI of the server.

This allows for easy detection of code degradation due to changes in software dependencies and periodically is supposed to replicate experiment results.

Source code for the host lies under `src/cc/<topic>/`, PPU program code is found in `src/ppu/<topic>/`. Every (logical) experiment can be executed by a shell script residing under `experiment/`, every experiment's evaluation resides under `evaluation/`.

[4]The CI job's results are found under `https://brainscales-r.kip.uni-heidelberg.de:11443/job/hw_model-hw-nsem/`.

Discussion

This thesis implements an neuromorphic adaption of *spike-based expectation maximization* on the HICANN-DLS version 2 prototype neuromorphic hardware featuring LIF neurons using the embedded general purpose processor PPU for learning rule update computation.

Building on the work of (Breitwieser 2015), the therein developed *neuromorphic spike-based expectation maximization* (NSEM), an adaption of spike-based expectation maximization geared towards implementation on neuromorphic hardware is implemented step-by-step on the neuromorphic hardware.

The network is comprised of a homeostatically stabilized winner-take-all cause layer consisting of stochastically firing LIF neurons receiving structured input encoded in spike-train rates from an input layer. It implements the concept of Expectation Maximization. Using a local STDP learning rule, the cause layer neurons' synapses are capable of inferring the spike-train input rate and the cause layer neurons can detect hidden causes in the input space in an unsupervised manner. The spiking behavior of the cause layer thereby constitutes the Expectation step, whereas the plastic synapses implement the Maximization step.

Adjustable implementation of the network requires that learning rule parameter are provided after compilation of PPU programs. In addition, computationally expensive factors are to be precomputed and stored on the PPU. Because of limited weight resolution on the hardware, and weight updates are additionally to be stochastically rounded. In order to fulfill these requirements, learning rule parameter communication to the PPU is improved, an arbitrary precision fractional number representation on the PPU is developed using its vector unit for acceleration and stochastic rounding to integers is implemented on that type.

The new hardware abstraction layer `haldls` is used in implementing host-computer parts of the network and experiments. Task scheduling is used on the PPU to allow for multiple plasticity rules (NSEM and homeostasis) to run simultaneously with different update cycles. Masking of the learning rules is implemented, which means the execution of the plasticity algorithms are restricted to an adjustable subset of all synapses. This plasticity rule execution masking is thereby implemented independent of the algorithm. In addition,

semi-automated calibration of the neuron activity for stochasticity and measurement of the parameters characterizing the exponentially decaying synapses' correlation measurement are implemented and successfully applied to experiments.

Each part of the NSEM network is shown to be working. A single synapse equipped with the NSEM local plasticity rule infers correct input rates. The implemented homeostasis adjusting the weight of excitatory and inhibitory background sources is capable of stabilizing neurons to a given target rate (below $\nu_{ik}^{\max}$) and to counteract changes in the excitability of the neurons. In combination, a neuron is shown to learn input rates of 5×5-pixel input pattern already with averaged correlation calibration. Unfortunately, fixed pattern variations in the correlation measurement of each synapse carry over to the inferred rates. Finally, the complete network combining the NSEM local plasticity rule and a homeostatically controlled winner-take-all cause layer consisting of three neurons is shown to be able to classify simple non-overlapping 5×5-pixel input pattern.

The performance analysis of the PPU learning rule implementation shows that usable update time periods of the NSEM plasticity rule in the order of seconds are reachable. Additionally, the `vector_fractional` type, used for sub-integer calculations of weight updates, shows the expected dependence of time consumption to precision for arithmetic operations. The memory consumption evaluation shows that it is possible to use several plasticity rules concurrently within a single PPU program.

In summary, it is thereby shown that the adaptions made in NSEM in (Breitwieser 2015) are applicable on the DLSv2. More generally, the DLSv2 is capable of emulating complex networks consisting of multiple different parts. The plasticity rule masking and scheduling on the PPU developed in (Spilger 2018) work in a real-world experiment. Furthermore, this implies that future systems composed of multiple chips can distribute parts of larger networks on a granularity smaller than a chip, i.e. execute multiple different network parts with differing plasticity on the same chip. Moreover, the `haldls` software layer (Electronic Vision(s) 2018) is shown to be suitable as basis for development and implementation of the experiments conducted.

To allow for combination and extension of the experiments conducted, each experiment is encapsulated in with a common interface allowing for combination and extension. The plasticity rule implementations for the PPU are developed with combinability, parameterization and restriction of execution in mind to enable extension and reuse in following experiments, outlined below.

Outlook

The NSEM learning rule application to the HICANN-DLS version 2 prototype and the capability of the platform and software environment to emulate a complex neural network has been shown in this thesis.

In the future, the ability of the implementation to classify more complex and in particular overlapping input pattern is the straight-forward next step to verify experiment wise. In addition, the time evolution of learning is to be investigated. Spike data is already available for the whole experiment duration. Especially the convergence rate of the classification can be analyzed. As during an experiment with enabled spike-router, no valid container data, e.g. weights, can be read during execution, their time course is to be recorded differently. A possible solution would be to split the experiment in multiple short time periods (e.g. $100\,\mathrm{s_{bio}}$) and to read out the weights at end of each part and thereby measure the time evolution of the weights.

A per-synapse correlation calibration factor could be implemented to counteract the fixed pattern variance. This could be compared to using averaged parameters in terms of convergence time or the ability to discern more complex, overlapping input pattern.

The measured temporal noise in the correlation measurements and calibration of the CADC is to be investigated. Also the behavior when using internal bias supply compared to the external supply with the DACs on the base board should be evaluated as this might reduce the crosstalk of the switching regulators.

Debugging the hang-ups of the homeostasis implementation, which alters the leak potential directly would permit comparison to the discussed implementation. Especially the adjustment range is expected to be higher with directly adjusting the leak and is to be compared to the current working implementation.

The Host-PPU communication of partial structures relies on knowing the internal placement of member variables. Instead of manually finding out these information, a method to automatically extract relative addresses of member variables is to be developed. For that, each structure-member's address relative to the structure's address is to be extracted, e.g. using `std::addressof`. However since `C++` does not directly implement reflection, iteration over all variables is still to be done manually at the moment. There exist compile-time re-

flection libraries that after registering allow automatic iteration over structure members.

For multi-platform experiments, in this case the host computer and the PPU, a messaging library with a common communication format for chip configuration is to be developed to easily distribute task information from one platform to the other. A long-term goal therefore is to port the API of the `haldls` containers to the PPU programs. Since the implementation however will differ because of different endianness and resources available, conversion of the containers between the platforms is to be developed. One idea proposed by (Müller 2018) is to use the developed serialization of the host's `haldls` containers with a common format to communicate container data cross-platform. This could also be expanded to multi-chip setups for inter-chip communication of configuration data.

The `haldls` playback creation only allows for time sequential adding of instructions. Experiments with multiple logically different events, i.e. different read, write or spike events however are simplified through grouping together the creation of similar events. The playback program creation process should therefore allow non-sequential adding of instructions. Arbitrary insertion would allow users to focus on logical clustering of instruction creation during experiment build. The already implemented temporal overlay of (different) spike-trains is therefore to be extended to be implemented for all playback program instructions. This would allow a more intuitive experiment creation from the author's point of view. A simple example usage scenario is specifying some readout instructions at defined times during experiment execution and afterwards overlay these with a pregenerated spike-train. The `haldls` software layer is however not supposed to supply this form of logical abstraction. A layer to be developed in the future building on top of `haldls` would greatly benefit from non-sequential playback program generation.

In the long term, an abstraction for PPU based offloading of plasticity computation is to be developed and integrated in a high level neural network API, e.g. PyNN (Davison et al. 2008). Especially the question how to retain as much performance and flexibility as possible while abstracting the actual implementation is to be answered.

While the network presented consists of two layers, an input and a hidden later, (Guo et al. 2018) present an approach with multiple hidden layers. This hierarchical SEM consists of several WTA-networks, stacked on top of each other. Each layer extracts more and more complex features from the input data, while still being unsupervised. This network could be implemented with different layers distributed over larger/multiple chips.

Bibliography

ABB Ltd (2013). *12V PicoTLynx^{TM} 6A: Non-Isolated DC-DC Power Module*. Website. URL: https://www.geindustrial.com/products/embedded-power/tlynx.

Bill, Johannes et al. (Aug. 2015). 'Distributed Bayesian Computation and Self-Organized Learning in Sheets of Spiking Neurons with Local Lateral Inhibition'. In: *PLOS ONE* 10.8, pp. 1–51. DOI: 10.1371/journal.pone.0134356. URL: https://doi.org/10.1371/journal.pone.0134356.

Billaudelle, Sebastian (2018). Personal communication.

Bishop, C. M. (2006). *Pattern Recognition and Machine Learning (Information Science and Statistics)*. Berlin, Heidelberg: Springer-Verlag. ISBN: 0387310738.

Breitwieser, Oliver (2015). 'Towards a Neuromorphic Implementation of Spike-Based Expectation Maximization'. Masterthesis. Heidelberg University.

Buesing, Lars et al. (2011). 'Neural Dynamics as Sampling: A Model for Stochastic Computation in Recurrent Networks of Spiking Neurons'. In: *PLoS computational biology* 7.11, e1002211–e1002211. ISSN: 1553-734X. DOI: 10.1371/journal.pcbi.1002211.

Crockford, Douglas (2018). *JavaScript Object Notation (JSON)*. Website. URL: https://www.json.org/.

Davison, Andrew et al. (Feb. 2008). 'PyNN: A Common Interface for Neuronal Network Simulators'. In: *Frontiers in neuroinformatics* 2, p. 11. DOI: 10.3389/neuro.11.011.2008.

Dempster, A. P., N. M. Laird and D. B. Rubin (1977). 'Maximum Likelihood from Incomplete Data via the EM Algorithm'. In: *Journal of the Royal Statistical Society. Series B (Methodological)* 39.1, pp. 1–38. ISSN: 00359246. URL: http://www.jstor.org/stable/2984875.

Electronic Vision(s) (Nov. 2018). 'electronicvisions/haldls 20181106_pspilger'. In: DOI: 10.5281/zenodo.1478481.

Friedmann, Simon (2013). 'A new approach to learning in neuromorphic hardware'. PhD thesis. Heidelberg, Univ., Diss., 2013.

Gerstner, Wulfram and Werner M. Kistler (2002). *Spiking Neuron Models: Single Neurons, Populations, Plasticity*. Cambridge University Press. DOI: 10.1017/CBO9780511815706.

Grant, W. Shane and Randolph Voorhies (2017). *cereal - A C++11 library for serialization*. Website. URL: http://uscilab.github.io/cereal/.

Guo, S. et al. (2018). 'Hierarchical Bayesian Inference and Learning in Spiking Neural Networks'. In: *IEEE Transactions on Cybernetics*, pp. 1–13. ISSN: 2168-2267. DOI: 10.1109/TCYB.2017.2768554.

Habenschuss, Stefan, Helmut Puhr and Wolfgang Maass (2013). 'Emergence of Optimal Decoding of Population Codes Through STDP'. In: *Neural Computation* 25.6. PMID: 23517096, pp. 1371–1407. DOI: 10.1162/NECO_a_00446. eprint: https://doi.org/10.1162/NECO_a_00446. URL: https://doi.org/10.1162/NECO_a_00446.

Heesch, Dimitri van (2018). *Doxygen*. Website. URL: http://www.stack.nl/~dimitri/doxygen/index.htm.

Heimbrecht, Arthur (Mar. 2017). 'Compiler Support for the BrainScaleS Plasticity Processor'. Bachelorthesis. Heidelberg University.

Hinton, G. E. (2007). 'Boltzmann machine'. In: *Scholarpedia* 2.5. revision #91076, p. 1668. DOI: 10.4249/scholarpedia.1668.

Hock, M. et al. (Sept. 2013). 'An analog dynamic memory array for neuromorphic hardware'. In: *2013 European Conference on Circuit Theory and Design (ECCTD)*, pp. 1–4. DOI: 10.1109/ECCTD.2013.6662229.

IBM (2010). *Power ISA^{TM} version 2.06 revision b*.

Jenkins (2018). Website. URL: https://jenkins.io.

Müller, Eric (2018). Personal communication.

Nagy, Thomas (2018). *Waf*. Website. URL: https://waf.io.

Petrovici, Mihai A. et al. (Oct. 2016). 'Stochastic inference with spiking neurons in the high-conductance state'. In: *Phys. Rev. E* 94 (4), p. 042312. DOI: 10.1103/PhysRevE.94.042312. URL: https://link.aps.org/doi/10.1103/PhysRevE.94.042312.

S. A. Aamir and Y. Stradmann et al. (2018). 'An Accelerated LIF Neuronal Network Array for a Large-Scale Mixed-Signal Neuromorphic Architecture'. In: *IEEE Transactions on Circuits and Systems I: Regular Papers*, pp. 1–14. ISSN: 1549-8328. DOI: 10.1109/TCSI.2018.2840718.

S. Friedmann and J. Schemmel et al. (Feb. 2017). 'Demonstrating Hybrid Learning in a Flexible Neuromorphic Hardware System'. In: *IEEE Transactions on Biomedical Circuits and Systems* 11.1, pp. 128–142. ISSN: 1932-4545. DOI: 10.1109/TBCAS.2016.2579164.

Schreiber, Korbinian (2018). Personal communication.

Spilger, Philipp (Aug. 2018). 'On parameterization and debugging of PPU programs'. Internship. Heidelberg University.

Stradmann, Yannik (2016). 'Characterization and Calibration of a Mixed-Signal Leaky Integrate and Fire Neuron on HICANN-DLS'. Bachelorthesis. Heidelberg University.

Wunderlich, Timo (Aug. 2016). 'Synaptic Calibration on the HICANN-DLS Neuromorphic Chip'. Bachelorthesis. Heidelberg University.

A. Parameter

A.1. Hardware Setup

Type	Identifier
Board name	Fantasio (v2 Board)
Chip ID	30
FPGA Flyspi ID	B291656

Table A.1.: Hardware setup used for all measurements.

A.2. Calibration

Parameter	Value [LSB]
`ramp_slope`	2470
`ramp_01`	2333
`ramp_bias`	2900
`v_bias`	1700
`syn_corout_bias`	700
`syn_v_reset`	3750

Table A.2.: CADC and source follower parameter results of calibration used for all measurements.

Parameter	Value
neuron	18
synapse driver	0
`syn_v_store`	1500 LSB
`syn_v_ramp`	900 LSB
`amp_calib`	1
`time_calib`	0
`pulselength`	8

Table A.3.: Correlation parameter used in single-synapse calibration measurements.

A.3. Experiment

Parameter	Value
neuron	18
synapse driver	0
`syn_v_store`	1500 LSB
`syn_v_ramp`	900 LSB
`amp_calib`	1
`time_calib`	0
`pulselength`	8
update period τ_{update}	$4\,\text{s}_{\text{bio}}$
post-synaptic rate ν_{post}	$30\,\text{Hz}_{\text{bio}}$
initial weight	0 LSB
experiment duration T_{exp}	$8096\,\text{s}_{\text{bio}}$
weight conversion factor w_{step}	0.02365
learning rate η	5×10^{-5}
correlation amplitude η	2.2824 LSB/spikepair
correlation time constant τ_{syn}	$24.339\,\text{ms}_{\text{bio}}$
correlation offset o	25 LSB
fractional digits exponential factor	4

Table A.4.: Parameter of single synapse NSEM learning rule experiment.

Parameter	Value
global synaptic strength `syn_v_bias`	1000 LSB
update period τ_{update}	$4\,\text{s}_{\text{bio}}$
target rate ν_k	$10\,\text{Hz}_{\text{bio}}$
experiment duration T_{exp}	$5000\,\text{s}_{\text{bio}}$
learning rate η_b	0.0156
membrane time constant τ_m	$1.0\,\text{ms}_{\text{bio}}$
exc./inh. synaptic time constant τ_{syn}	$30\,\text{ms}_{\text{bio}}$
background source rate ν_{bg}	$200\,\text{Hz}_{\text{bio}}$
reset potential	0.6 V
leak potential	0.5 V
threshold potential	0.8 V

Table A.5.: Parameter of single neuron homeostasis experiment.

Parameter	Value
experiment duration T_{exp}	$8096\,\text{s}_{\text{bio}}$
NSEM synapse initial weight	0 LSB
NSEM weight conversion factor w_{step}	0.0233
NSEM learning rate η	5×10^{-5}
NSEM fractional digits exponential factor	3
NSEM update period τ_{update}	$4\,\text{s}_{\text{bio}}$
NSEM synapse driver	0
homeostasis target rate ν_k	$28\,\text{Hz}_{\text{bio}}$
homeostasis learning rate η_b	0.0156
homeostasis background source rate o	$200\,\text{Hz}_{\text{bio}}$
homeostasis update period τ_{update}	$4\,\text{s}_{\text{bio}}$
neuron	18
`syn_v_store`	1500 LSB
`syn_v_ramp`	900 LSB
correlation `amp_calib`	1
correlation `time_calib`	0
correlation amplitude η	2.497 LSB/spikepair
correlation time constant τ_{syn}	$23.76\,\text{ms}_{\text{bio}}$
correlation offset o	calibrated per experiment run
`pulselength`	8
global synaptic strength `syn_v_bias`	1000 LSB
membrane time constant τ_m	$1.0\,\text{ms}_{\text{bio}}$
exc./inh. synaptic time constant τ_{syn}	$30\,\text{ms}_{\text{bio}}$
reset potential	0.6 V
leak potential	0.5 V
threshold potential	0.8 V

Table A.6.: Parameter of single synapse NSEM with homeostatically controlled neuron experiment.

Parameter	Value
experiment duration T_{exp}	$5000\,\text{s}_{\text{bio}}$
NSEM synapse initial weight	0 LSB
NSEM weight conversion factor w_{step}	0.019
NSEM null-cause rate ν_{i0}	$15\,\text{Hz}_{\text{bio}}$
NSEM learning rate η	5×10^{-5}
NSEM fractional digits exponential factor	3
NSEM update period τ_{update}	$4\,\text{s}_{\text{bio}}$
homeostasis target rate ν_k	$33\,\text{Hz}_{\text{bio}}$
homeostasis learning rate η_b	0.0156
homeostasis background source rate o	$250\,\text{Hz}_{\text{bio}}$
homeostasis update period τ_{update}	$1\,\text{s}_{\text{bio}}$
neuron	8
`syn_v_store`	1500 LSB
`syn_v_ramp`	900 LSB
correlation `amp_calib`	1
correlation `time_calib`	0
correlation amplitude η	2.43 LSB/spikepair
correlation time constant τ_{syn}	$21.3\,\text{ms}_{\text{bio}}$
correlation offset o	calibrated per experiment run
`pulselength`	8
global synaptic strength `syn_v_bias`	1000 LSB
membrane time constant τ_m	$1.0\,\text{ms}_{\text{bio}}$
exc./inh. synaptic time constant τ_{syn}	$30\,\text{ms}_{\text{bio}}$
reset potential	0.6 V
leak potential	0.5 V
threshold potential	0.8 V

Table A.7.: Parameter of 25 synapses NSEM with homeostatically controlled neuron experiment.

Parameter	Value
experiment duration T_{exp}	$5000\,\text{s}_{\text{bio}}$
homeostasis target rate ν_k	$7\,\text{Hz}_{\text{bio}}$
homeostasis learning rate η_b	0.0156
homeostasis background source rate o	$100\,\text{Hz}_{\text{bio}}$
homeostasis update period τ_{update}	$4\,\text{s}_{\text{bio}}$
neuron	22, 26, 29
number of inhibitory synapse drivers	3
inhibitory synapse weight	63 LSB
`pulselength`	8
global synaptic strength `syn_v_bias`	1000 LSB
membrane time constant τ_m	$1.0\,\text{ms}_{\text{bio}}$
exc./inh. synaptic time constant τ_{syn}	$30\,\text{ms}_{\text{bio}}$
reset potential	0.6 V
leak potential	using activation calibration to $\frac{1}{3}$ of maximal activation
threshold potential	0.8 V

Table A.8.: Parameter of three neuron winner-take-all network.

Parameter	Value
experiment duration T_{exp}	$10\,000\,\text{s}_{\text{bio}}$
NSEM synapse initial weight	0 LSB
NSEM weight conversion factor w_{step}	0.019
NSEM null-cause rate ν_{i0}	$15\,\text{Hz}_{\text{bio}}$
NSEM learning rate η	5×10^{-5}
NSEM fractional digits exponential factor	3
NSEM update period τ_{update}	$4\,\text{s}_{\text{bio}}$
homeostasis target rate ν_k	$10\,\text{Hz}_{\text{bio}}$
homeostasis learning rate η_b	0.0156
homeostasis background source rate o	$250\,\text{Hz}_{\text{bio}}$
homeostasis update period τ_{update}	$1\,\text{s}_{\text{bio}}$
neuron	8, 11, 14
`syn_v_store`	1500 LSB
`syn_v_ramp`	900 LSB
correlation `amp_calib`	1
correlation `time_calib`	0
correlation amplitude η	2.43 LSB/spikepair
correlation time constant τ_{syn}	$21.3\,\text{ms}_{\text{bio}}$
correlation offset o	calibrated per experiment run
`pulselength`	8
global synaptic strength `syn_v_bias`	1000 LSB
membrane time constant τ_m	$1.0\,\text{ms}_{\text{bio}}$
exc./inh. synaptic time constant τ_{syn}	$30\,\text{ms}_{\text{bio}}$
reset potential	0.6 V
leak potential	using activation calibration to $\frac{1}{3}$ of maximal activation
threshold potential	0.8 V

Table A.9.: Parameter of 25 synapses NSEM three neuron classification network.

B. Software

B.1. NSEM and homeostasis plasticity rule API/implementation

```
constexpr size_t num_slices = dls_num_columns / sizeof(vector uint8_t);
/**
 * \brief Homeostasis plasticity rule.
 *
 * The rule operates on either one or two synapse rows and can be excitatory and
 * inhibitory. The rate counters for each enabled neuron are compared to a
 * target rate issueing weight updates to align them.
 * The actual target rate therefore depends on the update period.
 * Stochastic fractional weight updates are used to allow for sub integer weight
 * settings and updates.
 */
template <class rng>
class homeostasis {
        rng* m_random;
        vector uint8_t m_eta[num_slices];
        vector_fractional<1> m_rate[num_slices];
        vector_fractional<2> m_rate_target[num_slices];
        vector uint8_t get_weight_update(size_t slice_num);
public:
        /// Initialize homeostasis rule.
        homeostasis(rng& random);

        /// Set rule parameters for one neuron.
        void set_parameters(uint8_t neuron, uint8_t eta, float rate, float factor);

        /// Read rate counter values.
        void get_rates();

        /// Update excitatory and inhibitory synapses with beforehand measured rates.
        void do_update(vector uint8_t const mask[num_slices],
                       uint8_t synapse_row_excitatory,
                       uint8_t synapse_row_inhibitory);

        /// Update excitatory and inhibitory synapses with measured rates.
        void run(
```

```
            vector uint8_t const mask[num_slices],
            uint8_t synapse_row_excitatory,
            uint8_t synapse_row_inhibitory);
};
```

Listing 3: Homeostasis API and internal structure.

```
namespace PlasticityRules {
namespace VectorRules {

/// NSEM plasticity rule using fractional vectors with stochastic draws to improve weight resolution.
template <class rng, size_t precision>
class StochasticSEM {
        rng* m_random;
        correlation* m_correlation;
        vector_fractional_lookup<num_hw_weights, precision> m_exp_lookup;
        vector_fractional<precision + 1> m_regulatory;

public:
        /// Create NSEM plasticity rule instance.
        StochasticSEM(rng* random, correlation* cor);

        /// Create SEM plasticity rule instance with rule parameters.
        StochasticSEM(float eta, float weight_step, float correlation_step, float lambda,
            float m, float tau_update, rng& random, correlation& cor);

        /// Set exponential term parameters.
        void set_exp_lookup(float eta, float weight_step, float correlation_step, float lambda);

        /// Set regulatory term parameters.
        void set_m(float eta, float weight_step, float m, float tau_update);

        /// Update algorithm implementation
        vector uint8_t kernel(vector uint8_t const& weights, vector uint8_t const& causal);

        /// Apply update algorithm operating masked on hardware values.
        void vector_rule(vector_synram_address const index, vector uint8_t const& mask);
};

template <class rng, size_t precision>
vector uint8_t StochasticSEM<rng, precision>::kernel(vector uint8_t const& weights,
    vector uint8_t const& causal) {
        // Use -= here, because it does not create an additional temporary fractional vector.
        return weights + draw((m_exp_lookup.lookup(weights) * causal) -= m_regulatory, m_random);
}

template <class rng, size_t precision>
void StochasticSEM<rng, precision>::vector_rule(vector_synram_address const address,
```

```
    vector uint8_t const& mask)
{
        // Accquire hardware weight values.
        vector uint8_t weights;
        asm volatile(
                "fxvinx %[weights], %[dls_weight_base], %[address]\n"
                : [weights] "=kv" (weights)
                : [dls_weight_base] "b" (dls_weight_base),
                  [address] "r" (address)
                :
        );
        // Accquire offset corrected causal correlation measurements.
        vector uint8_t causal = m_correlation->get_and_reset_causal(address);

        // Calculate new weights.
        vector uint8_t new_weights = saturate_weight(kernel(weights, causal));

        // Write calculated weights to hardware masked.
        asm volatile(
                "fxvcmpb %[mask]\n"
                "fxvsel %[new_weights], %[weights], %[new_weights], %[cond_gt]\n"
                "fxvoutx %[new_weights], %[base], %[address]\n"
                : [new_weights] "+&kv" (new_weights)
                : [weights] "kv" (weights),
                  [mask] "kv" (mask),
                  [base] "b" (dls_weight_base),
                  [address] "r" (address),
                  [cond_gt] "I" (__C_GT)
                :
        );
}

} // namespace VectorRules

// Plasticity rule for arbitrary mask using Mask
template <class Mask, class rng, size_t precision>
using StochasticSEM = MaskWrapper<VectorRules::StochasticSEM<rng, precision>, Mask>;

namespace Tagged {

// Plasticity rule for arbitrary mask using TaggedMask
template <class Mask, class rng, size_t precision>
using StochasticSEM = MaskWrapper<VectorRules::StochasticSEM<rng, precision>, Mask>;

} // namespace Tagged
} // namespace PlasticityRules
```

Listing 4: NSEM implementation.

B.2. Used Software

Repository / Description	Commit-ID / Identifier
singularity container	`/containers/stable/2018-11-02_1.img`
`gcc`	`7dec54439682a47e9c4380f79bbc93c697e1e053`
`waf`	`4dfe8ff9a36aa4f7e19cf6c5795eb674f0dcab2f`
`libnux`	`ed3f1176e03dd3da5e583af4b571c241a2148f12` (CS 3813)
`haldls`	`848d38d0be20a8534509e8d23bd69d664705ec05` (CS 3985), `752031ceb0a2d6a319dca540299094aee4bcfc21` (CS 4832)
`halco`	`d72eccbfcef010e723b78f0c675c6e2ded20d23f` (CS 4547)
`dls2calib`	`93b3872d5be1d78193ba5eed32e847a8775aa977`
`dls2calib-routines`	`f4e1d1f792953707049da977c45b1ae9bf7a3f99` (CS 4419)
`bitter`	`aa18d4a73a994a7e8590addbc40f6dc34a439b24`
`flyspi-rw_api`	`475f99bbf2f5bd4d5b065bd39ed9704e273ec748`
`hate`	`a31b694b47eef9bcdb2a04b088d86850cfe4b436`
`lib-boost-patches`	`5d74d1ddd3fa2e1da534c753e6fa58931fb8aed4`
`lib-rcf`	`2431b98495483d08d699ced28f631a8e4c51038b`
`logger`	`8355792fd3e591d08381575fcf5c4b2547b5fe3d`
`pygccxml`	`8ae9e19ae00c4152fa5a381eb9e663561c07345f`
`pyplusplus`	`064993baaea33e81c93655d79d9a1a6204b4acd0`
`pyublas`	`feaf60f2f920e30d588837dd6b0715eef23ed550`
`pywrap`	`158e5bb2b70f8c3f9b7d45e431c6105cb4dc301d`
`rant`	`acd5a3cd94fe91fa942e1da727e47025f00208c8`
`uni`	`e9135459841741e446e17f514e048fff74a8441f`
`ztl`	`2934a12003c14e08643cf2b4b3cbe7553e860f08`

Table B.1.: Software used. Unmerged changes are identified via the change set (CS) number additionally to the commit hash.

Acknowledgments (Danksagungen)

Professor Meier für die Betreuung meiner Arbeit.

Meinen Eltern für die Unterstützung und die Aufrechterhaltung eines bedingungslosen Zuhauses.

Oliver für die umfassenden Theorieerklärungen, die Hilfe bei Implementationsfragen, das unentwegte Verlangen nach ‘mehr Plots’ und alles andere.

Eric für das Streben nach besserem Code und einen bemerkenswerten Infrastruktursupport.

Christian für die langen Software-Diskussionen und die Hilfe jederzeit beim regelmäßigen Debuggen.

Yannik für all die Fehlersuche-Hilfe.

Sebastian für die Kalibration (der Neuronenschaltungen) des Chips in 30 Minuten (!) und das Teilen der Erfahrung in der Chipbenutzung.

Korbinian für die Hilfe bei der Inbetriebnahme eines neuen Setups, das Lehren von SMD-Löten, die Informationen zum CADC und die Hilfe bei Fragen um die Boards.

Aron für lange Sitzungen vor dem Oszilloskop.

Timo für Erklärung der Korrelationskalibration und die fruchtbaren Diskussionen über Theorie und Implementation.

Christian, Eric, Oliver und Yannik für das Korrekturlesen dieser Thesis.

Der Electronic Vision(s) Gruppe für eine tolle (Arbeits-)Athmosphäre und dem Container für das regelmäßige gute Grillen.

Erklärung

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