Department of Physics and Astronomy
Heidelberg University

Master Thesis in Physics
submitted by

Elias Arnold

born in Stuttgart (Germany)

2021

Biologically Inspired Learning in Recurrent Spiking Neural Networks on Neuromorphic Hardware

This Master Thesis has been carried out by Elias Arnold at the

Kirchhoff Institute for Physics in Heidelberg

under the supervision of

Dr. Johannes Schemmel

Biologically Inspired Learning in Recurrent Spiking Neural Networks on Neuromorphic Hardware

Typically, artificial *recurrent neural network*s (RNNs) are trained using *back-propagation through time* (BPTT). Since this gradient-based learning method requires propagating information back in time, it lacks biological plausibility. However, approximations to BPTT enable describing learning rules that only rely on information accessible forward in time. Such learning rules can also be found for *recurrent spiking neural network*s (RSNNs). Since spiking networks model the working principles of the biological brain, this enables biologically inspired learning, particularly with so-called *e-prop* learning algorithms. This thesis investigates the feasibility of *e-prop* inspired learning on the neuromorphic *BrainScaleS-2* (BSS-2) system, which emulates spiking neural networks in analog. Two distinct *e-prop* inspired learning rules are proposed that are compatible with BSS-2's intrinsic characteristics. As a first approach, a pattern-generation task is solved on the neuromorphic substrate via RSNNs with a spike-based learning rule. Further, a new *on-chip* plasticity rule is formulated that calculates weight updates with accumulated correlation measured inherently in parallel by all synapses on the BSS-2 system. The rule's capability to train RSNNs is demonstrated, and its implementation on the BSS-2 platform is detailed. Furthermore, software is presented, which provides modeling abstractions for hardware-in-the-loop and on-chip learning experiments.

Biologisch inspiriertes Lernen in rekurrenten spikenden neuronalen Netzwerken auf neuromorpher Hardware

Typischerweise werden künstliche rekurrente neuronale Netzwerke mit *back-propagation through time* (BPTT) trainiert. Diese gradientenbasierte Lernmethodik ist biologisch nicht plausibel, da diese Informationen in der Zeit zurück propagiert. Entsprechende Näherungen von BPTT ermöglichen es jedoch Lernregeln zu formulieren, die nur auf Informationen zugreifen, die vorwärts in der Zeit verfügbar sind. Solche Lernregeln können auch für *rekurrente spikende neuronale Netzwerke* (RSNN) gefunden werden. Da spikende Netzwerke Arbeitsprinzipien des biologischen Gehirns modellieren, wird dadurch, insbesondere mit sogenannten *e-prop* Lernalgorithmen, biologisch inspiriertes Lernen ermöglicht. Diese Arbeit untersucht die Umsetzbarkeit von *e-prop* inspiriertem Lernen auf dem neuromorphen *BrainScaleS-2* (BSS-2) System, welches in seinem analogen Kern spikende neuronale Netzwerke emuliert. Dafür werden zwei unterschiedliche *e-prop* inspirierte Lernregeln vorgeschlagen, die den Ansprüchen des BSS-2 Systems genügen. In einen ersten Ansatz wird gezeigt, dass eine spike-basierte Lernregel einen Mustergenerierungstask mit RSNNs auf dem neuromorphen Substrat lösen kann. Darauf aufbauend wird eine neue *on-chip* Plastizitätsregel vorgestellt, die mit akkumulierten Spikekorrelationsmessungen Gewichtsänderungen berechnet. Für diese Lernregel wird einerseits gezeigt, dass sie Lernen in RSNNs ermöglicht und zum anderen wird deren Implementierung auf der BSS-2 Plattform ausgearbeitet. Darüber hinaus werden Softwarelösungen aufgezeigt, die Lernen auf dem BSS-2 System in höheren Softwareschichten abstrahieren.

The Reader may pardon this long Discourse,
because the Subject so well deserved it,
and I wanted Art to make it shorter.

Edmund Bohun

Contents

1 Prologue

The human brain has an unparalleled capability to perceive and process complex information. As a highly efficient neural engine, it has evolved to model the environment and thus shapes human behavior. Especially, its ability to perform operations of high complexity drives scientists to gain a deeper understanding of its functionality — which still remains widely elusive — and to learn from its operating principles for application in artificial counterparts.

In the past years, the field of machine learning has been incredibly successful, primarily by arranging the most basic idea of neural components, the perceptrons [Rosenblatt, 1958], into large *artificial neural network*s (ANNs). ANNs have become a fairly capable tool for solving advanced tasks comprising pattern recognition, classification, and reinforcement learning [LeCun et al., 2015; Silver et al., 2016]. A large contribution to the success of ANNs is due to the increase of computational resources in recent decades, accelerating the training of networks with reasonable scale by orders of magnitude [Xu et al., 2018]. This comes at the expense of enormous energy consumption [Strubell et al., 2019].

Neural networks in the brain work fundamentally differently than ANNs. Biological neurons are dynamic entities, continuously evolving in time. They form networks by interconnecting via synapses and, in contrast to most ANNs, communicate information mainly via the timing of their all-or-nothing spike events [Alberts et al., 1994; Gerstner et al., 2002; M. Petrovici, 2015]. These working principles are modeled with *spiking neural network*s (SNNs) [Gerstner et al., 2014], which are promising candidates for energy-efficient *neuromorphic* hardware implementations [Pfeiffer et al., 2018] of neural networks, given their (usually) sparse event-based nature.

Such brain-inspired neuromorphic hardware systems [Mead, 1990; Young et al., 2019], like Intel's Loihi [Davies et al., 2018], IBM's TrueNorth [Akopyan et al., 2015], SpiNNaker [Painkras et al., 2013], Neurogird [Benjamin et al., 2014], or DYNAPs [Moradi et al., 2018] are developed by many different research collaborations and companies. Due to its mixed-signal architecture, the accelerated neuromorphic *BrainScaleS-2* (BSS-2) system [Schemmel et al., 2020; Billaudelle et al., 2020] developed at the Heidelberg University within the *Human Brain Project* (HBP) [*Human Brain Project* 2021] collaboration is particularly interesting. This system implements the dynamics of adaptive exponential leaky integrate-and-fire neurons [Brette et al., 2005] in analog circuits and communicates spike events between neurons digitally at high bandwidth. Furthermore, configurable synapse matrices on its *analog network core* (ANNCORE) process synaptic activity and collect correlation information inherently in parallel, thereby enabling learning on the BSS-2 system. Notably, its ANNCORE is tightly coupled to two digital embedded *single instruction multiple data* (SIMD) general-purpose processors with the ability to alter

synaptic weights and neural parameters, and, thus, perform local plasticity *on-chip* at high speed [Friedmann, 2013]. Learning on the BSS-2 system has been successfully demonstrated for different applications [Schreiber, 2020; Weis et al., 2020; Billaudelle et al., 2021; Cramer et al., 2020; T. Wunderlich et al., 2019].

The underlying topology of neural networks in the brain is different from (deep) feed-forward networks since the brain is "essentially a multitude of superimposed and ever-growing loops between the input from the environment and the brain's outputs" [Buzsáki, 2009]. Experimental findings support that recurrent connections in biological networks are indeed the fundamental characteristic of circuits in neural tissue [Kandel et al., 2000]. In fact, recurrently connected networks allow the brain to propagate information over time and thus incorporate memory in tasks it has to perform [Bellec et al., 2019]. However, it is not clear how the brain enables plasticity in such *recurrent spiking neural network*s (RSNNs) [Bellec et al., 2019]. In machine learning, artificial *recurrent neural network*s (RNNs) are usually trained with *back-propagation through time* (BPTT) [Werbos, 1990]. From a biological perspective, this is implausible since it requires propagating information backward in time to perform synaptic plasticity. A major finding of Bellec et al. [2019] is that the gradient for BPTT can be factorized into a temporal sum over products of *eligibility traces* that can be computed forward in time and *learning signals* depending on the network's error. Learning algorithms emerging from this factorization that allow for an *online* merging of learning signals and eligibility traces forward in time — by providing suitable approximations — are referred to as *e-prop* [Bellec et al., 2019]. Most importantly, Bellec et al. [2019] find online plasticity rules for learning in RSNNs that exhibit an appealing biological interpretation. Since these plasticity rules calculate weight updates simultaneously to the forward pass, they are promising candidates for neuromorphic on-chip implementations.

This thesis demonstrates *e-prop*-inspired learning in RSNNs on the BSS-2 system by solving a pattern-generation task inspired by [Bellec et al., 2019, page 28]. Since BSS-2's architecture has different requirements for *on-chip* implementable learning rules, the rules derived in [Bellec et al., 2019] are adapted. As a first approach, the eligibility traces are replaced by approximated spike-based versions [Bellec et al., 2019]. The resulting learning rule, called *spike-based eligibility propagation* (s-prop), enables learning with BSS-2 in the loop [Schmitt et al., 2017], where the forward pass is emulated on the neuromorphic substrate and weights are optimized off-chip. After testing s-prop in simulation, experiments on BSS-2 show that it enables RSNNs to learn a pattern-generation task surprisingly well. In a second approach, the actual on-chip implementation is tackled. As this comes with further requirements, an on-chip learning rule is derived based on accumulated spike correlation information, giving it the name: *Neuromorphic Accumulative Spike Propagation* (NASProp). Simulations verify the rule's capability to train RSNNs. In a single synapse test setup, the on-chip implementation shows the desired learning behavior.

In the end, a crucial component of this thesis is the development of software, abstracting the BSS-2 system for experiment execution in the high-level software framework `PyTorch`

[Paszke et al., 2019]. Therefore, all training-related experiments on BSS-2 are controlled and evaluated within the `PyTorch` learning environment.

1.1 Thesis Outline

In the following, the structure of this thesis is outlined. Chapter 2 gives an overview of computational neuroscience with a brief insight into the biological neuron and a mathematical description of it, followed by an introduction to RSNNs for which the e-prop learning framework is described. In Chapter 3, the neuromorphic BSS-2 system is explained, and necessary properties are elaborated. As a first result, the developed software for experiments on analog hardware is given in Chapter 4. Here, the abstraction of the BSS-2 system for experiments in RSNN in high-level software framework `PyTorch` is explained. This encompasses the software setup for in-the-loop and on-chip learning and the description of an event routing algorithm. First experiments with the approximated s-prop learning rule are conducted in Chapter 5. This includes a discussion of the s-prop learning rule, its verification in simulated networks, and also experiments on the actual neuromorphic hardware. The challenge of a full on-chip implementation is tackled in Chapter 6. Therefore, in this chapter, the adjusted NASProp learning rule is derived. Motivated by simulations, verifying its feasibility, the learning rule is implemented on-chip. This implementation is exemplified in a single synapse experiment, after which full on-chip training is approached. Results are summarized and briefly discussed in Chapter 7. This also includes a short outlook.

2 Theoretical Background

Computational Neuroscience is an interdisciplinary field of research that combines computer science and the biological understanding of the brains in living creatures, therefore, reaching a wide variety of knowledge and methods. This chapter starts with a biological description of the brain's most basic components, the neurons, and will outline a possible mathematical description of it. Based upon this knowledge, *recurrent neural network*s (RNNs) will be considered from a general perspective. This will ease the transition to *recurrent spiking neural network*s (RSNNs), for which a biologically inspired learning algorithm is discussed.

2.1 Computational Neuroscience in a Nutshell

The biological brain of living creatures is fascinating for many reasons. While it performs complex computations and processes vast amounts of data, its energy consumption remains very low. Therefore, the brain is not just subject to computational neuroscience but also acts as a paragon to neuromorphic computing and, in particular, to Machine Learning with (deep) *artificial neural network*s (ANNs). These disciplines benefit from each other. While biological experiments and paradigms improve theoretical models and learning algorithms, computational simulations can refine understanding and predict biological behavior.

In essence, the brain is a composition of neurons connected via synapses, forming a neural network. For modeling such networks computationally, it is crucial to align the mathematical description of the neurons with their biological counterpart. Hence, the following will give an overview of the biological neuron based on M. A. Petrovici [2015] and refers to [Alberts et al., 1994] for the biological descriptions in Section 2.1.1 and to [Gerstner et al., 2002] for the mathematical formulations in Section 2.1.2, if not referenced otherwise.

2.1.1 Biological Neuron

Neurons are cells consisting of plasma membranes separating the cell's interior from the environment they are embedded in. Since this membrane is basically a lipid bilayer, which allows charge-free small molecules to pass, while being impermeable to charge carriers, like polar molecules and ions, it can be considered a capacitor. This alone, however, would be a static system since no charge is exchanged between the neuron's

interior and the outside. In addition to the lipid bilayer, the membrane includes different proteins able to transfer specific ions from the neuron's inside to the outside and vice versa, enabling a dynamic behavior of the potential between the neuron's interior and the environment. In fact, only a few thousand ions crossing a membrane area of $1\,\mu\text{m}^2$ are enough to change the membrane potential by a magnitude of $\mathcal{O}(10\,\text{mV})$.

The main reason for the cross-membrane potential is the Na^+-K^+ pump, realized by a protein in the neuron's membrane. This protein transfers Na^+ ions from the interior of the cell to the outside, while transporting K^+ ions in the other direction into the neuron. However, the pumping mechanism is unbalanced; more Na^+ ions are transported to the outside than K^+ ions to the inside, leaving the neuron's interior charged negatively. This increases the ionic gradient, which in turn decreases the ion flow due to pumping, until a equilibrium state is reached. Thus, the membrane potential in this state is referred to as the leak or resting potential. Note that this potential is pulled down further by the K^+ channel, allowing only potassium ions to pass. Due to the K^+ excess in the neuron's interior, created by the Na^+-K^+ pump, potassium ions use the K^+ channel to flow outwards. This increases the deficit of positive ions in the neuron, yielding a resting potential of typically $-70\,\text{mV}$.

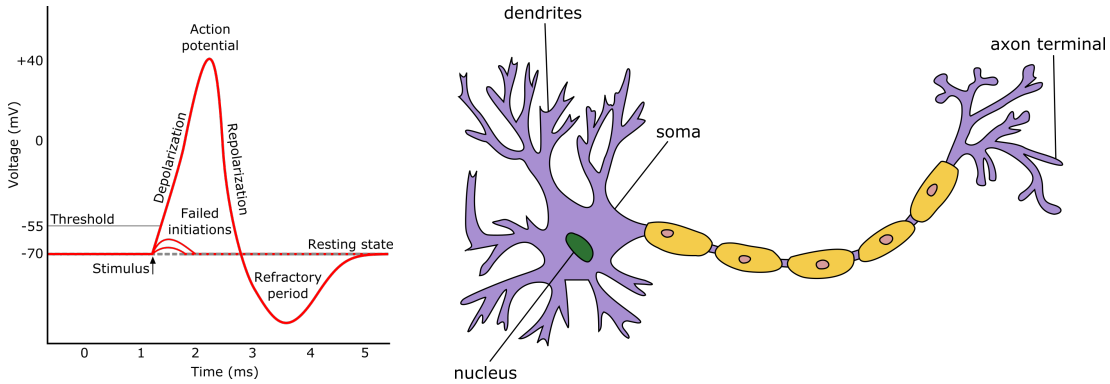
Nonetheless, several more protein channels and ions participate in the process described before. However, since corresponding ion concentrations effectively act as batteries with potential E, and the leakage channels can be understood as conductances g, merging each, the specific batteries and conductances for different ions into single representative ones, a neuron can be simplified described by an RC circuit. Hence, using Ohm's law, the membrane dynamic of a neuron is described by the *ordinary differential equation* (ODE)

$$C_\text{m}\frac{\text{d}v}{\text{d}t} = -g_\text{l}\,(v - E_\text{l}) + I. \tag{2.1}$$

Here E_l is the leak potential, g_l the leak conductance, C_m the membrane capacitance, and v the membrane potential. The current I is an external stimulus. The membrane time constant is given by $\tau_\text{m} = {}^{C_\text{m}}\!/_{g_\text{l}}$. Equation (2.1) describes the fundamental dynamics of the neuron model described in Section 2.1.2.

Action Potential

So far, the processes discussed apply to cells in general. However, excitable cells — like neurons — are endowed with additional voltage-gated ion channels providing the capability to change their membrane potential differently. These types of channels are also ion-specific but adjust their functionality and permeability depending on the membrane potential v_m and give rise to the phenomenon of *action potential*s (APs). That is, as soon as the neuron's membrane potential exceeds a threshold potential ϑ, the membrane depolarizes quickly, resulting in a sharply increasing potential followed by repolarization of the membrane, pulling the potential below the leak potential. The potential remains in this state for a time period, called the *refractory period*, in which the neuron is very unlikely to create a subsequent AP. Figure 2.1a illustrates this process. The APs are often referred to as spikes and describe the act of a neuron firing.



(a) Action potential **(b)** Neuron schematics

Figure 2.1: **(a)** Illustration of an action potential. If the neuron's membrane potential exceeds a threshold the membrane potential increases sharply followed by quick reploarization and a dropping potential. Thereafter, the neuron enters the refractory period. Image taken from [Iberri, 2007]. **(b)** Schematic of a neuron. It consists of a soma, which receives input over its dendrites and outputs its action potential along the axon. The spike event is distributed to subsequent neurons by the axon terminals. Image taken from [Jarosz, 2009].

APs enable neurons in a network to exchange information. In fact, spikes are the primary information carrier in the brain. Hence, a neuron (usually) only provides information at the occurrence of an AP and, in principle, not by other (usually slower) chemical processes. The advantage is that information distribution and processing in brains is very fast while consuming little energy. The following section will give an overview of how these spike events are communicated in a neural network.

Synapses

Figure 2.1b depicts the structure of a typical neuron. It consists of the cell, described in the previous sections, referred to as soma. If the neuron exhibits an AP, it travels from the soma through the axon to the axon terminals. These terminals connect to the dendrites of subsequent neurons via synapses. A pre-synaptic neuron releasing an AP triggers a *post-synaptic potential* (PSP) in the post-synaptic neuron through the corresponding synapse. This PSP is understood as the temporal modification of post-synaptic neuron's membrane potential due to the pre-synaptic spike event. Typically two distinct classes of PSPs exist. A PSP pushing the membrane potential upwards has an *excitatory* effect and is called *excitatory post-synaptic potential* (EPSP). Vice versa, if the membrane potential is decreased, the PSP is *inhibitory* and called *inhibitory post-synaptic potential* (IPSP). In that way, pre-synaptic neuron partners can time their action potentials such that the neuron's membrane potential is modified in a way that the neuron exhibits a desired behavior, e.g., spike. Note that a neurons' spatial structure is very influential on how the received spiking information is processed. Neuron models neglecting the spatial extent are called point-like models and are usually easier and more efficient to implement in software.

The biological functionality of synapses, categorized into chemical and electronical ones, will not be discussed in detail here. However, to motivate the modeling of PSPs in Section 2.1.2, the functionality of chemical synapses is described briefly. Roughly, a pre-synaptic AP causes voltage-gated calcium channels in the axon terminal to open, resulting in an influx of Ca^{++} ions. In turn, this releases *neurotransmitters* to the synaptic cleft — the space between the axon terminal and the dendrite of the target neuron. The neurotransmitters activate ligand-gated ion channels in the target neuron's membrane, such that ions can move across the membrane and thus change its membrane potential. The neurotransmitters activate the channels only temporarily until they are removed and thereby reduce the net influx of ions steadily. This is often modeled by an exponentially decaying current onto the membrane at the occurrence of a pre-synaptic spike event.

The previous sections provide a rather simplistic description of neurons from a biological perspective. Note, however, the neurons' biological and chemical behavior and how they interact are usually complex processes of which not all are understood yet. Nevertheless, the properties and functionalities discussed so far will be sufficient to reason a simple mathematical neuron model in the following.

2.1.2 Leaky-Integrate-and-Fire Model

One of the most basic neuron models has been introduced by Lapicque [1907] and was later renamed to *leaky integrate-and-fire* (LIF) model. While this model introduces strong simplifications of the biological neuron, it describes the dynamics of the membrane potential sufficiently well for a large field of applications [Bellec et al., 2019; Breitwieser, 2015; Kanya, 2020] and, therefore, has great relevance. In particular, the model's simplicity enables an efficient implementation in software and on hardware, making it a convenient choice for neuroscientific simulations and neuromorphic hardware implementations.

The LIF model assumes that the neuron has no spatial extent and is considered point-like [Gerstner et al., 2014]. This implies that the synaptic currents onto the membrane are not delayed and affect the potential immediately. In contrast, biological observations show that neurons in the brain differ much in their size and layout. By strategically placing inhibitory and excitatory inputs on the dendrites relative to the soma, neurons exploit the spatial component to exhibit non-linear behavior in their membrane potential, allowing more complex operations [Gerstner et al., 2014]. However, modeling this spatial component is computationally expensive and is therefore neglected here.

Further, biological experiments find that individual action potentials do not vary much and have very similar shapes. This suggests that action potentials do not propagate information themselves, but the timing of the spike events holds information. Hence, it seems appropriate for the LIF model to consider spikes as binary stereotyped events, such that explicit modeling of action potentials is redundant [Gerstner et al., 2014].

A LIF neuron is said to "send out" a spike $z(t)$ at time $t = t^{\mathrm{s}}$ if the membrane potential $v(t)$ exceeds a certain threshold ϑ,

$$z(t) = \begin{cases} 1 & \text{if } v(t) \geq \vartheta, \\ 0 & \text{else,} \end{cases} \tag{2.2}$$

after which the membrane potential is reset to a *reset potential* v_{r} for a *refractory period* τ_{ref},

$$v(t) = v_{\mathrm{r}} \quad \forall t \in (t^{\mathrm{s}}, t^{\mathrm{s}} + \tau_{\mathrm{ref}}] \,. \tag{2.3}$$

The dynamics of the membrane potential v of a LIF neuron with membrane capacitance C_{m} is described by a first order differential equation,

$$C_{\mathrm{m}} \frac{\mathrm{d}v}{\mathrm{d}t} = -g_{\mathrm{leak}} \left(v - v_{\mathrm{l}}\right) + I_{\mathrm{syn}}, \tag{2.4}$$

where a synaptic input current I_{syn} is integrated onto the membrane modeling the unbalanced ion concentration in the neurons membrane on present synaptic input. The membrane potential is permanently striving back towards its equilibrium resting potential v_{l} with a *leakage conductance* given by g_{l}. The membrane time constant τ_{m} of a LIF neuron is then described by

$$\tau_{\mathrm{m}} = \frac{C_{\mathrm{m}}}{g_{\mathrm{l}}} \tag{2.5}$$

and defines its temporal extent. Since later chapters will refer rather to the membrane time constant than to the leakage conductance equation 2.4 is rewritten as

$$\frac{\mathrm{d}v}{\mathrm{d}t} = -\frac{1}{\tau_{\mathrm{m}}} \left(v - v_{\mathrm{l}}\right) + \tilde{I}_{\mathrm{syn}}, \tag{2.6}$$

where $\tilde{I}_{\mathrm{syn}}$ absorbs the constant $1/C_{\mathrm{m}}$. This equation describes the fundamental dynamics of all neurons considered from here on.

Synaptic Input

So far the synaptic current I_{syn} was not specified in detail but was assumed as an arbitrary function of time. In neural networks, a neuron j usually receives pre-synaptic spike events from a pre-synaptic partner i while in turn sending its post-synaptic events to neurons deeper in the network, acting itself as a pre-synaptic partner. A temporal sequence of incoming events $z_i(t)$ from neuron i is often referred to as a *spike train*,

$$z_i(t) = \sum_s \delta \left(t - t_i^s\right), \tag{2.7}$$

where δ denotes the δ-distribution and t_i^s the time neuron i spiked. Every pre-synaptic event triggers a PSP whose shape is defined by a kernel ϵ that convolves the input spike train z_i, and thus modeling the ion influx triggered by a chemical synapse. The synaptic input current of neuron j is then given by the sum over the weighted PSPs of all pre-synaptic partners,

$$I_{j,\mathrm{syn}}(t) = \sum_i w_{ji} \left(\epsilon \star z_i(t)\right), \tag{2.8}$$

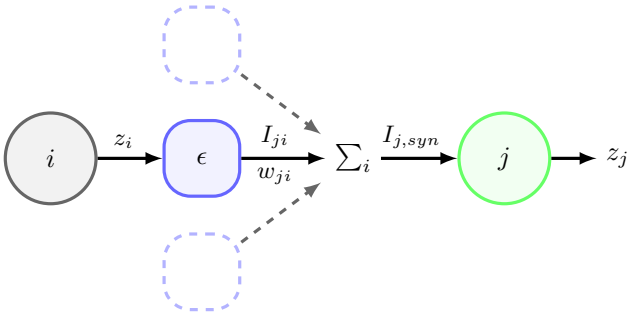


Figure 2.2: Computational graph of a synapse. The pre-synaptic spike trains z_i are convolved by kernels ϵ, modeling the corresponding synaptic current I_{ji}. Neuron j integrates the weighted sum $I_{j,\text{syn}}$ of the each synaptic currents I_{ji} onto its membrane. This leads to a change in the membrane potential inducing a spike eventually.

where w_{ji} is the synaptic strength. A computational graph is given in Figure 2.2.

The kernel ϵ is a synapse-specific property and depends on the extent to which biology needs to be mimicked. One possibility is given by a difference of exponential functions,

$$\epsilon(t) = A\Theta(t)\frac{1}{\tau_{\text{rise}} - \tau_{\text{fall}}}\left[\exp\left(-\frac{t}{\tau_{\text{rise}}}\right) - \exp\left(-\frac{t}{\tau_{\text{fall}}}\right)\right], \tag{2.9}$$

which describes the fast increasing ion influx into the post-synaptic neuron at a pre-synaptic AP by the first exponential term (neurotransmitters arrive at post-synaptic neuron) and the subsequent decreasing influx by the second exponential term (neurotransmitters are removed). The time constant τ_{rise} adjusts how fast the influx increases and τ_{fall} how fast the influx decreases. Here, A is an arbitrary scaling factor and Θ the Heaviside step function. Assuming the arrival of neurotransmitters to be infinitely fast, such that $\tau_{\text{rise}} \longrightarrow 0$, gives the commonly used single exponential kernel

$$\epsilon^{\text{single}}(t) = A'\Theta(t)\exp\left(-\frac{t}{\tau_{\text{syn}}}\right), \tag{2.10}$$

with $\tau_{\text{syn}} = \tau_{\text{fall}}$ and A' the scaling factor absorbing $1/\tau_{\text{fall}}$. Inserting the kernel in Equation (2.10) into Equation (2.8) and choosing $A' = 1$ gives the synaptic input current

$$I_{j,\text{syn}}(t) = \sum_i \sum_s w_{ji}\Theta(t - t_i^s)\exp\left(-\frac{t - t^s}{\tau_{\text{syn}}}\right). \tag{2.11}$$

This equation describes the total current onto neuron j's membrane at time t. Note, these types of synapses are called current-based. Another approach is conductance-based synapses which claim to be biologically more plausible and allow describing stochastic neuron dynamics in the *high-conductance* state. This, of course, comes with a higher computational cost and will not be discussed.

Neglecting all temporal effects in a synapse, the most simple dynamic is given by a δ-kernel

$$\epsilon^{\delta}(t) = A\delta(t). \tag{2.12}$$

This leads leads to an input current

$$I^{\delta}_{j,\text{syn}}(t) = \sum_i \sum_s w_{ji}\delta(t - t_i^s) = \sum_i w_{ji} z_i^t, \tag{2.13}$$

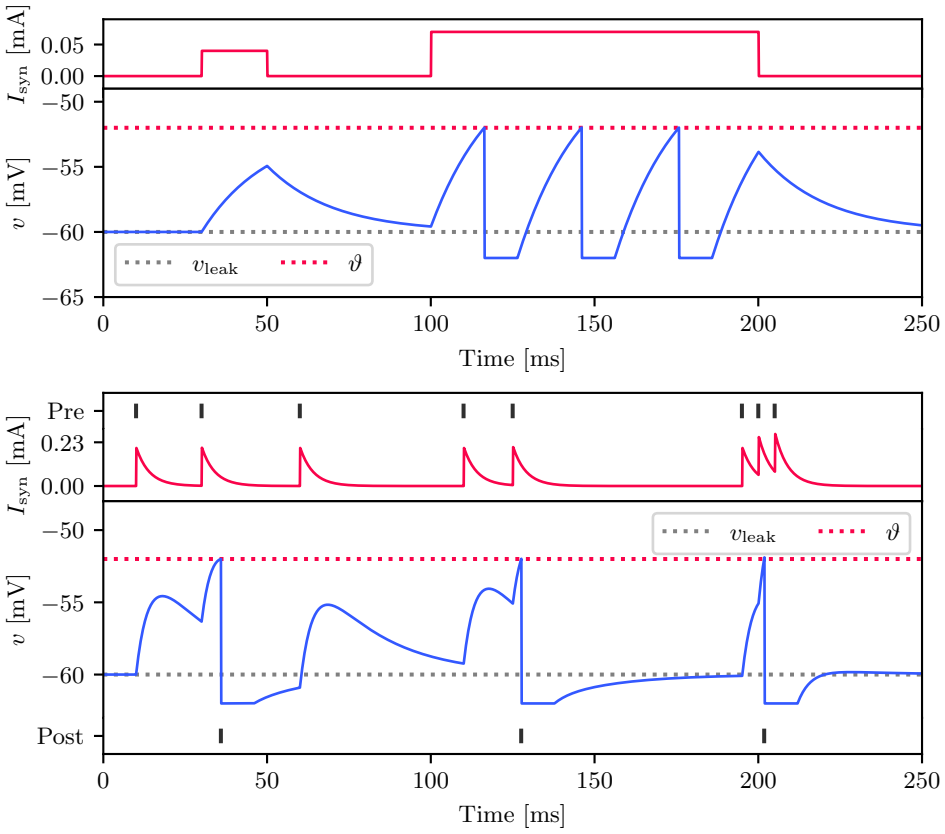


Figure 2.3: Example membrane traces simulated with the LIF model. The upper plot shows the membrane dynamic with a piece-wise constant input current (red). As the potential (blue) reaches the threshold ϑ, the neuron spikes and the membrane is clamped to the reset potential for a refractory period. In the lower plot the neuron receives input current from a single exponential synapse kernel (red), triggered by pre-synaptic spikes.

with the consequence that a pre-synaptic event changes the membrane potential only at spike-time t^s without causing any further current onto the membrane in the future. Of course, this is a rather rough approximation of chemical synapses and lacks biological plausibility.

Numerical LIF Neuron

To make the LIF model described by Equation (2.6) accessible in software it needs to be integrated numerically in discrete time. Therefore a continuous time sequence of length T is split into $N \in \mathbb{N}$ equidistant time steps δt, such that $T = N\delta t$. Then the time t_n at time step n is defined by $t_n = n\delta t$ with $n \in [0, N]$. One possible approach to define the LIF model on this discrete time lattice is given by (see [Gerstner et al., 2014]),

$$v^{n+1} = \alpha v^n + (1 - \alpha)v_{\text{l}} + I^n. \tag{2.14}$$

Here α denotes a constant decay

$$\alpha = \exp\left(-\frac{1}{\tau_{\mathrm{m}}}\delta t\right), \tag{2.15}$$

and $I^n = \tilde{I}(t_n)$ the potential modification due to synaptic input current at time t_n. If the membrane potential v^n crosses the threshold ϑ at time step n, the neuron sends out a spike, such that $z(t_n) = z^n = 1$, after which the membrane potential is pulled towards the reset potential v_{r} for

$$n_{\mathrm{ref}} = \frac{\tau_{\mathrm{ref}}}{\delta t} \in \mathbb{N}_0 \tag{2.16}$$

time steps, keeping the neuron in its refractory period. An example is given in figure 2.3.

For a PSP with a single exponential kernel, I^n is calculated in a similar fashion by

$$I^{n+1} = \gamma I^n + \sum_i w_i z_i^n \tag{2.17}$$

with $z_i^n \in \{0, 1\}$ indicating whether pre-synaptic neuron i spiked at time t_n, the synaptic strength w_i (absorbing $1/C_{\mathrm{m}}$ from $\tilde{I}$), and γ being the synaptic decay constant

$$\gamma = \exp\left(-\frac{1}{\tau_{\mathrm{syn}}}\delta t\right). \tag{2.18}$$

All simulations made in the following chapters rely on these definitions.

2.2 Introduction to Recurrent Neural Networks

Recently ANNs have gained great attention in a vast amount of different scientific and commercial areas and have become arguably one of the most powerful machine learning tools. Especially very popular architectures of ANNs like *feedforward neural network*s (FNNs) or *convolutional neural network*s (CNNs) have been highly successful in plenty of different tasks, including classification, reinforcement learning, and object detection [Silver et al., 2016; Cai et al., 2018]. However, for machine learning tasks that involve temporal or ordinal sequences, RNNs are often superior. Since this thesis deals with learning in RSNNs, it seems crucial to elaborate a general mathematical framework that describes RNNs to ease the transition to spiking networks. A brief look at the basic structure of FNNs based on Goodfellow et al. [2016, Chapter 6] will help define RNNs and emphasize the differences.

Basically FNNs are (mostly very complex) mappings that receive an input vector $\mathbf{x} \in \mathbb{R}^D$, with D the feature dimension, and map it to an output vector $\mathbf{y} \in \mathbb{R}^{N^{L-1}}$, with N^{L-1} the output dimension, via a function Φ — called *model* — which is parameterized by a parameter vector θ,

$$\mathbf{y} = \Phi(\mathbf{x}, \theta). \tag{2.19}$$

Deep FNNs are typically a composition of L layers where each layer l receives the N^{l-1} dimensional output $\mathbf{x}^{(l)} = \mathbf{y}^{(l-1)} \in \mathbb{R}^{N^{l-1}}$ from the previous layer $l-1$ as input and performs a non-linear operation ϕ_l,

$$\mathbf{y}^{(l)} = \phi_l\left(\mathbf{x}^{(l)}\middle|\theta^l\right), \tag{2.20}$$

parametrised by θ^l, such that the whole network is then described by

$$\mathbf{y} = \phi_{L-1} \circ \cdots \circ \phi_l \circ \cdots \circ \phi_0(\mathbf{x}) \tag{2.21}$$

with parameters $\theta = \{\theta^l\}_{l=0}^{L-1}$. The first layer $l = 0$ is referred to as the *input* layer and the last layer $l = L-1$ is the *output* layer. All layers in between are *hidden* layers. While the input layer usually only provides the input data point $\mathbf{x}$ with $\phi_0 = \mathrm{id}$, a hidden layer in a *multi-layer perceptron* (MLP) consists of a set of N^l neural units $\{n_i^l\}_{i=0}^{N^l-1}$ — called *perceptrons* [Rosenblatt, 1958] — which output a weighted sum of the inputs,

$$o_i^{(l)} = \sum_{j=0}^{N^{l-1}} w_{ji}^{(l)} x_j^{(l)} + b_i^{(l)}, \tag{2.22}$$

where N^{l-1} is the number of units in the previous layer, $w_{ji}^{(l)}$ the synaptic strength from neuron j in layer $l-1$ to neuron i in layer l and $b_i^{(l)}$ a bias. For the whole set of neurons in the layer this is a simple linear projection,

$$\mathbf{o}^{(l)} = \mathbf{W}^{(l)}\mathbf{x}^{(l)} + \mathbf{b}^{(l)}, \tag{2.23}$$

where $\theta^l = \{\mathbf{W}^{(l)} \in \mathbb{R}^{N^{l-1}\times N^l}, \mathbf{b}^{(l)} \in \mathbb{R}^{N^l}\} = \{\{w_{ji}^{(l)}\}, b_i^{(l)}\}$ is the weight matrix and the bias vector. To introduce non-linearity the output $\mathbf{o}^l$ is activated by a (usually) non-linear activation function a, giving

$$\mathbf{y}^{(l)} = a\left(\mathbf{W}^{(l)}\mathbf{x}^{(l)} + \mathbf{b}^{(l)}\right). \tag{2.24}$$

Among others, a widespread activation function is, for instance, the *rectified linear unit* (ReLU) [Nair et al., 2010]. The choice of ϕ_{L-1} in the output layer depends on the task and is part of the design process; noteworthy is the softmax activation function often used for classification tasks. Note that by this definition, a layer l does only receive input from the earlier layer $l-1$; however, usually, FNNs exhibit more complex architectures but have in common that they are unidirectional and information flows only from earlier layers to *deeper* layers in the network. Hence they are called deep feedforward neural networks.

Supervised Learning The parameters θ in Equation (2.19) need to be inferred by a learning process. In *supervised* training, this is achieved by defining a loss function $\mathcal{L}_i(\Phi(\mathbf{x}_i, \theta), \mathbf{y}_i^*)$ that measures the quality of the network's prediction $\mathbf{y}_i$ compared to a target value $\mathbf{y}_i^*$. Given a training set $\{\mathbf{x}_i, \mathbf{y}_i\}_i^N$ of N individual input-output pairs, the model's parameters are optimized by an optimization algorithm in such a way that the

network minimizes $\mathcal{L} = \sum_i \mathcal{L}_i$ over the dataset while generalizing well to unseen data $\mathbf{x}^{\text{new}}$. A crucial step is to find a definition of a loss function that suits the task. Even if there exist plenty of different optimization algorithms to find (sub-) optimal model parameters, a simple one is *stochastic gradient descent* (SGD),

$$\Delta\theta_{ij}^{l} = -\eta \nabla_{\theta_{ij}^{l}} \mathcal{L}_i \left(\Phi(\mathbf{x}_i, \theta), \mathbf{y}_i^*\right), \tag{2.25}$$

where η is a learning rate and $\Delta\theta_{ij}^{l}$ the resulting weight update for weight θ_{ij}^{l}. The weights are then updated iteratively for each data point in the training set. This procedure is repeated over many epochs until convergence. Usually, more sophisticated optimizers, like the Adam optimizer [Kingma et al., 2017], are used; however, almost all optimizers have in common that they utilize the gradient of the loss function. Due to the composed structure of FNNs in Equation (2.21), the chain rule can be applied to calculate the gradient, allowing to *backpropagate* the gradient back through the network. This can be implemented very efficiently and is known as *backpropagation* (BP) [Kelley, 1960]. The supervised learning process will be discussed in more detail in the context of RSNNs in section 2.3.2.

Recurrent Neural Networks

A FNN gets an feature vector $\mathbf{x}_i$ as input and outputs a prediction $\mathbf{y}_i$. If the input data, however, is given as a sequence of T elements

$$\mathcal{X}_i = \left(\mathbf{x}_i^t\right)_{t=1,\dots,T}, \tag{2.26}$$

each element $\mathbf{x}_i^t \in \mathbb{R}^D$, with D the feature dimension, would have to be forwarded by the network individually. Often, elements in sequences are correlated, meaning an element $\mathbf{x}_i^{t-1}$ might hold information that helps to explains features in $\mathbf{x}_i^t$. Using a simple FNN, this sequential correlation cannot be exploited, since neurons in FNNs do not maintain an internal state and information is never related between elements $\mathbf{x}_i^t$. RNNs use therefore recurrent layers with feedback connections that allow to propagate information over the sequence (the following is based on Goodfellow et al. [2016, Chapter 10]). Assuming the recurrent network to be a parameterized mapping

$$\mathbf{y}_i^t = \Phi^{\text{rnn}}\left(\mathbf{x}_i^t, \mathcal{S}^{t-1} \middle| \theta\right), \tag{2.27}$$

each element in the sequence $\mathcal{X}_i$ is forwarded successively, resulting in a predicted sequence[1]

$$\mathcal{Y}_i = \left(\mathbf{y}_i^t\right)_{t=1,\dots,T}. \tag{2.28}$$

Here, $\mathcal{S}^{t-1}$ is the internal state of the network that holds processed information of previous elements $\mathbf{x}_i^{t'<t}$, allowing Φ^{rnn} to incorporate the past into the current prediction $\mathbf{y}_i^t$. This state is updated recursively with each forwarded element $\mathbf{x}_i^t$ according to

$$\left(\mathbf{x}_i^t, \mathcal{S}^{t-1}\right) \mapsto \mathcal{S}^t, \tag{2.29}$$

[1] Since each element in $\mathcal{X}_i$ is mapped to one element in $\mathcal{Y}_i$ this is called a "many-to-many" mapping. Depending on the task, other approaches are "one-to-many" or "many-to-one" mappings.

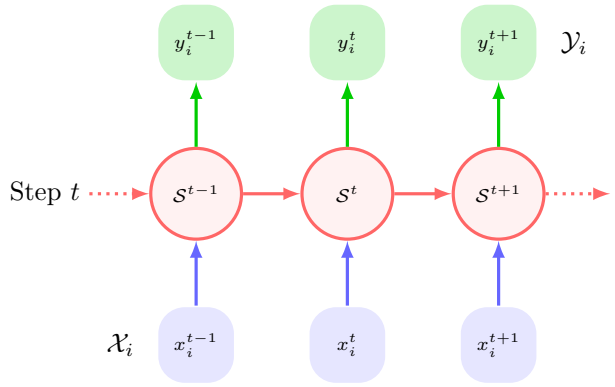


Figure 2.4: Processing of sequential data by a RNN. Each element $\mathbf{x}_i^t \in \mathcal{X}_i$ is processed individually, resulting in an output sequence $\mathcal{Y}_i$. The recurrent network holds an internal state $\mathcal{S}^t$ that propagates information over the sequence.

such that information is propagated along the sequence. Figure 2.4 shows the corresponding computational graph of the unrolled network. As in Equation (2.21) for FNNs, the mapping Φ^{rnn} for RNNs is usually given by a composition of layers. However, in RNNs some layers are recurrent, utilizing the internal state $\mathcal{S}$.

Each neuron n_j^l in a recurrent layer l of size N^l, as depicted in Figure 2.5, holds an layer-specific internal state $s_j^{(l),t} \in \mathcal{S}^t$ which is shared between the layer's neurons by recurrent connections. This allows the layer to perform an operation

$$\mathbf{s}^{(l),t} = f\left(\mathbf{s}^{(l),t-1}, \mathbf{x}_i^t \middle| \theta^l\right), \tag{2.30}$$

which maps the neurons' states from $t-1$ to t by a parameterized function f using the output $\mathbf{y}_i^{(l-1),t} = \mathbf{x}_i^{(l),t}$ of the previous layer and the states $\mathbf{s}^{(l),t-1}$ from the previous step. Here θ^l denotes the parameters of the operation. $\mathbf{s}^{(l),t} \in \mathbb{R}^{N^l}$ is the state vector holding the states of all recurrent neurons in the layer. Concretely, the function f can, for instance, be a linear projection followed by an activation function a,

$$\mathbf{s}^{(l),t} = a\left(\mathbf{W}^{(l),\text{hh}}\mathbf{s}^{(l),t-1} + \mathbf{W}^{(l),\text{ih}}\mathbf{x}_i^{(l),t} + \mathbf{b}^{(l)}\right), \tag{2.31}$$

with the recurrent weights $\mathbf{W}^{(l),\text{hh}} \in \mathbb{R}^{N^l \times N^l}$ weighting the states, $\mathbf{W}^{(l),\text{ih}} \in \mathbb{R}^{N^{l-1} \times N^l}$ weighting the inputs $\mathbf{x}_i^{(l),t}$ and $\mathbf{b}^{(l)} \in \mathbb{R}^{N^l}$ the bias vector. The states $\mathbf{s}^{(l),t}$ are then forwarded to the next layer $l+1$ as input $\mathbf{x}_i^{(l+1),t}$. It is noteworthy that the parameters do not depend on t and are shared across the sequence. The layer described by Equation (2.31) is a very simple example and only one out of many possible designs of recurrent layers. The most common ones are certainly the *long short-term memory* (LSTM) [Hochreiter et al., 1997] and the *gated recurrent unit* (GRU) [Cho et al., 2014].

2.3 Recurrent Spiking Neural Networks

RSNNs follow the general idea of recurrence described in Section 2.2. However, in contrast to RNNs, an RSNN contains spiking neurons. In particular, this means a spiking neuron j also maintains an internal state $\mathbf{s}_j^t \in \mathbb{R}^d$ but only exhibit an observable state $z_j^t \in \{0, 1\}$ that describes the neuron's output as a binary spike event [Bellec et al., 2019].

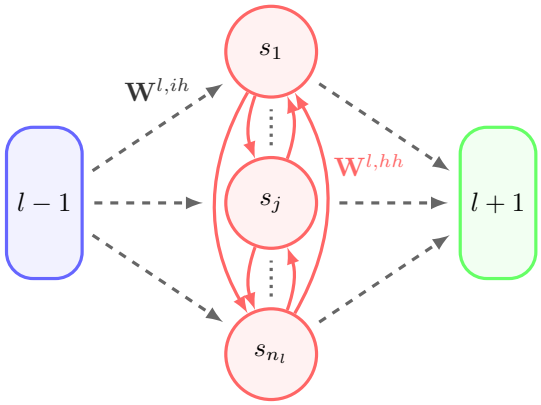


Figure 2.5: Visualization of a recurrent spiking layer. The layer receives input from its preceding layer $l-1$ projected onto layer l with weights $\mathbf{W}^{l,\text{ih}}$. This input is used together with the layers observable state $\mathbf{z}^{t-1}$ fed-back with weights $\mathbf{W}^{l,\text{ih}}$ to update the internal states $\mathbf{s}^t$. The resulting vector $\mathbf{z}^t$ is forwarded to the subsequent layer $l+1$.

Here, d is the dimension of the neuron's state, given for LIF neurons by the membrane potential v_j^t with $d=1$. In contrast to the recurrent layer in Equation (2.31), spiking neurons do only expose the observable state z_j^t to other neurons but not their internal state vector $\mathbf{s}_j^t$ [Bellec et al., 2019]. That is, the state $\mathbf{s}_j^{(l),t}$ of neuron j in a recurrent spiking layer l of size N^l only depends on other neurons in the layer by the observable state vector $\mathbf{z}^{(l),t}$. The observable state vector represents the output of all neurons in the layer. The internal state dynamic of neuron j in layer l is then given by a function M [Bellec et al., 2019],

$$\mathbf{s}_j^{(l),t} = M\left(\mathbf{s}_j^{(l),t-1}, \mathbf{z}^{(l),t-1}, \mathbf{x}^{(l),t} \middle| \theta^l\right), \tag{2.32}$$

which is mapped to the observable state $z_j^{(l),t}$ by f

$$z_j^{(l),t} = f\left(\mathbf{s}_j^{(l),t}, \mathbf{z}^{(l),t-1}, \mathbf{x}^{(l),t} \middle| \theta^l\right), \tag{2.33}$$

where $\mathbf{x}^{(l),t}$ is the input from the previous layer. Specifically, for LIF neurons, M is defined by Equation (2.14) and f by Equation (2.35). Figure 2.6 shows the corresponding computational graph.

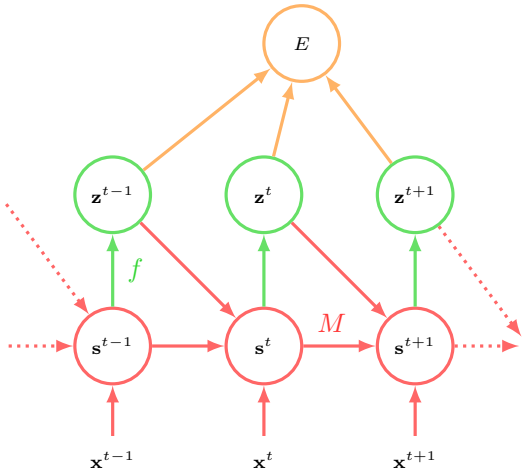


Figure 2.6: Computational graph of a recurrent spiking layer unrolled in time. The red nodes describe the neurons internal states $\mathbf{s}_j$, the green nodes the observable state vector $\mathbf{z}^t$. Each internal state at t is mapped to a new state at $t+1$ by M which incorporates the inputs $\mathbf{x}^t$, $\mathbf{z}^{t-1}$ and $\mathbf{s}_j^{t-1}$. f defines the mapping from the internal state to the observable state, given by the spike events z_j^t for spiking LIF neurons. Graphic inspired by [Bellec et al., 2020].

2.3.1 Network under Consideration

Without loss of generality, the network considered in the following consists of an input layer, a single recurrent layer followed by an output layer, as depicted in Figure 2.7 (the notation follows [Bellec et al., 2019]). The input layer has n_i neurons, providing an input vector $\mathbf{x}^t \in \{0,1\}^{n_\mathrm{i}}$ to the recurrent layer weighted by θ^ih. Each of the n_h LIF neurons in the recurrent layer has an internal state $\mathbf{s}_j^t = \left[v_j^t\right]$, evolving according to Equation (2.14) (for simplicity setting $v_\mathrm{l} = v_\mathrm{r} = 0$ and replacing n with t),

$$v_j^{t+1} = \alpha v_j^t + \sum_i \theta_{ji}^\mathrm{ih} x_i^{t+1} + \sum_{i \neq j} \theta_{ji}^\mathrm{hh} z_i^t \tag{2.34}$$

with a δ-kernel as given in Equation (2.13). For LIF neurons, the observable state z_j^t is defined by the Heaviside function

$$z_j^t = \Theta\left(v_j^t - \vartheta\right), \tag{2.35}$$

where ϑ is the threshold. The observable state vector $\mathbf{z}^t$ is projected onto the recurrent layer at $t+1$ weighted by θ^hh, as well as on the output layer at t with weights θ^ho. The output layer is assumed to consist of n_o *leaky integrate* (LI) neurons with a membrane dynamic like LIF neurons, however, without the ability to spike,

$$y_k^t = \kappa y_k^{t-1} + \sum_j \theta_{kj}^\mathrm{ho} z_j^t. \tag{2.36}$$

These types of neurons are called *readout* neurons from here on. Note, readout neurons have a decay constant $\kappa = \exp\left(-\Delta T / \tau_{\mathrm{m},k}\right)$ to distinguish membrane time constants between spiking and non-spiking neurons. Due to the simplified network structure, the layer index l is omitted.

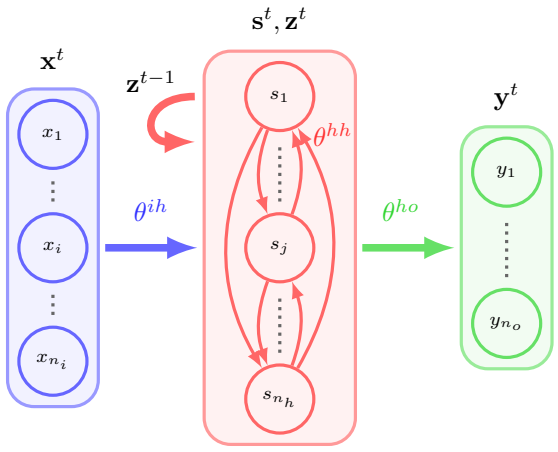


Figure 2.7: A recurrent spiking neural network with one input layer, a single recurrent layer consisting of LIF neurons and a output layer with non-spiking readout neurons. At each time t an input spike vector $\mathbf{x}^t$ is integrated in the recurrent neurons together with the observable state vector $\mathbf{z}^t$ — the spike events from the previous time step — by recurrent connection. Spike events arriving at a neuron j change its internal state $\mathbf{s}_j^t$, given by the membrane potential. The recurrent spikes are projected onto the readout neurons, which define the network's output $\mathbf{y}^t$.

2.3.2 A Learning Framework

In order to enable learning in the network at hand, a learning framework is outlined as described by Bellec et al. [2019]. For learning, the network's synaptic weights need

to be adjusted such that the loss function E of the network is minimized. E might depend on the network's observable state $\mathbf{z}$ or on a subset of it. For a regression task, E measures the deviation of the readout neuron's membrane trace y_k^t to some target values $y_k^{*,t}$. Regardless of the explicit form of E, for gradient-based learning, an expression for the gradient of E with respect to the network's weights $\theta^{\text{ih}}, \theta^{\text{hh}}$, and θ^{ho} needs to be found. Then, the gradient $\frac{\mathrm{d}E}{\mathrm{d}\theta_{ji}}$ suggests the direction in which the weight θ_{ji} needs to be adjusted in order to minimize E. According to Werbos [1990], the gradient with respect to the recurrent weights θ^{hh} can be decomposed into

$$\frac{\mathrm{d}E}{\mathrm{d}\theta_{ji}^{\text{hh}}} = \sum_t \frac{\mathrm{d}E}{\mathrm{d}\mathbf{s}_j^t} \cdot \frac{\partial \mathbf{s}_j^t}{\partial \theta_{ji}^{\text{hh}}}, \tag{2.37}$$

which is widely known as *back-propagation through time* (BPTT). Assuming the network's loss to depend exclusively on the observable state vector $\mathbf{z}^t$, $E = E\left(\mathbf{z}^1, \ldots, \mathbf{z}^T\right)$, Bellec et al. [2019] factorize the gradient in Equation (2.37) into a sum of products,

$$\frac{\mathrm{d}E}{\mathrm{d}\theta_{ji}^{\text{hh}}} = \sum_t L_j^t \cdot e_{ji}^t, \tag{2.38}$$

with L_j^t being the *learning signal* for neuron j and e_{ji}^t the *eligibility trace* of the corresponding synapse ji (for a proof see [Bellec et al., 2019, page 20-22]). While the learning signals depend on the network's error E, the eligibility traces are performance independent and represent all local information available at a synapse at time t. Approximated learning algorithms emerging from Equation (2.38) that can be computed *online* (i.e., forward in time) are referred to as *e-prop*.

Eligibility Traces

Considering Equation (2.32), the dynamic of the internal state $\mathbf{s}_j^t$ of a neuron j isolated from all other neurons is given by

$$D_j^{t-1} \coloneqq \frac{\partial}{\partial \mathbf{s}_j^{t-1}} M\left(\mathbf{s}_j^{t-1}, \mathbf{z}^{t-1}, \mathbf{x}^t \middle| \theta^{\text{ih}}, \theta^{\text{hh}}\right) = \frac{\partial \mathbf{s}_j^t}{\partial \mathbf{s}_j^{t-1}} \in \mathbb{R}^{d \times d}. \tag{2.39}$$

For LIF neurons, this describes how the membrane potential intrinsically evolves over time. Further, if the internal state changes with its corresponding weights according to

$$\frac{\partial \mathbf{s}_j^t}{\partial \theta_{ji}^{\text{hh}}} \coloneqq \frac{\partial}{\partial \theta_{ji}^{\text{hh}}} M\left(\mathbf{s}_j^{t-1}, \mathbf{z}^{t-1}, \mathbf{x}^t \middle| \theta^{\text{ih}}, \theta^{\text{hh}}\right) \in \mathbb{R}^d, \tag{2.40}$$

the derivation of Equation (2.38) arrives at the recursively defined *eligibility vectors*

$$\boldsymbol{\epsilon}_{ji}^t = D_j^{t-1} \cdot \boldsymbol{\epsilon}_{ji}^{t-1} + \frac{\partial \mathbf{s}_j^t}{\partial \theta_{ji}^{\text{hh}}} \in \mathbb{R}^d, \tag{2.41}$$

which, intuitively, propagate local synapse activation information from the past to the current time t. This will become clear in the context of LIF neurons. The eligibility traces are then given by

$$e_{ji}^t = \frac{\partial z_j^t}{\partial \mathbf{s}_{ji}^t} \cdot \boldsymbol{\epsilon}_{ji}^t \tag{2.42}$$

and quantify how much a synapse ji remembers of its activation in the past. Note, the eligibility traces are not an approximation and accumulate all contributions to the gradient, which can be computed forward in time.

Learning Signals

For Equation (2.38) to hold true, the learning signals need to be given by the total derivative of the loss E with respect to the neuron's observable state,

$$L_j^t \coloneqq \frac{\mathrm{d}E}{\mathrm{d}z_j^t}. \tag{2.43}$$

These learning signals define how the activity of neuron j at time t influences the network's error in the future. The dependency of future errors on the current activity poses a problem for a biologically-motivated learning rule because the computation of L_j^t requires the knowledge of how much the activity at t is responsible for the network's error in the future. In order to calculate the mathematically correct gradient, this future information needs to be backpropagated through time, which lacks biological plausibility. Additionally, BPTT it is also not very appealing from a computational point of view; calculating the learning signals requires saving the network's states for all times t during the forward pass, such that afterward, all information is present to perform the backward pass. This results in *locking* since the weights can only be updated after a full emulation of the network. Therefore, Bellec et al. [2019] propose an *online* approximation of the learning rules, allowing to compute L_j^t, and thus the gradient, forward in time.

Given an online approximation for the learning signals, Equation (2.38) suggests a clear learning strategy: At all times t, adjust the weights $\theta_{ji}^{\mathrm{hh}}$ by $-\eta L_j^t e_{ji}^t$, either in an online fashion or accumulative (with η the learning rate). It is important to note that the learning framework described is not limited to spiking neuron models but can also be applied to non-spiking networks with appropriate definitions of M and f in Equation (2.32) and (2.33). In particular, this holds for artificial neural networks with LSTM units.

2.3.3 Biologically inspired Alternative to BPTT

BPTT can be avoided by ignoring the impact of the network's activity at t on future errors at $t' > t$ and only considering the instantaneous error. Mathematically, this means expanding the total derivative in the learning signals and neglecting the derivative with respect to future states [Bellec et al., 2019],

$$L_j^t = \frac{\mathrm{d}E}{\mathrm{d}z_j^t} = \frac{\partial E}{\partial z_j^t} + \frac{\mathrm{d}E}{\mathrm{d}\mathbf{s}_j^{t+1}} \frac{\partial \mathbf{s}_j^{t+1}}{\partial z_j^t} \longrightarrow \hat{L}_j^t = \frac{\partial E}{\partial z_j^t}, \tag{2.44}$$

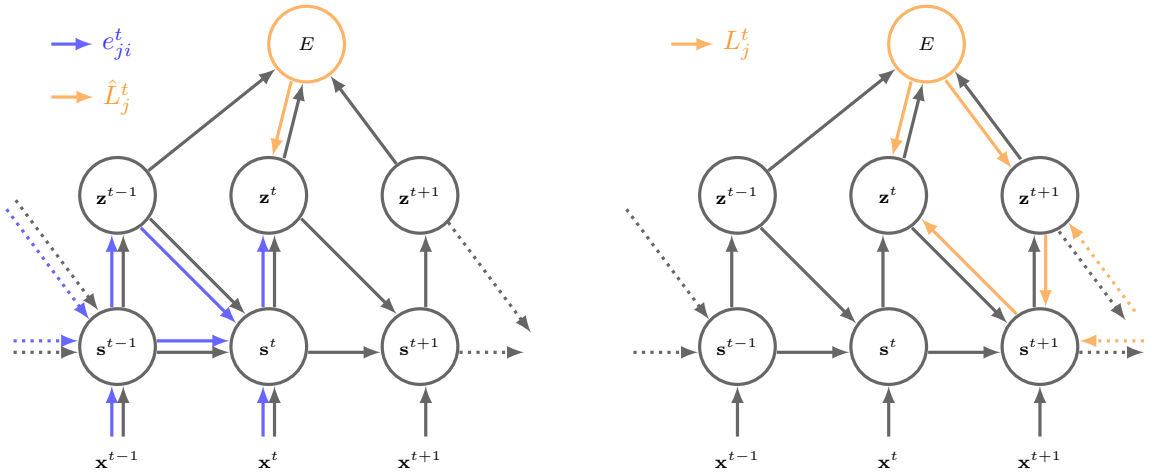
such that the approximated learning signals $\hat{L}_j^t$ are given by the remaining partial derivative[2]. These learning signals can now be computed in a forward-manner. Therefore, the

[2]Note that this is only one way to find online learning signal. They can for instance also be generated by a neural network, see [Bellec et al., 2019].

approximated version of the gradient in Eq. (2.38) result in weight updates (fc. Equation (2.25))

$$\Delta\theta_{ji}^{\text{hh}} = -\eta \frac{\widehat{\mathrm{d}E}}{\mathrm{d}\theta_{ji}^{\text{hh}}} = -\eta \sum_t \hat{L}_j^t \cdot e_{ji}^t, \tag{2.45}$$

that can be calculated in parallel to the forward pass. The corresponding computational graph is visualized in Figure 2.8a.



(a) Computational Graph of e_{ji}^t and $\hat{L}_j^t$

(b) Computational Graph of L_j^t

Figure 2.8: Computational graphs for *e-prop*. **(a)** Computing the gradient under online approximation requires to compute e_{ji}^t and $\hat{L}_j^t$. Both can be calculated in a forward-fashion without propagating the gradient back in time. **(a)** Using the mathematical correct gradient with L_j^t results in BPTT, since the network's activity at time t influences the error in the future. BPTT is considered biologically implausible. Graphs reproduced from [Bellec et al., 2019].

Emerging Learning Rule for LIF Neurons

Now, as a general learning framework for recurrent spiking (and non-spiking) networks is provided in Section 2.3.2, the corresponding learning rules for the LIF network in Section 2.3.1 can be specified. A detailed explanation of the following derivation can be found in [Bellec et al., 2019; Bellec et al., 2020].

In order to calculate the gradient in Equation (2.45), the eligibility traces e_{ji}^t and the learning signals $\hat{L}_j^t$ need to be defined for LIF neurons. For the LIF neurons, the functions M and f are described by Equation (2.34) and Equation (2.35), respectively. Inserting Equation (2.34) into Equation (2.39) and (2.40), with $\mathbf{s}_j^t = \left[v_j^t\right]$, gives

$$D_j^{t-1} = \alpha \quad \text{and} \quad \frac{\partial v_j^t}{\partial \theta_{ji}^{\text{hh}}} = z_i^{t-1}. \tag{2.46}$$

The resulting eligibility vector in Equation (2.41) is then given by the low-pass filtered recurrent spike train z_i^t,

$$\boldsymbol{\epsilon}_{ji}^{t+1} = \alpha \boldsymbol{\epsilon}_{ji}^t + z_i^t = \sum_{t' \leq t} \alpha^{t-t'} z_i^{t'} \stackrel{\text{def}}{=} \hat{z}_i^t. \tag{2.47}$$

For spiking neurons, the partial derivative $\frac{\partial z_j^t}{\partial v_{\mathrm{m},j}^t}$ in Equation (2.42) is ill-defined due to the discontinuous observable state z_j^t. Therefore, this derivative is replaced by a pseudo-derivative h_j^t,

$$h_j^t = \gamma \max\left(0, 1 - \left|\frac{v_j^t - \vartheta}{\vartheta}\right|\right), \tag{2.48}$$

with a tunable smoothing factor γ. Finally, inserting h_j^t and the expression for $\boldsymbol{\epsilon}_{ji}^{t+1}$ into Equation (2.42) yields the eligibility traces

$$e_{ji}^{t+1} = h_j^{t+1} \cdot \hat{z}_i^t. \tag{2.49}$$

The learning signals $\hat{L}_j^t$ depend on the loss function E. Assuming a regression task, where the membrane potentials y_k^t of the readout neurons have to resample a target trace $y_k^{*,t}$, the loss is given by the *residual sum of squares* (RSS),

$$E = \frac{1}{2} \sum_{t,k} (y_k^{*,t} - y_k^t)^2. \tag{2.50}$$

Inserting the membrane potential y_k^t in Equation (2.36) into E and executing the partial derivative with respect to z_j^t gives the online learning signals

$$\hat{L}_j^t = \sum_{t' \geq t} \sum_k \theta_{kj}^{\mathrm{ho}} \left(y_k^{t'} - y_k^{*,t'}\right) \kappa^{t'-t}. \tag{2.51}$$

Observe that the learning signals $\hat{L}_j^t$ are now given by a sum over the future. For online learning, this, obviously, is a problem. However, this can be addressed by inserting $\hat{L}_j^t$ and e_{ji}^t into Equation (2.45) and interchanging the sum indices,

$$\Delta\theta_{ji}^{\mathrm{hh}} = -\eta \sum_t \sum_{t' \geq t} \sum_k \theta_{kj}^{\mathrm{ho}} \left(y_k^{t'} - y_k^{*,t'}\right) \kappa^{t'-t} e_{ji}^t \tag{2.52}$$

$$= -\eta \sum_t \sum_k \theta_{kj}^{\mathrm{ho}} \left(y_k^t - y_k^{*,t}\right) \sum_{t' \leq t} \kappa^{t-t'} e_{ji}^{t'}. \tag{2.53}$$

This is the desired online plasticity rule for the recurrent synaptic weights. Due to the similarity of Equation (2.53) and the approximated gradient in Equation (2.45), the term $\sum_k \theta_{kj}^{\mathrm{ho}} \left(y_k^t - y_k^{*,t}\right)$ is from here on called *learning signal* (change in terminology) unless not explicitly referring to Equation (2.51). The plasticity rule for the input weights $\theta_{ji}^{\mathrm{ih}}$ can be derived in the same way. In fact, solely replacing the recurrent spike train z_i^{t-1} with the input spike train x_i^t in Equation (2.46) gives the weight updates $\Delta\theta_{ji}^{\mathrm{ih}}$. The updates for the output weights are not subject to the *e-prop* framework and arise from simple backpropagation,

$$\Delta\theta_{kj}^{\mathrm{ho}} = -\eta \sum_t \left(y_k^t - y_k^{*,t}\right) \sum_{t' \leq t} \kappa^{t-t'} z_j^{t'}. \tag{2.54}$$

Regularization

To prevent the recurrent neurons to fire unrealistically strong a firing rate regularization term is introduced and added to the weight updates. For an average firing rate $f_j^{\text{av}} = \frac{\delta t}{T} \sum_t z_j^t$ of neuron j and a desired target rate f^{target}, the regularization loss is given by

$$E^{\text{reg}} = \frac{1}{2} \sum_j \left(f_j^{\text{av}} - f^{\text{target}} \right)^2 . \tag{2.55}$$

Applying the just derived learning framework gives a regularization weight update for the recurrent and input weights (see Bellec et al. [2019]),

$$\left(\Delta\theta_{ji}^{\text{ih, hh}} \right)^{\text{reg}} = \eta^{\text{reg}} \sum_t \frac{\delta t}{T} \left(f^{\text{target}} - f_j^{\text{av}} \right) h_j^t \hat{z}_i^{t-1}. \tag{2.56}$$

with η^{reg} being the regularizing learning rate, defining how strong deviations from the target rate are penalized. The total weight updated it then simply the sum of $\Delta\theta_{ji}^{\text{ih, hh}}$ in Equation (2.53) and $\left(\Delta\theta_{ji}^{\text{ih, hh}} \right)^{\text{reg}}$.

Error Broadcasting

In Equation (2.53), the update rule consists of the local eligibility traces e_{ji}^t, filtered by the readout decay κ and weighted by a neuron-specific learning signal $\sum_k \theta_{kj}^{\text{ho}} (y_k^t - y_k^{*,t})$. This error signal can be related to the experimental finding of *error-related negativity* (ERN) in the brain [Bellec et al., 2019]. It is suggested that the ERN is a signal which accounts for behavioral errors and gates learning. Interestingly, this ERN can be measured before an error is received by sensory feedback [MacLean et al., 2015, Fig. 4], leading to the assumption that the brain uses an error prediction network to guide synaptic plasticity. Further, experimental data shows that error signals emitted in the brain, for instance, by dopaminergic neurons in form of neuromodulators like dopamine, interact with synapse-specific eligibility traces [Gerstner et al., 2018]. Therefore, this learning rule has an appealing biological interpretation. For a more in-depth interpretation, see [Bellec et al., 2019].

However, when modeling biology, the dependency of the error signal on θ^{ho} seems problematic. Usually, the feedback connections from readout neurons are biologically given by different cells than the feed-forward connections. Therefore, it is unlikely that this feedback weight θ_{kj}^{ho}, given in the error signal (imagine this signal to be transmitted to neuron j by a feedback connection), has the exact same weight as the feed-forward connection. Bellec et al. [2019] replace this weight with a random feedback weight B_{kj},

$$\Delta\theta_{ji}^{\text{hh}} = -\eta \sum_t \sum_k B_{kj} \left(y_k^t - y_k^{*,t} \right) \sum_{t' \leq t} \kappa^{t-t'} e_{ji}^{t'}. \tag{2.57}$$

This is motivated by deep FNNs, for which it was found that a great amount of learning can be realized by replacing the backpropagated error signals with layer-specific randomly weighted sums of the network's global output error [Samadi et al., 2017; Nøkland, 2016].

An example of the derived online plasticity rule for a pattern-generation task can be seen in Figure 2.9.

Please note, error broadcasting is not element of this thesis and is only shown for the sake of completeness.

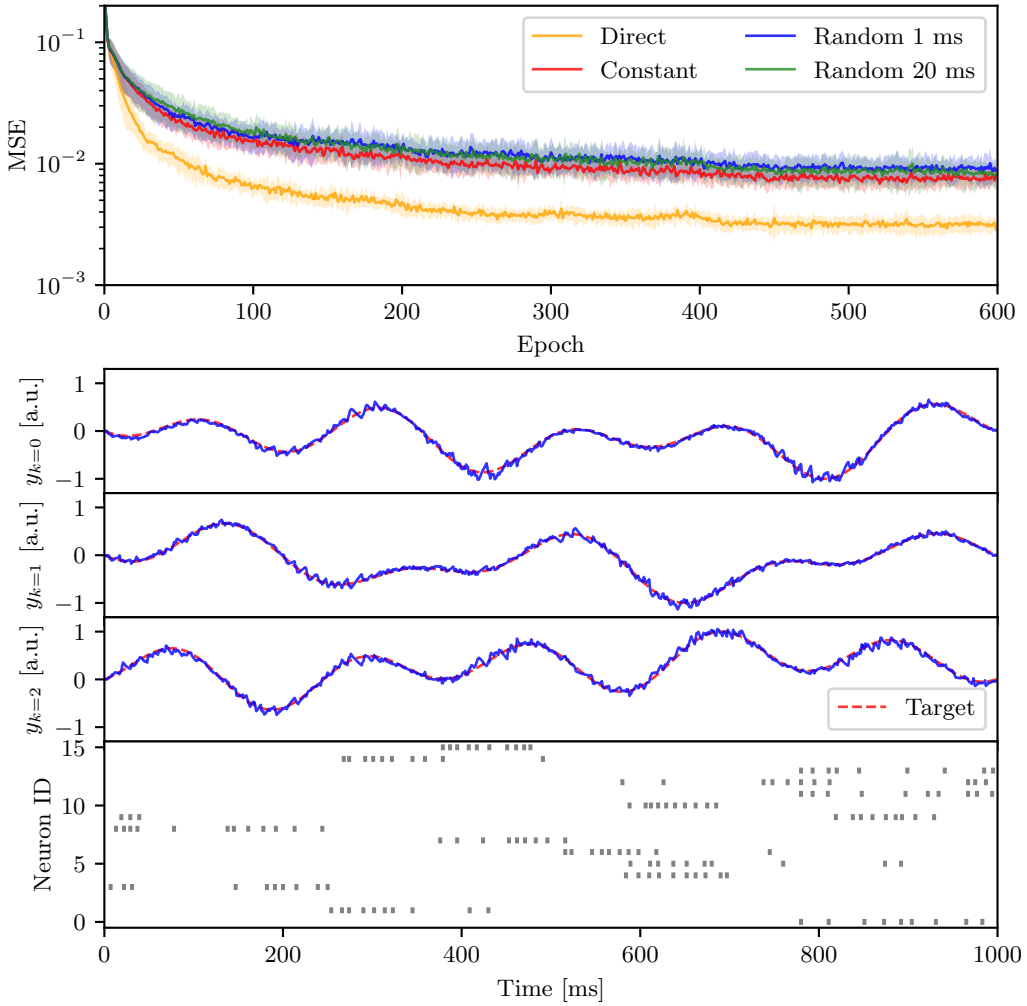


Figure 2.9: Pattern-generation task example trained with *e-prop*. A input layer provides Poisson distributed input events and to a single recurrent layer. The spike events of the recurrent neurons are integrated onto the membranes of three readout neurons over a time period of 1 s. Using the described online approximation of the true gradient, the network can learn to solve the task such that the membrane traces y_k^t do resemble the target traces $y_k^{*,t}$ very well. Instead of backpropagating the gradient through time, it is calculated completely forward in time, making it more plausible from a biological perspective. The lowermost plot shows the spike events of the 16 first recurrent neurons. The different colors in the upper plot describe different feedback strategies. Orange: Direct feedback with feedback weights $\theta_{kj}^{\mathrm{ho}}$. Red: Same random feedback matrix B_{kj} for all times t. Blue: Sampling new feedback weights every 1 ms. Green: Sampling new feedback weights every 20 ms. Network parameters: $n_{\mathrm{i}} = 20$, $n_{\mathrm{h}} = 600$, $n_{\mathrm{o}} = 3$, $\tau_{\mathrm{m}} = 20\,\mathrm{ms}$, $\tau_{\mathrm{ref}} = 5\,\mathrm{ms}$, $\vartheta = 0.6$.

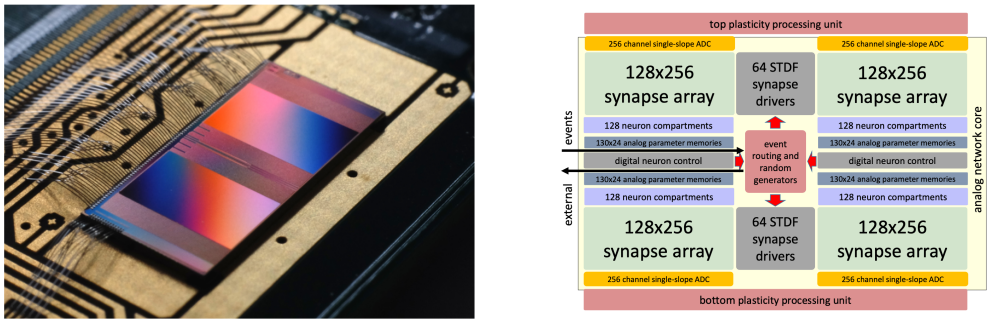
3 Neuromorphic Hardware

Neuromorphic hardware architectures approach the computation of neural networks by mimicking the behavior of their biological counterpart. Biologically inspired neuron models, like the *leaky integrate-and-fire* (LIF) neuron, are emulated in hardware rather than being simulated in a discrete fashion numerically [Schemmel et al., 2010]. Therefore, neurons are realized *in-silicio*, constituted by electronic components, allowing to build a system operating massively in parallel. This comes with the advantage that electrical neurons can have very small time constants, enabling in-silicio neurons to work on a much smaller time scale than biology. Especially neuro-scientific simulations are promising to profit from this acceleration. Numerical computer simulations taking weeks, or even months, can be reduced to merely a few seconds, opening up the possibility to conduct experiments on a much larger time frame. Further, compared to classical hardware, simulations on accelerated neuromorphic chips are also considered to be more energy-efficient [E. C. Müller, 2014].

3.1 The BrainScaleS System

The *BrainScaleS-2* (BSS-2) system is an accelerated neuromorphic hybrid architecture unifying both digital and analog technologies. On the very heart of the system is the *HICANN-X v2* (HX) chip (full name: HICANN-DLS-SR-HXv2) emulating analog *spiking neural network*s (SNNs) [Schemmel et al., 2020]. *Field-programmable gate array*s (FPGAs) allow, via Gigabit Ethernet, to configure and control the chip in real-time from a host computer. Therefore, experiments can be described in software, executed on the chip, and experiment observables, like spike times, can be accessed via the FPGA on the host-side. Several software abstraction layers enable high-level experiment descriptions [E. Müller et al., 2020a] without the need to handle host-chip communication manually. Notably, BSS-2 extensions for common frameworks like `PyNN` [Davison et al., 2009] or `PyTorch` [Paszke et al., 2019] facilitate chip usage for non-expert users (more on software in Chapter 4).

As depicted in Figure 3.1, the *analog network core* (ANNCORE) of HX is split into two hemispheres. Each hemisphere emulates up to 256 analog neuron circuits, allowing to perform 512 neural operations in parallel. Neuron circuits on a hemisphere are arranged on two sides; on each side, a synapse matrix provides 256 synaptic connections to each neuron compartment, making in total 131,072 synapses on HX. In the following, the two synapse matrices on the same hemisphere are considered one. The neuron's binary spike events are routed via a *crossbar* within the chip and are injected into the rows



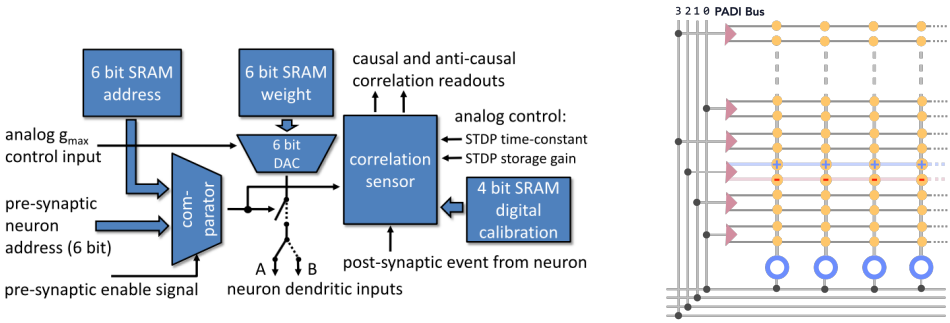
(a) HICANN-X v2 **(b)** Chip layout

Figure 3.1: *HICANN-X v2* (HX) chip. **(a)** Single HX chip close-up. The chip is approximately 4 mm × 8 mm in size. Photo taken by Eric Müller, 2020. **(b)** Schematic block diagram HX's *analog network core* (ANNCORE), which is composed of two hemispheres constituted by two quadrants each. Both hemispheres feature 256 neuron circuits, synapse matrices with 256 rows, and a digital general purpose processor (PPU). Among other, the PPU can access synapse weights, correlation data, and the neurons' membrane potentials via a *vector unit* (VU), allowing implementing high-speed on-chip plasticity rules. Image from [Schemmel et al., 2020].

of the synapse matrices by *synapse drivers*. All events are annotated with an address allowing direct addressing to specific groups of synapses and identifying them with the corresponding neuron when read out with the FPGA. External spike events can be injected into the chip and directed to synapses using the FPGA but can also be created on-chip by *spike generators*. As a crucial element of HX, each hemisphere has a dedicated digital general purpose processor with the purpose to execute plasticity algorithms on-chip — hence they are called *plasticity processing unit*s (PPUs). Synapses, Neurons, and PPUs are discussed in more detail in the following paragraphs.

Synapses Synapses have configurable 6 bit weights and 6 bit labels (see Figure 3.2a). On an incoming spike event, the address carried by the event is compared to the label stored in the synapse, and on match, the synapse triggers a short analog pulse with a height proportional to the synaptic weight. This pulse is translated to an exponentially decaying current onto the membrane of the corresponding neuron, which effectively emulates current-based synapses with a single exponential kernel. In addition, each synapse has a *correlation sensor* incorporated, described in Section 3.2. A single synapse driver governs two adjacent synapse rows in a synapse matrix. The synapse driver is configured to treat the corresponding lines row-wise excitatory or inhibitory. In essence, this means all synapses on the same row have either an inhibitory or an excitatory effect on the neurons' membranes. Unused synapse rows can be disabled individually.

On each hemisphere are four PADI bus lines on which the synapse drivers are connected alternately (see Figure 3.2b). Hence, a PADI bus is connected to 32 drivers. A event on HX are given by a 13 bit address. While the two uppermost bits select the PADI bus, the next 5 bits constitute a `row_select_address` that selects the driver on the bus. The



(a) Synapse block diagram **(b)** Synapse matrix

Figure 3.2: Synapses on HICANN-X v2. **(a)** Pre-synaptic spike events are provided row-wise to a synapse-row by synapse drivers, the events are 6 bit address is compared to the 6 bit synapse address. On match, an exponential decaying current is triggered on the neuron's membrane. The amplitude of the current depends on the synaptic 6 bit weight. Additionally, a correlation sensor measures the correlation between the pre- and post-synaptic events. Synapses can be configured row-wise excitatory (line A) or inhibitory (line B). Image from [Friedmann et al., 2017]. **(b)** Each synapse driver provides events to two synapse rows and is connected to a specific PADI bus line. A synapse column projects its events onto one neuron. External and internal spikes are routed by a crossbar.

lowermost 6 bits are injected into the synapse rows to target specific the synapse labels. Further, each driver has a 5 bit `row_address_compare_mask` that configures which bits of the `row_select_address` the driver should consider when deciding whether the event is addressed to it or not.

Neurons The neuron circuits on HX are designed to emulate *adaptive exponential integrate-and-fire* (AdEx) neurons in analog [Brette et al., 2005; Naud et al., 2008]. However, they can be configured to behave as LIF neurons according to Equation (2.6). Due to the neuron's *in-silicio* implementation, they have much smaller time constants than their biological counterparts. While biology usually operates with time constants of about $\mathcal{O}(1\,\mathrm{ms} - 100\,\mathrm{ms})$, neurons on HX can operate approximately $\mathcal{O}(10^3)$ faster. Thus, experiments on hardware need to be translated to the biological time domain by

$$1000 \cdot t^{\mathrm{bio}} = t^{\mathrm{hw}} \quad \longrightarrow \quad 1\,\mu\mathrm{s}_{\mathrm{hw}} = 1\,\mathrm{ms}_{\mathrm{bio}}. \tag{3.1}$$

Experiments in the thesis are always given in the biological time domain.

For neurons on HX to exhibit some desired membrane dynamic, they need to be parameterized individually. Therefore, each neuron has 24 analog *capacitive memory* (CapMem) cells, which are adjusted appropriately by a calibration process. This allows configuring the time constants τ_{m} and τ_{ref}, as well potentials ϑ, v_{l} and v_{r} as demanded. Since LIF neurons spike when the membrane potential exceeds the threshold, neuron circuits on HX have a *threshold comparator* that detects whether the membrane potential did indeed cross the threshold and produces a binary spike event correspondingly. This comparator

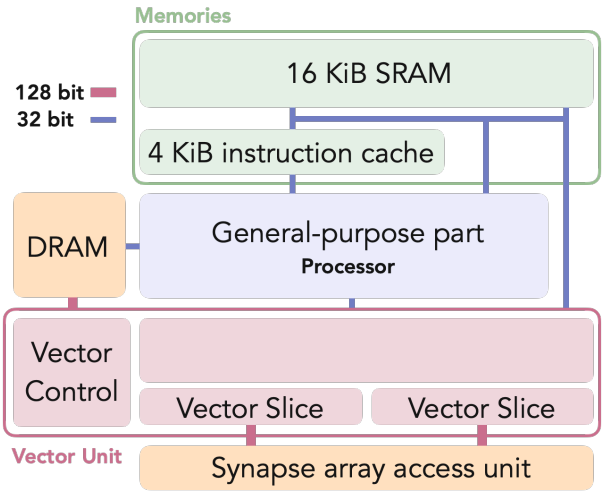


Figure 3.3: The *plasticity processing unit* (PPU) is a digital general purpose processor with a *vector unit* (VU) extension, present on each hemisphere of HX. The processor is based on a 32-bit architecture and can access the synapse matrix rows via the VU in parallel. It features 16 KiB SRAM and 4 KiB instruction cache. Code, as well as data, can optionally be placed on an external DRAM allowing to implement larger PPU programs. The PPU is intended to realize on-chip plasticity rules. Image inspired by [Friedmann et al., 2017].

can also be turned off, allowing the membrane potential to evolve freely within the possible hardware ranges without sending out a spike. For realizing readout neurons on HX, this non-spiking mode comes in handy. Further, the membrane trace of a single neuron on HX can be sampled with high resolution by a *membrane ADC* (MADC).

In order to force a neuron to spikes independently from its membrane potential, neurons have an excitatory and an inhibitory bypass-mode, which allows excitatory or inhibitory pre-synaptic spikes to trigger a post-synaptic event immediately. An additional important feature is the neuron's spike counter. Each neuron features a 9 bit digital counter, which is incremented by one at the occurrence of a post-synaptic spike. The first 8 bits represent the spike count, while the uppermost bit serves as an overflow detection. These counters can be reset and read out from the PPU.

Plasticity Processing Unit The analog circuits on HX are tightly coupled to the PPU [Friedmann, 2013] on each hemisphere. The PPU consists of a microprocessor with a 32-bit PowerISA 2.06 [PowerISA, 2010] architecture that is programmable in C/C++ (and assembler) in connection with a standalone C/C++ compiler, based on a custom extension of the `gcc` 8.1 toolchain. Per default, the PPU has a 4 KiB instruction cache and 16 KiB SRAM. However, depending on the software state, an external DRAM with up to 128 MB[1] is available on which program code as well as data can be placed. This allows implementing larger programs than possible with the 16 KiB SRAM.

As depicted in Figure 3.3, the PPU has a 128-byte wide *vector unit* (VU) extension. While the VU enables the PPU to perform vectorized operations and access the DRAM vector-wise, its primary purpose is to read and manipulate synaptic weights in the synapse matrices row-wise in parallel, effectively making on-chip plasticity feasible. In addition, the VU endows the PPUs with the ability to read out analog observables — digitized by 512-channel single-slope *analog-to-digital converter* (ADC) on each hemisphere — in parallel. The ADC has two channels per synapse column (casual and acausal, see

[1] Theoretically, the DRAM can have a size of 1 GiB when using the corresponding software.

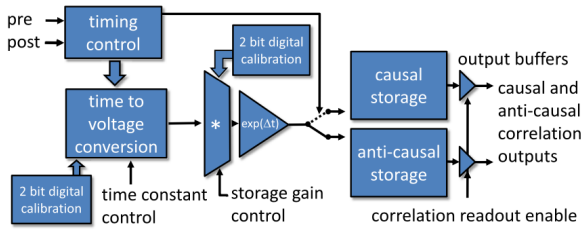


Figure 3.4: The correlation sensor measures the time between a pre- and the next post-synaptic event by a time to voltage conversion circuit. This voltage is weighted exponentially and stored in the causal storage. Correspondingly, a anti-causal storage stores the correlation between a post- and the next pre-synaptic event. These storages are digitized by the CADC and read out by the PPU via the VU. The amplitude and the time constant of the correlation measurements can be calibrated by two calibration bits each. Image from [Friedmann et al., 2017].

Section 3.2) and is referred to as *column ADC* (CADC) [Schreiber, 2021]. Among others, analog observables that can be read out with the VU are membrane potentials and correlation measurements (see Section 3.2).

3.2 Correlation Sensors

Each synapse in the synapse matrix on HX features an additional analog circuit — the *correlation sensor* — to measure local correlation information between pre-synaptic events arriving at a synapse and the post-synaptic spike events of the corresponding neuron. Correlation measurements in a synaptic connection is the fundamental element of plenty of Hebbian-inspired plasticity rules [Hebb, 2005], such as *spike timing dependent plasticity* (STDP), and is, therefore, a vital component for on-chip learning. As visualized in the block diagram in Figure 3.4, the sensor accumulates *causal* correlation between pre- and post-event pairs and *anti-causal* (acausal) correlation measurements between pairs of post- and pre-events simultaneously.

To measure correlation in analog, the sensors need to generate an internal timing between the latest pre-synaptic (post-synaptic) and the next post-synaptic (pre-synaptic) spike and translate it into a voltage via a time to voltage conversion circuit. This is achieved by setting a capacitor $C_{(\mathrm{a})\mathrm{causal}}$ to an initial value and triggering a constant discharge process at the occurrence of a pre-synaptic (post-synaptic) event, which is stopped on the subsequent post-synaptic (pre-synaptic) spike [Friedmann et al., 2017]. Then, the voltage on the capacitor is a measure for the time difference between the spike pair. This voltage is scaled via a storage gain parameter, weighted by an exponentially decaying function, and finally added to the causal (acausal) storage circuits, accumulating the correlation, as depicted in Figure 3.5. Using the 512-channel CADC (two ADCs, one causal, and one acausal channel per column), the causal and acausal storages of a whole synapse row can be measured in parallel via the PPU. This enables the PPU to calculate on-chip weight updates based on accumulated correlation.

Since pre-events (post-events) do start the discharge process of capacitor $C_{\text{(a)causal}}$ after a quick reset to its initial value, multiple succeeding pre-events (post-events) do merely retrigger the measurement; however, they do not change the storage. Therefore, the sensors do only measure the correlation in a nearest-neighbor fashion.

Due to transistor variations, the sensors are subject to fixed-pattern noise. In order to take care of this, each synapse is equipped with four digital calibration bits. The correlation amplitude η_c can be calibrated by adjusting the storage gain parameter with a 2 bit digital input. The remaining two bits calibrate the time to voltage conversion circuit which is defining the correlation time constant τ_c.

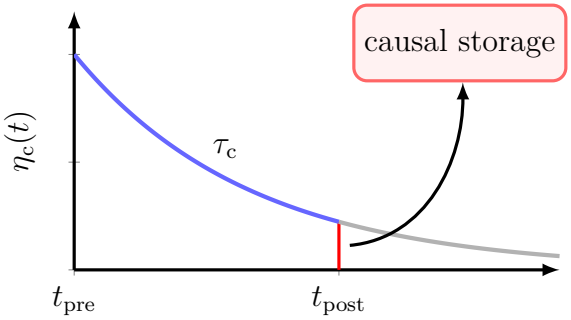


Figure 3.5: Behavior of the causal correlation curves. A pre-synaptic event at time t_{pre} triggers the discharge process. At the following post-synaptic event at t_{post} the remaining amplitude $\eta_c(t_{\text{post}})$ (red line) is read out and accumulated in the causal storage. Time constant τ_c and amplitude η_c are set by quadrant-global analog parameters.

4 Developed Software

Developing neuromorphic hardware goes hand-in-hand with providing software solutions that ensure seamless chip usage. Ideally, machine learning implementations on HX are described in common high-level software frameworks — such as `PyTorch` [Paszke et al., 2019] — and executed on the chip implicitly, allowing effortless experiment design by non-expert users. From low-level chip communication to software representations of experiments conducted on hardware and analog chip configuration, top-level chip usage requires several levels of chip abstraction to interact smoothly. This comes with many challenges that need to be tackled. One of them is certainly the interfacing of training in analog spiking neural networks to the digital learning environment `PyTorch` — subject of this chapter and one main element of this thesis.

Experiments conducted in later chapters are based on the software developed and elaborated in the following sections. These will encompass the description of a high-level e-prop experiment framework abstracting execution of experiments with *recurrent spiking neural network*s (RSNNs) in software and on hardware. Besides simulating networks of spiking neurons in software, the main focus is here the integration of HX in `PyTorch` for learning in RSNNs on hardware.

4.1 E-prop Framework

For simulations and experiments on HX, a top-level `EProp` framework is developed, written in `Python` and inspired by `BindsNET` [Hazan et al., 2018]. This framework allows describing abstract networks of spiking neurons by a high-level interface. Most importantly, it allows performing learning in RSNNs simulated in software and emulated on HX by defining several e-prop-inspired *online* learning rules. The interface is designed such that learning with HX in-the-loop (see Section 4.2.1) is equally abstracted as learning in software. This is achieved by interfacing the communication to HX by the C++ `PyTorch` extension `hxtorch` [Spilger et al., 2020]. Since `hxtorch` supports only non-spiking networks, so far, by providing analog matrix multiplications and convolutional operations [Weis, 2020; Emmel, 2020], support for spiking networks is introduced in Section 4.2 by incorporating `grenade` [Spilger, 2021]. `grenade` is a framework that finds graph-based experiment descriptions for HX. Therefore, a network is defined by populations of hardware neurons and projections between these populations, which are then mapped to HX by configuring the chip accordingly and handling all event-routing implicitly. After providing the network's inputs, the experiment is executed, and the required network observables returned by `grenade`. However, `grenade`'s default event-routing algorithm

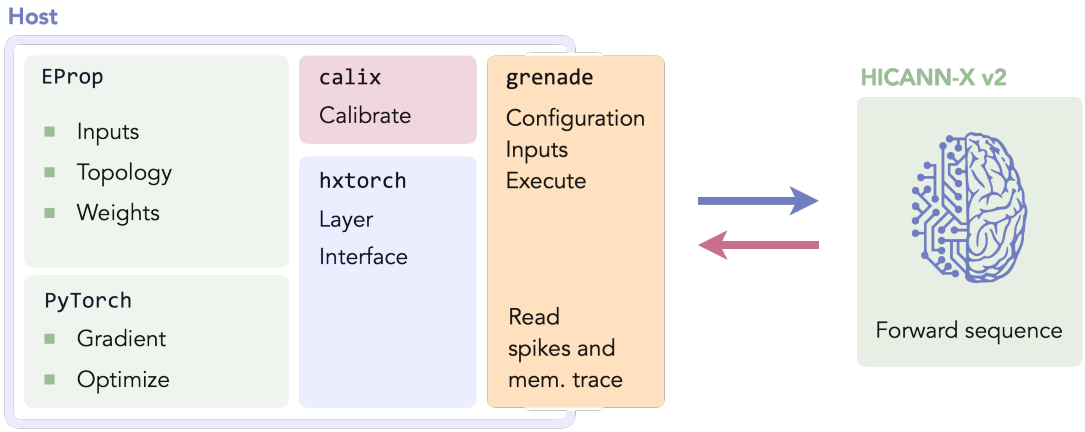


Figure 4.1: Software stack for experiments on HX. The high-level experiment framework is implemented in `PyTorch`, which utilizes `hxtorch` to perform the forward pass on HX. Within `hxtorch` the `grenade` interface is used to map the network's topology to a hardware representation on HX. `grenade` handles all spike routing, executes the experiment and returns desired observables. These hardware observables are translated to `PyTorch` tensors in `hxtorch`. The chip is calibrated by `calix`.

does not support recurrent projections; Hence, a corresponding algorithm for recurrent networks is contributed (see Section 4.2.3). The chip itself is brought into the desired working state by calibration with `calix` [Weis, 2020]. The described software stack is visualized in Figure 4.1.

4.1.1 Network Representation

In the top-level software layer, network topologies are defined in `Model` classes, derived from a `PyTorch` module `AbstractNetwork`. Similar to `PyNN` [Davison et al., 2009; Czierlinski, 2020], populations of neurons (layers) are added by the method `add_population`. As in Listing 4.1, populations are described by the population's size, neuron-type object, and the neurons' parameters. A `source` population is connected to a `target` population via a `Synapse` type and registered by `add_projection`. The synapse object implements the projection's functionality and handles all synapse-specific properties. In particular, this includes performing synaptic plasticity according to an *online* update rule defined in an injected projection-specific `LearningRule` class.

In principle, this interface supports network descriptions of arbitrary complexity, of which, of course, not all can be realized on hardware. Note that the interface is designed in a plug-in fashion. This is convenient for simulating various functionalities; Different implementations of neuron types, synapses, and learning rules can be exchanged without interfering with the network description.

Listing 4.1: Desciption of a spiking network in software.

```
class Model(AbstractNetwork):
    def __init__(self, *args, **kwargs):
        ...
        # Input neurons
        self.add_population(Input(n_i, ...), name="input")
        # Recurrent neurons
        self.add_population(CurrentLIFNodes(n_h, ...), name="hidden")
        # Readout neurons
        self.add_population(CurrentLINodes(n_o, ...), name="output")

        # Projections
        self.add_projection(Synapse(learning_rule=LearningRule, ...),
                source="input", target="hidden")
        ...
```

4.1.2 Simulating RSNNs in Software

Simulating RSNNs in software is fundamentally different from emulating spiking networks on analog neuromorphic hardware. On HX, spikes are represented event-based by addresses emitted at spike times and neurons are emulated in continuous time. In contrast, the simulations here work on a discrete time lattice.[1]

Therefore, the neuron object in each population (except the input population) stores the internal states $\mathbf{s}^t$ of its neurons — for instance, the membrane potentials v_j^t for LIF neurons. At each time t, each population in the network receives input from its corresponding projection, which maps the pre-synaptic events to neuron-specific inputs to the subsequent population by a dense synaptic weight matrix. The inputs are then used to evolve the neurons' internal states by one time step δt to $\mathbf{s}^{t+1}$. After integrating the internal dynamics, the population exposes an observable state $\mathbf{z}^{t+1}$ serving as input at step $t+1$ to subsequent projections. That is, at all steps t the neuron dynamics in the whole network are evolved to $t+1$ in parallel in terms of time discretization. As a consequence, there is no time delay due to spatial information flow from an earlier population to populations deeper in the network. Further, the recurrence in the network has an adjustable transmission delay given by δt, in contrast to HX, where spike events are fed back with fixed (but usually very small) latency. The network is propagated one time step by calling the models `forward` method [Paszke et al., 2019] with the network's current input events. A computational graph is given in Figure 4.2.

For the network described in Section 2.3.1, the dynamic of recurrent LIF neurons is implemented in a class `CurrentLIFNodes`, which integrates membrane potentials numerically according to Equation (2.14) when calling `forward`. In addition, this class

[1] In principle, it is also possible to implement event-based simulations. This approach is not considered in this thesis.

simulates the synaptic input current by neuron-specific single exponential kernels as in Equation (2.17). The readout neurons implemented in `CurrentLINodes` behave identically, however, without the ability to spike.

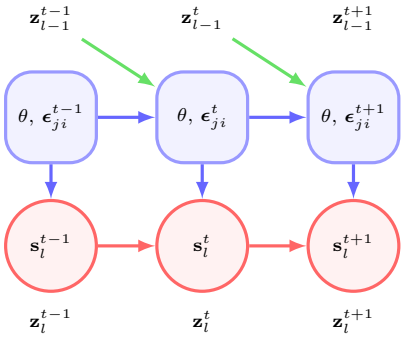


Figure 4.2: Computational graph of the network simulation engine unrolled in time. The network is described by populations (red) and projections between populations (blue). Spiking neurons in population l hold internal states $\mathbf{s}^t$ and exhibit spikes $\mathbf{z}^t$. The projections receive spike events of population $l-1$ at time $t-1$ to provide input to the layer l at t. Projections hold the synaptic weights and propagate the eligibility vectors $\boldsymbol{\epsilon}_{ji}^t$.

4.1.3 Learning

The synapses between two populations are managed by a `Synapse` object in the network's projection. This object holds the synaptic weight matrix as `torch.Parameters` [Paszke et al., 2019] and a dedicated `LearningRule` instance to adjust these synaptic weights appropriately. Therefore, after evolving the network's populations by δt, an `update` method is called on the synapse object, which, firstly, computes the inputs for the next time step and, secondly, evokes the plasticity rule instance to compute, depending on the update rule, parts of the gradient, or the weight updates directly. Hence, this implementation supports calculating weight updates in parallel to the forward pass.

Since the plasticity rule in a `Synapse` has full access to its `target`'s population state variables $\mathbf{z}$ and $\mathbf{s}$ and to the eligibility vectors $\boldsymbol{\epsilon}_{ji}^t$ simulated within each synapse, the eligibility traces e_{ji}^t — and approximations of it — can be computed at each step t. This local information is either processed directly and merged with externally injected learning signals L_j^t to compute the contribution of the current time step to the weight update or stored and used with a sequence of learning signals after simulating the network. In the first case, the weight updates can be applied *on-the-fly* to the synaptic weights or buffered and summed up in the end according to Equation (2.45). This is possible by calling `store_gradient` on the learning rule instance after forwarding the whole input sequence, giving the opportunity to post-process accumulated information to a weight update. The update is then stored in the projection's parameter tensor as `grad` [Paszke et al., 2019] or applied directly to the synaptic weights. While the first allows taking advantage of `PyTorch`'s optimizers to optimize the weights more sophisticatedly by incorporating momentum into the updates, the second is used to model momentum-free updates as expected on-chip when using the PPU (see Chapter 6).

Eligibility Vectors Eligibility vectors are a crucial element for learning in spiking networks. Therefore, they are implemented as a synapse-specific property and are given by

the pre-synaptic spike events convolved with an exponential filter (see Equation (2.41)). In order to simulate hardware-like eligibility vectors, as required in Section 6.1.2, the vectors can be evolved in different modes. Additionally, the simulation supports fixed-pattern noise on the decay constants and amplitudes of the vectors for more realistic behavior (see Section 6.2.1).

4.2 Integrating HICANN-X

To enable learning on hardware, the chip is interfaced to the `PyTorch`-based `EProp` framework via `hxtorch` by utilizing lower-level chip abstraction layers such as `haldls` and `lola` [E. Müller et al., 2020b] encapsulated in `grenade`. Even though the software developed within `hxtorch` is used by `EProp`, it is not limited to this software layer but can be accessed, in principal, in all applications written in Python (and also C++).

Learning on HX can be realized in different ways. One approach is *in-the-loop* training, discussed in the next section. The implementation of a full *on-chip* learning environment is elaborated in Section 4.2.4.

4.2.1 In-the-loop Learning

The `EProp` framework outlined in the previous section is designed with the goal of performing learning on HX while supporting simulations. For training on HX, the network is mapped to a hardware representation, and the forward pass is executed on-chip rather than simulated in software. While emulating the network on HX, observables of interest are recorded and read back to the host computer after processing the whole time sequence. With these observables, the weights are then optimized on the host side. This procedure is repeated until convergence of the network and is hence referred to as hardware-*in-the-loop* training [Schmitt et al., 2017].

In-the-loop learning is realized for RSNN with a topology described in Section 2.3.1 by a C++ `hxtorch` layer, exposed to Python as `recurrent_to_readout`:

```
1 y, z = hxtorch.recurrent_to_readout(inputs, w_ih, w_hh, w_ho, runtime=runtime)
```

Basically, this layer handles all hardware mapping and communication implicitly. Simply calling the layer will execute a forward-pass on HX. The layer gets the `input` spikes of the input layer as a `torch.Tensor` of shape `(n_events, 3)`. Each element in this tensor is given by `(spike_time, population_id, neuron_id)`, where `spike_time` defines in FPGA clock-cycles the time input neuron `neuron_id` spikes. `population_id` associates neurons with a population (for experiments conducted in this thesis, `population_id=0`, since the network considered uses only a single input population). Further, the network's dense weight tensors `w_ih`, `w_hh` and `w_ho` are passed to the layer. Due to the 7 bit

resolution of signed hardware weights (see Section 4.2.3), synaptic weights are provided as 7 bit signed integers. The `runtime` argument limits the hardware experiment execution to the desired time frame. Since during execution, the membrane potential of the readout neuron is measured with the MADC, which can only sample one hardware neuron, the layer supports only a readout population of size one. The corresponding membrane trace is returned as a sequence `y` that holds the membrane samples annotated with timestamps. Spike events of the recurrent neurons are given in a tensor `z` of shape `(n_events, 3)` with elements `(spike_time, rec_population_id, neuron_id)`. Here is `rec_population_id` the population ID of the recurrent layer and `neuron_id` the ID of a recurrent neuron within this layer.

Weight Updates For an learning rule that only depends on the spike trains z_j^t and the potential of the readout neuron $y_{k=0}^t$ (as derived in Chapter 5), the layer provides all required information to optimize weights. Therefore, in each training epoch, the experiment is executed on hardware, and the returned observables are mapped to a discrete time lattice of desired resolution. In order to calculate the weight updates, the network is simulated in software. However, instead of simulating neuron dynamics, the measured observable states $\mathbf{z}^t$ are injected at each time t into the network's populations, with the effect that the simulation resembles the spiking activity of the hardware run. This allows the learning rules to compute weight updates like in a simulation run. Note, since the weights are discrete on HX, the updates need to be applied stochastically as described in Section A.2.1. Hence, the `EProp` experiment framework provides an adjusted version of `PyTorch`'s `Adam` [Kingma et al., 2017] optimizer that optimizes updates with momentum but applies them as integers.

As a convenient consequence, the implementation of in-the-loop learning and the design of the `EProp` framework make experiments and learning on HX behave almost identical to simulations. Nonetheless, in-the-loop learning is considerably slower than later full on-chip learning, described in Section 4.2.4, since the network has to be mapped to HX in each training epoch due to changing weights.

4.2.2 Interfacing HX

The `recurrent_to_readout hxtorch` layer uses the `grenade` [Spilger, 2021] interface for experiment execution. The `grenade` C++ interface allows mapping networks to a hardware representation by describing the network in a `NetworkBuilder`:

```
1 grenade::vx::network::NetworkBuilder network_builder;
```

This builder provides functionality similar to the `AbstractNetwork` class to `add` populations of neurons. Therefore, a population of software neurons is represented by an array of hardware neurons (`AtomicNeuronOnDLS`) and added to the builder. A boolean `record_spike` flag indicates whether the population's spike events are recorded:

```
Population::Neurons neurons{AtomicNeuronOnDLS(...), ...};
Population population{neurons, record_spikes};
auto const population_descriptor = network_builder.add(population);
```

Input populations correspond to externally injected spike events and do not reserve hardware neurons. These types of populations are fully defined by the size of the input layer. The `hxtorch` layer in the previous section defines three populations: The input population, the recurrent population, and a size-one population for the readout neuron. Projections between a `source` and a `target` population are added by

```
Projection::Connections projection_conns;
projection_conns.push_back({i, j, Weight(std::abs(w_ij))}); // Synapse ij
... // Add all connections
Projection projection{Projection::ReceptorType::excitatory,
    projection_conns, source_descriptor, target_descriptor};
network_builder.add(projection);
```

Since `grenade` projections are receptor-type specific[2] (i.e., excitatory *or* inhibitory) and software projections consist of signed weights (i.e., excitatory *and* inhibitory), each software projection is mapped to two hardware projections; An excitatory projection representing all connections with positive weights and an inhibitory one for the negative weights. Hence, the `recurrent_to_readout` layer uses six different hardware projections.

Finally, the information in the builder is used to define an event routing on HX. This is done by the `build_routing` function developed within the scope of this thesis and explained in Section 4.2.3. The experiment is described as a computational graph by using the routing result together with the network and executed via `run` by taking into account the network's `inputs`:

```
auto const network = network_builder.done();
auto const routing_result = build_routing(network);
auto const network_graph = build_network_graph(network, routing_result);

auto const result_map = run(*hxtorch::detail::getConnection(),
↪  hxtorch::detail::getChip(), network_graph, inputs);
```

Note, the inputs here are spike events translated implicitly from the input as `torch.Tensor` to hardware affine data types. After execution, the desired observables are accessed in the `result_map`, processed, and returned as `PyTorch` data types. A dedicated interface for MADC recording of the readout neuron's membrane is also provided by `grenade`. More details can be found in [Spilger, 2021].

[2] This corresponds to the current software state. In the future, `grenade` projections with signed synapses are desirable.

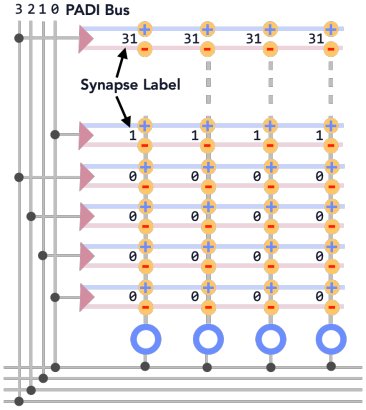


Figure 4.3: Schematic of a synapse matrix on HX. Synapse drivers are connected alternately to four vertical PADI buses. Spike events provided on the PADI bus can only reach the connected synapse drivers. Each PADI bus has 32 synapse drivers. The neurons are connected to four horizontal lines alternately in blocks of 32 neurons. External and internal spike events are routed by a crossbar (lower left). Events can only "travel" along connected lines. A synapse driver's rows are configured inhibitory (bottom) and excitatory (top) in order to realize signed synapses.

4.2.3 Routing Algorithm

As mentioned, `hxtorch` creates the network representation on hardware by `grenade`'s `build_routing` algorithm. Per default, this algorithm does not support recurrent projections and hence is adjusted [Czierlinski, 2020]. The developed implementation is based on signed synapses. This is, each signed software weight reserves two hardware synapses — one on an inhibitory row and one on an excitatory row in the same synapse column and the same synapse driver. Therefore, on each used synapse driver, the bottom row is configured inhibitory and the top row excitatory (see Figure 4.3). The absolute value of a negative software weight is placed on the inhibitory row while setting the weight on the excitatory row to zero and vice versa for positive weights. In effect, a software weight $\theta_{ji}^{\mathrm{sw}}$ is then represented on HX by signed integers $\theta_{ji}^{\mathrm{hw}} \in [-63, 63]$.

The placement algorithm expects dense software projections, split by the sign of the weights into two disjoint `grenade` projections — an inhibitory and an excitatory one — with the same source and target population. Each synapse in the internal projections (i.e., no external source population) is then placed one after another. Therefore, the algorithm finds the next synapse driver on a PADI bus (see Figure 4.3) reachable from the synapse's *source* neuron whose column to the *target* neuron is not occupied by another used synapse. Then the synapse's signed weight is placed on the corresponding driver in the target neuron's column together with a synapse label. Note, the algorithm sets the synapse label on the inhibitory *and* excitatory row and is given by the index of the source neuron on its crossbar channel. This allows using all synapse drivers on a hemisphere of HX. A source neuron's event address associated with a specific synapse driver is configured equal to the synapse labels in the driver's rows, such that the neuron's output events are matched in these synapses. Synapse drivers used for internal projections have `row_address_compare_masks` with bits disabled for the number of synapse drivers used on the PADI bus.

External populations are realized by injecting external events representing external neurons by event addresses that target synapses on rows of a dedicated driver. The events are injected at the corresponding PADI bus and carry addresses with a `row_select_address` and a synapse label both, somewhat redundant given by the driver's ID on the

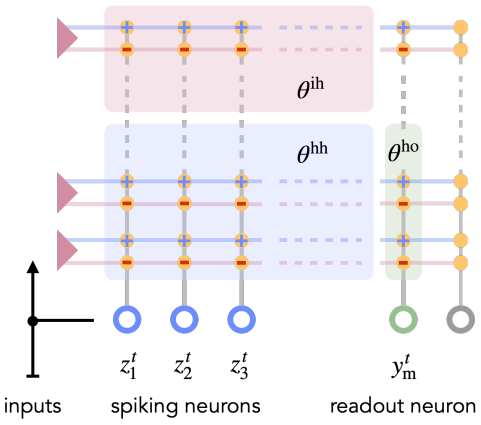


Figure 4.4: To account for negative and positive software weights, the `grenade` routing algorithm allocates two hardware connections on the same synapse driver for each software connection. One row mode on a synapse driver is configured excitatory, corresponding to positive weights in this row, and the other one as inhibitory, modeling negative weights. The recurrent spike events z_j^t are fed back via the crossbar and projected onto the recurrent layer as well as the readout neuron, thus the recurrent weights θ^{hh} and the output weights θ^{ho} are placed next to each other. The input weights θ^{ih} are placed above the recurrent projection. Note, synapse drivers are grouped chronologically by PADI buses for illustration purposes.

PADI bus. Used synapses on the rows of the synapse drivers are labeled therefore equally. For the external synapse drivers to be able to select external events addressed to them, their `row_address_compare_mask` is set to `11111`. In order to place the connections from an external to an internal population, the algorithm finds the next unused driver that contains free synapses to the target neuron and places labels and weights accordingly. Further, the algorithm assigns each neuron in an external population its event address which can be accessed with `grenade`'s interface to provide corresponding event `inputs` to the graph execution function `run` (see Section 4.2.2).

Assuming a network as in Section 2.3.1 and grouping drivers chronologically by PADI buses the routing algorithm finds a network representation on hardware as given in Figure 4.4.

Implicit Constraints Since this algorithm uses signed synapses, rows of a software projection are associated with a synapse driver; therefore, on a single synapse matrix on HX, this allows for a maximum of 128 software rows. For a topology as in Section 2.3.1 with n_{i} input and n_{h} recurrent neurons, applies $n_{\text{i}} + n_{\text{h}} \leq 128$. Further, for internal populations that feedback events into the synapse matrix, the algorithm only allows the usage of maximally 32 neurons connected to the same PADI bus since each PADI bus only provides 32 signed synapse rows. So far, connection placement is only supported on the top hemisphere on HX.

4.2.4 On-chip Learning

For on-chip learning, a pre-compiled plasticity program is loaded onto the PPU, which computes weight updates in parallel to the forward pass. Regardless of the explicit form of the learning rule implementation, a general software architecture for learning on-chip is proposed in the following and depicted in Figure 4.5.

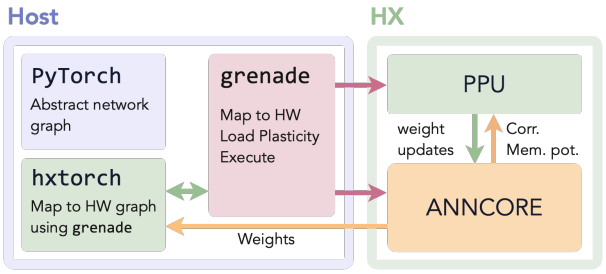


Figure 4.5: Software stack for on-chip learning. `grenade` loads a plasticity program onto the PPU, communicates parameters and triggers the start of the plasticity rule. After a batch of training trials is completed, the weights are read back and translated to software weights stored in the model as `torch.Parameters`. On a test trial, the neuron spikes and the membrane potentials of the readout neurons are sampled and returned.

As a high-level interface, a `PyTorch` model is derived from a `pybinded` [Jakob et al., 2017] `hxtorch` class `RecurrentToReadout` written in C++ that inherits `torch::nn::Module` [Paszke et al., 2019] to provide the common `PyTorch` infrastructure:

```
class Model(hxtorch.RecurrentToReadout):
    def __init__(self, ...):
        super().__init__(n_i, n_h, n_o, fit_slice=True)
    ... # Implement other stuff

model = Model(...)
model(inputs, runtime) # Training
model.eval() # Inference
z, v = model(inputs, runtime)
```

The `hxtorch` class assumes a fixed network topology as given in Section 2.3.1 with `n_i` input, `n_h` recurrent, and `n_o` output neurons. Its `forward` method implements the `grenade` interface as outlined in the previous section, however, with additional batch support. This allows performing multiple forward passes after another within the same `grenade` graph instance, resulting in an enormous speed up in the training process. When forwarding a training batch, the model performs on-chip plasticity implicitly without returning any observables. Hence, to test the model's performance in an inference run, `model.eval()` disables PPU learning and configures the forward method to record recurrent spikes and the membranes of the readout neurons. Since a PPU program records the membranes via the CADC in parallel, this on-chip learning setup allows multiple readout neurons.

Per default, the neurons are placed column-wise chronologically on HX, i.e., `n_h` recurrent neurons followed by `n_o` output neurons. If `fit_slice=True`, only every second hardware neuron is used. In that way, neurons and related synapse columns are placed on the same vector "slice". Since a single VU readout accesses either all odd or all even synapse columns on HX row-wise in parallel[3], this effectively reduces the number of instructions needed to read out hardware observables and to process them (since all observables are present in a single vector) via the VU.

[3]This is only true for the current chip version. Older chip versions have a different column ordering.

Reverse Weight Mapping The main difference between on-chip learning and the described in-the-loop training is that weights change during experiment execution on HX. Therefore, after forwarding a batch, it is required to read back hardware weights and map them to software weight tensors. Hence, the `hxtorch` forward method employs an inherent reverse mapping from hardware projections to signed `torch.Parameter`s by reading out the hardware weight matrix after batch execution is completed and inferring hardware weight placement from the connections defined by `build_routing`. It is desirable that such a reverse mapping is performed implicitly within `grenade` as `grenade` also creates the network's hardware representation and has all mapping information at hand. This, however, is not implemented yet.

Triggering Plasticity In order to deploy this high-level on-chip learning framework, `grenade` is adjusted to trigger PPU plasticity programs. This is enabled by endowing the `grenade` graph execution with an optional `setup` object:

```
auto const result_map = run(*hxtorch::detail::getConnection(),
↪  hxtorch::detail::getChip(), network_graph, inputs, setup);
```

This setup provides PPU plasticity-specific functionality such as enable/disable plasticity and enabling serialization of plasticity parameters (see Section 4.2.5):

```
struct Setup {
    bool write_settings = false;
    bool enable_plasticity = false;
    bool set_masks = true;
    ...
    struct Config {
        PPUParams params = PPUParams();
        ...
        row_type row_mask;
        UpdateParams sample(size_t epoch);
    };
    std::map<PPUOnDLS, Config> ppu_configs;
};
```

Further, the `setup` object holds a `Setup::Config` instance for each PPU on HX in `ppu_configs` which contain a `PPUParams` object with all constant parameters required by the PPU to perform learning. In order to make this learning setup configurable in Python, the `RecurrentToReadout` class has a `Setup` instance as a member, which is and exposed to Python. When calling the `forward` method, the configured setup is passed to `grenade`. Before batch execution, `grenade` communicates the `setup.params` object to the PPU to set up the learning rule.

When `grenade` executes experiments on HX, it starts a default PPU program which enters a `while`-loop around a `switch` with a case variable of type `Status`, controlled by

the host computer within `grenade`. This switch is expanded by three cases to control on-chip learning:

- `Status::setup_plasticity`: Triggered before executing the given batch. This enables deserialization of the `PPUParams` object written into the PPU's SRAM via `bitsery` [Vinkelis, 2020] (see Section 4.2.5).
- `Status::update_plasticity`: Triggered before each execution of each element in the batch after `grenade` wrote an `UpdateParams` object into the PPU's SRAM. This object contains trial-specific configurations (such as the synapse row to perform updates for, random offsets, etc., see Chapter 6) and is created by `Setup::Config::sample`.
- `Status::start_plasticity`: Triggered right before emulating the network for the given batch element. This command starts the PPU plasticity program. After the network is emulated, the plasticity program is stopped and `Status::update_plasticity` repeated for the next batch element.

These steps are repeated for each batch during the training process.

Communicating Network Topology to the PPU For the PPU to be able to address correct synapses for weight updates, it requires knowledge about the network's hardware topology. Firstly, this applies to used synapse columns defined by the hardware populations — more explicitly, the recurrent and the readout neurons — and, secondly, the used synapse rows. Synapse columns are communicated by boolean mask arrays `spiking_mask` and `readout_mask` in `setup.params`. The distinction between recurrent and readout neurons is necessary since the corresponding projections have different plasticity rules. Masks are derived implicitly in the `forward` method if `setup.set_masks=True` and need to be set manually otherwise.

The routing algorithm in Section 4.2.3 identifies rows of the software weight matrices with synapse drivers (due to signed weights). Hence, used signed hardware rows are given in `row_mask` as a dictionary `{driver_id: (neuron_rec, is_rec)}`, where `driver_id` is the signed row index, `is_rec` indicates whether the row is used by the recurrent projection, and `neuron_rec` gives the hardware neuron index projecting its events in this row. The routing algorithm places the recurrent and the output projection next to each other (see Figure 4.4). Hence, `is_rec` indicates whether the update rule has to perform weight updates for the recurrent *and* output weights. Since weight updates for a row might depend on observables of its corresponding neuron (such as spike counts), the index `neuron_rec` is crucial.

Note, the general infrastructure does not depend on the explicit form of `Config` and can be adjusted as needed; however, it should have members `params` and `sample`. All other functionality applies to the on-chip training performed in Chapter 6.

Debugging For debugging purposes, the PPU's mailbox writer is redirected to write at a specific address range of the external memory. Thus, the `hxtorch` class provides

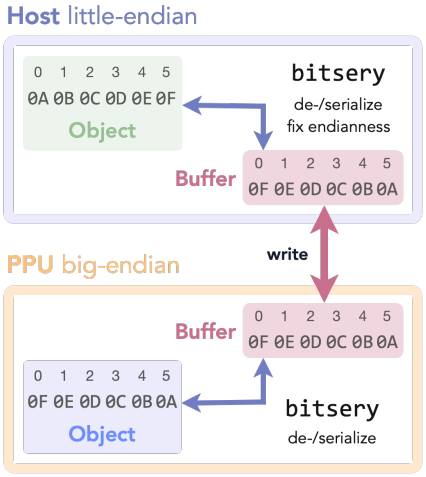


Figure 4.6: The host to PPU communication is realized by `bitsery`, which enables the serialization of arbitrary objects. An object containing all parameters needed on the PPU is serialized into a buffer on the host side. This buffer can easily be written into a buffer on the PPU, and deserialized into a corresponding object, which then contains the desired data. Since the PPU is a big-endian system, whereas the host is a little-endian, the host-side serialization is configured to take the different endianness into account implicitly. The computational cost to translate between different endieanness is thus on the host-side. `bitsery` is easing the host to PPU communication enormously.

a method `read_mailbox(n_words, print=False)` which reads back data from the external memory and returns it as a byte-array. On demand, the data is reinterpreted as strings and printed out. Since debugging software for embedded systems, such as PPU programs, is always challenging, this feature is very convenient.

4.2.5 Host-PPU Communication

Data transfer from the host to the PPU is realized by the C++ header-only serialization library `bitsery` [Vinkelis, 2020]. On the host side, the software layers `halco` and `haldls` [E. Müller et al., 2020a] allow creating PPU-memory blocks that can be written (read) word-wise at (from) specific address blocks of the PPU memory. Therefore, the host computer can manipulate data on the PPU at runtime. However, turning arbitrary complex C++ data types manually into memory blocks is inconvenient. Using `bitsery`'s serialization tools, (almost) any object can be serialized via a `Writer` into a `buffer`, given as a byte array that can easily be formed to a PPU-memory block and written to the PPU:

```
Object object_host; // Some object on host
uint32_t size = bitsery::quickSerialization<Writer>(buffer, object_host);
... // Write size and buffer to PPU
```

On the PPU, the buffer is deserialized via a `Reader` into a corresponding object in the PPU-memory:

```
Object object_ppu; // Some object on PPU
uint32_t size;
auto status =
    bitsery::quickDeserialization<Reader>({buffer.begin(), size}, object_ppu);
```

An important feature of `bitsery` is that the `Writer/Reader` for de-/serialization on the host-side can be configured to assume a particular endianness. Since the host computer is little-endian whereas the PPU is a big-endian system, the de-/serialization on the host-side translates between the different endianness implicitly. The schematic is pictured in Figure 4.6.

In order to enable de-/serialization via `bitsery`, the corresponding data object has to provide a `serialize` method, defining how members are treated. For serializing `std::array<uint8_t, 128>` from the host into PPU-specific vector types `__vector uint8_t` on the PPU, a customized treatment is developed which reinterprets the data type implicitly when deserialized on the PPU. This enables serialization of mask arrays into data types the VU can work on. Hence, `bitsery` allows seamless host-PPU communication.

5 Spike-based Eligibility Propagation

The learning algorithms derived in 2.3.3 cannot trivially be utilized to train *recurrent spiking neural network*s (RSNNs) on the *HICANN-X v2* (HX). A first attempt to train RSNNs on the chip can be realized by incorporating a simplifying hardware constraint into the learning rule. Hence, this chapter will define a *spike-based* version of e-prop that allows training on HX *in-the-loop*. However, this learning rule comes with some drawbacks and will, therefore, first be simulated and the shortcomings discussed. It will turn out that under little constraints, the approximated learning rule enables RSNNs to learn still surprisingly well. Supported by these simulations, the network will then be trained in-the-loop, where the forward-pass is emulated on HX, and the gradient is computed on the host-side. This will show that the a pattern-generation task, described in 5.1.2 can, principally, be solved on HX while introducing a constraint that later on-chip learning has to do implicitly.

5.1 Task

The e-prop learning framework outlined in Section 2.3.3 is, in principle, applicable to any definition of a loss function E; however, not all resulting learning rules are realizable on hardware. Here, a pattern generation-task is motivated in the following and described thereafter.

5.1.1 Motivation

Since this thesis aims for an e-prop-inspired on-chip learning rule implementation, the task considered needs to align with the chip's limits. This means, the chosen task must not have a loss function E that results in learning signals (and thus in learning rules) which explicitly depend on observables the *plasticity processing unit* (PPU) cannot access. This categorically excludes tasks depending on explicit spike events, such as spike-pattern generation, since the PPU cannot (efficiently) access the neurons' spike times. However, loss functions utilizing the membrane potentials of the readout neurons are promising candidates since the membranes can be read out by the PPU using the CADC, provided the resulting learning rule allows sampling the readout neurons with a *feasible* rate. Such tasks comprise the pattern-generation task explained in the following, but also classification tasks where sub-sequences of the membrane potential are mapped to probabilities via the softmax loss (see [Bellec et al., 2019] for details).

5.1.2 Description

For the regression task mentioned in Section 2.3.3, a recurrent network is trained to modify the membrane potentials y_k^t of a set of readout neurons $\{n_k\}$ such that the membrane traces minimize the *residual sum of squares* (RSS) to corresponding target traces $y_k^{*,t}$ over a time sequence of length T. The target traces can, for instance, be joint angle velocities of a robotic arm the readout neurons have to resemble. Here, the target traces $y_k^{*,t}$ are simply a superposition of sinusoids,

$$y_k^{*,t} = \xi_k \sum_i^{n_p} w_{i,k} \sin\left(\frac{2\pi}{T_{i,k}} \cdot t \cdot \delta t + \varphi_{i,k}\right), \tag{5.1}$$

with randomly sampled weights $w_{i,k}$, periods $T_{i,k}$, and phases $\varphi_{i,k}$. ξ_k denotes a scaling factor adjusting the target pattern such that $\max(\left|y_k^{*,t}\right|) = \eta_T$, with η_T being the desired extremum. The time steps t are restricted to the sequence length T, $t \in [0, T)$. This pattern-generation task is inspired by [Bellec et al., 2019]. For this task, the learning rules are given in Equation (2.53) and (2.53).

5.2 Adjusting the Learning Rule

This section will derive an approximated version of the learning rule given in Equation (2.53), allowing to train recurrent networks in-the-loop on HX. While this will result in a simplification of the learning rule, it is also a necessary step towards full on-chip learning.

The eligibility traces e_{ji}^t in the learning rules for the recurrent and input weights depend on the membrane potentials v_j^t via pseudo derivative h_j^t. For full on-chip learning using the PPU, this poses a problem since the membrane potentials of all recurrent neurons need to be read out in parallel with a high temporal resolution. In principle, this can be achieved with a sampling rate of about $1.7\,\mathrm{ms_{bio}}$ using the CADC [Cramer et al., 2020], which (depending on the speed of the neuron dynamics) arguably is sufficient to resolve the pseudo derivative. However, the PPU would have to handle a great amount of data that either needs to be stored during the forward pass and fetched afterwards to compute the gradient on-chip or the measured membrane potentials need to be used to calculate the gradient in an online fashion. While the first causes memory issues on the PPU, the latter would result in a decreasing sampling rate due to intermediate operations occupying the PPU. Hence, a learning rule independent of the membrane potentials of the recurrent neurons needs to be found.

Therefore, the pseudo-derivative h_j^t in Equation (2.49) is simply replaced with the recurrent spike train z_j^t. Correspondingly, the eligibility traces become

$$e_{ji}^{t+1} = h_j^{t+1} \cdot \hat{z}_i^t \tag{5.2}$$

$$h_j^t \approx z_j^t \Longrightarrow e_{ji}^{t+1} \approx z_j^{t+1} \cdot \hat{z}_i^t \coloneqq \hat{e}_{ji}^{t+1}. \tag{5.3}$$

Replacing the eligibility traces e_{ji}^t in Equation (2.53) with the approximation $\hat{e}_{ji}^t$ gives the update rule for the recurrent weights

$$\Delta\theta_{ji}^{\mathrm{hh}} = -\eta \sum_t \left(\sum_k \theta_{kj}^{\mathrm{ho}} \left(y_k^t - y_k^{*,t} \right) \right) \sum_{t' \leq t} \kappa^{t-t'} \hat{e}_{ji}^{t'}. \tag{5.4}$$

The same rule applies to the input weights θ^{ih} when replacing the recurrent spike train z_i^{t-1} with the input spike train x_i^t in the eligibility vector $\hat{z}_i^t$. The update rule for the output projection θ^{ho} remains unaffected since it does not depend on the membrane potentials of the recurrent neurons. The firing rate regularization in Equation (2.56) is also subject to the approximation in Equation (5.3). Although not outlined explicitly, the idea of replacing the pseudo-derivative with the post-synaptic spike train is also mentioned in [Bellec et al., 2019].

5.2.1 Consequences

The learning rule in Equation (5.4) is still not implementable on the PPU since it depends on explicit spike times. However, it can be implemented on HX with in-the-loop training. The recurrent spike events z_j^t can be accessed via the *field-programmable gate array* (FPGA) on the host-side, allowing computing the eligibility traces on the host engine. When assuming a network with a single readout neuron $k = 0$, the readout membrane trace $y_{k=0}$ can be measured by the chip's MADC, which is capable of sampling the membrane potential of a single neuron with a high temporal resolution. Now, where all needed observables $\{y_{k=0}, \{\mathbf{z}^t\}\}_{t=1}^T$ can be accessed on the host-side, the forward pass can be emulated on HX while the weight updates are calculated off-chip.

The approximation made in Equation (5.3) does affect learning. In Figure 5.1 the approximated eligibility traces are compared to the true traces. The upper plot visualizes an artificial spiking setup of a LIF neuron, receiving pre-synaptic events through synapse ji, causing the neuron j to emit post-synaptic events itself. The second plot shows the corresponding eligibility vectors $\hat{z}_i^t$ together with the pseudo-derivative h_j^t and the recurrent spike train z_j^t. Each pre-synaptic event starts a synapse-specific exponential decay with decay constant α, describing the eligibility vectors $\hat{z}_i^t$. According to Equation (2.49), these vectors are scaled by the pseudo-derivative h_j^t, giving the eligibility traces depicted in the third plot. With the approximation made, the eligibility traces now have only non-vanishing values at post-synaptic events $z_j^t = 1$ and are then given by the value of the eligibility vector at the corresponding spike time. These traces are solely spike-based and have no contributions by the membrane potential of the recurrent neurons. In essence, this means the learning rule in Equation (5.4) does not rely on any sub-threshold information. Consequently, the learning rule cannot adjust weights such that the potential of a neuron close to spiking is modulated in a fashion that a spike in the future is inhibited or emitted. Since the approximated eligibility vectors are solely based on spike information propagated along the sequence, this learning rule is named *spike-based eligibility propagation* (s-prop) from here on.

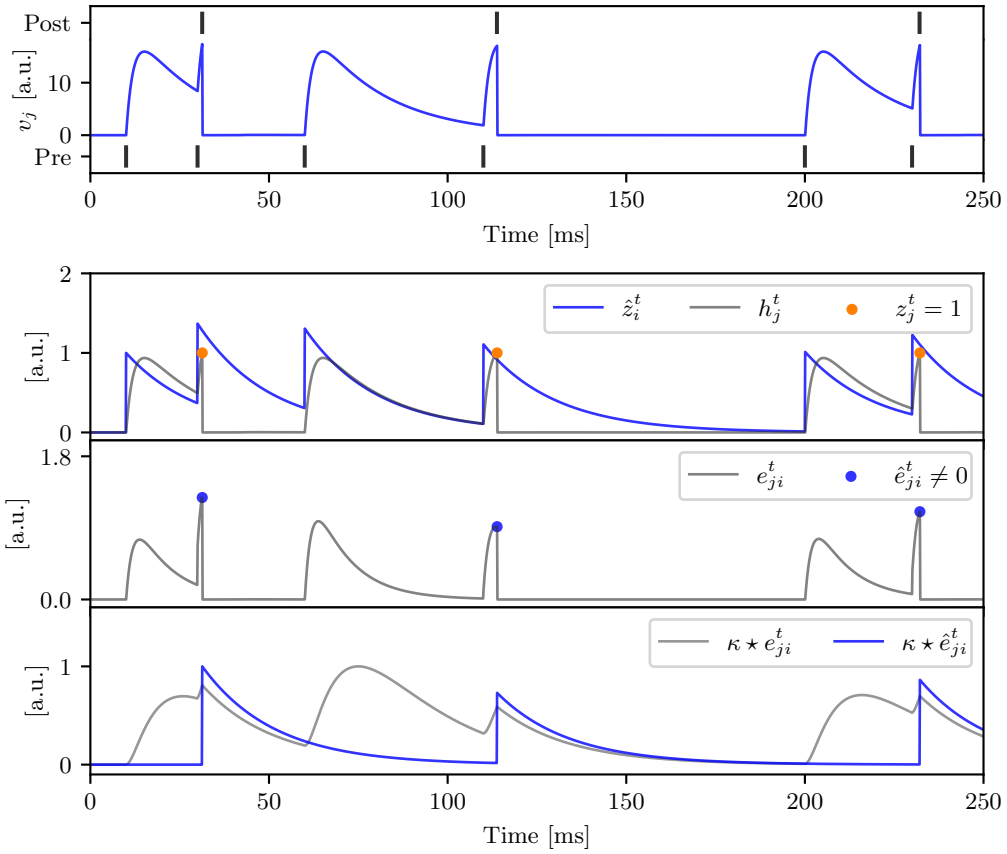


Figure 5.1: Comparison of eligibility traces. The upper plot shows an artificial spiking setup of a LIF neuron, receiving pre-synaptic events and emitting post-synaptic spikes. The second plot visualizes the corresponding eligibility vector $\hat{z}_i^t$ of a synapse ji together with the pseudo-derivative h_j and the recurrent spike train z_j^t. A step towards on-chip learning is to replace the pseudo derivative h_j^t with the recurrent spike train z_j^t. While this approximation has to be done to utilize the correlation sensors in later chapters, this will enable implementing HX in-the-loop training since then the weight updates do only depend on the recurrent spikes and the membrane potential of the readout neuron (assuming a single readout neuron whose membrane potential is sampled with the MADC). The recurrent spikes and the readout potential can be sampled with high resolution. As can be seen in the third graph, the eligibility traces $\hat{e}_{ji}$ under this approximation are only the product of recurrent spikes and the eligibility vector. Therefore, this approximation neglects essential information about the neurons sub-threshold behavior. For instance, the eligibility traces provide no information about whether a neuron is close to spiking at a certain time or not but do rely solely on spike events. The lowermost plot shows the eligibility traces filtered with the readout kernel κ. Note, the filtered traces are scaled for illustrating purposes.

However, since the eligibility vectors propagate synapse activation information into the future, the weight update has a non-vanishing contribution at post-synaptic spike times $z_j^t = 1$, i.e. $\hat{e}_{ji}^t > 0$. Hence, past activity at $t' < t$ is held accountable for the network's error at time t (given by the learning signal) and thus allows changing the weights such that an error at t is minimized by adjusting the network's activity at earlier times. Therefore, the approximated learning rule seems promising to endow RSNNs with the capability to learn.

According to Equation (5.3), the eligibility traces $\hat{e}_{ji}^t$ are always zero if neuron j does not emit a spike over the whole time sequence, leading to vanishing weight updates. Consequently, if a neuron j is silent the update rule is not capable of adjusting the corresponding weights due to missing sub-threshold information. Hence, neuron j stays silent except if other neurons adjust their activity such that neuron j emits a spike coincidentally.

In Equation (5.4), the eligibility traces are convolved with the readout decay κ, taking the temporal extent of the readout neurons $\{n_k\}$ into account. This is shown in the lowermost plot of Figure 5.1. Intuitively, if a readout neuron receives a pre-synaptic event, its membrane potential is deflected, after which it strives back towards the resting potential with time constant $\tau_{\mathrm{m},k}$; a pre-synaptic spike at t, hence, contributes to the error at $t' \geq t$. Finally, the convolved eligibility traces are weighted by the learning signal at each time and summed over the whole sequence, giving the weight update.

After describing the s-prop learning rule it is now evaluated for the task explained in Section 5.1.2. After verifying the rule in simulations in Section 5.3, it is used to train RSNNs on HX in Section 5.4.

5.3 Simulations

Testing the update rules given in Equation (5.4) and Equation (2.54) in simulations will help to find a setting for which training on HX seems possible. Therefore, the simulations have to take several hardware properties into account. Please note that the simulations done in the following are meant to show that s-prop enables RSNNs to solve the pattern-generation task in principle while capturing important hardware characteristics. However, the simulations do not claim to mimic the chip entirely.

5.3.1 Hardware Constraints

With regard to in-the-loop learning, hardware-specific aspects that might influence learning include noise in the network emulation process, weight discretization, and the synaptic input current. These properties are briefly discussed in the following.

Noise Parameters that are subject to fixed-pattern noisy due to their analog nature on HX are the synaptic time constant τ_{syn}, the membrane time constants τ_{m}, and the synaptic strength [Dauer, 2020]. Simulations of the HX environment assume the fixed-pattern noise of these parameters to be normally distributed around their target values. Additionally, all neurons integrate Gaussian noise onto their membrane potential. This is achieved by adding a random offset, sampled from a normal distribution with zero mean, to v_j^t in Equation (2.35) and y_k^t in Equation (2.36) after each numerical integration step.

Discrete Weights Since the synaptic weights on HX are given by signed integers $\theta_{ji} \in \mathbb{N}_{-63}^{63}$, the weights in simulations are discretized equally and saturate at the given boundaries. Therefore, the weight updates are rounded stochastically to corresponding integers, as described in Section A.2.1, with $\Delta\theta^{\text{hw}} = 1$. Stochastic rounding is the preferred way to handle updates calculated with floating point numbers as this increases the resolution of the weight updates artificially.

Synaptic Input The e-prop learning rules derived in Chapter 2 assume synapses with a δ-kernel (see Equation (2.13)). However, on HX, the synapses are emulated with a single-exponential kernel in analog. Even though the e-prop learning framework allows for deriving update rules for exponential synaptic kernels, they lack a form that enables on-chip learning and would need to incorporate the assumption of δ-like synapses.[1] Therefore, the synaptic time constants are chosen small compared to the membrane time constants in order to approximate the effect of δ-synapses.

Additionally, the simulated network should only operate on parameter ranges feasible on HX. The used parameters are outlined in the following.

5.3.2 Network Setup and Training Procedure

Training an RSNN is manifold, requiring multiple factors to interact in harmony to enable successful learning. LIF parameters governing the neuron dynamics, network complexity, and learning parameters have to be adjusted appropriately. Therefore, the shared learning and network setup for the simulations is outlined in the following. It is crucial to keep in mind that the focus of this thesis is to examine the feasibility of e-prop inspired learning rather than finding fine-tuned networks. Hence, some hyperparameters chosen in the following remain subject to further optimization but are found to work well for the task at hand.

[1] Learning rules with single-exponential kernels use double-exponential filtering of pre-synaptic events to compute the eligibility vectors and can, therefore, only be modeled by the correlation sensors (*cf.* Chapter 6) when assuming a δ-kernel (resulting in the eligibility vectors as given here).

Network

Unless stated otherwise, the network consists of an input layer with 30 input neurons projecting onto a recurrent layer constituted by 70 LIF neurons. Spike events of all recurrent neurons are projected onto a single *leaky integrate* (LI) readout neuron as well as the recurrent layer itself. The neurons are implemented as given in Section 4.1.2. All projections between layers are dense. The architecture is chosen such that the limitations of the event routing algorithm on hardware (see Section 4.2.3) are not exhausted. Input neurons are firing Poisson distributed with a mean *inter-spike interval* (ISI) of $T_{\text{ISI}} = 40\,\text{ms}_{\text{bio}}$.

Inspired by Bellec et al. [2019], the recurrent neurons $\{n_j\}$ have vanishing leak and reset potentials, $v_{\text{l},j} = 0\,\text{a.u.}$ and $v_{\text{r},j} = 0\,\text{a.u.}$, respectively. The threshold is set to $\vartheta = 40\,\text{a.u.}$. This value is somewhat related to a relative leak-to-threshold potential of 40 DAC values of spiking neurons on HX (see Section 5.4.1). Analogously to the recurrent neurons, the readout neuron leaks towards $y_{\text{l},k=0} = 0\,\text{a.u.}$. All neurons in the network, except the input neurons, have a membrane time constant $\tau_{\text{m},j} = \tau_{\text{m},k=0} = 20\,\text{ms}_{\text{bio}}$. The refractory time τ_{ref} for spiking neurons is chosen as $\tau_{\text{ref}} = 1\,\text{ms}_{\text{bio}}$. This is a value realizable on HX and does not decrease the network's degree of freedom significantly. Synaptic time constants are chosen to be small compared to the membrane time constants. However, since longer constants effectively yield stronger synaptic inputs, they must not be too small to ensure a reasonable signal-to-noise ratio for experiments on HX (*cf.* Section 5.4.1) and are chosen as $\tau_{\text{syn}} = 2\,\text{ms}_{\text{bio}}$. This will turn out to work well.

Training

The networks in the simulations are trained over 1000 epochs, where each epoch consists of a single forward pass (trial). The weight updates for the input and recurrent projections are calculated according to Equation (5.4), the output projection by Equation (2.54), respectively. Weights are optimized after each epoch by the Adam optimizer with default parameters suggested in Kingma et al. [2017]. In the case of discrete weights, the modified optimizer mentioned in Section 4.2.1 (also *cf.* Section A.2.1) is used. The input weights are initialized normally distributed with $\mathcal{N}(\mu = 0, \sigma = 15)$ centered around zero. In order to reduce the number of silent neurons at the start of the training process, the recurrent weights are initialized close to zero and sampled from $\mathcal{N}(\mu = 0, \sigma = 1)$ since strong inhibitory recurrent connections in RSNNs tend to inhibit activity. This also is found to stabilize training. The output weights are set to zero at the beginning. Unrealistically high firing rates are avoided by regularizing the recurrent neurons towards an average event density of $f_{\text{av}} = 40\,\text{Hz}_{\text{bio}}$. The regularization update $\left(\Delta\theta_{ji}^{\text{ih, hh}}\right)^{\text{reg}}$ in Equation (2.56) is added to the s-prop update $\left(\Delta\theta_{ji}^{\text{ih, hh}}\right)^{\text{s-prop}}$ in Equation (5.4) before multiplying the sum with the learning rate η,

$$\Delta\theta_{ji}^{\text{ih, hh}} = \eta \left(\frac{1}{\eta} \left(\Delta\theta_{ji}^{\text{ih, hh}}\right)^{\text{s-prop}} + \left(\Delta\theta_{ji}^{\text{ih, hh}}\right)^{\text{reg}} \right). \tag{5.5}$$

Note that per definition, the learning rate η is also included in the s-prop update and chancels out the factor $1/\eta$ in Equation (5.5). The regularization strength is included in

the regularization update and chosen as $\eta^{\text{reg}} = 10000$. In noisy simulations, noise is applied as discussed in Section 5.3.1 with a standard deviation of 10% of the corresponding target value. The Gaussian noise on the membranes is sampled from $\mathcal{N}(\mu = 0,\sigma = 0.4)$. An overview of all parameters is given in Appendix A.1.

After conducting the hyperparameter-search in Section 5.3.3, the learning rate η_{r} for the input and recurrent weights, as well as the learning rate η_{o} for the output projection are chosen — if not stated otherwise — as $\eta_{\text{r}} = \eta_{\text{o}} = 0.05$ (learning rates are renamed to distinguish between weight matrices). The learning rates are multiplied by a decay constant of $\Gamma = 0.8$ every 200 epochs.

Target Pattern

As presented in Section 5.1.2, the network is trained to solve the pattern-generation task. The corresponding target pattern $y^{*,t}_{k=0}$ is given in Equation (5.1) by a superposition of $n_p = 3$ sinusoids with uniformly drawn weights $w_{i,k=0} \in [0.5, 2)$, periods $T_{i,k=0} \in [0.6\pi, 2\pi)$, and phases $\varphi_{i,k=0} \in \left[\frac{0.5}{1000}, \frac{2\pi}{1000}\right)$.[2] The patterns are rescaled to $\max(y^{t,*}_k) = \eta_T = 100$ in order to mimic a possible range of the readout neuron's membrane potential on HX (*cf.* Section 5.4.1). All networks simulated in the following are trained for an ensemble of 16 (8 for the hyperparameter-search in Figure 5.2) random patterns.

5.3.3 Baseline

Before conducting simulations incorporating hardware constraints, a model in absence of any hardware properties is considered. This shows that the network setup and training procedure, together with the learning rule in Equation (5.4), enables learning in RSNNs in principle and serves as a baseline later on.

In Figure 5.2, the MSE and the average firing rates of the recurrent neurons are depicted over epochs for a range of learning rates $\eta_{\text{o}}, \eta_{\text{r}} \in \{0.5, 0.05, 0.03, 0.005\}$. These experiments show that firing rates increase for low output learning rates η_{o}. Intuitively, small η_{o} result in a slower decreasing error $\left(y^t_{k=0} - y^{*,t}_{k=0}\right)$ due to slowly adjusting output weights θ^{ho}, and effectively, causing the recurrent neurons to fire stronger to account for the error. Especially for $\eta_{\text{o}} = 0.005$, this results in unstable learning. The same applies for too large learning rates η_{r}, except for $\eta_{\text{o}} = 0.5$, where the loss drops very fast. However, in this case, the firing rate of the network enters a state with very sparse activity. The best performance with a moderate activity is observed for $\eta_{\text{o}}, \eta_{\text{r}} = 0.05$ and chosen for experiments in the following.

The baseline experiment is shown in Figure 5.3 in comparison to e-prop. As indicated by the MSE, s-prop performs slightly worse than e-prop; however, it still solves the pattern-generation task well. The recurrent neurons learn to form spike clusters and specialize to specific time periods in order to minimize the error. In the upper-right plot,

[2] This has turned out to be a bug. However, verification has shown that this does not constraint the diversity and complexity of considered patterns.

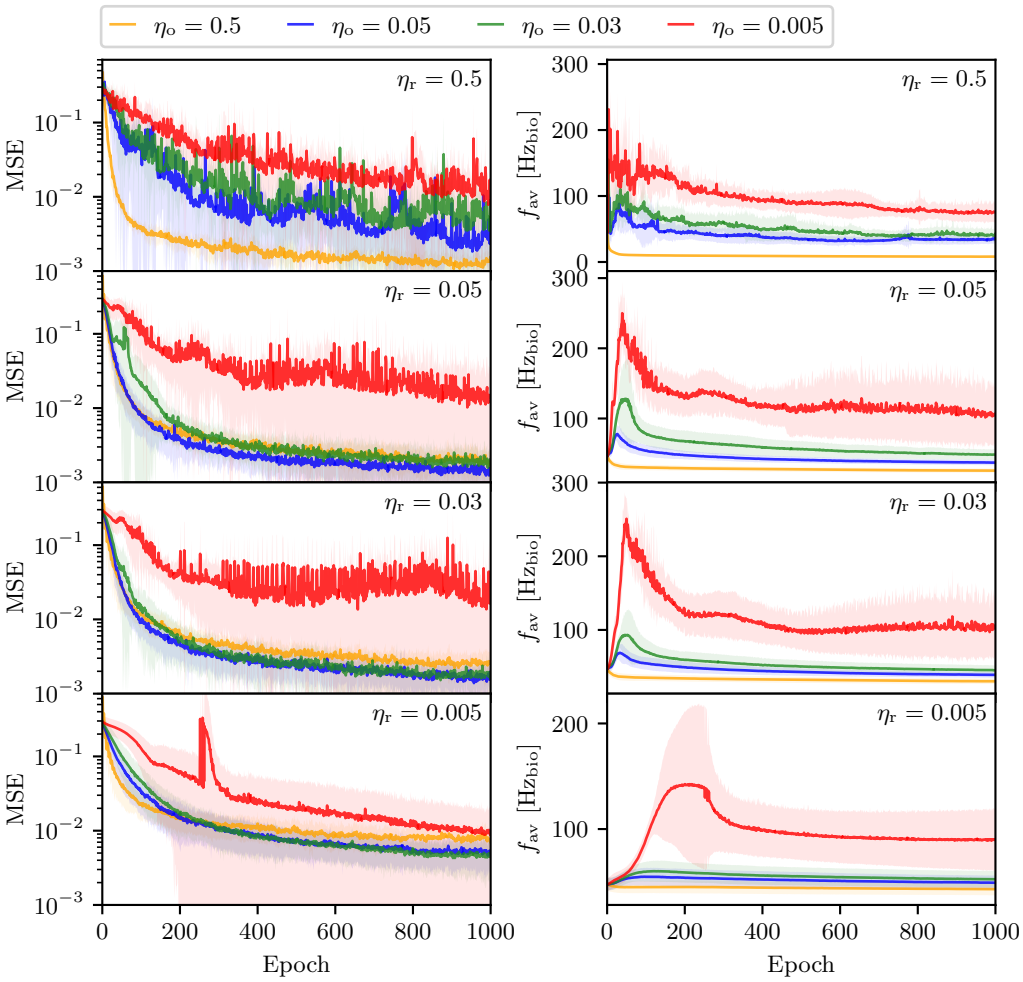


Figure 5.2: Experiments for a grid of learning rates $\eta_o, \eta_r \in \{0.5, 0.05, 0.03, 0.005\}$. **Left** plots show the MSE over epochs, the plots on the **right** the corresponding average firing rates. Output learning rates η_o are given in the legend, the learning rate η_r for input and recurrent projections are denoted on the upper-right of each individual plot. The confidence bands are standard deviations over the ensemble. Experiment parameters can be found in Table A.2.

the development of the spiking activity over the training epochs is compared to e-prop. While e-prop tends to converge to a state with a higher firing rate, s-prop training tends towards less dense spiking activities. Nevertheless, the standard deviation of the activity is larger for s-prop than for e-prop, indicating that the spiking activity depends more on the target pattern for s-prop. Even though the network and training process is not entirely fine-tuned, the baseline experiment shows that s-prop is able to solve the pattern-generation task sufficiently.

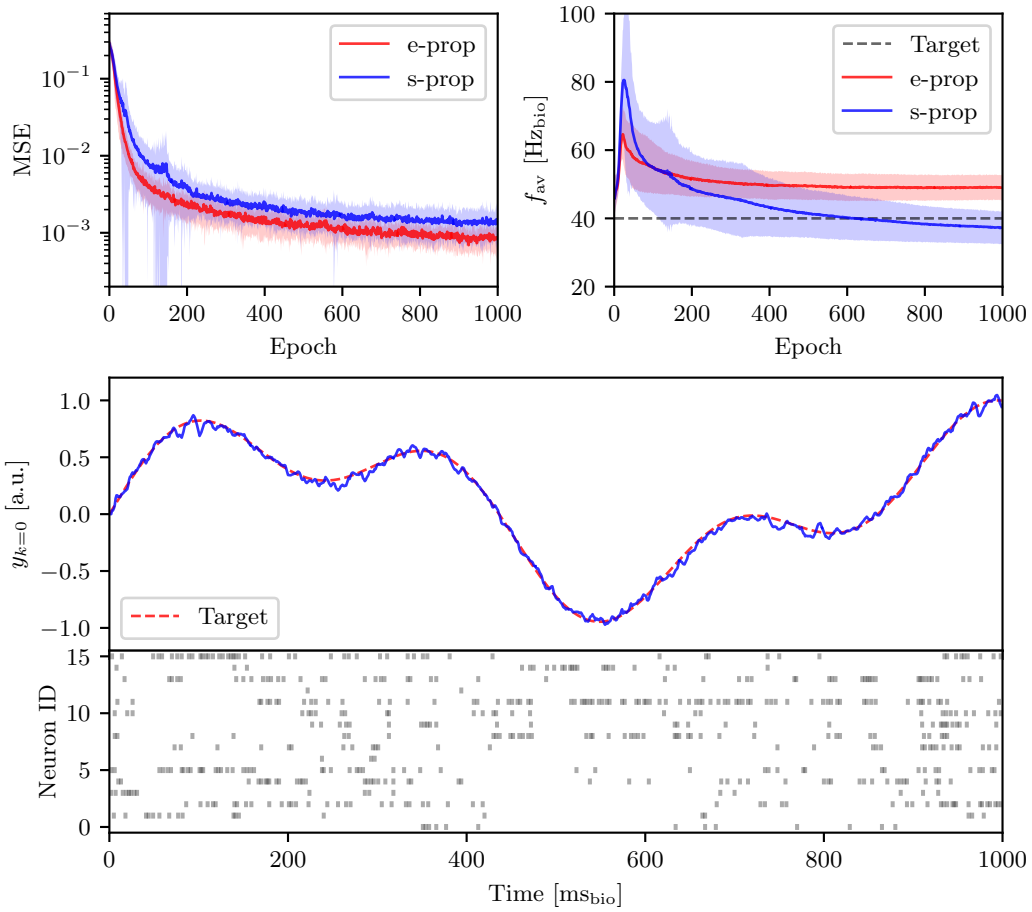


Figure 5.3: Comparison of e-prop and s-prop. s-prop can still solve the pattern-generation task well, however, the MSE is slightly worse than for e-prop. The plot in the **middle** shows the learned membrane trace after 1000 training epochs, compared to the target trace, which is resembled well. The **lowermost** plot depicts the spiking activity of the first 16 recurrent neurons. It can be seen, that s-prop enables the recurrent neurons to cluster their activity over certain time ranges. Hence, the recurrent neurons specialize to provide the right amount of events onto the readout neuron's membrane at the right times, in order to minimize the MSE. In the two uppermost plots, thick lines visualize the ensemble average, the confidence bands show the corresponding standard deviation. Parameters can be found in Table A.3.

5.3.4 Discrete Weights

The first hardware properties that are incorporated in the simulations are discrete weights and noise. Therefore, the weights $\theta^{\text{ih,hh,ho}}$ are limited to integer values as on HX and updated stochastically (see Section 5.3.1).

In Figure 5.4 the MSE and the average activities f_{av} are compared between simulations taking into account different hardware properties. Firstly, it is observed that weight

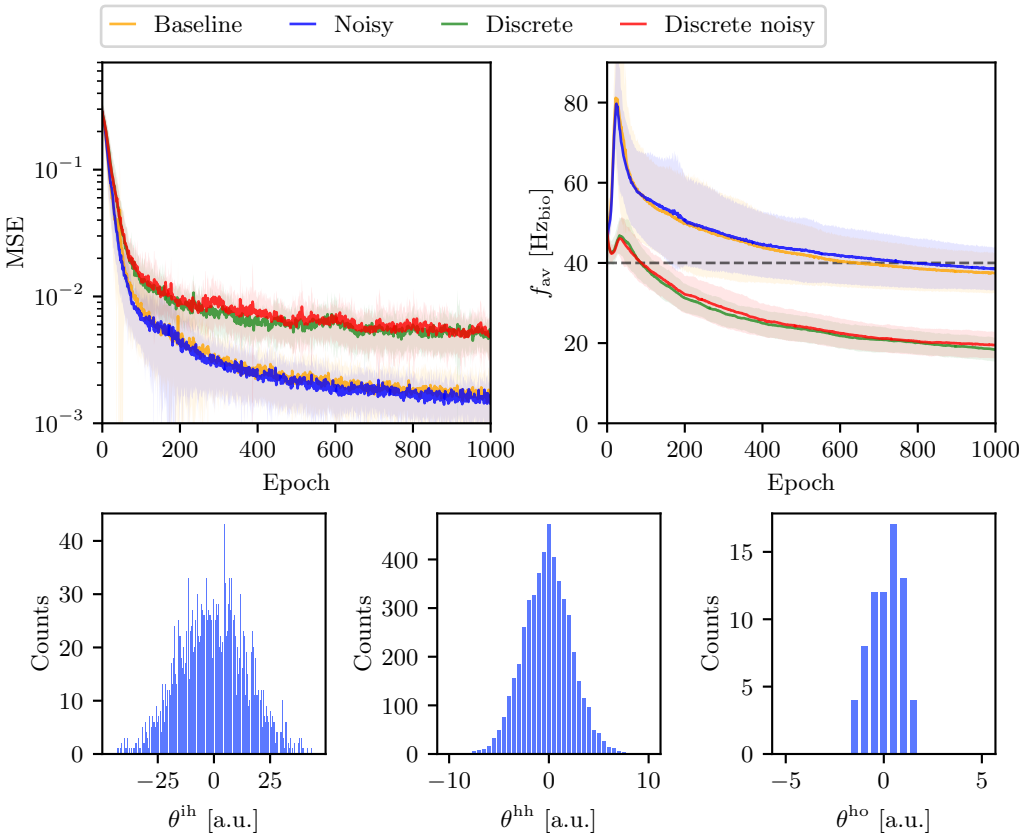


Figure 5.4: Comparison of the baseline (orange), a "Noisy" (blue) experiment simulated in a noisy environment, a simulation "discrete" (green) with discrete weights, and a simulation "Discrete noisy" (red) with both discrete weights and noise. On HX the synaptic weights have limited resolution. For a signed synapse this means $\theta \in \mathbb{N}_{-63}^{63}$. In simulation this results in a increasing MSE, as shown in the **upper-left** plot and a decreasing firing rate, as depicted in the **upper-right** plot. The histograms in the **lower** plots depict the distributions of learned input weights θ^{ih}, recurrent weights θ^{hh}, and output weights θ^{ho} of the baseline simulation without discretization after 1000 epochs of training. While the input weights cover a wide range of possible values, the output weights become too small to be resolved by integer weights. Parameters are given in Table A.4.

discretization yields a considerably increasing MSE. At the same time, the average firing rate decreases strongly. When considering the histogram of the output weights θ^{ho} without discretization, it can be seen that these become small compared to the hardware weight resolution. Hence, the discrete weights have an insufficient resolution for solving the task properly. This is mitigated in the next section. The small output weights are assumed to result from the high-activity regime in which the network operates; When many events are projected onto the readout neuron's membrane, small output weights suffice to adjust the neuron's membrane towards the target trace. Secondly, the impact of noise is negligible and the simulations in a noisy environment do perform equally well than without. Since learning is purely gradient-based, the fixed-pattern element in the

noise is not expected to disturb learning dramatically. However, the Gaussian noise on the membranes is expected to influence learning. Since this is not the case, a large signal-to-noise ratio is assumed with neurons being very responsive to synaptic input and thus reducing the influence of noise on the membranes.

5.3.5 Small Output Weights

An increasing loss due to small output weights can be prevented by increasing the weight resolution of θ^{ho} artificially. One possibility to achieve this is to increase the amplitude of the target pattern and adjust the learning rates appropriately. This causes the output weight distribution to become broader. On HX, however, this is not possible since this increases the range in which the readout neuron evolves and would cause the neuron to exhausts its physical limits. Instead, the synaptic input to the readout neuron is downscaled by a constant σ_{o}. Essentially, this corresponds to a decreasing learning rate η_{o}, such that, according to Figure 5.2, a higher firing rate and an increasing loss are expected. This is circumvented by correcting the learning rate, $\eta'_{\text{o}} = \eta_{\text{o}}/\sigma_{\text{o}}$. Downscaling the synaptic input of the readout neuron can be achieved on HX via calibration; hence, this strategy is in line with on-chip and in-the-loop training.

The effect of artificially increasing the output weight resolution in simulation (noisy and discrete) is shown in Figure 5.5, where losses and activities with $\sigma_{\text{o}} \in \{0.1, 0.5, 0.8, 1.0\}$ are depicted. The MSEs, clearly indicate an improvement in performance with decreasing σ_{o}. At the same time, the average firing rate increases. For $\sigma_{\text{o}} = 0.1$, an MSE[3] of about $2.68 \cdot 10^{-3} \pm 1.07 \cdot 10^{-3}$ is obtained, compared to the baseline loss of $1.68 \cdot 10^{-3} \pm 9.38 \cdot 10^{-5}$, this is slightly worse, however, significantly better than for $\sigma_{\text{o}} = 1$ with a loss of $5.41 \cdot 10^{-3} \pm 2.07 \cdot 10^{-3}$. The histograms exemplify the tendency to larger weights with decreasing σ_{o}. These observations show that increasing the output weights allows the network to learn well with discrete weights and noise. Hence, training with HX in-the-loop seems promising. This is confirmed in the following section.

5.4 HICANN-X in the loop

The simulations done in the previous sections can, in principle, be mapped directly to HX. Therefore, instead of simulating the network, it is emulated on HX in analog while the learning procedure remains widely untouched. The seamless transition from simulating the network in software to emulating it on HX is enabled by chip abstraction in software and the developed framework outlined in Section 4.2.1. However, since parameters on HX, such as synaptic input strength, are subject to calibration, simulation and hardware runs have no one-to-one correspondence but are aligned closely by an appropriate chip setup.

[3]MSE values are given as average over the 50 last epochs.

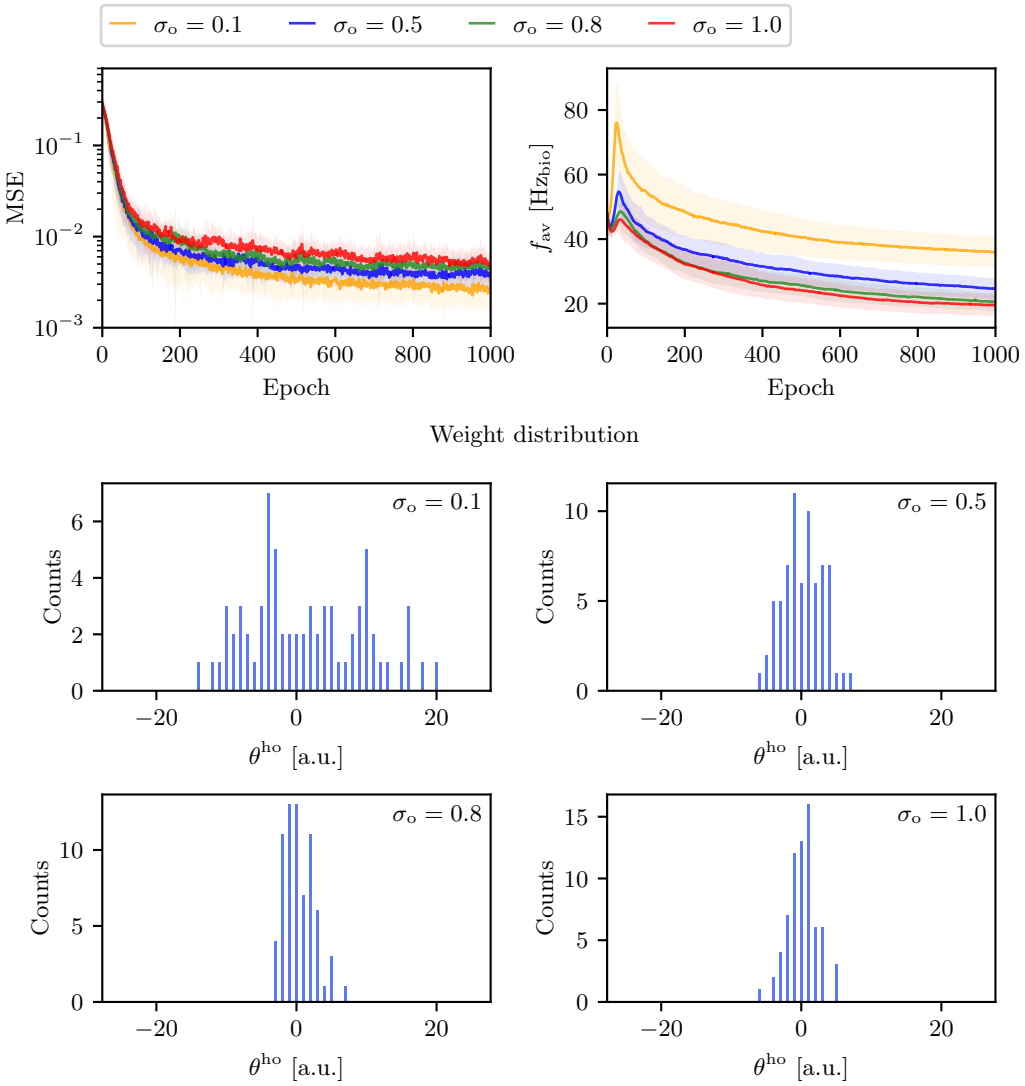


Figure 5.5: To gain a higher resolution of the output weights, the strength of the synaptic connections to the output neuron is scaled by $\sigma_o \in \{0.1, 0.5, 0.8, 1.0\}$, while the learning rate for the corresponding weights is increased. This ensures a balanced learning between the input/recurrent weights and the output weights. The **upper-left** plot shows the MSE, the **upper-right** plot the corresponding average activity for a given σ_o. Smaller σ_o yield decreasing losses and higher firing rates. The histograms in the **lower plots** show the learned output weight distributions of a single example for corresponding σ_o. Parameters are given in Table A.5.

5.4.1 Chip Setup and Training

The chip is calibrated towards desired parameters by the calibration library `calix` [Weis, 2020]. In order to mimic the simulation, the neuron's membrane time constants are chosen to $\tau_m = 20\,\text{ms}_{bio}$. The leak potential and the reset potential are both calibrated

to $v_\mathrm{l} = v_\mathrm{r} = 120\,\mathrm{DAC}$ values, while the threshold is set to $\vartheta = 160\,\mathrm{DAC}$ values. Effectively, this realizes a relative rest-to-threshold potential of 40 DAC values. The synaptic strength of the spiking neurons is adjusted empirically by trying different settings. Intuitively, a synapse is calibrated such that a single pre-synaptic event has the ability to trigger a post-synaptic spike. This is motivated by the simulations, where it can be seen that input weights get as large as the neurons' thresholds. Therefore, calibration targets a synaptic input strength to spiking neurons with $I_{\mathrm{syn,\,gm}} = 800\,\mathrm{DAC}$ values (see [Weis, 2020]). As in Section 5.3.5, the synaptic input strength to the readout neuron is downscaled with assumed $\sigma_\mathrm{o} = 0.1$ and calibrated to $I_{\mathrm{syn,\,gm}} = 200\,\mathrm{DAC}$ values. For the readout neuron, the threshold comparator is disabled to impose a non-spiking behavior. Synaptic time constants are set to $\tau_\mathrm{syn} = 4\,\mathrm{ms_{bio}}$ globally and the refractory period to $\tau_\mathrm{ref} = 1\,\mathrm{ms_{bio}}$. On HX, little longer synaptic time constants than in simulation seem necessary in order to increase the signal-to-noise ratio and make the neurons more responsive to input. Despite potentially exhausting the approximation of δ-like synapses (see Section 5.3.1), it poses no serious issue. Input weights are initialized from a normal distribution $\theta^\mathrm{ih}_\mathrm{init} \sim \mathcal{N}(25, 0)$ and the recurrent weights from $\theta^\mathrm{hh}_\mathrm{init} \sim \mathcal{N}(2, 0)$. As in simulation, the output weights are initially set to zero, which is found empirically to work well. The training procedure for training on HX is identical to the simulations (*cf.* Section 5.3.2), except fore the learning rates, which are chosen to $\eta_\mathrm{r} = 0.05$ and $\eta_\mathrm{r} = 0.02$.

Readout Trace

The membrane trace of the readout neuron is measured by the MADC and evolves in a range $y^t_k \in [0{,}1022]$ ADC values. Via calibration, the leak potential is adjusted such that the trace can move above and below the membrane's baseline by roughly the same amount. After a forward pass, the measured MADC trace is shifted by an offset $\bar{y}_0$ to get a sequence centered around zero with a vanishing baseline, i.e. $y^t_k - \bar{y}_0$. Here, the offset is given by the first MADC sample measured in the current trial, $\bar{y}_0 = y^0_k$. The assumption is that for this sample, the neuron's potential is still at rest, and there is no activity present yet. Since subtracting a global offset in each trail yields a jittering baseline because of a slightly changing resting potential with each run[4], this technique has proven itself to stabilize training. Finally, the membrane trace is scaled by a tunable constant ι, chosen as $\iota = 0.7$.

5.4.2 Application on Hardware

The hardware experiment is visualized in Figure 5.6. On HX, the loss converges towards $3.58 \cdot 10^{-3} \pm 1.28 \cdot 10^{-3}$; this is just slightly worse than the simulation with $2.68 \cdot 10^{-3} \pm 1.07 \cdot 10^{-3}$ and about twice as high as the baseline run with $1.67 \cdot 10^{-3} \pm 0.08 \cdot 10^{-3}$. The activity of the recurrent layer shows a lower activity on hardware and stabilizes at around $f_\mathrm{av} = 25\,\mathrm{Hz_{bio}}$. This is due to different weight initialization and different

[4]This is due to the analog nature of the readout neuron.

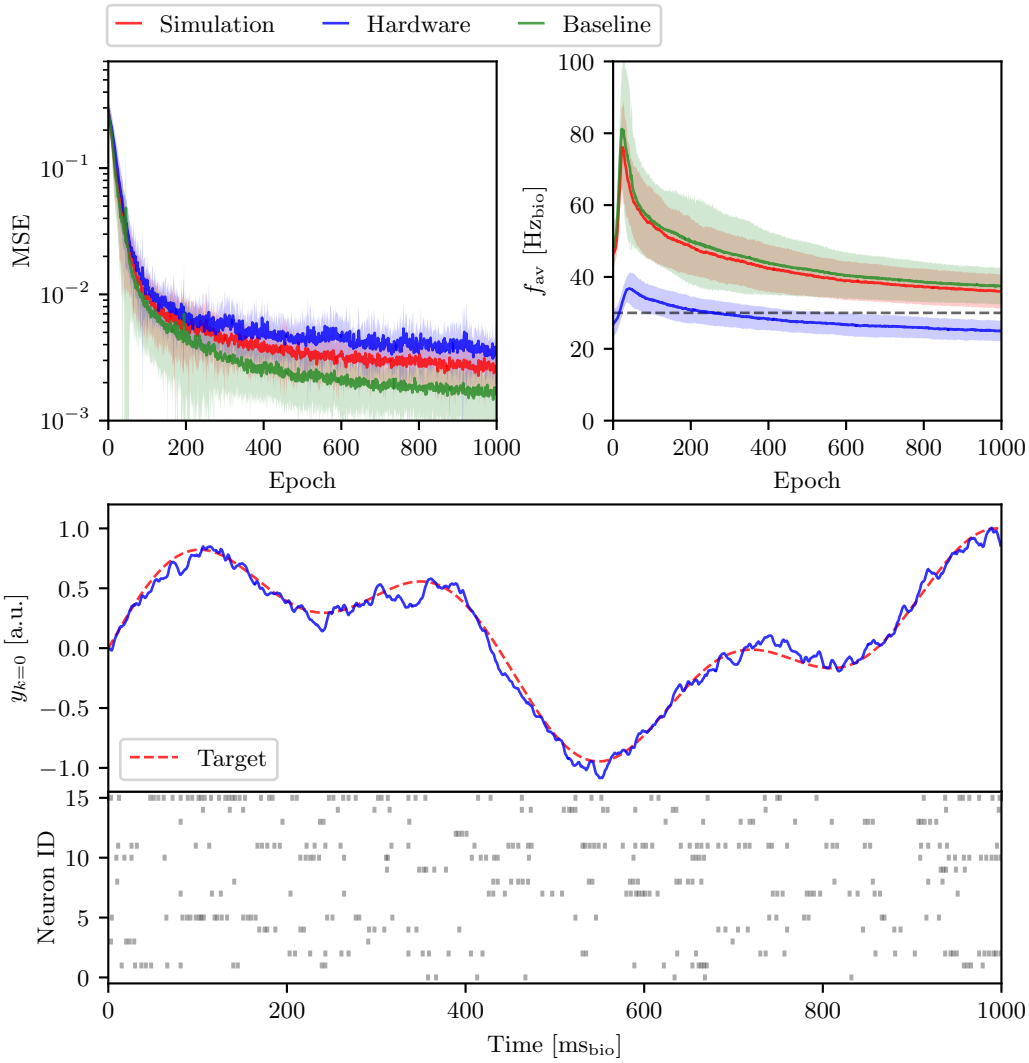


Figure 5.6: Pattern-generation on HX. The **top-left** plot shows the MSE over training epochs compared to the simulation (noise and discrete weights) and the s-prop baseline experiment (no hardware properties). On the **top-right** the evolution of the activities are compared. An example trace is plotted in the **middle** (after 1000 training epochs), where the analog membrane trace of the readout neuron on HX resembles the target pattern closely. The **lower-most** plot shows the spike events of 16 recurrent neurons. Similar to the simulation, spike clusters are formed, indicating that the network learned to control its activity in order to minimize the MSE. Parameters are given in Table A.6.

synaptic strengths resulting from the calibration process, effectively, yielding a slightly different experiment setup than in simulation. The network has clearly learned a spike pattern such that the readout neuron's membrane is adjusted to resemble the target trace well. As indicated by the recurrent spike trains, the recurrent neurons learn to cluster

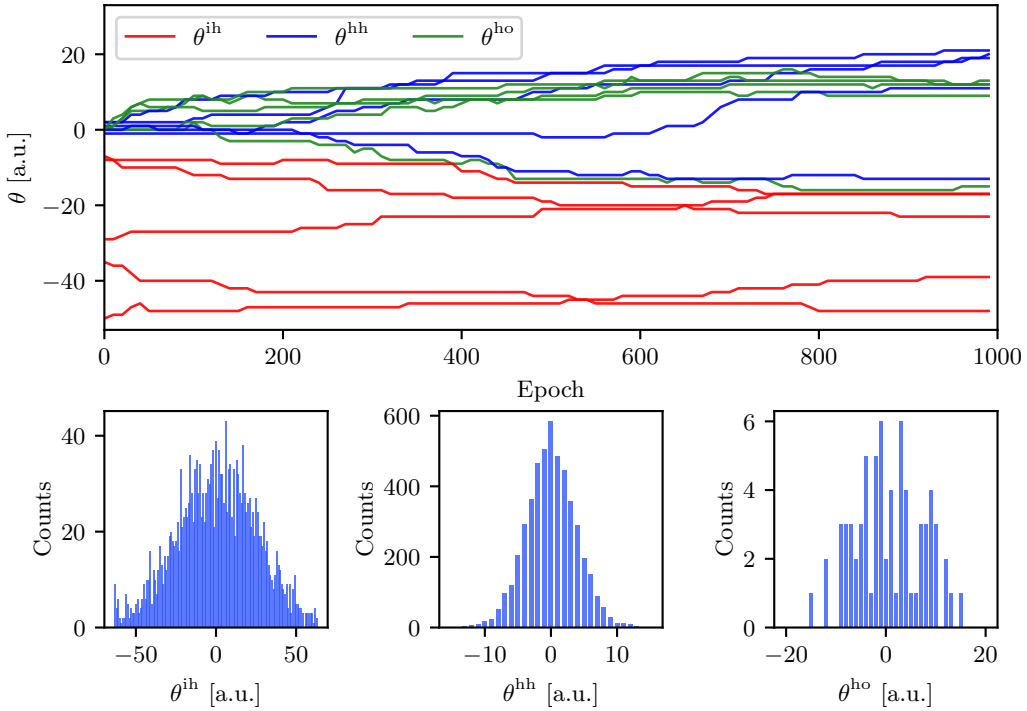


Figure 5.7: Weight evolution of a single hardware run. Although the MSE converges fast, the model still adjust its weights in later epochs, possibly, due to firing rate regularization. This can be seen in the **upper** plot, where the weights still change after the MSE remains rather stable (*cf.* Figure 5.6). In the **lower** plots the learned weight distributions for the inputs weights θ^{ih}, recurrent weights θ^{hh} and the output weights θ^{ho} are shown for as an example. Parameters are given in Table A.6.

their activity over short time periods in a sensible manner. The hardware experiment shows a similar behavior as the simulations and therefore confirms the assumed hardware properties incorporated in the software experiments. Further, it supports the observation in Section 5.3.4 that the impact of noise is small since the training process on hardware yields only a marginally worse performance than in simulations.

The development of the network's weights is exemplified in Figure 5.7 for a training run on HX. Interestingly, after Figure 5.6 indicates a converged loss, weights do still change in later epochs, however, after about 800 epochs they remain rather stable. Presumably, this is due to the regularization term adjusting the networks firing rates even after the MSE has stabilized. This remains subject to investigation.

5.4.3 The Role of Recurrence

Since the considered task is not very complex, it possibly can be solved to a fair degree without recurrent connections. It thus seems necessary to investigate whether s-prop

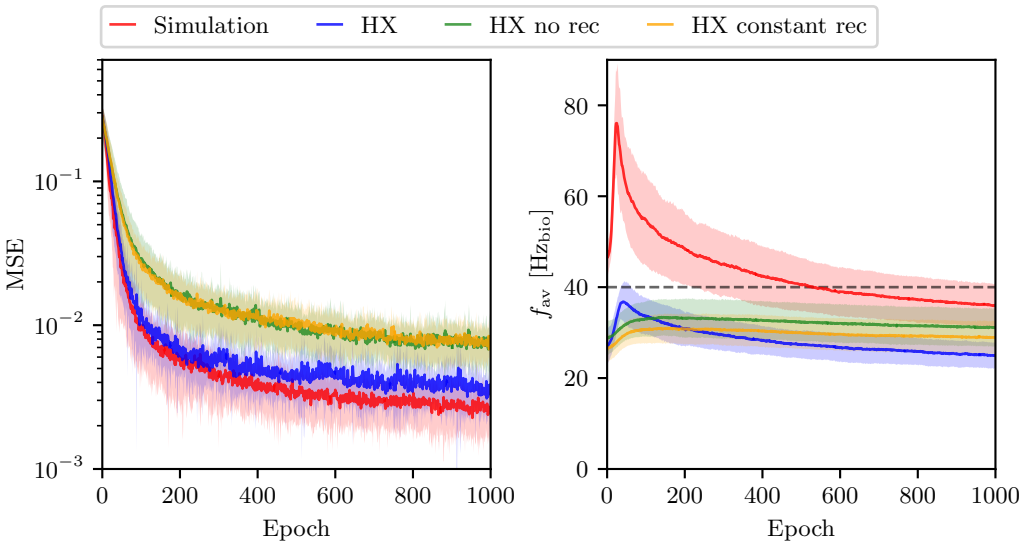


Figure 5.8: Comparison of networks trained with recurrence, without recurrence and a constant recurrent projection. The **left** plot shows the development of the MSE over epochs, the **right** plot the corresponding average firing frequency of the network's spiking neurons. It can be observed, that in simulation (red line, including hardware constraints) as well as on HX (blue line) the MSE is decreased when recurrent projections are learned, compared to a model without recurrence (green line). Using a fixed, i.e. constant over epochs, recurrent projection (same weight distribution as a the trained recurrent projection, but scrambled), performs equally to non-recurrent learning. While a fair amount is learned without recurrence, training recurrent weights with s-prop does indeed decrease the loss. Parameters are given in Table A.7.

does indeed exhibit the ability to adjust recurrent weights in a meaningful fashion. This is done in the following experiments.

The most intuitive way to measure the impact of recurrence is, obviously, to compare performances with and without recurrent connections. Therefore, the experiment on HX is repeated with the exact same learning and configuration setting, however, with recurrent weights fixed to zero, effectively disabling recurrent exchange of information (this is a simple spiking FNN).

Since a decreasing performance when discarding the recurrent projection can also be due to fewer parameters and missing information propagation over time, a further experiment is conducted. Instead of removing recurrence completely, the recurrent weights are solely excluded from the training process and remain constant over epochs. In order to ensure a sensible weight distribution, the constant recurrent weights are initialized with weights from a scrambled but learned recurrent projection (taken from the network of the corresponding pattern after training for 1000 epochs with learning in the recurrent projection enabled). This ensures information propagation over time.

As shown in Figure 5.8, the experiments with constant recurrence and without recurrence perform equally well (MSE: $7.71 \cdot 10^{-3} \pm 2.35 \cdot 10^{-3}$ and $7.49 \cdot 10^{-3} \pm 2.62 \cdot 10^{-3}$), while incorporating learning within the recurrent projection does, in fact, improve the performance (MSE: $3.61 \cdot 10^{-3} \pm 1.28 \cdot 10^{-3}$). Hence, s-prop enables learning in RSNNs; however, the task at hand combined with the network setup does not heavily rely on recurrence since a significant amount can be learned without. It is assumed that the role of recurrence is increased as the network's degree of freedom is further constrained, for instance, by increasing τ_{ref} (more in Chapter 7). Nonetheless, the task-selection and the chosen network parameters have proven themselves to be a good choice as a first step towards neuromorphic on-chip learning since the effects of learning recurrence are clearly visible. Note, the network considered is by no means fully tuned; this is subject to future works.

5.4.4 Investigating Stability

Emulating networks on analog neuromorphic hardware is not deterministic. Different runs yield slightly varying analog network parameters and noise on the membranes, and as a result, differing forward passes. Especially in recurrent networks, this can pose a problem since variations in the network emulation process are propagated along the sequence and can evoke completely different spike patterns. To investigate this, the trained networks are used for inference and are emulated over 100 epochs with learning disabled. The results are depicted in Figure 5.9.

The MSE loss shows the effect of analog network emulation immediately. It jitters around its mean with an average ensemble standard deviation of $1.14 \cdot 10^{-3} \pm 0.49 \cdot 10^{-3}$ over epochs, as indicated by the box-plot. While the network's average firing frequency f_{av} remains largely constant, on average, the firing rate of a particular recurrent neuron j has a standard deviation of $1.412\,\text{Hz}_{\text{bio}} \pm 0.787\,\text{Hz}_{\text{bio}}$ between inference runs. For a considered sequence length of $1000\,\text{ms}_{\text{bio}}$, this corresponds to a trial-to-trial variation of about 1.5 spikes per recurrent neuron on average. For the example pattern in the figure, the best and worst inference trials do not vary much. Even in the worst trial, the pattern is resembled quite well. This advocates the assumption in Section 5.4.3 that the network does not rely essentially on recurrence and suggests that the network propagates information over short time windows to fine-tune performance, however, not along the whole sequence. In that way, spike-time deviations at the beginning of the sequence do not have a big impact on the network's error at later times. In fact, this is expected since the information content encoded on the membranes disintegrates with the membrane time constants. Therefore, LIF neurons are not able to propagate information over longer time frames. Generally, the preferred choice for tasks that require higher temporal processing capabilities are LIF neurons with adaption (see Section 7.1) [Bellec et al., 2019; Brette et al., 2005].

With the results obtained in the previous sections, the next step is to finally adjust the learning rules to be implementable on the PPU and investigate their feasibility. This is done in the next chapter.

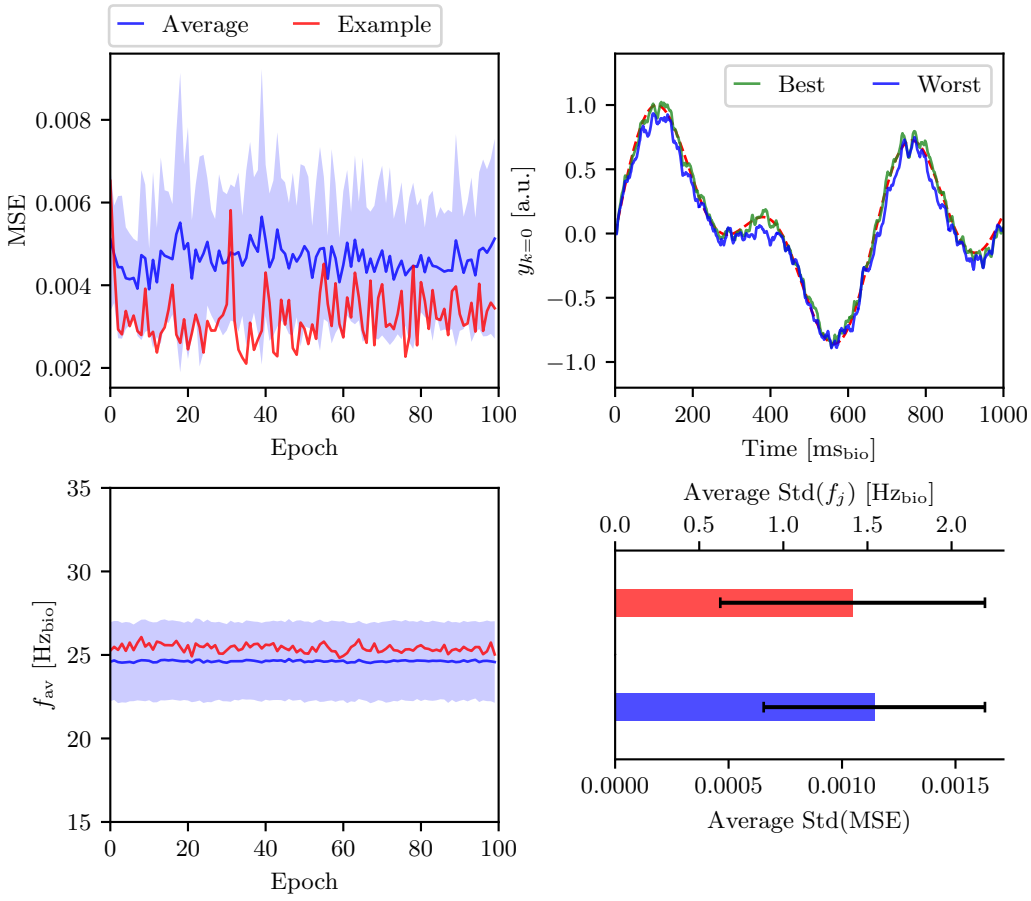


Figure 5.9: After training, inference of the learned model is performed on HX for 100 trials. Due to the analog nature of the chip, HX is subject to noise, yielding different results in each trial. This can be seen in the **upper-left** plot depicting the MSE over inference trials. "Average" corresponds to the ensemble average, "Example" to the pattern shown in the **upper-right** plot. Here, the best and worst inference trials are shown. Both do resemble the target pattern. The stability of the average ensemble firing rate is depicted in the **lower-left** over epochs and remains rather constant. The **lower-right** box-plot shows the average standard deviation of the MSE and the average standard divination of the firing frequency of a recurrent neuron j, both over epochs, i.e. error bars are uncertainties over the ensemble. Parameters are given in Table A.8.

6 On-chip Learning

After demonstrating e-prop-inspired learning with HX in-the-loop in the previous chapter, the desire arises to implement on-chip learning using the PPU. This comes with additional difficulties and constraints that need to be handled. In particular, this includes an approximated learning rule that operates with PPU-accessible observables. At the same time, full on-chip learning can exploit several advantages. Since plasticity is computed on-chip in parallel to the forward pass, the speed-up factor of HX can be fully exploited, and time-consuming data transfers between host and HX avoided. While this is expected to be very energy-efficient, a neuromorphic on-chip implementation is also appealing from a biological perspective; Like the brain deploys dedicated error computation networks generating learning signals in order to perform synaptic plasticity in certain brain areas [Buzzell et al., 2017], on HX, the calculation of weight updates and the emulation of the actual analog neural network are also both done in parallel on the same substrate.

This chapter proposes a mathematical derivation of an adjusted learning rule that turns out to be a promising candidate for a PPU implementation. Simulations verify that this rule enables learning in RSNNs in general and also when incorporating basic hardware constraints. Finally, the plasticity rule is implemented and tested on the actual hardware. This also includes a final discussion of encountered issues.

6.1 Learning Rule under Hardware Constraints

Full on-chip learning is only feasible under constraints defined by the hardware. A direct implementation of the learning rule described in Section 5.2 on HX is not possible for reasons becoming evident in the following. However, an adjusted plasticity rule can be found that indeed enables on-chip learning. While in-the-loop training computes the backward-pass on the host side, full on-chip learning uses the PPU to perform weight updates by incorporating correlation measurements. Therefore, the adjusted learning rule makes rather rough approximations; nonetheless, the on-chip implementation comes with the advantage of fast batch execution. This is assumed to partially outweigh the simplifications made when learning with a reasonably small learning rate.

The main issue that arises with the learning rule in Equation (2.53) is the limited access to the neurons' state variables v^t and z^t via the PPU. As already discussed in Section 5.2, measuring the membrane potentials v_j^t of the recurrent neurons is undesirable. In contrast to in-the-loop training with s-prop, as described in Chapter 5, where the backward pass is performed on the host-side by incorporating the recurrent spike trains z_j^t, the PPU is not

capable of accessing the latter (at least not trivially, possible workarounds can be found and are discussed in Chapter 7). Since these observables need to be known explicitly at each time step t to calculate the eligibility trace e_{ji}^t, it is infeasible to implement the original e-prop as well as the s-prop algorithm on-chip. Thus, the challenge is to find a good representation of e_{ji}^t by observables the PPU has access to.

6.1.1 Utilizing Correlation Measurements

Under the assumption made in Equation (5.3), the approximated *spike-based* eligibility traces $\hat{e}_{ji}^t$ become the exponentially filtered pre-synaptic spike trains z_i^t evaluated at a post-synaptic spike event $z_j^{t' \geq t}$. The causal correlation sensors on HX show a very similar behavior: A pre-synaptic spike triggers an exponential decay with a time constant τ_c, and the remaining amplitude is read out on the next post-synaptic event (see Section 3.2). It is thus a natural step to utilized these sensors in order to model the eligibility traces on-chip. This is convenient since the correlations sensors emulate an approximated version of the traces without any computational cost on the PPU and can be digitized by the CADC and read out with the VU. However, the behavior of the correlation sensors differs in two different aspects from the spike-based traces $\hat{e}_{ji}^t$. They measure the correlation in a *nearest-neighbor* approximation and allow only access to *accumulated* correlation. This will be clarified and discussed in detail now. Please remember for the following that t is a dimensionless time step and $t \cdot \delta t$ the actual time.

Nearest-Neighbour Approximation

The eligibility vectors $\hat{z}_i^t$ in Equation (5.3) quantify how much a synapse ji remembers of its activation history. Each pre-synaptic event $z_i^t = 1$ adds an exponential decay, with decay constant α, to the eligibility vectors $\hat{z}_i^t$, which correspondingly propagates past activation information into the future. In Equation (5.3), the eligibility vectors $\hat{z}_i^t$ are accessed on a post-synaptic event $z_j^t = 1$, giving the eligibility traces $\hat{e}_{ji}^t$ only a non-vanishing value if the post-synaptic neuron j spiked. It is thus sufficient to know the eligibility vectors at all post-synaptic events.

Ideally, one would like the correlations sensors to model the same additive behavior such that a correlation measurement on a post-synaptic event at time step $t' \geq t$ corresponds to $\hat{e}_{ji}^{t'}$ and contains propagated information of *all* past pre-synaptic activations. Since the correlation sensors are implemented as analog circuits, this poses two problems. Firstly, if a pre-synaptic neuron i fires strongly, it causes a high activation of the synapse. However, the correlation sensors model the eligibility vectors by charge, so they have an upper limit. Secondly, the charge in the correlation sensor is accumulated to the storage capacitor at a post-synaptic event. To preserve the activation history of all past pre-synaptic events, the current charge, representing the eligibility vector, needs to be measured and copied, resulting in a more complex sensor circuit requiring more space on the chip die [Breitwieser, 2015]. The correlation sensors on HX implement the eligibility traces, therefore, in a nearest-neighbor approximation. A pre-synaptic event puts the eligibility

vector to an initial amplitude η_c, and the next post-synaptic event applies the charge to the storage capacitor after which the eligibility vector is reset to zero. The charge applied to the storage at time step t' is then a measure for the eligibility trace $\hat{e}_{ji}^{\mathrm{nn},t'}$ under *nearest-neighbor approximation*. A direct consequence is that these eligibility traces contain only activation information from the most recent pre-synaptic event. This scheme is depicted and compared to the spike-based eligibility traces in Figure 6.1.

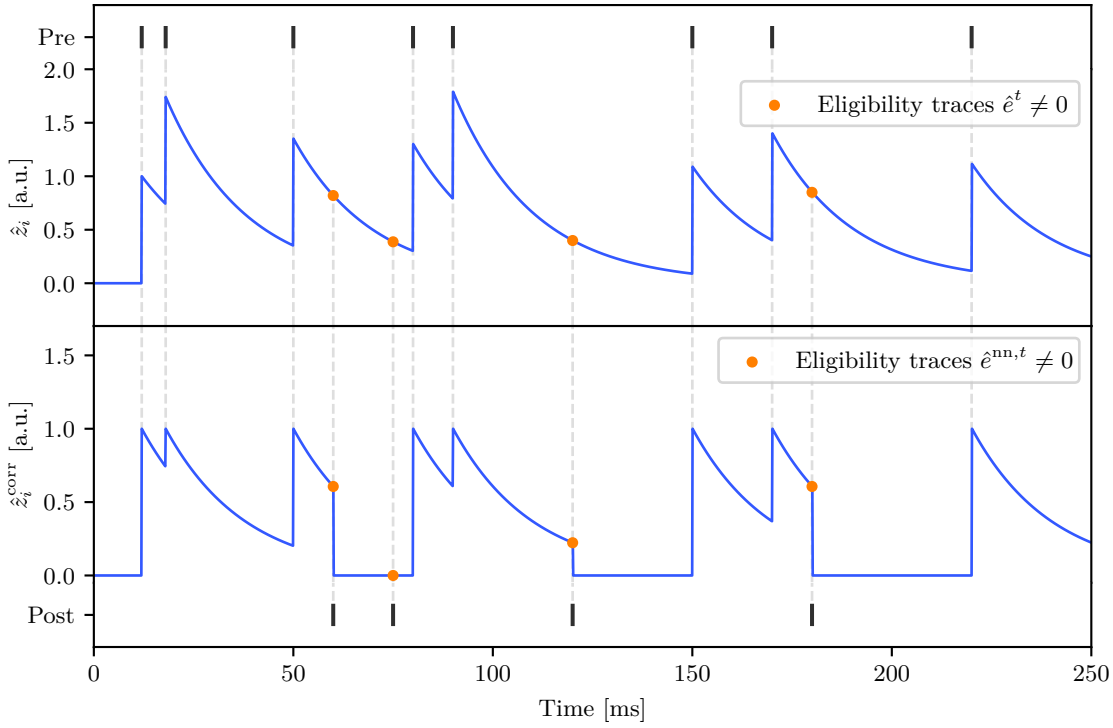


Figure 6.1: Comparison of spike-based eligibility vectors and the eligibility vectors emulated by the correlation sensors. A synapse ji receives pre-synaptic events (**uppermost** ticks) and the corresponding neurons emits post-synaptic spikes (**lowermost** ticks). For s-prop, the eligibility traces are the exponentially filtered pre-synaptic spikes (blue line in the **upper** plot, depicting the eligibility vector $\hat{z}_i$) evaluated at a post-synaptic event (orange dots). The correlation sensor emulate the vectors $\hat{z}_i^{\mathrm{corr}}$ in a nearest–neighbor fashion (**lower** plot) and are measured (corresponding to $\hat{e}^{\mathrm{nn},t^{\mathrm{post}}}$, orange dots) and reset by each post-synaptic event.

The exponentially decaying correlation amplitude η_{ji}^t emulated by the causal correlation sensors is (numerically) described by

$$\eta_{ji}^t = \begin{cases} \eta_c \exp\left[-\frac{1}{\tau_c}\left(t - t_i^{\mathrm{pre}}\right)\delta t\right] & \text{if } t_i^{\mathrm{pre}} \leq t \leq t_j^{\mathrm{post}} \\ 0 & \text{else,} \end{cases} \tag{6.1}$$

where η_c is the initial amplitude restored in the correlation sensor at the latest pre-synaptic event at time step t_i^{pre}. t_j^{post} is the subsequent post-synaptic spike-time, i.e. $z_j^{t^{\mathrm{post}}} = 1$. Assuming the correlation time constant τ_c to be equal to the membrane time

constant τ_m of the recurrent neurons gives for $t \in \left[t_i^\mathrm{pre}, t_j^\mathrm{post}\right]$

$$\tau_\mathrm{c} = \tau_\mathrm{m} \Longrightarrow \eta_{ji}^t = \eta_\mathrm{c} \exp\left[-\frac{1}{\tau_\mathrm{m}}\left(t - t_i^\mathrm{pre}\right)\delta t\right] = \eta_\mathrm{c} \cdot \alpha^{t - t_i^\mathrm{pre}}. \tag{6.2}$$

Here, the exponential term is identified with the membrane decay constant α defined in Equation (2.15). In comparison to the eligibility vectors in Equation (5.3), the correlation amplitude in Equation (6.2) does not propagate any information from activations $t' < t_i^\mathrm{pre}$. Thus, when modeling $\hat{z}_i^t$ by η_{ji}^t, this means that the sum over previous time steps t' in Equation (2.47) has only a contribution for $t' = t_i^\mathrm{pre}$ and the eligibility vectors $\epsilon_{ji}^{\mathrm{nn},t}$ under nearest neighbor approximation become

$$\epsilon_{ji}^{\mathrm{nn},t} \coloneqq \frac{1}{\eta_\mathrm{c}}\eta_{ji}^t = \begin{cases} \alpha^{t - t_i^\mathrm{pre}} & \text{if } t_i^\mathrm{pre} \leq t \leq t_j^\mathrm{post} \\ 0 & \text{else.} \end{cases} \tag{6.3}$$

According to Equation (5.3), the eligibility traces are then given by

$$e_{ji}^{\mathrm{nn},t} = z_j^t \cdot \epsilon_{ji}^{\mathrm{nn},t-1}. \tag{6.4}$$

These eligibility traces have only non-vanishing values for post-synaptic spike times t^post, i.e. $z_j^{t^\mathrm{post}} = 1$, and are provided by $\epsilon_{ji}^{\mathrm{nn},t^\mathrm{post}-1}$. This corresponds to the amplitude applied to the correlation storage at the post-synaptic event divided by η_c,

$$e_{ji}^{\mathrm{nn},t} = \frac{1}{\eta_\mathrm{c}}\eta_{ji}^{t-1} \cdot z_j^t. \tag{6.5}$$

Note, the minus one in the time index $t-1$ of the eligibility vectors is due to a recurrent transmission latency of δt. The eligibility vectors for the input projection are obtained by replacing the recurrent spike train z_i^{t-1} (i.e. the corresponding pre-synaptic spike time) with the inputs x_i^t since there is no transmission delay per definition. On HX, the recurrent latency is usually very small[1] and, therefore, $\eta_{ji}^{t-1} \approx \eta_{ji}^t$ can be assumed. Thus, on HX, recurrent and input synapses have the same eligibility traces.

Accumulated Correlation

To use the eligibility traces in Equation (6.5), they need to be known at each step t. As already mentioned, the correlation sensors, however, store the correlation information in a storage capacitor (see Section 3.2). After resetting the causal storages, the sensors start to accumulate correlation information of all pre-post spike pairs occurring in a given time interval P until the storage is read out and reset again. Since the PPU can only access the amount of accumulated correlation in the storage, precise time information is lost. The consequence is that $e_{ji}^{\mathrm{nn},t}$ cannot be known exactly, and the update rule has to deal with accumulated traces. Still, the accumulated correlation readout is a proxy for the activation of a synapse in the time period P and therefore contains valuable information. Hence, it is promising to use accumulated correlation measurements to model *accumulated* eligibility traces.

[1] Compared to the latency in simulations done in this chapter.

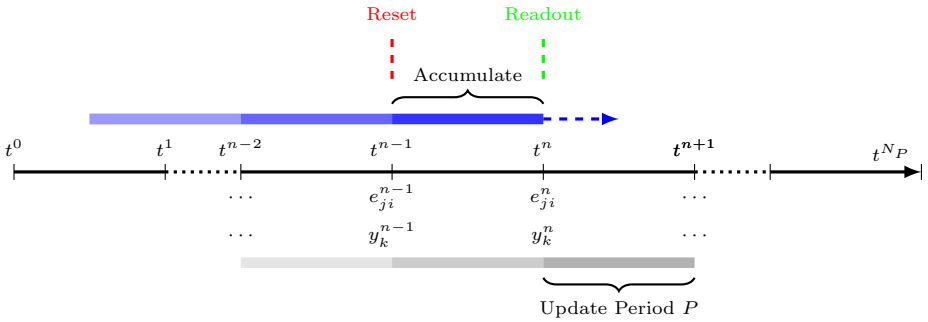


Figure 6.2: Assumed time line of correlation and membrane readouts. Before each update interval n of length P, the correlation is reset and then read out at t^n to compute $e^{\mathrm{acc},n}$. y_k^n corresponds to the membrane readout and is used to calculate the learning signal.

Discretising the whole time sequence of length T into N_P intervals of size P, such that $T = N_P \cdot P$ with P itself discretized by $P = p \cdot \delta t \in \mathbb{N}$, the correlation sensors are read out at time $t^n = n \cdot P$ (note, t^n has the dimension of a time and corresponds to the time step $t = n \cdot p$,), where $n \in [1, N_P]$. The correlation accumulated at the end of interval n is then given by

$$c_{ji}^n = \sum_{t=(n-1)p+1}^{np} \eta_{ji}^{t-1} \cdot z_j^t, \tag{6.6}$$

if the correlation storage is reset at the beginning of each interval t^{n-1}. Here again, the sum has only a contribution if $z_j^t = 1$, corresponding to a post-synaptic spike on which the amplitude η_{ji}^{t-1} is applied to the storage. Accumulating the eligibility traces in Equation (6.5) over the same interval gives

$$e_{ji}^{\mathrm{acc},n} = \frac{1}{\eta_\mathrm{c}} \sum_{t=(n-1)p+1}^{np} \eta_{ji}^{t-1} \cdot z_j^t = \frac{1}{\eta_\mathrm{c}} c_{ji}^n. \tag{6.7}$$

Incorporating the expression in Equation (6.7) into the update rule and assuming c_{ji}^n is described with sufficient accuracy by the correlation sensors allows now an on-chip implementation.

6.1.2 Adjusting the Learning Rule

An on-chip update rule for the recurrent weights can be derived starting from the update in Equation (2.45). In addition to the accumulative eligibility traces $e_{ji}^{\mathrm{acc},n}$, appropriate learning signals $\hat{L}_{ji}^t$ need to be found.

According to Equation (2.44), the learning signals are task-specific and depend on the error function E. Choices of E that result in learning signals depending on observables inaccessible by the PPU are obviously not realizable on-chip (*cf.* Section 5.1.2). The

residual sum of squares (RSS) used in the pattern-generation task depends on the membrane potentials y_k^t of the readout neurons, resulting in suitable learning signals if the membranes are measured with a feasible sample rate $1/P$ via the CADC. Therefore, this task is qualified for an on-chip implementation. Assuming P to be the time interval in which the correlation sensors accumulate correlation information as in Equation (6.7), a correlation readout coincides with a membrane readout. This is visualized in Figure 6.2. For the pattern-generation task the learning signals $\hat{L}_j^t$ are given in Equation (2.51). Since these learning signals assume y^t to be sampled with $1/\delta t$, they need to be adjusted. Therefore, the time step t is replaced with the update interval index n and δt with the update period P. Then, the learning signals access the readout membranes only at $t^n = nP = n \cdot p \cdot \delta t$ and are assumed as

$$\hat{L}_j^n := \sum_k \theta_{kj}^{\mathrm{ho}} \sum_{n' \geq n} \left(y_k^{n'} - y_k^{*,n'} \right) \hat{\kappa}^{n'-n}, \tag{6.8}$$

where $y_k^{n'}$ is the membrane potential of the readout neuron k at time $t^{n'} = n'P$, $y_k^{*,n'}$ the corresponding target value, and $\hat{\kappa} = \kappa^p = \kappa^{P/\delta t} = \exp\left(-P/\tau_{\mathrm{m}}\right)$ the decay constant. Discretising the sum in Equation (2.45) over t into N_P intervals of length $p = P/\delta t$,

$$\frac{\widehat{\mathrm{d}E}}{\mathrm{d}\theta_{ji}^{\mathrm{hh}}} = \sum_t \hat{L}_j^t \cdot e_{ji}^t = \sum_{n=1}^{N_P} \sum_{t=(n-1)p+1}^{np} \hat{L}_j^t \cdot e_{ji}^t, \tag{6.9}$$

and assuming $\hat{L}_j^n$ to be constant in the interval $((n-1)p, np]$, suggests under nearest-neighbor approximation

$$\sum_{n=1}^{N_P} \hat{L}_j^n \sum_{t=(n-1)p+1}^{np} e_{ji}^{\mathrm{nn},t} \approx \sum_{n=1}^{N_P} \hat{L}_j^n \cdot e_{ji}^{\mathrm{acc},n} =: \left(\frac{\mathrm{d}E}{\mathrm{d}\theta_{ji}^{\mathrm{hh}}} \right)^{\mathrm{acc}}. \tag{6.10}$$

Inserting the definitions of $e_{ji}^{\mathrm{acc},n}$ in Equation (6.7) and $\hat{L}_j^n$ in Equation (6.8) yields the on-chip learning rule for the recurrent weights,

$$\left(\Delta\theta_{ji}^{\mathrm{hh}} \right)^{\text{on-chip}} = -\eta' \left(\frac{\mathrm{d}E}{\mathrm{d}\theta_{ji}^{\mathrm{hh}}} \right)^{\mathrm{acc}} \tag{6.11}$$

$$= -\eta' \frac{1}{\eta_{\mathrm{c}}} \sum_{n=1}^{N_P} \sum_k \theta_{kj}^{\mathrm{ho}} \sum_{n' \geq n} \left(y_k^{n'} - y_k^{*,n'} \right) \hat{\kappa}^{n'-n} c_{ji}^n \tag{6.12}$$

$$= -\eta \sum_{n=1}^{N_P} \sum_k \theta_{kj}^{\mathrm{ho}} \left(y_k^{n'} - y_k^{*,n'} \right) \sum_{n' \leq n} \hat{\kappa}^{n-n'} c_{ji}^{n'}. \tag{6.13}$$

Here η denotes the *learning rate* which absorbs the constant $1/\eta_{\mathrm{c}}$. In the last step, the sum indices are swapped and renamed such that the sum over future intervals becomes a sum over the past. Since $\hat{\kappa}$ is a decay constant of the readout neurons, the sum propagates the correlation readouts $c_{ji}^{n'}$ from the past into the present, taking into account contributions to the current error from neuron activities in past intervals. If the period P is much larger

than the time window of the readout neurons (i.e. their membrane time constant), the contribution from previous intervals vanishes.

The update rule uses only accumulated information collected in each time interval $((n-1)P, nP]$. For the weight update in Equation (6.13), this means that correlation information of all pre-post spike pairs accumulated in an interval is considered equally responsible for the error $\left(y_k^{n'} - y_k^{*,n'}\right)$ in the contribution to the weight update at t^n. This is a distortion of reality since — for instance — a high synapse activation at the beginning of an interval should be more responsible for errors at times $t' < t^n$ than for errors at t^n. Note, however, as the true eligibility traces propagate activation information into the future, the high activity at the beginning of an interval is supposed to impact the weight update at time t^n, but just by the amount propagated by α.

The update rule for the input weights θ_{ji}^{ih} can be inferred analogously for the recurrent weights by replacing the correlation measurements c_{ji}^n of the recurrent synapses with correlation measurements of the input synapses. This is simply achieved by expressing the eligibility traces $e_{ji}^{\text{acc},n}$ in Equation (6.7) with the eligibility traces $e_{ji}^{\text{nn},t}$ of the input synapses (see Section 6.1.1).

Output Weights

So far, the on-chip learning rules for the input and recurrent weights are derived. When considering the update rule for the output weights in Equation (2.54), the weight update again depends on the spike train z_j^t of the recurrent neurons filtered by the readout decay κ. As for the recurrent update rule, each pre-synaptic event — here the spikes of the recurrent neurons — triggers an exponential decay. However, due to non-spiking readout neurons, this behavior cannot be modeled by the correlation sensors since they only accumulate correlation on post-synaptic events. In order to avoid the dependence on explicit spike times, the neurons spike counters are used, accumulating spikes within the interval P (see Section 3.1). These counters can be read out and reset by the PPU and are, therefore, a convenient choice to adjust the output update rule.

Assuming the counter ζ_j of neuron j to be reset at time t^{n-1}, the counter readout after time interval $P = p \cdot \delta t$ at time t^n is given by

$$\zeta_j^n = \sum_{t=(n-1)p+1}^{np} z_j^t \tag{6.14}$$

and corresponds to the number of accumulated spike events in P. If the counter is read out simultaneously with the membrane potential y_k^n, the update rule in Equation (2.54) can be written as

$$\left(\Delta\theta_{kj}^{\text{ho}}\right)^{\text{on-chip}} = -\eta \sum_{n=1}^{N_P} \left(y_k^n - y_k^{*,n}\right) \sum_{n' \leq n} \hat{\kappa}^{n-n'} \zeta_j^{n'}, \tag{6.15}$$

where the spike train z_j^t is replaced by the counter ζ_j^n and δt in κ is replaced with P. In consequence, the signed error $\left(y_k^t - y_k^{*,t}\right)$ at each time step t is not weighted by

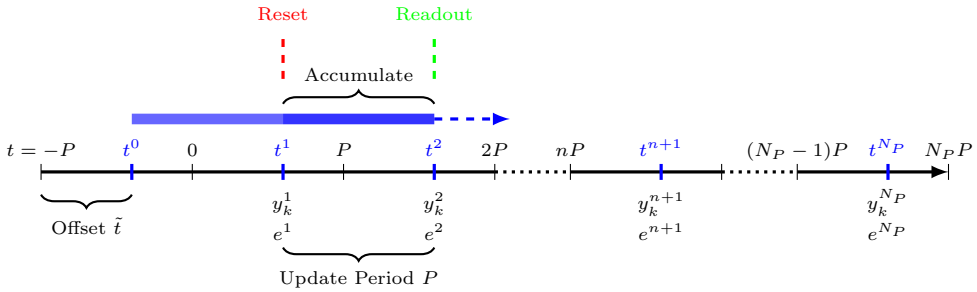


Figure 6.3: In order to make the on-chip learning rule take all elements of the time sequence into account, in each trial a new random offset $\tilde{t}$ is sampled. Therefore, the sequence is augmented by one interval such that accumulation begins at $t^0 = -P + \tilde{t}$. In this way, each trial considers different time intervals.

the propagated spike events z_j^t anymore, but the error at time t^n is weighted by the accumulated spikes propagated by $\hat{\kappa}$. As for correlation, all events at times $t' \in (t^{n-1}, t^n]$ are thus held equally responsible for the signed error at t^n.

Since the learning rules in Equation (6.13) and (6.15) propagate spike information along the sequence by $\hat{\kappa}$ (and implicitly by the correlation sensors), their neuromorphic implementation is referred to as *Neuromorphic Accumulative Spike Propagation* (NASProp) from here on.

Regularization

The adjustments for the eligibility traces can also be applied to the regularization update in Equation (2.56). Here, the term $h_j^t \hat{z}_i^{t-1}$ is replaced with the correlation measurements in Equation (6.7). The sum over t is split into chunks of size $P = p \cdot \delta t$ and summed up over n,

$$\Delta\theta_{ji}^{\text{ih,hh}} = \eta^{\text{reg}} \left(f^{\text{target}} - f_j^{\text{av}} \right) \sum_n e_{ji}^{\text{acc},n}. \tag{6.16}$$

The average firing rates f_j^{av} can be computed on-chip by reading out the recurrent neurons' spike counters.

Random Offsets

The on-chip update rules consider the readout traces $y_k^{*,t}$ only at times $t^n = nP$, respectively time steps $t = n \cdot p$. Hence, errors at time steps $t' \in ((n-1)p, np)$ have never a contribution to the weight update, and $y_k^{t'}$ is allowed to evolve freely within this interval. This, of course, poses a problem since the traces y_k^t are supposed to minimize the error to $y_k^{*,t}$ over the whole time sequence.

This issue is counteracted by introducing a random offset. Therefore, the time sequence is augmented in the beginning by one interval of length P, such that the sequence starts

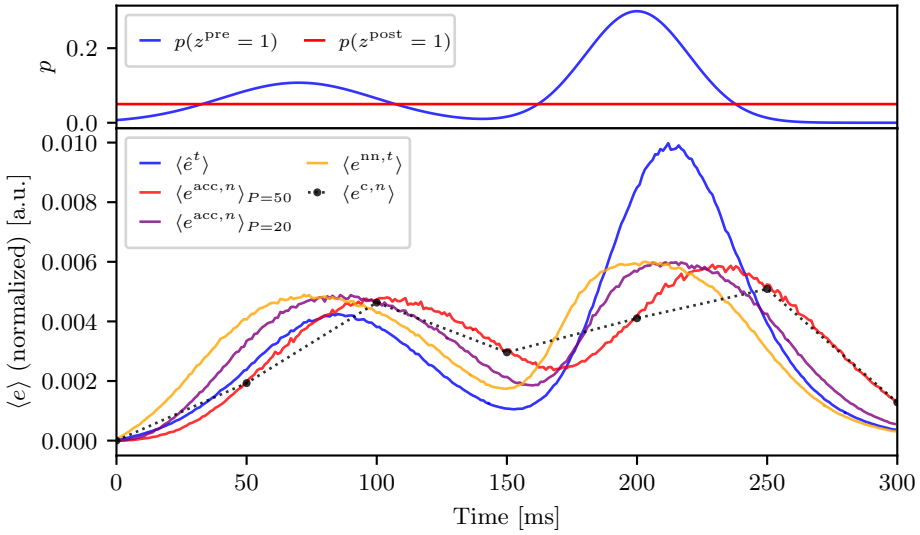


Figure 6.4: Qualitative comparison of different eligibility trace approximations for a single synapse. To create a artificial spiking environment, pre-synaptic events are sampled from a distribution $p(z^{\text{pre}} = 1)$ and post-synaptic spikes are drawn uniformly. The expected eligibility traces $\langle e \rangle$ are the traces e^t averaged over many trials, each with newly sampled spike events. A more in-depth interpretation is given in the text.

at time $-P$. Instead of having constant time points t^n over which the weight update is calculated, in each training trial a new random offset $\tilde{t} \in [0, P)$ is drawn to shift t^n,

$$t^n = -P + \tilde{t} + nP \quad \text{with } n \in \mathbb{N}_0^{\leq N_p}. \tag{6.17}$$

Since the sum over n in the update rules starts at $n = 1$, the first contribution to the update is at $t^{n=1} > 0$; however, accumulating starts at $t^{n=0} < 0$. For time steps $t < 0$, the input activity $x_i^{t<0}$ and the target trace $y_k^{*,t<0}$ are assumed to be zero. This leads to non-zero contributions in the first interval for $t \geq 0$ only. See Figure 6.3 for clarification.

When using a different offset in each training trial, each weight update will take distinct time points t^n of the target pattern y_k^n into account, such that over many epochs, the whole pattern contributes to weight updates, and the network can learn to minimize the error over all time steps t. Further, this will weight the corresponding learning signals with different synapse activation information given by the accumulated eligibility traces with every trial. Thus, as depicted in Figure 6.4, a random offset allows the learning algorithm to gain a more realistic image of the networks spiking activity as the training process evolves.

In Figure 6.4, an artificial spiking setup is created. Be aware that this is meant as a toy example supposed to show the qualitative behavior of the different approximations made. Here, pre-synaptic spike events are sampled from a distribution $p(z^{t,\text{pre}} = 1)$, modeling the activation of a synapse. In order to calculate eligibility traces, post-synaptic events

are drawn uniformly. The resulting eligibility traces for different approximations are calculated and averaged over many experiments, given by the expected eligibility trace $\langle e \rangle$ over t. Here, $\langle \hat{e}^t \rangle$ corresponds to the traces under spike-based approximation made in Equation (5.3). In comparison to $\langle e^{\mathrm{nn},t} \rangle$ (nearest-neighbor approximation), the traces $\hat{e}^t$ do resemble the amount of activity at a specific time much better — a high synapse activation results in bigger eligibility traces. This is because the traces $\hat{e}^t$ are additive where each pre-synaptic event triggers an additional exponential decay, while the traces $e^{\mathrm{nn},t}$ do discard the activation history on each pre-synaptic event (and on each post-synaptic event). This is also reflected by the position of the corresponding maxima. The maxima of $\langle \hat{e}^t \rangle$ do not coincide with the activation maxima given by $p(z^{t,\mathrm{pre}} = 1)$ since the eligibility vectors propagate the activation information into to future, such that the pre-synaptic events push the eligibility traces within the propagation time window (here given by the membrane time constant $\tau = 20\,\mathrm{ms}$). In essence, this means that a spiking activity a time step $t' < t$ in the past is held accountable for the error the trace y_k^t produces at t. In contrast, the maxima of $\langle e^{\mathrm{nn},t} \rangle$ are rather coincidental with the maxima of $p(z^{t,\mathrm{pre}} = 1)$ since the synapse remembers only the latest pre-synaptic event at a given time step t. The averaged accumulated traces in Equation (6.7) are visualized by $\langle e^{\mathrm{c},n} \rangle$ with $P = 50\,\mathrm{ms}$. Here, no random offset is used, such that the trace is only given at time $t^n = nP$, and the weight update is limited to this information in all training trials. This trace has low resolution and, therefore, cannot represent the underlying spiking activity properly. The random offset $\tilde{t}$ does somewhat remedy this, as can be seen by the average trace $\langle e^{\mathrm{acc},n} \rangle_{P=50}$. Over many training trials, these expected eligibility traces captured the shape of $\langle \hat{e}^t \rangle$ much better because, in each trial, the spiking activity is considered in different time intervals. However, the maxima are shifted with respect to the spike-based traces $\hat{e}^t$ as well as to $e^{\mathrm{nn},t}$. This is due to the accumulating nature of these traces, where the eligibility traces under nearest-neighbor approximation, accumulated over a period P, is assigned to the time t^n at the end of the corresponding interval. Hence, as P increases, more past spiking activities are taken into account and are penalized for the error at t^n. The traces $\langle e^{\mathrm{acc},n} \rangle_P$ converge towards $\langle e^{\mathrm{nn},t} \rangle$ for $P \to \delta t$.

6.2 Simulation

Before proposing a feasible on-chip implementation, the learning rule derived is subject to investigation in simulation. This will encompass a baseline simulation, showing that the learning rule enables learning in RSNNs in principle, followed by simulations that take hardware properties into account. Before explaining the training procedure, the hardware constraints are discussed.

6.2.1 Hardware Constraints

Basic hardware properties like weight discretization and noise, as outlined in Section 5.3.1, do apply here as well. Additionally, three aspects need to be considered:

Firstly, on-chip weight optimization, secondly, firing rate regularization on the PPU, and lastly, noise in the correlation sensors.

So far, no sophisticated weight optimizer is implemented on the PPU. While this is possibly part of future works, the on-chip weight updates cannot be optimized efficiently with momentum at the time of writing. Consequently, simulations should rely solely on momentum-free SGD-like weight optimization, meaning applying weight updates directly as given by the learning rules without further processing.

On-chip firing-rate regularization can only be achieved by accessing the spike counts of *all* recurrent neurons with the PPU. While this is preferably done rather fast via VU access, for the current chip version, this is solely achievable by reading out and resetting all counters successively with the scalar unit. As shown in Table 6.2, this is very time-consuming.[2] Nonetheless, regularization can be implemented with the scalar unit since this requires reading out the spike counters only at the end of the sequence (and not at each t^n) and, hence, is not time-critical. However, regularization is not implemented on-chip yet. Simulations will take this into account.

On HX, correlation is measured in analog and subject to fixed-pattern noise (see Figure 3.5). Hence, the correlation amplitudes η_c and the correlation time constants τ_c differ from synapse to synapse. In simulations, these constants are sampled from a normal distribution $\mathcal{N}(\eta_\mathrm{c}^\mathrm{target}, \sigma_{\eta_\mathrm{c}})$ and $\mathcal{N}(\tau_\mathrm{c}^\mathrm{target}, \sigma_{\tau_\mathrm{c}})$, respectively, with $\eta_\mathrm{c}^\mathrm{target}$ and $\tau_\mathrm{c}^\mathrm{target}$ being the desired means and the corresponding σ the expected standard deviations. The standard deviations are assumed to be 10% of the means. The sampled values do not change over training epochs.

6.2.2 Network Setup and Training Procedure

To align the experiments with s-prop in Chapter 5 to the simulation done in the following, the network setup and chosen LIF constants in Section 5.3.2 remain untouched. This also applies to the general training procedure and task definition.

Additionally, in each training trial, a new random offset $\tilde{t}$ is drawn uniformly from the set $\{\tilde{t} \in \mathbb{N}_0 | 0 \leq \tilde{t} < P\}$. The update periods are set to $P \in \{25\,\mathrm{ms_{bio}}, 50\,\mathrm{ms_{bio}}\}$ if not stated otherwise. The baseline experiments use non-discrete weights, optimized by the Adam optimizer with learning rates $\eta_\mathrm{r}, \eta_\mathrm{o} = 0.03$. Hardware-close experiments have discrete weights $\theta \in \mathbb{N}_{-63}^{63}$. Therefore, the weight updates are rounded stochastically (see Section A.2.1) and applied directly to the weights with adjusted learning rates of $\eta_\mathrm{r} = 9 \cdot 10^{-5}$ and $\eta_\mathrm{o} = 1 \cdot 10^{-4}$ without further processing. These learning rates were found to work well after a hyperparameter-search. Note, simulations with discrete weights utilize additional artificial weight resolution enlargement as described in Section 5.3.5 with $\sigma_\mathrm{o} = 0.1$. For experiments with regularization, the average firing rates f_j^av are regularized

[2] For the current on-chip implementation in Section 6.3 this does not pose an issue for the output weight updates in Eq. (6.15) (which also need counter readouts) since in each update interval n only a *single* counter is accessed (*cf.* Section 6.3) which can be achieved in a reasonable time.

towards $f^{\text{target}} = 40\,\text{Hz}_{\text{bio}}$ with a regularization strength of $\eta^{\text{reg}} = 5000$ according to Equation (6.16). The remaining parameters and procedures are given in Section 5.3.2 if not replaced here.

6.2.3 Baseline Experiment

In order to have a baseline to simulations with hardware characteristics and to show that the on-chip learning rules enable learning in RSNNs, an experiment without any hardware constrains is conducted with $P = 25\,\text{ms}_{\text{bio}}$ and $P = 50\,\text{ms}_{\text{bio}}$. The training process is visualized in Figure 6.5.

Both runs, with $P = 25\,\text{ms}_{\text{bio}}$ and $P = 50\,\text{ms}_{\text{bio}}$, converge properly. The shorter update period results in a loss of about $1.54 \cdot 10^{-3} \pm 0.48 \cdot 10^{-3}$ and the longer update period in a loss of $3.15 \cdot 10^{-3} \pm 0.96 \cdot 10^{-3}$, which is just slightly worse. Compared to e-prop, the loss with $P = 25\,\text{ms}_{\text{bio}}$ is just marginally above. However, this comparison is arguably unfair since training parameters are not tuned for e-prop but NASProp. This also becomes visible for the average firing rates. For NASProp, the firing rates are regularized close to the target, while the networks recurrent neurons fire much stronger when trained with e-prop. It is observed that e-prop needs to be regularized stronger.[3] The example trace in the figure gives an impression of how well the task can be solved with $P = 50\,\text{ms}_{\text{bio}}$. Here, the actual readout trace $y^t_{k=0}$ imitates the target sequence narrowly. As for s-prop, the network learns to form spike clusters, indicating that the network learns to specialize its activity to dedicated time windows to minimize the network's error.

6.2.4 Update Period

The baseline experiments in Section 6.2.3 indicate a dependency of the network's performance on the update period P. This is investigated in the following since later on-chip implementations on HX will define a minimal update period P (see Section 6.3.1). Additionally, the discussed properties of the HX chip are incorporated into the simulation. All simulations are done for $P \in \{p \in \mathbb{N}_{10}^{\leq 120} | p \bmod 10 = 0\}$. The results are depicted in Figure 6.6. It is observed that SGD-like weight optimization results in an increasing loss compared to Adam weight optimization. Using discrete weights with SGD does increase the MSE further. Noticeable, however, is that (similar to s-prop) introducing noise does not affect the performance severely. One explanation is that gradient-based learning can mitigate the effect of fixed-pattern noise by adjusting weights appropriately. The Gaussian noise on the membranes is assumed to be negligible due to a large signal-to-noise ratio. Further, turning off regularization does result in equal performance as with. Admittedly, the effect of turning off regularization is considered in the selection of the learning rates to ensure good performance without regularization. Nonetheless, no regularization, of course, results in different average firing rates. The example runs in the figure (upper-right) show that the traces get rough with a large $P = 100\,\text{ms}_{\text{bio}}$

[3] e-prop training with stronger regularization is shown in Figure 5.3.

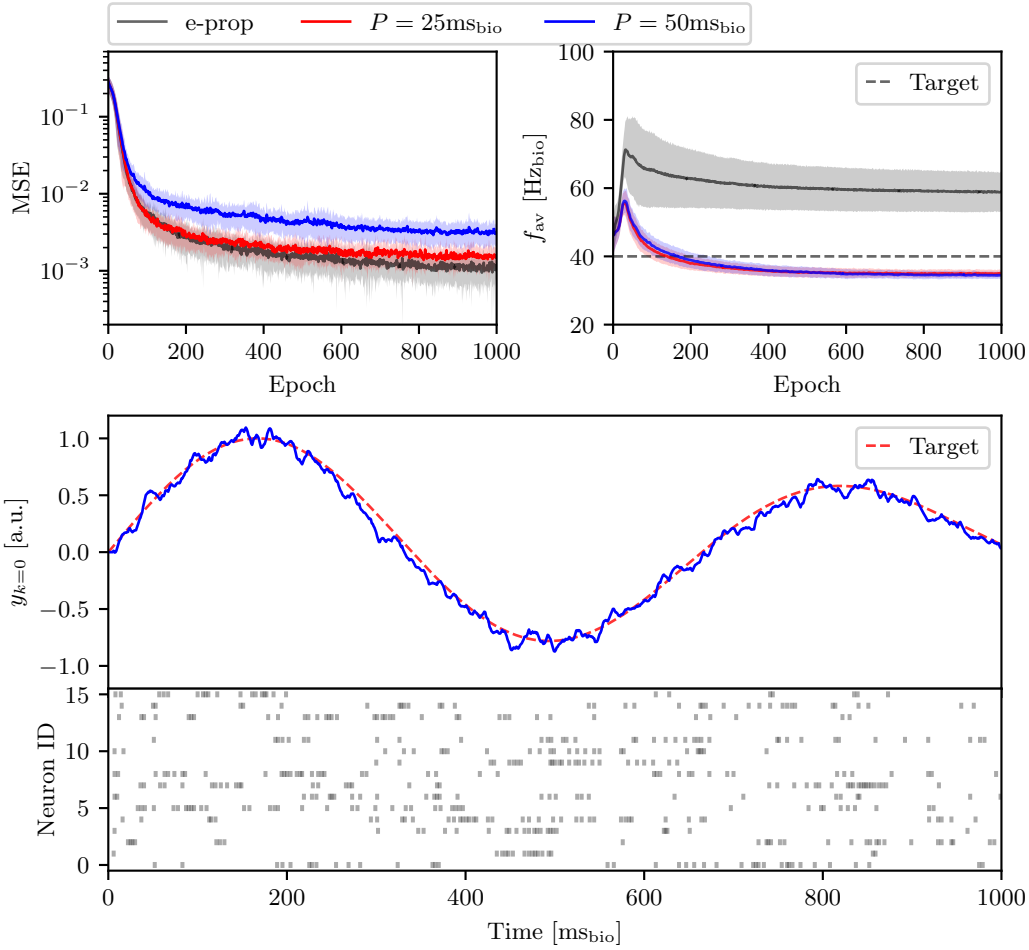


Figure 6.5: NASProp baseline experiments in comparison to e-prop. The **upper-left** plot shows the MSE over epochs, with the red and blue line corresponding to training with NASProp and the gray line to e-prop. A bigger update period P yields increasing losses. The **upper-right** plot shows the corresponding average activity of the recurrent layer. The **lower** plot exemplifies the performance of NASProp with $P = 50\,\text{ms}_{\text{bio}}$ after 1000 training epochs, where the readout traces resembles the target closely. The lower part shows the activity of the 16 recurrent neurons over the time sequence. It can be observed that the network clusters its activity. Parameters are given in Table A.9.

but they still follow the general curvature of the target trace. All simulations exhibit a relatively linear dependency on P. While the baseline MSEs do only increase slightly with P, for simulations with discrete weights, the MSE increases faster. However, these simulations show that even if an increasing P decreases performance, acceptable results can still be achieved with a longer P, as indicated by the example traces.

Although the simulations do not capture all properties of HX, these simulations motivate the step towards implementing the learning rules on HX using the PPU.

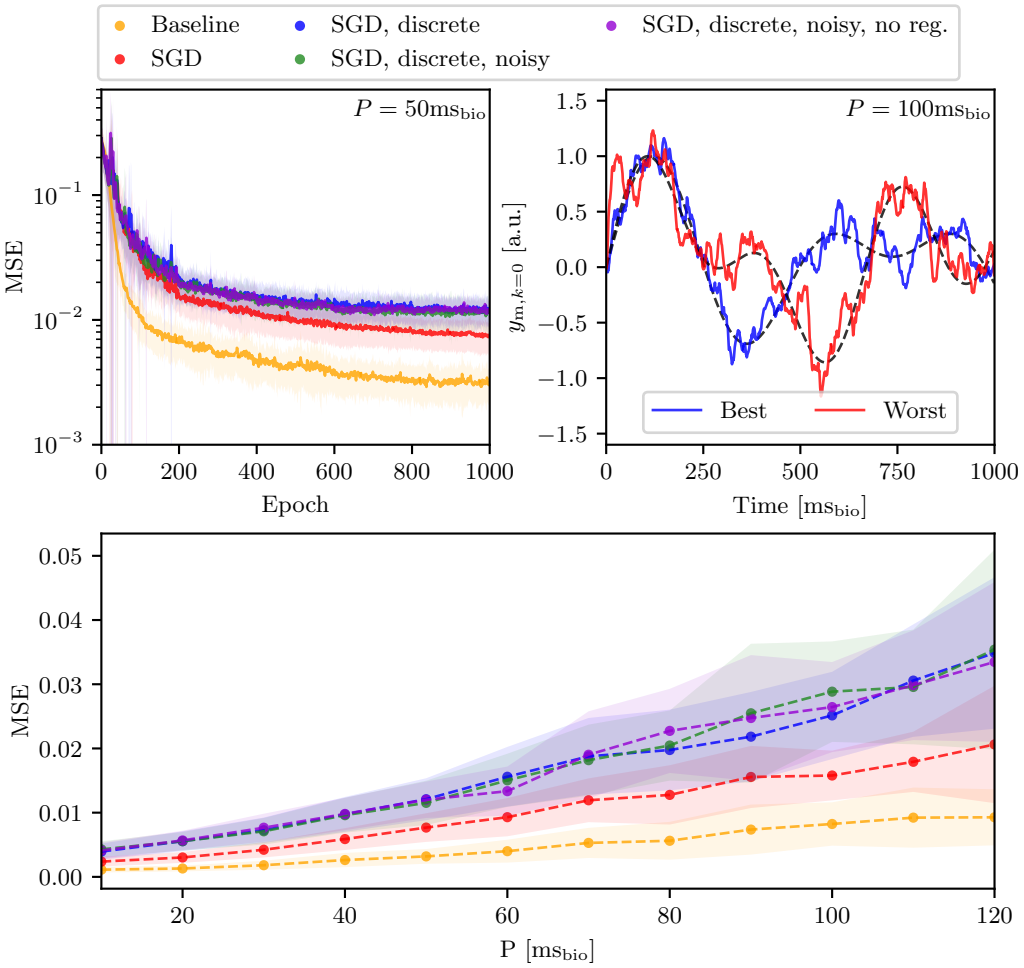


Figure 6.6: NASProp performances in dependence of P. "SGD" corresponds to the simulation without any hardware properties, but using SGD for weight updates, "SGD, discrete" introduces additional weight discretization and "SGD, dicrete, noisy" is the simulation done in a noisy environment. In addition, "SGD, discrete, noisy, no reg." does neglect any firing rate penalization. The **upper-left** plot shows the MSE over epochs for the different learning settings, trained with $P = 50\,\text{ms}_{\text{bio}}$. On the **upper-right**, the best and worst performing traces are given together with their target, both trained with $P = 100\,\text{ms}_{\text{bio}}$. Even with long update periods, the traces do still follow the general idea of the target pattern. In the **lower** plot, the MSE loss is depicted over the update period P (ensemble average over the last 50 epochs). It can be observed that the loss increases linearly with P. For discrete weights the loss increases faster. Parameters are given in Table A.10.

6.2.5 The Role of Recurrence

As learning recurrence is a crucial element of this thesis, the NASProp learning rules should exhibit positive effects when training recurrent weights. Therefore, similar to the experiments in Section 5.4.3, performances are compared between recurrent and non-

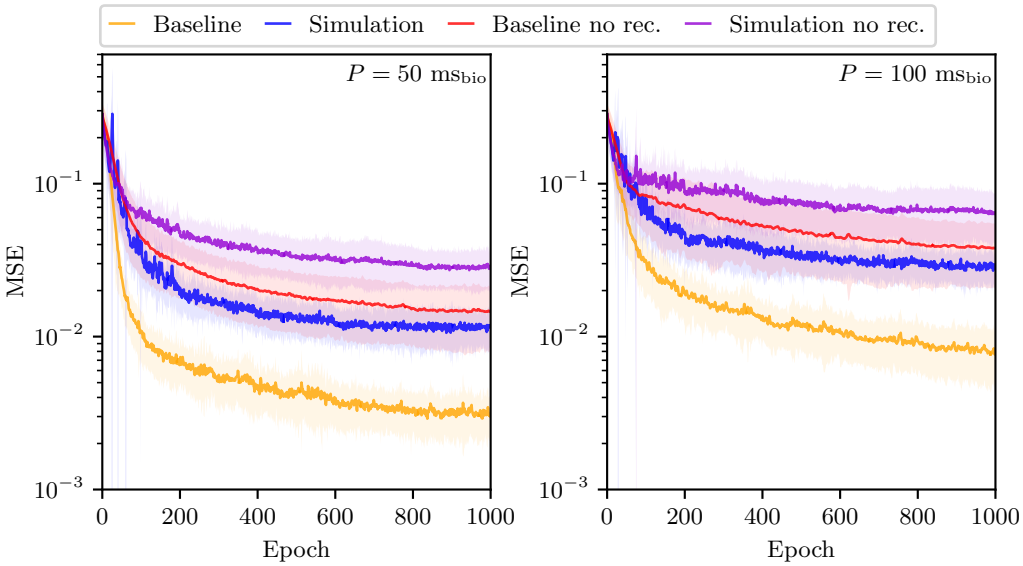


Figure 6.7: Comparison of non-recurrent and recurrent NASProp training with $P = 50\,\text{ms}_{\text{bio}}$ (**left**) and $P = 100\,\text{ms}_{\text{bio}}$ (**right**). The orange (recurrent) and red (non-recurrent) lines correspond to training without hardware constrains. Blue (recurrent) and violet (non-recurrent) curves incorporate hardware constraints (noise and discrete weights). In both cases it can be observed that using recurrent connections does clearly improve the performance. Parameters are given in Table A.11.

Setup	$P = 50\,\text{ms}_{\text{bio}}$ $[10^{-3}]$	$P = 100\,\text{ms}_{\text{bio}}$ $[10^{-3}]$
Baseline	3.15 ± 0.95	8.12 ± 3.30
Baseline no rec.	14.77 ± 6.76	38.18 ± 17.27
Simulation	11.51 ± 2.99	28.66 ± 6.63
Simulation no rec.	28.51 ± 8.66	65.14 ± 24.04

Table 6.1: Ensemble average MSEs of the experiments in Figure 6.7. The errors denote the standard deviation over the ensemble.

recurrent networks, both trained with the update rules at hand. The training process is shown in Figure 6.7 and the resulting MSEs are summarized in Table 6.1.

The figure compares performances of the baseline setup and the simulation setup with hardware constrains between corresponding non-recurrent and recurrent networks, both trained with $P = 50\,\text{ms}_{\text{bio}}$ and $P = 100\,\text{ms}_{\text{bio}}$. The baseline and the simulation experiments converge to lower MSE values for both update periods when adding recurrent connections. This MSE values in Table 6.1 support this and show an increasing error compared to the mean values as recurrence is removed, suggesting that the network's performance depends stronger on the target patterns. Even though the baseline network seems to exploit the recurrence stronger than the network with hardware constraints, both do benefit from it. Hence, these experiments suggest that NASProp enables RSNNs to utilize their recurrent connections in a meaningful fashion.

The simulations and experiments in the previous sections indicate that the NASProp learning rules can solve the pattern-generation task sufficiently, even when taking hardware properties into account and using longer update periods P. At this point, it hence seems promising to implement NASProp on the PPU to enable on-chip plasticity. This is approached in the subsequent sections.

6.3 Implementation on-chip

The learning rule described in Equation (6.13) and (6.15) is implemented in a class `NASProp` written in C++ based on 8-bit fractional arithmetic. On the PPU, this class is instantiated as `rule` when the PPU program is loaded by `grenade` (see Section 4.2.4) and performs on-chip weight updates. This section will outline the implementation and discuss associated issues.

Training Preparation

Learning on-chip is abstracted in software as outlined in Section 4.2.4. Before batch execution, PPU parameters `PPUParams` are serialized onto the PPU into an object `setup`. This object contains hyperparameters `RuleParams` for the learning rule and is set by `rule.setup(setup.params)`. This object defines the update period P by `period` and the number of weight updates `n_periods` needed for update timing. Additionally, this setup object adjusts some debugging options, like logging of parameters and update times.

```
struct PPUParams {
    // ... Ctors
    uint32_t period;
    uint32_t n_periods;
    uint32_t runtime;
    bool log_params;
    bool log_timing;
    RuleParams params;
};
```

```
struct UpdateParams {
    // ... Ctors
    uint8_t row;
    uint8_t neuron_rec;
    uint32_t offset;
    bool is_rec_row;
    std::array<int8_t, 100>
    ↪ target;
};
```

The `params` object contains all *constant* parameters (i.e., parameters that do not change from trial to trial) needed for learning, such as $\hat{\kappa}$, η, and boolean masks defining the network's topology (see Section 4.2.4). *Dynamic* parameters (i.e., parameters changing from trial to trial) are serialized onto the PPU before the execution of every single trial and are given by an instance of type `UpdateParams`. Firstly, since the learning rule performs synapse-row-wise updates (*cf.* following sections), this object tells the learning rule which `row` to perform updates for and, secondly, whether this row is part of the recurrent projection by `is_rec_row`. Here `neuron_rec` is the index of the recurrent neuron projecting into the `row`. The trial's random offset $\tilde{t}$ is given by `offset`. The

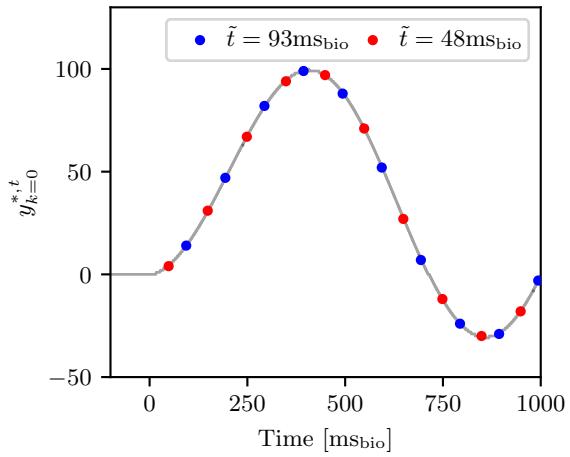


Figure 6.8: The target pattern $y_k^{*,t}$ is communicated to the PPU with an 8-bit resolution. The PPU does only need to know the pattern's values at update times t^n. These values change with each trial due to the random offset. Two examples are depicted by the dots for different offsets.

target pattern at the expected weight updates at t^n for the current offset is given by `target` with an 8-bit resolution (see Figure 6.8). Note, all data in this object is sampled randomly before each training trial (by `Setup::Config::sample`, see Section 4.2.4).

Timing

When `grenade` triggers the start of the plasticity rule, the weight updates are timed on the PPU-side as defined in Algorithm 1. Therefore, weight updates at t^n are calculated when the method `update` is called on the learning rule. Before the first update period begins — after the random offset $\tilde{t}$ has passed — some hardware observables need to be reset by calling `prelude`.

Algorithm 1 Timing of Weight Updates

```
1: Input n_periods, offset
2: procedure TIMING                                  ▷ While forward pass
3:     wait(offset)                                  ▷ Wait for offset t̃
4:     rule.prelude()                                ▷ Reset observables, measure baselines
5:     for n ← 1 to n_periods do
6:         wait(period)                              ▷ Wait for period P
7:         rule.update()                             ▷ Calcualte weight updates
8:     end for
9: end procedure
```

Row-wise Updates

The on-chip implementation is designed to perform synapse-row-wise updates, such that with each training trial, only the weights on a single signed hardware row are updated. This is motivated by the VU, which allows accessing hardware observables row-wise and performing vector arithmetic to process this data in parallel. This would still allow to loop over all used hardware rows and thus compute updates for all used weights, which is desirable. However, computing these updates requires measuring the correlation at each

Table 6.2: Time measurements of PPU operations. The column label **row** addresses operations corresponding to a signed row (correlation, weights) and **neuron** a single neuron (counter), respectively. **All** corresponds to 128 signed rows, resp. 128 neurons. Time units are given in hardware time. Cycles correspond to PPU processing cycles.

Operation	**Row / Neuron**		**All**	
	[cycles]	[μs]	[cycles]	[μs]
Correlation read	402	1.608	49712	198.8
Correlation reset	151	0.604	54781	219.12
Counter read	276	1.104	33740	134.96
Counter reset	20	0.08	2621	10.484
Weights read	137	0.548	14755	59.02
Weights write	40	0.160	3078	12.312

time t^n followed by a reset (plus all operations necessary to compute the final update). As each measurement is time-consuming (see Table 6.2), this is rather slow and would result in time delays between rows-wise updates and thus biased weights[4]. Assuming a time sequence of $1000\,\mathrm{ms_{bio}}$, measuring the correlation for 128 signed rows alone already takes about 20% of the whole sequence for *each* t^n. This is only acceptable if the relative time delay between row-wise updates is reduced by slowing down the neuron dynamics and considering much longer sequences (this is subject to future work), or as stated, by only updating a single signed row with each trial. Due to the speed-up factor of HX, row-wise updates are not assumed to pose a big issue as the total weight update (i.e update of all weights at once) is expected to be approximated well over many epochs, updating different rows with a small enough learning rate.

Calculating Weight Updates

The learning rules in Equation (6.13) can be written as,

$$\Delta\theta_{ji}^{\{\mathrm{ih},\ \mathrm{hh}\}} = -\eta \sum_n \Delta\theta_{ji}^{\{\mathrm{ih},\ \mathrm{hh}\},n}, \tag{6.18}$$

where $\Delta\theta_{ji}^{\{\mathrm{ih},\ \mathrm{hh}\},n}$ is the contribution to the weight update at t^n. Note, for row-wise updates i is constant within each trial (*cf.* Figure 4.4). In order to compute $\Delta\theta_{ji}^{\{\mathrm{ih},\ \mathrm{hh}\},n}$ for the weights θ^{ih} and θ^{hh}, the learning rule requires reading out the correlation c_{ji}^n for the eligibility traces and the membrane potentials y_k^n to compute the learning signals (so far only a single readout neuron is supported, therefore index k is omitted in the following.). Since both observables are measured via the CADC, the PPU redirects the CADC to digitize the desired analog entity. The resulting 8-bit values are read out by the VU and stored in a 128 8-bit word vector, with the vectors entries at index j corresponding to neuron j. Therefore, the network's neurons on HX are only placed on even columns, such that all used neuron columns are on the same "vector slice" (see Section 4.2.4). This reduces the VU instructions needed, and thus the time to read out

[4]Provided each row update assumes the same learning signal, i.e., membrane readout and target value. If this were not the case, for each row, the target pattern taking into account this delay would need to be provided together with the corresponding membrane readouts.

and process observables . In addition to the correlation and membrane measurements, the output weights $\theta_{0j}^{\mathrm{ho}}$ need to be given (see Equation (6.13)). Due to the routing algorithm in Section 4.2.3, these weights are arranged in columns. Hence, they are read out once before batch execution and stored in a vector where the entry of output weight $(0,j)$ matches the index of the corresponding recurrent neuron column j relative to row-wise vector readouts. This eases later vector operations enormously. Once the output weights are read out, the vector keeps track of corresponding updates such that no further column-wise access is required across trials.

The CADC measurements are shifted by their baseline, measured before trial execution, and converted from `uint8` to a signed `int8` representation. The signed error $(y^{*,n} - y^n)$ is equal for all used neuron columns, and is calculated with the scalar unit and subsequently provided in a vector. This vector is multiplied with the vector holding $\theta_{0j}^{\mathrm{ho}}$, giving the learning signals $\theta_{0j}^{\mathrm{ho}}(y^{*,n} - y^n)$. Correlation measurements c_{ji}^n at t^n are added to a propagation vector which is propagated at each n by multiplication with the decay constant $\hat{\kappa}$ and afterward multiplied with the learning signals, finally giving the weight updates. Thus, the nth contribution to weight update is calculated in parallel for the given `row` according to the summand in Equation (6.13).

As depicted in Figure 4.4, recurrent and output weights $\theta^{\{\mathrm{hh,ho}\}}$ are arranged next to each other in a synapse row on HX, while rows used for the input projection hold input weights θ^{ih} exclusively. Hence, for the first row type, two distinct learning rules need to be applied. Therefore, these two row-types are distinguished by `is_rec_row`. Due to the similarity of the learning rule for recurrent and output weights, the weight update $\Delta\theta_{0j}^{\mathrm{ho}}$ can be computed together with $\Delta\theta_{ji}^{\mathrm{hh}}$ by replacing the correlation in the correlation readout vector at the index of the readout neuron's column with the corresponding spike counter readout ζ_j^n of the recurrent neuron j at index `neuron_rec`. In contrary to the input and recurrent weight updates, the signed error for the output weights $(y^{*,n} - y^n)$ is not multiplied with $\theta_{0j}^{\mathrm{ho}}$ and is masked out when calculating the learning signals by multiplying the signed error $(y^{*,n} - y^n)$ and $\theta_{0j}^{\mathrm{ho}}$ vectors.

Applying Stochastic Weight Updates

The resulting weight updates are given in a vector $\Delta\theta^n$ with `int8` entries $\Delta\theta_l^n$ (where l indexes the vector) and cannot be applied directly to the 6-bit synaptic hardware weights. Instead, each weight update in $\Delta\theta^n$ is interpreted as a signed 8-bit fractional two's-complement number with lower bound $-128 \to -1_{\mathrm{F}}$ and upper bound $127 \to 0.9921875_{\mathrm{F}}$. Then, the updates in $\Delta\theta^n$ at update time t^n are rounded stochastically with probability p according to,

$$\Delta\theta_l^{\mathrm{hw},n} = \begin{cases} \mathrm{sign}\left(\Delta\theta_l^n\right) \cdot 1 & \text{with } p = \left|\Delta\theta_{l,\mathrm{F}}^n\right| \\ 0 & \text{with } 1-p, \end{cases} \tag{6.19}$$

where $\Delta\theta^{\mathrm{hw},n}$ is the resulting hardware update vector for the synapses in the considered (signed) row. In order to compute the stochastic updates on the PPU, the chip's random number generator is used, which allows sampling 128-byte vectors r uniformly. The random entries of r are bit-shifted to the right, yielding positive signed 8-bit values,

Table 6.3: Time measurements of `NASProp` operations. "Time critical" describes measurements and resets of all observables that should be done in a minimal time window. "Time critical rec. row" encompasses additional counter readout. Time values are averaged over 100 measurements and refer to accumulating updates.

Operation	Time	
	[cycles]	[μs]
`NASProp::update`	13137	52.548
`NASProp::prelude`	458	1.832
`NASProp::set_params`	3355	13.42
Time critical	1282	5.128
Time critical rec. row	1800	7.2

which can be compared element-wise to $|\Delta\theta^n|$ to decide whether to adjust a weight θ_l by $1 \cdot \text{sign}\,(\Delta\theta_l^n)$ or not. Finally, the updates are applied to the synaptic weight directly *on-the-fly* or are accumulated according to

$$\Delta\theta^{\text{hw}} = \sum_n \Delta\theta^{\text{hw},n} \tag{6.20}$$

and applied after calculating the last update at t^N. As making weight update less probable corresponds to reducing the learning rate, the random numbers in r can be bit-shifted additionally, which effectively makes updates more (shift to the right) or less (shift to the left) likely. Stochastic updates can also be applied to the accumulated updates $\Delta\theta^{\text{hw}}$ which decreases the learning rate even more. Note, the PPU places negative weights on the inhibitory row of the signed synapse row, and positive weights on the excitatory row as required in Section 4.2.3.

Fractional Arithmetic

The learning rules are implemented on the PPU in assembler code using the VU's instruction set [Friedmann et al., 2020]. Vector operations are performed in saturating fractional arithmetic to prevent overflow as the instruction set does not provide normal saturating byte-vector operations with overflow detection. Since the learning rule in Equation (6.13) is a three-factor rule (weight, signed error, correlation) that runs easily into overflow when operating on 8-bit data representations, saturating arithmetic is important. However, as shown later, for a three-factor rule, fractional arithmetic can result in vanishing weight updates if the factors become too small.

6.3.1 Speed of Weight Updates

In theory, the weight updates are computed at times t^n. However, in practice, the PPU takes time for measurements and calculations. While the time the PPU needs to compute an update $\Delta\theta^n$ defines a minimal update period P (which preferably is small), the actual duration of the computation is not time-critical if the weight updates are accumulated.

Table 6.3 shows the operating times of `NASProp` methods. If updates are accumulated, calculating the updates of a signed synapse row takes about $53\,\text{ms}_{\text{bio}}$. Taking into account

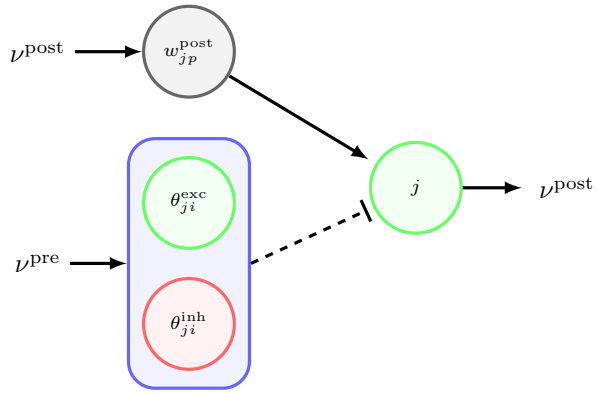


Figure 6.9: Setup for a single signed synapse experiment. The synapse ji receives Poisson-like pre-synaptic spike events with frequency ν_{ji}^{pre}. Neuron j is triggered to spike with frequency ν_{ji}^{post}, by events through synapse jp. This is possible by enabling the neuron's bypass mode and disabling the synaptic connection from signed synapse ji to neuron j.

a safety margin, an update period of $P = 60\,\text{ms}_{\text{bio}}$ is realistic. Ideally, measurements and resets of all required observables should happen at t^n to coincide with the provided target $y^{*,n}$. As the PPU accesses and resets observables successively, this is not possible, and measurements are time-shifted to each other, which possibly biases the updates. Therefore, these time-critical operations are done at the very beginning of `NASProp::update` and take about $5.128\,\text{ms}_{\text{bio}}$ for an input row and $7.2\,\text{ms}_{\text{bio}}$ if a recurrent row is updated (due to additional counter readout). Considering a total time sequence of $1000\,\text{ms}_{\text{bio}}$ with $P = 100\,\text{ms}_{\text{bio}}$, the time-critical operations for $N = 10$ updates require about 5.128%, respectively 7.2%, of the whole sequence. Hence, the total time-shift in observable related computations is not assumed to impact the updates dramatically; however this remains to be investigated.

6.4 Single Synapse Experiment

In order to test the on-chip implementation of learning rule on HX, an artificial single-synapse learning environment is set up. This allows verifying the implementation in a simple experiment. Since writing PPU programs in assembler code is prone to errors, such test-driven development is crucial.

6.4.1 Experiment Setup

To investigate the weight evolution of a *signed* synapse ji over time, according to Equation (6.13), the synapse requires pre-synaptic input and post-synaptic spike events in order to measure correlation. However, since only a single synapse is considered, it is difficult for the target neuron j to exhibit a reasonable spiking behavior. Fortunately, the learning rule does not depend on the neuron's membrane dynamic, and post-synaptic spike events can be triggered artificially. Therefore, on HX, post-synaptic spikes are imposed through another synapse jp with weight w_{jp}^{post}. By enabling the bypass-mode for neuron j, each pre-synaptic event arriving in synapse jp will cause neuron j (almost) immediately to spike. To prevent events through synapse ji to trigger spikes itself, the connection from synapse ji to neuron j is disabled. In effect, the correlation sensor

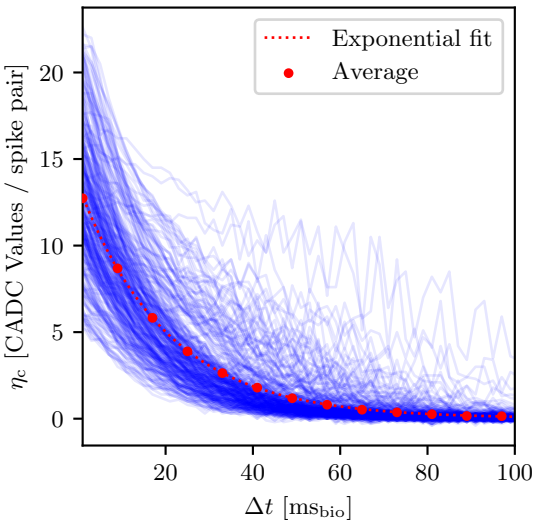


Figure 6.10: Example correlation between pre- and post-events over Δt for a subset of synapses on HX. Due to transistor variations, each synapse measures the correlation slightly different. The curve differ in amplitude, time constant and offset. Some synapses behave rather undesired and have rough, non-exponential correlation traces. This is assumed to be due to a hardware bug in the current chip release. Parameters are given in Table A.12.

in synapse ji measures correlation due to pre-synaptic events in synapse ji and post-synaptic events of neuron j enforced by synapse pj. Figure 6.9 visualizes the setup. The signed synapses weight θ_{ji} is realized by a weight θ_{ji}^{inh} on an inhibitory synapse row, and a weight θ_{ji}^{exc} on an excitatory row (*cf.* Section 4.2.3). Pre- and post-synaptic events are generated by the on-chip spike-generator as Poisson-spike trains with mean frequency ν^{pre} and ν^{post}, respectively.

6.4.2 Correlation Measurements

Since the learning rule for the input and recurrent weights depends on correlation measurements, the shape of the correlation curves influences learning. The learning rule in Equation (6.13) requires the correlation time constant to be equal to the membrane time constant of the recurrent neurons, i.e. $\tau_{\text{c}} = \tau_{\text{m}}$. Assuming $\tau_{\text{m}} = 20\,\text{ms}_{\text{bio}}$, the correlation sensors on HX are parameterized to meet this requirement as close as possible. Since correlation sensors measure correlation in analog, they differ from synapse to synapse and are subject to fixed-pattern noise (see Section 3.2). This can be seen in Figure 6.10, which depicts the measured correlation amplitude of several synapses over the time difference between a pre-synaptic and a post-synaptic spike event Δt. While some curves are smooth, others are rather rough and do not satisfy the requirement of an exponential dependence of the amplitude on Δt. It remains to be investigated how the variation of correlation curves disturbs learning. An exponential fit on the average curve (red dots) suggests a time constant of $\tau_{\text{c}}^{\text{av}} = 20.179\,\text{ms}_{\text{bio}}$ and a correlation amplitude of $\eta_{\text{c}}^{\text{av}} = 13.475\,\text{CADC}$ values per spike pair. The synapses' correlation calibration bits are chosen such that the synapses' correlation curves are as close to the target as possible.

6.4.3 Learning Setup

In order to mimic learning in a full network setup, solving the pattern-generation task, each training epoch considers a sequence of length $T = 1000\,\mathrm{ms_{bio}}$ ($+P$ due to the random offset) and an update period of $P = 100\,\mathrm{ms_{bio}}$, making $N_P = 10$ updates per trial. Assuming $\tau_{\mathrm{m},k} = 20\,\mathrm{ms_{bio}}$ for the readout neuron, $\hat{\kappa}$ is approximately zero.[5] In each epoch, a new offset is drawn uniformly. The updates at t^n are accumulated and applied at the end of the sequence. When considering only a single synapse, the membrane of the readout neuron does not receive any input and cannot follow a given target pattern. Therefore, the error signal $(y^{*,n}_{k=0} - y^n_{k=0}) = 100\,\mathrm{CADC}$ values is kept constant. Both the pre- and post-synaptic average firing frequency is chosen to $\nu^{\mathrm{pre}} = \nu^{\mathrm{post}} = 100\,\mathrm{Hz_{bio}}$. If these parameters change, it is stated explicitly. Note, the experiment configurations in the following do not claim to exhibit a realistic learning environment by any means but are intended to emphasize crucial learning characteristics.

6.4.4 Exemplified Weight Evolution

For the single synapse setup just stated, the weight evolution is investigated for recurrent and output weights. Updating input weights behaves equally to the recurrent weights.

Recurrent and Input Weight

As suggested by the NASProp plasticity rule in Equation (6.13), the weight update $\Delta\theta^{\mathrm{hh}}_{ji}$ has a positive sign if the output weight $\theta^{\mathrm{ho}}_{kj}$ and the signed error $(y^{*,n}_k - y^n_k)$ are positive. Thus, when pinning the output weight to a constant value $\theta^{\mathrm{ho}}_{kj} = 25\,\mathrm{a.u.}$, the recurrent weight, initialized as $\theta^{\mathrm{hh}}_{ji} = -63\,\mathrm{a.u.}$, is expected to increase approximately linearly over epochs if the correlation readout remains constant on average.

In Figure 6.11, this behavior is confirmed. Over approximately 90 epochs, the weight θ^{hh} constantly grows until it saturates at $\theta^{\mathrm{hh}} = 63\,\mathrm{a.u.}$. When the weight changes its sign from minus (red) to plus (blue), the PPU places the weight from the inhibitory to the excitatory hardware row. The jitter in the weight evolution is due to stochastic updates and the stochasticity of correlation readouts owing to Poisson-like spike events. The middle plot in the figure shows the weight update $\Delta\theta^{\mathrm{hh},n}$ at each t^n for each epoch (i.e. $N_P = 10$ update values per epoch). Note, even if the update might indicate a weight increase, θ^{hh} does not adapt the update immediately since updates are applied accumulatively at the end of each epoch. The lowermost plot depicts the correlation measured before each update computation.

[5] This is because $\hat{\kappa}$ is represented on the PPU by a positive fractional signed 8-bit number with a smaller precision than $\hat{\kappa} = \exp(-P/\tau_{\mathrm{m}})$ for the given values.

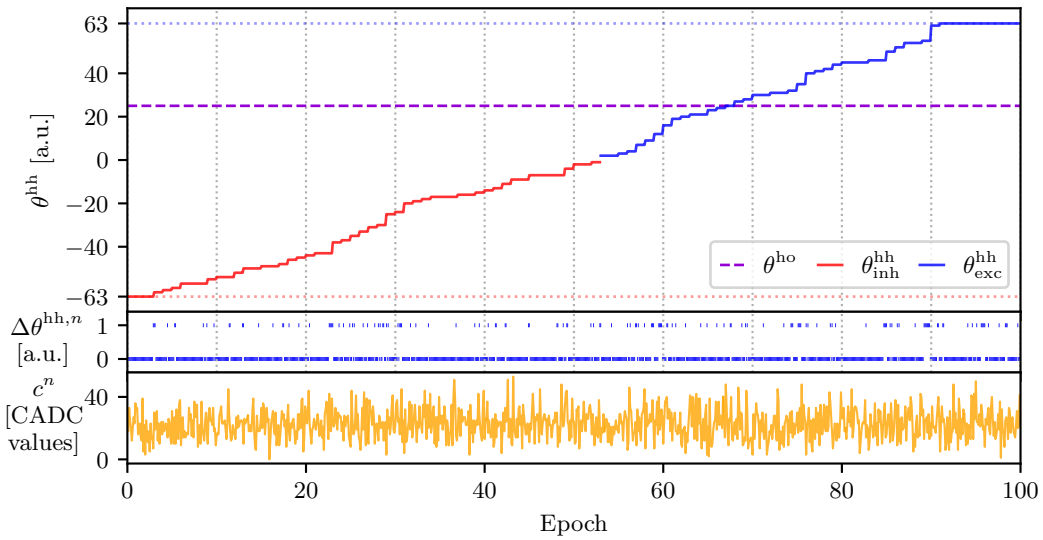


Figure 6.11: Weight evolution of a single recurrent synapse. Due to a constant output weight θ^{ho} and a constant signed error of 100 a.u., the weight θ^{hh} increases linearly over epochs. This is shown in the **upper** plot. The red line corresponds to the inhibitory weight and the blue to the excitatory, respectively. For each epoch, the **middle** plot shows the corresponding updates at t^n. The updates get accumulated over $N = 10$ update intervals and are applied in the end of each trial. The **lower** plot shows the corresponding correlation readouts. Parameters are given in Table A.13.

Output Weights

The output weight update $\Delta\theta^{\mathrm{ho}}$ depends, according to Equation (6.15), only on the signed error and the counter readout. Since the spike counts are always positive, the weight θ^{ho} must change at each update n in the direction given by the error's sign.

This is also observed in Figure 6.12. The output weight θ^{ho}, initialized at $\theta^{\mathrm{ho}} = -63$ a.u., adjusts constantly upwards — again with a stochastic component — until saturation. The corresponding counter readouts ζ^n fluctuate around $\langle\zeta^n\rangle = 9.326$ spikes per update period, as demanded by the post-synaptic event density of $\nu^{\mathrm{post}} = 100\,\mathrm{Hz_{bio}}$ and $P = 100\,\mathrm{ms_{bio}}$.

As the output weight changes, it influences the evolution of the recurrent weight update $\Delta\theta^{\mathrm{hh}}$. As long as the output weight θ^{ho} has a negative sign (red), the recurrent weight θ^{hh} decreases, however, not linearly but slower with decreasing output weight until it reaches a minimum where θ^{ho} approaches zero. When θ^{ho} becomes positive (blue), the recurrent weight does increase as well. The corresponding updates are given by $\Delta\theta^{\mathrm{hh},n}$ and are negative for epochs with a negative output weight.

Generally, the weight evolutions shown in the upper plots are expected to flip signs when choosing a negative signed error $(y_k^{*,n} - y_k^n)$. This is visualized in the upper plot in Figure 6.13. Here, the first 100 epochs assume a positive error signal, and the next 100 epochs a negative one. As long as the error signal is positive, the output weight

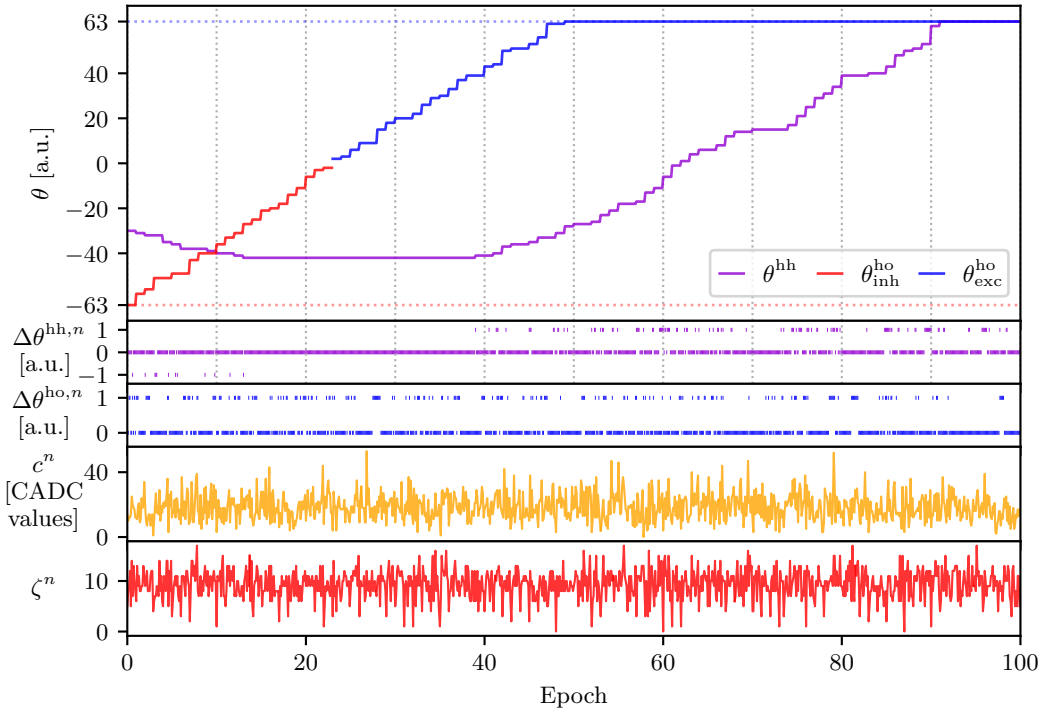


Figure 6.12: Weight evolution of a output weight θ^{ho}. Since this weight is element of the recurrent plasticity rule, the evolution of weight θ^{hh} depends on it. The **uppermost** plot shows the development of these weights over epochs. Updates are applied accumulative. For each epoch the updates at t^n are depicted by $\Delta\theta^{\text{hh},n}$ and $\Delta\theta^{\text{ho},n}$, respectively. The corresponding correlation readout for the recurrent weight update c^n is shown in orange, the counter readout ζ^n for the output weight update in red. Both observables fluctuate due to the Poisson-like pre- and post-synaptic spike events. Parameters are given in Table A.14.

θ^{ho} increases. After the error flips its sign, the weight decreases again. Correspondingly, in epochs with positive error, the recurrent weight θ^{hh} decreases as long as the output weight is negative. When the output weight becomes positive, θ^{hh} increases. However, if the error signal has a negative sign, the recurrent weight decreases for positive and increases for negative output weights. This is exactly what the learning rules suggest.

Error Signal

In Figure 6.13, the middle plot shows the evolution of a recurrent weight with different signed errors (colors), $(y_k^{*,n} - y_k^n) \in \{-100, -80, -40, -20, 20, 40, 80, 100\}$ CADC values. Measurements are averaged over 20 runs with different seeds. Again, the output weight $\theta^{\text{ho}} = 30$ a.u. remains constant. The recurrent weight is initialized to the maximum value with the inverse sign as the corresponding error signal. For runs with a negative error signal, the recurrent weight decreases constantly. Correspondingly, if the error signal is positive, the weight increases. As desired, the speed of weight adjustment increases with

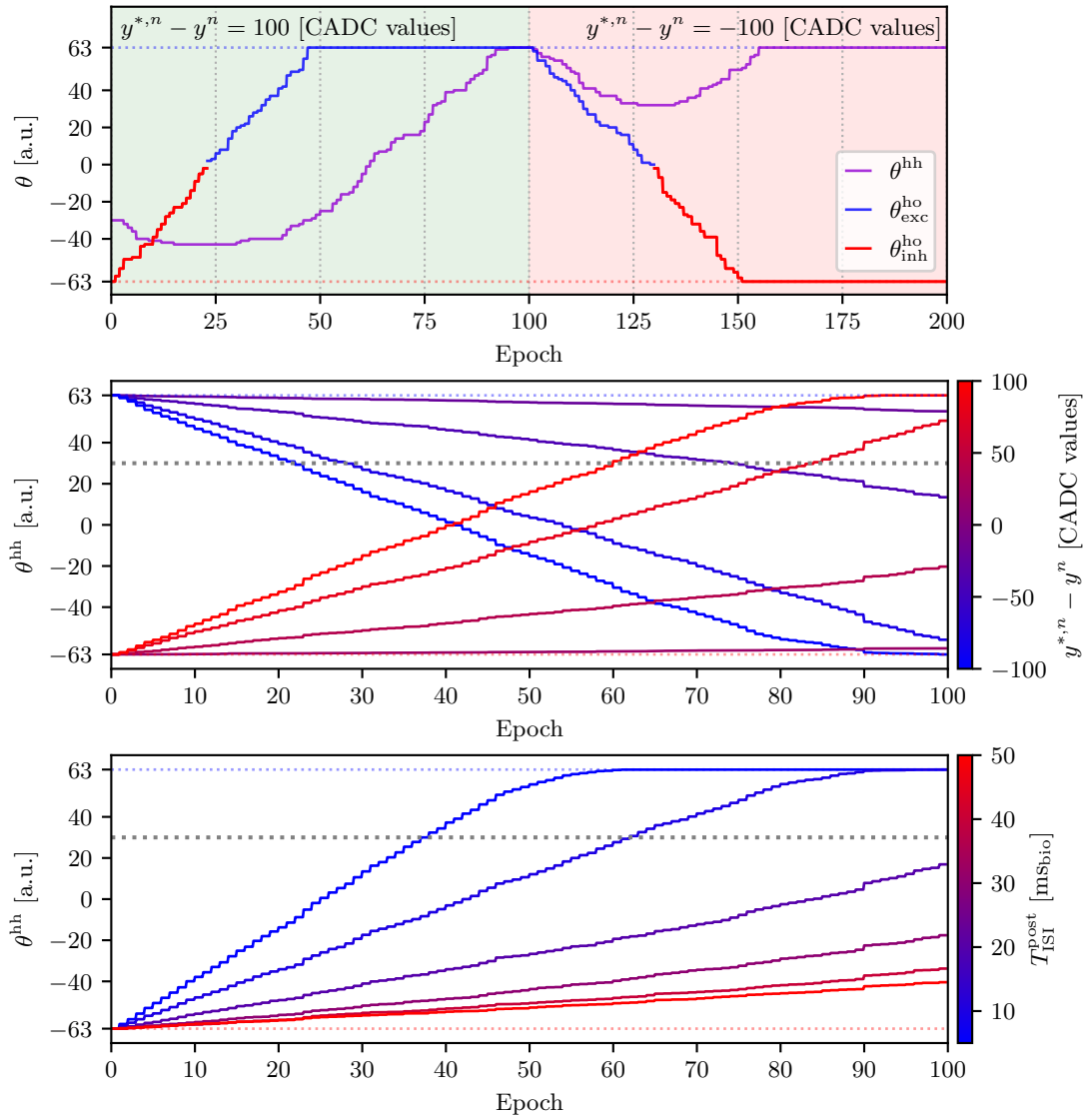


Figure 6.13: Investigation of weight development over epochs. **Upper**: Example of relative weight development of a recurrent weight θ^{hh} (violet) and a corresponding output weight θ^{ho}. After 100 epochs the error signal changes its sign. The output weight evolves into the direction of the error signal. For a positive error, θ^{hh} changes into the direction given by the output weight's sign. And vice versa for a negative error. Parameters: Table A.15. **Middle**: Weight development of the recurrent weight for different error signals, indicated by the color for a constant output weight. Curves are averaged over 20 runs. Parameters: Table A.16. **Lower**: Weight development of the recurrent weight for different post-synaptic inter-spike intervals $T^{\mathrm{post}}_{\mathrm{ISI}}$, indicated by the color. Curves are averaged over 20 runs. Parameters: Table A.15, Table A.16, Table A.17.

the absolute value of the error signal $|y_k^{*,n} - y_k^n|$ and for small errors, the weight does only change slowly. For $|y_k^{*,n} - y_k^n| = 20$ CADC values, the weight remains almost constant. Choosing smaller errors (or smaller θ^{hh}), would result in vanishing weight updates due to fractional multiplications in the learning rule.

Correlation

By increasing the post-synaptic inter-spike interval $T_{\text{ISI}}^{\text{post}}$ (i.e., decreasing ν^{post}), the correlation sensor should accumulate less correlation. Consequently, for a constant learning signal, the recurrent weight must change slower. This is confirmed in the lowermost plot in Figure 6.13. Increasing $T_{\text{ISI}}^{\text{post}}$ (indicated by the colors), yields smaller correlation readouts and thus less weight adjustments on average. Hence, the weight increases slower.

Finally, the observations show the desired behavior and verify the on-chip PPU plasticity implementation within the scope of testing. Furthermore, this single synapse experiment gives a clear insight into how NASProp learning behaves and adjusts weights.

6.4.5 Synapse Variations

The correlation curves in Figure 6.10 suggest a varying weight update dynamic between synapses, since the actually correlation readout differs widely. Differing correlation measurements translate into a changing weight updates in two ways. Firstly, a smaller (bigger) correlation amplitude of a synapse corresponds to a smaller (bigger) synapse-specific learning rate as this decreases (increases) the correlation readout c_{ji}^n. Secondly, a smaller (larger) correlation time constant leads to a smaller (bigger) penalization of a wrongly placed post-synaptic spike event since the correlation curve models the eligibility vector, which itself tracks how much a pre-synaptic event is responsible for the networks error at the occurrence of the next post-synaptic spike (see Section 6.1.2). Hence, smaller (bigger) time constant yield smaller (bigger) correlation readouts c_{ji}^n at t^n and, effectively, giving different absolute update contributions at time t^n.

The just discussed effect is visualized in Figure 6.14. Here, the speed of weight adjustment is compared between 70 different synapses on the same synapse row, again with constant signed error, and constant output weight, as in Section 6.4.4. Therefore, the weight traces are assumed to be approximately a linear function of the epoch. Then, a linear function is fitted on each weight trajectory such that the slope of the function is a proxy of how fast a particular synapse adjusts its weight.

As can be seen in the given figure, the weights of 10 example traces (faint violet) develop differently fast. The weight of the "fastest" synapse is increased by 4.063 a.u. per epoch on average over 10 runs (dotted orange line), while the "slowest" synaptic weight is increased much slower with only 0.769 weight increments per epoch (dotted red line). The synapse-ensemble mean suggest that a synapse changes its weight by $2.337\,\text{a.u.} \pm 0.914\,\text{a.u.}$ per epoch on average.

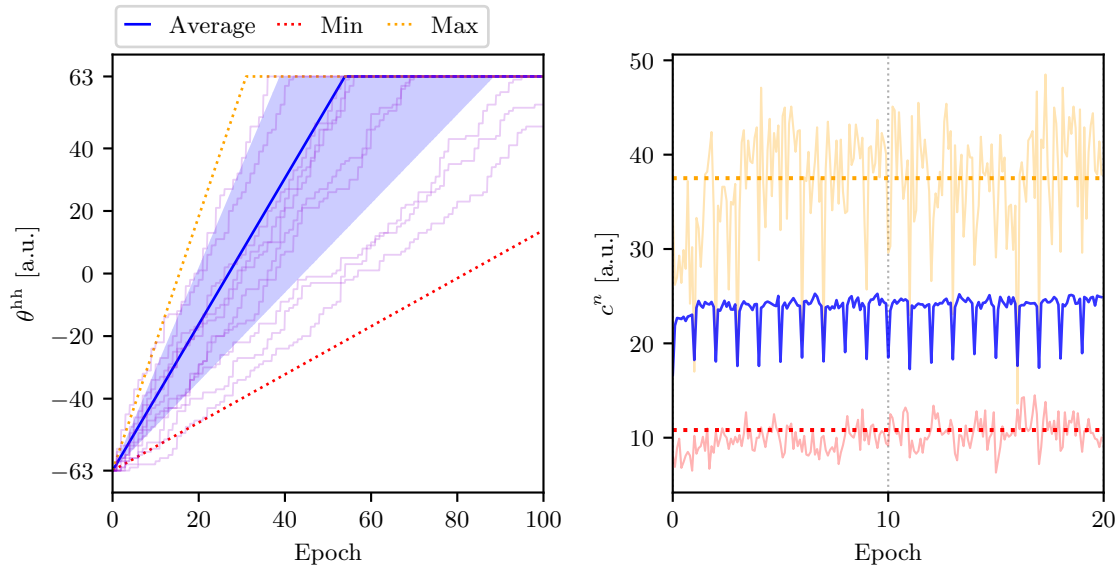


Figure 6.14: The **left** plot shows 10 example traces of different synapses (faint violet). On each weight trace, a linear function is fitted to get a proxy of the average speed of weight growth. The fits are averaged over 10 runs. The blue line corresponds to the average weight evolution over 70 considered synapses. The "slowest" changing synaptic weight is depicted by the dotted red line, and the "fastest" weight evolution by the dotted blue line. The faint blue area shows the standard deviation, with borders being the slope of the average line plus/minus the standard deviation of all slopes (averaged over 10 runs). The **right** plot depicts the corresponding measured correlation after each update period n. Here, the blue line is the synapse-ensemble mean at each t^n. The drops are due to the random offset causing less correlation accumulation in the first update interval of each epoch. The correlation measurement depicted in red, corresponds to the "fastest" increasing synapse, and the correlation shown in blue to the "slowest" increasing synapse. Correlation measurements are also averaged over 10 runs. Parameters are given in Table A.18.

For the "fastest" increasing synapse in the figure is significantly more correlation measured than for the "slowest" synapse. This is due to correlation curve variation and explains the different speeds of weight adjustment. Since the time interval $t \in [-P, 0)$ of a certain trail is per definition free of activity (due to the random offset), the first correlation readout $c^{n=1}$, accumulating correlation in $t \in (-P + \tilde{t}, \tilde{t}]$, contains less correlation than for update intervals $n > 1$. This is why the correlation averaged over the synapse ensemble drops in the beginning of each epoch.

6.5 Full Network

The previous sections aim all towards the higher goal of actual training a full-scale network on-chip. This challenge is motivated by the simulations in Section 6.2 and sup-

ported by the verification of the learning rule implementation presented in the previous Section 6.4. However, at the time of writing, this goal is not entirely reached yet. Therefore, this section is intended to show that the general software setup, as outlined in Section 4.2.4, provides the ability to perform on-chip learning by using high-level chip abstraction software frameworks in principle. Further, the obtained results for a given training procedure are shown, and the encountered issues are discussed.

6.5.1 Training Procedure

As in simulations, a network with $n_\mathrm{i} = 30$ input neurons and $n_\mathrm{h} = 70$ recurrent neurons, projecting on a single readout neuron, is considered. According to the routing algorithm in Section 4.2.3, HX allocates $n_\mathrm{h} + n_\mathrm{i} = 100$ signed synapse rows for this. Since only a single row is updated in each trial, a batch size of 100 is used. Each element in this batch holds a randomly sampled offset with corresponding targets $y^{n,*}$ and a randomly chosen row to perform updates for. This ensures that each row is updated once per batch execution on average. The network's inputs remain equal across batches and epochs. After each epoch, a test trial is performed. Different update periods P have been tried out. In order to be in line with the single synapse experiment and to ensure enough correlation accumulation, the period is chosen as $P = 100\,\mathrm{ms_{bio}}$. Input neurons are firing Poisson-distributed with $T_\mathrm{isi} = 20\,\mathrm{ms_{bio}}$. The input weights are initialized from a normal distribution $\mathcal{N}(\mu = 0, \sigma = 20)$, and the recurrent and output weights are sampled from $\mathcal{N}(\mu = 0,\sigma = 1)$. The correlation sensors are parameterized as in Section 6.4.2, and neurons are configured as in Section 5.4.

For demonstration purposes, two simple target patterns are chosen. The first target is simply a sinus with a period $2T$; the second pattern has the same period but is additionally shifted by T (see Figure 6.15).

6.5.2 Result

While many different settings and parameters have been tested out, a sweet spot in which learning is successful is not yet found. The so far achieved results are depicted in Figure 6.15. The two different patterns are labeled as "Positive" (first pattern) and "Negative" (second pattern), respectively.

It can be observed that for both patterns, the loss drops until around epoch 30, after which it stabilizes at an MSE of about 0.2 and does not decrease any further. However, since the average firing rate decreases slightly until epoch 200, the network still adjusts its weights even if the loss does not improve anymore. This mainly results from an increasing number of neurons becoming silent. The two example trials show that the network can learn the sign of the target patterns but cannot capture the target's actual shape.

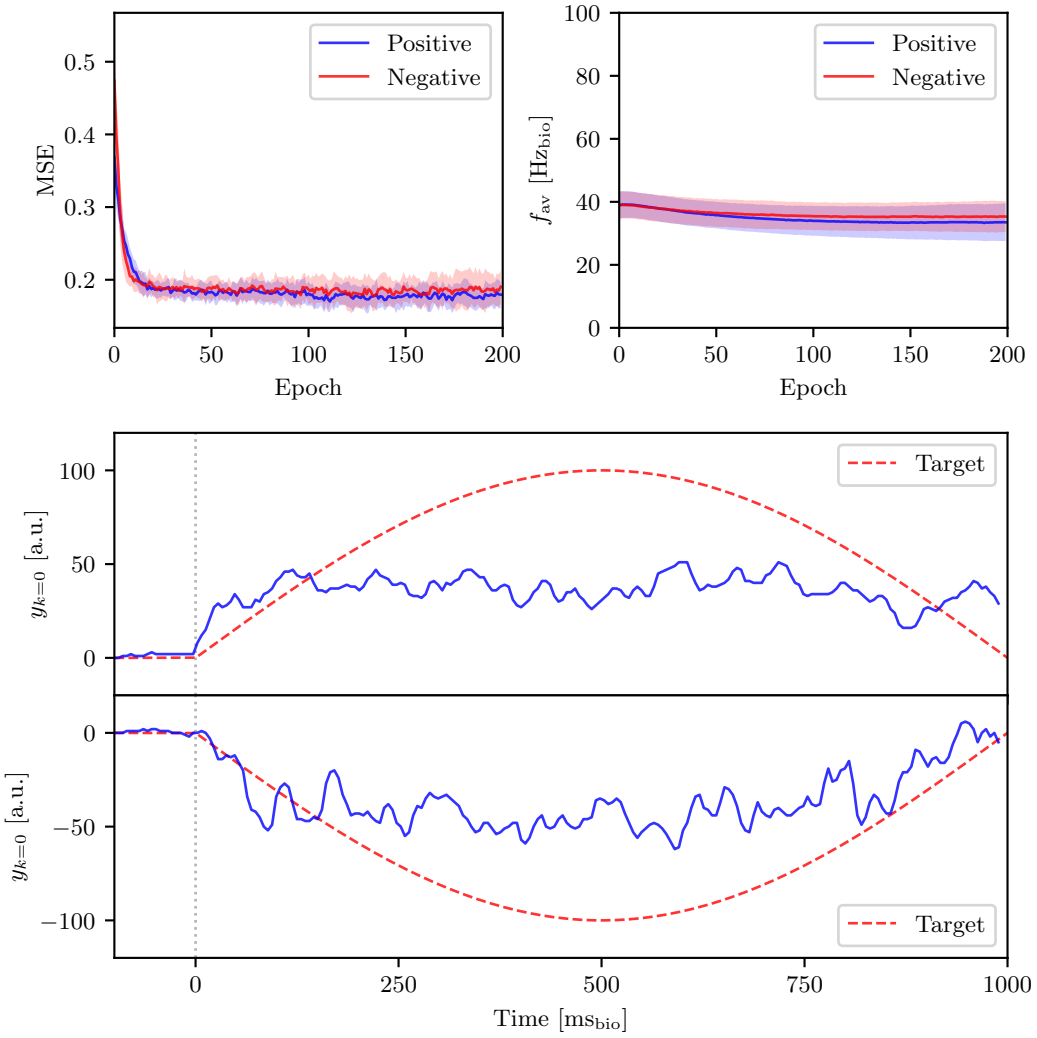


Figure 6.15: Example of full on-chip learning for two different target patterns. The label "Positive" corresponds to the first target pattern in the lower plot and "Negative" to the second pattern. The confidence bands are the standard deviation over multiple training runs. The **upper-left** plot depicts the evolution of the MSE over epochs. The loss converges fast to a rather constant value, indicating that the network does not improve its performance anymore. The **upper-right** plot shows the average firing rates over epochs. For both patterns, the network's neurons decrease their activity slightly. In the **lower** the "learned" example traces in comparison to their target pattern are exemplified. While the networks are able to adjust the readout neurons membrane trace into the right direction, the curvature of the pattern is not captured. Parameters are given in Table A.19.

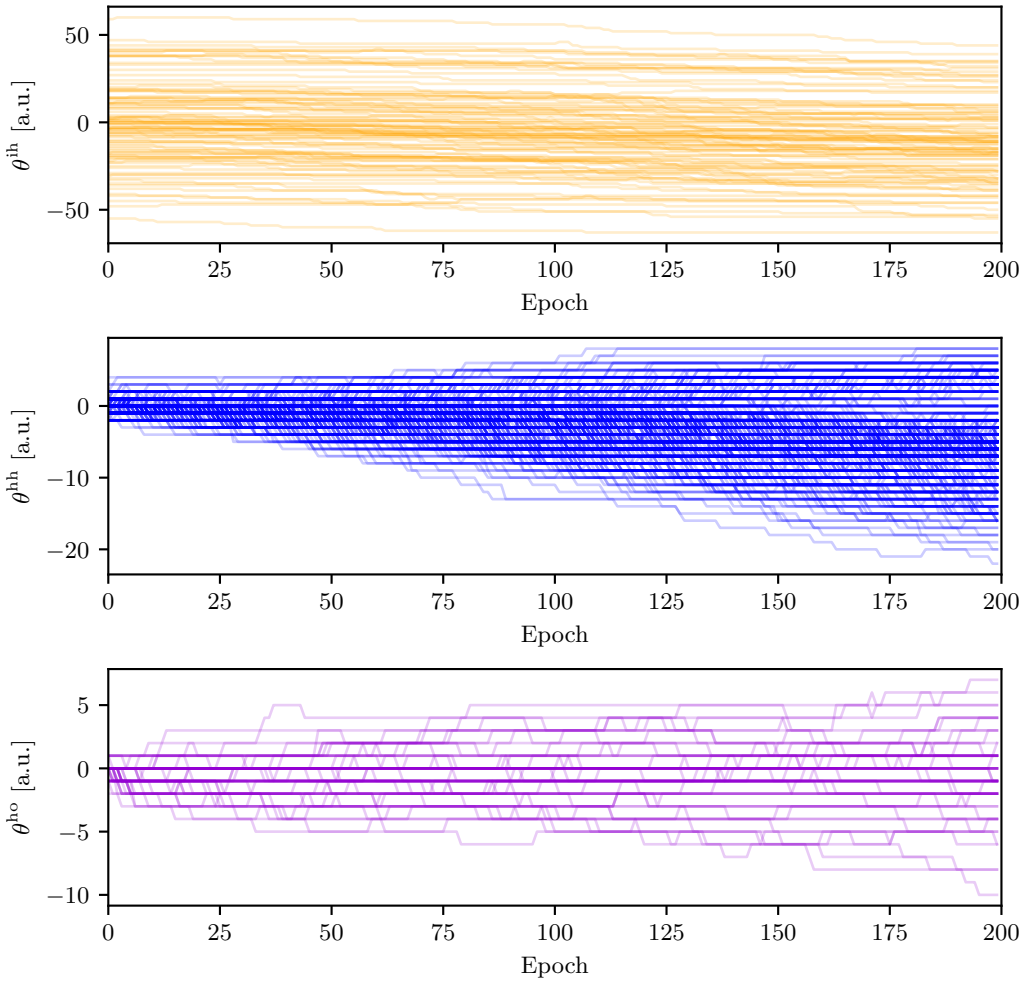


Figure 6.16: Weight evolution in a single training process. The **upper** plot shows the input weights θ^{ih}, which are initialized broadly and develop towards smaller values. Some recurrent weights θ^{hh} in the **middle** plot increase with the epochs; however, most weights tend towards negative values. In the **lower** plot, the development of the output weights θ^{ho} is shown. In contrary to the recurrent weights, these weights learn a narrow distribution, approximately centered around zero. Parameters are given in Table A.19.

In Figure 6.16, the evolution of the network's weights over epochs is exemplified. It is observed that the weights θ^{ih} of the input projection develop towards smaller values over epochs. However, in some epochs, some weights are getting increased occasionally. The same phenomenon can be seen for the recurrent weights θ^{hh}. While some weights become larger, the vast majority of the weights are getting decreased. The tendency towards smaller weights in the input and recurrent projection is also the reason for the decreasing firing rates in Figure 6.15 since synaptic input becomes more and more inhibitory. In contrast, the output weights θ^{ho} learn a rather centered but narrow distribution. The behavior of the output weights seems plausible since similar observations are made in simulations (see Figure 5.7 for s-prop), which suggests that the "learned" sign of the

traces in Figure 6.15 is only due to the output weights, without contributions of recurrent and input weight adjustments. So far, this observation of falling weights is not fully understood; however, possible reasons are discussed in the next section.

6.5.3 Possible Issues

Due to absent learning success, possible issues are discussed, and strategies to overcome them are proposed.

Vanishing Updates due to Fractional Arithmetic

The most difficult part of implementing the update rule on the PPU is to exploit the available vector arithmetic properly since the VU operates on vectors of 128 bytes. Therefore, the weight updates are preferably calculated with 8-bit values. For the input and recurrent weights, this is challenging since the plasticity rule in Equation (6.13) is essentially the product of three factors: The correlation readout, the signed error, and the output weights. Multiplying these factors results most certainly in overflow. This is especially a problem when working with signed bytes because overflow might change the update's sign. For this reason, fractional arithmetic is used. While this does prevent overflow, another issue arises. Multiplying two fractional numbers, a and b, always gives a number $c = a \cdot b \leq \min(a,b)$, such that for a three-factor learning rule, vanishing weight updates are likely. Hence, the range in which the update rule performs well is small; Correlation readouts, counter readouts, and learning signals need to be scaled appropriately.

A first attempt to address this issue is to implement the rule with a 16-bit resolution. This is possible since the VU allows packing 8-bit vectors into two 16-bit vectors, on which the VU is also able to operate. This doubles the number of operations the PPU has to perform and, thus, increases the minimal P. Nonetheless, a 16-bit implementation is already under development; unfortunately, it is not fully finished yet.

The preferred way to find optimal operating ranges and appropriate scales for the factors in the learning rule is to incorporate PPU-like arithmetic into the simulations. This is already achieved by using the Python library `fxpmath` [Franco, 2021], which allows performing arbitrary fixed-point fractional arithmetic. Since this was implemented as the thesis approaches its end, results are not present yet. Implementing fractional arithmetic in Python comes at the cost of heavily increasing simulation times. However, first attempts have shown that 8-bit fractional arithmetic can indeed make training difficult when factors are not scaled appropriately.

Regularization

The observed behavior that neurons tend to become silent as weights become more inhibitory with epochs might be impacted positively by applying firing rate regularization updates to the weights. In that way, a tilting learning dynamic can be prevented by

pulling neurons towards a target event frequency. Hence, an important step is to implement on-chip firing rate regularization. As mentioned in Section 6.2.1, this can be achieved by accessing the spike counters with the scalar-unit.

Weight Updates and Bugs

Admittedly, the effect of falling weights in Figure 6.16 appears to have structure. Even after extensive debugging of the PPU plasticity rule implementation, and its verification within the scope of the single synapse experiment, it might be possible that bugs remained undiscovered. On the one hand, that concerns the implementation on the PPU side, where, for instance, the subtle effects of 8-bit vector operations are biasing the weight updates into a certain direction due to limited resolution. And on the other hand, the software for the full network setup on the host side might not behave as expected (see Section 4.2.4). This, for instance, could (even though highly unlikely) comprise a wrongly communicated network topology from the host to the PPU. In order to exclude bugs, more tests and artificial learning setups should be created, where each addresses simple scenarios with predictive behavior. A first step is done with the single synapse experiment in Section 6.4.

Correlation Measurements

Since the learning rule for the input and recurrent projections depends on the correlation readout, the quality of the correlation curves in the current chip version might disturb learning. For some synapses, the correlation measurement does not represent (even if noise is neglected) the actual spike information in a given update interval n. As a consequence, this can cause a total weight update $\Delta\theta_{ij}$ in the wrong direction. This is because, if — for instance — the learning signal $\sum_k \theta^{\text{ho}}_{kj} \left(y^{*,n}_k - y^n_k\right)$ in an update interval n is weighted more strongly by the correlation readout than it should and in a later interval n' the signed error flips its sign and the learning signal is weighted less than expected, then the weight update $\Delta\theta_{ij} = \sum_n \Delta\theta^n_{ij}$ changes not just its absolute value, but also its sign relative to the "true" weight update.

A first step to investigate the effect of non-representative correlation measurements is to perform NASProp in-the-loop training by incorporating actual correlation readouts. In this way, other difficulties for on-chip learning, such as vanishing updates due to fractional arithmetic, and update computations with limited 8-bit precision, can be excluded and weight updates can be calculated as in simulation by solely replacing the simulated correlation with the correlation measurements. This allows verifying that learning can work with analog correlation readouts. Therefore, for each update interval n in a trial, the correlation is written into the external memory and accessed to calculate the weight update on the host side. However, this only allows for row-wise updates, since correlation measurements are still time extensive and can only be measured for a single row in a trial (see Section 6.3). A possible workaround is to perform the same forward pass multiple times (without weight updates in between), and measure correlation for all used hardware rows successively with each trial. In that way, the obtained correlation measurements of

all rows can then be used to compute updates for *all* weights at once. The updates are then applied from the hosted side as explained in Section 4.2.1.

7 Epilogue

This thesis approaches the challenge of realizing a biologically inspired learning algorithm for recurrent spiking neural networks on the analog neuromorphic *BrainScaleS-2* (BSS-2) system. This complex task is structured into three parts building upon each other: Software development, *in-the-loop* training with a spike-based learning rule, and finally, implementing an e-prop-inspired [Bellec et al., 2019] plasticity rule *on-chip*. For each of these elements, the results are summarized and briefly discussed in the following.

To facilitate learning on *HICANN-X v2* (HX), the high-level Python experiment framework `EProp` is developed. This framework allows describing abstract networks of spiking neurons within the `PyTorch` infrastructure by defining populations of neurons and projections between them. Weights in a projection are adjusted by *online* learning rules of arbitrary form. Here, the e-prop, as well as the s-prop and the NASProp plasticity rules, are implemented, enabling learning in RSNNs in simulation and on the actual hardware. Therefore, the main task is to integrate the analog chip into the framework such that experiments can be mapped to HX and transparently executed as in software simulations. This is achieved in two different ways.

Firstly, for in-the-loop training, the forward pass is done on HX, and the weights are optimized on the host. For this purpose, a `recurrent_to_readout` layer is developed within the `PyTorch` extension `hxtorch`. This layer is exposed to Python, operates on `PyTorch` data types, and enables seamless emulation of the network on HX by utilizing the `grenade` interface. In order to map the abstract network topology to a representation on hardware, a routing algorithm is developed, supporting signed synapses and recurrence. This introduces support for recurrent spiking neural networks in `hxtorch`.

Secondly, for on-chip learning, a `PyTorch` module is contributed to `hxtorch`, which interfaces HX and performs all software-to-hardware mapping implicitly while allowing monitoring training within `PyTorch`'s ecosystem. Finally, the classes' forward method triggers a network emulation run including on-chip weight updates.

The experiments conducted in this thesis demonstrate the success of abstracting learning on HX with the developed software in high-level software layers and underpin its importance.

The first step towards e-prop-inspired learning on hardware is achieved with HX in-the-loop and an approximated learning rule. The membrane potentials of the recurrent neurons in the eligibility traces are replaced with the neurons' post-synaptic spike trains. This allows calculating approximated weight updates with knowledge of spike times and the readout neuron's membrane trace only, and, hence, is called *spike-based eligibility propagation* (s-prop). On the one hand, this facilitates an initial implementation on HX since the spikes emitted during emulation of the network in analog can be read out easily

together with the readout neuron's membrane trace from the host computer. On the other hand, this approximation allows utilizing the correlation sensors for full on-chip learning.

The s-prop learning rule is used to train a network consisting of a single recurrent spiking layer and a readout neuron to solve a pattern-generation task. This task is chosen due to its simplicity, making it a suitable choice to approach on-chip learning while satisfying the hardware constraints defined by the system architecture.

A baseline simulation shows that s-prop can solve this task, in principle, surprisingly well. Incorporating basic hardware constraints, like noise and discrete weights, yields an increasing MSE and dropping network activity. It is observed that the output weights learn a narrow distribution, which is insufficiently resolved by discrete integer weights. Small output weights are assumed to be a result of the high-activity regime in which the network operates. These weights are enlarged by decreasing the synaptic strength to the readout neuron and scaling up the output learning rate. This ensures that the membrane trace of the readout neuron does not exhaust its physical boundaries while training on HX as possible when up-scaling the target's amplitude to achieve the same effect. Simulations show that this technique improves learning when using discrete weights.

Further, the effect of fixed-pattern noise on the network's parameters and Gaussian noise on the membranes seems negligible. However, experiments on HX perform almost equally well as in simulation. This suggests a good signal-to-noise ratio and neurons being very responsive to synaptic input, such that deviations on the neurons' membranes due to noise do not disturb learning significantly. This originates from the broad weight initialization of the input projection, ensuring a reasonable spiking activity in the network, which is necessary for spike-based learning. It is assumed that the effect of fixed-pattern noise is mostly absorbed by gradient-based learning.

Finally, the learning procedure is mapped directly to HX. This seamless transition is enabled by the hardware abstraction in software. With in-the-loop training on HX, an MSE of about $3.58 \cdot 10^{-3}$ is achieved for the pattern-generation task. This is just slightly worse than the simulation with hardware constraints, which reaches an MSE of $2.68 \cdot 10^{-3}$, and thus confirms the feasibility of the pattern-generation task for on-chip learning and demonstrates learning in RSNNs on neuromorphic hardware by describing experiments in high-level software. In comparison to the simulation, the network has a sparser activity on hardware. This is due to experiment setup and suggests that good results can still be obtained when the network operates at lower firing rates.

In order to investigate whether s-prop does indeed enable learning recurrence, the hardware experiment is repeated without recurrent connections. Additionally, a further experiment with a fixed recurrent connection pattern, initialized by a scrambled learned recurrent weight matrix, is conducted to ensure recurrent spike-information exchange along the sequence. Section 5.4.3 shows that both experiments result in worse performance than with learning recurrence, indicating that s-prop endows RSNNs with the ability to exploit recurrence in a sensible manner. Nonetheless, a significant amount of the task can be learned without recurrence. The network does not rely heavily on it, suggesting that information is not propagated along the whole sequence but only over short time periods to refine the network. Besides the network's complexity, this is due to the nature of LIF neurons whose information content encoded in the membranes dis-

integrates with their time constants. However, especially these properties make the task and the network's setup an excellent choice to approach on-chip learning; While recurrence clearly improves learning, the weak dependency makes training less challenging and, thus, facilitates the first step towards a full on-chip implementation.

For on-chip learning, an adjusted learning rule is derived based on the e-prop learning framework. The learning rule for the recurrent and input projection uses the correlation sensors to emulate eligibility traces on HX in analog. Besides the s-prop assumption, this requires two additional approximations: Eligibility traces are modeled in a nearest-neighbor fashion and get accumulated over a time period P. This eliminates the dependence of the s-prop rule on exact spike times and can, therefore, efficiently be implemented on the PPU. For the output weight updates, this is achieved by utilizing the spike counters. Since these learning rules propagate accumulated spike information over time, they are referred to as *Neuromorphic Accumulative Spike Propagation* (NASProp).

NASProp is simulated in software. A baseline simulation in Figure 6.5 shows that NASProp can solve the pattern-generation task properly with $P = 25\,\mathrm{ms_{bio}}$. Increasing the accumulation period to $P = 50\,\mathrm{ms_{bio}}$ increases the MSE from $1.54 \cdot 10^{-3}$ to $3.15 \cdot 10^{-3}$. Since the PPU implementation defines a minimal P by the number of required operations to compute an online update, the dependence of the network's performance on P is investigated in Figure 6.6. Here, simulations consider hardware constraints and perform weight updates in an SGD-manner since weights cannot be optimized more sophisticated on the PPU. For weight optimization with Adam, the loss increases only slightly with P. Using SGD weight optimization causes the MSE to increases faster; however, it yields good results with small update periods. As for s-prop, discrete weights decrease performance further. Nonetheless examples show, that even with $P = 100\,\mathrm{ms_{bio}}$, acceptable results can be achieved and justify an on-chip implementation.

In simulations, the network's performance is compared to a network without recurrence, also trained with NASProp. Especially when updating weights with the Adam optimizer, recurrence increases performance significantly. The same holds for simulations in the hardware environment. Therefore, NASProp is capable of adjusting recurrent connections in a meaningful manner.

Finally, the NASProp learning rules are implemented in a PPU program, able to perform on-chip plasticity. This allows for a considerably faster learning process than in-the-loop training since (almost) no data is transferred between host and HX and, thus, makes training energy efficient. The learning rule on the PPU is mainly implemented in assembler code by utilizing the VU's instruction set and, hence, is able to perform vectorized weight updates based on 8-bit fractional arithmetic. To save computation time, in each trial, the PPU computes updates for a single randomly chosen hardware row in parallel to the forward pass by utilizing correlation measurements. A single weight row update along the sequence requires only about $53\,\mathrm{ms_{bio}} = 53\mu s_{\mathrm{hw}}$ and defines the minimal achievable update period P. Time-critical measurements, i.e., correlation, counter and membrane readout, and resets, are achieved within $7\,\mathrm{ms_{bio}}$.

On-chip implementations are prone to errors. In order to test the implementation, a single synapse experiment is set up in a spiking environment. Experiments confirm the desired weight dynamic over epochs. The recurrent weight is observed to be adjusted

slower with decreasing correlation measurements into the direction of the learning signal; the output weights follow the direction of the error's sign and updates become smaller as the error decreases. Correlations sensors are analog circuits and, thus, subject to variation. Experiments in Figure 6.14 show that this causes different synapse-specific learning rates. In this artificial learning setup, the speed of weight adaption ranges from 0.769 a.u. per epoch to 4.063 a.u. per epoch between different synapses. The author discussed and worked on the correlation sensor noise with hardware experts. However, a satisfying result was not reached in time with the end of this work.

In the final step, the PPU implementation is used to train the full network on-chip. The network "learns" to adjust the readout neuron's membrane potential in the correct direction; however, the curvature of the pattern cannot be captured (see Figure 6.15). As result, the MSE drops rather fast until it maintains a constant value and performance does not increase further. It is assumed that the implementation is able to only adjust the output weights properly. Input and recurrent weights are observed to evolve over epochs but drift towards inhibitory weights. Reasons for this behavior are discussed, and possibilities to overcome them proposed. One critical point is the 8-bit fractional multiplications when computing weight updates; this results easily in vanishing updates. Appropriate scaling of factors can be found by incorporating PPU-like vector operations into the simulations. Another possibility is the noisiness of correlation measurements which might bias updates. This could be investigated by performing NASProp in-the-loop training with actual correlation readouts. Lastly, an implementation bug cannot be excluded, and is assumed to be hidden somewhere. As debugging PPU-programs is challenging, more test and predictable experiments are needed.

Although the on-chip implementation has not reached its working state yet, overcoming discussed issues will most certainly enable full on-chip learning. Therefore, this thesis has taken a crucial first step towards biologically inspired *full* on-chip learning on analog neuromorphic hardware.

7.1 Outlook

This thesis should be considered a first investigation of the feasibility of e-prop-inspired learning on HX. Therefore, not all aspects of this broad task could have been taken into account. Crucial elements of future work are briefly outlined in the following.

The software developed in this thesis has the potential to be reused and extended. Especially, the routing algorithm with signed synapses is appealing for plenty of applications. Such functionality should be provided by `grenade` per default. Since, at the moment, only projections with a specific receptor type are available (i.e., excitatory or inhibitory), this requires the development of projections with a signed receptor type (*cf.* Section 4.2). Additionally, the edge cases of the current implementation need to be worked out, and routing on both hemispheres of HX should be enabled.

The developed `hxtorch` layer allows easy usage for non-expert users and promises to

facilitate in-the-loop training for many applications. However, the current implementation only allows recording the membrane of a single neuron. Since many applications require measuring membrane potentials of more than one neuron, this layer could be extended with the capability to sample all neurons' membranes via the CADC in parallel. This would open up the possibility for seamless training on HX with plenty of different learning rules, such as SuperSpike [Zenke et al., 2018; Kanya, 2020] or BPTT with surrogate gradients [Cramer et al., 2020], within `PyTorch`'s ecosystem. In principle, this can also enable solving the pattern-generation task for more than one readout neuron with HX in the loop (an initial version of the CADC readout in the `grenade` layer is already in progress).

The s-prop learning rule is a promising candidate for a broad range of different applications on HX. In order to improve the pattern-generation task and to exploit the recurrent nature of the network even further, its parameters should be investigated in more depth. In particular, this encompasses the number of input neurons and their firing frequency, but also the network's topology in general. Additionally, the LIF parameters of the neurons on HX, such as refractory periods, membrane time constants, and thresholds, should be tuned appropriately. Certainly, it would be interesting to see how the network can handle the analog environment as its complexity decreases. Eventually, s-prop can be benchmarked for the pattern-generation task with parameters as proposed by Bellec et al. [2019].

LIF neurons are not the preferred choice for tasks that demand propagation of information over longer time periods since their temporal extent is limited by the membrane time constant. Instead, *Long short-term memory spiking neural network*s (LSNNs) use, therefore, LIF neurons with threshold adaptation [Bellec et al., 2019], where the threshold increases with each post-synaptic spike, after which it leaks back towards a baseline-threshold with an adaptation time constant that is usually much longer than the membrane time constant. Hence, the information encoded in the threshold is propagated over longer time periods. For such neurons, e-prop learning rules are derived in Bellec et al. [2019]. However, HX emulates *adaptive exponential integrate-and-fire* (AdEx) neurons with an adaptation current, where each post-synaptic spike increases a current flowing off the membrane, after which the current decays back towards a baseline. Hence, they can, principally, also encode information in their adaptation variable and, thus, propagate information along the sequence. In order to tackle more exciting tasks, learning rules need to be derived for AdEx neurons within the e-prop learning framework and the feasibility of their implementation on HX— by incorporating the s-prop assumption — investigated. This would allow solving advanced classification tasks and could also enable performing biologically inspired reinforcement learning with HX in-the-loop.

Before considering on-chip learning in LSNNs, the step towards a working on-chip implementation of NASProp done in this thesis needs to be completed. Different possibilities to achieve this have already been discussed (see Section 6.5.3). Only a working and optimized implementation will allow investigating further properties such as speed and energy consumption, both significant aspects of neuromorphic computing.

Regarding the current chip version, solving the issues with the correlation sensors to gain more realistic eligibility vectors is encouraged and assumed to ease learning. Fur-

ther, providing vector-unit access to the spike counters would facilitate incorporating firing rate regularization into the learning rule. Nonetheless, this can, technically, also be achieved with slow scalar-unit spike counter readouts (see Section 6.2.1). In the long term, implementing regularization updates is essential as this stabilizes the learning dynamics.

As it has been seen that more sophisticated weight optimization algorithms improve learning dramatically, future works have to examine the possibility of implementing advanced optimizers on the PPU. Even though no effort has been put into this, it is well imaginable that the external memory could be used to store momentum matrices that can be accessed to compute weight updates. As accessing the external memory from the PPU is slow, this can cause issues when weight updates need to be computed with low latency. However, if weight updates are not time-critical this can be a promising approach.

Thinking into the future, the e-prop idea of synapse-specific eligibility traces combined with externally provided learning signals is generally appealing for multi-chip setups. Therefore, local information, processed in analog by the correlation sensors, is merged with appropriately designed learning signals — calculated by the different chips on the setup — to compute weight updates. While this comes with the difficulty of designing suitable learning signals, it would undoubtedly be a big step towards learning in functional large-scale spiking neural networks and could challenge machine learning performances with artificial neural networks in plenty of different tasks. Due to accelerated neuromorphic hardware this would also come with the benefit of high energy-efficiency and low processing latency.

These outlined possibilities are only a subset of further research on e-prop inspired learning in RSNNs on neuromorphic hardware. Therefore, the work in this thesis unfolds a broad field of applications waiting to be explored.

Acknowledgments

This thesis has been carried out in a tough year in difficult times and would have never been possible without the contribution of many people on different levels, to whom I want to express my deepest gratitude.

First and foremost, I would like to thank Dr. Johannes Schemmel for giving me the opportunity to be part of the Electronic Vision(s) group and agreeing to be my supervisor.

I extend my thanks to the whole Electronic Vision(s) group for the support and great atmosphere. I am looking forward to finally meet you in person.

I am incredibly grateful to Dr. Eric Müller for being my advisor and guide throughout this thesis. His valuable knowledge, effort, and support have majorly shaped this work.

Further, I thank Philipp Spilger and Oliver Breitwieser for the help in plenty software-related issues and Johannes Weis for patiently answering uncountable hardware-related questions. I really appreciate all your effort. In addition, I thank Sebastian Billaudelle and Benjamin Cramer for all the modeling discussions, information, and advice.

I am deeply grateful for all my brothers and sisters who remind me that there is more in life. Each of you is truly special. Without the support of my parents Magdalene and Jörg Arnold, this thesis would not have been possible. They encouraged my throughout my studies and beyond. You are my inspiration. Finally, I want to express my deepest gratitude to Malin, who gave me so much strength and has been incredibly patient.

The work carried out in this Master's Thesis used systems, which received funding from the European Union's Horizon 2020 Framework Programme for Research and Innovation under the Specific Grant Agreements Nos. 785907 and 945539 (Human Brain Project, HBP)

Bibliography

Akopyan, F., J. Sawada, A. Cassidy, R. Alvarez-Icaza, J. Arthur, P. Merolla, N. Imam, Y. Nakamura, P. Datta, G.-J. Nam, B. Taba, M. Beakes, B. Brezzo, J. B. Kuang, R. Manohar, W. P. Risk, B. Jackson, and D. S. Modha (2015). TrueNorth: Design and Tool Flow of a 65 mW 1 Million Neuron Programmable Neurosynaptic Chip. In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 34.10, pp. 1537–1557. DOI: 10.1109/TCAD.2015.2474396.

Alberts, B., D. Bray, J. Lewis, M. Raff, K. Roberts, and J. D. Watsonx (1994). Molecular Biology of the Cell, third edition. Garland Publishing, Inc. Chap. 10 and 11. ISBN: 0815316208.

Bellec, G., F. Scherr, E. Hajek, D. Salaj, R. Legenstein, and W. Maass (2019). Biologically inspired alternatives to backpropagation through time for learning in recurrent neural nets. arXiv: 1901.09049 [cs.NE].

Bellec, G., F. Scherr, A. Subramoney, E. Hajek, D. Salaj, R. Legenstein, and W. Maass (2020). A solution to the learning dilemma for recurrent networks of spiking neurons. In: *Nature Communications* 11.1, p. 3625. DOI: 10.1038/s41467-020-17236-y. URL: https://doi.org/10.1038/s41467-020-17236-y.

Benjamin, B., P. Gao, E. McQuinn, S. Choudhary, A. Chandrasekaran, J.-M. Bussat, R. Alvarez-Icaza, J. Arthur, P. Merolla, and K. Boahen (May 2014). Neurogrid: A Mixed-Analog-Digital Multichip System for Large-Scale Neural Simulations. In: *Proceedings of the IEEE* 102, pp. 1–18. DOI: 10.1109/JPROC.2014.2313565.

Billaudelle, S., B. Cramer, M. A. Petrovici, K. Schreiber, D. Kappel, J. Schemmel, and K. Meier (2021). Structural plasticity on an accelerated analog neuromorphic hardware system. In: *Neural Networks* 133. 0893-6080, pp. 11–20. DOI: https://doi.org/10.1016/j.neunet.2020.09.024. URL: http://www.sciencedirect.com/science/article/pii/S0893608020303555.

Billaudelle, S., Y. Stradmann, K. Schreiber, B. Cramer, A. Baumbach, D. Dold, J. Göltz, A. F. Kungl, T. C. Wunderlich, A. Hartel, et al. (2020). Versatile emulation of spiking neural networks on an accelerated neuromorphic substrate. In: *2020 IEEE International Symposium on Circuits and Systems (ISCAS)*. IEEE, pp. 1–5.

Breitwieser, O. J. (2015). Towards a Neuromorphic Implementation of Spike-Based Expectation Maximization. MA thesis. University of Heidelberg, pp. 61–66.

Brette, R. and W. Gerstner (2005). Adaptive Exponential Integrate-and-Fire Model as an Effective Description of Neuronal Activity. In: *J. Neurophysiol.* 94, pp. 3637–3642. DOI: 10.1152/jn.00686.2005.

Buzsáki, G. (Jan. 2009). Rhythms of The Brain. In: xiv, 448 p. ISBN: 9780195301069 (alk. paper) 0195301064 (alk. paper). DOI: 10.1093/acprof:oso/9780195301069.001.0001.

Buzzell, G., J. Richards, L. White, T. Barker, D. Pine, and N. Fox (May 2017). Development of the error-monitoring system from ages 9–35: Unique insight provided by MRI-constrained source localization of EEG. In: *NeuroImage* 157. DOI: 10.1016/j.neuroimage.2017.05.045.

Cai, Z. and N. Vasconcelos (June 2018). Cascade R-CNN: Delving Into High Quality Object Detection. In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*.

Cho, K., B. van Merrienboer, C. Gulcehre, D. Bahdanau, F. Bougares, H. Schwenk, and Y. Bengio (2014). Learning Phrase Representations using RNN Encoder-Decoder for Statistical Machine Translation. arXiv: 1406.1078 [cs.CL].

Cramer, B., S. Billaudelle, S. Kanya, A. Leibfried, A. Grübl, V. Karasenko, C. Pehle, K. Schreiber, Y. Stradmann, J. Weis, J. Schemmel, and F. Zenke (2020). Training spiking multi-layer networks with surrogate gradients on an analog neuromorphic substrate. In: *CoRR* abs/2006.07239. arXiv: 2006.07239. URL: https://arxiv.org/abs/2006.07239.

Czierlinski, M. (2020). PyNN for BrainScaleS-2. Bachelorarbeit. Universität Heidelberg.

Dauer, P. (2020). Characterization of silicon neurons on HICANN-X v2. Bachelorarbeit. Universität Heidelberg.

Davies, M., N. Srinivasa, T.-H. Lin, G. Chinya, Y. Cao, S. H. Choday, G. Dimou, P. Joshi, N. Imam, S. Jain, Y. Liao, C.-K. Lin, A. Lines, R. Liu, D. Mathaikutty, S. McCoy, A. Paul, J. Tse, G. Venkataramanan, Y.-H. Weng, A. Wild, Y. Yang, and H. Wang (2018). Loihi: A Neuromorphic Manycore Processor with On-Chip Learning. In: *IEEE Micro* 38.1, pp. 82–99. DOI: 10.1109/MM.2018.112130359.

Davison, A. P., D. Brüderle, J. Eppler, J. Kremkow, E. Muller, D. Pecevski, L. Perrinet, and P. Yger (2009). PyNN: a common interface for neuronal network simulators. In: *Front. Neuroinform.* 2.11. DOI: 3389/neuro.11.011.2008.

Emmel, A. (Nov. 2020). Inference with Convolutional Neural Networks on Analog Neuromorphic Hardware. Master's Thesis. Universität Heidelberg.

Friedmann, S., J. Schemmel, A. Grübl, A. Hartel, M. Hock, and K. Meier (2017). Demonstrating Hybrid Learning in a Flexible Neuromorphic Hardware System. In: *IEEE Transactions on Biomedical Circuits and Systems* 11, pp. 128–142. DOI: 10.1109/TBCAS.2016.2579164.

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

Friedmann, S. and C. Pehle (2020). Nux User Guide.

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

Gerstner, W., W. M. Kistler, R. Naud, and L. Paninski (2014). Neuronal Dynamics: From Single Neurons to Networks and Models of Cognition. USA: Cambridge University Press. ISBN: 1107635195.

Gerstner, W., M. Lehmann, V. Liakoni, D. Corneil, and J. Brea (2018). Eligibility Traces and Plasticity on Behavioral Time Scales: Experimental Support of NeoHebbian Three-Factor Learning Rules. In: *Frontiers in Neural Circuits* 12, p. 53. ISSN: 1662-5110. DOI: 10.3389/fncir.2018.00053. URL: https://www.frontiersin.org/article/10.3389/fncir.2018.00053.

Goodfellow, I., Y. Bengio, and A. Courville (2016). Deep Learning. The MIT Press. ISBN: 0262035618.

Hebb, D. (2005). The Organization of Behavior: A Neuropsychological Theory. Taylor & Francis. ISBN: 9781135631918. URL: https://books.google.de/books?id=uyV5AgAAQBAJ.

Hochreiter, S. and J. Schmidhuber (1997). Long short-term memory. In: *Neural computation* 9.8, pp. 1735–1780.

Human Brain Project (2021). https://www.humanbrainproject.eu/en/. Accessed: 2021-06-16.

Iberri, D. (2007). File:Action potential.svg. URL: https://upload.wikimedia.org/wikipedia/commons/4/4a/Action_potential.svg (visited on 05/17/2021).

Jarosz, Q. (2009). File:Neuron Hand-tuned.svg. URL: https://upload.wikimedia.org/wikipedia/commons/b/bc/Neuron_Hand-tuned.svg (visited on 05/17/2021).

Kandel, E., J. Jessell, J. Schwartz, T. Jessell, P. of Biochemistry, M. Molecular Biophysics Thomas M Jessell, S. Mack, and J. Dodd (2000). Principles of Neural Science, Fourth Edition. McGraw-Hill Companies, Incorporated. ISBN: 9780838577011.

Kanya, S. (2020). Deep Learning on Analog Neuromorphic Hardware. Masterarbeit. Universität Heidelberg.

Kelley, H. J. (1960). Gradient theory of optimal flight paths. In: *Ars Journal* 30.10, pp. 947–954.

Kingma, D. P. and J. Ba (2017). Adam: A Method for Stochastic Optimization. arXiv: 1412.6980 [cs.LG].

Kurtzer, G. M., V. Sochat, and M. W. Bauer (May 2017). Singularity: Scientific containers for mobility of compute. In: *PLOS ONE* 12.5, pp. 1–20. DOI: 10.1371/journal.pone.0177459. URL: https://doi.org/10.1371/journal.pone.0177459.

Lapicque, L. (1907). Recherches Quantitatives sur l'Excitation Electrique des Nerfs Traitée comme une Polarization. In: *Journal de Physiologie et Pathologie General* 9, pp. 620–635.

LeCun, Y., Y. Bengio, and G. Hinton (2015). Deep learning. In: *Nature* 521.7553, pp. 436–444. DOI: 10.1038/nature14539. URL: https://doi.org/10.1038/nature14539.

MacLean, S., C. Hassall, Y. Ishigami, O. Krigolson, and G. Eskes (2015). Using brain potentials to understand prism adaptation: the error-related negativity and the P300. In: *Frontiers in Human Neuroscience* 9.

Mead, C. (1990). Neuromorphic electronic systems. In: *Proceedings of the IEEE* 78.10, pp. 1629–1636. DOI: 10.1109/5.58356.

Moradi, S., N. Qiao, F. Stefanini, and G. Indiveri (2018). A Scalable Multicore Architecture With Heterogeneous Memory Structures for Dynamic Neuromorphic Asynchronous Processors (DYNAPs). In: *IEEE Transactions on Biomedical Circuits and Systems* 12.1, pp. 106–122. DOI: 10.1109/TBCAS.2017.2759700.

Müller, E. C. (2014). Novel Operation Modes of Accelerated Neuromorphic Hardware. HD-KIP 14-98. PhD thesis. Ruprecht-Karls-Universität Heidelberg. URL: http://www.kip.uni-heidelberg.de/Veroeffentlichungen/details.php?id=3112.

Müller, E., C. Mauch, P. Spilger, O. J. Breitwieser, J. Klähn, D. Stöckel, T. Wunderlich, and J. Schemmel (Mar. 2020a). Extending BrainScaleS OS for BrainScaleS-2. In: *arXiv preprint*. arXiv: 2003.13750 [cs.NE]. URL: http://arxiv.org/abs/2003.13750.

Müller, E., S. Schmitt, C. Mauch, H. Schmidt, J. Montes, J. Ilmberger, J. Klähn, F. Passenberg, C. Koke, M. Kleider, S. Jeltsch, M. Güttler, D. Husmann, S. Billaudelle, P. Müller, A. Grübl, J. Kaiser, J. Weidner, B. Vogginger, J. Partzsch, C. Mayr, and J. Schemmel (Mar. 2020b). The Operating System of the Neuromorphic BrainScaleS-1 System. In: *arXiv preprint*. arXiv: 2003.13749 [cs.NE]. URL: http://arxiv.org/abs/2003.13749.

Nair, V. and G. E. Hinton (2010). Rectified Linear Units Improve Restricted Boltzmann Machines. In: *ICML*. Ed. by J. Fürnkranz and T. Joachims. Omnipress, pp. 807–814. URL: http://dblp.uni-trier.de/db/conf/icml/icml2010.html#NairH10.

Naud, R., N. Marcille, C. Clopath, and W. Gerstner (Nov. 2008). Firing patterns in the adaptive exponential integrate-and-fire model. In: *Biological Cybernetics* 99.4, pp. 335–347. DOI: 10.1007/s00422-008-0264-7. URL: http://dx.doi.org/10.1007/s00422-008-0264-7.

Nøkland, A. (2016). Direct Feedback Alignment Provides Learning in Deep Neural Networks. arXiv: 1609.01596 [stat.ML].

Painkras, E., L. A. Plana, J. Garside, S. Temple, F. Galluppi, C. Patterson, D. R. Lester, A. D. Brown, and S. B. Furber (2013). SpiNNaker: A 1-W 18-Core System-on-Chip for Massively-Parallel Neural Network Simulation. In: *IEEE Journal of Solid-State Circuits* 48.8, pp. 1943–1953. DOI: 10.1109/JSSC.2013.2259038.

Paszke, A., S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Kopf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala (2019). PyTorch: An Imperative Style, High-Performance Deep Learning Library. In: *Advances in Neural Information Processing Systems 32*. Ed. by H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett. Curran Associates, Inc., pp. 8024–8035. URL: http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf.

Petrovici, M. (2015). Function vs. Substrate: Theory and Models for Neuromorphic Hardware. PhD thesis.

Petrovici, M. A. (2015). Form vs. Function - Theory and Models for Neuronal Substrates. PhD thesis. Universität Heidelberg.

Pfeiffer, M. and T. Pfeil (2018). Deep Learning With Spiking Neurons: Opportunities and Challenges. In: *Frontiers in Neuroscience* 12, p. 774. ISSN: 1662-453X. DOI: 10.3389/fnins.2018.00774. URL: https://www.frontiersin.org/article/10.3389/fnins.2018.00774.

PowerISA (July 2010). PowerISA Version 2.06 Revision B. Specification. Power.org. URL: http://www.power.org/resources/reading/.

Rosenblatt, F. (1958). The perceptron: A probabilistic model for information storage and organization in the brain. In: *Psychological Review* 65.6, pp. 386–408. ISSN: 0033-295X. DOI: 10.1037/h0042519. URL: http://dx.doi.org/10.1037/h0042519.

Samadi, A., T. Lillicrap, and D. Tweed (Jan. 2017). Deep Learning with Dynamic Spiking Neurons and Fixed Feedback Weights. In: *Neural Computation* 29, pp. 1–25. DOI: 10.1162/NECO_a_00929.

Schemmel, J., S. Billaudelle, P. Dauer, and J. Weis (2020). Accelerated Analog Neuromorphic Computing. In: *arXiv* 2003.11996. cs.NE. URL: https://arxiv.org/abs/2003.11996.

Schemmel, J., D. Brüderle, A. Grübl, M. Hock, K. Meier, and S. Millner (2010). A wafer-scale neuromorphic hardware system for large-scale neural modeling. In: *2010 IEEE International Symposium on Circuits and Systems (ISCAS)*, pp. 1947–1950. DOI: 10.1109/ISCAS.2010.5536970.

Schmitt, S., J. Klähn, G. Bellec, A. Grübl, M. Güttler, A. Hartel, S. Hartmann, D. Husmann, K. Husmann, S. Jeltsch, V. Karasenko, M. Kleider, C. Koke, A. Kononov, C. Mauch, E. Müller, P. Müller, J. Partzsch, M. A. Petrovici, S. Schiefer, S. Scholze, V. Thanasoulis, B. Vogginger, R. Legenstein, W. Maass, C. Mayr, R. Schüffny, J.

Schemmel, and K. Meier (2017). Neuromorphic hardware in the loop: Training a deep spiking network on the BrainScaleS wafer-scale system. In: *2017 International Joint Conference on Neural Networks (IJCNN)*, pp. 2227–2234. DOI: 10.1109/IJCNN.2017.7966125.

Schreiber, K. (2020). Closed-loop experiments on the BrainScaleS-2 architecture. In:

Schreiber, K. (Jan. 2021). Accelerated neuromorphic cybernetics. PhD thesis. Universität Heidelberg.

Silver, D., A. Huang, C. Maddison, A. Guez, L. Sifre, G. Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, S. Dieleman, D. Grewe, J. Nham, N. Kalchbrenner, I. Sutskever, T. Lillicrap, M. Leach, K. Kavukcuoglu, T. Graepel, and D. Hassabis (Jan. 2016). Mastering the game of Go with deep neural networks and tree search. In: *Nature* 529, pp. 484–489. DOI: 10.1038/nature16961.

Spilger, P. (2021). From Neural Network Descriptions to Neuromorphic Hardware — A Signal-Flow Graph Compiler Approach. Masterarbeit. Universität Heidelberg.

Spilger, P., E. Müller, A. Emmel, A. Leibfried, C. Mauch, C. Pehle, J. Weis, O. Breitwieser, S. Billaudelle, S. Schmitt, T. C. Wunderlich, Y. Stradmann, and J. Schemmel (2020). hxtorch: PyTorch for BrainScaleS-2 — Perceptrons on Analog Neuromorphic Hardware. In: *IoT Streams for Data-Driven Predictive Maintenance and IoT, Edge, and Mobile for Embedded Machine Learning*. Cham: Springer International Publishing, pp. 189–200. ISBN: 978-3-030-66770-2. DOI: 10.1007/978-3-030-66770-2_14.

Strubell, E., A. Ganesh, and A. McCallum (July 2019). Energy and Policy Considerations for Deep Learning in NLP. In: *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*. Florence, Italy: Association for Computational Linguistics, pp. 3645–3650. DOI: 10.18653/v1/P19-1355. URL: https://www.aclweb.org/anthology/P19-1355.

Weis, J. (Sept. 2020). Inference with Artificial Neural Networks on Neuromorphic Hardware. Master's thesis. Universität Heidelberg.

Weis, J., P. Spilger, S. Billaudelle, Y. Stradmann, A. Emmel, E. Müller, O. Breitwieser, A. Grübl, J. Ilmberger, V. Karasenko, M. Kleider, C. Mauch, K. Schreiber, and J. Schemmel (2020). Inference with Artificial Neural Networks on Analog Neuromorphic Hardware. In: *IoT Streams for Data-Driven Predictive Maintenance and IoT, Edge, and Mobile for Embedded Machine Learning*. Vol. 1325. Springer International Publishing, pp. 201–212.

Werbos, P. (Nov. 1990). Backpropagation through time: what it does and how to do it. In: *Proceedings of the IEEE* 78, pp. 1550–1560. DOI: 10.1109/5.58337.

Wunderlich, T., A. F. Kungl, E. Müller, A. Hartel, Y. Stradmann, S. A. Aamir, A. Grübl, A. Heimbrecht, K. Schreiber, D. Stöckel, C. Pehle, S. Billaudelle, G. Kiene, C. Mauch, J. Schemmel, K. Meier, and M. A. Petrovici (2019). Demonstrating Advantages of Neuromorphic Computation: A Pilot Study. In: *Frontiers in Neuroscience*

13, p. 260. ISSN: 1662-453X. DOI: 10.3389/fnins.2019.00260. URL: https://www.frontiersin.org/article/10.3389/fnins.2019.00260.

Xu, X., Y. Ding, S. X. Hu, M. Niemier, J. Cong, Y. Hu, and Y. Shi (2018). Scaling for edge inference of deep neural networks. In: *Nature Electronics* 1.4, pp. 216–222. DOI: 10.1038/s41928-018-0059-3. URL: https://doi.org/10.1038/s41928-018-0059-3.

Young, A. R., M. E. Dean, J. S. Plank, and G. S. Rose (2019). A Review of Spiking Neuromorphic Hardware Communication Systems. In: *IEEE Access* 7, pp. 135606–135620. ISSN: 2169-3536. DOI: 10.1109/ACCESS.2019.2941772.

Zenke, F. and S. Ganguli (June 2018). SuperSpike: Supervised Learning in Multilayer Spiking Neural Networks. In: *Neural Computation* 30.6, pp. 1514–1541. ISSN: 1530-888X. DOI: 10.1162/neco_a_01086. URL: http://dx.doi.org/10.1162/neco_a_01086.

Software References

Franco (2021). fxpmath. https://github.com/francof2a/fxpmath.

Hazan, H., D. J. Saunders, H. Khan, D. Patel, D. T. Sanghavi, H. T. Siegelmann, and R. Kozma (2018). BindsNET: A Machine Learning-Oriented Spiking Neural Networks Library in Python. In: *Frontiers in Neuroinformatics* 12, p. 89. ISSN: 1662-5196. DOI: 10.3389/fninf.2018.00089. URL: https://www.frontiersin.org/article/10.3389/fninf.2018.00089.

Jakob, W., J. Rhinelander, and D. Moldovan (2017). pybind11 – Seamless operability between C++11 and Python. https://github.com/pybind/pybind11.

Vinkelis, M. (2020). Bitsery. https://github.com/fraillt/bitsery.

Acronyms and Technical Terms

ADC analog-to-digital converter ..... (pp. 28, 29)

AdEx adaptive exponential integrate-and-fire ..... (pp. 27, 103)

ANN artificial neural network ..... (pp. 1, 5, 12)

ANNCORE analog network core ..... (pp. 1, 25)

AP action potential ..... (pp. 6, 7, 10)

BP backpropagation ..... (p. 14)

BPTT back-propagation through time ..... (pp. V, VII, 2, 18–20, 102)

BSS-2 BrainScaleS-2 ..... (pp. V, VII, 1–3, 25, 99)

CADC column ADC ..... (pp. 29, 28, 29, 40, 45, 46, 66, 69, 82, 83, 102)

CapMem capacitive memory ..... (p. 27)

CNN convolutional neural network ..... (p. 12)

e-prop eligibility propagation ..... (pp. 3, 31, 45, 50, 52, 53, 65, 76, 99)

EPSP excitatory post-synaptic potential ..... (p. 7)

ERN error-related negativity ..... (p. 22)

FNN feedforward neural network ..... (pp. 12–14, 22, 61)

FPGA field-programmable gate array ..... (pp. 25, 47)

GRU gated recurrent unit ..... (p. 15)

HBP Human Brain Project ..... (p. 1)

HX HICANN-X v2 ..... (pp. 25–28, 27–29, 31, 33, 35–38, 37–41, 45–47, 49–53, 56, 58, 60–62, 65, 66, 68, 75–77, 81–83, 85, 86, 93, 99, 101–103, 122)

IPSP inhibitory post-synaptic potential ..... (p. 7)

ISI inter-spike interval ..... (p. 50)

LI leaky integrate ..... (pp. 17, 50)

LIF leaky integrate-and-fire ..... (pp. 8, 9, 11, 15–18, 20, 25, 27, 33, 47, 50, 62, 75, 99, 103)

LSNN long short-term memory spiking neural network ..... (p. 103)

LSTM long short-term memory ..... (pp. 15, 19)

MADC membrane ADC ..... (pp. 27, 35, 37, 47, 58)

MLP multi-layer perceptron ..... (p. 13)

NASProp Neuromorphic Accumulative Spike Propagation. (pp. 2, 3, 72, 76, 77, 79, 87, 91, 97, 99, 101, 103)

ODE ordinary differential equation ..... (p. 6)

PPU plasticity processing unit. (pp. 25, 28, 29, 34, 39–43, 42–47, 62, 65, 66, 68, 69, 71, 74, 75, 77, 79–85, 87, 86, 91, 96, 97, 101, 103, 122)

PSP post-synaptic potential ..... (pp. 7, 9, 12)

ReLU rectified linear unit ..... (p. 13)

RNN recurrent neural network ..... (pp. V, 2, 5, 12, 14, 15)

RSNN recurrent spiking neural network. (pp. V, VII, 2, 3, 5, 12, 14, 15, 17, 31, 33, 35, 45, 47, 49–52, 61, 65, 74, 76, 79, 99, 104)

RSS residual sum of squares ..... (pp. 21, 45, 69)

s-prop spike-based eligibility propagation (pp. 2, 3, 47, 49, 52, 53, 58, 60, 61, 65, 75, 76, 99)

SGD stochastic gradient descent ..... (pp. 13, 75, 76, 101)

SIMD single instruction multiple data ..... (p. 1)

SNN spiking neural network ..... (pp. 1, 25)

STDP spike timing dependent plasticity ..... (p. 29)

VU vector unit ..... (pp. 25, 28, 29, 28, 40, 44, 66, 75, 81, 82, 84, 96, 101)

A Appendix

A.1 Parameter

Parameter tables for individual experiments refer to a Parameter Base, accumulating common parameters. If parameters of an individual experiment are also given in the parameter base, they are overwritten by the experiment-specific ones. Time units are always given in $\mathrm{ms}_{\mathrm{bio}}$.

A.1.1 S-prop

Table A.1: Parameter Base

Parameter	Value
Learning	
Regularization strength η^{reg}	10000
f^{target}	$40\,\mathrm{Hz}_{\mathrm{bio}}$
Γ	0.8, every 200 epochs
Optimizer	Adam
Simulation	
δT	$1\,\mathrm{ms}_{\mathrm{bio}}$
T	$1000\,\mathrm{ms}_{\mathrm{bio}}$
Network	
n^{i}	30
n^{h}	70
n^{o}	1
$\mathrm{Std}(\theta^{\mathrm{ih}})_{\mathrm{init}}$	15
$\mathrm{Std}(\theta^{\mathrm{hh}})_{\mathrm{init}}$	1
$\mathrm{Std}(\theta^{\mathrm{ho}})_{\mathrm{init}}$	0
Neurons	
τ_{syn}	$2\,\mathrm{ms}_{\mathrm{bio}}$
τ_{ref}	$1\,\mathrm{ms}_{\mathrm{bio}}$
$\tau_{\mathrm{m}}^{\mathrm{h}}$	$20\,\mathrm{ms}_{\mathrm{bio}}$
$\tau_{\mathrm{m}}^{\mathrm{o}}$	$20\,\mathrm{ms}_{\mathrm{bio}}$
v_{leak}	0 a.u.
v_{r}	0 a.u.
v_{th}	40 a.u.
Data	
η_{T}	100
n_p	3
$w_{i,k}$	$[0.5, 2)$
$T_{i,k}$	$[0.6\pi, 2\pi)$
$\varphi_{i,k}$	$\left[\frac{0.5}{1000}, \frac{2\pi}{1000}\right)$
$T_{\mathrm{ISI}}^{\mathrm{in}}$	$40\,\mathrm{ms}_{\mathrm{bio}}$, Poisson

Table A.2: Grid Search

Parameter	Value
Parameter Base	Table A.1
$\eta_{\mathrm{r}}, \eta_{\mathrm{o}}$	$\{0.5, 0.05, 0.03, 0.005\}$

Table A.3: Baseline

Parameter	Value
Parameter Base	Table A.1
$\eta_{\mathrm{r}}, \eta_{\mathrm{o}}$	0.05
e-prop specific	
γ	3

Table A.4: Stochastic Weight Updates

Parameter	Value
Parameter Base	Table A.1
$\eta_{\mathrm{r}}, \eta_{\mathrm{o}}$	0.05
Weights θ	$\in \mathbb{N}_{-63}^{63}$
Noise	
Gaussian noise on membranes	$\sim \mathcal{N}(0, 0.4)$, Gaussian
τ_{syn}	$\sim \mathcal{N}(2, 0.2)$, Gaussian
τ_{m}	$\sim \mathcal{N}(20, 2)$, Gaussian
Synaptic strength scale	$\sim \mathcal{N}(1, 0.1)$, Gaussian

Table A.5: Small Output Weights

Parameter	Value
Parameter Base	Table A.1
$\eta_\mathrm{r}, \eta_\mathrm{o}$	0.05
σ_o	$\in \{0.1, 0.5, 0.8, 1.0\}$
Weights θ	$\in \mathbb{N}_{-63}^{63}$
Noise	
Gaussian noise on membranes	$\sim \mathcal{N}(0, 0.4)$, Gaussian
τ_syn	$\sim \mathcal{N}(2, 0.2)$, Gaussian
τ_m	$\sim \mathcal{N}(20, 2)$, Gaussian
Synaptic strength scale	$\sim \mathcal{N}(1, 0.1)$, Gaussian

Table A.6: Hardware Experiment

Parameter	Value
Parameter Base	Table A.1
η_r	0.05
η_o	0.02
σ_o	0.1
Weights θ	$\in \mathbb{N}_{-63}^{63}$
$\mathrm{Std}(\theta^\mathrm{ih})_\mathrm{init}$	25
$\mathrm{Std}(\theta^\mathrm{hh})_\mathrm{init}$	2
$\mathrm{Std}(\theta^\mathrm{ho})_\mathrm{init}$	0
Neurons	
τ_syn	4 ms
τ_ref	1 ms
$I_\mathrm{syn,\,gm}$ spiking neurons	800 DAC Values
$I_\mathrm{syn,\,gm}$ readout neuron	200 DAC Values
v_leak	120 DAC Values
v_r	120 DAC Values
v_th	160 DAC Values

Table A.7: Role of Recurrence

Parameter	Value
Parameter Base	Table A.6
Non-recurrent	
θ^hh	0 for all epochs
Liquid	
θ^hh	Constant for all epochs. Initialized from scrambled learned recurrent projection

Table A.8: Stability

Parameter	Value
Parameter Base	Table A.6
$\eta_\mathrm{r}, \eta_\mathrm{o}$	0.0
$\theta^\mathrm{ih}, \theta^\mathrm{hh}, \theta^\mathrm{ho}$	Given by projections trained for 1000 epochs
Epochs	100

A.1.2 On-Chip Learning

Simulation

Table A.9: Baseline

Parameter	Value
Parameter Base	Table A.1
$\eta_\mathrm{r}, \eta_\mathrm{o}$	0.03
η^reg	5000
NASProp	
P	$\{25, 50\}$ ms_bio
e-prop specific	
γ	3

Table A.10: P sweep

Parameter	Value
Parameter Base	Table A.1
P	$\{10, 20, 30, 40, 50, 60, 70, 80, 90, 100, 110, 120\}$ ms_bio
Baseline	Table A.9
SGD	
Optimizer	SGD
η_r	$9 \cdot 10^{-5}$
η_o	$1 \cdot 10^{-4}$
η^reg	5000
SGD, discrete	In addition to above
σ_o	0.1
Weights θ	$\in \mathbb{N}_{-63}^{63}$
SGD, discrete, noisy	In addition to above
Gaussian noise on membranes	$\sim \mathcal{N}(0, 0.4)$, Gaussian
τ_syn	$\sim \mathcal{N}(2, 0.2)$, Gaussian
τ_m	$\sim \mathcal{N}(20, 2)$, Gaussian
Synaptic strength scale	$\sim \mathcal{N}(1, 0.1)$, Gaussian
τ_c	$\sim \mathcal{N}(20, 2)$, Gaussian
η_c	$\sim \mathcal{N}(1, 0.1)$, Gaussian
SGD, discrete, noisy no reg	In addition to above
η^reg	0

Table A.11: Recurrence

Parameter	Value
Parameter Base	Table A.1
P	$\{50, 100\}$ ms_bio
η^reg	5000
Baseline	
η_r	$3 \cdot 10^{-2}$
η_o	$3 \cdot 10^{-2}$
Simulation	
η_r	$9 \cdot 10^{-5}$
η_o	$1 \cdot 10^{-4}$
Optimizer	SGD
σ_o	0.1
Weights θ	$\in \mathbb{N}_{-63}^{63}$
Gaussian noise on membranes	$\sim \mathcal{N}(0, 0.4)$, Gaussian
τ_syn	$\sim \mathcal{N}(2, 0.2)$, Gaussian
τ_m	$\sim \mathcal{N}(20, 2)$, Gaussian
τ_c	$\sim \mathcal{N}(20, 2)$, Gaussian
η_c	$\sim \mathcal{N}(1, 0.1)$, Gaussian
Synaptic strength scale	$\sim \mathcal{N}(1, 0.1)$, Gaussian

Table A.12: Correlation Base

Parameter	Value
Cube Setup	69
Constants	
Target η_c	15 a.u. per pre-post pair
Target τ_c	20 $\mathrm{ms_{bio}}$
Analog Parameters	
`mux_dac_25`	3200
`syn_i_bias_corout`	400
`syn_i_bias_ramp`	45
`syn_i_bias_store`	50
Calibration	
`amp_calibs`, `time_calibs`	Chosen such that error to target curve is minimized.

Table A.13: Example

Parameter	Value
Cube Setup	69
Correlation Base	Table A.12
Training	
Epochs	100
$\theta_{\mathrm{init}}^{\mathrm{ho}}$	25 a.u., fixed
$\theta_{\mathrm{init}}^{\mathrm{hh}}$	−63 a.u.
$y^{*,n} - y^n$	100 CADC values
Rec Neuron ID	0 (on HX)
Out Neuron ID	60 (on HX)
Signed row ID	0 (on HX)
$T_{\mathrm{ISI}}^{\mathrm{pre}}$	10 $\mathrm{ms_{bio}}$, Poisson
$T_{\mathrm{ISI}}^{\mathrm{post}}$	10 $\mathrm{ms_{bio}}$, Poisson
P	100 $\mathrm{ms_{bio}}$
T	1000 $\mathrm{ms_{bio}}$
Updates	Accumulative
Info	Random numbers are bit-shifted by 3 positions to the right.

Table A.14: Example Output Weight

Parameter	Value
Cube Setup	69
Correlation Base	Table A.12
Training	
Epochs	100
$\theta_{\mathrm{init}}^{\mathrm{ho}}$	−63 a.u., variable
$\theta_{\mathrm{init}}^{\mathrm{hh}}$	−30 a.u.
$y^{*,n} - y^n$	100 CADC values
Rec Neuron ID	0 (on HX)
Out Neuron ID	60 (on HX)
Signed row ID	0 (on HX)
$T_{\mathrm{ISI}}^{\mathrm{pre}}$	10 $\mathrm{ms_{bio}}$, Poisson
$T_{\mathrm{ISI}}^{\mathrm{post}}$	10 $\mathrm{ms_{bio}}$, Poisson
P	100 $\mathrm{ms_{bio}}$
T	1000 $\mathrm{ms_{bio}}$
Updates	Accumulative
Info	Random numbers are bit-shifted by 2 positions to the right.

Table A.15: Example Error Sign Change

Parameter	Value
Cube Setup	69
Correlation Base	Table A.12
Training	
Epochs	200
$\theta_{\mathrm{init}}^{\mathrm{ho}}$	−30 a.u., variable
$\theta_{\mathrm{init}}^{\mathrm{hh}}$	−63 a.u.
$y^{*,n} - y^n$	$\{100, -100\}$ CADC values
Rec Neuron ID	0 (on HX)
Out Neuron ID	60 (on HX)
Signed row ID	0 (on HX)
$T_{\mathrm{ISI}}^{\mathrm{pre}}$	10 $\mathrm{ms_{bio}}$, Poisson
$T_{\mathrm{ISI}}^{\mathrm{post}}$	10 $\mathrm{ms_{bio}}$, Poisson
P	100 $\mathrm{ms_{bio}}$
T	1000 $\mathrm{ms_{bio}}$
Updates	Accumulative
Info	Random numbers are bit-shifted by 2 positions to the right.

Table A.16: Error Sweep

Parameter	Value
Cube Setup	69
Correlation Base	Table A.12
Training	
Epochs	100
Runs	20, with different seeds
$\theta_{\mathrm{init}}^{\mathrm{ho}}$	30 a.u., fixed
$\theta_{\mathrm{init}}^{\mathrm{hh}}$	$\{-63$ a.u., 63 a.u.$\}$
$y^{*,n} - y^n$	$\{100, 80, 60, 40, 20, 10, -10, -20, -40, -60, -80, -100\}$ CADC values
Rec Neuron ID	0 (on HX)
Out Neuron ID	60 (on HX)
Signed row ID	0 (on HX)
$T_{\mathrm{ISI}}^{\mathrm{pre}}$	10 $\mathrm{ms_{bio}}$, Poisson
$T_{\mathrm{ISI}}^{\mathrm{post}}$	10 $\mathrm{ms_{bio}}$, Poisson
P	100 $\mathrm{ms_{bio}}$
T	1000 $\mathrm{ms_{bio}}$
Updates	Accumulative
Info	Random numbers are bit-shifted by 3 positions to the right.

Table A.17: Correlation Sweep

Parameter	Value
Cube Setup	69
Correlation Base	Table A.12
Training	
Epochs	100
Runs	20, with different seeds
$\theta^{\mathrm{ho}}_{\mathrm{init}}$	30 a.u., fixed
$\theta^{\mathrm{hh}}_{\mathrm{init}}$	$\{-63\,\mathrm{a.u.}, 63\,\mathrm{a.u.}\}$
$y^{*,n} - y^n$	100 CADC values
Rec Neuron ID	0 (on HX)
Out Neuron ID	60 (on HX)
Signed row ID	0 (on HX)
$T^{\mathrm{pre}}_{\mathrm{ISI}}$	$10\,\mathrm{ms_{bio}}$, Poisson
$T^{\mathrm{post}}_{\mathrm{ISI}}$	$\{2, 5, 10, 20, 30, 40\}$ $\mathrm{ms_{bio}}$, Poisson
P	$100\,\mathrm{ms_{bio}}$
T	$1000\,\mathrm{ms_{bio}}$
Updates	Accumulative
Info	Random numbers are bit-shifted by 3 positions to the right.

Table A.18: Multiple Synapses

Parameter	Value
Cube Setup	69
Parameter Base	Table A.13
Rec Neuron ID	$[0, 138]$, every second (on HX)
Out Neuron ID	140 (on HX)
Signed row ID	0 (on HX)

On-Chip Full Network

Table A.19: Full Network on-chip

Parameter	Value
Correlation Base	Table A.12
Learning	
Epochs	200
T	$1000\,\mathrm{ms_{bio}}$
P	$100\,\mathrm{ms_{bio}}$
Updates	Accumulative
Batch size	100
Info	Random numbers are bit-shifted by 2 positions to the right.
Network	
n^{i}	30
n^{h}	70
n^{o}	1
$\mathrm{Std}(\theta^{\mathrm{ih}})_{\mathrm{init}}$	25
$\mathrm{Std}(\theta^{\mathrm{hh}})_{\mathrm{init}}$	1
$\mathrm{Std}(\theta^{\mathrm{ho}})_{\mathrm{init}}$	1
Neurons	
τ_{syn}	4 ms
τ_{ref}	1 ms
$I_{\mathrm{syn,\,gm}}$ spiking neurons	800 a.u.
$I_{\mathrm{syn,\,gm}}$ readout neuron	200 a.u.
v_{leak}	120 a.u.
v_{r}	120 a.u.
v_{th}	160 a.u.
Data	
η_{T}	100
n_p	1
$w_{i,k}$	1
$T_{i,k}$	$2000\,\mathrm{ms_{bio}}$
$\varphi_{i,k}$	$\{0, 100\}$ $\mathrm{ms_{bio}}$
$T^{\mathrm{in}}_{\mathrm{ISI}}$	$20\,\mathrm{ms_{bio}}$, Poisson

A.2 Further Methods

A.2.1 Stochastic Weight Updates

A crucial hardware property that has to be taken into account when implementing learning on HX are the discretized synaptic weights described in Section 3.1. On hardware a weight is given by a integer $n \in \mathbb{N}_0^{63}$ which increments the synaptic strength with $\Delta\theta^{\text{hw}}$. These increments are subject to chip setup and calibration. For gradient-based learning, the knowledge of their actual value is not so critical. However, to increase learning accuracy they should be chosen such, that the learned hardware weights

$$\theta_{ji}^{\text{hw}} = n \cdot \Delta\theta^{\text{hw}} \quad \text{with } n \in \{0, \dots, 63\} \tag{A.1}$$

exploit the possible range without saturating. The issue with these discrete weights arises when performing weight updates. Assuming a weight update $\Delta\theta_{ji}$, the weights $\theta_{ji}^{\text{hw, old}}$ are adjusted according to

$$\theta_{ji}^{\text{hw, new}} = \theta_{ji}^{\text{hw, old}} + \Delta\theta_{ji}. \tag{A.2}$$

Since the weight updates are usually smaller than the increments on hardware, the new weights cannot be calculated directly. One possible solution is to round $\Delta\theta_{ji}$ to the nearest multiple of $\Delta\theta^{\text{hw}}$. However, if the updates are too small, the weights remain constant. The solution proposed are stochastic weight updates [Breitwieser, 2015]. That is, the weights are increased by the updates up rounded to the next higher multiple of the hardware increment $\lceil \Delta\theta_{ji} \rceil$ with a probability p given by the decimal number of the given update,

$$p = \frac{\Delta\theta_{ji} - \lfloor \Delta\theta_{ji} \rfloor}{\Delta\theta^{\text{hw}}}. \tag{A.3}$$

Here $\lfloor \Delta\theta_{ji} \rfloor$ denotes the down rounded update to the next lower multiple of the hardware increment. Correspondingly, the weights are updated according to

$$\theta_{ji}^{\text{hw, new}} = \theta_{ji}^{\text{hw, old}} + \begin{cases} \lceil \Delta\theta^{\text{hw}} \rceil & \text{with } p \\ 0 & \text{with } 1-p, \end{cases} \tag{A.4}$$

which leads to the correct gradient on average over many training epochs. In simulation and for in-the-loop training, as described in Section 4.2.1, this can be implemented easily on the host-side. For on-chip learning on the PPU the on-chip random number generator is utilized to update the weights stochastically.

A.3 Software

Experiments conducted in this thesis used to the software state in Table A.20.

Table A.20: Software State

Repository	Commit-Hash	Commit Message
libnux	94a8fc588365ba180573817f1d45deb22937d786	Safety commit of final msc state for EA
haldls	0c75dfdcc48d9377ebf5eabef31be4800faee427	Safety commit of final msc state for EA
code-format	5d55a9952d4b6400fa5b2baeff9be546e45bf76d	Pylint: Disable 'duplicate-code' warning
logger	bc006238ecfdc483d5b96ce5f5bb62e5a93e99dd	Add log4cxx_level_v2
halco	6aded742a6694adff5a1296e66ddc08894b4e4b6	Update README
hate	c7483cedc3d76b8e7a4a65e7bc9a423131f40ce1	Enable ccache in CI
hxcomm	a91f0cdc5d6470a060a0adadccf7b33a1c2d42a7	Add initial doc/ for user documentation
pywrap	83ddbad8a114b4730b82d299e8bd9da2a6ca5ebb	Support builds w/o generating Python bindings
fisch	cef78ebf3aa8e009f5d7b2a2120b377d0fddb59f	Add initial doc/ for user documentation
rant	4fc2cc3689c9b141708dafbcc5f9d3c7c2b7f18d	Update to gtest 2.0.0
ztl	d900ab073f6aa8df4bf7f187bdbb65f1f6cac2f6	Add .gitreview
lib-boost-patches	2d7e07d4e74827c42d9e1a51f8d180af9907f7cb	Update content description in Readme
sctrltp	227a5ddc0333b88a2a043e1927c1f549b8247487	Handle init response over multiple packets
hwdb	fa2060f560bacf0dd91294dbd44f64524073997e	Update cube 3's triggers for FPGA 0
visions-slurm	5e7ea560235b068fc12f26e3f0d002d415f76cf9	Add hwdb yaml environment export
flange	fcde2aafe69805487789ca0b1a8a245caf5fb8ed	Support builds w/o generating Python bindings
lib-rcf	5b16326ae30ee08a322a6569887ca8bd2684c252	Fixes for log4cxx@0.11.0
bss-hw-params	a82226052ae8e434c1a4b4f9ec781ff1578427d9	Add compatibility constant for cube_-ethernet
grenade	80985b242cc7b76278410c4bc9b4322795448335	Safety commit of final msc state for EA
pyublas	fb538e8c313a3f04d1a5b77200d192fece3ea901	Add .gitreview
model-hw-eprop	ea86d3d88778b76f2b0282821e3952dbbd77ff16	Safety commit of final msc state for EA
calix	5b3871532c000dfb7572034386ac49edb5b3c696	Individually select leak div. or mult.
hxtorch	f7221ad40f6e73de72faa256784e1fad40a2330f	Safety commit of final msc state for EA

Used `singularity` [Kurtzer et al., 2017] container is given in Table A.21.

Table A.21: Container

Path	`/containers/stable/2021-03-26_1.img`
App	`dls`
Overlay	`/containers/manual/mueller/overlays/2020-12-15_2.img`

Erklärung:

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

Heidelberg, 21 Juni 2021

.....
(Unterschrift)