Department of Physics and Astronomy

Heidelberg University

Master thesis

in Physics

submitted by

Luca Blessing

born in Pforzheim

2023

Gradient Estimation With Sparse Observations for Analog Neuromorphic Hardware

This Master thesis has been carried out by Luca Blessing at the

Kirchhoff-Institute for Physics in Heidelberg

under the supervision of

Dr. Johannes Schemmel

Gradient Estimation With Sparse Observations for Analog Neuromorphic Hardware

Spiking neural networks (SNNs) take into account the temporal dimension of biological neurons, in contrast to *artificial neural networks* (ANNs). Due to the discontinuous nature of spiking neurons, this leads to challenges in gradient-based optimization of such a model using *backpropagation through time* (BPTT). The novel EventProp algorithm is based on a general treatment of mathematical neuron models and derives adjoint dynamical equations and transitions for networks of non-refractory *leaky-integrate and fire* (LIF) neurons from which exact parameter gradients in *spiking neural networks* (SNNs) can be obtained. This algorithm is studied in this thesis, where it is extended to the refractory *leaky-integrate and fire* (LIF) model and a closed gradient expression is derived. The EventProp algorithm is implemented in a discrete-time form in a high-level software framework, and this learning rule is used for the first time to train SNNs with the analog neuromorphic *BrainScaleS-2* (BSS-2) system. The results demonstrate that gradient-based learning on sparse measurements is feasible in analog neuromorphic hardware, and establish EventProp as an important alternative to common surrogate gradient approaches. The high adaptability of the underlying adjoint sensitivity analysis would allow EventProp to be extended to other neuron models and even online-learning methods which would broaden the applications of this method widely.

Gradientenbestimmung mit wenigen Beobachtungen für analoge neuromorphe Hardware

Spikende neuronale Netzwerke (SNNs) berücksichtigen im Gegensatz zu *künstlichen neuronalen Netzwerken* (KNNs) die zeitliche Dimension von biologischen Neuronen. Aufgrund der diskontinuierlichen Natur von spikenden Neuronen führt dies zu Herausforderungen bei der gradientenbasierten Optimierung eines solchen Modells mit *backpropagation through time* (BPTT). Der neuartige EventProp-Algorithmus basiert auf einer allgemeinen Behandlung mathematischer Neuronenmodelle und leitet adjungierte dynamische Gleichungen und Übergänge für Netzwerke nicht-refraktärer *leaky-integrate and fire* Neuronen ab, aus denen genaue Parametergradienten in SNNs gewonnen werden können. Dieser Algorithmus wird in dieser Arbeit untersucht, wobei er auf das refraktäre LIF Modell erweitert und ein geschlossener Gradientenausdruck abgeleitet wird. Der EventProp-Algorithmus wird in einer zeitdiskreten Form in einem High-Level-Software-Framework implementiert, und diese Lernregel wird zum ersten Mal verwendet, um SNNs mit dem analogen neuromorphen *BrainScaleS-2* (BSS-2) System zu trainieren. Die Ergebnisse zeigen, dass gradientenbasiertes Lernen auf spärlichen Messungen in analoger neuromorpher Hardware möglich ist, und etablieren EventProp als eine wichtige Alternative zu herkömmlichen Surrogat-Gradienten-Ansätzen. Die hohe Anpassungsfähigkeit der zugrundeliegenden adjungierten Sensitivitätsanalyse würde es ermöglichen, EventProp auf andere Neuronenmodelle und sogar Online-Lernmethoden zu erweitern, was die Anwendungsbereich dieser Methode deutlich erweitern würde.

Contents

1 Introduction

Seeking to understand our physical surroundings is intrinsic to human behavior, and scientific progress leads to ever-new innovations and findings about the world around and inside us. Crucial to this progress are the enormous capabilities of the human brain, but the brain itself, as an object of scientific study, still holds many unimagined insights. Inspired by its ability to perceive and efficiently process complex information and striving for a deeper understanding, scientists turn to the application of artificial machines to mimic at least part of the processes in our brains.

Ideas of modeling neurons by condensing key characteristics into building blocks like the perceptron [Rosenblatt 1958] and training *artificial neural networks* (ANNs) comprised of these units [Rumelhart et al. 1986] have already been studied quite early. Extensive developments of computational resources in recent decades have made the application of such networks to solve pattern recognition, classification, or reinforcement learning tasks increasingly successful [LeCun et al. 2015]. Although these computational advances enable solving ever more complex tasks, ANNs largely ignore the time-dependent and sparse nature of the human brain.

Moving closer to the original biological counterparts, SNNs model dynamical systems evolving in time and handling inter-neuron communication by all-or-nothing spike events [Gerstner and Kistler 2002]. Alongside the efforts to gain a deeper understanding of neural computations conceptually, computational neuroscience applies these models in neuromorphic systems [Mead 1990]. There are multiple approaches implementing such neuromorphic systems, examples being SpiNNaker [Furber et al. 2012, Mayr et al. 2019], IBM's TrueNorth [Merolla et al. 2014], Intel's Loihi [Davies et al. 2018, Orchard et al. 2021], DYNAPs [Moradi et al. 2018] or Neurogrid [Benjamin et al. 2021].

Another neuromorphic system, which is also used in the work of this thesis, is the *BrainScaleS-2* (BSS-2) analog neuromorphic hardware [Schemmel et al. 2022, Pehle et al. 2022], a mixed-signal accelerator developed in Heidelberg as part of the *Human Brain Project* (HBP). It implements the *adaptive exponential integrate-and-fire* (AdEx) neuron model [Brette and Gerstner 2005] in analog circuits and provides high-bandwidth digital communication of spike events. The BSS-2 system achieves 10^3 speed-up compared to biological time, regardless of network size, and high energy-efficiency compared to traditional simulators [Müller 2014]. Using different approaches, learning was demonstrated successfully for networks on the BSS-2 platform [Cramer et al. 2022, Göltz et al. 2021, Arnold et al. 2023].

Compared to the already established and highly researched optimization techniques in ANNs, the possibilities of learning in SNNs remain subject to study on a quite fundamental level. Different gradient-based solutions have been applied to supervised training of SNNs with *backpropagation through time* (BPTT) inspired by *recurrent neural networks* (RNNs) [Neftci et al. 2019] and also for online learning [Bellec et al. 2020]. A core challenge for both methods remains the discontinuity introduced by the spike-based inter-neuron communication. A common approach is introducing continuous auxiliary functions, whose gradients are used as surrogates during optimization [Zenke and Ganguli 2018, Neftci et al. 2019]. As the implementations of this algorithm rely on observations of the neuron state at times other than only the spike times, other approaches emerged, formulating equations to compute exact gradients based only on observations at spike times [Göltz et al. 2021, Wunderlich and Pehle 2021, Pehle 2021].

Göltz et al. [2021] derived closed-form equations for gradients for specific ratios of a LIF neuron's time constants and applied these to *in-the-loop* (ITL) training on BSS-2. Meanwhile, Pehle [2021] took inspiration from optimal control theory and proposed a general framework to derive sets of *adjoint equations* in networks of spiking neurons and Wunderlich and Pehle [2021] formulated the EventProp algorithm by deriving the equations explicitly for the non-refractory LIF neuron model. In simulation, successful optimization of models using EventProp was already achieved, but demonstrating training of networks on analog neuromorphic hardware with this novel algorithm remained an open task and was addressed during this thesis.

The work of this thesis aims to demonstrate the capabilities of the adjoint method by extending EventProp to the LIF neuron model with a refractory period. Furthermore, the correspondence of EventProp to the closed-form analytical equations in the work of Göltz et al. [2021] is explicitly derived. A time-discrete version of the EventProp algorithm was implemented and applied to ITL training with the BSS-2 chip on the low-dimensional Yin-Yang task [Kriener et al. 2022] and in simulation on the MNIST dataset [LeCun et al. 1998] . The implementation is incorporated into `hxtorch.snn` [Spilger et al. 2022], which has a high-level, PyTorch-based frontend, giving non-expert users access to this gradient estimation method, additionally to the previously available surrogate gradient implementation.

Thesis Outline

In chapter 2, an overview of the theoretical and methodological background is given, including an introduction to biological neurons and the mathematical model of the LIF neuron, gradient-based learning in SNNs, and the derivation of exact gradient estimation algorithms for such models. Additionally, the BSS-2 platform and the used software frameworks are outlined.

In chapter 3, I derive a set of equations similar to EventProp for the neuron model LIF with a refractory period. In addition, I derive a gradient expression in closed form from EventProp and show its correspondence to the work of Göltz et al. [2021].

Chapter 4 goes into detail on my software contributions, some streamlining experiment execution on BSS-2 and the other enabling the usage of spike-time based loss functions for SNNs within `hxtorch.snn`.

Chapter 5 covers the time-discrete implementation of the EventProp algorithm, a single-synapse and a single-hidden-neuron experiment meant to verify the implementation and training on two benchmark tasks: the Yin-Yang and the MNIST dataset. For the former, training was done in simulation and with BSS-2 ITL, while for the latter, preliminary simulation results were obtained with a network suitable for hardware execution.

Chapter 6 summarizes and discusses the results, puts those into context with other current work, and gives an outlook on what will come next.

2 Theoretical Background

The foundation of this thesis is built upon a comprehensive understanding of the biological neuron and its mathematical models, which are crucial in developing *spiking neural networks* (SNNs). Through this chapter, the building blocks of such networks will be described in detail, providing a solid grounding for the subsequent research. The chapter will also explore the execution and optimization of SNNs on neuromorphic hardware, paving the way for the work conducted for this thesis.

2.1 Computational Neuroscience

Drawing on the work of [Gerstner et al. 2014] and [Petrovici 2015], I will outline the fundamental workings of biological neurons in section 2.1.1, and go into the details of the LIF neuron model and its mathematical description in section 2.1.2.

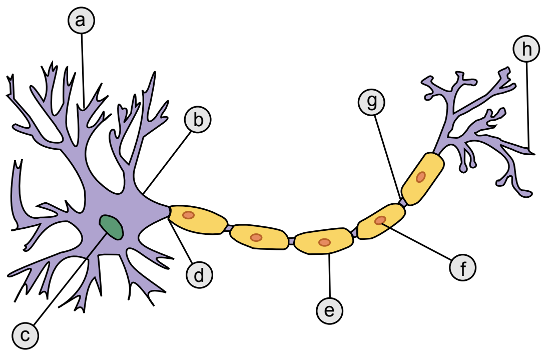
2.1.1 Elements of a Neuron

Cells are enclosed by a membrane separating their interior from the extracellular environment. Most of this membrane is a lipid bilayer, which is permeable for small charge-free particles, but impermeable to ions and polar molecules. Apart from that, the membrane also contains proteins enabling the transport of ions accross the membrane and therefore driving the dynamics of the voltage accross the membrane. Those ion gates and channels lead to a resting potential of about $-70\,\mathrm{mV}$.

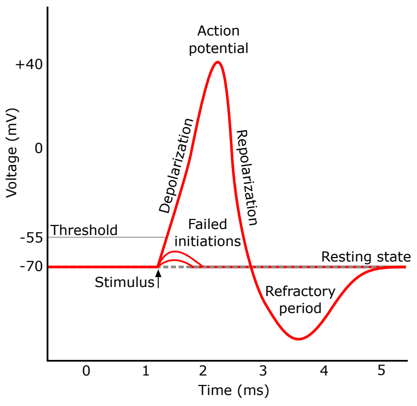
In contrast to other cells, neurons are exciteable and have additional ways of manipulating the membrane potential, which ultimately allows them to communicate by events called *action potentials*, or *spikes*, cf. fig. 2.1. Action potentials are triggered by the membrane potential exceeding a threshold. The membrane potential is then subject to a spontaneous depolarization followed by a repolarization or even hyperpolarization, letting the potential even drop below the neurons resting potential. This phase of hyperpolarization is also called the *refractory period*, during which even strong stimuli are unable to invoke another action potential.

Action potentials travel from the soma along the axon to its terminals, where the neuron connects to the dendrites of subsequent neurons. These connections are called synapses. If an action potential of a neuron is transmitted to the axon terminal, this triggers a cascade of biochemical processes effectively resulting in a *post-synaptic transmembrane current* (PSC) in the connected neuron. The PSC is transmitted by the dendrites to the soma and might lead to further action potentials.

Even though the precise spatial structure of neurons is likely to be important for the way they process signals, the models considered in this work reduce the neuron



(a) Schematics of a neuron



(b) Action potential

Figure 2.1: (a) The three main functional parts of a typical neuron are the dendrites “a”, the soma “b” and the axon “d”. The dendrites forward input signals from other neurons to the soma, which processes them and eventually generates an output signal, which is then delivered to other neuron by the axon. Where the axon terminals “h” connect to post-synaptic neuron’s dendrites are called synapses. Image taken from [Jarosz 2009]. **(b)** The action potential is generated by the soma if the neuron’s membrane voltage exceeds a threshold. After the pulse-like swing, the membrane potential enters a hyperpolarization phase called the ‘refractory period’. Image taken from [Iberri 2007].

to a point-like, spike-generating object, modeling its behaviour by analogous electrical circuits. A well known and very accurate description of such point-like neurons is the model introduced by Hodgkin and Huxley [1952]. The rapid depolarization and following repolarization arise from the model itself, but due to the detailed description using multiple differential equations is quite computationally expensive to consider the dynamics of larger networks of such neurons. Another model, which is also implemented with analog circuits on the BSS-2 chip, is the *adaptive exponential integrate-and-fire* (AdEx) neuron [Brette and Gerstner 2005]. This model describes well the depolarization of the membrane when exceeding a soft threshold, but it needs to be force-reset when the depolarized membrane crosses a higher, hard threshold. A simpler version of this model, which is used throughout this work, is the LIF neuron model, described in the subsequent section. It represents the spiking behaviour in a very simplified but efficient way, by artificially enforcing them when a transition condition is met. Due to its simplicity, this mathematical description of neurons is often used in modelling of large-scale networks.

2.1.2 Leaky-Integrate and Fire Neuron Model

First introduced by Lapicque [1907] and later named the “leaky-integrate and fire” model, the neuron model described here simplifies the complexity of biological neurons but still describes key dynamics sufficiently well. This makes this model useful in computational neuroscience in general and hence, also for research and applications of neuromorphic computing in simulations and on hardware [Bellec et al. 2020, Cramer et al. 2022, Göltz et al. 2021].

The subthreshold dynamics of the membrane potential V can be described by the first order *ordinary differential equation* (ODE)

$$C_\mathrm{m}\frac{\mathrm{d}V}{\mathrm{d}t} = -g_\mathrm{l}\left(V - V_\mathrm{l}\right) + I_\mathrm{syn}, \tag{2.1}$$

with the membrane capacitance C_m, leak conductance g_l and leak reversal potential V_l, towards which the membrane potential decays back over time. The synaptic current I_syn accounts for any external currents arriving through synapses, and is integrated over time onto the membrane. The time t^post of a post-synaptic spike is defined by the threshold crossing condition $V^-(t^\mathrm{post}) - V_\mathrm{th} = 0$. After spike-emission the membrane potential is reset to

$$V^+(t^\mathrm{post}) = V_\mathrm{reset}. \tag{2.2}$$

Note the order of occurence, where first the threshold condition is fullfilled and then, after spike emission, the transition to a reset is forced. To emphasize this,the left-/right-hand limit $V^{-/+}(t^\mathrm{post}) = \lim_{t\to t^\mathrm{post}-/+} V(t)$ is used.

The time-scale on which the membrane responds to its input can be described by the membrane time constant

$$\tau_\mathrm{m} = \frac{C_\mathrm{m}}{g_\mathrm{l}}. \tag{2.3}$$

For a simpler notation, from here the membrane time constant τ_m is used and the membrane is considered with respect to the leak reversal potential by using $v = V - V_\mathrm{l}$. Then eq. (2.1) is rewritten as

$$\frac{\mathrm{d}v}{\mathrm{d}t} = -\frac{1}{\tau_\mathrm{m}}v + \frac{1}{C_\mathrm{m}}I_\mathrm{syn}. \tag{2.4}$$

The neuron model which is considered in this work uses *current-based* (CUBA) synapses. For those, the synaptic input current of a neuron j can be written as

$$I_{\mathrm{syn},j} = \sum_i w_{ji}\left(\epsilon \star z_i\right), \tag{2.5}$$

where the spikes z_i of pre-synaptic neurons i are convolved with a synaptic kernel ϵ and weighted with w_{ji}. Spikes are considered as instantaneous events triggering a synaptic signal and can be described by δ-distributions. The sequence of spikes from a pre-synaptic neuron i is referred to as a *spike train* and given by

$$z_i(t) = \sum_s \delta\left(t - t_i^s\right). \tag{2.6}$$

The synaptic kernel ϵ is specific to the synapse and models the shape of the synaptic input current. Depending on the degree to which the complexity of synaptic behavior should be modeled, different kernels can be used. In this work, all experiments and simulations are conducted using a single exponential kernel

$$\epsilon(t) = \Theta(t) \exp\left(-\frac{t}{\tau_\mathrm{s}}\right), \tag{2.7}$$

with the Heaviside function $\Theta(t)$ and synaptic time constant τ_s denoting the time scale on which the synaptic current decreases. Examples for other kernels are simple δ-kernels, which have no temporal extent, or a more complex double-exponential kernel, adding a second exponential with different time scale to take into account the finite time it takes for the synaptic signal to arrive.

If the threshold condition and the subsequent triggering of spikes is omitted such that the dynamics of the neuron can evolve freely, the resulting non-spiking neuron model is referred to as a *leaky integrator* (LI).

2.2 Gradient Based Learning in Spiking Neural Networks

Building networks of such neuron models and optimizing them is a challenging but promising task. In comparison to classic ANNs, which disregard any temporal dimension, SNNs contain dynamic states evolving in time and furthermore, due to their event-based inter-neuron communication, incorporate temporal sparsity by design. But exactly this event-based nature introduces disontinuities, which need to be addressed when turning to gradient-based optimization. Based on descriptions in [Billaudelle 2022] and [Goodfellow et al. 2016] I will give a brief overview of gradient based learning on neural networks in general followed by a description on how this is approached in networks consisting of (spiking) neurons, like the LIF neuron.

Gradient Descent Optimization

The aim of models considered in this work is to approximate a function ϕ^*, , e.g. in classification tasks the mapping of an input $\boldsymbol{x}$ to a target $\boldsymbol{y}^*$. The model then can be implemented as a function

$$\boldsymbol{y} = \phi(\boldsymbol{x}, \theta), \tag{2.8}$$

where θ are parameters, for which the aim is to find a set of parameters resulting in the best approximation of ϕ^* by the network ϕ. To evaluate how well the model approximates the true mapping, a (differentiable) *loss function*

$$\mathcal{L}\left(\boldsymbol{y}, \boldsymbol{y}^*, ...\right) \tag{2.9}$$

is defined, assigning a value to the networks output $\boldsymbol{y}$ depending on the target values $\boldsymbol{y}^*$ and optionally other observabes. In *gradient descent* learning, the loss function is intended to be minimized by descending along the slope of its parameter gradient $\mathrm{d}\mathcal{L}/\mathrm{d}\theta$ with the impact of the parameter update defined by a scaling η, referred to as a *learning rate*. The simplest optimization step would be

$$\theta \rightarrow \theta - \eta \frac{\mathrm{d}\mathcal{L}}{\mathrm{d}\theta}. \tag{2.10}$$

For non-convex problems, or loss landscapes respectively, this *gradient descent* might potentially converge to a non-optimal local minimum. To improve convergence, modifications are applied to this simple algorithm, e.g. starting by incorporating a *momentum* to overcome saddle points leading to find potentially more optimal minima.

Feed-Forward Networks

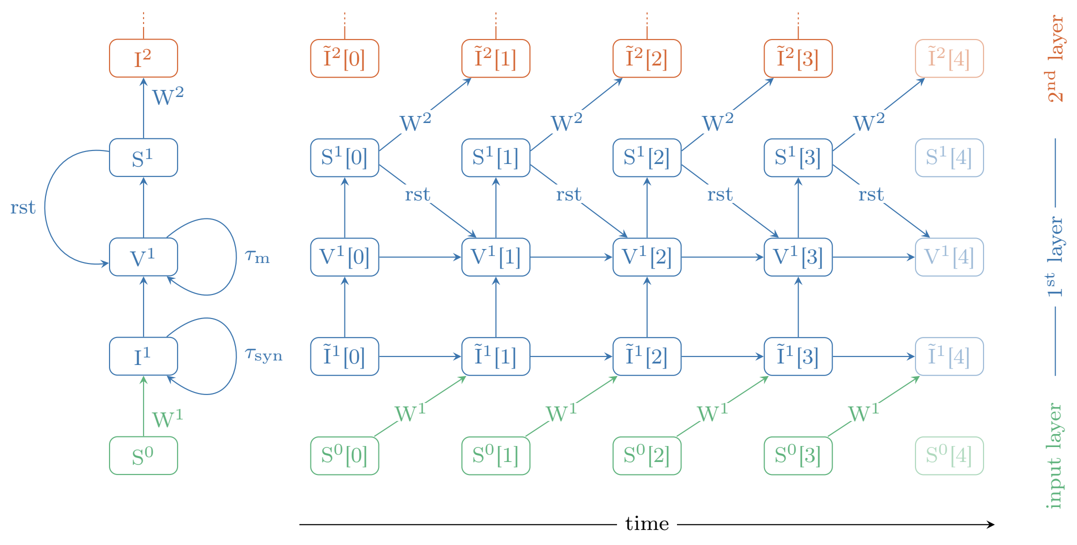
Typically, to accomodate the potentially arbitrary high complexity of nonlinear descision boundaries in a task, networks are built in layers of nonlinear functions ϕ^l with layer-specific parameters θ^l. In a feed-forward network with L layers, e.g., these functions are simply chained together

$$\boldsymbol{y} = \phi^{L-1} \circ ... \circ \phi^l \circ ... \circ \phi^0(\boldsymbol{x}). \tag{2.11}$$

Assuming, that the functions ϕ^l implementing each layer are differentiable with respect to their input and parameters, the gradient $\mathrm{d}\mathcal{L}/\mathrm{d}\theta^l$ can then be computed using the chain rule, which is referred to as *backpropagation*. This chaining of gradients in layered networks addresses and solves the *spatial credit assignment problem*.

Recurrent and Time-Dependent Networks

In SNNs, additionally to the spatial aspect of layers, the temporal dimension of current and membrane dynamics as well dependencies on spiking activity present the challenge of the *temporal credit assignment*. At least in discrete implementations on SNNs, where the dynamics are solved step-wise, these dependencies can be interpreted as recurrencies and solutions can be adopted from the treatment of *recurrent neural networks* (RNNs). There, the network is unrolled along its recurrent (time) dimension and a two-dimensional computational graph arises, in which temporal and spatial connections point only in one direction on their respective axes (cf. fig. 2.2). Traversing the connections in reversed direction along the time dimension then allows to assign temporal credit through *backpropagation through time* (BPTT) [Werbos 1990].



(a) SNN as RNN **(b)** SNN unrolled in time

Figure 2.2: SNN with recurrent feedback through time- and state-dependent dynamics (depending on τ_m and τ_s) and reset ("rst") and two-dimensional graph of time-unrolled SNN. **(a)** The example considers the first two LIF layers of a network and its spike input. The SNN can be interpreted as a RNN. **(b)** Time-discrete SNNs can be unrolled in time to achieve a two-dimensional computational graph, allowing for propagation of gradients also along the reversed time dimension and hence assigning temporal credit. Figures adopted from [Billaudelle 2022].

Surrogate Gradients

As backpropagation can be applied onto time-unrolled SNNs, another issue surfaces as soon as one arrives at the time of the forced reset of the membrane potential V after spike emission. In case of the LIF neuron, the spiking neurons' activation function can be defined using a Heaviside function

$$S(V(t)) = \Theta\left(V(t) - V_\mathrm{th}\right). \tag{2.12}$$

The derivative of the activation function, though, is mostly zero, except at the threshold crossing of the membrane, where it is technically defined as the Dirac delta function δ. To avoid this mostly zero derivative when deriving the weight updates applying the chain rule, a *surrogate gradient* can be introduced [Neftci et al. 2019]. A typical choice, introduced by Zenke and Ganguli [2018], is to approximate the left side of the Heaviside function by the left side of a fast sigmoid

$$\sigma(V) = \frac{\beta(V - V_\mathrm{th})}{1 + \beta\left|V - V_\mathrm{th}\right|} + 1. \tag{2.13}$$

The factor β determines the steepness of the fast sigmoid. The derivative here is

$$\sigma'(V) = \frac{1}{(1 + \beta\,|V - V_{\mathrm{th}}|)^2}. \tag{2.14}$$

When computing parameter gradients in layered networks, the derivative of the activation function is replaced by its surrogate

$$\ldots \cdot \frac{\partial S^l}{\partial V^l} \cdot \ldots \rightarrow \ldots \cdot \sigma'(V^l) \cdot \ldots \tag{2.15}$$

Using surrogate gradients has certainly many advantages, e.g. non-zero gradients are available even in the absence of spikes, allowing to learn from an initially silent network. Nevertheless, gradients here are only approximating the dependence of post-synaptic activity on pre-synaptic spikes and weight. Hence, other approaches emerged, deriving exact closed-form gradient equations or systems of equations, that can be solved analytically of numerically to compute gradients in SNNs. Those other approaches will be described in the following section.

2.3 Exact Gradients for Spiking Point Neurons

Approaches using surrogate gradients for backpropagation of errors often rely on the dense representation of state variables, which, when using analog hardware, e.g., requires sampling of membrane voltages with high enough frequency. As an alternative the EventProp algorithm was proposed by Wunderlich and Pehle [2021], which applies well-known concepts from optimal control theory [Galán et al. 1999] and the study of hybrid dynamical systems to spiking neural networks. For the considered system of LIF neurons, the algorithm provides equations for event-based gradient estimation, only requiring observables of the state variables at spike times. Nevertheless, the algorithm also allows for the choice to incorporate state variables at times other than the spike times. Prior to the above-mentioned work, Göltz et al. [2021] derived closed-form equations for gradients in networks of LIF neurons with CUBA synapses based on first spike times for specific ratios of the synaptic and membane time constants.

2.3.1 Adjoint Dynamics and Transitions

I will go through the derivation of the adjoint equations in a quite general way based on [Pehle 2021, Ch. 3], which for parts of the derivation references [Galán et al. 1999, Gronwall 1919, Rozenvasser 1967].

Consider a system whose state is described by x and the dynamics of this state, given an initial state x_0 at $t = 0$, are governed by the linear ODE

$$f\left(\dot{x}, x, p, t\right) = 0, \tag{2.16}$$

where t is the time and p is a set of parameters. Additionally a jump condition

$$j\big(\dot{x}^-, x^-, p, t\big) = 0 \tag{2.17}$$

is introduced, which, when fulfilled, defines an event time t_k^{post}, where k implies that multiple such events can happen over the course of time. Given some intial conditions x_0, the system is evaluated over time according to f and if the state x of the system fulfills this jump condition j, it undergoes a transition

$$g(\dot{x}^+, x^+, \dot{x}^-, x^-, p, t) = 0. \tag{2.18}$$

The considered loss function consists of two functions $l_\text{p}\left(t^{\text{post}}\right)$ and $l_x\left(x(t), t\right)$ depending on post-synaptic spike times t^{post}, the state variables x and time t. The complete loss is

$$\mathcal{L} = l_\text{p}\left(t^{\text{post}}\right) + \int_0^T l_x\left(x(t), t\right) \mathrm{d}t. \tag{2.19}$$

The system dynamics are added to the loss function as constraints via Lagrange multipliers $\lambda(t)$ and the parameter derivative is taken, leading to the calculations

$$\begin{aligned}
\frac{\mathrm{d}\mathcal{L}}{\mathrm{d}p} &= \frac{\mathrm{d}}{\mathrm{d}p}\left[l_\text{p}\left(t^{\text{post}}\right) + \int_0^T \left[l_x\left(x(t), p, t\right) + \lambda f(\dot{x}, x, p, t)\right] \mathrm{d}t\right] &(2.20)\\
&= \frac{\mathrm{d}}{\mathrm{d}p}\left[l_\text{p}\left(t^{\text{post}}\right) + \sum_{k=0}^{N_\text{post}} \int_{t_k^{\text{post}}}^{t_{k+1}^{\text{post}}} \left[l_x\left(x(t), p, t\right) + \lambda f(\dot{x}, x, p, t)\right] \mathrm{d}t\right] &(2.21)\\
&= \sum_{k=0}^{N_\text{post}} \Bigg[\frac{\partial l_\text{p}}{\partial t_k^{\text{post}}} \frac{\mathrm{d}t_k^{\text{post}}}{\mathrm{d}p} + l_{x,k+1}^- \frac{\mathrm{d}t_{k+1}^{\text{post}}}{\mathrm{d}p} - l_{x,k}^+ \frac{\mathrm{d}t_k^{\text{post}}}{\mathrm{d}p}\\
&\quad + \int_{t_k^{\text{post}}}^{t_{k+1}^{\text{post}}} \left[\frac{\partial l_x}{\partial x}\frac{\partial x}{\partial p} + \frac{\partial l_x}{\partial p} + \lambda\left(\frac{\mathrm{d}}{\mathrm{d}t}\frac{\partial f}{\partial \dot{x}}\frac{\partial x}{\partial p} - \frac{\partial f}{\partial x}\frac{\partial x}{\partial p} - \frac{\partial f}{\partial p}\right)\right] \mathrm{d}t\Bigg], &(2.22)
\end{aligned}$$

where N_post spikes are considered and the additional times $t_0^{\text{post}} = 0$ and $t_{N_\text{post}+1}^{\text{post}} = T$ are introduced. With partial integration

$$\int_{t_k^{\text{post}}}^{t_{k+1}^{\text{post}}} \lambda\left(\frac{\mathrm{d}}{\mathrm{d}t}\frac{\partial f}{\partial \dot{x}}\frac{\partial x}{\partial p}\right) \mathrm{d}t = -\int_{t_k^{\text{post}}}^{t_{k+1}^{\text{post}}} \dot{\lambda}\frac{\partial f}{\partial \dot{x}}\frac{\partial x}{\partial p}\mathrm{d}t + \left[\lambda\frac{\partial f}{\partial \dot{x}}\frac{\partial x}{\partial p}\right]_{t_k^{\text{post}}}^{t_{k+1}^{\text{post}}}, \tag{2.23}$$

the loss gradient is rewritten to

$$\begin{aligned}
\frac{\mathrm{d}\mathcal{L}}{\mathrm{d}p} = \sum_{k=0}^{N_\text{post}} \Bigg[&\int_{t_k^{\text{post}}}^{t_{k+1}^{\text{post}}} \left[\left(\frac{\partial l_x}{\partial x} - \dot{\lambda}\frac{\partial f}{\partial \dot{x}} - \lambda\frac{\partial f}{\partial x}\right)\frac{\partial x}{\partial p} + \frac{\partial l_x}{\partial p} - \lambda\frac{\partial f}{\partial p}\right] \mathrm{d}t\\
&\frac{\partial l_\text{p}}{\partial t_k^{\text{post}}} \frac{\mathrm{d}t_k^{\text{post}}}{\mathrm{d}p} + l_{x,k+1}^- \frac{\mathrm{d}t_{k+1}^{\text{post}}}{\mathrm{d}p} - l_{x,k}^+ \frac{\mathrm{d}t_k^{\text{post}}}{\mathrm{d}p} + \left[\lambda\frac{\partial f}{\partial \dot{x}}\frac{\partial x}{\partial p}\right]_{t_k^{\text{post}}}^{t_{k+1}^{\text{post}}}\Bigg]. \qquad (2.24)
\end{aligned}$$

The dynamics of the Lagrange multipliers, or adjoint variables, are then chosen to be such, that the integrand containing parameter sensitivities $\frac{\partial x}{\partial p}$ in eq. (2.24) vanishes. The time reversed derivative $\frac{\mathrm{d}}{\mathrm{d}t} \to -\frac{\mathrm{d}}{\mathrm{d}t}$ is denoted by $'$ and then the adjoint differential equations are

$$\lambda' \frac{\partial f}{\partial \dot{x}} = \lambda \frac{\partial f}{\partial x} - \frac{\partial l_x}{\partial x}. \tag{2.25}$$

The initial conditions of x are assumed to be parameter independent and the boundary conditions of the adjoint variables are chosen to $\lambda(T) = 0$. This causes the evaluations of the last term in eq. (2.24) at times $t_0^{\mathrm{post}} = 0$ and $t_{N_{\mathrm{post}}+1}^{\mathrm{post}} = T$ to also vanish. The loss gradient then reads

$$\frac{\mathrm{d}\mathcal{L}}{\mathrm{d}p} = \sum_{k=0}^{N_{\mathrm{post}}} \int_{t_k^{\mathrm{post}}}^{t_{k+1}^{\mathrm{post}}} \left[\frac{\partial l_x}{\partial p} - \lambda \frac{\partial f}{\partial p} \right] \mathrm{d}t + \sum_{k=1}^{N_{\mathrm{post}}} \xi_k, \tag{2.26}$$

$$\xi_k = \frac{\partial l_{\mathrm{p}}}{\partial t_k^{\mathrm{post}}} \frac{\mathrm{d}t_k^{\mathrm{post}}}{\mathrm{d}p} + \left(l_{x,k}^- - l_{x,k}^+ \right) \frac{\mathrm{d}t_k^{\mathrm{post}}}{\mathrm{d}p} - \lambda^+ \frac{\partial f^+}{\partial \dot{x}} \frac{\partial x^+}{\partial p} + \lambda^- \frac{\partial f^-}{\partial \dot{x}} \frac{\partial x^-}{\partial p}. \tag{2.27}$$

The parameter derivative $\mathrm{d}_p j$ of the tranistion condition is used to relate the paramater sensitivity $\mathrm{d}_p t_k^{\mathrm{post}}$ of the k-th spike time to the parameter sensitivity $\partial_p x^-$ of the state before the jump

$$\begin{aligned} \mathrm{d}_p t_k^{\mathrm{post}} &= -\frac{\left(\partial_{x^-} j - \partial_{\dot{x}^-} j \left(\partial_{\dot{x}} f^- \right)^{-1} \partial_x f^- \right) \partial_p x^- + \partial_{\dot{x}^-} j \partial_p f^- + \partial_p j}{\partial_{x^-} j \dot{x}^- + \partial_t j - \partial_{\dot{x}^-} j \left(\partial_{\dot{x}} f^- \right)^{-1} \left(\partial_x f^- \dot{x}^- + \partial_t f^- \right)} && (2.28) \\ &= A \partial_p x^- + B. && (2.29) \end{aligned}$$

The parameter derivative $\mathrm{d}_p g$ of the jump is

$$\begin{aligned} 0 = \mathrm{d}_p g = &\left(\partial_{x^+} g - \partial_{\dot{x}^+} g \left(\partial_{\dot{x}} f^+ \right)^{-1} \partial_x f^+ \right) \partial_p x^+ \\ &+ \left(\partial_{x^+} g \dot{x}^+ - \partial_{\dot{x}^+} g \left(\partial_{\dot{x}} f^+ \right)^{-1} \partial_x f^+ \dot{x}^+ \right) \mathrm{d}_p t_k^{\mathrm{post}} \\ &+ \left(\partial_{x^-} g - \partial_{\dot{x}^-} g \left(\partial_{\dot{x}} f^- \right)^{-1} \partial_x f^- \right) \left(\partial_p x^- + \dot{x}^- \mathrm{d}_p t_k^{\mathrm{post}} \right) \\ &+ \left(\partial_t g - \partial_{\dot{x}^+} g \left(\partial_{\dot{x}} f^+ \right)^{-1} \partial_t f^+ - \partial_{\dot{x}^-} g \left(\partial_{\dot{x}} f^- \right)^{-1} \partial_t f^- \right) \mathrm{d}_p t_k^{\mathrm{post}} \\ &- \partial_{\dot{x}^+} g \left(\partial_{\dot{x}} f^+ \right)^{-1} \partial_p f^+ - \partial_{\dot{x}^-} g \left(\partial_{\dot{x}} f^- \right)^{-1} \partial_p f^- + \partial_p g. \end{aligned} \tag{2.30}$$

This, together with eq. (2.29), is then used to relate the sensitivities $\partial_p x^-$ before and $\partial_p x^+$ after the jump

$$\partial_p x^+ = \hat{A}\partial_p x^- + \hat{B} \tag{2.31}$$

$$\text{with } \hat{A} = D^{-1}\left(CA + \partial_{\dot{x}^-} g \left(\partial_{\dot{x}} f^-\right)^{-1} \partial_x f^- + \partial_{x^-} g\right), \tag{2.32}$$

$$\hat{B} = D^{-1}\left(CB + \partial_{\dot{x}^+} g \left(\partial_{\dot{x}} f^+\right)^{-1} \partial_p f^+ + \partial_{\dot{x}^-} g \left(\partial_{\dot{x}} f^-\right)^{-1} \partial_p f^- + \partial_p g\right) \tag{2.33}$$

$$\begin{aligned} C = &-\partial_{\dot{x}^+} g \left(\partial_{\dot{x}} f^+\right)^{-1} \left(\partial_x f^+ \dot{x}^+ + \partial_t f^+\right) + \partial_{x^+} g \dot{x}^+ \\ &- \partial_{\dot{x}^-} g \left(\partial_{\dot{x}} f^-\right)^{-1} \left(\partial_x f^- \dot{x}^- + \partial_t f^-\right) + \partial_{x^-} g \dot{x}^- + \partial_t g \end{aligned} \tag{2.34}$$

$$D = \partial_{\dot{x}^+} g \left(\partial_{\dot{x}} f^+\right)^{-1} \partial_x f^+ - \partial_{x^+} g \tag{2.35}$$

Equations (2.29) and (2.31) are now used to replace the corresponding terms in eq. (2.27), resulting in the gradient contribution

$$\begin{aligned} \xi_k = &\left(\partial_{t_k^{\mathrm{post}}} l_{\mathrm{p}} + l_{x,k}^- - l_{x,k}^+\right) B - \lambda^+ \partial_{\dot{x}} f^+ \hat{B} \\ &+ \left[\left(\partial_{t_k^{\mathrm{post}}} l_{\mathrm{p}} + l_{x,k}^- - l_{x,k}^+\right) A - \lambda^+ \partial_{\dot{x}} f^+ \hat{A} + \lambda^- \partial_{\dot{x}} f^-\right] \partial_p x^-. \end{aligned} \tag{2.36}$$

In order for the parameter gradient of the loss to be independent of the parameter sensitivity of the system state, the adjoint variables are demanded to undergo the transition

$$\lambda^- = \left(\lambda^+ \partial_{\dot{x}} f^+ \hat{A} - \left(\partial_{t_k^{\mathrm{post}}} l_{\mathrm{p}} + l_{x,k}^- - l_{x,k}^+\right) A\right) \left(\partial_{\dot{x}} f^-\right)^{-1}, \tag{2.37}$$

and the gradient contribution therefore simplifies further to

$$\xi_k = \left(\partial_{t_k^{\mathrm{post}}} l_{\mathrm{p}} + l_{x,k}^- - l_{x,k}^+\right) B - \lambda^+ \partial_{\dot{x}} f^+ \hat{B}. \tag{2.38}$$

2.3.2 EventProp

The EventProp algorithm, as first presented by Wunderlich and Pehle [2021], considers LIF neurons with CUBA, single-exponential synapses. Instead of explicitly giving an expression for the synaptic currents, one can consider the additional differential equation

$$\frac{\mathrm{d}I}{\mathrm{d}t} = -\frac{1}{\tau_{\mathrm{s}}} I, \tag{2.39}$$

where $I = \frac{1}{g_{\mathrm{l}}} I_{\mathrm{syn}}$ for easier readability. Considering a network of N neurons, the state variables are $x^\top = (v_0, I_0, ..., v_{N-1}, I_{N-1})$ and the equatios 2.4 and 2.39 can be combined into a system of $2N$ linear differential equations

$$f(\dot{x}, x, p) = \left(\mathbb{1}_N \otimes \begin{bmatrix} \tau_{\mathrm{m}} & 0 \\ 0 & \tau_{\mathrm{s}} \end{bmatrix}\right) \dot{x} - \left(\mathbb{1}_N \otimes \begin{bmatrix} -1 & 1 \\ 0 & -1 \end{bmatrix}\right) x = 0, \tag{2.40}$$

where the Kronecker product $\otimes$ is used. If the membrane potential v_n of neuron n reaches the threshold $\vartheta = V_{\text{reset}} - V_{\text{l}}$, expressed in the transition condition

$$j(x^-, p) = \left(e_n^\top \otimes [1, 0]\right) x^- - \vartheta = 0, \tag{2.41}$$

the the state variables are subject to jumps given by

$$g(x^+, x^-, p) = x^+ - \left(\mathbb{1}_{2N} - P_n^\top T P_n\right) x^- - p_n = 0, \tag{2.42}$$

$$\text{with } P_n = e_n^\top \otimes \mathbb{1}_2,\ T = \begin{bmatrix} 1 & 0 \\ 0 & 0 \end{bmatrix},\ p_n = (W e_n) \otimes \begin{bmatrix} 0 \\ 1 \end{bmatrix} + e_n \otimes \begin{bmatrix} V_{\text{reset}} - V_{\text{l}} \\ 0 \end{bmatrix}, \tag{2.43}$$

where e_n denotes the N-dimensional, n-th Cartesian unit vector. Note that the transformation (2.42) consist of a simple linear transformation and a translation.

The adjoint dynamical system with variables $\lambda = (\lambda_{v,0}, \lambda_{I,0}, ..., \lambda_{v,N-1}, \lambda_{I,N-1})$ is given by inserting eq. (2.40) into eq. (2.25) and reads

$$\lambda' \partial_{\dot{x}} f = \lambda \partial_x f - \partial_x l_x. \tag{2.44}$$

The dynamics in between the times of jumps only couple membrane v and current I of individual neurons, and therefore, the adjoint dynamics behave equivalently. Then, for a single neuron with adjoints λ_v and λ_I given a loss l_V depending on the membrane trace, but not the current, the dynamics are

$$\tau_{\text{m}} \lambda_v' = -\lambda_v - \partial_v l_V, \tag{2.45}$$
$$\tau_{\text{s}} \lambda_I' = -\lambda_I + \lambda_v. \tag{2.46}$$

The loss with respect to a synaptic weight w_{ji} can be obtained from eq. (2.26) using the contributions at spike times in eq. (2.38). Considering the dependencies of dynamics $f(\dot{x}, x, p)$, transition condition $j(x^-, p)$, and jump $g(x^+, x^-, p)$, the weight gradient then is

$$\frac{\mathrm{d}\mathcal{L}}{\mathrm{d}w_{ji}} = \sum_{k=1}^{N_{\text{post}}} \lambda^+ \partial_{\dot{x}} f^+ \partial_{w_{ji}} g \tag{2.47}$$
$$= -\tau_{\text{s}} \sum_{\text{spikes from } i} \lambda_{I,j}^+ \tag{2.48}$$

From eq. (2.37) the jumps in the adjoint variables to

$$\begin{aligned}\lambda^- = \lambda^+ \partial_{\dot{x}} f^+ &\left(\left(\dot{x}^+ + \partial_{x^-} g \dot{x}^-\right) \frac{\partial_{x^-} j}{\partial_{x^-} j \dot{x}^-} - \partial_{x^-} g \right) \left(\partial_{\dot{x}} f^-\right)^{-1} \\ &+ \left(\partial_{t_k^{\text{post}}} l_{\text{p}} + l_{x,k}^- - l_{x,k}^+\right) \frac{\partial_{x^-} j}{\partial_{x^-} j \dot{x}^-} \left(\partial_{\dot{x}} f^-\right)^{-1}.\end{aligned} \tag{2.49}$$

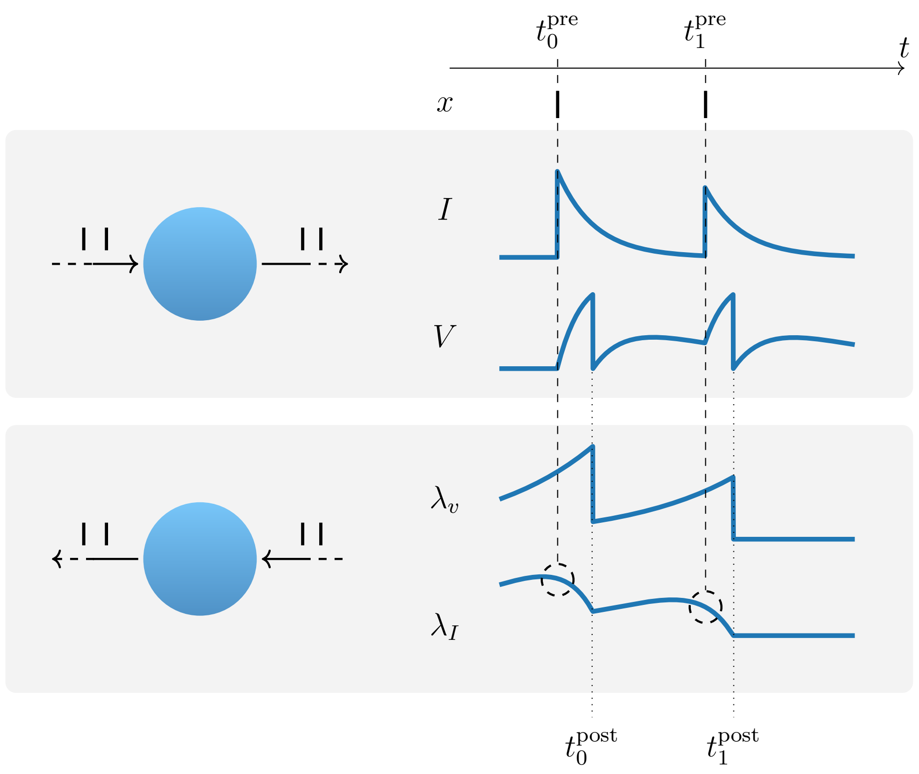


Figure 2.3: Forward and adjoint (backward) dynamics of a LIF neuron receiving two input spikes at t_1^{pre} and t_2^{pre}. The membrane potential experiences two jumps, giving the post-synaptic spike times t_1^{post} and t_2^{post}. In the computation of the adjoint dynamics, which happens backwards in time, the adjoint variable λ_v experiences jumps at the two post-synaptic spike times. The gradient contributions are then computed by sampling the adjoint state λ_I at the respective pre-synaptic spike times, indicated by dashed circles.

Looking at eq. (2.41), the term $\partial_{x^-} j$ is a $2N$-sized row-vector with almost all entries being 0 except for the $2n(k)$-th entry, where $n(k)$ is the index of the neuron that emitted the spike at t_k^{post}. Also, the term $\partial_{x^-} g$ is a diagonal matrix with -1 in all diagonal entries except for the $2n(k)$-th, where it is 0. With those two considerations, it's then quite clear that only the n-th neuron's adjoint variable $\lambda_{v,n}$ regarding its membrane undergoes a jumps. All other adjoint variables stay the same, which one can write as

$$\lambda^-_{I,n(k)} = \lambda^+_{I,n(k)}, \tag{2.50}$$

$$\forall m \neq n(k) :\ \lambda^-_{v,m} = \lambda^+_{v,m},\ \lambda^-_{I,m} = \lambda^+_{I,m}. \tag{2.51}$$

The jump in $\lambda_{v,n(k)}$ is given by

$$\lambda_{v,n}^{-} = \left(\left(\lambda^{+}\partial_{\dot{x}} f^{+}\left(\dot{x}^{+} + \partial_{x^{-}} g\dot{x}^{-}\right)\right)_n + \left(\partial_{t_k^{\text{post}}} l_{\text{p}} + l_{x,k}^{-} - l_{x,k}^{+}\right)\right)_n \frac{1}{\tau_{\text{m}}(\dot{v}^{-})_n} \tag{2.52}$$

$$= \frac{(\dot{v}^{+})_n}{(\dot{v}^{-})_n}\lambda_{v,n}^{+} + \frac{1}{\tau_{\text{m}}(\dot{v}^{-})_n}\left(\sum_{m=0}^{N-1}\left(\lambda_{v,m}^{+} - \lambda_{I,m}^{+}\right) w_{mn} + \partial_{t_k^{\text{post}}} l_{\text{p}} + l_{x,k}^{-} - l_{x,k}^{+}\right). \tag{2.53}$$

This set of equations allows to compute exact weight gradients (cf. eq. (2.48)) in SNNs of LIF neurons by solving the adjoint dynamics in eqs. (2.45) and (2.46) while considering the jumps in eq. (2.53). Using the same approach, similar sets of equations could be derived for gradients with respect to time constants or other neuron parameters.

2.3.3 Closed-Form Equations in Special Cases

The work of Göltz et al. [2021] considers the cases of $\tau_{\text{m}} \to \infty$, $\tau_{\text{m}} = \tau_{\text{s}}$ and $\tau_{\text{m}} = 2\tau_{\text{s}}$ and they derive explicit gradient expressions by studying the membrane of a single LIF neuron and its jump condition. The derived gradient expressions and application of those onto learning with SNNs will be referred to as “Fast And Deep” (or “F&D” in short) throughout this thesis.

In all cases, the input current

$$I(t) = \sum_i w_i \exp\left(-\frac{t - t_i}{\tau_{\text{s}}}\right) \tag{2.54}$$

is considered, where the spike times t_i are treated independent of their source neuron and have an associated weight w_i. In the mentioned special cases, the first spike time of a neuron can be derived explicitely and from there, gradients with respect to weights and pre-synaptic spike times can be calculated.

Learning rule for $\tau_{\text{m}} = \tau_{\text{s}}$

The membrane voltage in this case follows

$$v(t) = \frac{1}{\tau_{\text{s}}}\sum_i w_i \Theta\left(t - t_i\right)\left(t - t_i\right)\exp\left(-\frac{t - t_i}{\tau_{\text{s}}}\right), \tag{2.55}$$

and the post-synaptic spike time t^{post} is defined through the jump condition $v(t^{\text{post}}) = \vartheta$. Solving this jump condition for t^{post} yields

$$t^{\text{post}} = \frac{\tau_{\text{s}} b}{a_1} - \tau_{\text{s}}\mathcal{W}\left(-\frac{\vartheta}{a_1}\exp\left(\frac{b}{a_1}\right)\right), \tag{2.56}$$

with the Lambert W function $\mathcal{W}$, evaluated on its branch $\mathcal{W}(x) > -1$, and the expressions

$$a_n := \sum_{k \in C} w_k \exp\left(\frac{t_k}{n\tau_\mathrm{s}}\right), \tag{2.57}$$

$$b := \sum_{k \in C} w_k \frac{t_k}{\tau_\mathrm{s}} \exp\left(\frac{t_k}{\tau_\mathrm{s}}\right), \tag{2.58}$$

where $C = \{k | t_k < t^\mathrm{post}\}$ is the set of causal pre-synaptic spikes. Introducing the variable $z := -\frac{\vartheta}{a_1} \exp\left(\frac{b}{a_1}\right)$ to abbreviate the argument of the Lambert W function, the gradients of the first spike time are

$$\frac{\partial t^\mathrm{post}}{\partial w_i} = -\frac{1}{a_1} \frac{t^\mathrm{post} - t_i}{\mathcal{W}(z) + 1} \exp\left(\frac{t_i}{\tau_\mathrm{s}}\right), \tag{2.59}$$

$$\frac{\partial t^\mathrm{post}}{\partial t_i} = -\frac{w_i}{\tau_\mathrm{s} a_1} \frac{t^\mathrm{post} - t_i - \tau_\mathrm{s}}{\mathcal{W}(z) + 1} \exp\left(\frac{t_i}{\tau_\mathrm{s}}\right). \tag{2.60}$$

Learning rule for $\tau_\mathrm{m} = 2\tau_\mathrm{s}$

The membrane voltage in this case is

$$v(t) = \sum_i w_i \Theta\left(t - t_i\right) \kappa\left(t - t_i\right), \tag{2.61}$$

$$\kappa(t) = \exp\left(-\frac{t}{2\tau_\mathrm{s}}\right) - \exp\left(-\frac{t}{\tau_\mathrm{s}}\right). \tag{2.62}$$

Solving the jump condition for t^post and utilizing a_1 and a_2 as defined in eq. (2.57), along with $x := \sqrt{a_2^2 - 4\vartheta a_1}$, gives

$$t^\mathrm{post} = 2\tau_\mathrm{s} \ln\left(\frac{2a_1}{a_2 + x}\right). \tag{2.63}$$

The gradients then become

$$\frac{\partial t^\mathrm{post}}{\partial w_i} = \frac{2\tau_\mathrm{s}}{a_1}\left[1 + \frac{\vartheta}{x}\exp\left(\frac{t^\mathrm{post}}{2\tau_\mathrm{s}}\right)\right]\exp\left(\frac{t_i}{\tau_\mathrm{s}}\right) - \frac{2\tau_\mathrm{s}}{x}\exp\left(\frac{t_i}{2\tau_\mathrm{s}}\right), \tag{2.64}$$

$$\frac{\partial t^\mathrm{post}}{\partial t_i} = \frac{2w_i}{a_1}\left[1 + \frac{\vartheta}{x}\exp\left(\frac{t^\mathrm{post}}{2\tau_\mathrm{s}}\right)\right]\exp\left(\frac{t_i}{\tau_\mathrm{s}}\right) - \frac{w_i}{x}\exp\left(\frac{t_i}{2\tau_\mathrm{s}}\right). \tag{2.65}$$

The gradients in eqs. (2.59), (2.60), (2.64) and (2.65) can be used to compute errors in layered networks. Since gradient expressions are available for weights and pre-synaptic spike times, the error information can be backpropagated in a layer-wise fashion by applying the chain rule. They also succesfully apply this gradient estimation using analytical expressions to hardware ITL training on the BSS-2 platform (cf. section 2.4). The only constraint here, is that neurons in hidden layers can spike only once, since gradients are not derived for later spike times. But as the authors in Göltz et al. [2021] mention, and as was recently published by Bacho and Chu [2022], the approach can be extended to neurons emitting multiple spikes.

2.4 BrainScaleS-2

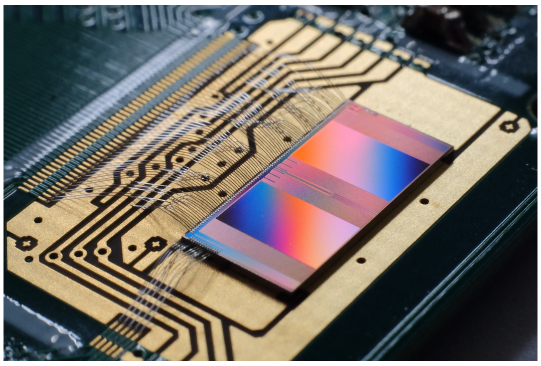
The BSS-2 system is a mixed-signal neuromorphic accelerator for SNNs [Pehle et al. 2022]. The BSS-2 *application-specific integrated circuit* (ASIC) (full chip specifier: HICANN-DLS-SR-HX v3), has an *analog network core* to emulate the dynamics of neurons and synapses in continuous time and surrounding digital circuitry communicating and handling spike events and configuration data. The neuron and synapse circuits in the system allow for modeling inspired by neuroscience. Through *field-programmable gate arrays* (FPGAs), a host computer can control and configure the system and execute experiments. A set of specially designed hardware abstraction layers allows the user to describe experiments in software at a high level and eliminate the need to manually write low-level instructions for host-chip and on-chip communication. Extensions have also been developed for common frameworks like PyNN [Davison et al. 2009] and PyTorch [Paszke et al. 2019], which allow non-expert users to use the chip more easily. This chapter delves into the architecture of the BSS-2 system, the capabilities of the ASIC, and the various software tools and frameworks developed to facilitate experimentation with the analog neuromorphic substrate.

2.4.1 System Setup

The BSS-2 chip, depicted in fig. 2.4, comprises 512 neuron and 131.072 synapse circuits split into two hemispheres. The substrate allows physical emulation of neuron dynamics with an approximately 1000-fold acceleration compared to the biological time domain. All spike events, either external or ones generated by neuron circuits during emulation are handled digitally by the event handling block. They are collected and can be sent off the chip via the FPGA to the host, recording those spike observables, or routed to post-synaptic neuron circuits on the chip via synapse drivers. Additionally to the spike events, membrane voltages of neurons can be accessed and measured parallely by *columnar analog-to-digital converters* (CADCs).

The synapses are divided into four quadrants, and two of these quadrants are present in each hemisphere. Each quadrant contains a synapse array, made up of 256 rows and 128 columns. The synaptic weights are represented by 6 bit values stored in static random-access memory and modulate the height of the pulse signal triggered by an event transmitted along a synapse to a neuron. BSS-2 features both current-based (CUBA) and conductance-based (COBA) synapses. The synapse driver in BSS-2 is organized such that all connections along a synapse row have the same effect on the neurons' membranes, either excitatory or inhibitory. Other important data sources are the correlation sensors, measuring the correlation between pre- and post-synaptic spikes, the spike counters, and the membrane analog-to-digital converter, allowing to sample the membrane of a single neuron with high resolution.

The neuron circuits on BSS-2 implement the AdEx neuron model [Brette and



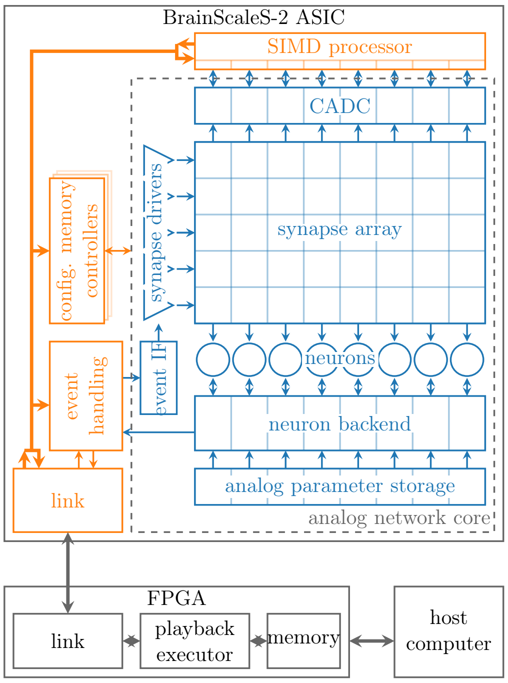


Figure 2.4: Left: Close-up image of BSS-2 chip bonded to its carrier board. Image taken from [Müller et al. 2022]. **Right:** Schematics of signal flow on BSS-2. The schematic of a single hemisphere outlines the analog network core and the single instruction, multiple data processor. The real chip has two hemispheres with 256 neurons each, mirrored vertically along the neurons. Spike events are processed and routed on-chip posing as input to post-synaptic neuron circuits or sent off-chip via the FPGA to the host for recording. Figure taken from [Müller et al. 2022].

Gerstner 2005]

$$C_m \dot{V} = -g_{\mathrm{l}} \left(V - V_{\mathrm{l}}\right) + g_{\mathrm{l}} \Delta_{\mathrm{T}} \exp\left(\frac{V - V_{\mathrm{T}}}{\Delta_{\mathrm{T}}}\right) - w + I, \tag{2.66}$$

$$\tau_w \dot{w} = a \left(V - V_{\mathrm{l}}\right) - w. \tag{2.67}$$

Other than the LIF neuron model (cf. eq. (2.1)), this model additionally includes an exponential feedback term and an adaptation term. The exponential feedback is defined by its slope Δ_{T}, and soft threshold V_{T} and mimics the depolarization in the neuron's action potential. The adaptation term w acts as an additional current flowing off the membrane V and has its own dynamic, allowing the neuron to adapt to afferent activations and its own activity. The adaptation dynamics are defined by the decay time constant τ_w and the adaptation strength a.

Parameters stored in the analog parameter storage allow custom control over the conductances and potentials in each neuron circuit. Therefore, those parameters can

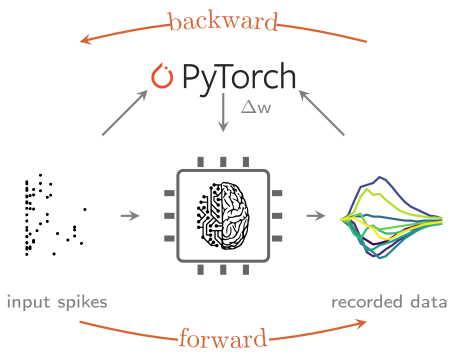


Figure 2.5: Illustration of *in-the-loop* (ITL) training with BSS-2. Forward execution happens on BSS-2 and a host computer models the dynamics to build a computational graph. The hardware observables are used to compute gradients in the model and update hardware weights accordingly. Figure adapted from [Cramer et al. 2022].

be calibrated so that each neuron's membrane dynamic on the BSS-2 chip shows the desired behavior. To model the LIF neuron, the exponential term and adaptation current (cf. eq. (2.66)) can be switched off. A comparator detecting threshold crossings of the membrane triggers spike emissions. Switching off this comparator allows the emulation of non-spiking neurons.

2.4.2 Software Framework

The software stack of the BSS-2 ecosystem allows users to describe experiments in a high-level fashion and translates them to an equivalent experiment configuration to be executed on the BSS-2 hardware. This translation is done by converting the high-level description to a signal-flow graph description and applying a place-and-route algorithm to convert the graph to a valid hardware configuration. This allows users without expert knowledge to execute experiments on the hardware substrate, provide input stimuli and record back observables [Müller et al. 2022].

To calibrate the dynamics on hardware to a set of model parameters defined by a user, the calibration library `calix` is provided. This library allows the user to inject a set of target parameters (e.g., for neuron time constants, leak, reset, and threshold potentials) and returns a calibration data set, which provides operation point settings for each circuit.

To facilitate the modeling of SNNs on BSS-2 the `hxtorch.snn` software library [Spilger et al. 2022] was designed with a frontend built upon the PyTorch framework. `hxtorch.snn` enables users to describe a SNN model using custom modules for synapse and neuron layers corresponding to hardware entities. The library structure utilizes PyTorch's model construction interface while separating the construction from its execution on the hardware. This allows to build the computational graph, enabling to backpropagate errors and hence optimizing networks of SNNs, but it uses the hardware observables by forward execution on BSS-2 (cf. fig. 2.5).

This separation is achieved by providing an instance into which each module invocation is registered. If executing on hardware, invocations of modules' forward calls are done on a promise handle, which is filled with the corresponding data after hardware execution. The library also allows evaluating models in software, for which handles are directly filled with corresponding data and invocations operate on the data directly. Building a layered network and execution can be done according to the following exemplary structure:

```
instance = hxtorch.snn.Instance()

synapse_ih = hxtorch.snn.Synapse(n_h, n_i, instance, ...)
neuron_h = hxtorch.snn.Neuron(n_h, instance, ...)
synapse_ho = hxtorch.snn.Synapse(n_o, n_h, instance, ...)
neuron_o = hxtorch.snn.ReadoutNeuron(n_o, instance, ...)

input_handle = hxtorch.snn.NeuronHandle(input)

sh0 = synapse_ih(input_handle)
nh0 = neuron_h(h1)
sh1 = synapse_ho(h2)
nh1 = neuron_o(h3)

hxtorch.snn.run(instance, ...)
```

After the `hxtorch.snn.run()` call, when using the BSS-2 chip for forward execution, the `NeuronHandles` `nh0` and `nh1` are filled with the hardware observables, which for spiking neurons are the spikes and if enabled the CADC membrane measurements, for non-spiking neurons only the latter. The `SynapseHandles` `sh0` and `sh1` are filled with the multiplication result of the weights and the input spikes, or observed spikes respectively.

Gradient-based optimization on the parameters is enabled by equipping the modules with PyTorch-differentiable functions [Paszke et al. 2017] that are either provided directly as an `pytorch.autograd.Function` or through a function that defines a simulated forward pass implicitly containing the backward pass. The built-in functions have an additional function argument through which hardware observables, if provided, are handled, allowing for seamless backpropagation of gradients based on those hardware observations. Additionally, users can provide custom functions implementing individual handling of hardware observables and associated gradient estimation algorithms.

At the time of the experiments conducted in this thesis, the `hxtorch.snn` library supported LIF (`hxtorch.snn.Neuron`) and LI (`hxtorch.snn.ReadoutNeuron`) neuron layers. Additionally, `hxtorch.snn` includes a dropout module that applies a batch-wise spiking mask to a preceding neuron layer, disabling the spike output on hardware accordingly. This provides a typical machine learning functionality to users.

Remark on Module and API Changes

Due to the `hxtorch.snn` library being just recently developed, active work is still being conducted on it, and additional functionalities are planned to be incorporated successively. Therefore, the *application programming interface* (API) and underlying software might be subject to modification, and even in the time between recording the experiments for this thesis and writing it, the library has undergone some changes. The software state used for experiments conducted for this thesis is listed in appendix F.

3 Extensions on EventProp

The adjoint sensitivity analysis, as described in section 2.3, is quite general in its form and can be applied to numerous systems with discontinuous jumps, especially to neuron models. In the case of the EventProp algorithm the adjoint formalism was used to derive an exact gradient estimation method for the LIF neuron model (cf. section 2.3.2). In this chapter, I use the same reasoning to derive the equations of the adjoint dynamical system, its jumps and the weight gradients in the case of the LIF neuron model including a refractory period. The aim here is to show how the adjoint formalism can be applied to other systems and derive parameter gradients.

I will also go through the derivation of explicit expressions for weight gradients of the first spike time of a LIF neuron by solving the ODEs in EventProp and show that those results are equal to the expressions derived by Göltz et al. [2021]. This is to show the similarity of the methods, even though they are derived in different ways, and to justify why I use the analytical expressions later to check my implementation of the EventProp algorithm.

3.1 Refractory Leaky-Integrate and Fire Neuron

The refractory LIF model extends the LIF model considered in EventProp by including a refractory period during which the neuron cannot fire again, even if the incoming current exceeds the threshold. This models the period of time during which a biological neuron is unresponsive to incoming stimuli after firing an action potential and thus incorporates this important, biological property. By deriving the adjoint equations for a system of refractory LIF neurons, I complement the work of Wunderlich and Pehle [2021] by applying the methods used therein, described in section 2.3 of this thesis.

The dynamics of a single refractory LIF neuron n can be described by

$$\begin{bmatrix} \tau_\mathrm{m}\dot{v}_n \\ \tau_\mathrm{s}\dot{I}_n \\ \tau_\mathrm{r}\dot{z}_n \end{bmatrix} = \begin{bmatrix} -\Theta(-z_n) & \Theta(-z_n) & 0 \\ 0 & -1 & 0 \\ 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} v_n \\ I_n \\ z_n \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \\ -1 \end{bmatrix} \tag{3.1}$$

Compared to eq. (2.40) the additional variable $z(t)$ is introduced to measure the refractory period of a neuron. In this model, after a neuron spikes, its membrane stays constant at the reset potential over the refractory time period τ_r. After this period, the membrane follows the usual dynamics again and therefore only then is able to cross the threshold and spike again.

I consider a system of N neurons with state $x = (v_0, I_0, z_0, ..., v_{N-1}, I_{N-1}, z_{N-1})$. The differential equation describing the dynamics of the complete system can be written as

$$f(\dot{x}, x, p, t) = M_1\dot{x} + M_2(z)x + b_r, \tag{3.2}$$

$$\text{with } M_1 = \mathbb{1}_N \otimes \begin{bmatrix} \tau_\mathrm{m} & 0 & 0 \\ 0 & \tau_\mathrm{s} & 0 \\ 0 & 0 & \tau_\mathrm{r} \end{bmatrix}, \tag{3.3}$$

$$M_2(z) = \sum_{n=0}^{N-1} P_n^\top \begin{bmatrix} -\Theta(-z_n) & \Theta(-z_n) & 0 \\ 0 & -1 & 0 \\ 0 & 0 & 0 \end{bmatrix} P_n, \tag{3.4}$$

$$b_\mathrm{r} = \sum_{n=0}^{N-1} e_n \otimes \begin{bmatrix} 0 \\ 0 \\ -1 \end{bmatrix} \tag{3.5}$$

where I use $z = (z_0, ..., z_{N-1})$ to refer to only the refractory state variables. The jump condition for the neuron model I consider here is

$$j(x^-, p) = \left(e_n^\top \otimes [1, 0, 0]\right) x^- - \vartheta = 0, \tag{3.6}$$

with the threshold $\vartheta = V_\mathrm{th} - V_\mathrm{l}$ for the relative membrane voltage $v(t) = V(t) - V_\mathrm{l}$. If a neuron n fulfills the transition condition j, the state variables jump according to

$$g(x^+, x^-, p) = x^+ - \left(\mathbb{1}_{2N} - P_n^\top T P_n\right) x^- - p_n = 0, \tag{3.7}$$

$$\text{with } P_n = e_n^\top \otimes \mathbb{1}_3,\ T = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix},\ p_n = (W e_n) \otimes \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix} + e_n \otimes \begin{bmatrix} V_\mathrm{reset} - V_\mathrm{l} \\ 0 \\ 1 \end{bmatrix}. \tag{3.8}$$

The dynamics of the adjoint dynamics of a single refractory LIF neuron follow from eq. (2.25) and are

$$\tau_\mathrm{m}\lambda_v' = -\lambda_v\Theta(-z), \tag{3.9}$$
$$\tau_\mathrm{s}\lambda_I' = \lambda_v\Theta(-z) - \lambda_I, \tag{3.10}$$
$$\tau_\mathrm{r}\lambda_z' = -\left(-v + I\right)\lambda_v\delta(-z), \tag{3.11}$$

where the Dirac delta distribution $-\delta(-z)$ arises by taking the derivative of the Heaviside step function $\Theta(-z)$.

The weight gradient of the loss is the same as in eq. (2.48),

$$\frac{\mathrm{d}\mathcal{L}}{\mathrm{d}w_{ji}} = -\tau_\mathrm{s} \sum_{\text{spikes from } i} \lambda_{I,j}^+. \tag{3.12}$$

The jumps in the adjoint variables are quite similar to the ones in section 2.3.2 and the derivation follows the same pattern. Due to the same dependencies of dynamics

f, transition condition j and jumps g, the general form is the same as in eq. (2.49). But the term $\partial_{x^-} g$ here is a diagonal matrix with -1 in all diagonal corresponding to the membrane $v_{n(k)}$ and refractory state $z_{n(k)}$ of the n-th neuron, where it is zero. Hence, all adjoint variables except for $\lambda_{v,n(k)}$ and $\lambda_{z,n(k)}$ stay the same

$$\lambda^-_{I,n(k)} = \lambda^+_{I,n(k)}, \tag{3.13}$$

$$\forall m \neq n(k) : \; \lambda^-_{v,m} = \lambda^+_{v,m}, \; \lambda^-_{I,m} = \lambda^+_{I,m}, \lambda^-_{z,m} = \lambda^+_{z,m}. \tag{3.14}$$

The adjoint of the refractory state is simply reset to 0 at the spike time t^{post}_k

$$\lambda^-_{z,n} = 0. \tag{3.15}$$

To derive the jump of $\lambda_{v,n}$, I apply the same steps as in eqs. (2.52) and (2.53). This then results in

$$\lambda^-_{v,n} = \frac{1}{\tau_\text{m}(\dot{v}^-)_n}\left(\sum_{m=0}^{N-1}\left(\lambda^+_{v,m}\Theta(-z^+_m) - \lambda^+_{I,m}\right) w_{mn} - \lambda^+_{z,n} + \partial_{t^{\text{post}}_k} l_\text{p} + l^-_{x,k} - l^+_{x,k}\right). \tag{3.16}$$

The forward and adjoint dynamics of a refractory LIF neuron are illustrated in fig. 3.1. In the forward direction, the refractory state z holds the membrane at a reset value until the refractory period ends. As can be seen from eqs. (3.11) and (3.16), the adjoint of the refratory state λ_z stores the value $v - I$, which equals $-\tau_\text{m}\dot{v}$, after the end of the refractory period and enters into λ_v at the spike time. This is also visible in the illustration of the adjoint traces in fig. 3.1.

In the limit $\tau_\text{r} \to 0$, the refractory LIF converges against and ultimately becomes the non-refractory LIF neuron model. Therefore, the adjoint equations and gradients contributions should also become similar. This can easily be shown by taking this limit for the refratory adjoint state, leading to

$$\lim_{\tau_\text{r}\to 0} \lambda_z(t^{\text{post}} + \tau_\text{r}) = -\tau_\text{m} \lim_{\tau_\text{r}\to 0} \dot{v}(t^{\text{post}} - \tau_\text{r}) = -\tau_\text{m} \lim_{t\to t^{\text{post}}+} \dot{v}(t) = -\tau_\text{m}\dot{v}^+. \tag{3.17}$$

Injecting this into eq. (3.16) the adjoint of the refractory state can be dropped and the jumps become similar to the ones in EventProp in eq. (2.53), making the methods similar in the limit $\tau_\text{r} \to 0$.

Although the extension here is relatively modest, since the equations describing the refractory LIF neuron are not too different from the non-refractory LIF neuron, the steps taken and the equations derived show that the adjoint formalism can be applied to neuron models other than a purely linear system of ODEs.

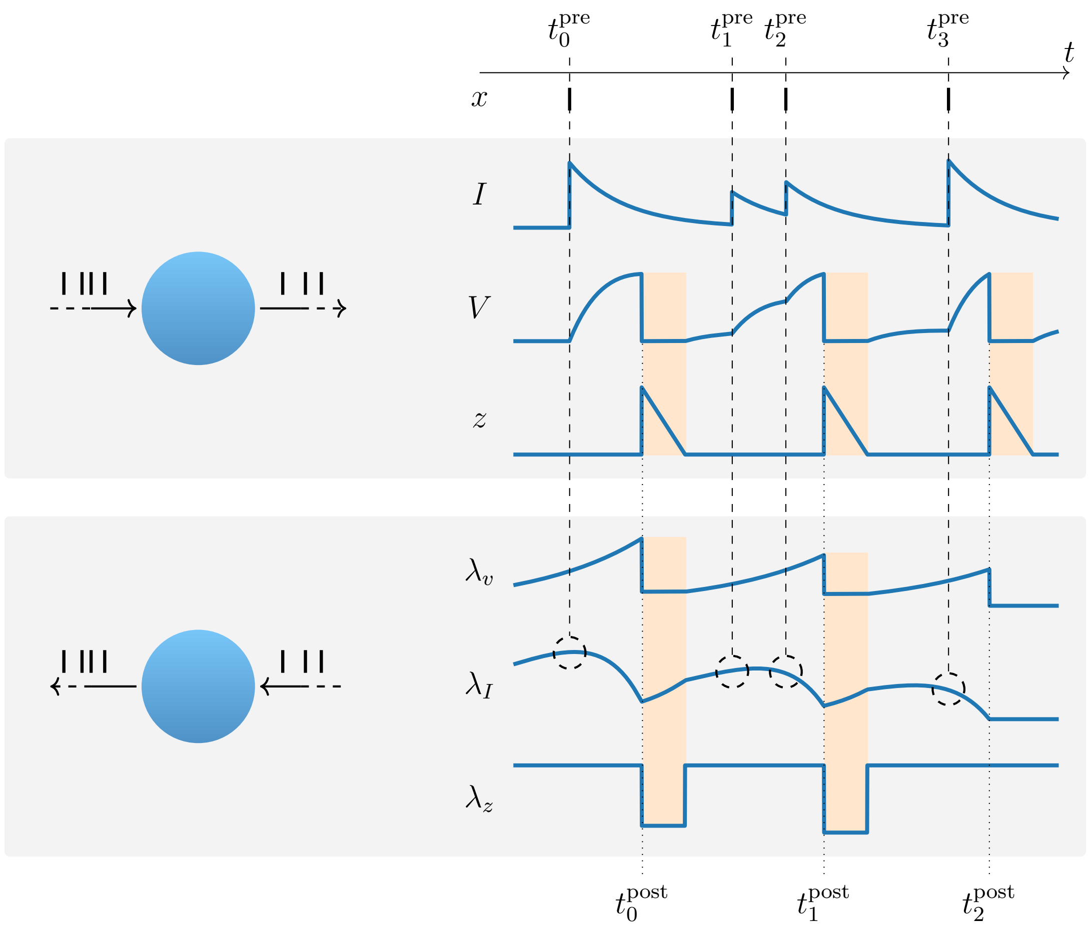


Figure 3.1: Forward and adjoint (backward) dynamics of a refractory LIF neuron receiving four input spikes. The neuron emits three spikes, after which its membrane voltage is refractory for a certain time period. This refractory period enters also into the dynamics and jumps of the adjoint variables. The adjoint variable λ_z of the refratory state effectively stores information during this period and enters into λ_v at the spike time, cf. eqs. (3.11) and (3.16). Similar to the non-refractory LIF neuron, the gradient contributions are computed by sampling the adjoint state λ_I at pre-synaptic spike times.

3.2 Explicit Gradient Expressions From EventProp

As was described in section 2.3.3, for special cases explicit expressions can be calculated for the first spike time of a neuron and its derivatives with respect to pre-synaptic spike times or the associated synaptic weights. When considering the same starting point of a single neuron receiving pre-synaptic spikes at times t_i with weights w_i, the gradient of the first post-synaptic spike time with respect to the weight can also be calculated explicitely without restricting the choice of time constants.

Here, I derive the explicit expression of the weight gradient from integrating the adjoint dynamics and explicitely injecting the jumps of the adjoint variables. I also show that the result equals the expressions derived by Göltz et al. [2021] for the cases where $\tau_\mathrm{m} = \tau_\mathrm{s}$ and $\tau_\mathrm{m} = 2\tau_\mathrm{s}$.

General Weight Gradient

I consider a neuron emitting only a single spike at t^post, and since the gradient $\partial_{w_i} t^\mathrm{post}$ is desired, choose $\mathcal{L} = l_\mathrm{p} = t^\mathrm{post}$ (cf. eq. (2.19)). The jump in the adjoint variable λ_v at the post-synaptic spike time, given in eq. (2.53), then simply becomes

$$\lambda_v^- \left(t^\mathrm{post}\right) = \frac{1}{\tau_\mathrm{m} \dot{v}^- \left(t^\mathrm{post}\right)}, \tag{3.18}$$

where the exponent "$-$" denotes the left-hand limit. Injecting this into the dynamics eqs. (2.45) and (2.46), gives

$$\lambda_v(t) = \frac{1}{\tau_\mathrm{m} \dot{v}^- \left(t^\mathrm{post}\right)} \exp\left(-\frac{t^\mathrm{post} - t}{\tau_\mathrm{m}}\right). \tag{3.19}$$

$$\lambda_I(t) = \frac{1}{\tau_\mathrm{m} \dot{v}^- \left(t^\mathrm{post}\right)} \frac{\tau_\mathrm{m}}{\tau_\mathrm{m} - \tau_\mathrm{s}} \left[\exp\left(-\frac{t^\mathrm{post} - t}{\tau_\mathrm{m}}\right) - \exp\left(-\frac{t^\mathrm{post} - t}{\tau_\mathrm{s}}\right)\right], \tag{3.20}$$

$$= \frac{1}{\dot{v}^- \left(t^\mathrm{post}\right)} \frac{1}{\tau_\mathrm{m} - \tau_\mathrm{s}} \kappa\left(t^\mathrm{post} - t\right), \tag{3.21}$$

where $\kappa(t) \coloneqq \exp(-t/\tau_\mathrm{m}) - \exp(-t/\tau_\mathrm{s})$ from eq. (2.62) is used. Using the expression of the membrane voltage $v(t)$ in eq. (2.61) and the definition of the set of causal pre-synaptic spikes $C = \{k \,|\, t_k < t^\mathrm{post}\}$, the term $\dot{v}^- \left(t^\mathrm{post}\right)$ can be written as

$$\dot{v}^- \left(t^\mathrm{post}\right) = \frac{\tau_\mathrm{s}}{\tau_\mathrm{m} - \tau_\mathrm{s}} \sum_{k \in C} w_k \left[-\frac{1}{\tau_\mathrm{m}} \exp\left(-\frac{t^\mathrm{post} - t_k}{\tau_\mathrm{m}}\right) + \frac{1}{\tau_\mathrm{s}} \exp\left(-\frac{t^\mathrm{post} - t_k}{\tau_\mathrm{s}}\right)\right] \tag{3.22}$$

$$= \frac{\tau_\mathrm{s}}{\tau_\mathrm{m} - \tau_\mathrm{s}} \sum_{k \in C} w_k \dot{\kappa}\left(t^\mathrm{post} - t_k\right). \tag{3.23}$$

Equation (3.21) can then be rewritten to depend only on the causal pre-synaptic spike times t_i and their associated weights w_i, the post-synaptic spike time t^post, the time constants τ_m and τ_s and used to compute the gradient of t^post from eq. (2.48), ultimately resulting in

$$\frac{\mathrm{d}t^\mathrm{post}}{\mathrm{d}w_i} = -\tau_\mathrm{s}\lambda_I\left(t_i\right) \tag{3.24}$$

$$= -\tau_\mathrm{s}\left[\frac{1}{\tau_\mathrm{m}-\tau_\mathrm{s}}\kappa\left(t^\mathrm{post}-t_i\right)\right]\left[\frac{\tau_\mathrm{s}}{\tau_\mathrm{m}-\tau_\mathrm{s}}\sum_{k\in C} w_k\dot{\kappa}\left(t^\mathrm{post}-t_k\right)\right]^{-1}. \tag{3.25}$$

Following, I will show that this gradient is equal to the gradients in “Fast And Deep” in the cases $\tau_\mathrm{m} = 2\tau_\mathrm{s}$ and $\tau_\mathrm{m} = \tau_\mathrm{s}$.

Weight Gradient for $\tau_\mathrm{m} = 2\tau_\mathrm{s}$

To show that eq. (3.25) and eq. (2.64) are equal, I insert the constraint $\tau_\mathrm{m} = 2\tau_\mathrm{s}$ into the former equation and separate into terms proportional to $\exp\left(t_i/\tau_\mathrm{s}\right)$ and $\exp\left(t_i/(2\tau_\mathrm{s})\right)$. To The calculations are

$$\frac{\mathrm{d}t^\mathrm{post}}{\mathrm{d}w_i} = -\left[\exp\left(-\frac{t^\mathrm{post}}{2\tau_\mathrm{s}}+\frac{t_i}{2\tau_\mathrm{s}}\right)-\exp\left(-\frac{t^\mathrm{post}}{\tau_\mathrm{s}}+\frac{t_i}{\tau_\mathrm{s}}\right)\right] \times\left[\sum_{k\in C} w_k\dot{\kappa}\left(t^\mathrm{post}-t_k\right)\right]^{-1}, \tag{3.26}$$

$$= \left[-\exp\left(-\frac{t^\mathrm{post}}{2\tau_\mathrm{s}}+\frac{t_i}{2\tau_\mathrm{s}}\right)+\exp\left(-\frac{t^\mathrm{post}}{\tau_\mathrm{s}}+\frac{t_i}{\tau_\mathrm{s}}\right)\right] \times\left[\frac{1}{\tau_\mathrm{s}}\exp\left(-\frac{t^\mathrm{post}}{\tau_\mathrm{s}}\right)a_1-\frac{1}{2\tau_\mathrm{s}}\exp\left(-\frac{t^\mathrm{post}}{2\tau_\mathrm{s}}\right)a_2\right]^{-1}, \tag{3.27}$$

$$= A_1\exp\left(-\frac{t_i}{\tau_\mathrm{s}}\right)+A_2\exp\left(-\frac{t_i}{2\tau_\mathrm{s}}\right), \tag{3.28}$$

$$\text{with}\quad A_1 = 2\tau_\mathrm{s}\left[2a_1-\exp\left(\frac{t^\mathrm{post}}{2\tau_\mathrm{s}}\right)a_2\right]^{-1} = \frac{2\tau_\mathrm{s}}{x}\exp\left(-\frac{t^\mathrm{post}}{2\tau_\mathrm{s}}\right), \tag{3.29}$$

$$A_2 = -2\tau_\mathrm{s}\left[2\exp\left(\frac{-t^\mathrm{post}}{2\tau_\mathrm{s}}\right)a_1-a_2\right]^{-1} = -\frac{2\tau_\mathrm{s}}{x}. \tag{3.30}$$

Th factors in front of the exponentials with t_i from eq. (2.64) can now be compared to A_1 and A_2 and shown that they are equal. For A_2 this can directly be seen, while for A_1 and the factor in front of $\exp\left(t_i/\tau_\mathrm{s}\right)$ the steps to show equality, using eqs. (2.57) and (2.63), are

$$\frac{2\tau_\mathrm{s}}{a_1}\left[1+\frac{\vartheta}{x}\exp\left(\frac{t^\mathrm{post}}{2\tau_\mathrm{s}}\right)\right] \tag{3.31}$$

$$= \frac{2\tau_\mathrm{s}}{a_1 x}\left[x+\vartheta\exp\left(\frac{t^\mathrm{post}}{2\tau_\mathrm{s}}\right)\right] \tag{3.32}$$

$$
\begin{aligned}
&= \frac{2\tau_\mathrm{s}}{a_1 x}\left[2a_1 \exp\left(-\frac{t^\mathrm{post}}{2\tau_\mathrm{s}}\right) - a_2\right. \\
&\quad \left. + \left(\exp\left(-\frac{t^\mathrm{post}}{2\tau_\mathrm{s}}\right) a_2 - \exp\left(-\frac{t^\mathrm{post}}{\tau_\mathrm{s}}\right) a_1\right) \exp\left(\frac{t^\mathrm{post}}{2\tau_\mathrm{s}}\right)\right] && (3.33) \\
&= \frac{2\tau_\mathrm{s}}{a_1 x}\left[a_1 \exp\left(-\frac{t^\mathrm{post}}{2\tau_\mathrm{s}}\right)\right] && (3.34) \\
&= 2\tau_\mathrm{s}\left[x \exp\left(\frac{t^\mathrm{post}}{2\tau_\mathrm{s}}\right)\right]^{-1} && (3.35) \\
&= 2\tau_\mathrm{s}\left[2a_1 - \exp\left(-\frac{t^\mathrm{post}}{2\tau_\mathrm{s}}\right) a_2\right]^{-1} = A_1. && (3.36)
\end{aligned}
$$

The weight gradient in eq. (3.25) for the case of $\tau_\mathrm{m} = 2\tau_\mathrm{s}$ therefore equals the expression in eq. (2.64) of "Fast And Deep".

Weight Gradient for $\tau_\mathrm{m} = \tau_\mathrm{s}$

In the case of equal time constants, the limit $\tau_\mathrm{m} \to \tau_\mathrm{s}$ has to be taken in the gradient in eq. (3.25). This can be done for the terms with κ and $\dot{\kappa}$ seperately, which are

$$
\begin{aligned}
\lim_{\tau_\mathrm{m}\to\tau_\mathrm{s}} \frac{1}{\tau_\mathrm{m} - \tau_\mathrm{s}} \kappa(t) &= \frac{t}{\tau_\mathrm{s}^2} \exp\left(-\frac{t}{\tau_\mathrm{s}}\right), && (3.37) \\
\lim_{\tau_\mathrm{m}\to\tau_\mathrm{s}} \frac{\tau_\mathrm{s}}{\tau_\mathrm{m} - \tau_\mathrm{s}} \dot{\kappa}(t) &= \frac{1}{\tau_\mathrm{s}}\left(1 - \frac{t}{\tau_\mathrm{s}}\right) \exp\left(-\frac{t}{\tau_\mathrm{s}}\right). && (3.38)
\end{aligned}
$$

The gradient of the spike time can be rewritten and exponential terms with t_i (or t_k inside the sum) can be separated from exponentials with t^post to be able to use the definitions of a_1 and b as in eqs. (2.57) and (2.58). Using eq. (2.56) to relate spike time t^post and the Lambert W function $\mathcal{W}(z)$ will ultimately show that the gradient here is the same as in eq. (2.59). The calculations are

$$
\begin{aligned}
\frac{\mathrm{d}t^\mathrm{post}}{\mathrm{d}w_i} &= -\tau_\mathrm{s}\left[\frac{t^\mathrm{post} - t_i}{\tau_\mathrm{s}^2} \exp\left(-\frac{t^\mathrm{post}}{\tau_\mathrm{s}}\right) \exp\left(\frac{t_i}{\tau_\mathrm{s}}\right)\right] \\
&\quad \times \left[\sum_{k\in C} w_k \frac{1}{\tau_\mathrm{s}}\left(1 - \frac{t^\mathrm{post}}{\tau_\mathrm{s}} + \frac{t_k}{\tau_\mathrm{s}}\right) \exp\left(-\frac{t^\mathrm{post}}{\tau_\mathrm{s}}\right) \exp\left(-\frac{t_i}{\tau_\mathrm{s}}\right)\right]^{-1} && (3.39) \\
&= -\left[\frac{t^\mathrm{post} - t_i}{\tau_\mathrm{s}} \exp\left(\frac{t_i}{\tau_\mathrm{s}}\right)\right]\left[\frac{1}{\tau_\mathrm{s}}\left(a_1 - \frac{t^\mathrm{post}}{\tau_\mathrm{s}} a_1 + b\right)\right]^{-1} && (3.40) \\
&= -\left[\left(t^\mathrm{post} - t_i\right) \exp\left(\frac{t_i}{\tau_\mathrm{s}}\right)\right]\left[a_1\left(1 + \mathcal{W}(z)\right)\right]^{-1}. && (3.41)
\end{aligned}
$$

Equation (3.41), the explicit weight gradient derived starting from EventProp, is the same as the expression for equal time constants $\tau_\mathrm{m} = \tau_\mathrm{s}$ in eq. (2.59) from "Fast And Deep".

The calculations above demonstrate that an explicit expression for the weight gradient can be obtained from the EventProp formalism [Wunderlich and Pehle 2021], matching the expressions derived by Göltz et al. [2021] for the special cases considered there. This result is, of course, expected since both algorithms analytically derive gradients for the same system, only that EventProp provides a system of equations that still needs to be solved. Nevertheless, this shows that starting from EventProp, explicit gradients can be formulated, which only depend on pre- and post-synaptic spike times, weights, and time constants, while not needing to restrict the choice of those time constants. The same can be expected for gradients concerning pre-synaptic spike times since the method described in section 2.3 allows the derivation of an analogous system of equations from which explicit expressions can also be determined.

4 Supporting Software Implementations

The experiments in this thesis were done using `hxtorch.snn` [Spilger et al. 2022] and the BSS-2 ASIC. The `hxtorch.snn` library allows to evaluate a network of neurons either completely by simulating it in software or by emulating it on the ASIC. For experiments optimizing on or computing gradients with respect to spike times, I contributed a spike time decoder function to the `hxtorch.snn` software stack, to enable using this across experiments and beyond the scope of this work. I also added two methods that aim to align observables and parameters in simulation with execution on BSS-2, specifically a software to hardware weight conversion and a method to measure time domain misalignment between spike and membrane measurements.

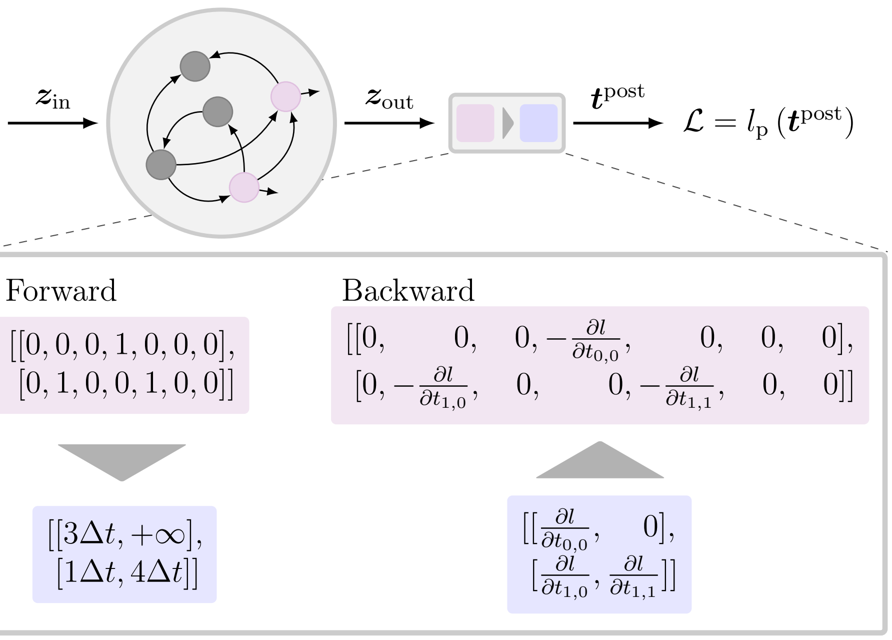
4.1 Spike Time Decoder

A key advantage of the EventProp algorithm compared to the way surrogate gradient approaches are commonly used is the possibility to train networks in a purely spike-based manner without recording membrane traces. To include spike time into the loss calculations, like the loss term l_p in eq. (2.19) allows, the spike times need to be extracted from the binary tensors holding spikes as 1s and otherwise 0s. To then optimize on such losses, the backpropagation of gradients through this conversion needs to be defined in some sensible way. Figure 4.1 shows an overview of the decoder's functionality and how it handles the backpropagation of gradients. I will describe the implementation below and show experiments using this decoder in section 5.2.

To illustrate the functionality of the spike-time decoder, consider an experiment with N_T time steps size Δt. The time-dependent variables and observables are then given on or mapped to a discrete time grid of length N_T and the total time of the experiment is $T = N_T \Delta t$. The output of a simulated SNN or hardware measurements from BSS-2 mapped to this discrete time grid is a binary spike tensor holding 1s at the index corresponding to the time at which this spike occurred. I implemented the decoder such that it converts this tensor $\boldsymbol{z}$ to a tensor $\boldsymbol{t}$ holding the spike times, i.e. the index of the spike along the time dimension times Δt. An example spike train and the output of the decoder would then be

$$\text{Forward:} \qquad \boldsymbol{z} = (0, 0, \underset{z_0}{1}, 0, 0, \underset{z_1}{1}, 0) \quad \rightarrow \quad \boldsymbol{t} = (t_0, t_1) = (2\Delta t, 5\Delta t)\,. \tag{4.1}$$

In most tasks there are multiple output neurons N_n which might emit different numbers of spikes each. Spike-time-based classification strategies either rely on the timing of only the first spike, cf. [Mostafa 2017, Göltz et al. 2021], or the first K



z_{in}
z_{out}
t^{post}
$\mathcal{L} = l_{\text{p}}\left(t^{\text{post}}\right)$
Forward
$[[0,0,0,1,0,0,0], [0,1,0,0,1,0,0]]$
$[[3\Delta t, +\infty], [1\Delta t, 4\Delta t]]$
Backward
$[[0, 0, 0, -\frac{\partial l}{\partial t_{0,0}}, 0, 0, 0], [0, -\frac{\partial l}{\partial t_{1,0}}, 0, 0, -\frac{\partial l}{\partial t_{1,1}}, 0, 0]]$
$[[\frac{\partial l}{\partial t_{0,0}}, 0], [\frac{\partial l}{\partial t_{1,0}}, \frac{\partial l}{\partial t_{1,1}}]]$

Figure 4.1: Illustration of spike time decoder with exemplary forward and backward translation. The binary spike tensor $\boldsymbol{z}_{out}$ is converted into floating point spike times $\boldsymbol{t}^{\text{post}}$. The custom PyTorch autograd function enables backpropagation of gradients into the network for loss functions depending on spike times. The decoder is exemplified using a time grid of size $N_{\text{T}} = 7$ with time step width Δt and $N_{\text{n}} = 2$ output neurons. The returned tensor is padded with $+\infty$ for neurons emitting less spikes than others. The backward pass injects gradients with respect to spike times, where $t_{n,k}$ marks the k-th spike time of neuron n, at their initial positions in the binary spike tensors.

spikes. Therefore, the decoder allows for the specification of the desired number K of spikes to retrieve. If a neuron spikes less often than K-times, the spike times are written in ascending order and the remaining entries are filled with floating point $+\infty$.

The desired scheme to backpropagate gradients of a loss l_{p} with respect to the spike times t_k, retrieved according to eq. (4.1), is to inject those gradients $\partial_{t_k} l_{\text{p}}$ at the place of the corresponding spikes z_k. The propagation of gradients then follows

$$\text{Backward:} \quad \partial_{\boldsymbol{t}} l_{\text{p}} = (\partial_{t_0} l_{\text{p}}, \partial_{t_1} l_{\text{p}}) \quad \rightarrow \quad \partial_{\boldsymbol{z}} l_{\text{p}} = (0, 0, -\partial_{t_0} l_{\text{p}}, 0, 0, -\partial_{t_1} l_{\text{p}}, 0) \tag{4.2}$$

The negative sign is necessary to obtain the correct gradients, as is later demonstrated for the novel EventProp implementation and the already available surrogate gradient implementation (cf. fig. C.1). A related description of this was also outlined by Billaudelle [2022].

I implemented the backpropagation of gradients in the case of multiple spikes such that gradients propagate only for finite times and only for spike entries of $\boldsymbol{z}$ decoded

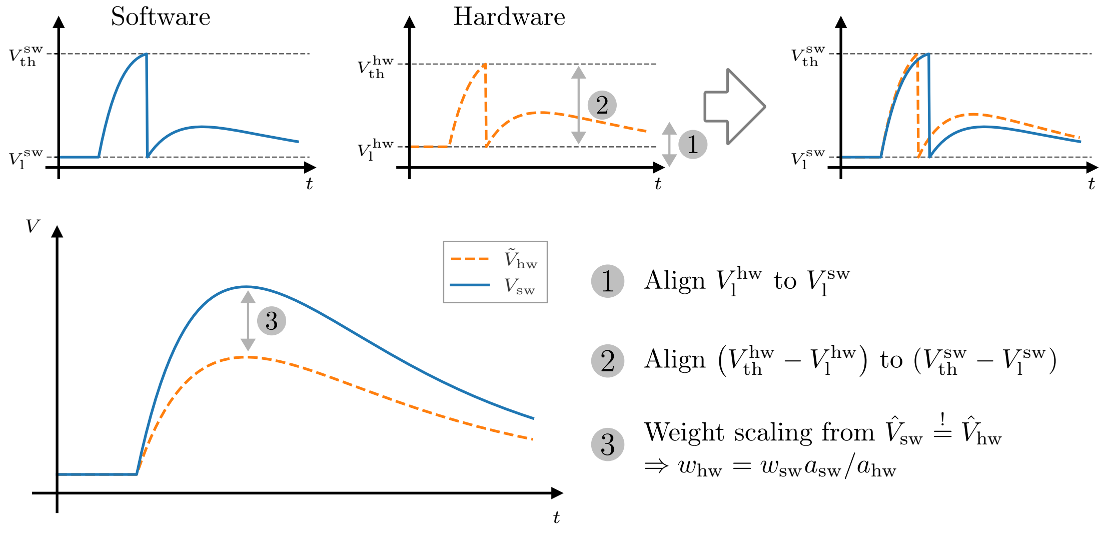

Software
Hardware
$V_{\mathrm{th}}^{\mathrm{sw}}$
$V_{\mathrm{l}}^{\mathrm{sw}}$
$V_{\mathrm{th}}^{\mathrm{hw}}$
$V_{\mathrm{l}}^{\mathrm{hw}}$
t
V
$\tilde{V}_{\mathrm{hw}}$
V_{sw}
1 Align $V_{\mathrm{l}}^{\mathrm{hw}}$ to $V_{\mathrm{l}}^{\mathrm{sw}}$
2 Align $\left(V_{\mathrm{th}}^{\mathrm{hw}} - V_{\mathrm{l}}^{\mathrm{hw}}\right)$ to $\left(V_{\mathrm{th}}^{\mathrm{sw}} - V_{\mathrm{l}}^{\mathrm{sw}}\right)$
3 Weight scaling from $\hat{V}_{\mathrm{sw}} \overset{!}{=} \hat{V}_{\mathrm{hw}}$
$\Rightarrow w_{\mathrm{hw}} = w_{\mathrm{sw}} a_{\mathrm{sw}} / a_{\mathrm{hw}}$

Figure 4.2: Illustration of the steps involved in converting software weights to hardware weights. First, the measured hardware data is offset to align the leak potential $V_{\mathrm{l}}^{\mathrm{hw}}$ with the software value. Next, the data is scaled to match the difference between the leak potential and threshold $V_{\mathrm{th}} - V_{\mathrm{l}}$. To determine the proportionality factor between the weight and the amplitude of the *post-synaptic potential* (PSP) ($\hat{V} = \max_t \left(\mathrm{sign}\left(w\right) V\left(t\right)\right)$), a non-spiking neuron receiving a single input signal with weight w is used in both simulation and hardware. By considering weights w along the entire hardware range of BSS-2, the factor $a_{\mathrm{sw}}/a_{\mathrm{hw}}$ is obtained from the linear relation $\hat{V} = aw$. This factor is necessary to observe the expected dynamics on hardware for a given set of software parameters and applied calibration.

into a spike time. If, for example, a neuron emits more than the K specified spikes, the neglected spikes are not used during loss calculation and therefore the gradient with respect to those is simply zero.

4.2 Software to Hardware Weight Conversion

The two relevant observables from the BrainScaleS-2 analog substrate are the membrane traces $V_{\mathrm{hw}}(t)$ and the emitted spikes $z_{\mathrm{hw}}(t)$, which can be processed into PyTorch tensors by interpolating them onto the discrete time grid chosen for the software representation of variables and observables. The membrane dynamics, specifically with regard to their weight dependence, are influenced by the selected hardware calibration. However, when training using hardware ITL, it is important to ensure that the parameters' effect on the dynamics of the physical substrate matches the simulated dynamics. This is necessary to enable a reasonable estimation of the gradient during the training process. Therefore, the weights specified in software need to be translated correctly to hardware weights on BSS-2. I have implemented such a translation method for CUBA synapses and make it available for general use by adding it to the `hxtorch.snn` software stack.

Instead of using explicit conversion relationships between the software parameters (e.g. $V_{\mathrm{th}}^{\mathrm{sw}}$) and the values chosen in calibration for the hardware, I adopted a more pragmatic approach that aligns the observations of the substrate with the expected dynamics. This approach is schematically represented in Figure 4.2. This has the advantage that an ideal behavior of the components is not essential, so that systematic errors in the components can be tolerated to a certain extent.

After normalizing the membrane measurements, the height of the PSP is compared to the expected value in simulation. Since the amplitude of the PSP is proportional to the synaptic weight by

$$\max_t \left(\mathrm{sign}\left(w\right) V(t)\right) = aw, \tag{4.3}$$

the factor needed to achieve analogous behaviour on hardware can be extracted from demanding

$$\begin{aligned} \max_t \left(\mathrm{sign}\left(w_{\mathrm{sw}}\right) V_{\mathrm{sw}}(t)\right) &\overset{!}{=} \max_t \left(\mathrm{sign}\left(w_{\mathrm{hw}}\right) V_{\mathrm{hw}}(t)\right) && (4.4)\\ \Leftrightarrow \quad w_{\mathrm{hw}} &\overset{!}{=} \frac{a_{\mathrm{sw}}}{a_{\mathrm{hw}}} w_{\mathrm{sw}}, && (4.5) \end{aligned}$$

where a_{sw}, or a_{hw} respectively, is the proportionality factor between the amplitude of the PSP and the weight w in software, or on BSS-2 respectively.

4.3 Time Domain Alignment

On BSS-2, the CADC samples of membrane voltages are processed by the on-chip processors, which is not synchronized to the system time which is used to assign time stamps to events. However, when using voltage-dependent loss functions in Event-Prop or when using surrogate gradient methods, it is crucial for the the membrane traces and spike trains to be aligned to ensure correct temporal credit assignment. The required precision of this alignment in different learning algorithms or its impact on overall network performance remains subject to further studies.

I contributed such a method to the `hxtorch.snn` software stack, allowing users to quantify the misalignment of observed membrane traces and spike trains. The method uses the simple setup of a non-spiking neuron receiving an input spike at time t_{pre} through a synapse with maximum weight. The unprocessed membrane measurements, combining CADC values and corresponding time stamps, are used to fit a PSP

$$V_{\mathrm{fit}}(t) = a_{\mathrm{fit}} \Theta\left(t - t_{\mathrm{fit}}\right) \left(\exp\left(-\frac{t - t_{\mathrm{fit}}}{\tau_{\mathrm{m}}}\right) - \exp\left(-\frac{t - t_{\mathrm{fit}}}{\tau_{\mathrm{s}}}\right)\right) + b_{\mathrm{fit}}, \tag{4.6}$$

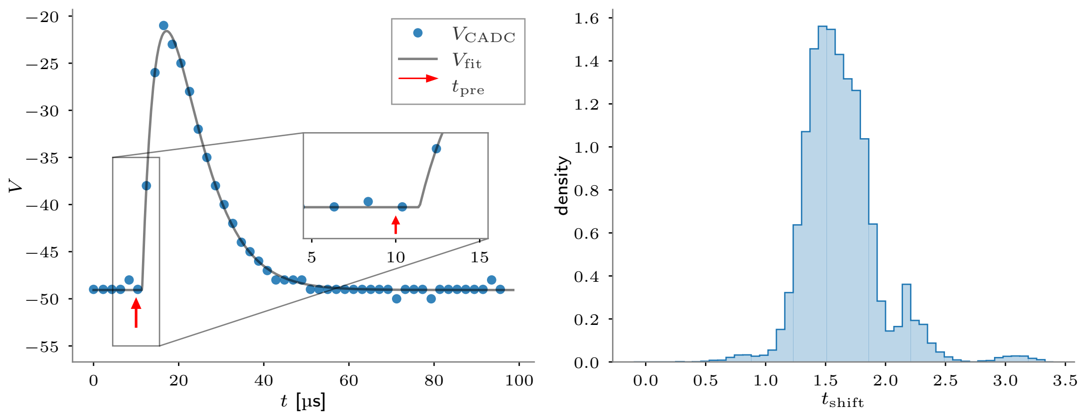


Figure 4.3: PSP fit on example CADC membrane measurements (left) and distribution of the time shift between membrane and spike observables on BSS-2 (right). An input spike is sent at $t_{\text{pre}} = 10\,\mu\text{s}$ (red arrow) and t_{shift} is obtained by comparing the input spike time to the time of the membrane voltage rise observed on BSS-2 by fitting a PSP to the measured data.

with the fit parameter of interest being the time shift t_{fit}. The other parameters a_{fit} and b_{fit} are used to make the function invariant to the scale and shift the measurements. The choice of fitting a function allows for precision below the sampling rate rather than detecting the rising edge by the pairwise comparison of samples.

The parameter t_{fit} in eq. (4.6) represents the time at which the input spike arrives at the neuron and its membrane rises. The time t_{fit} should be the same as that of the input spike t_{pre}, which is specified by the user. The method I implemented, allows to measure the difference of those times for all neurons on BSS-2 over multiple runs with batches of equal inputs. From this, the time shift can be extracted in each single case or averaged over all runs, batched inputs and neurons to obtain a average shift t_{shift}.

An example of such a fit to the CADC measurements and the distribution of resulting time shifts t_{shift} are displayed in fig. 4.3. The difference of the input spike time and the rise of the membrane is visible and over 5 batches of size 10 and all 512 neurons the shift is $t_{\text{shift}} = (1.62 \pm 0.31)\,\mu\text{s}$.

An important remark about this method is the assumption of exact timing of the input peaks without any fluctuations. This is probably not the case, but I have assumed that the fluctuations are small enough compared to a shift in the range of µs. To quantify this further, in future work multiple input spikes could be considered in a single trace and their spacing crosschecked.

Nevertheless, this utility allows users to quantifiy the misalignment of data along the time domain and give the opportunity to align the observables. In my experiments on the Yin-Yang dataset, described below in chapter 5, I used a voltage-based loss and therefore applied this method to measure the time shift and adjust the output traces used for training.

5 EventProp with BrainScaleS-2

Even though the BSS-2 platform allows to sample the membrane potentials of neurons over the whole experiment time window, the EventProp algorithm is able the compute gradients based on observables at spike times only. This makes it a desirable alternative alongside surrogate gradients to optimize and study SNNs, especially when training with hardware ITL.

I will describe the implementation of the algorithm using custom PyTorch autograd functions, in order to integrate EventProp as a gradient estimation method in `hxtorch.snn`, and show how the estimated gradient compares to the exact gradient in a special case by using formulas derived by Göltz et al. [2021] for "Fast And Deep". Then I present the training results achieved on the Yin-Yang task [Kriener et al. 2022] for training with EventProp in software as well as using BSS-2. For the MNIST dataset [LeCun et al. 1998], I show simulation results with a network architecture suitable for hardware execution. In both cases, I compare the results to additional experiments done with the surrogate gradient method.

5.1 Implementation

To implement the EventProp Algorithm, as derived for continuous time by Wunderlich and Pehle [2021] and here described in section 2.3.2, the forward and adjoint dynamics are discretized and integrated using the explicit Euler integration scheme with step size Δt. A version of this discrete EventProp algorithm was already implemented in Norse [Pehle and Pedersen 2021] preceeding my work, though, it makes a simplification by not accounting for the intricate way gradients must be handled in layered networks. To handle backpropagation in layered networks appropriately, I implemented PyTorch functions `EventPropSynapse` and `EventPropNeuron` with custom forward and backward methods. Following, I will go through the intricacies of those functions in detail.

The dynamics of the LIF neurons are either computed in simulation only or can be injected from observations when training with BSS-2 in the loop. This, together with computing the adjoint trajectories, is handled in `EventPropNeuron`, a custom PyTorch autograd function. The complete dataflow, together with `EventPropSynapse`, another function ensuring the correct backpropagation to the synaptic weights and the previous layer, is displayed in fig. 5.1. The gradient estimation for a layer of LIF neurons is described in Algorithm 1. While the full code of `EventPropSynapse` and `EventPropNeuron` is listed in listing B.1, this section contains portions of the code with slight modifications or abbreviations to allow focus on the important aspects.

Algorithm 1 Discrete EventProp gradient estimation for a single layer of LIF neurons

```
input { pre-synaptic spikes $\boldsymbol{z}_\mathrm{pre}$, weights $\boldsymbol{W}$, time constants $\tau_\mathrm{m}$, $\tau_\mathrm{s}$, total time $T$,
  integration step size $\Delta t$, optional: hardware data $(\boldsymbol{V}_\mathrm{hw}, \boldsymbol{z}_\mathrm{hw})$ }

                                                                        ▷ Forward pass
  if no hardware data then
      $(\boldsymbol{V}_\mathrm{sw}, \boldsymbol{I}_\mathrm{sw}, \boldsymbol{z}_\mathrm{sw}) \leftarrow$ Forward Euler integration with step size $\Delta t$ of forward dy-
  namics and jumps from 0 to $T$
  else
      Assume $\boldsymbol{I}_\mathrm{hw} = \boldsymbol{I}_\mathrm{sw} \leftarrow$ Forward Euler integration with step size $\Delta t$ from 0 to
  $T$ or current dynamics
  end if

                                                                        ▷ Backward pass
  $(\boldsymbol{\lambda}_I, \boldsymbol{\lambda}_V) \leftarrow$ Forward Euler integration with step size $\Delta t$ of adjoint dynamics and
  jumps from $T$ to 0

output $\frac{\mathrm{d}L}{\mathrm{d}\boldsymbol{W}} = -\tau_\mathrm{s}\boldsymbol{\lambda}_I^\top \mathbf{z}_\mathrm{pre}$, $\frac{\mathrm{d}L}{\mathrm{d}\mathbf{z}_\mathrm{pre}} = (\boldsymbol{\lambda}_V - \boldsymbol{\lambda}_I)\boldsymbol{W}$
```

EventProp in Layered Networks — A Remark

In section 2.3.2 the dynamics are considered for a complete network, where W holds the information over all connections. By using `hxtorch.snn`, the networks are built by layering `hxtorch.snn.Synapse` projections and `hxtorch.snn.Neuron` population:

```
linear = hxtorch.snn.Synapse(n_out, n_in, ...)
lif = hxtorch.snn.Neuron(n_out, ...)
```

Having an input spike train `z_in` with `shape=(batch_size, seq_length, n_in)`, the forward execution would be

```
z_projected = linear(z_in)
s_out = lif(z_projected)
```

where `s_out` is a `NeuronHandle` holding the membrane traces and output spike train.

By splitting the network into such layers, each projection layer is defined by a weight matrix $\boldsymbol{W}$ only representing a block of the matrix considered in the derivation of EventProp. However, this splitting prompts the need to correctly handle the computation of gradients, because the weights are not available in the `hxtorch.snn.Neuron` module in which the adjoint variables are computed during backpropagation. The weight gradient in EventProp, as derived in section 2.3.2, samples the adjoint variable λ_I at spike times of the prior layer according to

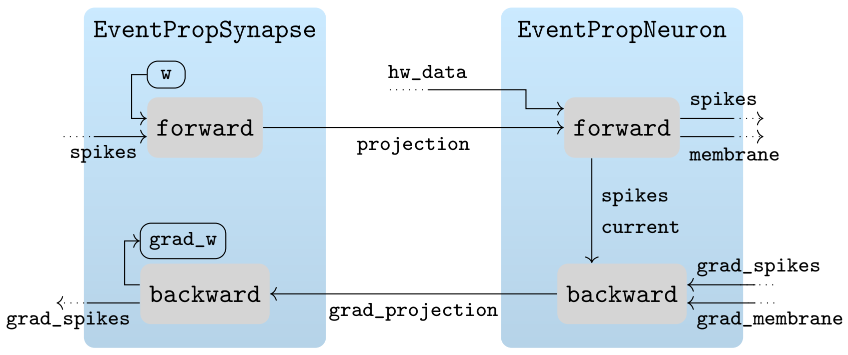


Figure 5.1: Dataflow occurring during BSS-2 ITL training. During the ITL training of BSS-2, input spikes are passed through `EventPropSynapse`, which returns weighted input spikes along with an empty tensor of the same shape to accommodate backpropagation of two separate gradient terms. The output of `EventPropNeuron` can be either the observations from BSS-2 or, if those are not present, the forward trajectories in simulation. In the backward pass, the adjoint dynamics are computed using the stored spikes from the forward call, and a tensor holding $\tau_\mathrm{s}\lambda_\mathrm{I}$ and $(\lambda_\mathrm{I} - \lambda_\mathrm{V})$ is returned. Returning the stacked tensor is only possible because the output of `EventPropSynapse` is already a stacked tensor with the same shape. In the backward direction, `EventPropSynapse` propagates $\tau_\mathrm{s}\lambda_\mathrm{I}^\top z_\mathrm{pre}$, the gradient with respect to the synaptic weights, and $(\lambda_\mathrm{I} - \lambda_\mathrm{V})w$, the gradient with respect to the pre-synaptic spikes.

eq. (2.48), which reads

$$\frac{\mathrm{d}\mathcal{L}}{\mathrm{d}w_{ji}} = -\tau_\mathrm{s} \sum_{\text{spikes from } i} \lambda^+_{I,j}. \tag{5.1}$$

So λ_I has to be backpropagated to the weights somehow. However, the jump of the adjoint variable λ_v contains terms depending on the adjoint variables of subsequenly connected neurons, as in eq. (2.53), which reads

$$\lambda^-_{v,n} = \frac{(\dot{v}^+)_n}{(\dot{v}^-)_n}\lambda^+_{v,n} + \frac{1}{\tau_\mathrm{m}(\dot{v}^-)_n}\left(\underbrace{\sum_{m=0}^{N-1}\left(\lambda^+_{v,m} - \lambda^+_{I,m}\right) w_{mn}}_{\substack{\text{gradient between}\\ \text{neuron layers}}} + \partial_{t_k^\text{post}} l_\mathrm{p} + l^-_{x,k} - l^+_{x,k}\right). \tag{5.2}$$

Therefore, two seperate terms have to be backpropagated between neuron layers in software, one containing $-\tau_\mathrm{s}\lambda_I$ of all post-synaptic neurons at pre-synaptic spike times and the other containing the product of $\lambda^+_v - \lambda^+_I$ from the post-synaptic layer with the weight matrix $\boldsymbol{W}$ specific to the synapse layer between neuron layers.

Forward

In a layered network, the synapse layer receives spikes `input` from an external source or from a pre-synaptic neuron layer and projects these onto the post-synaptic layer with weights contained in the tensor `weight`. This is handled by `EventPropSynapse`, which basically applies the same computations as `torch.nn.functional.linear`, but in forward direction returns the projection stacked onto a tensor of zeros with same shape to accomodate the backpropagation of the above-mentioned two gradient terms:

```
class EventPropSynapse(torch.autograd.Function):
  @staticmethod
  def forward(ctx, input, weight, _):
    ctx.save_for_backward(input, weight)
    output = input.matmul(weight.t())
    return torch.stack((output, torch.zeros_like(output)))
  ...
```

The `EventPropNeuron` function is used in the `hxtorch.snn.Neuron` module to simulate forward dynamics and handling the corresponding gradient backpropagation according to EventProp. `EventPropNeuron.forward` receives the projected, weighted spikes inside of `input` together with neuron parameters. The neuron parameters are stored a `hxtorch.snn.functional.LIFParams` object and hold the neurons inverse time constants `tau_syn_inv` and `tau_mem_inv`, leak, reset and threshold voltages `v_leak`, `v_reset`, and `v_th`, and time step size `dt`. If the experiment is executed on BSS-2, the `hxtorch.snn.Neuron` module overwrites `forward` and directly forwards observed spike trains and membrane traces. When the experiment is done in simulation only, the projected spikes in `input[0]` are extracted into `input_current` and the currents `i` and membrane voltages `v` are integrated over all time steps `ts`. The main loop of `EventPropNeuron.forward` is

```
  for ts in range(T - 1):
    # Current
    i = i * (1 - dt * tau_syn_inv) + input_current[:, ts]
    current.append(i)

    # Membrane
    dv = dt * tau_mem_inv * (v_leak - v + i)
    v = dv + v

    # Spikes
    spike = torch.gt(v - v_th, 0.0).to((v - v_th).dtype)
    z = spike

    # Reset
    v = (1 - z.detach()) * v + z.detach() * v_reset

    # Save state
    spikes.append(z)
    membrane.append(v)
```

where spikes and membrane are lists to which the observables are appended after each integration step and at the end are stacked into tensors to be returned by EventPropNeuron.forward. The input, the computed membrane and spike values, and if in simulation only, also the current, are saved for backward.

Backward

For the computation of the adjoint jumps in backward direction, the spike times and the time derivatives of the membrane $\dot{V}$ are needed. For ideal dynamics, those are determined only by the synaptic currents I at spike times. When training with BSS-2 ITL, enabling synaptic current measurements is not exposed in the hxtorch.snn API and therefore those are approximated to calculate the jumps of the adjoint variables. This approximation is done by numerically integrating eq. (2.39) using the binary tensors, to which the spike recordings are mapped, to apply the transitions as in eq. (2.42).

The computation of the adjoint jumps, cf. eq. (2.53), is split into two terms according to

$$\lambda^-_{v,n} = \underbrace{\frac{(\dot{v}^+)_n}{(\dot{v}^-)_n}\lambda^+_{v,n}}_{\texttt{jump\_term}} + \underbrace{\frac{1}{\tau_\mathrm{m}(\dot{v}^-)_n}\left(\sum_{m=0}^{N-1}\left(\lambda^+_{v,m} - \lambda^+_{I,m}\right) w_{mn} + \partial_{t_k^\mathrm{post}} l_\mathrm{p} + l^-_{x,k} - l^+_{x,k}\right)}_{\texttt{output\_term}} . \tag{5.3}$$

In EventPropNeuron.backward, the adjoint dynamics are integrated in reverse time according to eqs. (2.45), (2.46) and (2.53). The main loop here reads

```
for ts in range(T - 1, 0, -1):
  dv_m = v_leak - v_th + i[:, ts - 1]
  dv_p = i[:, ts - 1]

  dlambda_i = tau_syn_inv * (lambda_v[:, ts] - lambda_i[:, ts])
  lambda_i[:, ts - 1] = lambda_i[:, ts] + dt * dlambda_i

  dlambda_v = - tau_mem_inv * lambda_v[:, ts]
  lambda_v[:, ts - 1] = lambda_v[:, ts] + dt * dlambda_v

  output_term = z[:, ts] / dv_m * grad_spikes[:, ts]
  jump_term = z[:, ts] * dv_p / dv_m

  lambda_v[:, ts - 1] = \
    (1 - z[:, ts]) * lambda_v[:, ts - 1] \
    + jump_term * lambda_v[:, ts - 1] \
    + output_term

return torch.stack((lambda_i / params.tau_syn_inv,
                    lambda_v - lambda_i)), None
```

There, when computing the membrane time derivatives $\dot{V}^-$ (or dv_m) before and $\dot{V}^+$ (or dv_p) after the jump, I make the approximation of assuming the membrane voltage to have precisely the threshold value before the jump and being reset to exactly 0 afterward. The impact of this choice or alternatives of, e.g., incorporating membrane measurements at these points are not studied in this thesis but worth considering in future work.

The returned gradients of `EventPropNeuron.backward` are received by `EventPropSynapse.backward` as `grad_ouput`. The first part of the stacked tensor, holding the backpropagated $\tau_\mathrm{s}\lambda_I$, is multiplied with the pre-synaptic spike tensor $\boldsymbol{z}_\mathrm{pre}$, effectively sampling and summing over the adjoint state at pre-synaptic spike times. The result is backpropagated as the weight gradient. The second part, holding the difference $\lambda_v - \lambda_I$, is multiplied with the synaptic weights in $\boldsymbol{W}$ and backpropagated to the previous neuron layer. This is handled in the `backward` method of `EventPropSynapse`:

```python
class EventPropSynapse(torch.autograd.Function):
  ...
  @staticmethod
  def backward(ctx, grad_output):
    input, weight = ctx.saved_tensors
    grad_input = grad_weight = None
    grad_input = grad_output[1].matmul(weight)
    grad_weight = grad_output[0].transpose(1, 2).matmul(input)
    return grad_input, grad_weight, None
```

The two PyTorch autograd functions `EventPropSynapse` and `EventPropNeuron` were added to the `hxtorch.snn` software stack and can be used inside `hxtorch.snn.Synapse` and `hxtorch.snn.Neuron` as an alternative to gradient estimation with surrogate gradients. They allow for the correct backpropagation of numerically estimated gradients corresponding to the EventProp algorithm in a layered network of LIF neurons and enable users train and evaluate of SNNs either in simulation only or with BSS-2 ITL.

Sign-flip in EventProp implementation

While implementing and testing the discrete EventProp algorithm, I encountered the issue of a mismatch in the sign of gradients, when comparing to the well studied surrogate gradients method and therefore switched the sign of the returned `lambda_i` (or λ_I), as can be seen in line 112 of listing B.1. To demonstrate that this in fact leads to the correct gradient sign, I considered an experiment setup of a LIF and subsequent LI neuron. A single input spike is sent with strong enough weight into the first neuron, which in turn emits a spike. This hidden spike is received by the output LI neuron and the gradient of its maximum membrane value over time is computed through backpropagation. I simulated this setup with both the discrete EventProp implementation and surrogate gradients and tracked all state variables and their gradients (if existing) (cf. fig. 5.2). The gradients have the same sign, even though the EventProp implementation used for this work contains a different sign in the term backpropagated to the weights than compared to the underlying equation in eq. (2.48).

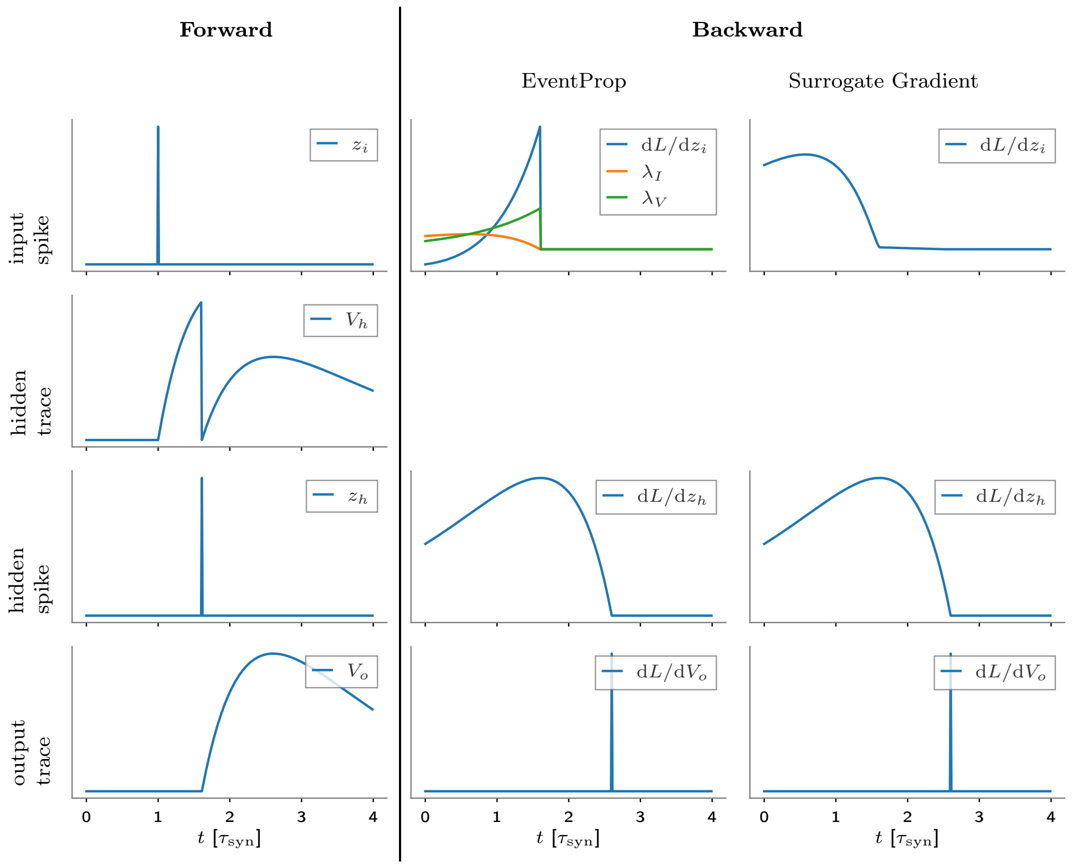


Figure 5.2: Gradients for a single hidden LIF and output LI neuron and max-over-time. A single input spike is sent to a LIF neuron, triggers a post-synaptic spike, which is forwarded to a LI readout neuron. Gradient of the maximum membrane value of the readout neuron is backpropagated through two the layers and visualized for gradient estimation using the discrete EventProp implementation of this thesis and surrogate gradients.

The same is the case for two consecutive LIF neurons when using the spike time decoder (cf. section 4.1) to compute a floating point spike time output and backpropagate the gradients (cf. fig. C.1). For both, the sign of the weight gradients would be sampling $\mathrm{d}L/\mathrm{d}z$ at the pre-synaptic spike times, which would yield a negative value for the gradient. The sign then is correct, since the neuron's membrane rises more quickly with larger synaptic input and therefore the spike occurs at an earlier time. This supports the choice of gradients injected in the spike time decoder (cf. eq. (4.2)) and the sign used in the EventProp implementation at line 112 in listing B.1.

Nevertheless, the fundamental reason for the necessity to make this choice is not yet fully understood and further investigations have to be carried out. For all of the experiments conducted throughout this thesis, this choice was kept (cf. listing B.1).

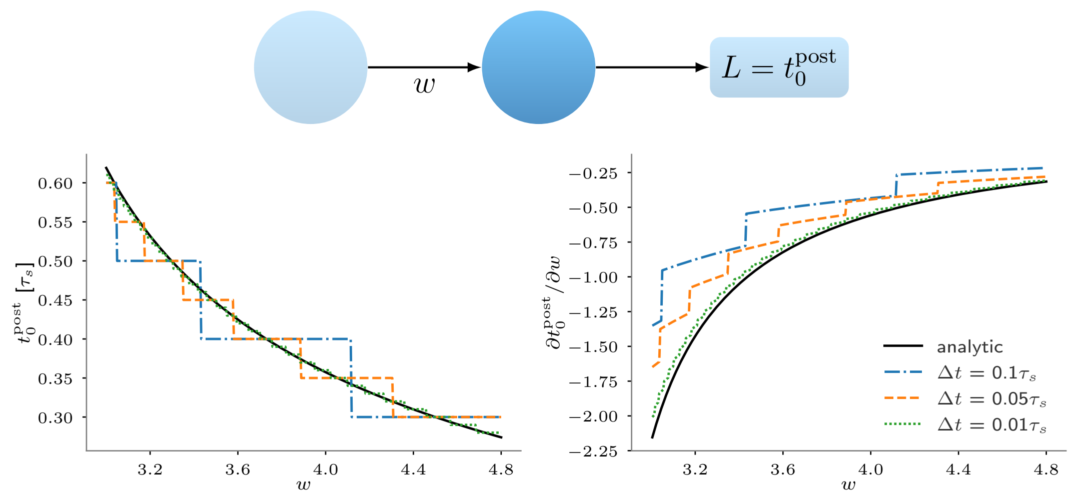


Figure 5.3: First output spike time and weight gradient in experiment setup with a single synapse. A LIF neuron is simulated receiving an input spike with weight w at time $t_{\text{pre}} = 0$. The time of the first post-synaptic spike t_0^{post} is used as a loss function, for which gradients are estimated using the EventProp algorithm. Spike time and gradient for a neuron receiving an input at $t_{\text{pre}} = 0$ through a single synapse with weight w in simulation. The numerically estimated spike time (left) and weight gradient (right) are shown as functions of the weight w for three integration step sizes Δt. The analytically known expressions are displayed for comparison eqs. (5.4) and (5.5).

5.2 Single Synapse Experiment

Before training on complete datasets with my implementation of the EventProp algorithm, I wanted to verify its correctness or check for eventual problems. Therefore I turned to the work of Göltz et al. [2021], which derives explicit expressions for weight gradients of the first spike time in LIF neurons in special cases of the time constants (cf. section 2.3.3). Since they also succesfully applied this gradient estimation using analytical gradient expressions to hardware ITL training with BSS-2, which was also the aim of my work, it seemed a natural choice to compare my implementation to their findings. Using these analytical expressions as references, I analyzed the first spike time and the estimated gradient for an experiment setup with a single synapse, compared my observations from simulation and also from using BSS-2 ITL, and found that the estimated gradients agree well with those found through evaluation of analytical expressions.

I considered the experiment setup of a LIF neuron receiving a single spike at t^{pre} through a synapse with weight w as in fig. 5.3 and having equal time constants $\tau_{\text{m}} = \tau_{\text{s}}$. When considering a single pre-synaptic spike with strong enough weight, the time of the first post-synaptic spike in eq. (2.56) and its gradient in eq. (2.59)

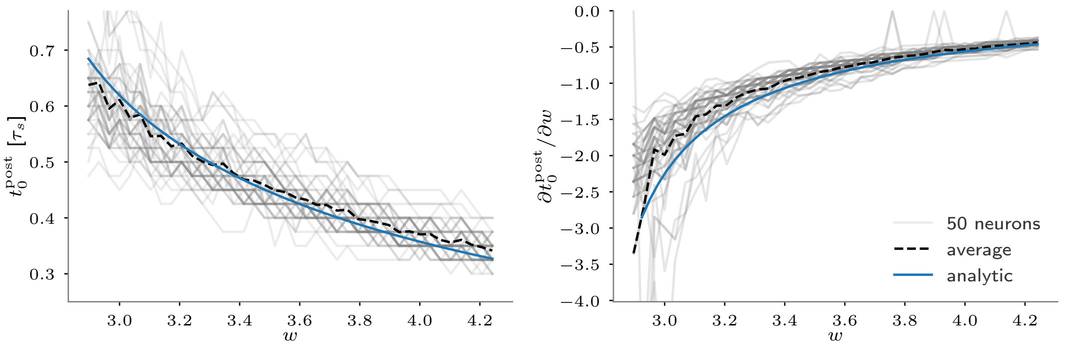


Figure 5.4: Spike time and gradient for a neuron receiving an input at $t_{\text{pre}} = 0\,\mu\text{s}$ through a single synapse with weight w with BSS-2 ITL. The spike time measured on BSS-2 (left) and numerically estimated gradient (right) are shown as functions of the weight w for 50 neurons. For comparison, the analytical values of eqs. (5.4) and (5.5) (blue line) and the average for the observations from BSS-2, or gradient estimation respectively, (black, dashed line) are shown.

become

$$t_0^{\text{post}} = t^{\text{pre}} - \tau_{\text{s}} \mathcal{W}\left(-\frac{V_{\text{th}}}{w}\right), \tag{5.4}$$

$$\frac{\partial t_0^{\text{post}}}{\partial w} = -\frac{\tau_{\text{s}}}{w} \frac{t_0^{\text{post}} - t^{\text{pre}}}{\mathcal{W}(-\frac{V_{\text{th}}}{w}) + 1}. \tag{5.5}$$

Using my discrete EventProp implementation, I simulated the system for three integration step sizes $\Delta t \in \{0.1\tau_{\text{s}}, 0.05\tau_{\text{s}}, 0.01\tau_{\text{s}}\}$ and pre-synaptic spike time $t_{\text{pre}} = 0$. In this setup, where the loss is dependent on the spike time only, those times need to be extracted from dense binary tensors. To be able to compute gradients for the weight w or, more generally speaking, optimize on such losses, the spike time decoder described in section 4.1 can be used. For a range of weights I compared the resulting first spike times and estimated gradients to the analytic formulas in eqs. (5.4) and (5.5). The findings are displayed in fig. 5.3.

After demonstrating that the spike time and gradient in simulation converge to the expected analytical values with decreasing integration step size Δt, I conducted the same experiment with the BSS-2 system for the forward pass. I considered the same case of equal time constants and use a calibration with $\tau_{\text{m}} = \tau_{\text{s}} = 6\,\mu\text{s}$. Figure 5.4 shows the spike times of 50 neurons measured on BSS-2 and the gradient estimated for the observed spike times. As can be seen in the left plot, the observed spike times were mapped to a discrete time grid, which in this case had a chosen resolution of $\Delta t = 0.25\,\mu\text{s}$. The mean of the estimated gradient $\partial t_0^{\text{post}}/\partial w$ using BSS-2 ITL agrees well with the analytical estimation.

To align hardware measurements to the simulated dynamics, a scaling and offset was applied to the membrane traces and also a weight scaling factor was used. Even

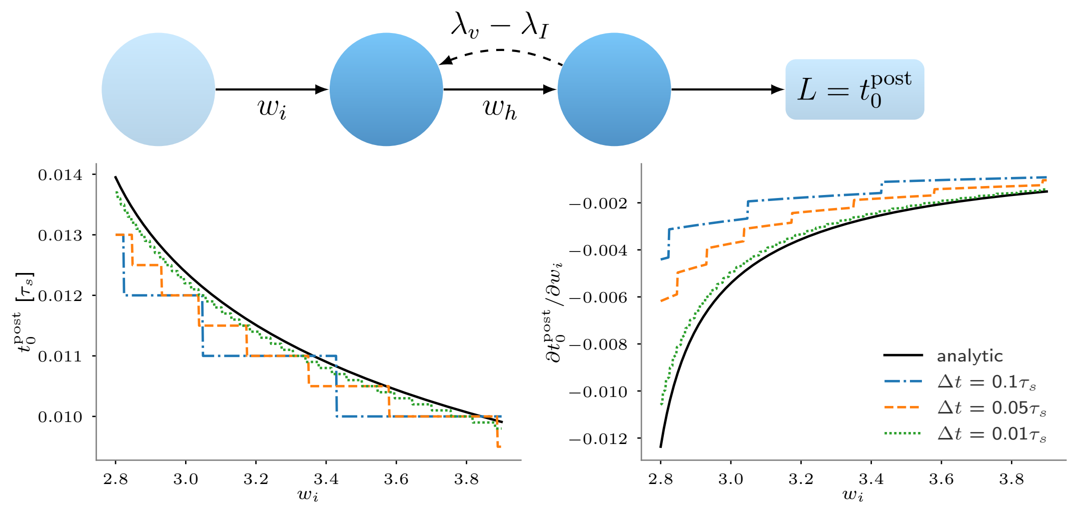


Figure 5.5: First output spike time and input weight gradient in experiment setup with hidden neuron. A LIF neuron is simulated receiving an input spike at time $t_{\text{pre}} = 0$ via a synapse with weight w_i, emitting a spike itself which is transmitted by a synapse with weight w_h to an output LIF neuron. The computed time of the first post-synaptic spike t_0^{post} of the latter neuron is considered as a loss function and gradients are estimated with EventProp. The term $\lambda_v - \lambda_I$, for which correct back-propagation between neuron layers is to be verified here, is additionally illustrated. The first output spike time (left) and estimated input weight gradient (right) are shown as functions of the input weight w_i for three integration step sizes Δt. The analytic spike time (cf. eq. (2.63)) and the gradient, by chaining eqs. (2.64) and (2.65), are shown for comparison.

though using a calibration, the exact dynamics of neurons and the dependence of synaptic currents on the weight might differ slightly for each circuit. Hence, spike times might differ slightly between each neuron also if the same input was supplied, which would explain the noisy spike time measurements in fig. 5.4.

5.3 Hidden Neuron Experiment

The same approach as in the previous section was taken to demonstrate the correctness of backpropagated gradients between neuron layers. The term in the EventProp algorithm holding the gradient information passed between layers is $\lambda_v - \lambda_I$ (cf. eq. (2.53) and line 113 in listing B.1). An extended experiment was considered in simulation to verify the implementation, which comprises two consecutive neurons with input and hidden synapses, with weights w_i and w_h, respectively. The hidden neuron receives a spike at time $t_{\text{pre}} = 0$ along the input synapse and emits a spike itself, which is transmitted to the output neuron along the hidden synapse. The first spike time of the output neuron is then considered similar to a loss function and gradients are computed by backpropagation.

Figure 5.5 displays the setup and the observed first output spike times t_0^{post} and the input weight gradients $\mathrm{d}t_0^{\text{post}}/\mathrm{d}w_i$ as functions of the input weight w_i. The analytically expected spike time and gradient obtained from eqs. (2.63) to (2.65) are shown for comparison. For decreasing integration step size Δt, the numerical estimate of the output spike time and gradient both converge to the values of the analytically known expressions. Therefore, I considered the discretized EventProp implementation using forward Euler integration to be sufficient for further experiments.

5.4 Yin-Yang

The Yin-Yang dataset [Kriener et al. 2022] is a low dimensional dataset developed specifically for exploratory, early-stage prototyping of models and hardware platforms. The low dimensionality of the task, with each sample consisting of only four values, makes this dataset a well suited task for first hardware ITL experiments with BSS-2 and EventProp. Also, Wunderlich and Pehle [2021] use this task to train a network of LIF neurons in simulation with the EventProp algorithm and Göltz et al. [2021] train similar networks with BSS-2. Therefore, the Yin-Yang task is well suited for a first demonstration of BSS-2 ITL training with EventProp.

I trained a network of LIF and LI readout neurons similar to mentioned prior work in simulation and with BSS-2 ITL. Studying its characteristics during training and the performance on the Yin-Yang testset, I try to solve arising problems and improve the initial experiment and model setup to eventually reach comparable, and in some cases better performance as prior work, cf. table 5.1.

5.4.1 Task

The samples of the Yin-Yang task are based on the area of a circle, described by the set

$$\left\{(x, y) \in \mathbb{R}^2 \mid (x-r)^2 + (y-r)^2 \leq r^2, r = 0.5\right\}, \tag{5.6}$$

where each point is classified by one of three classes “yin”, “yang”, “dot” (cf. fig. 5.6). The dataset comes symmetrized by including the values $1-x$ and $1-y$ for each sample, so a single dataset value is $\boldsymbol{x} = (x, y, 1-x, 1-y)$.

In order to use this dataset in spiking neural networks, the spatial data $\boldsymbol{x}$ has to be encoded into the temporal domain. For all my experiments with the Yin-Yang dataset, I use the suggested spatio-temporal input encoding

$$\boldsymbol{t} = (t_1, t_2, t_3, t_4) = t_{\text{early}} + \boldsymbol{x}\left(t_{\text{late}} - t_{\text{early}}\right), \tag{5.7}$$

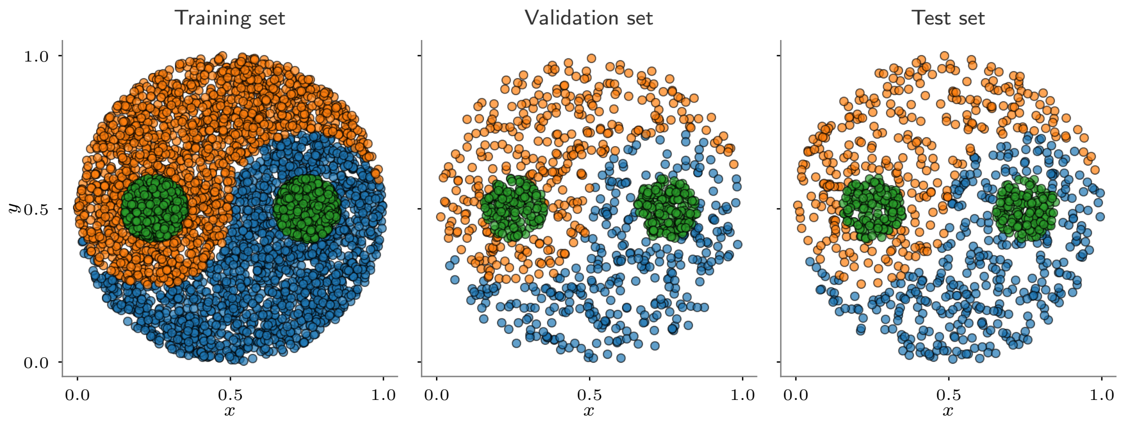


Figure 5.6: Default training, validation and testset of the Yin-Yang dataset [Kriener et al. 2022]. The dataset is represented by two-dimensional points classified into one of the three classes yin (blue), yang (orange) or dot (green).

Table 5.1: Accuracies on the Yin-Yang testset after training ANNs [Kriener et al. 2022] (* for frozen lower weights) and SNNs [Göltz et al. 2021, Wunderlich and Pehle 2021] with different gradient estimation methods.

Type	Grad. estimator	Acc. [%]
ANN	Backpropagation	97.6 ± 1.5
	Backpropagation (*)	85.5 ± 5.8
SNN	Fast And Deep	95.9 ± 0.7
	Fast And Deep (BSS-2)	95.0 ± 0.9
	EventProp	98.1 ± 0.2

with $t_{\text{early}} < t_{\text{late}}$, where $t_{\text{early/late}}$ are hyperparameters that can be chosen depending on the network architecture and learning algorithm.

The results of previous work [Kriener et al. 2022, Göltz et al. 2021, Wunderlich and Pehle 2021] on the Yin-Yang task are listed shortly in table 5.1 for ANNs and SNNs. This is to give an idea of what results are aimed for in this work.

5.4.2 Experiment Setup and Training Procedure

In the following, I will describe the architecture of the network and parameters used to train on the Yin-Yang task, the encoding and decoding schemes, and the loss used for optimization.

The samples $\boldsymbol{x} = (x, y, 1-x, 1-y)$ from the dataset are encoded into spike times $\boldsymbol{t}$ according to eq. (5.7). An additional bias spike time t_{bias}, which is constant for all samples, is added. This bias spike is supposed to increase activity and facilitate training [Göltz et al. 2021]. Its associated synaptic weight is learned like all other synaptic weights. The four sample spike times and the bias spike time are then mapped onto a discrete time grid into boolean spike tensors used as input to a network of LIF and LI neurons.

The feed-forward network consists of a hidden layer with 120 LIF neurons and an output layer of 3 LI neurons, one for each class. The loss function I consider for my experiments with the Yin-Yang task is composed of two terms $L = L_1 + L_2$. The first and main term is a max-over-time loss

$$L_1 = -\frac{1}{N_{\text{batch}}} \sum_{n=1}^{N_{\text{batch}}} \log \frac{\exp\left(\max_t V^{\text{out}}_{n,y_n}(t)\right)}{\sum_{c=1}^{C} \exp\left(\max_t V^{\text{out}}_{n,c}(t)\right)}, \tag{5.8}$$

with time t, the voltages of the output layer neurons V^{out}, the target y, the batch size N_{batch} and the number of classes C. To prevent amplitudes from being to high, which on BSS-2 would lead to saturation, I use the regularization loss

$$L_2 = \alpha \cdot \frac{1}{N_{\text{batch}}\, C} \sum_{n=1}^{N_{\text{batch}}} \sum_{c=1}^{C} \left(\max_t V^{\text{out}}_n(t)\right)^2, \tag{5.9}$$

where α is a scaling factor to adjust the influence of this amplitude regularization.

To optimize the network on the given loss functions in eqs. (5.8) and (5.9), I use the Adam optimizer [Kingma and Ba 2014] with its default settings. Additionally, all of my experiments on the Yin-Yang task use a step-wise learning rate scheduling, which adjusts the learning rate after a given number of epochs by a specified factor. In the initial experiments, I train the network for 30 epochs in simulation and 50 epochs with BSS-2 ITL. For comparison, I separately train networks using EventProp and also surrogate gradients.

All other parameters, which might vary depending on the used gradient estimator or between training in simulation and using BSS-2, are listed in appendix D.

5.4.3 Initial Experiment Results

To obtain the first results on the YinYang task, I train the described network arhcitecture in simulation only and with BSS-2 ITL. For gradient estimation, I compare the two available gradient estimators to allow fo better pinpointing of possible problems to the EventProp implementation or other sources.

Using the `hxtorch.snn` framework, switching between the built-in gradient estimator using surrogate gradients and my EventProp implementation is straightforward. The same applies to changing between forward execution on BSS-2 or simulating the forward dynamics of the network in software. This allows me to study both gradient estimators and forward execution types at the same time. In this way, I train a feed-forward network as described in section 5.4.2 using the parameters in tables D.1 and D.2. The results are displayed in fig. 5.7.

In simulation, the EventProp model reaches an average accuracy of 96.10 %, similar the surrogate gradient model. With BSS-2 though, both models initially only achieve slightly above 80 %. Comparing to other results using BSS-2 in table 5.1, this is not what one expects. Hence, possible issues need to be investigated.

Kriener et al. [2022] compare the performance of ANNs with single hidden layer for frozen and learnable lower weights (cf. table 5.2), where the network with frozen

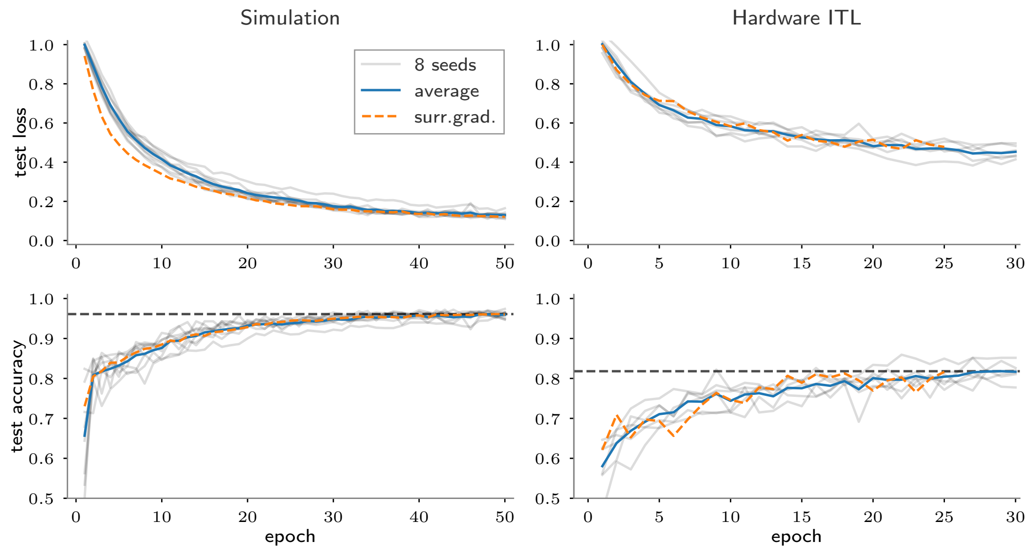


Figure 5.7: Test loss and accuracy on the Yin-Yang task over the training epochs for the untuned model using EventProp and without any non-model specific adjustments. The black, dashed lines show the average accuracy on the test set after the last training epoch, which is 96.10 % in simulation and 81.80 % with BSS-2 ITL. For comparison, the average loss and accuracy over training with surrogate gradients is also shown.

lower weights only reaches 85.5 %. This is supposed to show that simply projecting the Yin-Yang task to a higher-dimensional space is not sufficient to achieve high accuracies with linear classification in the output layer. This might also be the issue with the SNNs considered in this experiment. Therefore, a natural first subject to investigate, are the gradients in each layer and the change in weights over the course of the training.

5.4.4 Model and task specific modifications

To achieve higher accuracies in hardware ITL training with BSS-2, several possible limiting subjects are investigated. First, I will highlight and discuss the multiple *orders of magnitude* (OOM) difference in gradients for the hidden and output layer when using EventProp for gradient estimation. Second, I briefly discuss the relevance of dynamic range in synaptic input and possible ways to achieve stronger signals in tasks with low input dimensionality. At last, I will point out that encoding the data onto a discrete time grid creates an upper limit on the maximum possible performance.

Gradient Rescaling

In fig. 5.8, the layer-specific gradient distribution is shown at different points in training. Compared to the output layer gradient distributiion, the distribution of

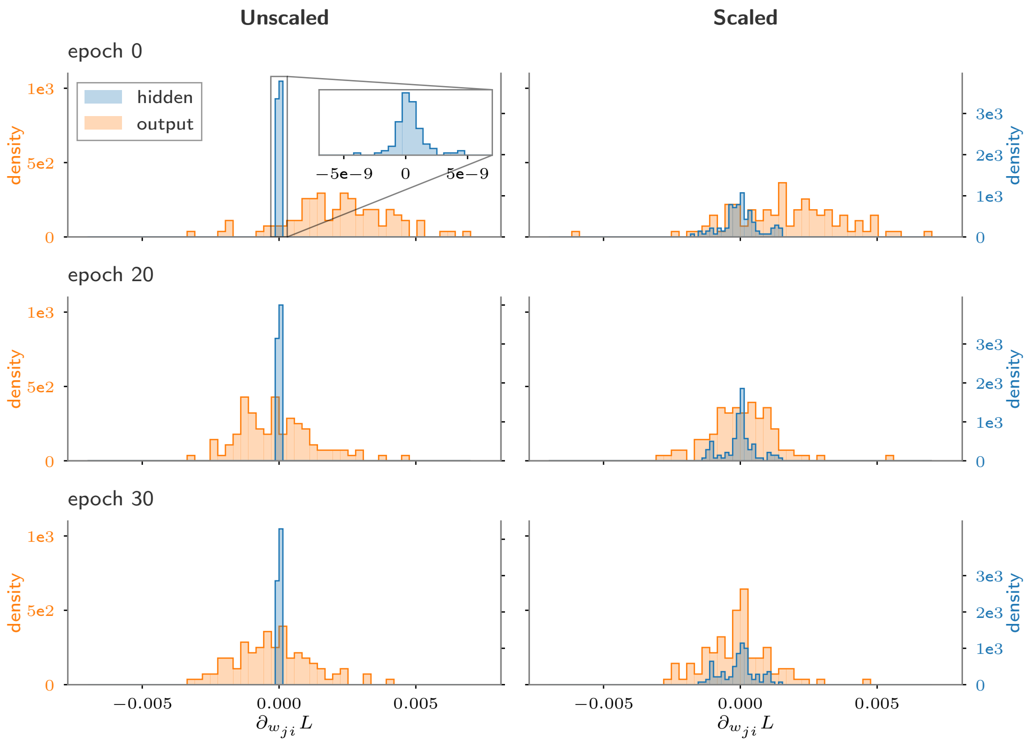


Figure 5.8: Layer-specific gradient distribution for unscaled and scaled EventProp. The distribution of gradients observed in the hidden (blue) and output layer (orange) during BSS-2 ITL training are shown for EventProp using the true gradient (left) in eq. (2.48) and for a gradient scaled by τ_s^{-1} (right).

gradients in the hidden layer is 6 OOM more narrow, with gradients in the range $[-10^{-8}, 10^{-8}]$. The layer-specific gradient distributions for training in simulation are displayed in fig. E.1.

The initial assumption was, that these multiple OOM difference in gradients when training with BSS-2 might lead to the hidden weights staying almost constant during training, while only the output layer learns. In the simulation based training, I use time constants in the 10^{-3} to 10^{-2} range and only observe a 3 OOM difference between the gradient distributons of the hidden and the output layer. When training with BSS-2, I use time constants around $10\,\mu s = 10^{-5}\,s$ and observe the mentioned 6 OOM difference in the width of the layer specific gradient distributions. Therefore, this roughly scales with the time constant, which is to be expected if looking at eq. (2.48), where the synaptic time constant τ_s directly enters into the gradient computed with EventProp. With this reasoning, I chose to scale the gradient by the inverse of the synaptic time constant and therefore counteract the imbalance of the hidden and output layer weight gradients.

However, this rescaling has no significant effect on the weight updates, because I use the Adam optimizer in training. A primary characteristic of the Adam optimizer

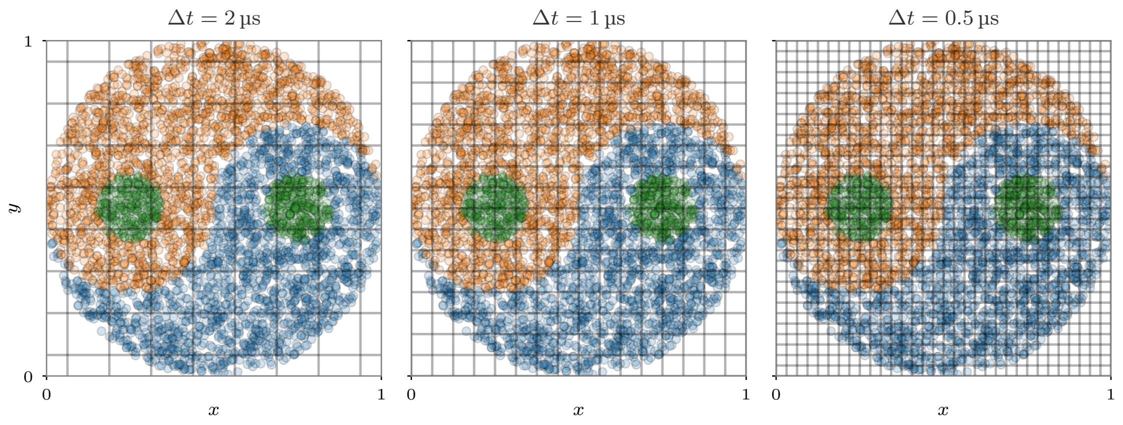


Figure 5.9: Resolution of spike encoding for different step sizes $\Delta t \in \{2\,\mu s, 1\,\mu s, 0.5\,\mu s\}$. The grid separates the dataset into regions inside which all samples are encoded into the same spike sequence and therefore are not distinguishable after encoding. Resolutions are shown for the encoding window $[t_{\text{early}}, t_{\text{late}}] = [1\,\mu s, 16\,\mu s]$ used to obtain the initial BSS-2 ITL training results with Yin-Yang.

is that it is almost invariant to the rescaling of the gradients. This near invariance however depends on the choice of its parameter ϵ, which is only added to prevent a division by zero. Usually, gradients are much larger than ϵ, but here they get small enough to have the same order of magnitude as ϵ, which can have a non-negligible effect on the weight updates. Even though the rescaling of gradients ultimately does not affect the training success here, different OOM in weight gradients are highly relevant if using other optimization algorithms that depend on the scaling of gradients, like stochastic gradient descent.

Input repetition

In the case of training the Yin-Yang task, Göltz et al. [2021] mention that in their work the five inputs are insufficient to achieve good classification results when training with BSS-2 ITL. Their suggested solution is to repeat the input per sample, including the bias, five times, ultimately having 25 input streams per data point. I adopted their suggestion with the adjustment of initializing the hidden weights as an $n_{\text{hidden}} \times 5$ matrix and repeat them 5 times along the input dimension, resulting in a $n_{\text{hidden}} \times 25$ weight matrix. This has the equivalent effect of increasing synaptic efficacy without changing the target model parameters and underlying calibration data set.

Spike Encoding Resolution

To encode the samples of the Yin-Yang dataset into spike times, eq. (5.7) is used. Those spike times are then mapped onto a discrete time grid into boolean spike tensors. The step size Δt between times represented by entries on the time grid can be chosen and therefore acts as a hyperparameter in training. As fig. 5.9 shows, the choice of Δt is highly relevant, since it specifies the resolution with which the samples of the dataset are encoded. If the step size is too large and therefore, the time grid onto which samples are encoded is too coarse, samples around the boundaries of classes may be encoded into the same input spike tensor. Hence, those samples are not distinguishable from each other for the network. This creates an upper limit on the possible classification accuracy.

Alternatively, the time window $[t_{\text{early}}, t_{\text{late}}]$ into which samples are encoded could be enlarged to have a similar effect. However, this would also cause the input spikes to be further apart and the time constants may need to be adjusted.

To obtain the initial experiment results on the Yin-Yang dataset, shown in fig. 5.7, the step size was chosen to $\Delta t = 1\,\mu\text{s}$, correspondig to the resolution displayed in the center of fig. 5.9. This is not the initially most limiting factor in achieving better performance, since synaptic niput strength and dynamic range turns out to be the most crucial adjustment. However, as soon as accuracies of over 95 % are achieved, this could well be an important starting point through which further improvements can be achieved.

5.4.5 Experiment Results After Adjustments

I investigated possible task and model specific limitations and adjustments to overcome them in the previous section 5.4.4. In brief, these adjustments are the

- scaling of gradients estimated with EventProp in the hidden layer by the synaptic time constant τ_{s}^{-1},
- repetition of inputs per sample 5 times to increase effective synaptic strength per input,
- increase of input encoding resolution by lowering the integration step size Δt.

I again trained a network as described in section 5.4.2 with the above listed modifications for 200 epochs in simulation and 300 epochs with BSS-2 ITL. The parameters for those trainings are listed in tables D.3 and D.4. The obtained loss and error on the test set over the course of training are displayed in fig. 5.10 and final accuracies are compared to results of other work [Kriener et al. 2022, Wunderlich and Pehle 2021, Göltz et al. 2021] in table 5.2.

The results with BSS-2 ITL are significantly better than the initial results from fig. 5.7 and are comparable to prior work of others. As I already briefly discussed in section 5.4.4, the major reason for this improvement is the repetition of inputs resulting in higher synaptic strength and dynamic range.

To visualize the activity in the network after training on Yin-Yang with BSS-2 ITL, I display the input encoding, observed hidden spikes, and output traces for an exemplary sample of the training set and the maximum of the membrane voltage of

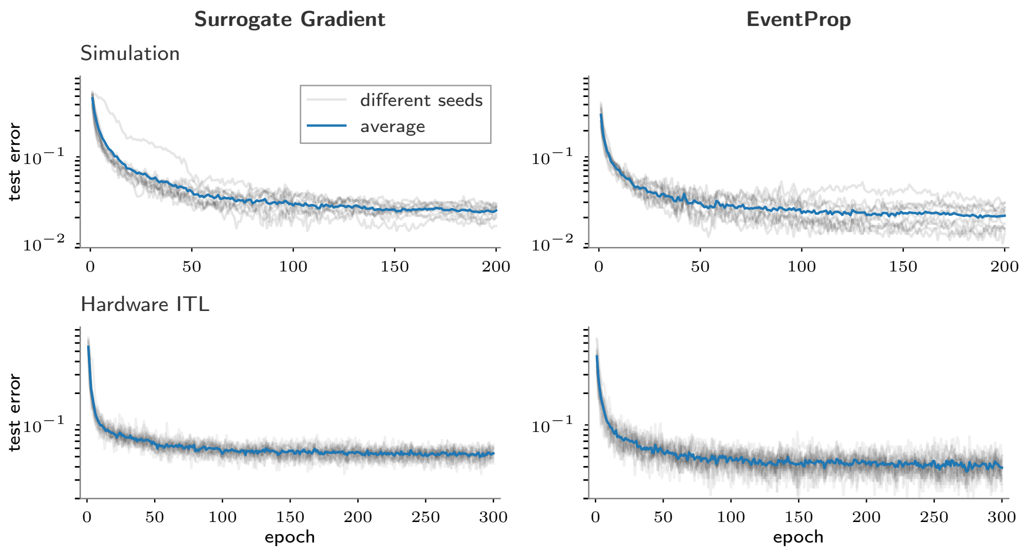
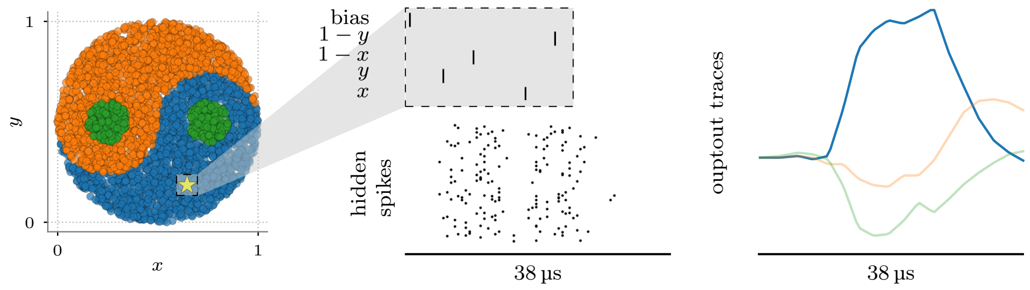
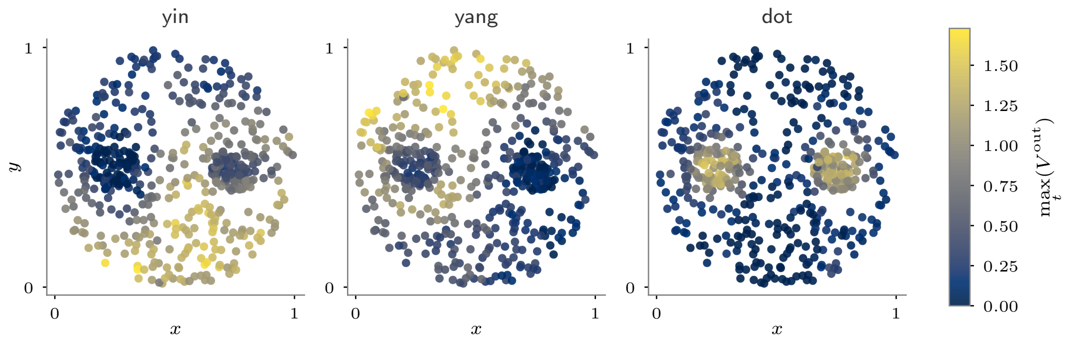


Figure 5.10: Test loss and error on Yin-Yang dataset after applying modifications described in section 5.4.4. Results using surrogate gradients or EventProp for gradient estimation are displayed. For each experiment the network is trained on 15 different seeds and the final performance is determined by the accuracy after the last training epoch averaged over runs. Training with BSS-2 ITL and EventProp results in an accuracy of $(96.1 \pm 0.8)\,\%$ on the test set after the last training epoch, average and standard deviation over all seeds. Performance is compared to prior work in table 5.2.



(a) Network activity for example Yin-Yang sample



(b) Maximum membrane values of trained model

Figure 5.11: Activity of model trained on Yin-Yang dataset. The maximum membrane values $\max_t(V^{\text{out}})$ of the readout neurons are used for classification into one of the three classes “yin”, “yang”, “dot”. **(a)** Illustration of input encoding and observed hidden spikes and output traces of trained model for an exemplary sample. **(b)** Maximum of observed membrane voltages of output neurons on Yin-Yang dataset after BSS-2 ITL training using EventProp. The boundaries between classes are not sharp, but still an average accuracy of 96.10 % can be achieved with BSS-2 ITL training (cf. fig. 5.10).

Ref.	**Type**	**Hidden**	**Loss**	**Gradient Estimator**	**Fwd.**	**Acc. [%]**
(a)	ANN	30	CE	Backpropagation	sim.	97.6 ± 1.5
				Backpropagation (frozen lower weights)	sim.	85.5 ± 5.8
(b)	SNN	120	TTFS	Fast And Deep	sim.	95.9 ± 0.7
				Fast And Deep	BSS-2	95.0 ± 0.9
(c)	SNN	200	TTFS	EventProp	sim.	98.1 ± 0.2
this work	SNN	120	MAX	surrogate gradient	sim.	97.6 ± 0.4
				surrogate gradient	BSS-2	94.6 ± 0.7
				EventProp	sim.	97.9 ± 0.6
				EventProp	BSS-2	96.1 ± 0.8

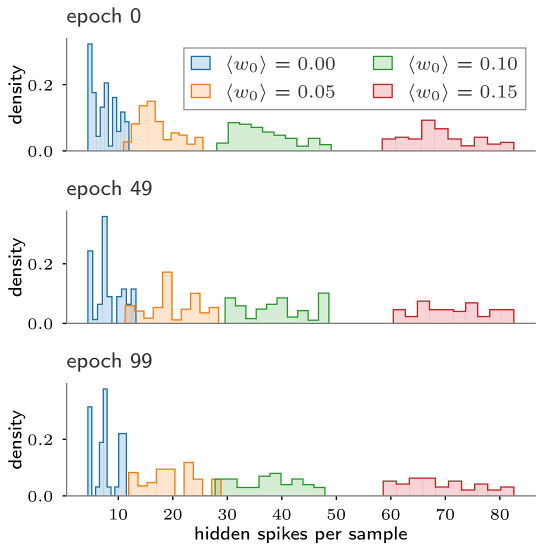
Table 5.2: Results achieved on Yin-Yang with different network types, ANNs and SNNs, and gradient estimation using explicit analytical expressions, surrogate gradients and EventProp. Prior results on the Yin-Yang task are referenced from (a) [Kriener et al. 2022], (b) [Göltz et al. 2021] and (c) [Wunderlich and Pehle 2021]. All networks use a single hidden layer with different numbers of neurons. Forward (Fwd.) execution of the network is either done by simulating the network (sim.) or on the BSS-2 system. The used loss functions are a cross-entropy loss 'CE', a loss based on time-to-first-spike decoding 'TTFS', and a max-over-time loss 'MAX'.

the output LI neurons on the test set (cf. fig. 5.11). The values $\max_t(V^{\text{out}})$ are used to classify the samples into one of the three classes of the Yin-Yang dataset. Even though the model achieves an average accuracy over multiple seeds of 96.1 %, the boundaries between classes are not sharply visible from the maximum membrane values of the output neurons.

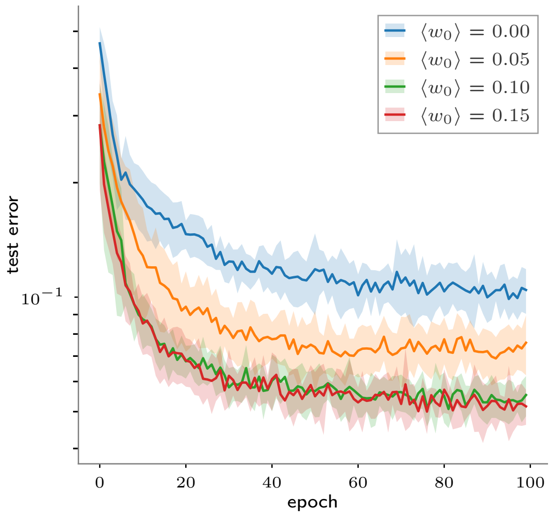
5.4.6 Dependence on Weight Initialization

A major limitation of the EventProp algorithm is that it does not take into account the influence of adding or deleting spikes. The gradient computed by solving the adjoint system and its jumps in its form presented in [Wunderlich and Pehle 2021], described in section 2.3.2 of this work, does return gradients that only quantify the shift of a spike time depending on a weight change. But, the possible emergence of new spikes or deletion of existing ones is not contained in those equations.

To study the consequences of this limitation, I trained the network described in section 5.4.2 on the Yin-Yang task with different hidden weight initializations. The weights are drawn from a normal distribution $\mathcal{N}(\langle w_0\rangle, \sigma^2_{w_0})$. The choice of $\langle w_0\rangle$ influences the activity of the hidden layer and as spike deletion or creation is not contained in the EventProp learning method, the activity does stay in the same



(a) Hidden spike rates



(b) Test error

Figure 5.12: Dependence of success in training on Yin-Yang on weight initialization. The network described in section 5.4.2 was trained with BSS-2 ITL on the Yin-Yang task for different values $\langle w_0 \rangle \in \{0.0, 0.05, 0.1, 0.15\}$, each for seeds 0 to 7, of the weight initialization distribution $\mathcal{N}(\langle w_0 \rangle, \sigma_{w_0}^2)$ with $\sigma_{w_0} = 0.2$. **(a)** The distribution of the hidden spikes per training sample is plotted for different stages of the training. In all cases, the rate stays in a certain range and learning to adjust weights for spike addition or deletion cannot be observed. **(b)** The average (line) and one standard deviation (shaded region) of the test set classification error throughout training on the Yin-Yang dataset is displayed for the different weight initializations. The success of training drastically depends on the weight initialization, as this determines the number of hidden spikes.

range throughout the whole training, which in turn drastically determines the final outcome (cf. fig. 5.12). This shows the missing property of the EventProp method to map the appearance or disappearance of spikes in the gradient. In the cases of small weight initializations $\langle w_0 \rangle \in \{0.0, 0.05\}$ a large portion of the hidden neurons never spike and could therefore simply be removed from the network, reducing its effective size. This partly explains the low classification accuracy for models with those weight initializations.

A possible solution would be to adjust the used loss function, as Nowotny et al. [2022] demonstrates. Another approach would be to study the incorporation of membrane observations of hidden neurons into the gradient estimation, which is already contained in the EventProp equations eqs. (2.45) and (2.53).

5.5 MNIST

As a second task I considered the MNIST dataset [LeCun et al. 1998]. I orientate myself strongly on the work of Cramer et al. [2022] regarding modifications made to the dataset, network topology and training procedure. They also trained SNNs on this task and use BSS-2 for training with hardware ITL.

In this section, I train a network using EventProp in simulation and present the results achieved on the MNIST dataset.

5.5.1 Experiment Setup and Training Procedure

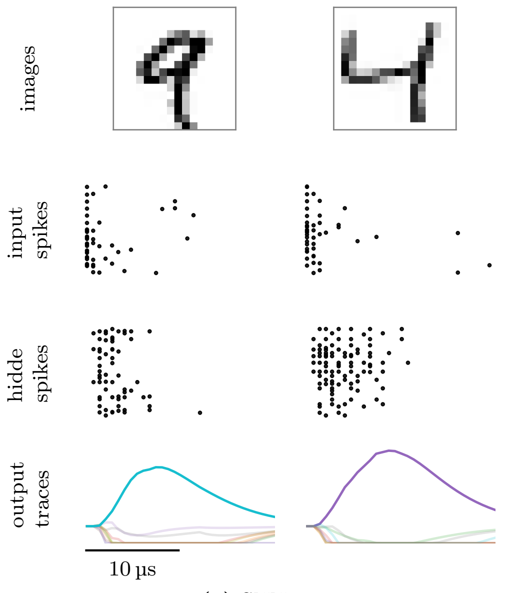
Even though, results in this section are only obtained for training in simulation, the network is constrained to a size suitable for hardware execution considering the limited resources on BSS-2. The original images, which are 28×28 pixels in size, are reduced to 16×16 pixels, to accomodate for the limited fan-in of BSS-2 while still allowing for a reasonably sized hidden layer. The reduction is done by cropping the two outer rows of an image and scaling the remaining image to the desired size.

The pixel values are encoded into spike latencies, according to the method proposed by Zenke and Vogels [2021]. The intensities of the pixels are first standardized into an interval $x \in [0, 1]$ and spike times are then determined by

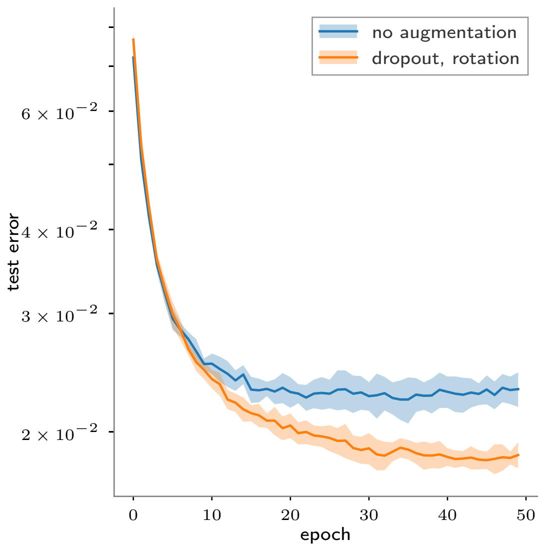
$$t_{\text{in}}(x) = \begin{cases} \tau_{\text{in}} \log \frac{x}{x - \vartheta_{\text{in}}} & \text{if } x > \vartheta_{\text{in}}, \\ \infty & \text{otherwise.} \end{cases} \tag{5.10}$$

The spike time t_{in} then corresponds to the time of a LIF neuron with time constant τ_{in} receiving an input current x. The parameter ϑ_{in} is the minimum current neccessary to provoke a spike.

The resulting spike times were used as an input to a feed-forward network of 246 hidden LIF and 10 output LI neurons. The network was trained for 50 epochs on a max-over-time loss together with readout regularization (cf. eqs. (5.8) and (5.9)). To apply weight updates, the Adam optimizer was used with a step-wise decaying



(a) SNN activity



(b) Test error

Figure 5.13: Results achieved on MNIST with SNNs and EventProp in simulation. **(a)** Activity of the trained SNN on example images. For visualization, output traces are clipped from below. The images of the dataset are latency coded into spike trains to serve as input to a hidden layer of LIF neurons. The spikes are projected onto a layer of readout LI neurons and samples are classified by the max-over-time membrane values of those output neurons. Target neuron traces are highlighted having a brighter color. **(b)** Error on the test set over training epochs. The model was trained with dropout and rotation and without those augmentations. The average error (line) and its standard deviation (shaded region) is shown for training with 10 different seeds.

learning rate. To avoid overfitting, the input images in training were randomly rotated within a specified range. Additionally, dropout is applied to silence a given fraction of the hidden neurons.

All testing and verification of experiment code and training procedure was done using a validation split of 5.000 random samples drawn and excluded from the training set. Only at the final stage, the test set was used to obtain the results described below.

5.5.2 Experiment Results

The results of training the model with 10 different seeds are depicted in fig. 5.13 together with exemplary network activity and output traces. The separation of target and non-target readout traces by lifting the former and inhibiting the latter has clearly been learned for the example images. The accuracy on the test set

Ref.	**Hidden**	**Loss**	**Gradient Estimator**	**Fwd.**	**Remarks**	**Acc. [%]**
(a)	246	TTFS	F&D	sim.		97.4 ± 0.2
				BSS-2		96.9 ± 0.1
(b)	300	MAX	EP	sim.	dropout	97.6 ± 0.1
(c)	246	MAX	SG	sim.		97.5 ± 0.1
				sim.	dropout, rotation	98.0 ± 0.0
				BSS-2		97.2 ± 0.1
				BSS-2	dropout, rotation	97.6 ± 0.1
this work	246	MAX	EP	sim.		97.7 ± 0.1
				sim.	dropout, rotation	98.2 ± 0.1

Table 5.3: Results achieved on MNIST for SNNs and different gradient estimators using explicit analytical expressions (F&D), surrogate gradients (SG) and EventProp (EP). Prior results on the MNIST task using SNNs with comparable hidden layer sizes are referenced from (a) [Göltz et al. 2021], (b) [Wunderlich and Pehle 2021], (c) [Cramer et al. 2022]. All networks use a single hidden layer with different number of neurons. Forward (Fwd.) execution of the network is either done by simulating the network (sim.) or on the BSS-2 system. The used loss functions are a loss based on time-to-first-spike decoding 'TTFS' and a max-over-time loss 'MAX'.

after training is $97.7 \pm 0.1\%$ without droupout and rotation on training images and $98.2 \pm 0.1\%$ with dropout and rotation. The results, together with results from previously published work of others, are listed in table 5.3.

This is an intermediate result on the way to execution on hardware. The network size is similar to the models used by Göltz et al. [2021] and Cramer et al. [2022], which both did train with BSS-2 ITL. To train this model with forward execution on hardware only the experiment code and EventProp implementation have to be aligned with the latest `hxtorch.snn` API, but underlying model architecture changes should not be required.

6 Conclusion

In this work, by leveraging the theoretical foundation of the EventProp algorithm [Wunderlich and Pehle 2021], I extended its applicability to the refractory *leaky-integrate and fire* (LIF) neuron model. By deriving a similar set of adjoint equations, I have demonstrated the potential for this approach to be used more broadly in neural network training. Moreover, I have explicitly shown the correspondence between EventProp and the approach of Göltz et al. [2021] that derives closed-form gradient equations, providing further evidence for the utility and versatility of this mathematical framework.

To demonstrate the ability of EventProp to be used for *in-the-loop* (ITL) training with neuromorphic hardware, I implemented and tested a time-discrete version and trained a network of LIF neurons with forward execution on *BrainScaleS-2* (BSS-2). Specifically, my results show that EventProp can successfully train such a network for an example task, paving the way for further exploration of this technique with neuromorphic systems.

The refractory LIF neuron introduces a period after activation of a neuron, in which further activation is suppressed. This can be introduced by adding an additional state variable with its own dynamics, depending on the refractory time τ_r. In section 3.1, I derived a set of equations similar to the EventProp algorithm and show equality of those equations in the case of a vanishing refractory time $\tau_\mathrm{r} \to 0$. This highlights the potential of the underlying approach, computing adjoint equations for dynamical systems with discontinuous transitions. It demonstrates that this lends itself to further studies of the computation of parameter gradients for other, potentially more complex neuron models.

In the case of LIF neurons, Göltz et al. [2021] derived closed-form equations for first spike times and gradients for fixed ratios of time constants, specifically $\tau_\mathrm{m} = \tau_\mathrm{s}$ and $\tau_\mathrm{m} = 2\tau_\mathrm{s}$. I derived an explicit expression for the weight gradient of the first spike time from the EventProp algorithm by integrating the dynamics and incorporating the jumps of the adjoint variables (cf. section 3.2 and eq. (3.25)). In the cases of equal or double time constants, I show that the gradient equation mentioned above is equal to the equations of weight gradients in the work of Göltz et al. [2021]. This closes the gap between using an adjoint dynamical system that needs to be solved to compute gradient values and explicitly solving the threshold condition of a neuron's membrane for the spike time and taking derivatives. An important fact to note is that only the first spike time was considered in my calculations. Recent work by Bacho and Chu [2022] extends the approach of Göltz et al. [2021] to multiple spike times and derives closed-form equations taking earlier spike times of the neuron itself into account and describes how this can be applied to learning in SNNs. It remains open work to be shown that those equations can also be derived starting with the set of equations provided in EventProp.

A time-discrete version of the EventProp algorithm, using simple forward Euler integration, was implemented in two PyTorch autograd functions (cf. section 5.1), which can be used with the high-level, PyTorch based `hxtorch.snn` API [Spilger et al. 2022]. This allows users to train SNNs in simulation and with the BSS-2 chip ITL by using EventProp for gradient estimation. The choice to split the algorithm into two separate functions is necessary due to the topological description of SNNs in `hxtorch.snn`. The description of projections in `hxtorch.snn.Synapse` modules allows for flexible topologies, e.g. different synapse types connecting to the same neuron populations contained in a `hxtorch.snn.Neuron` module. The theoretical framework in which EventProp is derived does not treat this seperately but as a closed network of neurons. Implementing this into closed all-in-one modules would require generating new modules for each indivdual synapse-neuron combination.

For a single-synapse experiment, the weight gradient of the first spike time for a range of weights was computed by simulation and with hardware ITL in section 5.2. The results were compared with the analytically expectated values using the equations derived by Göltz et al. [2021]. In simulation, the first spike time and estimated gradients converge to the analytical values for decreasing integration step sizes Δt of the implemented Euler scheme. Using BSS-2, the measured first spike time and numerically computed gradient are consistently close to the analytical expectation on average, although some variations between different neuron circuits were observed.

After this verification, a network with a single hidden layer of 120 LIF neurons was trained on the Yin-Yang task [Kriener et al. 2022], classifying samples by the maximum membrane voltage over time of three readout LI neurons, each corresponding to one class (cf. section 5.4). For ITL training with BSS-2 and the discrete EventProp implementation for gradient estimation, the model reached accuracies of $(96.1\pm0.8)\%$. This is comparable to other results of $(95.0\pm0.9)\%$ [Göltz et al. 2021] and $(94.6\pm0.7)\%$ (this work, with surrgate gradients) achieved with hardware ITL training on the BSS-2 platform.

An issue that remains with EventProp, is the lack of taking appearing or disappearing events and its influence on the networks output into account (cf. section 5.4.6). For surrogate gradients, this issue does not arise, since they explicitely take into account the membranes proximity to its threshold. Recent work of Nowotny et al. [2022] adresses this issue by studying different loss functions and their effect on the gradients. This certainly provides impactful insights, but does not solve the underlying problem of not considering spike creation or deletion in the fundamental description of the problem. Future work could address the appropriate description of the finite variations of the loss function due to infinitesimal parameter changes and the resulting (dis-)appearance of spikes.

In section 5.5, I describe the training of a SNN in simulation with 246 hidden LIF neurons and 10 LI readout neurons similarly sized to the ones used by Cramer et al.

[2022] and Göltz et al. [2021]. They trained their networks on a max-over-time and time-to-first-spike loss, respectively, with BSS-2 ITL. The network in this work was trained on the max-over-time loss within the `hxtorch.snn` framework. The model achieved a final accuracy of (98.2±0.1)% on the test set, using dropout and rotation as augmentation techniques during training to facilitate generalization. This is a slight improvement compared to other simulation results with comparable network sizes by Göltz et al. [2021], Cramer et al. [2022], Wunderlich and Pehle [2021], which are listed in table 5.3. At the time of the experiments, to be able to run this topology described in the high-level `hxtorch.snn` API, manually routing and placement of synapes would have been required, which I was not able to address in time. However, at the time of writing and thanks to the ongoing development of `hxtorch.snn` by the BrainScaleS software team, the experiment would now be possible on hardware using the high-level API.

This work is a another step in the direction to event-driven, information-efficient encoding and overall energy-efficient functional modeling of SNNs on neuromorphic systems. To make further progress into this direction, a fully event-based handling of gradient computations, instead of using grid based integration of dynamics, is inevitable and current work is done in this group to achieve such a treatment using JAX [Bradbury et al. 2018]. Other contributions to the progress in this direction includes to work of Bacho and Chu [2022], which allows for explicit spike time and gradient computations in multi-spike LIF networks.

Outlook

The adjoint sensitivity analysis used to derive EventProp can also be leveraged to obtain learning rules for neuron specific parameters like time constants, and leak, threshold, or reset voltages. This could greatly facilitate training by eliminating the need to individually tune these parameters, which was previously a tedious task of hyperparameter search.

At present, efforts are being made in the group to enhance the ecosystem in which the BSS-2 chip is integrated, with the goal of enabling multi-chip experiments either through low-latency chip-to-chip communication using different setups or by streamlining hybrid training with the subsequent execution of different model parts on one or more chips. These developments will facilitate the exploration of the applicability of EventProp in large-scale networks deployed on neuromorphic hardware. One limitation of the algorithm — which is also present in other learning methods — is the scaling of computations required to propagate errors back along the time dimension and through the deep spatial dimension of large-scale networks with an increasing number of parameters and dynamic variables.

To address this problem, online-learning rules have been developed, like e-prop [Bellec et al. 2020], and applied to training on BSS-2 [Arnold 2021]. In Chapter 5 of Pehle [2021], a set of equations is derived for online learning that builds on the same foundational principle of adjoint sensitivity analysis as EventProp. Other than the beforementioned online-learning rule e-prop, the novel approach of Pehle [2021] treats the model in continuous time and without turning to surrogate (or pseudo-) derivatives. As is already mentioned briefly there, the derivation of those equation could also be adjusted, e.g. to arrive at a hybrid algorithm partitioning learning into online parameters updates applied on a short time scale and batch-based updates computed on the host similar to the original EventProp method. Adapting and implementing this on BSS-2 hardware would be an intriguing challenge, as it could potentially provide a more exact and simultaneously efficient and scalable method for training SNNs in real time.

Bibliography

Elias Arnold. Biologically inspired learning in recurrent spiking neural networks on neuromorphic hardware. Master thesis, Heidelberg University, 2021.

Elias Arnold, Georg Böcherer, Florian Strasser, Eric Müller, Philipp Spilger, Sebastian Billaudelle, Johannes Weis, Johannes Schemmel, Stefano Calabrò, and Maxim Kuschnerov. Spiking neural network nonlinear demapping on neuromorphic hardware for im/dd optical communication. *Journal of Lightwave Technology*, pages 1–8, 2023. doi: 10.1109/JLT.2023.3252819.

Florian Bacho and Dominique Chu. Exact error backpropagation through spikes for precise training of spiking neural networks. *arXiv preprint*, 2022.

Guillaume Bellec, Franz Scherr, Anand Subramoney, Elias Hajek, Darjan Salaj, Robert Legenstein, and Wolfgang Maass. A solution to the learning dilemma for recurrent networks of spiking neurons. *Nature Communications*, 11(1):3625, 2020. doi: 10.1038/s41467-020-17236-y. URL `https://doi.org/10.1038/s41467-020-17236-y`.

Ben Varkey Benjamin, Nicholas A Steinmetz, Nick N Oza, Jose J Aguayo, and Kwabena Boahen. Neurogrid simulates cortical cell-types, active dendrites, and top-down attention. *Neuromorphic Computing and Engineering*, 1(1):013001, 2021.

Sebastian Billaudelle. *From transistors to learning systems: Circuits and algorithms for brain-inspired computing*. Phd thesis, Heidelberg University, May 2022.

James Bradbury, Roy Frostig, Peter Hawkins, Matthew James Johnson, Chris Leary, Dougal Maclaurin, George Necula, Adam Paszke, Jake VanderPlas, Skye Wanderman-Milne, and Qiao Zhang. JAX: composable transformations of Python+NumPy programs, 2018. URL `http://github.com/google/jax`.

R. Brette and W. Gerstner. Adaptive exponential integrate-and-fire model as an effective description of neuronal activity. *J. Neurophysiol.*, 94:3637 – 3642, 2005. doi: 10.1152/jn.00686.2005.

Benjamin Cramer, Sebastian Billaudelle, Simeon Kanya, Aron Leibfried, Andreas Grübl, Vitali Karasenko, Christian Pehle, Korbinian Schreiber, Yannik Stradmann, Johannes Weis, et al. Surrogate gradients for analog neuromorphic computing. *Proceedings of the National Academy of Sciences*, 119(4), 2022.

Mike Davies, Narayan Srinivasa, Tsung-Han Lin, Gautham Chinya, Yongqiang Cao, Sri Harsha Choday, Georgios Dimou, Prasad Joshi, Nabil Imam, Shweta Jain, et al. Loihi: A neuromorphic manycore processor with on-chip learning. *IEEE Micro*, 38(1):82–99, 2018. doi: 10.1109/MM.2018.112130359.

Andrew P. Davison, Daniel Brüderle, Jochen Eppler, Jens Kremkow, Eilif Muller, Dejan Pecevski, Laurent Perrinet, and Pierre Yger. PyNN: a common interface for neuronal network simulators. *Front. Neuroinform.*, 2(11), 2009. doi: 10.3389/neuro.11.011.2008.

Steve B. Furber, David R. Lester, Luis A. Plana, Jim D. Garside, Eustace Painkras, Steve Temple, and Andrew D. Brown. Overview of the SpiNNaker system architecture. *IEEE Transactions on Computers*, 99(PrePrints), 2012. ISSN 0018-9340. doi: 10.1109/TC.2012.142.

Santos Galán, William F. Feehery, and Paul I. Barton. Parametric sensitivity functions for hybrid discrete/continuous systems. *Applied Numerical Mathematics*, 31 (1):17–47, 1999. ISSN 0168-9274. doi: https://doi.org/10.1016/S0168-9274(98)00125-1.

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

Wulfram Gerstner, Werner Kistler, Richard Naud, and Liam Paninski. *Neuronal Dynamics*. Cambridge University Press, 2014.

Julian Göltz, Laura Kriener, Andreas Baumbach, Sebastian Billaudelle, Oliver Breitwieser, Benjamin Cramer, Dominik Dold, Ákos Ferenc Kungl, Walter Senn, Johannes Schemmel, Karlheinz Meier, and Mihai A. Petrovici. Fast and energy-efficient neuromorphic deep learning with first-spike times. *Nature Machine Intelligence*, 3(9):823–835, 2021. doi: 10.1038/s42256-021-00388-x.

Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. `http://www.deeplearningbook.org`.

T. H. Gronwall. Note on the derivatives with respect to a parameter of the solutions of a system of differential equations. *Annals of Mathematics*, 20(4):292–296, 1919. ISSN 0003486X. URL `http://www.jstor.org/stable/1967124`.

Alan Lloyd Hodgkin and Andrew F. Huxley. A quantitative description of membrane current and its application to conduction and excitation in nerve. *J Physiol*, 117 (4):500–544, August 1952. ISSN 0022-3751. URL `http://view.ncbi.nlm.nih.gov/pubmed/12991237`.

David Iberri. File:action potential.svg, 2007. URL `https://upload.wikimedia.org/wikipedia/commons/4/4a/Action_potential.svg`. [Online; accessed February 27, 2023].

Quasar Jarosz. File:neuron, langneutral.svg, 2009. URL https://upload.wikimedia.org/wikipedia/commons/0/08/Neuron%2C_LangNeutral.svg. [Online; accessed February 28, 2023].

Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *International Conference on Learning Representations*, 2014.

Laura Kriener, Julian Göltz, and Mihai A. Petrovici. The yin-yang dataset. In *Neuro-Inspired Computational Elements Conference*, NICE 2022, pages 107–111, New York, NY, USA, 2022. Association for Computing Machinery. ISBN 9781450395595. doi: 10.1145/3517343.3517380.

Gregory M. Kurtzer, Vanessa Sochat, and Michael W. Bauer. Singularity: Scientific containers for mobility of compute. *PLOS ONE*, 12(5):1–20, 05 2017. doi: 10.1371/journal.pone.0177459.

Louis Lapicque. Recherches quantitatives sur l'excitation electrique des nerfs traitee comme une polarization. *Journal de Physiologie et Pathologie General*, 9:620–635, 1907.

Y. LeCun, L. Bottou, Y. Bengioa, and P. Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep learning. *Nature*, 521 (7553):436–444, may 2015. ISSN 0028-0836. doi: http://dx.doi.org/10.1038/nature1453910.1038/nature14539.

Christian Mayr, Sebastian Hoeppner, and Steve Furber. Spinnaker 2: A 10 million core processor system for brain simulation and machine learning. *arXiv preprint arXiv:1911.02385*, 2019.

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

Paul A. Merolla, John V. Arthur, Rodrigo Alvarez-Icaza, Andrew S. Cassidy, Jun Sawada, Filipp Akopyan, Bryan L Jackson, Nabil Imam, Chen Guo, Yutaka Nakamura, et al. A million spiking-neuron integrated circuit with a scalable communication network and interface. *Science*, 345(6197):668–673, 2014. doi: 10.1126/science.1254642.

Saber Moradi, Ning Qiao, Fabio Stefanini, and Giacomo Indiveri. A scalable multicore architecture with heterogeneous memory structures for dynamic neuromorphic asynchronous processors (DYNAPs). *IEEE Trans. Biomed. Circuits Syst.*, 12(1):106–122, 2018.

Hesham Mostafa. Supervised learning based on temporal coding in spiking neural networks. *IEEE transactions on neural networks and learning systems*, 29(7): 3227–3235, 2017.

Eric Müller, Elias Arnold, Oliver Breitwieser, Milena Czierlinski, Arne Emmel, Jakob Kaiser, Christian Mauch, Sebastian Schmitt, Philipp Spilger, Raphael Stock, Yannik Stradmann, Johannes Weis, Andreas Baumbach, Sebastian Billaudelle, Benjamin Cramer, Falk Ebert, Julian Göltz, Joscha Ilmberger, Vitali Karasenko, Mitja Kleider, Aron Leibfried, Christian Pehle, and Johannes Schemmel. A scalable approach to modeling on accelerated neuromorphic hardware. *Front. Neurosci.*, 16, 2022. ISSN 1662-453X. doi: 10.3389/fnins.2022.884128.

Eric Christian Müller. *Novel Operation Modes of Accelerated Neuromorphic Hardware*. Phd thesis, Heidelberg University, Dec 2014.

Emre O. Neftci, Hesham Mostafa, and Friedemann Zenke. Surrogate gradient learning in spiking neural networks: Bringing the power of gradient-based optimization to spiking neural networks. *IEEE Signal Processing Magazine*, 36(6):51–63, 2019. doi: 10.1109/MSP.2019.2931595.

Thomas Nowotny, James P. Turner, and James C. Knight. Loss shaping enhances exact gradient learning with eventprop in spiking neural networks. *arXiv preprint*, 2022.

Garrick Orchard, E. Paxon Frady, Daniel Ben Dayan Rubin, Sophia Sanborn, Sumit Bam Shrestha, Friedrich T. Sommer, and Mike Davies. Efficient neuromorphic signal processing with loihi 2. In *2021 IEEE Workshop on Signal Processing Systems (SiPS)*, pages 254–259, 2021. doi: 10.1109/SiPS52927.2021.00053.

Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. Automatic differentiation in pytorch. 2017.

Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems 32*, pages 8024–8035. Curran Associates, Inc., 2019.

Christian Pehle and Jens Egholm Pedersen. Norse - A deep learning library for spiking neural networks, January 2021. URL `https://doi.org/10.5281/zenodo.4422025`. Documentation: https://norse.ai/docs/.

Christian Pehle, Sebastian Billaudelle, Benjamin Cramer, Jakob Kaiser, Korbinian Schreiber, Yannik Stradmann, Johannes Weis, Aron Leibfried, Eric Müller, and Johannes Schemmel. The BrainScaleS-2 accelerated neuromorphic system with hybrid plasticity. *Frontiers in Neuroscience*, 16, 2022. ISSN 1662-453X. doi: 10.3389/fnins.2022.795876.

Christian-Gernot Pehle. *Adjoint Equations of Spiking Neural Networks*. Phd thesis, Heidelberg Univeristy, February 2021.

Mihai A. Petrovici. *Function vs. Substrate: Theory and Models for Neuromorphic Hardware*. Phd thesis, Heidelberg University, July 2015.

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

E. N. Rozenvasser. General sensitivity equations of discontinuous systems. *Avtomatika i Telemekhanika*, 3:52–56, 1967.

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

Johannes Schemmel, Sebastian Billaudelle, Philipp Dauer, and Johannes Weis. *Accelerated Analog Neuromorphic Computing*, pages 83–102. Springer International Publishing, Cham, 2022. ISBN 978-3-030-91741-8. doi: 10.1007/978-3-030-91741-8_6. URL https://doi.org/10.1007/978-3-030-91741-8_6.

Philipp Spilger, Elias Arnold, Luca Blessing, Christian Mauch, Christian Pehle, Eric Müller, and Johannes Schemmel. hxtorch.snn: Machine-learning-inspired spiking neural network modeling on brainscales-2. *arXiv preprint*, 2022.

Paul J. Werbos. Backpropagation through time: what it does and how to do it. *Proceedings of the IEEE*, 78(10):1550–1560, 1990. doi: 10.1109/5.58337.

Timo C Wunderlich and Christian Pehle. Event-based backpropagation can compute exact gradients for spiking neural networks. *Scientific Reports*, 11(1):1–17, 2021. doi: 10.1038/s41598-021-91786-z.

Friedemann Zenke and Surya Ganguli. SuperSpike: Supervised learning in multilayer spiking neural networks. *Neural Computation*, 30(6):1514–1541, June 2018. ISSN 1530-888X. doi: 10.1162/neco_a_01086.

Friedemann Zenke and Tim P. Vogels. The remarkable robustness of surrogate gradient learning for instilling complex function in spiking neural networks. *Neural Computation*, 33(4):899–925, 03 2021. ISSN 0899-7667. doi: 10.1162/neco_a_01367.

Acronyms

BSS-2 BrainScaleS-2 1–3, 6, 18–22, 33, 35–37, 39, 41–44, 46, 47, 49–60, 62, 64–66, 77, 91

AdEx adaptive exponential integrate-and-fire 1, 19

ANN artificial neural network 1, 8, 50, 51, 58

API application programming interface 22, 43, 62, 64, 65

ASIC application-specific integrated circuit 19, 33

CADC columnar analog-to-digital converter 19, 22, 36, 37

CUBA current-based 7, 11, 14, 36

FPGA field-programmable gate array 19, 20

ITL in-the-loop 2, 3, 18, 35, 39, 41, 43, 44, 46, 47, 49, 51–57, 59, 60, 62, 64, 65, 77, 78, 86, 88, 91

LI leaky integrator 8, 22, 44, 45, 49–51, 58, 60, 61, 64, 77

LIF leaky-integrate and fire 2, 5, 6, 8, 10, 11, 14, 16, 17, 20–22, 25–28, 39, 44–46, 48–50, 60, 61, 63–65, 77, 83

ODE ordinary differential equation 7, 11, 25, 27

OOM orders of magnitude 52–54

PSC post-synaptic transmembrane current 5

PSP post-synaptic potential 35–37

RNN recurrent neural network 2, 10

SNN spiking neural network 1, 2, 5, 8–11, 17, 19, 21, 33, 39, 44, 50, 52, 58, 60–62, 64–66, 77, 78

Part I

Appendix

A Lists

A.1 List of Figures

A.2 List of Tables

B Custom PyTorch autograd functions

Listing B.1: Discrete EventProp in custom PyTorch autograd functions.

```python
class EventPropNeuron(torch.autograd.Function):
  # pylint: disable=line-too-long
  """
  Define gradient using adjoint code (EventProp) from norse
  """
  # Allow redefining builtin for PyTorch consistancy
  # Allow names z (spikes), v (membrane), i (current) and T (time dimension
    length)
  # Allow different argument params to use dt, tau_mem_inv etc.
  # pylint: disable=redefined-builtin, invalid-name, arguments-differ
  @staticmethod
  def forward(ctx, input: torch.Tensor,
        params: NamedTuple) -> Tuple[torch.Tensor]:
    """
    Forward function, generating spikes at positions > 0.

    :param input: Weighted input spikes in shape (2, batch, time, neurons).
      The 2 at dim 0 comes from stacked output in EventPropSynapse.
    :param params: LIFParams object holding neuron prameters.

    :returns: Returns the spike trains and membrane trace.
      Both tensors are of shape (batch, time, neurons)
    """
    input_current = input[0]
    z, i, v = (
      torch.zeros(input_current.shape[0], input_current.shape[2]),
      torch.zeros(input_current.shape[0], input_current.shape[2]),
      torch.empty(input_current.shape[0],
            input_current.shape[2]).fill_(params.v_leak),
    )
    spikes, current, membrane = [z], [i], [v]
    T = input_current.shape[1]
    for ts in range(T - 1):
      # Current
      i = i * (1 - params.dt * params.tau_syn_inv) + input_current[:, ts]
      current.append(i)

      # Membrane
      dv = params.dt * params.tau_mem_inv * (params.v_leak - v + i)
      v = dv + v

      # Spikes
      spike = torch.gt(v - params.v_th, 0.0).to((v - params.v_th).dtype)
      z = spike

      # Reset
      v = (1 - z.detach()) * v + z.detach() * params.v_reset

      # Save state
      spikes.append(z)
      membrane.append(v)
    forward_result = (
      torch.stack(spikes).transpose(0, 1),
      torch.stack(membrane).transpose(0, 1)
    )
    ctx.current = torch.stack(current).transpose(0, 1)
    ctx.save_for_backward(input, *forward_result)
    ctx.extra_kwargs = {"params": params}

    return (*forward_result,)

  # pylint: disable=invalid-name
  @staticmethod
  def backward(ctx, grad_spikes: torch.Tensor,
         _: torch.Tensor) -> Tuple[Optional[torch.Tensor], ...]:
    """
    Implements 'EventProp' for backward.

    :param grad_spikes: Backpropagted gradient wrt output spikes.
    :param _: backpropagated gradient wrt to membrane trace (not used)

    :returns: Gradient given by adjoint function lambda_i of current
    """
    # input and layer data
    input = ctx.saved_tensors[0]
    input_current = input[0]
    z = ctx.saved_tensors[1]
    params = ctx.extra_kwargs["params"]

    # adjoints
    lambda_v, lambda_i = torch.zeros_like(z), torch.zeros_like(z)

    try:
      i = ctx.current
    except AttributeError:
      i = torch.zeros_like(z)
      # compute current
      for ts in range(z.shape[1] - 1):
        i[:, ts + 1] = \
          i[:, ts] * (1 - params.dt * params.tau_syn_inv) \
          + input_current[:, ts]

    for ts in range(z.shape[1] - 1, 0, -1):
      dv_m = params.v_leak - params.v_th + i[:, ts - 1]
      dv_p = i[:, ts - 1]

      lambda_i[:, ts - 1] = lambda_i[:, ts] + params.dt * \
        params.tau_syn_inv * (lambda_v[:, ts] - lambda_i[:, ts])
      lambda_v[:, ts - 1] = lambda_v[:, ts] * \
        (1 - params.dt * params.tau_mem_inv)

      output_term = z[:, ts] / dv_m * grad_spikes[:, ts]
      output_term[torch.isnan(output_term)] = 0.0

      jump_term = z[:, ts] * dv_p / dv_m
      jump_term[torch.isnan(jump_term)] = 0.0

      lambda_v[:, ts - 1] = (
        (1 - z[:, ts]) * lambda_v[:, ts - 1]
        + jump_term * lambda_v[:, ts - 1]
        + output_term
      )
    return torch.stack((lambda_i / params.tau_syn_inv,
              lambda_v - lambda_i)), None


class EventPropSynapse(torch.autograd.Function):
  """
  Synapse function for proper gradient transport when using EventPropNeuron
  """
  @staticmethod
```

```python
  # pylint: disable=arguments-differ, redefined-builtin
  def forward(ctx, input: torch.Tensor, weight: torch.Tensor,
        _: torch.Tensor = None
        ) -> Tuple[torch.Tensor, torch.Tensor]:
    """
    This should be used in combination with EventPropNeuron. Apply linear
    to input using weight and use a stacked output in order to be able to
    return correct terms according to EventProp to previous layer and
    weights.

    :param input: Input spikes in shape (batch, time, in_neurons)
    :param weight: Weight in shape (out_neurons, in_neurons)
    :param _: Bias, which is unused here

    :returns: Returns stacked tensor holding weighted spikes and
      tensor with zeros but same shape
    """
    ctx.save_for_backward(input, weight)
    output = input.matmul(weight.t())
    return torch.stack((output, torch.zeros_like(output)))

  @staticmethod
  # pylint: disable=arguments-differ, redefined-builtin
  def backward(ctx, grad_output: torch.Tensor,
          ) -> Tuple[Optional[torch.Tensor],
              Optional[torch.Tensor]]:
    """
    Split gradient_output coming from EventPropNeuron and return
    weight * (lambda_v - lambda_i) as input gradient and
    - tau_s * lambda_i * input (i.e. lambda_i at spiketimes)
    as weight gradient.

    :param grad_output: Backpropagated gradient with shape (2, batch, time,
      out_neurons). The 2 is due to stacking in forward.

    :returns: Returns gradients w.r.t. input, weight and bias (None)
    """
    input, weight = ctx.saved_tensors
    grad_input = grad_weight = None

    if ctx.needs_input_grad[0]:
      grad_input = grad_output[1].matmul(weight)
    if ctx.needs_input_grad[1]:
      grad_weight = \
        grad_output[0].transpose(1, 2).matmul(input)

    return grad_input, grad_weight, None
```

C Gradient Sign in EventProp

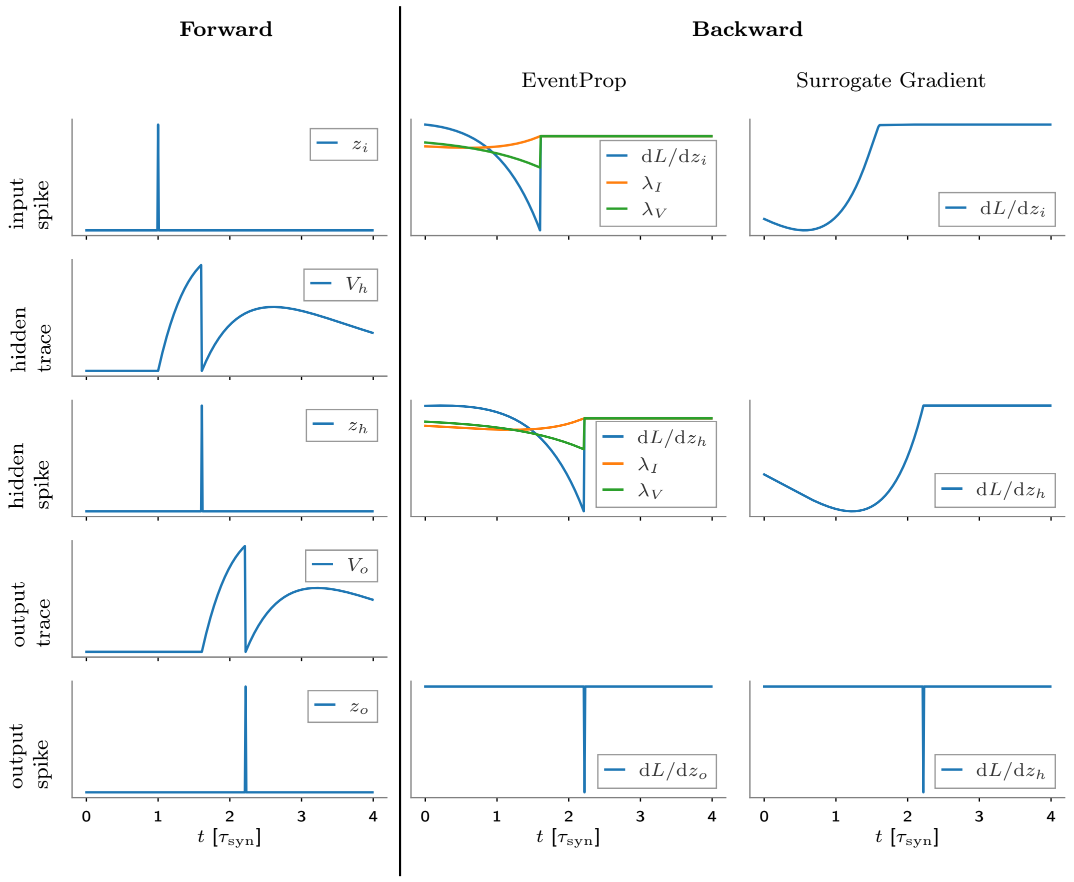


Figure C.1: Gradients for two consecutive LIF neurons and first spike time of the output neuron. A single input spike is sent to a LIF neuron, triggers a post-synaptic spike, which is forwarded to another LIF output neuron. The gradient of the first spike time with respect to the ouput spike train is computed using the differentiable spike time decoder described in section 4.1. The further backpropagation through the two layers is visualized for gradient estimation using the discrete EventProp implementation of this thesis and surrogate gradients.

D Training Parameters

Parameter	EeventProp / Surrogate Gradient
input size	5
hidden size	120
output size	3
weight init	[mean, stdev]
hidden	[1.0, 0.4]
output	[0.01, 0.04]
τ_m, τ_s	0.008
dt	0.00125
t_bias [τ_s]	0
t_early [τ_s]	0.15
t_late [τ_s]	2
t_sim [τ_s]	6.25
training epochs	50
batch size	20 / 16
optimizer	Adam
Adam parameter β	(0.9, 0.999)
Adam parameter ϵ	10^{-8}
learning rate	0.001
SG parameter α	- / 50

Table D.1: Parameters of neuron dynamics, network and training used to produce the initial training results in simulation on the Yin-Yang dataset using EventProp (EP) and surrogate gradients (SG) for gradient estimation, displayed in fig. 5.7.

Parameter	**EeventProp / Surrogate Gradient**
Calibration	
τ_m, τ_s	8 µs
τ_r	1 µs
`leak`	80
`reset`	80
`threshold`	125
`i-synin-gm`	800
`synapse-dac-bias`	600
Experiment	
input size	5
hidden size	120
output size	3
weight init	[mean, stdev]
hidden	[0.8, 0.4] / [1.0, 0.4]
output	[0.01, 0.04]
dt	1 µs / 0.5 µs
t_shift	-2 µs / -1.5 µs
t_bias	0 µs
t_early	1 µs
t_late	16 µs
t_sim	50 µs
training epochs	30
batch size	16
optimizer	Adam
Adam parameter β	(0.9, 0.999)
Adam parameter ϵ	10^{-8}
learning rate	0.001
SG parameter α	- / 50
trace offset	-50
trace scale	0.03
weight scale	50

Table D.2: Parameters of calibrartion, neuron dynamics, network and training used to produce the initial training results with hardware ITL on the Yin-Yang dataset using EventProp (EP) and surrogate gradients (SG) for gradient estimation, displayed in fig. 5.7.

Parameter	**EeventProp / Surrogate Gradient**
input size	5
hidden size	120
output size	3
weight init	[mean, stdev]
hidden	[1.0, 0.4]
output	[0.01, 0.04]
τ_m, τ_s	0.01
dt $[\tau_\mathrm{s}]$	0.01
t_bias $[\tau_\mathrm{s}]$	0
t_early $[\tau_\mathrm{s}]$	0
t_late $[\tau_\mathrm{s}]$	4
t_sim $[\tau_\mathrm{s}]$	6
training epochs	300
batch size	50 / 100
optimizer	Adam
Adam parameter β	(0.9, 0.999)
Adam parameter ϵ	10^{-8}
learning rate	0.0005 / 0.001
lr-scheduler	StepLR
lr-scheduler step size	50
lr-scheduler γ	0.5
readout reg.	0.0004
SG parameter α	- / 150

Table D.3: Parameters of neuron dynamics, network and training used to produce the training results, after appliying modifications from section 5.4.4, in simulation on the Yin-Yang dataset using EventProp (EP) and surrogate gradients (SG) for gradient estimation, displayed in fig. 5.10 and table 5.2.

Parameter	**EeventProp / Surrogate Gradient**
Calibration	
τ_m, τ_s	6 µs
τ_r	2 µs
`leak`	80
`reset`	80
`threshold`	150
`i-synin-gm`	500
`synapse-dac-bias`	1000
Experiment	
input size	25
hidden size	120
output size	3
weight init	[mean, stdev]
hidden	[0.2, 0.2] / [0.001, 0.15]
output	[0.01, 0.04] / [0.0, 0.1]
τ_m, τ_s	6 µs
dt	0.5 µs
t_shift	-2 µs
t_bias	2 µs
t_early	2 µs
t_late	26 µs
t_sim	36 µs
training epochs	300
batch size	50 / 100
optimizer	Adam
Adam parameter β	(0.9, 0.999)
Adam parameter ϵ	10^{-8}
lr-scheduler	StepLR
lr-scheduler step size	50
lr-scheduler γ	0.5
learning rate	0.0005 / 0.001
SG parameter α	- / 150
readout reg.	0.0004
trace offset	-48
trace scale	0.0145
weight scale	figs. D.1 and D.2

Table D.4: Parameters of calibration, neuron dynamics, network and training used to produce the training results, after appliying modifications from section 5.4.4, with hardware ITL on the Yin-Yang dataset using EventProp (EP) and surrogate gradients (SG) for gradient estimation, displayed in fig. 5.10 and table 5.2. The same weight scale was used for EP and SG training, but was adjusted depending on the used chip setup.

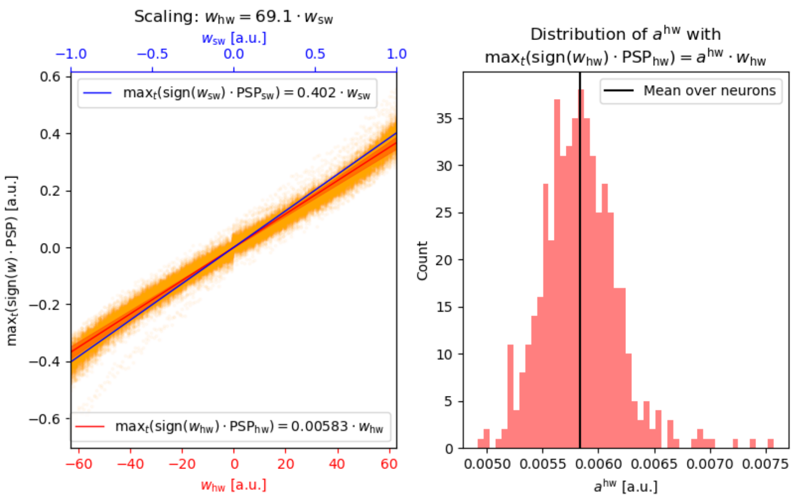


Figure D.1: Weight scaling measurement on setup W66F0. The scaling factor from software to hardware is $a_{\text{sw}}/a_{\text{hw}} = 69.1$.

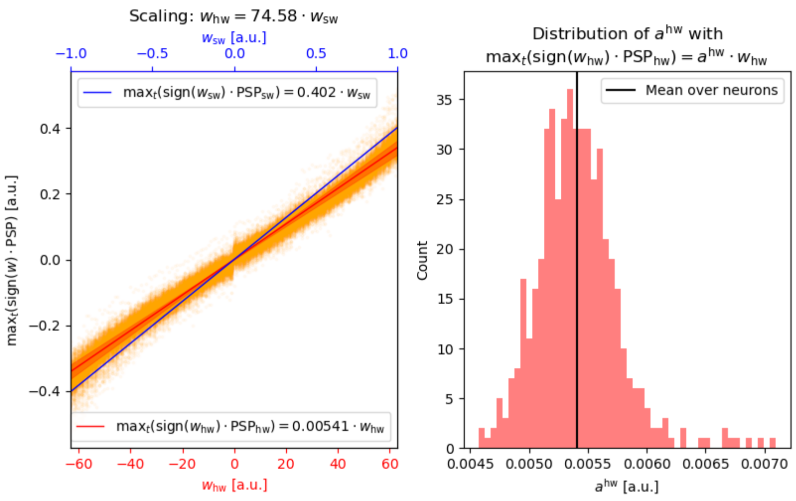


Figure D.2: Weight scaling measurement on setup W66F3. The scaling factor from software to hardware is $a_{\text{sw}}/a_{\text{hw}} = 74.58$.

Parameter	**EventProp**
input time constant τ_{in}	8×10^{-6}
input threshold ϑ_{in}	0.2
input size	256
hidden size	246
output size	10
hidden weight init [mean, stdev]	[0.05, 0.1]
τ_{m}, τ_{s} [a.u.]	6×10^{-6}
dt [a.u.]	5×10^{-7}
t_{sim} [a.u.]	2×10^{-5}
training epochs	50
batch size	25
optimizer	Adam
Adam parameter β	(0.9, 0.999)
Adam parameter ϵ	10^{-8}
lr-scheduler	StepLR
lr-scheduler step size	1
lr-scheduler γ	0.97
learning rate	0.0015
readout regularization	0.01
dropout	0 / 0.1
random rotation	0° / 10°

Table D.5: Parameters of neuron dynamics, network and training used to produce the results in simulation on the downscaled MNIST16x16 dataset using EventProp for gradient estimation. The achieved results are listed in table 5.3. Results of this work are given for models trained either without or with dropout and rotation (settings separated by "/").

E Gradient Distribution

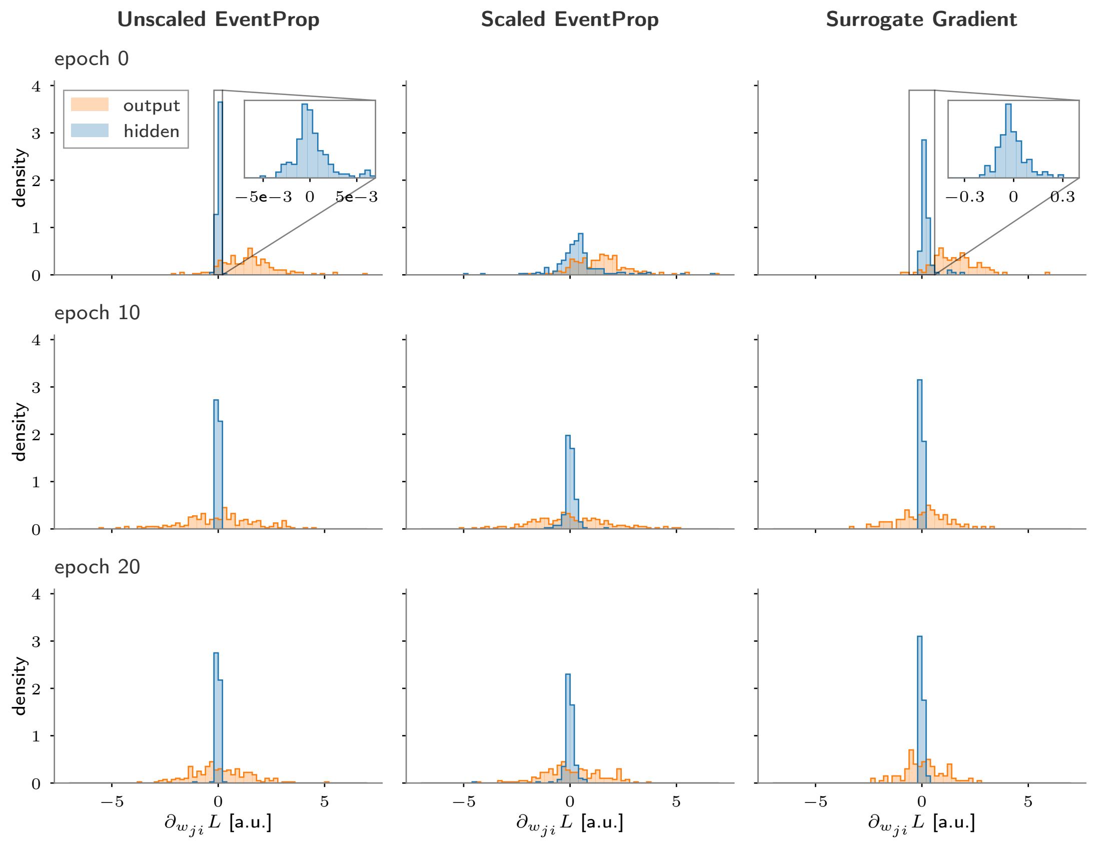


Figure E.1: Layer-specific gradient distribution for unscaled and scaled EventProp and with surrogate gradients. The distribution of gradients observed in the hidden (blue) and output layer (orange) during BSS-2 ITL training are shown for EventProp using the true gradient (left) in eq. (2.48), for the gradients estimated with EventProp and scaled by τ_s^{-1} (center), and for estimation using surrogate gradients (right).

F Software State

Experiments conducted in this thesis used the `singularity` [Kurtzer et al. 2017] container `/containers/stable/2022-11-28_1.img` and `dls` app. Experiments on the Yin-Yang dataset used the software state in table F.1, MNIST experiment the state in table F.2.

Repository	Commit-Hash	Commit Message
bss-hw-params	8e5e18ebbdece63890eebd4f082084111846e965	Increase for "Add omnibus to AXI-lite bridge"
code-format	b22b8fb9ef00b1a6c6294b029f45870e0ef847c4	Add pylint ignore for generated members in pygrenade_vx.logical_network
fisch	76a7d174d07e6409492c9504ff760e77bde0849f	Install headers into PREFIX/include/ for use in JIT PPU compiler
flange	2eb4e0dcf54c40e77505ef6a91334a96432f2a9b	Add withCcache to Jenkinsfile
grenade	3b5c9d49721e60e7748df461f5f9b836f7388f5a	Fixes for new log4cxx
halco	b1c4cba743f13d996ec91f1d77357dadcc0e9c0f	Fix clang warnings
haldls	cd42b67f6b5f32b72581f6b9336a273e4e3091e4	Reduce build parallelism
hate	0143bd6e177cdc2ec1f49e9276884078f3976b22	Add for_each(function, tuples...)
hwdb	8ebd5607a602ce91a90f567bfd13711e0a5363fe	Add optional extoll_node_id to HX cube FPGA
hxcomm	ef99900e12eafbd52fe0b1603a6bf3bd2611e15a	Fixes for new log4cxx
hxtorch	85abda8208976708ec0f82bf06a56e4fcd401d33	Add pytorch lightning YinYang example
lib-boost-patches	ed89665b4c066629b69617ede2e8b1fbe65822d9	Install headers into PREFIX/include/ for use in JIT PPU compiler
lib-rcf	8f928103ada425be321f33eb599e93eebabd81f4	Fixes for new log4cxx
libnux	302407544a65231f3b63d6a82e66146810712e8a	Fix fstrings
logger	87c2b33573dd63141861a6ab96a4a5d0de145b42	Fixes for new log4cxx
pywrap	8eda91fcca8bfccb946a0ee5b40ca82b5b15650e	Install headers into PREFIX/include/ for use in JIT PPU compiler
rant	0d494ce6eedfb74889cf7cee09105258819acb35	Install headers into PREFIX/include/ for use in JIT PPU compiler
sctrltp	42a988e986906f177102813418d5fd22dd646b44	Fix configure for modern pybind11
visions-slurm	8f41ea4f5bd1573d8f4623e9ed698a29f30036a3	Define env name for quiggeldy client username
ztl	773660f435e56b1ee7b962e8babfe004ff487cdd	Clean up test/wscript

Table F.1: Yin-Yang experiments software state

Repository	Commit-Hash	Commit Message
bss-hw-params	8e5e18ebbdece63890eebd4f082084111846e965	Increase for "Add omnibus to AXI-lite bridge"
code-format	b22b8fb9ef00b1a6c6294b029f45870e0ef847c4	Add pylint ignore for generated members in pygrenade_vx.logical_network
fisch	9db0b1ee31649391af90771c79bb5dbc5ff0d1e5	Add support for systime-correction barrier
flange	2eb4e0dcf54c40e77505ef6a91334a96432f2a9b	Add withCcache to Jenkinsfile
grenade	9abe51b2889b93507e8debc606dae4114db6d75f	Fix add print of realtime duration to ExecutionTimeInfo
halco	4f6d4ffaa9d293b95dad63d7c7727de3c4b5f6e4	Allow to transform LogicalNeuronOnDLS to list of AtomicNeurons
haldls	cd42b67f6b5f32b72581f6b9336a273e4e3091e4	Reduce build parallelism
hate	0143bd6e177cdc2ec1f49e9276884078f3976b22	Add for_each(function, tuples...)
hwdb	be09114e962e4bbf7bf571c0816f79fda07e23b9	Swap W60F3 and W65F3 chips for link debuggung
hxcomm	680bdcfd6678e5a561b0ba884d65dcde571523ff	Add support for systime-correction barrier
lib-boost-patches	ed89665b4c066629b69617ede2e8b1fbe65822d9	Install headers into PREFIX/include/ for use in JIT PPU compiler
lib-rcf	8f928103ada425be321f33eb599e93eebabd81f4	Fixes for new log4cxx
libnux	0542e2ba632f7d7135005b227e76431ccba530e1	Reduce test runtime of soft fractional saturating arithmetic
logger	87c2b33573dd63141861a6ab96a4a5d0de145b42	Fixes for new log4cxx
pywrap	8eda91fcca8bfccb946a0ee5b40ca82b5b15650e	Install headers into PREFIX/include/ for use in JIT PPU compiler
rant	0d494ce6eedfb74889cf7cee09105258819acb35	Install headers into PREFIX/include/ for use in JIT PPU compiler
sctrltp	42a988e986906f177102813418d5fd22dd646b44	Fix configure for modern pybind11
visions-slurm	8f41ea4f5bd1573d8f4623e9ed698a29f30036a3	Define env name for quiggeldy client username
ztl	773660f435e56b1ee7b962e8babfe004ff487cdd	Clean up test/wscript
hxtorch	aa8b2caa4d59f0834e7c6043717ba212a39e9fa1	Add MNIST EventProp example.

Table F.2: MNIST experiments software state

Acknowledgments

I want to express my gratitude to

- Dr. Johannes Schemmel for giving me the oportunity to write my thesis in the Electronic Vision(s) group and supervising my work.
- Elias Arnold, Dr. Christian Pehle and Dr. Eric Müller for their helpful guidance, countless discussions and personal support throughout the work on my thesis.
- Philipp Spilger for his enormous patience and helpfulness with all software-related questions.
- Sebastian Billaudelle and Julian Göltz for offering a relaxed atmosphere especially in the final stages of this work.
- the complete Electronic Vision(s) group for the support and open, welcoming atmosphere.
- Elias and Moritz for being great office neighbours and friends.
- my family, who has always been there for me and supported me no matter what path I took or what decisions I made.
- Line, who pushed me to start my master's thesis and has always supported me along the way and with everything else in life.

The work carried out in this Master's Thesis used systems and software, which received funding from the EC Horizon 2020 Framework Programme under grant agreements 785907 (HBP SGA2) and 945539 (HBP SGA3), the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) under Germany's Excellence Strategy EXC 2181/1-390900948 (the Heidelberg STRUCTURES Excellence Cluster), the German Federal Ministry of Education and Research under grant number 16ES1127 as part of the Pilotinnovationswettbewerb '*Energieeffizientes KI-System*', and the Lautenschläger-Forschungspreis 2018 for Karlheinz Meier.

Erklärung:

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

Heidelberg, den 30.03.2023 ....[illegible]........