Department of Physics and Astronomy

University of Heidelberg

Bachelor thesis

in Physics

submitted by

Florian Fischer

born in Mannheim

2025

Event-based Learning of Synaptic Delays and Arbitrary Topologies

This Bachelor thesis has been carried out by

Florian Fischer

at the

Kirchhoff Institute for Physics

under the supervision of

Prof. Dr. Johannes Schemmel

Event-basiertes Lernen von synaptischen Verzögerungen und arbiträre Topologien

Das Lernen von Verzögerungen ist ein entscheidender Schritt zur Verbesserung der zeitlichen Verarbeitungsfähigkeiten von spikenden neuronalen Netwerken. Dadurch lassen sich komplexere Probleme lösen, die sich vor allem durch eine zeitliche Struktur auszeichnen und bei denen präzise Zeiten wichtig sind, wie zum Beispiel Spracherkennung oder Sinnesverarbeitung. In dieser Arbeit wird der EventProp Algorithmus auf die Berechnung von exakten Gradienten von synaptischen Verzögerungen ausgeweitet. Event-basierte Methoden für den Forward Pass sowie den Backward Pass beim Trainieren von spikenden neuronalen Netzen bieten großes Potential, wenn es darum geht, energieeffiziente Lösungen zu finden. Außerdem passt dies sehr gut zu ereignisgesteuerter neuromorpher Hardware wie BrainScaleS-2. Die Methode zum Lernen von Verzögerungen wird ereignisbasiert in der Python-Bibliothek jaxsnn implementiert. Dies wird verwendet, um ein Modell auf dem Yin-Yang-Datensatz zu trainieren. Außerdem wird die jaxsnn Bibliothek zur Unterstützung von Machine Learning für arbiträre Topologien erweitert und eine Methode zum Aufschreiben von beliebig zusammensetzbaren Netzwerken eingeführt. Dabei werden auch rekurrente Netze mit Feedback-Verbindungen unterstützt. Dies wird schließlich in einigen kleinen Beispielen veranschaulicht.

Event-based Learning of Synaptic Delays and Arbitrary Topologies

Delay learning in spiking neural networks is a critical step in enhancing the network's temporal processing capabilities. It allows for more complex problems to be solved, especially for tasks with rich temporal structure where precise timing is crucial such as speech or sensory processing. In this work, the EventProp algorithm is extended to the calculation of exact gradients with respect to synaptic delays. Making use of event-based methods for both the forward and backward pass for training spiking neural networks, has high potential when it comes to exploiting sparsity and resulting efficiency gains. And it fits very well with event-driven neuromorphic hardware platforms like BrainScaleS-2. This method for delay learning is also implemented in an event-based fashion for the python library jaxsnn. Using this, a network is trained on the Yin-Yang dataset. Furthermore, the support for machine learning with arbitrary topologies is added to jaxsnn and a composable network definition paradigm is introduced. This also supports recurrent networks with feedback connections. These features will be displayed in simple examples, which also make use of the EventProp method for the delay case.

Contents

1 Introduction **5**

2 Theoretical Background **7**

2.1 Biological Neuron 7
2.2 Leaky Integrate-and-Fire Model 8
2.3 Encoding and Decoding Techniques 10
2.4 Machine Learning with Spiking Neural Networks 10
2.5 Approaches for Exact Gradients 12
2.6 Neuron Delays 13

3 Existing Software **15**

3.1 JAX 15
3.2 jaxsnn 16

4 Derivation of exact gradients for synaptic delays **19**

4.1 Gradients with respect to the weights 19
4.2 Gradients with respect to the synaptic delays 27
4.3 Additional Remarks 28

5 Implementation **30**

5.1 Arbitrary Topologies 30
5.2 Synaptic Delays 35
5.3 Membrane Potential and Current Traces 35

6 Experiments **37**

6.1 Arbitrary Topologies 37
6.2 Synaptic Delays 41

7 Discussion and Outlook **43**

8 References **46**

A Detailed Derivation of Gradients with Respect to Synaptic Delays **50**

B Training parameters **56**

1 Introduction

Due to the remarkable success of deep learning with artificial neural networks (ANNs) in recent years, ANNs have attracted a significant amount of attention [2]. They have already outperformed humans on many tasks, achieving superior accuracy and speed. For instance, they have completely transformed the field of computer vision, accomplishing breakthroughs in tasks such as object recognition, image segmentation or facial recognition [39]. Among other factors, a big part of these performance improvements can be attributed to the massive upscaling of model sizes and the drastic increase in the number of training parameters [17]. However, these advancements come at a cost, as the energy demands for training these models have surged [35].

Spiking Neural Networks (SNNs), which have been called the third generation of neural networks [22], present a promising solution to these challenges. SNNs are biologically-inspired networks that more closely mimic the behavior of the brain. In those networks a temporal dimension is introduced, allowing for the sparse and asynchronous activity to evolve dynamically. Unlike traditional neural networks, which process information with continuous activation functions, SNNs rely on discrete spikes. As stated by Maass [22], SNNs exploit both spatial and temporal patterns and are computationally more powerful than traditional neural networks.

Moreover, the sparse, event-driven processing of SNNs poses great opportunities when it comes to specialized hardware design. Brain-inspired computing systems have the potential to be transformative for energy-efficient and real-time computing in the future [34]. One example for a neuromorphic hardware platform is the BrainScaleS-2 (BSS-2) ASIC [30], a mixed-signal neuromorphic substrate. It is designed to emulate the behavior of spiking neural networks by representing the neuronal differential equations with electrical circuits. Since SNNs inherently rely on temporal information, they are extremely well suited for tasks that have a time dependent input like real-time sensory processing in fields such as robotics or autonomous systems [5].

Furthermore, SNNs as well as neuromorphic hardware present many possibilities for biology-focused research. They can help researchers model neuronal dynamics to gain insights on the processes that happen in the brain. This can then be used to investigate the learning mechanisms of the brain by for example simulating spike-time-dependent plasticity rules [21].

Applying approaches from classical machine learning to spiking neural networks, however, is challenging mainly because of the discrete state transitions, which happen at spike times. One approach to bypass this problem was proposed by Neftci, Mostafa, and Zenke [28], where surrogate gradients are introduced. In this method, the SNN is discretized in time and the forward pass is calculated by a time-step based evolution of the neuronal dynamics. For the

backward pass, the "hard" threshold is replaced with a differentiable approximation to allow backpropagation through time (BPTT). While this learning algorithm is widely used in practice, it is only an approximation and is not fully aligned with the event-driven nature of SNNs and neuromorphic hardware.

It therefore would be a natural choice to aim for an event-based simulation and backpropagation framework. A method utilizing exact spike time formulas for the leaky integrate-and-fire model was proposed by Göltz et al. [11], where they consider special cases of the synaptic time constants. In this algorithm the forward pass happens event-based and the backward pass is calculated with the derivatives of the differentiable spike time functions. The EventProp algorithm [37] takes this one step further where the adjoint method is used to derive exact gradients without restrictions on the synaptic time constants. In this method the adjoint variables are evolved backwards in time and sampled at spike times to calculate the gradients. This allows for both the forward and backward passes to be computed in an event-based fashion or let the forward pass be emulated on neuromorphic hardware.

SNNs are governed by temporal dynamics and the communication between neurons happens with discrete spikes. In this context, the notion of delays that influence the propagation of spikes arises naturally. Moreover, delays are biologically plausible because information cannot be transmitted instantly. Recently, an increasingly popular research topic has been the learning of delays. An example of such a learning method was proposed by Hammouamri, Khalfaoui-Hassani, and Masquelier [14] where the delays are learned through dilated convolution with learnable spacing (DCLS). Moreover, Göltz et al. [12] extended their framework for exact gradients to support the learning of synaptic delays by differentiating the exact spike time functions.

In this work the EventProp method is extended to synaptic delays to both support the training of weights with constant delays as well as the gradient calculation with respect to both weights and synaptic delays. This is then also implemented in an event-based fashion in the python library jaxsnn. Moreover, arbitrary topologies are implemented for event-based machine learning or simulations of SNNs.

2 Theoretical Background

2.1 Biological Neuron

Neurons are the basic building blocks of our nervous system. They allow us to perceive, process and respond to the world around us. While a single or a handful of neurons might not be very powerful for processing information, their real strength lies in the complex networks that they form. Our brain is believed to have neurons on the order of 10^{11} and synapses on the order of 10^{15} [38]. However, incredibly, the brain with this massive network of interconnected neurons only consumes about 20 watts of power on average [4].

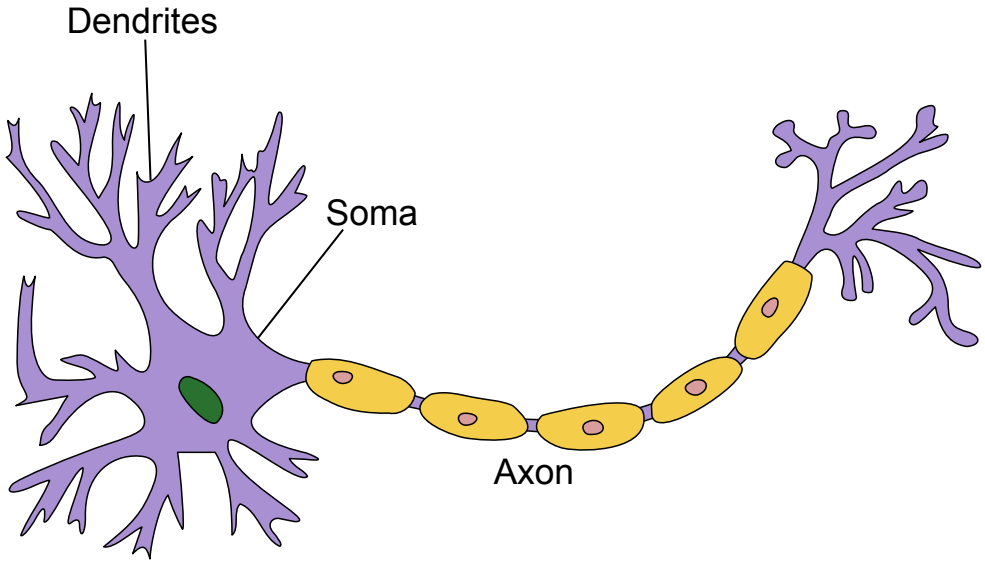


Figure 2.1: Biological neuron, adapted from Jarosz [16].

Now the basic mechanisms of biological neurons will be explained, based on the explanations by Zhang [38]. In fig. 2.1, the basic structure of a neuron is displayed. The dendrites are branch-like extensions around the cell body, which receive electrical input signals from other neurons. The soma (cell body) contains the nucleus and it is also where the incoming spikes are integrated. These lead to a change in the neuron's membrane potential, which at rest lies around -70mV. If the membrane potential reaches a certain threshold, it generates an action potential, also referred to as a spike. When this happens, voltage-gated sodium (Na^+) channels open,

causing a rapid depolarization of the neuron. While the action potential is then propagated through the axon, the membrane potential of the presynaptic neuron is reset. Next, the spike arrives at a synapse, where the release of neurotransmitters is triggered, which eventually leads to the signal reaching the postsynaptic neuron via its dendrites. As a result, the membrane potential of the postsynaptic neuron is changed which can again lead to the emission of an action potential.

The strength of the connection between a presynaptic and postsynaptic neuron is governed by the synaptic weight. It directly determines the magnitude of the postsynaptic response to incoming spikes. Not all spikes have a depolarizing, increasing effect on the membrane potential. Synapses are categorized into excitatory and inhibitory, which either cause an increase or a decrease in the postsynaptic potential.

2.2 Leaky Integrate-and-Fire Model

The leaky integrate-and-fire (LIF) neuron is a simple yet powerful mathematical model to describe neuronal dynamics. It goes back to Lapicque [20] who proposed a mathematical description in 1907. The LIF model only captures the very basic aspects of neuronal dynamics and does not model the biophysical processes in detail. However, it balances mathematical simplicity with biological plausibility, making it very widely used, especially for SNN-inspired machine learning. Other more sophisticated models include the AdEx [7] or the Hodgkin-Huxley model [15].

The following explanations of the LIF model are inspired by the ones from Gerstner et al. [10]. The basic idea is that neurons receive an input current which is integrated over time. This leads to a change in the membrane potential. However, the membrane potential naturally decays over time, which causes it to converge towards a resting potential in the absence of inputs.

This dynamical behavior of the membrane potential can be represented by a parallel circuit, composed of a capacitor with capacitance C and a resistor with resistance R. For the described circuit the following differential equation can be derived,

$$\tau_{\text{mem}} \frac{\mathrm{d}V}{\mathrm{d}t} = -[V(t) - V_{\text{rest}}] + RI(t) \tag{2.1}$$

where in the biological context $I(t)$ refers to the synaptic input current and $V(t)$ to the membrane potential. Furthermore, V_{rest} corresponds to the resting membrane potential and $\tau_{\text{mem}} = RC$ represents the membrane time constant.

As already mentioned in section 2.1, the membrane potential when crossing a specific threshold is reset to a predefined value V_{reset}. This corresponds to a discrete transition when the following condition is satisfied,

$$V(t^{\text{s}}) - \vartheta = 0, \tag{2.2}$$

where t^{s} is the time of the spike. Using $-$ and $+$ to denote variables before and after the transition as done by Wunderlich and Pehle [37], we get the following jump of the membrane potential at the transition

$$V^{+}(t^{\mathrm{s}}) = V_{\mathrm{reset}}. \tag{2.3}$$

An essential feature of neurons in the brain is their connectivity. The connections via synapses allow the propagation of spikes through the network. Similarly, LIF neurons can be extended to connected networks. One choice to do this, is to link the spikes of presynaptic neurons to the synaptic input current of postsynaptic neurons. A common approach is to choose an exponential model to describe the synaptic input dynamics. For this choice, the synaptic input current of a postsynaptic neuron also experiences a discrete state transition at the spike time t^{s}:

$$I^{+} = I^{-} + w, \tag{2.4}$$

where w is the weight that connects the presynaptic neuron to the postsynaptic neuron. For the case of no delays between the neurons, the presynaptic and postsynaptic spike time are the same. Moreover, the free dynamics of the synaptic current $I(t)$ between state transitions are described by a simple exponential decay:

$$\tau_{\mathrm{syn}} \frac{\mathrm{d}}{\mathrm{d}t} I = -I. \tag{2.5}$$

Now we can consider spiking neural networks with N neurons, which are connected by a weight matrix $W \in \mathbb{R}^{N \times N}$. The state variables of the membrane potential and the synaptic current thus become vector-valued functions: $V : t \mapsto V(t) \in \mathbb{R}^N$ and $I : t \mapsto I(t) \in \mathbb{R}^N$. Table 2.1 shows the set of equations for a spiking neural network of N LIF neurons without delays. When excluding self-recurrent connections, the weight matrix W has a zero diagonal and $e_n \in \mathbb{R}^N$ is a unit vector with a 1 at index n.

Free dynamics	**Transition condition**	**Jumps at transition**
$\tau_{\mathrm{mem}} \frac{\mathrm{d}V}{\mathrm{d}t} = -V + I$	$(V)_{\mathrm{n}} - \vartheta = 0$	$(V^{+})_n = 0$
$\tau_{\mathrm{syn}} \frac{\mathrm{d}I}{\mathrm{d}t} = -I$	$(\dot{V})_{\mathrm{n}} \neq 0$ for any n	$I^{+} = I^{-} + We_n$

Table 2.1: The LIF model for a network of N neurons without delays, listed as by Wunderlich and Pehle [37]

2.3 Encoding and Decoding Techniques

Spiking neural networks intrinsically take spikes as input, which raises the question how the gap between real-world data and spike-based processing of SNNs can be bridged. It is desirable that SNNs can work with all kinds of data, such as images or sounds, which have to be encoded into spike trains. In the end, the spike-based output also needs to be decoded into interpretable results. The two most common approaches to input encoding are rate coding and temporal encoding.

For rate coding, the information is encoded via the frequency of spikes in a given time interval, where the frequency of spikes corresponds to the intensity of the signal. However, it does not fit very well with time-sensitive applications and requires long observation times to be accurate. Moreover, biologically, rate codes also seem to be less plausible in many cases, because the human brain is able to process object recognition tasks in around 150ms, which suggests that rate codes would be too slow [3].

For temporal encoding, intensities of input signals can be encoded using the precise timing of individual spikes [9]. For this approach, there is a crucial difference between a time-grid and continuous time based paradigm. As displayed in fig. 2.2, by introducing time steps, the resolution of the input gets, depending on the time step size, significantly worse compared to an event-based approach where the time of the spike is exact.

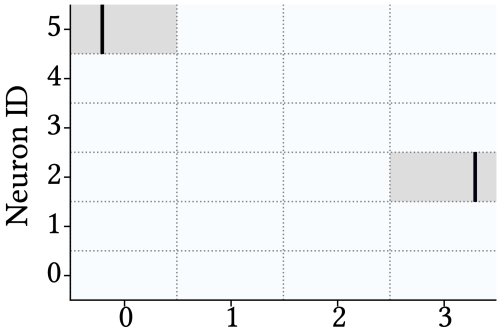


Figure 2.2: Continuous vs discrete spike times. Taken from [27]

One popular approach for decoding is to consider the max-over-time value of the membrane potential of non-spiking output neurons. Also, rate codes are used for decoding by considering the spike frequency of output spikes of the network. However, in an event-based approach, encoding with spike times is desired because it exploits the intrinsic sparsity, possibly exhibiting advantages for energy-efficient computing. For example, for classification tasks, time-to-first-spike (TTFS) [11] can be used, where the neuron that spikes first can be interpreted as the predicted class. Another way is to use the distance to specific target times for the output neurons of each class.

2.4 Machine Learning with Spiking Neural Networks

Machine Learning with artificial neural networks has achieved remarkable success across a wide range of fields. One of the key components behind this success has been the backpropagation algorithm which enables the optimization of neural networks through gradient descent. So the question arises how well these well-proven methods from traditional machine learning can be

applied to learning in spiking neural networks.

Traditional ANNs are typically multilayer networks of artificial neurons. In the forward pass of these networks, each neuron in a layer computes the weighted sum of its input and applies a non-linear activation function. A classical activation function is the sigmoid function

$$\sigma(x) = \frac{1}{1 + e^{-x}} \tag{2.6}$$

The result is then passed as input into the next layer. In the end, the output of the last layer is used to calculate a loss. This loss function is chosen to be high for wanted outputs and low for unwanted ones. To improve the performance of the network after the forward pass, the gradients with respect to the weights can be computed in order to update the network's parameters and hence improve its performance. This is done using backpropagation which makes use of the chain rule to propagate errors backward through the network.

Biologically inspired SNNs are fundamentally different from ANNs in mainly two ways: Time plays a crucial role because SNNs receive temporal input and also exhibit temporal dynamics in continuous time. And secondly, the transmission of information through the network happens with binary spikes whereas ANNs utilize continuous outputs.

To make SNNs compatible with the backpropagation methods of traditional machine learning, the continuous time dynamics can be changed to a time step based evolution by applying discretization methods on SNN dynamics. This way, SNNs almost resemble Recurrent Neural Networks (RNNs). RNNs are a type of ANN, which can handle sequential data and allow information to persist through recurrent connections [26]. These types of networks can also be optimized by applying backpropagation through time (BPTT) [36].

However, when trying to apply BPTT to discretized spiking neural networks, one encounters a problem which is caused by the binary nature of spikes. Since they only happen when a hard threshold is crossed, they correspond to a discrete activation function $\sigma(V) = \theta(V - \vartheta)$. One solution to this problem was proposed by Neftci, Mostafa, and Zenke [28] where they introduce surrogate gradients. In this method, the forward pass is computed using the normal spiking behavior of the SNN while for the backward pass, the discrete activation function is replaced with a differentiable function like a sigmoid (eq. (2.6)).

Although this method is widely used, it has certain limitations. On one hand, the gradients are only approximations, resulting from the exchange of the activation function in the backward pass. Moreover, the time-step based approach to spiking neural networks which are inherently event-based is something that may diminish some advantages like sparsity in data structures and temporal resolution. This leads to the search for exact event-based backpropagation algorithms. Those will be discussed in the next section.

2.5 Approaches for Exact Gradients

Exact gradients offer the potential to train with higher accuracies and to leverage sparse data structures. This could be particularly important for tasks that involve temporal precision or sparse input data. In this section, two approaches for exact gradients will be discussed.

Göltz et al. [11] look at the special cases of $\tau_{\text{mem}} = 2\tau_{\text{syn}}$ and $\tau_{\text{mem}} = \tau_{\text{syn}}$ to derive exact formulas for the time of the next spike T. Equation (2.7) shows the formula for $\tau_{\text{mem}} = 2\tau_{\text{syn}}$.

$$\frac{T}{\tau_{\text{syn}}} = 2\ln\left[\frac{2a_1}{a_2 + \sqrt{a_2^2 - 4a_1 g_l \vartheta}}\right] \tag{2.7}$$

where a_1 and a_2 are terms that depend on the times of previous spikes and g_l is the leak conductance of the neuron's membrane. In the forward pass, the next spike times can be found with the analytical solution and the state variables can be evolved to the transition times in an event-based fashion. The gradients of the spike times with respect to the weights can then be calculated in the backward pass because the formula is differentiable and thus compatible with backpropagation.

Another more general approach was first proposed by Wunderlich and Pehle [37] where they apply the adjoint method to the case of spiking neural networks. This leads to the derivation of adjoint equations, which define dynamics for adjoint variables $\lambda_V : t \mapsto \lambda_V(t) \in \mathbb{R}^N$ and $\lambda_I : t \mapsto \lambda_I(t) \in \mathbb{R}^N$. The gradients can be computed by a backward pass in which the adjoint variables are evolved backwards in time. This can be done event-based where discrete transitions happen at the spike times, which were obtained from the forward pass. Moreover, the gradients are updated at these spike times. They consider a loss of the form

$$\mathcal{L} = l_p(t^{\text{post}}) + \int_0^T l_V(V(t), t)\mathrm{d}t \tag{2.8}$$

where $l_p(t^{\text{post}})$ and $l_V(V(t), t)$ are smooth loss functions. The loss l_p depends on the postsynaptic spike times $t^{\text{post}} \in \mathbb{R}^{N_{\text{post}}}$, with N_{post} being the number of postsynaptic spikes. The loss l_V depends on the membrane potential V as well as the time t.

They show that the gradients with respect to a weight of a presynaptic neuron i and postsynaptic neuron j can be computed by sampling the adjoint variable λ_{I} of the postsynaptic neuron at the input spike times:

$$\frac{\mathrm{d}\mathcal{L}}{\mathrm{d}w_{ji}} = -\tau_{\text{syn}} \sum_{\text{spikes from i}} (\lambda_I)_j. \tag{2.9}$$

The adjoint equations governing the hybrid dynamics with state transitions of the adjoint variables are derived by Wunderlich and Pehle [37]. The jump transition of the adjoint variable λ_V of the spiking neuron n is given by eq. (2.10) and the adjoint equations are listed in table 2.2.

$$(\lambda_V^-)_{n(k)} = (\lambda_V^+)_{n(k)} + \frac{1}{\tau_{\text{mem}}(\dot{V}^-)_{n(k)}} \left[\vartheta(\lambda_V^+)_{n(k)} + \left(W^\top (\lambda_V^+ - \lambda_I) \right)_{n(k)} + \frac{\partial l_p}{\partial t_k^{\text{post}}} + l_V^- - l_V^+ \right] \tag{2.10}$$

Free dynamics	**Transition condition**
$\tau_{\text{mem}} \lambda_V' = -\lambda_V - \frac{\partial l_V}{\partial V}$	$t - t_k^{\text{post}} = 0$
$\tau_{\text{syn}} \lambda_I' = -\lambda_I + \lambda_V$	for any k

Table 2.2: The adjoint equations for a network of N neurons, from Wunderlich and Pehle [37]

These equations will later be compared with the ones that are derived in chapter 4 by extending the original derivation to synaptic delays.

2.6 Neuron Delays

Since spiking neural networks have an inherent temporal dimension, the natural question arises if the time between the firing of a presynaptic neuron and the time of arrival at the postsynaptic neuron can be adapted. Delays are very plausible because the transmission cannot occur instantaneously. This applies both to biological neurons as well as to neuromorphic hardware like BrainScaleS-2. Delays have also been observed to play a crucial role in neuronal dynamics, influencing processes such as synchronization, information processing and learning in biological networks [23]. The main distinction is between axonal, synaptic and dendritic delays.

Axonal delays refer to the time that it takes for the action potential to travel down the axon. A cause for these delays is myelination because myelinated axons conduct signals faster than unmyelinated ones [24]. As can be seen in fig. 2.3 (a), axonal delays vary for each presynaptic neuron. So for a connection between two given feed-forward layers, the axonal delays can be written as a vector of size n_I where n_I is the size of the presynaptic layer.

Synaptic delays are the most flexible because they occur at each connection between neurons. At this step, neurotransmitters are released and processed. Since each pair of neurons from an input and output layer has a different synaptic delay, the synaptic delays connecting two layers can be written as an $n_I \times n_{I+1}$ matrix where n_{I+1} is the size of the postsynaptic layer, see fig. 2.3.

Dendritic delays correspond to the time that electrical signals take to propagate within the dendritic tree of the postsynaptic neuron before reaching the soma. Since they vary for each postsynaptic neuron, in a multilayer feed-forward example, they can be written as a vector of size n_{I+1}.

Recently, there has been an increasing interest in the learning of delays. For example, Hammouamri, Khalfaoui-Hassani, and Masquelier [14] learn delays by modeling synaptic

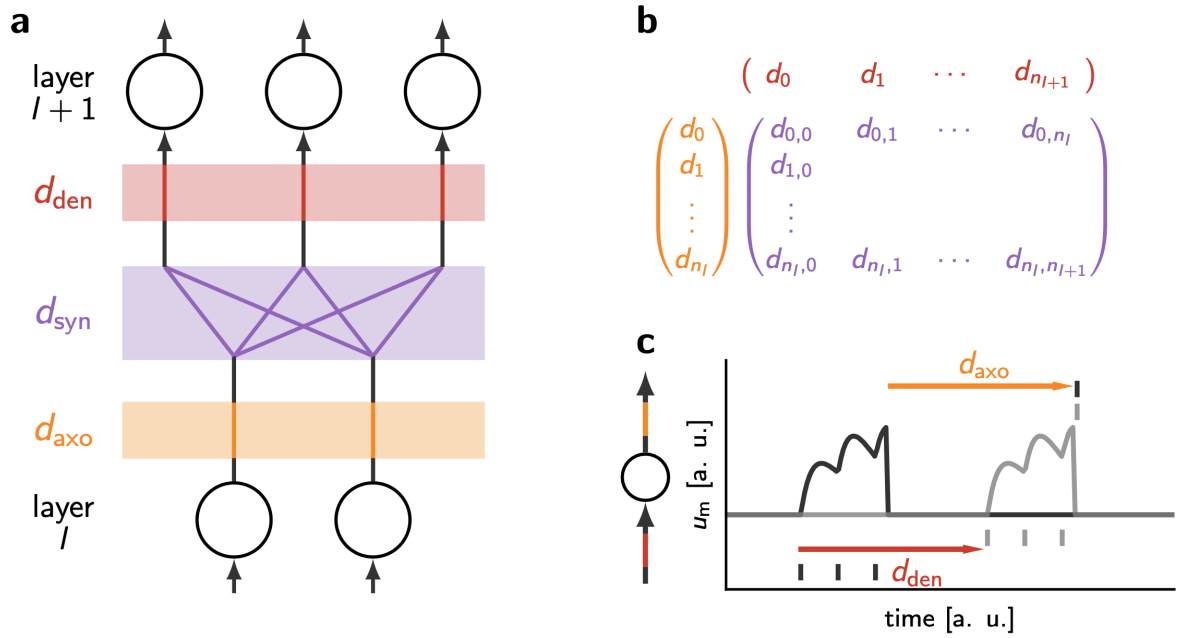


Figure 2.3: Showcasing the differences between the different types of delays, adapted from Göltz et al. [12].

connections as 1D convolutions across time, where each kernel has a weight representing the synaptic delay. The learning can be done alongside the synaptic weights using the Dilated Convolution with Learnable Spacings (DCLS) technique [18].

Another approach was done by Göltz et al. [12] where they extended their method for exact gradients to the learning of delays. As discussed in section 2.5, they use differentiable functions for the spike times, which can be easily extended by adding differentiable delay terms.

3 Existing Software

3.1 JAX

JAX [6] is a python package for high performance numerical computing. It provides a lightweight NumPy-like API for array-based computing. Its main advantages come from the support for powerful, composable function transformations. These include automatic differentiation (`jax.grad`), just-in-time (JIT) compilation (`jax.jit`), vectorization (`jax.vmap`) and parallelization (`jax.pmap`). Moreover, the code can directly be executed on different backends like CPU, GPU or TPU.

JAX achieves this by tracing Python functions when they are first called or the input shapes have changed. For this, functions are called on a tracer value, which records all the computations that happen on it. In this process, all JAX operations are reduced to a set of primitive operations defined in `jax.lax` which mirrors XLA (Accelerated Linear Algebra) [32]. The primitive operations are used to store the whole computational graph in JAX' intermediate representation JAXPR. Because it only consists of the predefined primitive operations, JAX can easily apply composable function transformations to it. Then it can be JIT-compiled into high level optimized code (HLO) which is read by XLA. Next, XLA compiles this HLO code and sends it to the requested backend.

However, these functionalities place certain constraints on the way JIT-compatible code can be written. JAX transformations are designed to only work with pure python functions. So the return values always have to be identical for the same arguments and the functions cannot have side effects like the mutation of non-local variables. Moreover, JAX does not allow in-place updates of arrays which can only be updated with functional array updates. Also, for code used within transformations the output arrays and intermediate arrays are required to have a static shape.

When it comes to control flow, also special care has to be taken. Code inside transformations does not support flexible python control flow that depends on the traced variables. This is because during tracing, all operations have to be recorded by passing in abstract tracer values where the concrete values are not available. However, JAX provides its own `jax.lax` primitives for control flow. Listing 1 shows how for example `jax.lax.cond` and `jax.lax.scan` can be used to replace if statements and loops.

JAX provides powerful automatic differentiation capabilities with `jax.grad` that are both efficient and flexible. It supports both forward- and reverse-mode differentiation. Moreover, it

allows higher order differentiation as well as computations of Jacobian and Hessian matrices. And it also permits flexible gradient override.

```python
import jax

# Example of loop control flow
def body_fun(carry, inputs):
    ...
    return carry, output

# Pass in initial carry value and inputs which are scanned
carry, output = jax.lax.scan(body_fun, initial_carry, inputs)

# Example of conditional control flow
def true_fun(val):
    ...
    return output

def false_fun(val):
    ...
    return output

res = jax.lax.cond(condition, true_fun, false_fun)
```

Listing 1: Example of JAX control flow operations using `jax.lax.scan` and `jax.lax.cond`.

3.2 jaxsnn

jaxsnn [27] is a python library for machine learning with SNNs. It is split up into a time step based and an event-based implementation. The time-step-based implementation was inspired by Norse [31]. In the following, the state of the event-based implementation before this thesis will be explained.

Only feed-forward layers are directly implemented. These feed forward layers are simulated sequentially where the output spikes of a given layer serve as the input queue of the next layer. The simulation happens for a predefined number of steps where at each step an event happens and is recorded. This event can either be an input spike, internal spike or no spike. The fundamental data structure are `EventPropSpike`s. They are `tree_math.struct`s which makes

them compatible with JAX' transformations.

```python
@dataclasses.dataclass
@tree_math.struct
class EventPropSpike:
    time: jax.Array # float
    idx: jax.Array # int
    current: jax.Array # float
```

Listing 2: Data structure for `Eventpropspikes`.

At each step, the type of the next event is determined first. Therefore, the time of the next spike of all neurons is calculated using the function `ttfs_solver` which is vectorized with `jax.vmap`. Then the earliest internal spike is selected. This can be done using the analytical equations from [11]. Then, this internal time is compared with the time of the next spike from the input queue. Depending on which time is smaller, the corresponding spike is chosen as the next event. If both spikes occur later than a predefined time t_{late}, no event is simulated and an empty event is stored.

If the next event is an input or internal event, all neurons in the layer are evolved to that point in time using the implemented function `lif_exponential_flow`. Then a discrete state transitions is applied with the `transition` function. This means that for input spikes, the synaptic input current of all neurons in the layer is increased by the corresponding weight. For internal spikes, only the membrane potential of the firing neuron is reset. Then, the spike is stored and the step is repeated. The data flow of the step function is visualized in fig. 3.1.

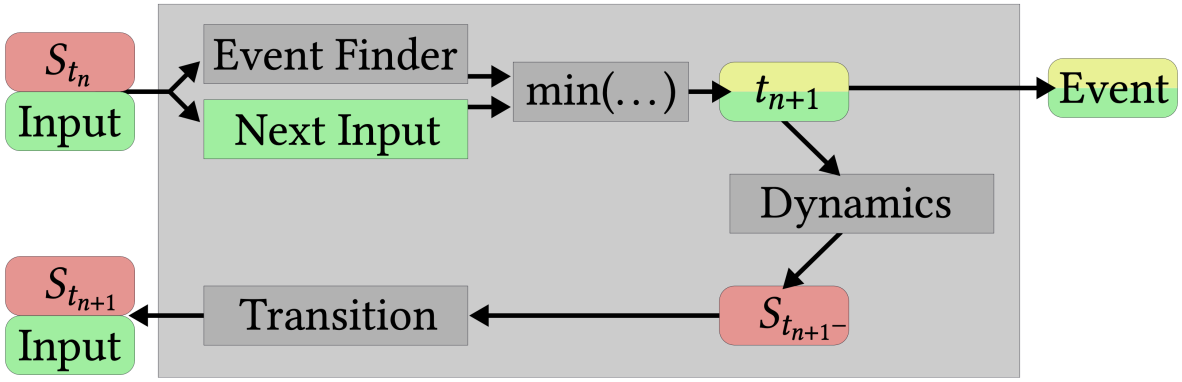


Figure 3.1: Data flow of the step function, adapted from Müller et al. [27].

The analytical gradients are calculated with JAX' automatic differentiation functionalities and can be used to train custom loss functions. However, also the EventProp algorithm [37] is

implemented for which a custom backward pass is defined. The custom backward pass contains a backward step function `step_bwd`, which evolves the adjoint variables backwards in time to the events that were calculated in the forward pass. If the event is an internal event, the discrete transition (2.10) is applied and if it is an input event the gradients with respect to the spiking input neuron are updated by adding $-\tau_{\text{syn}}\lambda_I$ as described in section 2.5.

4 Derivation of exact gradients for synaptic delays

In this section, the EventProp algorithm [37] is extended to support synaptic delays, as well as to calculate the gradients with respect to the delays. This derivation closely follows the original derivation [37], but some critical adjustments are made to generalize it to synaptic delays. First, the gradients with respect to the weights will be calculated for constant synaptic delays in section 4.1. Then, this will be extended to also calculate the gradients of the synaptic delays. As introduced in section 2.5, we consider a loss of the following form,

$$\mathcal{L} = l_p(t^{\text{pre}}) + \int_0^T l_V(V(t), t)\mathrm{d}t, \tag{4.1}$$

where it should be noted that t^{post} was changed to t^{pre} because the loss depends on the time at which the neurons fire. For the original derivation, this did not play any role because they had no delays and therefore had $t^{\text{pre}} = t^{\text{post}}$.

For a given spike event k, the presynaptic spike time t_k^{pre} is connected to the postsynaptic spike time t_{km}^{post} by the delay $d_{mn(k)}$:

$$t_{km}^{\text{post}} = t_k^{\text{pre}} + d_{mn(k)}, \tag{4.2}$$

where m and $n(k)$ are the indices of the post- and presynaptic neuron, respectively. We now consider a system of N recurrently connected neurons, excluding self connections. The differential equations governing the LIF dynamics, which were introduced in section 2.2, can be written in implicit form:

$$f_V = \tau_{\text{mem}}\dot{V} + V - I = 0 \tag{4.3}$$

$$f_I = \tau_{\text{syn}}\dot{I} + I = 0. \tag{4.4}$$

4.1 Gradients with respect to the weights

With the introduction of synaptic delays, the system experiences discrete state transitions at both the presynaptic and the postsynaptic spike times. This makes it natural to split up the integral at all times where discrete transitions happen. For simplicity reasons, we will first treat the special case where the postsynaptic spike time for all postsynaptic neurons is the same which corresponds to axonal delays. However, this can then later be easily extended to synaptic delays. Lagrange multipliers, $\lambda : t \mapsto \lambda(t) \in \mathbb{R}^N$, are introduced to constrain the system's dynamics to

f_I and f_V. Now, the gradient with respect to a weight w_{ji} of a postsynaptic neuron j and a presynaptic neuron i will be calculated:

$$\begin{aligned}\frac{\mathrm{d}\mathcal{L}}{\mathrm{d}w_{ji}} =& \frac{\mathrm{d}}{\mathrm{d}w_{ji}}\left[l_p(t^{\mathrm{pre}}) + \sum_{k=0}^{N_{\mathrm{pre}}} \int_{t_k^{\mathrm{pre}}+d_k}^{t_{k+1}^{\mathrm{pre}}} \left[l_V(V,t) + \lambda_V \cdot f_V + \lambda_I \cdot f_I\right] \mathrm{d}t \right. \\ &\left. + \int_{t_{k+1}^{\mathrm{pre}}}^{t_{k+1}^{\mathrm{pre}}+d_{k+1}} \left[l_V(V,t) + \lambda_V \cdot f_V + \lambda_I \cdot f_I\right] \mathrm{d}t \right],\end{aligned} \tag{4.5}$$

where d_k is a short notation for the delay corresponding to the event k and $\cdot$ signifies the dot product. This is possible because in the case of axonal delays each event has one corresponding delay d_k. We consider N_{pre} spikes with presynaptic spike times t_k^{pre}, $k \in 1, \ldots, N_{\mathrm{pre}}$. Consequently, for the integration to occur from 0 to T, the boundary integration limits are set to $t_0^{\mathrm{pre}} + d_0 = 0$ and $t_{N_{\mathrm{pre}}+1}^{\mathrm{pre}} + d_{N_{\mathrm{pre}}+1} = T$. As stated in [37], using Gronwall's theorem [13] and the commutativity of the derivatives, we get the following expressions:

$$\frac{\partial f_V}{\partial w_{ji}} = \tau_{\mathrm{mem}} \frac{\mathrm{d}}{\mathrm{d}t}\frac{\partial V}{\partial w_{ji}} + \frac{\partial V}{\partial w_{ji}} - \frac{\partial I}{\partial w_{ji}} \tag{4.6}$$

$$\frac{\partial f_I}{\partial w_{ji}} = \tau_{\mathrm{syn}} \frac{\mathrm{d}}{\mathrm{d}t}\frac{\partial I}{\partial w_{ji}} + \frac{\partial I}{\partial w_{ji}}. \tag{4.7}$$

The next steps are shown only for the first integral, but they work analogously for the second one. The corresponding terms and the loss l_p will be accounted for later. Using the Leibniz integral rule on the first integral leads to

$$\begin{aligned}\left(\frac{d\mathcal{L}}{dw_{ji}}\right)_{\text{integral 1}} =& \sum_{k=0}^{N_{\mathrm{pre}}} \left[\int_{t_k^{\mathrm{pre}}+d_k}^{t_{k+1}^{\mathrm{pre}}} \left[\frac{\partial l_V}{\partial V} \cdot \frac{\partial V}{\partial w_{ji}} + \lambda_V \cdot \left(\tau_{\mathrm{mem}} \frac{\mathrm{d}}{\mathrm{d}t}\frac{\partial V}{\partial w_{ji}} + \frac{\partial V}{\partial w_{ji}} - \frac{\partial I}{\partial w_{ji}} \right) \right.\right. \\ &\left.\left. + \lambda_I \cdot \left(\tau_{\mathrm{syn}} \frac{\mathrm{d}}{\mathrm{d}t}\frac{\partial I}{\partial w_{ji}} + \frac{\partial I}{\partial w_{ji}} \right) \right] dt + l_{V,k+1}^{-} \frac{\mathrm{d}t_{k+1}^{\mathrm{pre}}}{\mathrm{d}w_{ji}}\bigg|_{t_{k+1}^{\mathrm{pre}}} - l_{V,k}^{+} \frac{\mathrm{d}t_k^{\mathrm{pre}}}{\mathrm{d}w_{ji}}\bigg|_{t_k^{\mathrm{post}}} \right]\end{aligned} \tag{4.8}$$

where the terms outside the integral, which would include f_I/f_V vanish because $f_V = f_I = 0$ holds along all trajectories. Also, $l_{V,k}^{\pm}$ was introduced for the voltage loss before $(-)$ and after $(+)$ the transition and it was used that the delays do not depend on the weights. Now, we can use partial integration:

$$\int_{t_k^{\mathrm{pre}}+d_k}^{t_{k+1}^{\mathrm{pre}}} \lambda_V \cdot \frac{\mathrm{d}}{\mathrm{d}t}\frac{\partial V}{\partial w_{ji}} dt = -\int_{t_k^{\mathrm{pre}}+d_k}^{t_{k+1}^{\mathrm{pre}}} \dot{\lambda}_V \cdot \frac{\partial V}{\partial w_{ji}} \mathrm{d}t + \left[\lambda_V \cdot \frac{\partial V}{\partial w_{ji}}\right]_{t_k^{\mathrm{pre}}+d_k}^{t_{k+1}^{\mathrm{pre}}} \tag{4.9}$$

$$\int_{t_k^{\text{pre}}+d_k}^{t_{k+1}^{\text{pre}}} \lambda_I \cdot \frac{\mathrm{d}}{\mathrm{d}t}\frac{\partial I}{\partial w_{ji}} dt = -\int_{t_k^{\text{pre}}+d_k}^{t_{k+1}^{\text{pre}}} \dot{\lambda}_I \cdot \frac{\partial I}{\partial w_{ji}} \mathrm{d}t + \left[\lambda_I \cdot \frac{\partial I}{\partial w_{ji}}\right]_{t_k^{\text{pre}}+d_k}^{t_{k+1}^{\text{pre}}}. \tag{4.10}$$

By substituting this into (4.8) and collecting terms, we get

$$\begin{aligned}\left(\frac{d\mathcal{L}}{dw_{ji}}\right)_{\text{integral 1}} = &\sum_{k=0}^{N_{\text{pre}}} \left[\int_{t_k^{\text{pre}}+d_k}^{t_{k+1}^{\text{pre}}} \left[\left(\frac{\partial l_V}{\partial V} - \tau_{\text{mem}}\dot{\lambda}_V + \lambda_V\right)\cdot\frac{\partial V}{\partial w_{ji}} + \left(-\tau_{\text{syn}}\dot{\lambda}_I + \lambda_I - \lambda_V\right)\cdot\frac{\partial I}{\partial w_{ji}}\right]\mathrm{d}t\right. \\ &+ l_{V,k+1}^{-}\frac{\mathrm{d}t_{k+1}^{\text{pre}}}{\mathrm{d}w_{ji}}\bigg|_{t_{k+1}^{\text{pre}}} - l_{V,k}^{+}\frac{\mathrm{d}t_k^{\text{pre}}}{\mathrm{d}w_{ji}}\bigg|_{t_k^{\text{post}}} \\ &+\tau_{\text{mem}}\left[\lambda_V\cdot\frac{\partial V}{\partial w_{ji}}\right]_{t_k^{\text{pre}}+d_k}^{t_{k+1}^{\text{pre}}} + \tau_{\text{syn}}\left[\lambda_V\cdot\frac{\partial I}{\partial w_{ji}}\right]_{t_k^{\text{pre}}+d_k}^{t_{k+1}^{\text{pre}}}.\end{aligned} \tag{4.11}$$

Since the Lagrange multipliers are unconstrained, they can be chosen to satisfy the differential equations defined by the terms in the parentheses within the integral. By making this choice, the integral evaluates to 0 and we get the adjoint equations. Since adjoint variables are usually integrated from $t = T$ to $t = 0$, the time derivative is transformed to $\frac{\mathrm{d}}{\mathrm{d}t} \to -\frac{\mathrm{d}}{\mathrm{d}t}$. By writing the transformed derivative using $'$, we get the following adjoint equations which have to be fulfilled between transitions

$$\tau_{\text{mem}}\lambda_V' = -\lambda_V - \frac{\partial l_V}{\partial V} \tag{4.12}$$

$$\tau_{\text{syn}}\lambda_I' = -\lambda_I + \lambda_V. \tag{4.13}$$

The previous steps can be repeated for the second integral term which leads to the same adjoint equations. For the remaining terms, only the corresponding evaluation times have to be changed, which are simply the integration limits from eq. (4.5). By combining the terms from both integrals and also differentiating l_p we arrive at the following expression

$$\begin{aligned}\frac{d\mathcal{L}}{dw_{ji}} = &\sum_{k=0}^{N_{\text{pre}}}\left[\frac{\partial l_p}{\partial t_k^{\text{pre}}}\frac{\mathrm{d}t_k^{\text{pre}}}{\mathrm{d}w_{ji}} + l_{V,k+1}^{-}\frac{\mathrm{d}t_{k+1}^{\text{pre}}}{\mathrm{d}w_{ji}}\bigg|_{t_{k+1}^{\text{pre}}} - l_{V,k}^{+}\frac{\mathrm{d}t_k^{\text{pre}}}{\mathrm{d}w_{ji}}\bigg|_{t_k^{\text{post}}}\right. \\ &+\tau_{\text{mem}}\left[\lambda_V\cdot\frac{\partial V}{\partial w_{ji}}\right]_{t_k^{\text{pre}}+d_k}^{t_{k+1}^{\text{pre}}} + \tau_{\text{syn}}\left[\lambda_V\cdot\frac{\partial I}{\partial w_{ji}}\right]_{t_k^{\text{pre}}+d_k}^{t_{k+1}^{\text{pre}}} + l_{V,k+1}^{-}\frac{\mathrm{d}t_{k+1}^{\text{pre}}}{\mathrm{d}w_{ji}}\bigg|_{t_{k+1}^{\text{post}}} \\ &\left.-l_{V,k+1}^{+}\frac{\mathrm{d}t_{k+1}^{\text{pre}}}{\mathrm{d}w_{ji}}\bigg|_{t_{k+1}^{\text{pre}}} + \tau_{\text{mem}}\left[\lambda_V\cdot\frac{\partial V}{\partial w_{ji}}\right]_{t_{k+1}^{\text{pre}}}^{t_{k+1}^{\text{pre}}+d_{k+1}} + \tau_{\text{syn}}\left[\lambda_V\cdot\frac{\partial I}{\partial w_{ji}}\right]_{t_{k+1}^{\text{pre}}}^{t_{k+1}^{\text{pre}}+d_{k+1}}\right].\end{aligned} \tag{4.14}$$

Assuming parameter-independent initial conditions for V and I and also setting the initial

condition for the adjoint variables as $\lambda_V(T) = \lambda_I(T) = 0$ leads to all the boundary terms at $t = T$ and $t = 0$ being equal to zero:

$$0 = \lambda_V \cdot \frac{\partial V}{\partial w_{ji}}\bigg|_T = \lambda_I \cdot \frac{\partial I}{\partial w_{ji}}\bigg|_T = \lambda_V \cdot \frac{\partial V}{\partial w_{ji}}\bigg|_0 = \lambda_I \cdot \frac{\partial I}{\partial w_{ji}}\bigg|_0 \tag{4.15}$$

Because of the definitions $t_0^{\text{pre}} + d_0 = 0$ and $t_{N_{\text{pre}}+1}^{\text{pre}} + d_{k+1} = T$, the terms containing $\frac{\mathrm{d}t_k^{\text{pre}}}{\mathrm{d}w_{ji}}, \frac{\mathrm{d}t_{k+1}^{\text{pre}}}{\mathrm{d}w_{ji}}$ are also zero for $k = 0$ and $k = N_{\text{pre}}$.

$$0 = \frac{\partial l_p}{\partial t_0^{\text{pre}}} \frac{\mathrm{d}t_0^{\text{pre}}}{\mathrm{d}w_{ji}} = l_{V,0}^{+} \frac{\mathrm{d}t_0^{\text{pre}}}{\mathrm{d}w_{ji}}\bigg|_{t_0^{\text{post}}} = l_{V,N+1}^{-} \frac{\mathrm{d}t_{N+1}^{\text{pre}}}{\mathrm{d}w_{ji}}\bigg|_{t_{N+1}^{\text{pre}}} \tag{4.16}$$

With this simplification, we can now identify the boundary terms at the pre- and postsynaptic spike times as pairs where there is one evaluation before $(-)$ and one after $(+)$ the transition. Thus, we are only left with a sum over the N_{pre} spikes where there are terms evaluated before and after the presynaptic spike and analogously for the postsynaptic spike. The sum is split up into terms that are evaluated at the presynaptic spike time, denoted by ξ_k, and terms that are evaluated at the postsynaptic spike, denoted by ψ_k:

$$\frac{d\mathcal{L}}{dw_{ji}} = \sum_{k=1}^{N_{\text{pre}}} \xi_k + \psi_k \tag{4.17}$$

$$\begin{aligned} \xi_k = \Bigg[& \tau_{\text{mem}} \left(\lambda_V^- \cdot \frac{\partial V^-}{\partial w_{ji}} - \lambda_V^+ \cdot \frac{\partial V^+}{\partial w_{ji}} \right) + \tau_{\text{syn}} \left(\lambda_I^- \cdot \frac{\partial I^-}{\partial w_{ji}} - \lambda_I^+ \cdot \frac{\partial I^+}{\partial w_{ji}} \right) \\ & + \frac{\partial l_p}{\partial t_k^{\text{pre}}} \frac{\mathrm{d}t_k^{\text{pre}}}{\mathrm{d}w_{ji}} + l_{V,k}^{-} \frac{\mathrm{d}t_k^{\text{pre}}}{\mathrm{d}w_{ji}} - l_{V,k}^{+} \frac{\mathrm{d}t_k^{\text{pre}}}{\mathrm{d}w_{ji}} \Bigg] \Bigg|_{t_k^{\text{pre}}} \end{aligned} \tag{4.18}$$

$$\psi_k = \left[\tau_{\text{mem}} \left(\lambda_V^- \cdot \frac{\partial V^-}{\partial w_{ji}} - \lambda_V^+ \cdot \frac{\partial V^+}{\partial w_{ji}} \right) + \tau_{\text{syn}} \left(\lambda_I^- \cdot \frac{\partial I^-}{\partial w_{ji}} - \lambda_I^+ \cdot \frac{\partial I^+}{\partial w_{ji}} \right) \right] \Bigg|_{t_k^{\text{post}}}. \tag{4.19}$$

It is to be noted that ψ_k does not contain terms that depend on l_V because these terms cancel out. This is due to the fact that at $t = t^{\text{post}}$ the membrane potential V does not experience a jump and thus $l_V^- = l_V^+$ holds.

The only difference with respect to the derivation in [37] is that instead of having only the term ξ_k at the presynaptic spike times, now there is also the term ψ_k at the postsynaptic spike times.

Now, the relations of the adjoint variables before and after the transitions will be derived. This is done by first deriving the relationships between the partial derivatives of the state variables with respect to the weights before and after the transition. In contrast to the original derivation, we now have to find these relationships at both the presynaptic as well as the postsynaptic spike times. For each pair of transitions we consider a spike which is caused by the nth neuron where all other neurons $m \neq n$ do not spike.

For partial derivative jumps to occur in $\frac{\partial V}{\partial w_{ji}}$ for a specific neuron at a discrete transition, either the neuron's potential or synaptic current have to experience a jump because $\dot{V}$ depends on I. For $\frac{\partial I}{\partial w_{ji}}$, there is only a jump if the neuron's synaptic current jumps.

Transition at presynaptic spike time

Membrane potential transition. At the presynaptic spike time, the only state transition that happens is the reset of the membrane potential from the threshold ϑ to the reset potential V_{reset} of neuron n. So for the membrane potential before the transition we get:

$$(V^-)_n - \vartheta = 0. \tag{4.20}$$

By differentiating this, it follows that

$$\left(\frac{\partial V^-}{\partial w_{ji}}\right)_n + (\dot{V}^-)_n \frac{\mathrm{d}t^{\text{post}}}{\mathrm{d}w_{ji}} = 0. \tag{4.21}$$

For $(\dot{V}^-)_n \neq 0$ (excluding the edge case where neuron reaches the threshold as the maximum of its potential) we have

$$\frac{\mathrm{d}t^{\text{post}}}{\mathrm{d}w_{ji}} = -\frac{1}{(\dot{V}^-)_n}\left(\frac{\partial V^-}{\partial w_{ji}}\right)_n. \tag{4.22}$$

After the transition, the spiking neuron's membrane potential is reset, so we get

$$(V^+)_n = 0. \tag{4.23}$$

By differentiation we have

$$\left(\frac{\partial V^+}{\partial w_{ji}}\right)_n + (\dot{V}^+)_n \frac{\mathrm{d}t^{\text{post}}}{\mathrm{d}w_{ji}} = 0. \tag{4.24}$$

Combining this with (4.22) yields the following condition for the jump:

$$\left(\frac{\partial V^+}{\partial w_{ji}}\right)_n = \frac{(\dot{V}^+)_n}{(\dot{V}^-)_n}\left(\frac{\partial V^-}{\partial w_{ji}}\right)_n. \tag{4.25}$$

Because both the membrane potential and the synaptic current of the non-spiking neurons do not experience any jumps, we have

$$\left(\frac{\partial V^+}{\partial w_{ji}}\right)_m = \left(\frac{\partial V^-}{\partial w_{ji}}\right)_m. \tag{4.26}$$

Synaptic current transition. Since at the presynaptic spike time, all synaptic currents do not experience jumps, the derivatives of the current for all neurons do not experience jumps

$$\frac{\partial I^+}{\partial w_{ji}} = \frac{\partial I^-}{\partial w_{ji}}. \tag{4.27}$$

Transition at postsynaptic spike times

Membrane potential transition. At the postsynaptic spike time, the spiking neuron's membrane potential and synaptic current do not experience any jumps and therefore neither do their respective gradients. So we have

$$\left(\frac{\partial V^+}{\partial w_{ji}}\right)_n = \left(\frac{\partial V^-}{\partial w_{ji}}\right)_n. \tag{4.28}$$

For non-spiking neurons we know that $(V^+)_m = (V^-)_m$ and thus by differentiating both sides with respect to the weights we get

$$\left(\frac{\partial V^+}{\partial w_{ji}}\right)_m + (\dot{V}^+)_m \frac{\mathrm{d}t^{\text{post}}}{\mathrm{d}w_{ji}} = \left(\frac{\partial V^-}{\partial w_{ji}}\right)_m + (\dot{V}^-)_m \frac{\mathrm{d}t^{\text{post}}}{\mathrm{d}w_{ji}}. \tag{4.29}$$

Because we know that the membrane potential V of the non-spiking neurons does not change at the postsynaptic spike time, but the synaptic current experiences a jump of w_{mn}, we immediately get from the differential equation of V (4.3) that

$$\tau_{\text{mem}}(\dot{V}^+)_m = \tau_{\text{mem}}(\dot{V}^-)_m + w_{mn} \tag{4.30}$$

and thus we then get with eq. (4.29) and eq. (4.22)

$$\begin{aligned}\left(\frac{\partial V^+}{\partial w_{ji}}\right)_m &= \left(\frac{\partial V^-}{\partial w_{ji}}\right)_m - \tau_{\text{mem}}^{-1} w_{mn} \frac{\mathrm{d}t^{\text{post}}}{\mathrm{d}_{ji}} \\ &= \left(\frac{\partial V^-}{\partial w_{ji}}\right)_m + \left[\frac{1}{\tau_{\text{mem}}(\dot{V}^-)_n} w_{mn} \left(\frac{\partial V^-}{\partial w_{ji}}\right)_n\right]\Bigg|_{t_k^{\text{pre}}}.\end{aligned} \tag{4.31}$$

Note that this partial derivative jump of neuron m at the postsynaptic spike time includes a term with a partial derivative of the spiking neuron n at the presynaptic spike time. This will be important when the terms are reordered again later.

Synaptic current transition. As argued above, the gradients of the spiking neurons do not experience any jumps:

$$\left(\frac{\partial I^+}{\partial w_{ji}}\right)_n = \left(\frac{\partial I^-}{\partial w_{ji}}\right)_n. \tag{4.32}$$

For the non-spiking neurons, the synaptic current jumps by the corresponding weight w_{mn}

$$(I^+)_m = (I^-)_m + w_{mn}. \tag{4.33}$$

Differentiation of this equation yields

$$\left(\frac{\partial I^+}{\partial w_{ji}}\right)_m + (\dot{I}^+)_m \frac{\mathrm{d}t^{\text{post}}}{\mathrm{d}w_{ji}} = \left(\frac{\partial I^-}{\partial w_{ji}}\right)_m + (\dot{I}^-)_m \frac{\mathrm{d}t^{\text{post}}}{\mathrm{d}w_{ji}} + \delta_{in}\delta_{jm}. \tag{4.34}$$

Moreover, from the differential equation of the current (4.4) we get

$$\tau_{\text{syn}}(\dot{I}^+)_m = \tau_{\text{syn}}(\dot{I}^-)_m - w_{mn} \tag{4.35}$$

Substituting this into eq. (4.34) and also using (4.22), we get the following expression for the jump of the partial derivative:

$$\begin{aligned}\left(\frac{\partial I^+}{\partial w_{ji}}\right)_m &= \left(\frac{\partial I^-}{\partial w_{ji}}\right)_m + \tau_{\text{syn}}^{-1} w_{mn} \frac{\mathrm{d}t^{\text{post}}}{\mathrm{d}w_{ji}} + \delta_{in}\delta_{jm} \\ &= \left(\frac{\partial I^-}{\partial w_{ji}}\right)_m - \left[\frac{1}{\tau_{\text{syn}}(\dot{V}^-)_n} w_{mn} \left(\frac{\partial V^-}{\partial w_{ji}}\right)_n\right]\Bigg|_{t_k^{\text{pre}}} + \delta_{in}\delta_{jm}.\end{aligned} \tag{4.36}$$

Similar to eq. (4.31), this equation also includes a term with the partial derivative of the spiking neuron n at t^{pre}. Now, we need to group all terms by their partial derivatives at the same time because this allows to define jump conditions, which lead to all partial derivatives vanishing from the sum. Thus, the two aforementioned terms from ψ_k (from eq. (4.31) and eq. (4.36)) need to be grouped with the other terms at the presynaptic spike time. By substituting all the partial derivative jumps at the presynaptic spike times into ξ_k and also including these two special terms with their preliminary factors from ψ_k, we get the following expression:

$$\begin{aligned}\xi_k' = \Bigg[&\sum_{m\neq k}\left[\tau_{\text{mem}}(\lambda_V^- - \lambda_V^+)_m \left(\frac{\partial V^-}{\partial w_{ji}}\right) + \tau_{\text{syn}}(\lambda_I^- - \lambda_I^+)_m \left(\frac{\partial I^-}{\partial w_{ji}}\right)\right] \\ &+ \left(\frac{\partial V^-}{\partial w_{ji}}\right)_{n(k)} \Bigg[\tau_{\text{mem}} \left(\lambda_V^- - \frac{(\dot{V}^+)_{n(k)}}{(\dot{V}^-)_{n(k)}}\lambda_V^+\right)_{n(k)} \\ &+ \frac{1}{(\dot{V}^-)_{n(k)}} \left(\sum_{m\neq k}\left[w_{mn(k)}(\lambda_I^+ - \lambda_V^+)_m + l_V^- - l_V^+\right]\Big|_{t_k^{\text{post}}} - \frac{\partial l_p}{\partial t_k^{\text{post}}} + l_V^+ - l_V^-\right)\Bigg] \\ &+ \tau_{\text{syn}}(\lambda_I^- - \lambda_I^+)\left(\frac{\partial I^-}{\partial w_{ji}}\right)_{n(k)}\Bigg]\Bigg|_{t_k^{\text{pre}}}.\end{aligned} \tag{4.37}$$

The scalar products are written as sums of the components where terms are grouped for the spiking neuron n and the non-spiking neurons $m \neq n$. In order to reduce this whole term to 0 and therefore not have any partial derivatives left, we get that all adjoint variables except $(\lambda_V)_n$ do not experience jumps at $t = t_k^{\text{pre}}$

$$(\lambda_V^-)_m = (\lambda_V^+)_m \tag{4.38}$$

$$\lambda_V^- = \lambda_I^+ \tag{4.39}$$

Now, we only need to find the jump condition for $(\lambda_V)_n$ for which ξ' resolves to zero. Rearranging of terms yields

$$\begin{aligned}(\lambda_V^-)_n =& \frac{(\dot{V}^+)_n}{(\dot{V}^+)_n}(\lambda_V^+)_n + \frac{1}{\tau_{\text{mem}}(\dot{V}^-)_n}\left[\sum_{m\neq k}\left[w_{mn(k)}(\lambda_V^+ - \lambda_I^+)_m\right]\Big|_{t_k^{\text{post}}}\right. \\ &\left. + \frac{\partial l_p}{\partial t_k^{\text{post}}} - l_V^+ + l_V^-\right].\end{aligned} \tag{4.40}$$

Now we will look at the second term at the time of the postsynaptic spike. It is obtained by substituting the previously derived jumping conditions of the partial derivatives at $t = t_k^{\text{post}}$ into eq. (4.19) where the two terms that already went into ξ' are omitted. We get

$$\begin{aligned}\psi_k' =& \left[\sum_{m\neq k}\left[\tau_{\text{mem}}(\lambda_V^- - \lambda_V^+)_m\left(\frac{\partial V^-}{\partial w_{ji}}\right) + \tau_{\text{syn}}(\lambda_I^- - \lambda_I^+)_m\left(\frac{\partial I^-}{\partial w_{ji}}\right) - \tau_{\text{syn}}\delta_{in(k)}\delta_{jm}(\lambda_I^+)_m\right]\right. \\ &\left. + \tau_{\text{mem}}(\lambda_I^- - \lambda_I^+)\left(\frac{\partial V^-}{\partial w_{ji}}\right)_{n(k)} + \tau_{\text{syn}}(\lambda_I^- - \lambda_I^+)\left(\frac{\partial I^-}{\partial w_{ji}}\right)_{n(k)}\right].\end{aligned} \tag{4.41}$$

By choosing all adjoint variables to not experience jumps at the postsynaptic spike times

$$\lambda_V^- = \lambda_V^+ \tag{4.42}$$

$$\lambda_I^- = \lambda_I^+ \tag{4.43}$$

the whole term reduces to

$$\psi_k' = -\tau_{\text{syn}}\sum_{m\neq n(k)}\delta_{in(k)}\delta_{jm}(\lambda_I^+)_m\Big|_{t^{\text{post}}} = -\tau_{\text{syn}}\sum_{m\neq n(k)}\delta_{in(k)}(\lambda_I^+)_j\Big|_{t^{\text{post}}}. \tag{4.44}$$

The remaining Kronecker delta is only non-zero if the presynaptic neuron is neuron i. So, overall we can write the gradients as

$$\frac{\mathrm{d}\mathcal{L}}{\mathrm{d}w_{ji}} = \sum_{k=0}^{N}\xi_k' + \psi_k' = -\tau_{\text{syn}}\sum_{\text{spikes from i}}(\lambda_I)_j\Big|_{t^{\text{post}}} \tag{4.45}$$

If we now instead look at synaptic delays where the $d_k = d_{mn(k)}$ vary for each m, we would need to split up the integral at all $N-1$ postsynaptic spike times for each spike k. This would just lead to the result from eq. (4.17), but with instead of just the term ψ_k, we would have to sum over all $N-1$ postsynaptic spikes and evaluate them at the corresponding postsynaptic spike time

$$\frac{d\mathcal{L}}{dw_{ji}} = \sum_{k=1}^{N_{\text{pre}}} \left[\xi_k + \sum_{m=1}^{N-1} \psi_{km} \right], \tag{4.46}$$

where ψ_{km} is the corresponding term for the postsynaptic neuron m given by

$$\psi_{km} = \left[\tau_{\text{mem}} \left(\lambda_V^- \cdot \frac{\partial V^-}{\partial w_{ji}} - \lambda_V^+ \cdot \frac{\partial V^+}{\partial w_{ji}} \right) + \tau_{\text{syn}} \left(\lambda_I^- \cdot \frac{\partial I^-}{\partial w_{ji}} - \lambda_I^+ \cdot \frac{\partial I^+}{\partial w_{ji}} \right) \right] \Big|_{t_k^{\text{pre}} + d_{mn(k)}} . \tag{4.47}$$

However, it is straightforward to argue that the arguments made in section 4.1 for the jumps at the transition times can be repeated for all terms with a few adjustments. The additional terms from the partial derivative jumps at postsynaptic spike times in eq. (4.31) and eq. (4.34) need to be adapted. These go into the jump condition and their evaluation just has to be changed to the correct postsynaptic spike time of the corresponding postsynaptic neuron m. For the transition of λ_V we then get

$$\begin{aligned} (\lambda_V^-)_n = & \frac{(\dot{V}^+)_n}{(\dot{V}^+)_n} (\lambda_V^+)_n + \frac{1}{\tau_{\text{mem}} (\dot{V}^-)_n} \left[\sum_{m \neq k} \left[w_{n(k)m} (\lambda_V^+ - \lambda_I^+)_m \right] \Big|_{t_k^{\text{pre}} + d_{mn(k)}} \right. \\ & \left. + \frac{\partial l_p}{\partial t_k^{\text{pre}}} - l_V^+ + l_V^- \right], \end{aligned} \tag{4.48}$$

where the postsynaptic spike time $t_k^{\text{pre}} + d_{mn(k)}$ varies for each k and m. For the Kronecker delta term in eq. (4.34), we again have a $\delta_{in}\delta_{jm}$. Therefore for all ψ_{km} where $m \neq j$, there is no additional term and we get the Kronecker delta term only in ψ_{kj}. This then results in the exact same expression for the gradients:

$$\frac{\mathrm{d}\mathcal{L}}{\mathrm{d}w_{ji}} = \sum_{k=0}^{N} \xi_k' + \psi_{km}' = -\tau_{\text{syn}} \sum_{\text{spikes from i}} (\lambda_I)_j \Big|_{t^{\text{post}}} . \tag{4.49}$$

This means that in this adapted version of EventProp with delays, the adjoint variables have to be sampled at each postsynaptic spike time, which differs among all neurons. At this time, also the term $w_{n(k)m}(\lambda_V^+ - \lambda_I^+)_m$ needs to be calculated, so that in can then be used when the internal transition of the corresponding spike k happens at time t_k^{pre}.

4.2 Gradients with respect to the synaptic delays

Since the derivation is very similar to the previous derivation of the gradients with respect to the weights, the more detailed derivation can be found in appendix A. However, the differences between the derivations will shortly be discussed and the result will be stated in this section.

In contrast to the derivation for the weights, in the derivation for the delays, the Leibniz rule yields additional terms because two integral limits contain the delay and must be differentiated. However, these terms cancel each other out because they reduce to $l_V^- - l_V^+$ at the time of the postsynaptic spike, which is zero, as already discussed.

In the derivation of the partial derivative jumps, the crucial difference is that for the derivative with respect to the delay we have

$$\frac{\mathrm{d}t^{\mathrm{post}}}{\mathrm{d}d_{ji}} = \frac{\mathrm{d}(t^{\mathrm{pre}} + d_{mn})}{\mathrm{d}d_{ji}} = \frac{\mathrm{d}t^{\mathrm{pre}}}{\mathrm{d}d_{ji}} + \delta_{jm}\delta_{in} \tag{4.50}$$

and that the derivative of the weights with respect to the delays is zero:

$$\frac{\mathrm{d}w_{mn}}{\mathrm{d}d_{ji}} = 0. \tag{4.51}$$

Overall, this leads to the exact same adjoint equations and jump conditions as in the derivation in section 4.1. However, because of eq. (4.50) and eq. (4.51), the terms that are left after substituting the partial derivative jumps change with respect to the previous derivation. The gradient with respect to the synaptic delays is then given by:

$$\frac{\mathrm{d}\mathcal{L}}{\mathrm{d}d_{ji}} = \sum_{\text{spikes from i}} w_{ji}(\lambda_V - \lambda_I)_j \Big|_{t^{\mathrm{post}}}. \tag{4.52}$$

4.3 Additional Remarks

As also stated in [37], with the differential equation of V (4.3), it holds that $(\dot{V}^+)_n - (\dot{V}^-)_n = \tau_{\mathrm{mem}}^{-1}\vartheta$. Thus, the following relation

$$\frac{(\dot{V}^+)_n}{(\dot{V}^-)_n} = \frac{(\dot{V}^+)_n - (\dot{V}^-)_n}{(\dot{V}^-)_n} + 1 = \frac{\vartheta}{\tau_{\mathrm{mem}}(\dot{V}^-)_n} + 1 \tag{4.53}$$

can be used to rewrite the jump of the adjoint variable as eq. (4.54). All other equations of the method are additionally listed in table 4.1.

$$\begin{aligned}(\lambda_V^-)_{n(k)} =& (\lambda_V^+)_{n(k)} + \frac{1}{\tau_{\mathrm{mem}}(\dot{V}^-)_{n(k)}} \Bigg[\vartheta(\lambda_V^+)_{n(k)} \\ &+ \sum_{m \neq k} \left[w_{mn(k)}(\lambda_V^+ - \lambda_I^+)_m \right] \Big|_{t_k^{\mathrm{pre}} + d_{mn(k)}} + \frac{\partial l_p}{\partial t_k^{\mathrm{pre}}} + l_V^- - l_V^+ \Bigg]\end{aligned} \tag{4.54}$$

These equations can now be compared with the original equations, which are stated in section 2.5. You can see that they only differ very slightly. By introducing the synaptic delay,

Free dynamics	**Transition condition**	**Gradients**
$\tau_{\text{mem}}\lambda_V' = -\lambda_V - \frac{\partial l_V}{\partial V}$	$t - (t_k^{\text{pre}} + d_{mn(k)}) = 0$	$\frac{\mathrm{d}\mathcal{L}}{\mathrm{d}w_{ji}} = -\tau_{\text{syn}} \sum_{\text{spikes from i}} (\lambda_I)_j \Big\vert_{t^{\text{post}}}$
$\tau_{\text{syn}}\lambda_I' = -\lambda_I + \lambda_V$	for any k, m	$\frac{\mathrm{d}\mathcal{L}}{\mathrm{d}d_{ji}} = \sum_{\text{spikes from i}} w_{ji}(\lambda_V - \lambda_I)_j \Big\vert_{t^{\text{post}}}$

Table 4.1: EventProp with synaptic delays together with jump of λ_V, see eq. (4.54).

the sampling of the adjoint variable λ_I for the weight gradients is moved to the postsynaptic spike times when the spike arrives at the postsynaptic neuron. Moreover, the jump of λ_V is adjusted such that the adjoint variables λ_V and λ_I are evaluated at t^{post}. So in the limit where all synaptic delays approach zero, the proposed model reverts back to the original one which shows that they are consistent with each other.

Moreover, the fact that the gradients of the delays and the weights are sampled at the same time, makes the extension to delay learning very straightforward because once EventProp with constant synaptic delays is implemented, the updating of the delay gradient only has to be added at the point where the weight gradients are updated. Thus, the complexity of the computation does not increase much when adding gradients with respect to delays.

5 Implementation

5.1 Arbitrary Topologies

In this section, it will be described how the event-based part of the python library jaxsnn was extended to arbitrary topologies. Arbitrary topologies refer to a multi-layer network, where layers of any size can be connected via feed-forward and feedback connections. In particular, layers can also be recurrently connected to themselves.

5.1.1 Topology Description

First, we need to find a description for arbitrary topologies in jaxsnn. JAX has a functional programming paradigm where the computation happens by passing inputs into pure functions and the outputs are then used by other functions. This stands in contrast to other programming paradigms such as object-oriented programming where classes are used to handle internal state. For example, JAX does not allow transformations on methods that act on attributes of a class, because it can only track the computations that happen on the input that was passed in. This is why one solution for this is to create an `init/apply` pair for the network, as it was described in section 3.2.

We will now define an object-oriented network description that allows flexible composing of the networks, but in the end also returns a single `init/apply` pair which can be transformed with JAX' transformations. For that, we define a `NeuronLayer` and an `InputLayer` class. Objects of these classes contain all the information about the layer like its size or the layers that have been added as input layers. Moreover, a method `add_input_layer` is implemented which accepts layer objects and adds them to the layer's inputs. This allows to establish the connections of the network.

Then in the end, a `get_init_apply` function can be called on the output layer to create the `init/apply` pair because all information for the layers and their connections is available and can be reached by going from the output layer recurrently down to all other layers. An example of how a small recurrently connected network can be build up this way is shown in fig. 5.1.

5.1.2 Algorithm

The previous implementation of feed-forward topologies made use of the fact that layers can be simulated sequentially because there are no feedback connections. For a time step based solution it would be possible to just add all contributions from the input layers at each time

```python
# Define layers/vertices
i1 = InputLayer()
l1 = NeuronLayer(layer_params2)
l2 = NeuronLayer(layer_params3)

# Define connections/edges
l2.add_input_layer(l1)
l2.add_input_layer(i1)
l1.add_input_layer(i1)
l1.add_input_layer(l2)

init, apply = l2.get_init_apply()
```

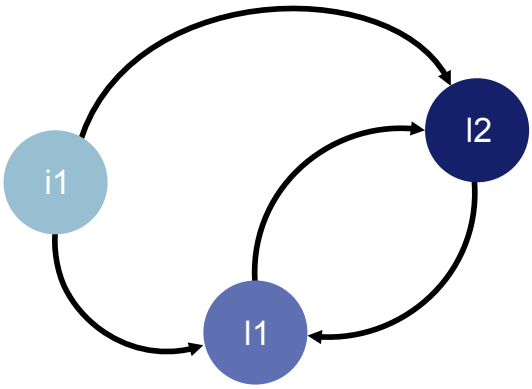


Figure 5.1: Code (left) that is used to define a recurrently connected network (right) with one input layer (light blue), one hidden layer (blue) and one output layer (dark blue).

step. This comes from the fact that the whole system is evolved in a synchronized fashion. However, for an event-based simulation, the layers only jump to the points in time where events happen and thus we lose simultaneity. For example, we can consider two layers l_1, l_2 which receive both input from each other, as show in fig. 5.2. It would never be possible to simulate this sequentially because they depend on the output of each other.

Moreover, if you alternate between the layers to calculate events, also certain problems arise. Let's say that those two connected layers are at times t_1 and t_2 where $t_1 < t_2$. If we want to do a simulation step for l_1 we can calculate when the next event will happen, which can either be an internal event or an input spike.

If the time of this next event t_e is earlier than t_2, no problems arise. However, if the calculated event is later then t_2, layer 1 cannot take this as its next event because there is a time window between t_2 and t_e where a spike could happen in the second layer. This spike would then arrive earlier than t_e in the first layer and the calculated event at t_e would be invalid. Thus l_1 cannot be evolved in time to the event with certainty and can only be evolved to t_2. If we look at l_2, it is obvious that it cannot be evolved to its next event either because there is no certainty if l_1 emits a spike earlier. Thus, if no special care is taken to compare the spikes which conflict with each other, the system is stuck. And this even becomes a much bigger problem if there are 3 or more recurrently connected layers. Theoretically, backtracking solutions could be implemented where spikes are calculated ‚greedily' and then the state is backtracked when there are conflicting spikes.

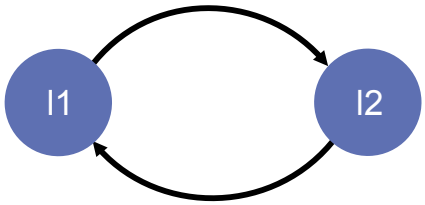


Figure 5.2: Two recurrently connected layers.

Another solution to this problem would be to introduce a delay between these layers. That

way, layer 1 can calculate until $t_2 + d$ and in case it does not find an event before that time, it can evolve to $t_2 + d$. Layer 2 can then safely calculate until the time $t_1 + d = t_2 + 2d$ and thus it is guaranteed that the system moves forward in time. It is to be noted, that time-grided approach actually also kind of make use of this method, where the minimal delays is just the time step.

The proposed algorithm for one step of a layer is listed in algorithm 1, adapted from [27]. The simulation happens for M iterations. At each iteration a step for each of the K layers is performed. First, the next internal event is calculated which corresponds to the neuron in the layer that spikes next. Then, the next input spike is determined by taking the minimum of the potential input spikes from all input layers. The next event is chosen as the earlier event of next internal and input spike. Then the time until which the current layer can safely evolve is calculated. This is done by adding the delay to the times of the input layers, and then taking the minimum. This makes sure that no input event can arrive before this t_{safe}. Then it is checked if the found event at time t is earlier than the safe time t_{safe}. If not, a dummy spike is returned. If the spike is allowed, the layer can be evolved to t and the discrete state transitions, depending on the type of event, can be applied. Then the spike is returned. It is to note, that the proposed algorithm is directly applicable to axonal delays where all neurons in a layer receive the same input times for the input spikes and thus can be evolved synchronously. It could be also extended to synaptic delays but for that each neuron of a layer has to be handled in parallel and the next internal event and safe time would have to be found for each neuron individually.

Algorithm 1 Event-based simulation for recurrently connected layers.

for $i \leftarrow 1$ to M **do**
  for $j \leftarrow 1$ to K **do**
    $t_{\text{internal}} \leftarrow$ find time of next internal event for N neurons
    $t_{\text{ix}} \leftarrow \min(t_{\text{internal}})$
    $t_{\text{external}} \leftarrow$ find time of next input for K input layers
    $t_{\text{ex}} \leftarrow \min(t_{\text{external}} + d)$
    $t \leftarrow \min(t_{\text{ex}}, t_{\text{ix}})$
    $t_{\text{layers}} \leftarrow$ find time for K input layers
    $t_{\text{safe}} \leftarrow \min(t_{\text{external}} + d)$
    if $t > t_{\text{safe}}$ **then**
      append **dummy spike** to spike list
    end if
    $V, I \leftarrow$ integrate neuron state
    $V, I \leftarrow$ apply discrete state transition
    append **spike** to spike list
    return spike list
  end for
end for

5.1.3 Implementation

Now, the implementation of algorithm 1 will be discussed. To fit into the description framework, which was introduced in section 5.1.1, we need to define a `get_init_apply` function.

This `get_init_apply` function starts from the output layer and collects all the layers in the network by recursively exploring all input connections. Thus, first the output layer is added to a list, then the output layer's input layers, then the input layers of the input layers of the output layer and so on. The list is then reversed so that the lists are in a more natural order, with the output layer being the last layer. All layers then have their own ́flatĺayer index which corresponds to their position in the list.

Now the step functions for each individual layer have to be constructed. For this, a `construct_step_fn` function is implemented for the NeuronLayer class which takes the input layers as an input parameter. The constructed step function needs to follow, what was proposed in algorithm 1. So first, the next internal event is found by using the already implemented `ttfs_solver` function. As, in the previous implementation, the `ttfs_solver` has to be vectorized with `jax.vmap` to calculate the next internal time for all N neurons of the layer. Then the earliest of these events is chosen. Next, a new function `next_input` is implemented, which compares the next input spikes from all input layers and returns the earliest. To do this, it finds the earliest internal spike in the spike recordings of the input layers. To make this check for internal spikes easy, the `EventPropSpike` class is extended to the following:

```
@dataclasses.dataclass
@tree_math.struct
class EventPropSpike:
    time: jax.Array # float
    idx: jax.Array # int
    current: jax.Array # float
    internal: jax.Array # bool
```

Listing 3: Extension of the `EventPropClass` to include information if the recorded spike was internal or external

Also the corresponding delay is added to the input time. The next event is then chosen as the minimum of the next internal and next input event. Also, a `min_delay_check` function is implemented that calculates the safe time from the times of the input layer states and checks if the time of the next event is earlier than this t_{safe}. Depending on, if the next event is allowed, the layer is evolved to the time of the next event or the safe time, using the already implemented `lif_exponential_flow`. Then, in case the event is allowed, the discrete transition is applied for which a new `transition` function was implemented. This `transition` function has to be

created for each input layer and is then chosen inside the step function via `jax.lax.switch`. This has to be done, because JAX requires static shapes and if the same transition function is used for different input layers, different shaped arrays of the weights would appear due to different layer sizes.

By constructing these `step` functions for all layers in the call of `get_init_apply`, the loops defined in algorithm 1 can be implemented. For the loop, `jax.lax.scan` is used and after each call of a `step` function, the spikes are updated by the result from the step function. The spikes are stored as a list of `EventPropSpike`s and they are passed into all layers, so that they can be used to find the input events.

5.1.4 EventProp

Moreover, the EventProp algorithm [37] was also extended to work with this algorithm. The extension of Eventprop to delays is done in chapter 4. For the implementation of the EventProp algorithm, the backpropagation cannot happen layer wise like in the original implementation because now also feedback connections are allowed. This is because the jumping condition eq. (4.54) contains the term

$$\left(W^\top(\lambda_V^+ - \lambda_I)\right)_{n(k)}\Big|_{t_k^{\text{post}}} \tag{5.1}$$

This term corresponds to the error propagation from layers that take input from the current layer. For the feed-forward case, one can start with the output layer and go back through the layers without running into problems because there the output layers of a given layer are available. However, if you allow arbitrary topologies this can no longer be guaranteed.

Thus the implementation is done in a way that we start with the last spike in absolute terms and always have to chose the next last spike from the recorded spikes from all layers. For this a new `step_bwd` function is implemented where first the next event is chosen. Then all states are evolved to the time of this event. Then the adjoint transition is applied. If it is an input spike, the adjoint variable λ_I is sampled for the calculation of the gradient and also eq. (5.1) is calculated and saved in the data structure of the adjoint spike. To every input transition there is a corresponding internal transition. For internal transitions, the jump is calculated according to eq. (4.54) where for the term in eq. (5.1) a function `forward_term` is implemented that checks if the internal spike happened in other layers as an input spike and then adds the term that was saved in the adjoint spike. This allows to calculate the gradients with EventProp which was also validated by comparing the analytical gradients and the EventProp gradients with each other.

5.2 Synaptic Delays

Now, synaptic delays are also implemented into jaxsnn. This is done for feed-forward networks because this way layers can still be evolved in a vectorized manner.

For the old implementation without synaptic delays, the output spikes were just taken as the input queue of the next layer because there they were already ordered. Now the order at the postsynaptic neurons is not guaranteed anymore because if one synaptic delay is very long and another one is short, the order can easily change. So before the `step` function can be used to step through the events, the delays have to be added to the presynaptic spike times and then ordered to get a correct input queue. Moreover, in contrast to the original implementation, neurons are evolved asynchronously in their layer. For that, a new `step` function was implemented, which is batched over the axis of the neurons in the layer with `jax.vmap`.

5.2.1 EventProp

In contrast to the implementation for arbitrary topologies, for the feed-forward case with synaptic delays the layers can be handled sequentially, because the jumping condition in eq. (4.54) only depends on forward connected layers which here have been already handled if we start with the last layer. Moreover, is a `step_bwd` function implemented which can be batched over, so that the spikes from all neurons in a layer are stepped through in a vectorized manner. For the forward term in (4.54)

$$\left(W^\top(\lambda_V^+ - \lambda_I)\right)_{n(k)}\Big|_{t_k^{\mathrm{pre}}+d_{mn(k)}}, \tag{5.2}$$

a similar implementation to the previous section is done where the term is stored in the data structure of the adjoint spike. When the input happens a `forward_term` function checks if the internal spike was an input spike and then retrieves the value for (5.2) from where it was saved in the adjoint spike.

5.3 Membrane Potential and Current Traces

The forward pass as well as the backward pass of the previously described implementations happen completely event-based. This means that only the output spikes are returned as a result. This turns the simulation with respect to the state variables in between transitions into a black box. To further examine the states between transitions and display the continuous evolving in time a new function `get_layer_traces` is implemented. This function takes the recorded spikes and the weights from a given network as inputs and then uses this information to calculate the voltage and current traces of the neurons for a specified time grid. This can be done by always evolving a given neuron in time to the next point in the time grid and saving the state variables, V and I. When a input transition happens before the next point in the time grid,

the corresponding discrete transition is applied, depending on whether the even is an input or internal spike. This way, the traces are constructed for the whole specified interval. An exemplary plot where this was used to plot the voltage trace and and synaptic input current for a single neuron with one input spike can be seen in fig. 5.3.

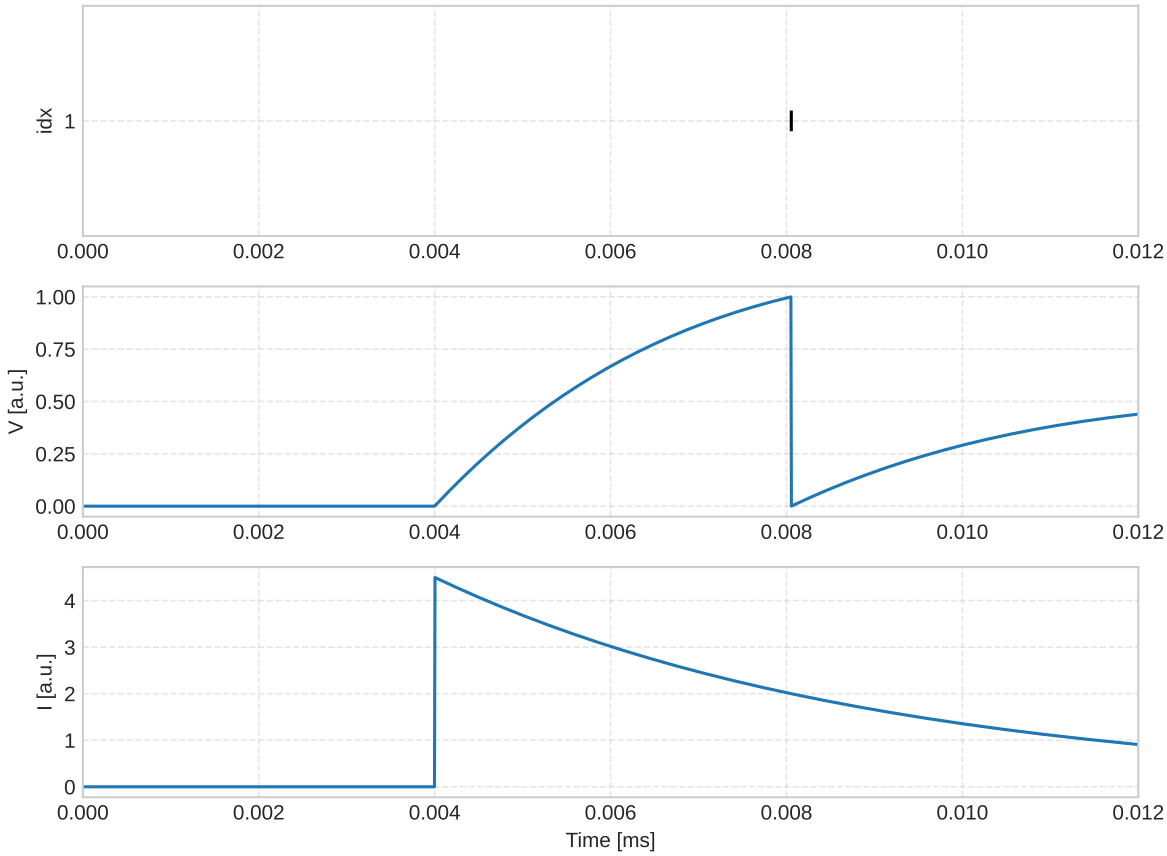


Figure 5.3: Plotting the traced values for the membrane potential and synaptic input current for the simple example of a single neuron and one input spike. The implemented `get_layer_traces` function can be used to calculate the traces from the recorded event-based spike data of a network.

6 Experiments

6.1 Arbitrary Topologies

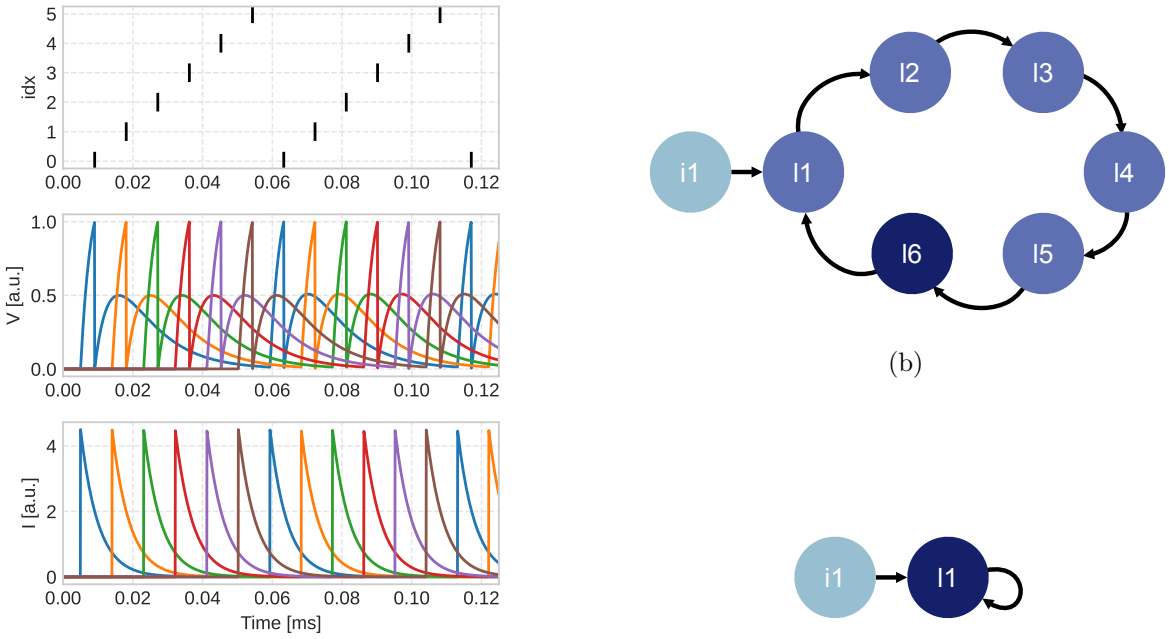
6.1.1 Simple Example

First, two simple examples are constructed to validate the creation of arbitrary topologies. The first example consists of six neurons which are connected in a chain. One of the neurons receives a single input spike. All weights are chosen such that an input spike increases the synaptic input current enough to cause a spike in the corresponding postsynaptic neuron. Furthermore, a single layer of six neurons is used as a consistency check. This layer is recurrently connected to itself. The weights are set such that the same topology as for the multilayer example is created. So all weights w_{ji} except where $j = i + 1$ holds are set to zero to establish the forward connection. Additionally, the recurrent connection between the last and first neuron is accounted for by setting w_{05}. The spikes that are recorded for each layer after passing in the input are, as expected, in both cases identical. The spikes as well as the membrane potential and current traces that were computed with `get_layer_traces` are shown in fig. 6.1.

6.1.2 Learning in arbitrary topologies

Moreover, a simple example is implemented that showcases that learning with the implemented topologies works with both analytical gradients and EventProp. This is done by constructing a simple network with an input layer connected to a layer with a single neuron. This layer is then recurrently connected with the output layer, as displayed in fig. 6.2. We will consider a single input spike. The initial weights are set such that the input spike causes a single spike in the first neuron which then causes a single spike in the output neuron, so that without recurrence nothing would happen after that. But because of the recurrent connection between the two neurons, the feedback connection causes the two neurons to fire again. Due to the way the implementation was done, also a short delay has to be included. We will now learn the time of the second spike of the neuron from the second layer l_2. Because at least initially, the system has to go through the recurrent connection, we can check if the implemented backpropagation works correctly for recurrent connections. We will consider the squared error loss of the second spike time of the second neuron t_2 and a predefined target time t_t:

$$\mathcal{L} = (t_2 - t_t)^2 \tag{6.1}$$



(a) (b) (c)

Figure 6.1: (a) Spikes, membrane traces, and current traces calculated with `get_layer_traces` for the recurrently connected network in (b), where all layers have size 1. The different colors in the plot correspond to the different neurons. The self recurrent network with only one self recurrent layer (c) (weight matrix chosen to construct the same topology), results in the exact same spikes and traces as it is expected. Output layers that are used to construct the layers' `init/apply` pair via the `get_init_apply` method are again displayed in dark blue, hidden layers in blue and input layers in light blue.

The learning is done with both analytical gradients and the implemented version of the EventProp algorithm. Here, the implementation described in section 5.1.4 already uses the derived theory from section 4.1 for EventProp with constant delays.

The gradients which are calculated with the analytical gradients and EventProp are compared and they are exactly the same. So this is already a verification of the derived equations. To calculate the gradient to update the weights, 100 steps are performed, where the `Adam` optimizer is used. The loss over these time steps is displayed in fig. 6.3. A very striking feature are sharp jumps which seem to increase the loss by a lot temporarily. These display a critical property of the EventProp algorithm. Because in the definition of our loss we state that a certain number of spikes is going to occur, spikes that are added or deleted, are not taken into account. This

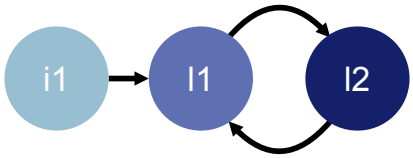


Figure 6.2: Simple learning example for recurrently connected network.

means that for the actual loss function which does not have these constraints, the calculated gradient is no longer exact.

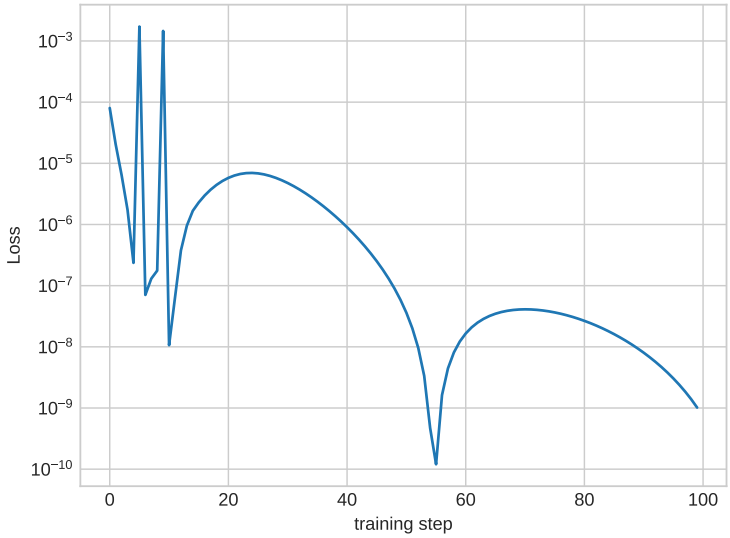


Figure 6.3: Loss curve for training of the second spike time of the output neuron from the network that was defined before. Jumps are observed which normally should not occur. However, in the EventProp algorithm this is a problem that can happen because by learning the weights via the spike times, spikes can be added or deleted which completely changes the system and which is not build into the derivation of the gradient.

6.1.3 Training on the Yin-Yang Dataset

Now, the implementation will also be tested on the Yin-Yang Dataset [19] to verify that different topologies created with the new implementation can reach the expected high accuracies. The Yin-Yang Dataset has three output classes which are displayed in fig. 6.4. Training the Yin-Yang Dataset in jaxsnn was already thoroughly investigated and discussed by Althaus [1]. For the training in [1], mainly topologies with 5 input neurons, 100 hidden neurons and 3 output neurons were used, although other layer sizes were also explored. For these topologies an accuracy of

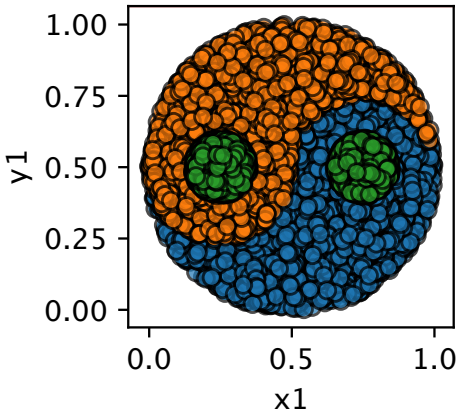
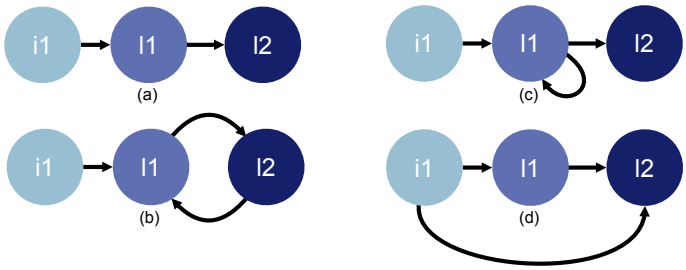


Figure 6.4: Classes ('yin', 'yang ' and 'dots') of the Yin-Yang Dataset, where each colored dot represents a sample. Adapted from [19].

about 96% is expected after training for 50 epochs. The training will follow what was done by Althaus [1]. The best results were achieved for a mean-squared error loss, where specific target times for the right and wrong classes are used for the three output neurons. So, we will build upon the original topology and loss and try some extensions and to verify that the training happens as expected. For this, the training is done with just one seed, because it is only used to verify that the backpropagation in bigger networks with custom defined topologies works as expected and that it can train to high accuracies. To achieve the highest possible accuracies the hyperparameters would have to be adjusted accordingly. But for this short comparison, the hyper parameters from Althaus [1] were used, also displayed in appendix B. The topologies and corresponding achieved accuracies are displayed in section 6.1.3.



Topology	Accuracy
(a)	96.8%
(b)	96.7%
(c)	96.1%
(d)	96.7%

Figure 6.5: Different SNN topologies were used to train on the Yin-Yang Dataset (left) for 50 epochs with one seed and their corresponding accuracies (right). This was done to verify that training works as expected for bigger topologies and high accuracies can be reached. Further hyperparameter optimization should then also be able to further increase accuracies.

6.2 Synaptic Delays

The implementation of EventProp gradients with respect to delays will now be tested using the Yin-Yang dataset. Weights and delays are trained at the same time. As a starting point, the topology, described in section 6.1, with 5 input neurons, 100 hidden neurons and 3 output neurons was used. Also the same hyperparameters were used as a starting point. And for the mean and the standard deviation of the delays, different values were tried. The used hyperparameters are listed in appendix B. Additionally, delays were truncated at zero to rule out negative delays to ensure causality and plausibility. Initially, there were difficulties to get good accuracies at all. After some fine tuning of hyperparameters, accuracies that are comparable to the training of only the weights were achieved. However, this is of course not the goal of training synaptic delays because we introduce a lot more parameters and expect the neuronal dynamics to be more adaptable. With ten different seeds a test accuracy of $(94.5 \pm 0.4)\%$ was achieved for training for 50 epochs. The test accuracies over the epochs for one seed during training are displayed in fig. 6.6.

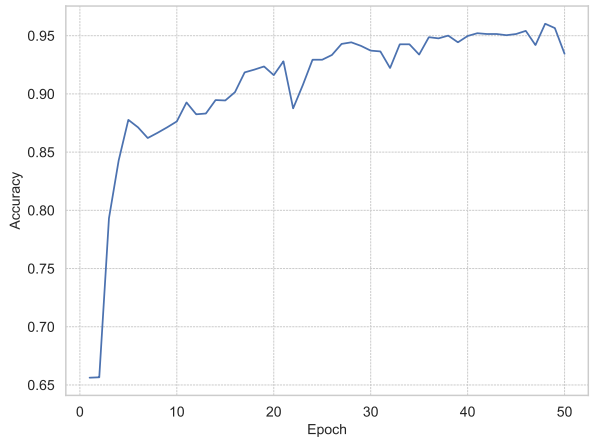


Figure 6.6: Test accuracies over the epochs for the training of synaptic delays.

The question arises what could be the reason for that inaccuracy. One possibility would be that the cutoff of the delays at zero causes the delays to be zero and in principal the optimization does not follow that constraint. If this happens for many delays, the gradients would not point in the right direction anymore. Another possibility would be that the hyperparameters are not fine tuned enough for which also not that much time was available. The train accuracy graph also looks like it experiences a temporary dip at a few points. This could be a hint for things going wrong which could be investigated at those points in more detail. For example the aforementioned deletion and adding of spikes might happen more quickly with the introduction of learnable delays which leads to more changes in the system.

It was also tested what happens if the synaptic delays are initialized randomly and not trained. For 10 different seeds a test accuracy of $(95.8 \pm 0.2)\%$ was achieved for training 50 epochs. So

this achieves a significantly higher accuracy then learning of delays, which is not what would be expected.

7 Discussion and Outlook

In this bachelor thesis, exact gradients for synaptic delays were derived and implemented. It is to note, that at the same time, this method was developed by Mészáros, Knight, and Nowotny [25] and recently published. They derive the same method and apply it to the Yin-Yang dataset [19] and the SHD dataset [8]. Moreover, this bachelor thesis implements arbitrary topologies for event-based simulations and SNN-inspired machine learning in jaxsnn. These can also be trained with the extended EventProp algorithm for constant delays which is part of the presented derivation in chapter 4. The main result of the derivation is that for delays, the evaluation of terms in the jump condition of the adjoint variable λ_V and in the sampling of the gradient with the adjoint variable λ_I happens when the presynaptic spike arrives at the postsynaptic neuron. Moreover this can then easily be extended to calculate the gradients with respect to the synaptic delays:

$$\frac{\mathrm{d}\mathcal{L}}{\mathrm{d}d_{ji}} = \sum_{\text{spikes from i}} w_{ji}(\lambda_V - \lambda_I)_j \Big|_{t^{\text{post}}}. \tag{7.1}$$

Discussion

The newly developed implementation for topologies is flexible and allows description of arbitrary networks. By connecting the layers with feedback-connections, the look-ahead time of the event-based computation can decrease a lot, which leads to more steps in the simulation where no event can be applied. However, this highly depends on the minimal delays which are used. In addition, the methods used to obtain exact gradients of spike times exhibit a few problems. While they offer sparsity and exact gradients, they also fail to incorporate the addition and deletion of spikes which can have a significant influence as was seen in section 6.1. One solution to handle this was proposed by Nowotny, Turner, and Knight [29] where they extend the EventProp formalism to a wider class of loss functions which allows them to penalize the deletion of spikes.

The accuracy results for training with synaptic delays are not satisfactory. This definitely needs to be investigated further. A few ideas already mentioned in section 6.2 will be elaborated further now. There are a few different plausible explanations. On one hand, more time is needed for optimizing hyperparameters by performing for example performing a grid search. One quantity of special interest could be the target times. The exact timing of those and also the distance between them might play a big role when additionally delays are incorporated because delays directly act on times.

Moreover, the known problem of adding and deleting spikes could also be a contributing factor. This should be further investigated. The fact that the training without delays worked better might also be an indication for that when the delays stay constant there is less change in the network and when moving spikes forward and backward in time this can easily lead to deleting/adding spikes as in the example in section 6.1. Thus integrating the solution form Nowotny, Turner, and Knight [29] would maybe help solving the problem.

Outlook

In the future, the implemented arbitrary topologies could be extended to also support synaptic delays, where the `step` function is vectorized and each neuron has its own queue. However, it would probably still make sense to introduce a minimal delay because in the proposed framework of evolving the layers, this minimal delay ensures that the layers are moving forward in time. Consequently, it would also make sense to extend the learning of synaptic delays to those arbitrary topologies. However, it should be noted that it would be less flexible if a minimal delay is kept as a constraint.

Another possible objective could be to use the extension of the EventProp algorithm to learn delays on the neuromorphic platform BrainScales-2. Although the chip does not natively support delays, there have been workarounds e.g. by Göltz et al. [12] or Tabel [33]. In Göltz et al. [12] 'parrot' neurons are used to exploit the on-chip dynamics to create artificial delays whereas Tabel [33] used rerouting of spikes through the host computer. Moreover, because hardware necessarily has delays, the proposed method for learning weights with constant delays could be used for more precise in-the-loop traininig.

Acknowledgments (Danksagungen)

In erster Linie gilt mein Dank meinen Betreuern Elias, Philipp und Eric, die mich sehr gut betreut haben und wenn ich Fragen hatte, immer gerne geholfen haben. Außerdem möchte ich mich für das Korrekturlesen dieser Arbeit bei Elias, Eric, Philipp, Ben, Nils und David bedanken. Bei Carl bedanke ich mich dafür, dass er immer sehr bereitwillig seine LaTeX Expertise mit mir geteilt hat.

Außerdem bedanke ich mich bei den besten beiden Bürokollegen Ben und Lennart, die meine Zeit im Büro zu einer entspannten, lustigen und schönen Zeit gemacht haben. Insbesondere bedanke ich mich bei Ben, dass er mich in den letzten, sehr hektischen Stunden vor der Abgabe sehr unterstützt hat.

Bei Philipp bedanke ich mich für unsere Tischtennismatches, die immer sehr viel Spaß gemacht haben. Des Weiteren bedanke ich mich bei Tim und Simon für unsere "Geheimmissionen", die uns alle sehr weiter gebracht haben. Mein Dank gilt auch allen Darts und Tischkicker Mitstreitern für die vielen spannenden und spaßigen Momente nach der Mensa. Außerdem bedanke ich mich bei Johannes, dass ich meine Bachelorarbeit in der Electronic Vision(s) Group schreiben durfte, von der ich sehr herzlich aufgenommen wurde.

Zuletzt will ich meinen Eltern und meinen beiden Brüdern dafür danken, dass sie mich immer unterstützen.

The work carried out in this Bachelor Thesis used systems, which received funding from the European Union's Horizon 2020 Framework Programme for Research and Innovation under the Specific Grant Agreements Nos. 720270, 785907 and 945539 (Human Brain Project, HBP) and Horizon Europe grant agreement No. 101147319 (EBRAINS 2.0).

8 References

[1] Moritz Althaus. "Efficient Software for Event-based Optimization on Neuromorphic Hardware". Master thesis. Ruprecht-Karls-Universität Heidelberg, 2023.

[2] Laith Alzubaidi, Jinglan Zhang, Amjad J. Humaidi, Abdulmotaleb Al-Dujaili, Yong Duan, Omar Al-Shamma, Jorge Santamaría, Muhammad A. Fadhel, Murtadha Al-Amidie, and Laith Farhan. "Review of deep learning: concepts, CNN architectures, challenges, applications, future directions". In: *Journal of Big Data* 8.1 (2021), p. 53. DOI: 10.1186/s40537-021-00444-8. URL: https://journalofbigdata.springeropen.com/articles/10.1186/s40537-021-00444-8.

[3] Dominic Auge, Johannes Hille, Eric Mueller, et al. "A Survey of Encoding Techniques for Signal Processing in Spiking Neural Networks". In: *Neural Processing Letters* 53 (2021), pp. 4693–4710. DOI: 10.1007/s11063-021-10562-2. URL: https://doi.org/10.1007/s11063-021-10562-2.

[4] Viswanathan Balasubramanian. "Brain power". In: *Proceedings of the National Academy of Sciences of the United States of America* 118.32 (Aug. 2021), e2107022118. DOI: 10.1073/pnas.2107022118.

[5] Zhenshan Bing, Claus Meschede, Florian Röhrbein, Kai Huang, and Alois Knoll. "A Survey of Robotics Control Based on Learning-Inspired Spiking Neural Networks". In: *Frontiers in Neurorobotics* 12 (2018). URL: https://api.semanticscholar.org/CorpusID:49567933.

[6] 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*. Version 0.3.13. 2018. URL: http://github.com/google/jax.

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

[8] Benjamin Cramer, Yannik Stradmann, Johannes Schemmel, and Friedemann Zenke. "The Heidelberg Spiking Data Sets for the Systematic Evaluation of Spiking Neural Networks". In: *IEEE Transactions on Neural Networks and Learning Systems* 33.7 (2022), pp. 2744–2757. DOI: 10.1109/TNNLS.2020.3044364.

[9] Jason K. Eshraghian, Max Ward, Emre O. Neftci, Xinxin Wang, Gregor Lenz, Girish Dwivedi, Mohammed Bennamoun, Doo Seok Jeong, and Wei D. Lu. "Training Spiking Neural Networks Using Lessons From Deep Learning". In: *Proceedings of the IEEE* 111.9 (2023), pp. 1016–1054. DOI: 10.1109/JPROC.2023.3308088.

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

[11] 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". In: *Nat. Mach. Intell.* 3.9 (2021), pp. 823–835. DOI: 10.1038/s42256-021-00388-x.

[12] Julian Göltz, Jimmy Weber, Laura Kriener, Peter Lake, Melika Payvand, and Mihai A. Petrovici. *DelGrad: Exact gradients in spiking networks for learning transmission delays and weights*. 2024. arXiv: 2404.19165 [cs.NE].

[13] T. H. Gronwall. "Note on the derivatives with respect to a parameter of solutions of a system of differential equations". In: *Ann. Math.* 20 (1919), pp. 292–296.

[14] Ilyass Hammouamri, Ismail Khalfaoui-Hassani, and Timothée Masquelier. *Learning Delays in Spiking Neural Networks using Dilated Convolutions with Learnable Spacings*. 2023. arXiv: 2306.17670 [cs.NE]. URL: https://arxiv.org/abs/2306.17670.

[15] Alan Lloyd Hodgkin and Andrew F. Huxley. "A quantitative description of membrane current and its application to conduction and excitation in nerve." In: *J Physiol* 117.4 (Aug. 1952), pp. 500–544. ISSN: 0022-3751. URL: http://view.ncbi.nlm.nih.gov/pubmed/12991237.

[16] Quasar Jarosz. *Neuron Hand-tuned*. Accessed: 2025-01-18. 2009. URL: https://commons.wikimedia.org/wiki/File:Neuron_Hand-tuned.svg.

[17] Jared Kaplan, Sam McCandlish, Tom Henighan, Tom B. Brown, Benjamin Chess, Rewon Child, Scott Gray, Alec Radford, Jeffrey Wu, and Dario Amodei. *Scaling Laws for Neural Language Models*. 2020. arXiv: 2001.08361 [cs.LG]. URL: https://arxiv.org/abs/2001.08361.

[18] Ismail Khalfaoui-Hassani, Thomas Pellegrini, and Timothée Masquelier. *Dilated convolution with learnable spacings*. 2023. arXiv: 2112.03740 [cs.CV]. URL: https://arxiv.org/abs/2112.03740.

[19] Laura Kriener, Julian Göltz, and Mihai A. Petrovici. "The Yin-Yang Dataset". In: *Neuro-Inspired Computational Elements Conference*. NICE 2022. Virtual Event, USA: Association for Computing Machinery, 2022, pp. 107–111. ISBN: 9781450395595. DOI: 10.1145/3517343.3517380.

[20] Louis Lapicque. “Recherches quantitatives sur l’excitation electrique des nerfs traitee comme une polarization”. In: *Journal de Physiologie et Pathologie General* 9 (1907), pp. 620–635.

[21] R. Legenstein, D. Pecevski, and W. Maass. “A learning theory for reward-modulated spike-timing-dependent plasticity with application to biofeedback”. In: *PLoS Computational Biology* 4.10 (2008), e1000180.

[22] W. Maass. “Networks of spiking neurons: the third generation of neural network models”. In: *Neural Networks* 10 (1997), pp. 1659–1671.

[23] Mahyar Madadi Asl, Alireza Valizadeh, and Peter A. Tass. “Dendritic and Axonal Propagation Delays May Shape Neuronal Networks With Plastic Synapses”. In: *Frontiers in Physiology* 9 (Dec. 2018), p. 1849. DOI: 10.3389/fphys.2018.01849. URL: https://www.frontiersin.org/articles/10.3389/fphys.2018.01849/full.

[24] Sean McDougall, Wanette Vargas Riad, Andrea Silva-Gotay, Elizabeth R. Tavares, Divya Harpalani, Geng-Lin Li, and Heather N. Richardson. “Myelination of Axons Corresponds with Faster Transmission Speed in the Prefrontal Cortex of Developing Male Rats”. In: *eNeuro* 5.4 (Sept. 2018), ENEURO.0203–18.2018. DOI: 10.1523/ENEURO.0203-18.2018. URL: https://doi.org/10.1523/ENEURO.0203-18.2018.

[25] Balázs Mészáros, James C. Knight, and Thomas Nowotny. *Efficient Event-based Delay Learning in Spiking Neural Networks*. 2025. arXiv: 2501.07331 [cs.NE]. URL: https://arxiv.org/abs/2501.07331.

[26] Ifiok D. Mienye, Theo G. Swart, and Gabriel Obaido. “Recurrent Neural Networks: A Comprehensive Review of Architectures, Variants, and Applications”. In: *Information* 15.9 (2024), p. 517. DOI: 10.3390/info15090517. URL: https://www.mdpi.com/2078-2489/15/9/517.

[27] Eric Müller, Moritz Althaus, Elias Arnold, Philipp Spilger, Christian Pehle, and Johannes Schemmel. “jaxsnn: Event-driven Gradient Estimation for Analog Neuromorphic Hardware”. In: *Neuro-inspired Computational Elements Workshop (NICE 2024)*. 2024. DOI: 10.1109/NICE61972.2024.10548709. arXiv: 2401.16841 [cs.NE].

[28] 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”. In: *IEEE Signal Processing Magazine* 36.6 (2019), pp. 51–63. DOI: 10.1109/MSP.2019.2931595.

[29] Thomas Nowotny, James P. Turner, and James C. Knight. *Loss shaping enhances exact gradient learning with EventProp in Spiking Neural Networks*. 2024. arXiv: 2212.01232 [cs.NE]. URL: https://arxiv.org/abs/2212.01232.

[30] 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". In: *Front. Neurosci.* 16 (2022). ISSN: 1662-453X. DOI: 10.3389/fnins.2022.795876. arXiv: 2201.11063 [cs.NE]. URL: https://www.frontiersin.org/articles/10.3389/fnins.2022.795876.

[31] Christian Pehle and Jens Egholm Pedersen. *Norse — A deep learning library for spiking neural networks*. Version 0.0.7. Documentation: https://norse.ai/docs/. Jan. 2021. DOI: 10.5281/zenodo.4422025. URL: https://doi.org/10.5281/zenodo.4422025.

[32] Amit Sabne. *XLA : Compiling Machine Learning for Peak Performance*. 2020.

[33] Lennart Tabel. *Enabling Delay Learning in a Scalable Machine Learning Framework for Neuromorphic Hardware*. Bachelor's thesis. HD-KIP 24-70, 2024.

[34] Guangzhi Tang, Arpit Shah, and Konstantinos P. Michmizos. *Spiking Neural Network on Neuromorphic Hardware for Energy-Efficient Unidimensional SLAM*. 2019. arXiv: 1903.02504 [cs.RO]. URL: https://arxiv.org/abs/1903.02504.

[35] Neil C. Thompson, Kristjan Greenewald, Keeheon Lee, and Gabriel F. Manso. *The Computational Limits of Deep Learning*. 2022. arXiv: 2007.05558 [cs.LG]. URL: https://arxiv.org/abs/2007.05558.

[36] P.J. Werbos. "Backpropagation through time: what it does and how to do it". In: *Proceedings of the IEEE* 78.10 (1990), pp. 1550–1560. DOI: 10.1109/5.58337.

[37] Timo C. Wunderlich and Christian Pehle. "Event-based backpropagation can compute exact gradients for spiking neural networks". In: *Scientific Reports* 11.1 (2021), pp. 1–17. DOI: 10.1038/s41598-021-91786-z.

[38] Jiawei Zhang. *Basic Neural Units of the Brain: Neurons, Synapses and Action Potential*. 2019. arXiv: 1906.01703 [q-bio.NC]. URL: https://arxiv.org/abs/1906.01703.

[39] Zhong-Qiu Zhao, Peng Zheng, Shou-tao Xu, and Xindong Wu. *Object Detection with Deep Learning: A Review*. 2019. arXiv: 1807.05511 [cs.CV]. URL: https://arxiv.org/abs/1807.05511.

A Detailed Derivation of Gradients with Respect to Synaptic Delays

The derivation of the gradients with synaptic delays will be done here. The derivation mostly follows the one from section 4.1 but has some deviations.

We start again by splitting up the loss integral at both the presynaptic and postsynaptic spike times, where we use the special case that all postsynaptic neurons spike at the same time. We also introduce Lagrange multipliers $\lambda : t \mapsto \lambda(t) \in \mathbb{R}^N$

$$\begin{aligned}\frac{\mathrm{d}\mathcal{L}}{\mathrm{d}d_{ji}} =& \frac{\mathrm{d}}{\mathrm{d}d_{ji}} \Bigg[l_p(t^{\mathrm{pre}}) + \sum_{k=0}^{N_{\mathrm{pre}}} \int_{t_k^{\mathrm{pre}}+d_k}^{t_{k+1}^{\mathrm{pre}}} \left[l_V(V,t) + \lambda_V \cdot f_V + \lambda_I \cdot f_I \right] \mathrm{d}t \\ &+ \int_{t_{k+1}^{\mathrm{pre}}}^{t_{k+1}^{\mathrm{pre}}+d_{k+1}} \left[l_V(V,t) + \lambda_V \cdot f_V + \lambda_I \cdot f_I \right] \mathrm{d}t \Bigg]\end{aligned} \tag{A.1}$$

Similarly to before, we get the derivatives of the implicit differential equations

$$\frac{\partial f_V}{\partial d_{ji}} = \tau_{\mathrm{mem}} \frac{\mathrm{d}}{\mathrm{d}t} \frac{\partial V}{\partial d_{ji}} + \frac{\partial V}{\partial d_{ji}} - \frac{\partial I}{\partial d_{ji}} \tag{A.2}$$

$$\frac{\partial f_I}{\partial d_{ji}} = \tau_{\mathrm{syn}} \frac{\mathrm{d}}{\mathrm{d}t} \frac{\partial I}{\partial d_{ji}} + \frac{\partial I}{\partial d_{ji}} \tag{A.3}$$

Using the following equation for the derivative of t^{post}

$$\frac{\mathrm{d}t^{\mathrm{post}}}{\mathrm{d}d_{\mathrm{ji}}} = \frac{\mathrm{d}(t^{\mathrm{pre}} + d_{\mathrm{mn}})}{\mathrm{d}d_{\mathrm{ji}}} = \frac{\mathrm{d}t^{\mathrm{pre}}}{\mathrm{d}d_{\mathrm{ji}}} + \delta_{\mathrm{jm}}\delta_{\mathrm{in}} \tag{A.4}$$

we can apply the Leibniz integral rule again, resulting in

$$\begin{aligned}\left(\frac{d\mathcal{L}}{dd_{ji}}\right)_{\text{integral 1}} =& \sum_{k=0}^{N_{\mathrm{pre}}} \Bigg[\int_{t_k^{\mathrm{pre}}+d_k}^{t_{k+1}^{\mathrm{pre}}} \Bigg[\frac{\partial l_V}{\partial V} \cdot \frac{\partial V}{\partial d_{ji}} + \lambda_V \cdot \left(\tau_{\mathrm{mem}} \frac{\mathrm{d}}{\mathrm{d}t} \frac{\partial V}{\partial d_{ji}} + \frac{\partial V}{\partial d_{ji}} - \frac{\partial I}{\partial d_{ji}} \right) \\ &+ \lambda_I \cdot \left(\tau_{\mathrm{syn}} \frac{\mathrm{d}}{\mathrm{d}t} \frac{\partial I}{\partial d_{ji}} + \frac{\partial I}{\partial d_{ji}} \right) \Bigg] dt + l_{V,k+1}^{-} \frac{\mathrm{d}t_{k+1}^{\mathrm{pre}}}{\mathrm{d}d_{ji}} \Bigg|_{t_{k+1}^{\mathrm{pre}}} \\ &- l_{V,k}^{+} \left(\frac{dt_k^{\mathrm{pre}}}{\mathrm{d}d_{ji}} + \delta_{jm(k)}\delta_{in(k)} \right) \Bigg|_{t_{\mathrm{k}}^{\mathrm{pre}}} \Bigg]\end{aligned} \tag{A.5}$$

Using partial integration we again get the following

$$\int_{t_k^{\mathrm{pre}}+d_k}^{t_{k+1}^{\mathrm{pre}}} \lambda_V \cdot \frac{\mathrm{d}}{\mathrm{d}t}\frac{\partial V}{\partial d_{ji}} dt = -\int_{t_k^{\mathrm{pre}}+d_k}^{t_{k+1}^{\mathrm{pre}}} \dot{\lambda}_V \cdot \frac{\partial V}{\partial d_{ji}} \mathrm{d}t + \left[\lambda_V \cdot \frac{\partial V}{\partial d_{ji}}\right]_{t_k^{\mathrm{pre}}+d_k}^{t_{k+1}^{\mathrm{pre}}} \tag{A.6}$$

$$\int_{t_k^{\mathrm{pre}}+d_k}^{t_{k+1}^{\mathrm{pre}}} \lambda_I \cdot \frac{\mathrm{d}}{\mathrm{d}t}\frac{\partial I}{\partial d_{ji}} dt = -\int_{t_k^{\mathrm{pre}}+d_k}^{t_{k+1}^{\mathrm{pre}}} \dot{\lambda}_I \cdot \frac{\partial I}{\partial d_{ji}} \mathrm{d}t + \left[\lambda_I \cdot \frac{\partial I}{\partial d_{ji}}\right]_{t_k^{\mathrm{pre}}+d_k}^{t_{k+1}^{\mathrm{pre}}} \tag{A.7}$$

and substituting it back into eq. (A.1) we arrive at

$$\begin{aligned}\left(\frac{d\mathcal{L}}{dd_{ji}}\right)_{\text{integral 1}} = \sum_{k=0}^{N_{\mathrm{pre}}} \Bigg[& \int_{t_k^{\mathrm{pre}}+d_k}^{t_{k+1}^{\mathrm{pre}}} \left[\left(\frac{\partial l_V}{\partial V} - \tau_{\mathrm{mem}}\dot{\lambda}_V + \lambda_V\right)\cdot\frac{\partial V}{\partial d_{ji}} + \left(-\tau_{\mathrm{syn}}\dot{\lambda}_I + \lambda_I - \lambda_V\right)\right]\mathrm{d}t \\ & + \frac{\partial l_p}{\partial t_k^{\mathrm{pre}}}\frac{\mathrm{d}t_k^{\mathrm{pre}}}{\mathrm{d}d_{ji}} + l_{V,k+1}^{-}\frac{\mathrm{d}t_{k+1}^{\mathrm{pre}}}{\mathrm{d}d_{ji}}\bigg|_{t_{k+1}^{\mathrm{pre}}} - l_{V,k}^{+}\left(\frac{dt_k^{\mathrm{pre}}}{\mathrm{d}d_{ji}} + \delta_{jm(k)}\delta_{in(k)}\right)\bigg|_{t_k^{\mathrm{post}}} \\ & + \tau_{\mathrm{mem}}\left[\lambda_V\cdot\frac{\partial V}{\partial d_{ji}}\right]_{t_k^{\mathrm{pre}}+d_k}^{t_{k+1}^{\mathrm{pre}}} + \tau_{\mathrm{syn}}\left[\lambda_V\cdot\frac{\partial I}{\partial d_{ji}}\right]_{t_k^{\mathrm{pre}}+d_k}^{t_{k+1}^{\mathrm{pre}}}\end{aligned} \tag{A.8}$$

Again, the same adjoint equations can be chosen

$$\tau_{\mathrm{mem}}\lambda_V' = -\lambda_V - \frac{\partial l_V}{\partial V} \tag{A.9}$$

$$\tau_{\mathrm{mem}}\lambda_I' = -\lambda_I + \lambda_V \tag{A.10}$$

Also including the terms from the second integral leads to

$$\begin{aligned}\frac{d\mathcal{L}}{dd_{ji}} = \sum_{k=0}^{N_{\mathrm{pre}}} \Bigg[& \frac{\partial l_p}{\partial t_k^{\mathrm{pre}}}\frac{\mathrm{d}t_k^{\mathrm{pre}}}{\mathrm{d}d_{ji}} + l_{V,k+1}^{-}\frac{\mathrm{d}t_{k+1}^{\mathrm{pre}}}{\mathrm{d}d_{ji}}\bigg|_{t_{k+1}^{\mathrm{pre}}} - l_{V,k}^{+}\left(\frac{dt_k^{\mathrm{pre}}}{\mathrm{d}d_{ji}} + \delta_{jm(k)}\delta_{in(k)}\right)\bigg|_{t_k^{\mathrm{post}}} \\ & + \tau_{\mathrm{mem}}\left[\lambda_V\cdot\frac{\partial V}{\partial d_{ji}}\right]_{t_k^{\mathrm{pre}}+d_k}^{t_{k+1}^{\mathrm{pre}}} + \tau_{\mathrm{syn}}\left[\lambda_V\cdot\frac{\partial I}{\partial d_{ji}}\right]_{t_k^{\mathrm{pre}}+d_k}^{t_{k+1}^{\mathrm{pre}}} + l_{\mathrm{V,k+1}}^{-}\left(\frac{\mathrm{d}t_{\mathrm{k+1}}^{\mathrm{pre}}}{d_{\mathrm{ji}}} + \delta_{\mathrm{jm(k+1)}}\delta_{\mathrm{in(k+1)}}\right)\bigg|_{t_{k+1}^{\mathrm{post}}} \\ & - l_{V,k+1}^{+}\frac{\mathrm{d}t_{k+1}^{\mathrm{pre}}}{\mathrm{d}d_{ji}}\bigg|_{t_{k+1}^{\mathrm{pre}}} + \tau_{\mathrm{mem}}\left[\lambda_V\cdot\frac{\partial V}{\partial d_{ji}}\right]_{t_{k+1}^{\mathrm{pre}}}^{t_{k+1}^{\mathrm{pre}}+d_{k+1}} + \tau_{\mathrm{syn}}\left[\lambda_V\cdot\frac{\partial I}{\partial d_{ji}}\right]_{t_{k+1}^{\mathrm{pre}}}^{t_{k+1}^{\mathrm{pre}}+d_{k+1}}\end{aligned} \tag{A.11}$$

By choosing parameter-independent initial conditions for the state variables and setting the adjoint variables to $\lambda_V(T) = \lambda_I(T) = 0$ the corresponding bounding terms vanish for $t = 0$ and $t = T$. Moreover, we know that $t_0^{\mathrm{pre}} + d_0 = 0$ and $t_{N_{\mathrm{pre}}+1}^{\mathrm{pre}} + d_{k+1} = T$ and thus the derivatives

of t_k^{pre} are equal to zero for $k = 0, N_{\text{pre}+1}$. The terms can again be grouped again into two variables ξ and ψ

$$\frac{d\mathcal{L}}{dd_{ji}} = \sum_{k=1}^{N_{\text{pre}}} \xi_k + \psi_k \tag{A.12}$$

$$\begin{aligned}\xi_k = \Bigg[\tau_{\text{mem}} \left(\lambda_V^- \cdot \frac{\partial V^-}{\partial d_{ji}} - \lambda_V^+ \cdot \frac{\partial V^+}{\partial d_{ji}}\right) + \tau_{\text{syn}} \left(\lambda_I^- \cdot \frac{\partial I^-}{\partial d_{ji}} - \lambda_I^+ \cdot \frac{\partial I^+}{\partial d_{ji}}\right) \\ + \frac{\partial l_p}{\partial t_k^{\text{pre}}} \frac{\mathrm{d}t_k^{\text{pre}}}{\mathrm{d}d_{ji}} + l_{V,k}^- \frac{\mathrm{d}t_k^{\text{pre}}}{\mathrm{d}d_{ji}} - l_{V,k}^+ \frac{\mathrm{d}t_k^{\text{pre}}}{\mathrm{d}d_{ji}}\Bigg] \Bigg|_{t_k^{\text{pre}}}\end{aligned} \tag{A.13}$$

$$\psi_k = \left[\tau_{\text{mem}} \left(\lambda_V^- \cdot \frac{\partial V^-}{\partial d_{ji}} - \lambda_V^+ \cdot \frac{\partial V^+}{\partial d_{ji}}\right) + \tau_{\text{syn}} \left(\lambda_I^- \cdot \frac{\partial I^-}{\partial d_{ji}} - \lambda_I^+ \cdot \frac{\partial I^+}{\partial d_{ji}}\right)\right] \Bigg|_{t_k^{\text{post}}} \tag{A.14}$$

where it was again used, that $l_V^+ - l_V^-$ is equal to zero at the time of the postsynaptic spike. Thus, we have arrived at the same form as in the derivation for the weights. Now the partial derivative jumps are again derived where we consider a spiking neuron n and all other silent neurons m.

Transition at presynaptic spike time Again only the membrane potential changes. The condition for the threshold from eq. (4.20) can be differentiated again and by using $(\dot{V}^-)_n \neq 0$ we get

$$\frac{\mathrm{d}t^{\text{post}}}{\mathrm{d}d_{ji}} = -\frac{1}{(\dot{V}^-)_n} \left(\frac{\partial V^-}{\partial d_{ji}}\right)_n \tag{A.15}$$

From the differentiation of the reset after the spike (4.23) we get

$$\left(\frac{\partial V^+}{\partial d_{ji}}\right)_n + (\dot{V}^+)_n \frac{\mathrm{d}t^{\text{post}}}{\mathrm{d}d_{ji}} = 0 \tag{A.16}$$

resulting in

$$\left(\frac{\partial V^+}{\partial d_{ji}}\right)_n = \frac{(\dot{V}^+)_n}{(\dot{V}^-)_n} \left(\frac{\partial V^-}{\partial d_{ji}}\right)_n \tag{A.17}$$

Because the membrane potential of non-spiking neurons and all synaptic currents do not experience jumps, the other conditions are

$$\left(\frac{\partial V^+}{\partial d_{ji}}\right)_m = \left(\frac{\partial V^-}{\partial d_{ji}}\right)_m \qquad \text{and} \qquad \frac{\partial I^+}{\partial d_{ji}} = \frac{\partial I^-}{\partial d_{ji}} \tag{A.18}$$

Transition at postsynaptic spike times The spiking neuron's state does not experience any transitions at the postsynaptic spike time and thus the partial derivatives do not experience any jumps:

$$\left(\frac{\partial V^+}{\partial d_{ji}}\right)_n = \left(\frac{\partial V^-}{\partial d_{ji}}\right)_n \qquad \text{and} \qquad \left(\frac{\partial I^+}{\partial d_{ji}}\right)_n = \left(\frac{\partial I^-}{\partial d_{ji}}\right)_n \tag{A.19}$$

By differentiating $(V^+)_m = (V^-)_m$, using the jumping condition for $\dot{V}$ (4.30) and substituting (A.15) we get

$$\begin{aligned}\left(\frac{\partial V^+}{\partial d_{ji}}\right)_m &= \left(\frac{\partial V^-}{\partial d_{ji}}\right)_m - \tau_{\text{mem}}^{-1} w_{mn} \frac{\mathrm{d}t^{\text{post}}}{\mathrm{d}_{ji}} \\ &= \left(\frac{\partial V^-}{\partial w_{\text{ji}}}\right)_{\text{m}} + \left[\frac{1}{\tau_{\text{mem}}(\dot{V}^-)_{\text{n}}} w_{\text{mn}} \left(\frac{\partial V^-}{\partial w_{\text{ji}}}\right)_{\text{n}}\right]\Bigg|_{t_{\text{k}}^{\text{pre}}} - \tau_{\text{mem}}^{-1} w_{\text{mn}} \delta_{\text{jm}} \delta_{\text{in}}\end{aligned} \tag{A.20}$$

where the relation from eq. (A.4) was used again resulting in the Kronecker deltas. Also, there is again a term with a partial derivative from the spiking neuron n at the time t^{pre}. This is a major difference compared to the gradients for the weights. For the synaptic current $(I^+)_m = (I^-)_m + w_{mn}$ can be differentiated leading to

$$\left(\frac{\partial I^+}{\partial d_{ji}}\right)_m + (\dot{I}^+)_m \frac{\mathrm{d}t^{\text{post}}}{\mathrm{d}d_{ji}} = \left(\frac{\partial I^-}{\partial d_{ji}}\right)_m + (\dot{I}^-)_m \frac{\mathrm{d}t^{\text{post}}}{\mathrm{d}d_{ji}} \tag{A.21}$$

where it was used that the weights do not depend on the synaptic delays. Now we can use the jumping condition for $\dot{I}$ (4.35) and (A.15) to arrive at

$$\begin{aligned}\left(\frac{\partial I^+}{\partial d_{ji}}\right)_m &= \left(\frac{\partial I^-}{\partial d_{ji}}\right)_m + \tau_{\text{syn}}^{-1} w_{mn} \frac{\mathrm{d}t^{\text{post}}}{\mathrm{d}d_{ji}} + \delta_{\text{in}} \delta_{jm} \\ &= \left(\frac{\partial V^-}{\partial w_{\text{ji}}}\right)_{\text{m}} + \left[\frac{1}{\tau_{\text{mem}}(\dot{V}^-)_{\text{n}}} w_{\text{mn}} \left(\frac{\partial V^-}{\partial w_{\text{ji}}}\right)_{\text{n}}\right]\Bigg|_{t_{\text{k}}^{\text{pre}}} - \tau_{\text{mem}}^{-1} w_{\text{mn}} \delta_{\text{jm}} \delta_{\text{in}}\end{aligned} \tag{A.22}$$

where eq. (A.4) as used again. There is also a partial derivative term from neuron n at time t^{pre} again.

We can substitute all partial derivative jumps at the presynaptic spike times into ξ_k now and also include the two terms which arise in eq. (A.20) and eq. (A.22). This yields

$$\begin{aligned}\xi_k' = \Bigg[&\sum_{m \neq k} \left[\tau_{\text{mem}}(\lambda_V^- - \lambda_V^+)_m \left(\frac{\partial V^-}{\partial d_{ji}}\right) + \tau_{\text{syn}}(\lambda_I^- - \lambda_I^+)_m \left(\frac{\partial I^-}{\partial d_{ji}}\right)\right] \\ &+ \left(\frac{\partial V^-}{\partial d_{ji}}\right)_{n(k)} \Bigg[\tau_{\text{mem}} \left(\lambda_V^- - \frac{(\dot{V}^+)_{n(k)}}{(\dot{V}^-)_{n(k)}} \lambda_V^+\right)_{n(k)} \\ &+ \frac{1}{(\dot{V}^-)_{n(k)}} \left(\sum_{m \neq k} \left[w_{mn(k)}(\lambda_I^+ - \lambda_V^+)_m + l_V^- - l_V^+\right]\Big|_{t_k^{\text{post}}} - \frac{\partial l_p}{\partial t_k^{\text{post}}} + l_V^+ - l_V^-\right)\Bigg] \\ &+ \tau_{\text{syn}}(\lambda_I^- - \lambda_I^+) \left(\frac{\partial I^-}{\partial d_{ji}}\right)_{n(k)}\Bigg]\Bigg|_{t_k^{\text{pre}}}\end{aligned} \tag{A.23}$$

This leads to the same jump conditions as in the derivation in section 4.1:

$$(\lambda_V^-)_m = (\lambda_V^+)_m \qquad \text{and} \qquad \lambda_V^- = \lambda_I^+ \tag{A.24}$$

and for λ_V

$$\begin{aligned}(\lambda_V^-)_n =& \frac{(\dot{V}^+)_n}{(\dot{V}^+)_n}(\lambda_V^+)_n + \frac{1}{\tau_{\text{mem}}(\dot{V}^-)_n}\left[\sum_{m\neq k}\left[w_{mn(k)}(\lambda_V^+ - \lambda_I^+)_m\right]\Big|_{t_k^{\text{post}}}\right. \\ &\left. + \frac{\partial l_p}{\partial t_k^{\text{post}}} - l_V^+ + l_V^-\right]\end{aligned} \tag{A.25}$$

By substituting the partial derivative jumps at the postsynaptic spike time into ψ_k and excluding the two terms that went into ξ_k' we have

$$\begin{aligned}\psi_k' = &\left[\sum_{m\neq k}\left[\tau_{\text{mem}}(\lambda_V^- - \lambda_V^+)_m\left(\frac{\partial V^-}{\partial d_{ji}}\right)_m + \tau_{\text{syn}}(\lambda_I^- - \lambda_I^+)_m\left(\frac{\partial I^-}{\partial d_{ji}}\right)_m + d_{ji}(\lambda_V - \lambda_I)_j\Big|_{t^{\text{post}}}\right]\right. \\ &\left. + \tau_{\text{mem}}(\lambda_I^- - \lambda_I^+)_{n(k)}\left(\frac{\partial V^-}{\partial d_{ji}}\right)_{n(k)} + \tau_{\text{syn}}(\lambda_I^- - \lambda_I^+)_{n(k)}\left(\frac{\partial I^-}{\partial d_{ji}}\right)_{n(k)}\right]\end{aligned} \tag{A.26}$$

By choosing all adjoint variables to not experience jumps

$$\lambda_V^- = \lambda_V^+ \qquad \text{and} \qquad \lambda_I^- = \lambda_I^+ \tag{A.27}$$

all partial derivatives vanish and we are left with

$$\psi_k' = -\tau_{\text{syn}}\sum_{m\neq n(k)}\delta_{in(k)}\delta_{jm}(\lambda_I^+)_m = -\tau_{\text{syn}}\sum_{m\neq n(k)}\delta_{in(k)}(\lambda_I^+)_j \tag{A.28}$$

The remaining Kronecker Delta only is not zero if the presynaptic neuron is i and thus we can write the gradients as

$$\frac{\mathrm{d}\mathcal{L}}{\mathrm{d}d_{ji}} = \sum_{k=0}^{N}\xi_k' + \psi_k' = -\tau_{\text{syn}}\sum_{\text{spikes from i}}(\lambda_I)_j\Big|_{t_k^{\text{post}}} \tag{A.29}$$

The generalization to different synaptic delays of the neurons leading to splitting up the integral at all transition times follows exactly the same way as was argued for the synaptic

weights, see section 4.1. So we get the jump transition

$$(\lambda_V^-)_n = \frac{(\dot{V}^+)_n}{(\dot{V}^+)_n}(\lambda_V^+)_n + \frac{1}{\tau_{\mathrm{mem}}(\dot{V}^-)_n}\left[\sum_{m\neq k}\left[w_{mn(k)}(\lambda_V^+ - \lambda_I^+)_m\right]\Big|_{t_k+d_{mn(k)}} + \frac{\partial l_p}{\partial t_k^{\mathrm{post}}} - l_V^+ + l_V^-\right] \tag{A.30}$$

B Training parameters

Parameter		**Parameter**	
weight init	mean, std	batch size	64
hidden	3.2, 1.6	optimizer	Adam
output	0.5, 0.8	β	(0.9, 0.999)
delay init	mean, std	ϵ	10^{-8}
hidden	0.025, 0.015	learning rate	0.005
output	0.025, 0.015	decay	0.98
τ_{syn}	0.005	$t_{\text{target,correct}}\,[\tau_s]$	0.9
τ_{m}	0.01	$t_{\text{target,wrong}}\,[\tau_s]$	1.1
V_{th}	1.0		
V_{reset}	-1000.0		
$t_{\text{bias}}\,[\tau_{\text{syn}}]$	0.0	input size	5
$t_{\text{early}}\,[\tau_{\text{syn}}]$	0.0	hidden size	120
$t_{\text{late}}\,[\tau_{\text{syn}}]$	1.5	output size	3

Table B.1: Hyperparameters for training synaptic delays and weights on the Yin-Yang dataset, adapted from [1]

Erklärung:

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

Heidelberg, den 23.01.2025 .....