Department of Physics and Astronomy

Heidelberg University

Bachelor Thesis in Physics submitted by

Lennart Tabel

born in Heidelberg

September 2024

Enabling Delay Learning in a Scalable Machine Learning Framework for Neuromorphic Hardware

This Bachelor Thesis has been carried out by

Lennart Tabel

at the

Kirchhoff-Institute for Physics

under the supervision of

Dr. Johannes Schemmel

Enabling Delay Learning in a Scalable Machine Learning Framework for Neuromorphic Hardware

This thesis explores the implementation of delay learning in Spiking Neural Networks (SNNs) on the BrainScaleS-2 (BSS-2) neuromorphic hardware. SNNs emulate biological neural networks and transmit information through discrete spikes, with the timing of these spikes playing a crucial role in neural computation. A key challenge is the incorporation and optimization of synaptic delays, which are essential for processing temporal information effectively. To address this, we investigate the application of a Gaussian convolution algorithm for learning delays, which has shown promising results in software, but has not been applied on neuromorphic hardware yet. The research first reproduces the algorithm in software, ensuring its learning capabilities. Subsequently, the forward pass of the learning process is implemented in-the-loop with hxtorch on the BSS-2 system. To do this, we need to approximate the Gaussian shape, which is used for the backward pass in the learning process. This is done through duplicating each spike, distributing the duplicated spikes in time around a delay value, and then sending them over differently weighted synapses to the same neuron. The results demonstrate the successful in-the-loop delay learning on the BSS-2 system, paving the way for learning delays in more complex networks.

Ermöglichen von Verzögerungs-Lernen in einem skalierbaren Maschinellen Lernen Framework für Neuromorphe Hardware

In dieser Arbeit wird die Implementierung von Verzögerungs-Lernen in SNNs auf der neuromorphen Hardware BSS-2 untersucht. SNNs bilden biologische neuronale Netze nach und übertragen Informationen durch diskrete Spikes, wobei der genaue Zeitpunkt dieser Spikes eine entscheidende Rolle bei der neuronalen Berechnung spielt. Eine zentrale Herausforderung ist die Einbeziehung und Optimierung der synaptischen Verzögerungen, die für eine effektive Verarbeitung zeitlicher Informationen unerlässlich sind. Um dieses Problem anzugehen, untersuchen wir die Anwendung eines Gaußschen Faltungsalgorithmus zum Erlernen von Verzögerungen auf neuromorphen Computern, der vielversprechendes Verhalten zeigt, aber noch nicht auf neuromorphen Systemen getestet wurde. In dieser Arbeit wird der Algorithmus zunächst in Software nachgebildet, um seine Lernfähigkeit zu bestätigen. Anschließend wird der Vorwärtsdurchlauf des Lernprozesses mit hxtorch auf dem BSS-2 System in-the-loop implementiert. Dazu wird jeder Spike dupliziert, die duplizierten Spikes zeitlich um einen Verzögerungswert verteilt und dann über unterschiedlich gewichtete Synapsen an dasselbe Neuron gesendet. Die Ergebnisse zeigen, dass das Lernen von Verzögerungen im BSS-2-System erfolgreich ist und ebnen somit den Weg für das Lernen von Verzögerungen in komplexeren Netzwerken.

Contents

1 Introduction

Recent advancements in Artificial Intelligence (AI) are leading to transformative changes across everyday life and entire industries. As the models become increasingly sophisticated, they consume vast amounts of power and their training time grows sharply [14]. A limiting factor is the basic architecture of computers, the so-called von-Neumann-architecture. The separation between the Central Processing Unit (CPU) and memory — also called von Neumann bottleneck — limits its speed, ability for parallel processing, and energy efficiency, since moving data is very energy intensive [22]. While less pronounced, the same problem also exists in Graphical Processing Units (GPUs), which are typically used to train modern AI models [19].

Neuromorphic hardware aims to solve these problems by mimicking biological brains on silicon hardware. Its processing is event-driven and can be highly parallel, since there is no separation between memory and computing. Unlike traditional Artificial Neural Network (ANN), which rely on continuous-valued signals, SNNs utilize discrete spikes to transmit information. This spiking mechanism introduces the concept of temporal coding, where the timing of spikes carries crucial information.

One critical aspect of SNNs is the incorporation of delays into the synaptic transmission, which can significantly influence the network's ability to learn and process information effectively [13]. It has been shown that biological brains can learn these delays and that these plastic delays are needed in order to learn certain tasks [2]. To replicate this on neuromorphic systems Machine Learning (ML) algorithms can be used, which proves to be a difficult endeavor, mainly because of the backpropagation step. The information transport in SNNs is non-differentiable by nature since the signal spike is a binary event. Therefore, some sort of continuity needs to be reintroduced into the network while the learning takes place, in order to calculate the gradient and determine how the delays have to be tweaked to improve the network's performance on a given task.

A possible method to do so is shown by Hammouamri, Khalfaoui-Hassani, and Masquelier [9], who used a Gaussian convolution algorithm to smear out the spike in time around its delay value in the form of a Gaussian distribution. This algorithm was benchmarked against the best-performing ML algorithms in SNNs on conventional computers and achieved the

highest accuracy on the Spiking Heidelberg Digits (SHD) [6] and SSC [23] audio datasets; two datasets commonly used to evaluate the classification accuracy of SNNs. In this thesis, the Gaussian convolution algorithm will be recreated and extended to learn delays on the BSS-2 neuromorphic system.

1.1 Thesis outline

Chapter 2 gives the reader the necessary background information by delving into the theoretical and methodological foundations of the research. It begins with an overview of SNNs and the role of delays, highlighting how they affect neural computation and learning processes. A specific focus is given to the use of Gaussian convolution for modeling delays. The ML library PyTorch and how to use it with SNNs will be explained. Chapter 3 gives an overview of the BSS-2 system's architecture and capabilities, along with a discussion of the hxtorch framework, which will be used to interact with the BSS-2 hardware.
Chapter 4 demonstrates the implementation of the Gaussian delay learning algorithm in software and how the delays, used in the learning algorithm, can be realized on the hardware. The chapter concludes with a demonstration of in-the-loop delay learning on the BSS-2 hardware.

2 Methods

2.1 Spiking Neural Networks

Neurons serve as the fundamental components of SNNs. Their discovery by Golgy and Cajal [20] in the early 20th century led to the formulation of the Neuron Doctrine, a breakthrough that significantly advanced the nascent field of neuroscience. In the 1950s, Hodgkin and Huxley [10] further explored the dynamics within neurons and synapses, modelling the electrical activity inside neurons with a series of differential equations. By abstracting away certain biological details, the model can be simplified into a more general form that preserves the essential dynamics, particularly the relationship between membrane voltage and ionic currents. It is called the Leaky Integrate-and-Fire (LIF) model [7] and is the most widely used model in SNNs because of its simplicity and efficiency:

$$\tau_{\text{m}} \frac{du(t)}{dt} = -(u(t) - u_{\text{rest}}) + RI(t). \tag{2.1}$$

This equation describes how the membrane potential evolves over time, balancing the effects of natural decay and external input. The term $\frac{du(t)}{dt}$ represents the rate of change of the membrane potential, with τ_{m} being the membrane time constant that controls how quickly the neuron responds to changes. The term $-(u(t) - u_{\text{rest}})$ models the decay of the potential towards its resting state u_{rest}, reflecting the "leaky" nature of the neuron.
The influence of external inputs is represented by $RI(t)$, where $I(t)$ is the input current. A neuron emits a unitary spike s when its membrane potential exceeds the threshold ϑ, after which it instantaneously resets to u_{reset}:

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

$I(t)$ can theoretically be an arbitrary current, but in SNNs it is typically modeled by weighted presynaptic spikes or an exponentially decaying current triggered by the presynaptic spikes which we assume here. This behavior is linear over time and also over synapses meaning

that the neuron integrates over all incoming signals:

$$I_i(t) = \sum_j w_{ij} \sum_f \alpha(t - t_j^f). \tag{2.3}$$

A spike from the presynaptic neuron j is being fired at time t_j^f. The connection to the postsynaptic neuron i has a synaptic weight w_{ij}. The synaptic response function $\alpha(t)$ models the response of a spike to the input current $I_i(t)$. This is an exponential decay in our case:

$$\alpha(t) = \frac{1}{\tau_s} \exp\left(-\frac{t}{\tau_s}\right) \Theta(t), \tag{2.4}$$

where the synaptic time constant τ_s decides over the rate of decay and the Heaviside step function $\Theta(t)$ ensures that the response is zero for $t < 0$. By summing over all the fired spikes f of all the neurons whose signals propagate to neuron i, the resulting current can be determined. In fig. 2.1 the dynamic of this model is shown.

Synapses in SNNs transmit spikes from one neuron to another. The neurons are typically organized in layers and the synapses are arranged as a matrix, connecting neurons of the pre-synaptic layer to neurons of the post-synaptic layer. Synapses can be excitatory (increasing the likelihood of a postsynaptic neuron firing) or inhibitory (decreasing this likelihood). Similarly to conventional ANNs the weight of a synapse determines the strength of the inter-neuron connection and can be adjusted to change the network's behavior. The weights are the parameters typically learned in neural networks through ML algorithms like backpropagation.

Since only discrete spikes can be injected into SNNs continuous signals like most real world data need to be encoded first. This process is called spike encoding [8] and uses the timing, frequency, and patterns of spikes to convey meaning. The two most commonly used methods to encode spikes are rate coding (the frequency of spikes represents information) and temporal coding (the precise timing of spikes conveys information).

2.1.1 Backpropagation in SNNs

To calculate the dynamics of a SNN with a digital computer, the time dimension is discretized, and the equations in section 2.1 are computed numerically. The SNN can be considered a recurrent network [16] and backpropagation through time can be applied.

Spikes are binary (either 0 or 1) and therefore non-differentiable. The absence of a smooth function to describe the spike generation means that gradient-based methods can not be applied to SNNs directly. We therefore use a differentiable surrogate function to approximate the gradient when performing the backward pass [4]. Common choices for this surrogate

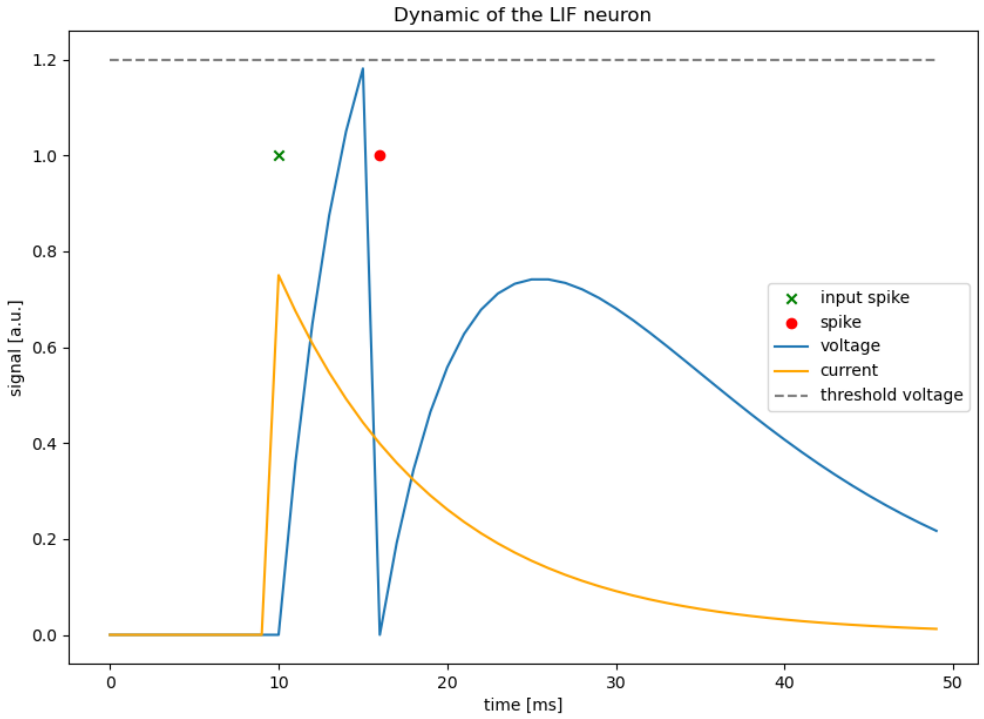

Figure 2.1: The dynamic inside a LIF neuron. The input spike causes an exponentially decaying current. The voltage inside the neuron changes by integrating the incoming current in time. When the voltage exceeds the threshold $\vartheta = 1.2$, it resets to $u_{\text{reset}} = 0$. The neuron 'leaks' causing its potential to steadily decay towards u_{rest} which equals zero here.

function include the fast sigmoid or a piecewise linear function to approximate the spike threshold behavior.

2.2 Delays

When multiple spikes arrive at the same neuron, the difference of their arrival time is a very important factor for the reaction caused in the neuron. Coincident spikes can interfere constructively and are more likely to make the neuron fire compared to spikes spread out in time. To achieve this, even though the spikes are not emitted simultaneously, the spike propagation inside synapses can be delayed. Figure 2.2 shows a demonstration of this behavior. Delays

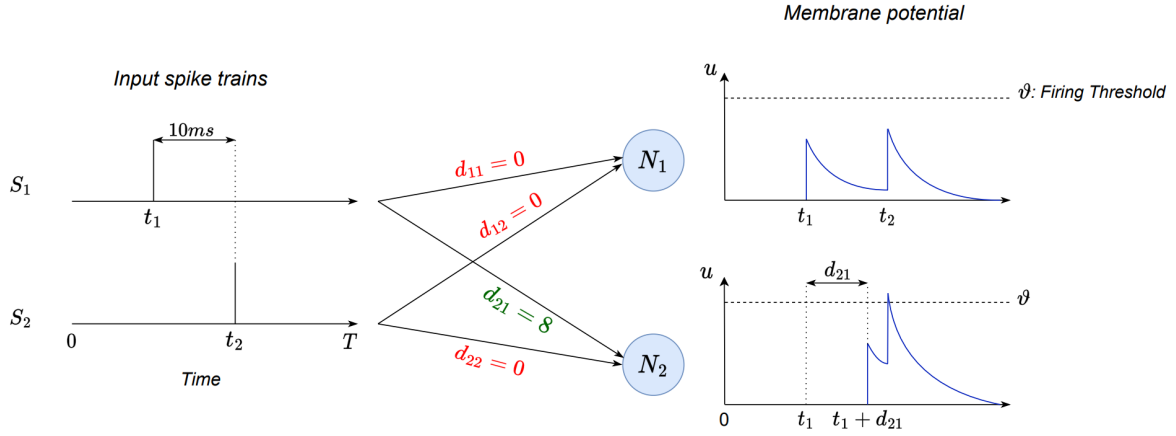

Figure 2.2: An illustration of the delay's impact. Two spike trains are being propagated to two neurons. While neuron N_1 does not spike because of the time delay of $10\,\mathrm{ms}$ between the incoming spikes, the membrane potential in N_2 exceeds the threshold ϑ. This is due to the time delay $d_{21} = 8\,\mathrm{ms}$ in the synapse from spike train S_1 to neuron N_2 which shortens the arrival time difference to only $2\,\mathrm{ms}$. Figure taken from [9].

can enable the network to not only detect synchrony patterns, but also complex spatiotemporal spike patterns. In the brain, the delay of a connection is the sum of axonal, synaptic, and dendritic delays. The axonal delay refers to the time it takes for a signal to leave a neuron and is identical for all spikes, which are emitted by the same neuron. The synaptic delay happens between two neurons and can be different for each synapse. The the dendritic delay is the time it takes for a signal to arrive at a neuron and is identical for all spikes that propagate to the same neuron. For instance, axonal delays can be shortened through myelination, an adaptive process essential for learning certain tasks [2]. This highlights that learning in the brain involves more than just synaptic plasticity; delay learning also plays a crucial role.
The synaptic transmission in SNNs with synaptic delays can be generalized by adding a de-

lay term d_{ij} to eq. (2.3), which represents the delay in the synapse connecting neuron j to neuron i:

$$I_i(t) = \sum_j w_{ij} \sum_f \alpha(t - t_j^f - d_{ij}). \tag{2.5}$$

2.2.1 Learning through Gaussian convolution

In order to find the optimal delay values for a given task inside a SNN by using backpropagation, we need to be able to differentiate a loss function with respect to the delay value. The method used in this work [9] achieves this by smearing out the delayed spikes in time around their discrete values by convolving them with a Gaussian kernel.
A spike train $S[t]$ is a sequence of discrete spikes (1 or 0) emitted or received by a neuron. A synaptic delay d_{ij} delays the transport of a spike train:

$$S_i[t] = \sum_j w_{ij} S_j[t - d_{ij}]. \tag{2.6}$$

The spike train $S_i[t]$ propagates from the neuron j through the synapse with weight w_{ij} and is being delayed by d_{ij} before it arrives at neuron i. This dynamic can also be modeled by a convolution with a kernel k_{ij}.

$$S_i[t] = \sum_j k_{ij} * S_j, \tag{2.7}$$

$$k_{ij}[n] = \begin{cases} w_{ij} & \text{if } n = T_\mathrm{d} - d_{ij} - 1, \\ 0 & \text{otherwise.} \end{cases} \tag{2.8}$$

The kernel has a length of T_d and only one non-zero element whose position decides over the delay time. This kernel is not yet differentiable, since it only has one discrete value. In order to approximate the influence of other delay times, we now spread out the non-zero element in the kernel in the form of a Gaussian distribution centered around its original position:

$$k_{ij}[n] = C_{ij} \exp\left(-\frac{1}{2}\left(\frac{n - T_d - d_{ij} + 1}{\sigma_{ij}}\right)^2\right), \tag{2.9}$$

where C_{ij} is chosen so that the sum over all kernel elements equals the synaptic weight w_{ij}. The standard deviation σ_{ij} is gradually reduced throughout the learning process for all kernels to initially capture long-term dependencies over large distances and then refine the delay with greater precision as learning progresses. Indeed, the Gaussian kernel is only used to train the model; for inference it is converted to a discrete kernel as described in eq. (2.8) by

rounding the delays. This process is shown in fig. 2.3.

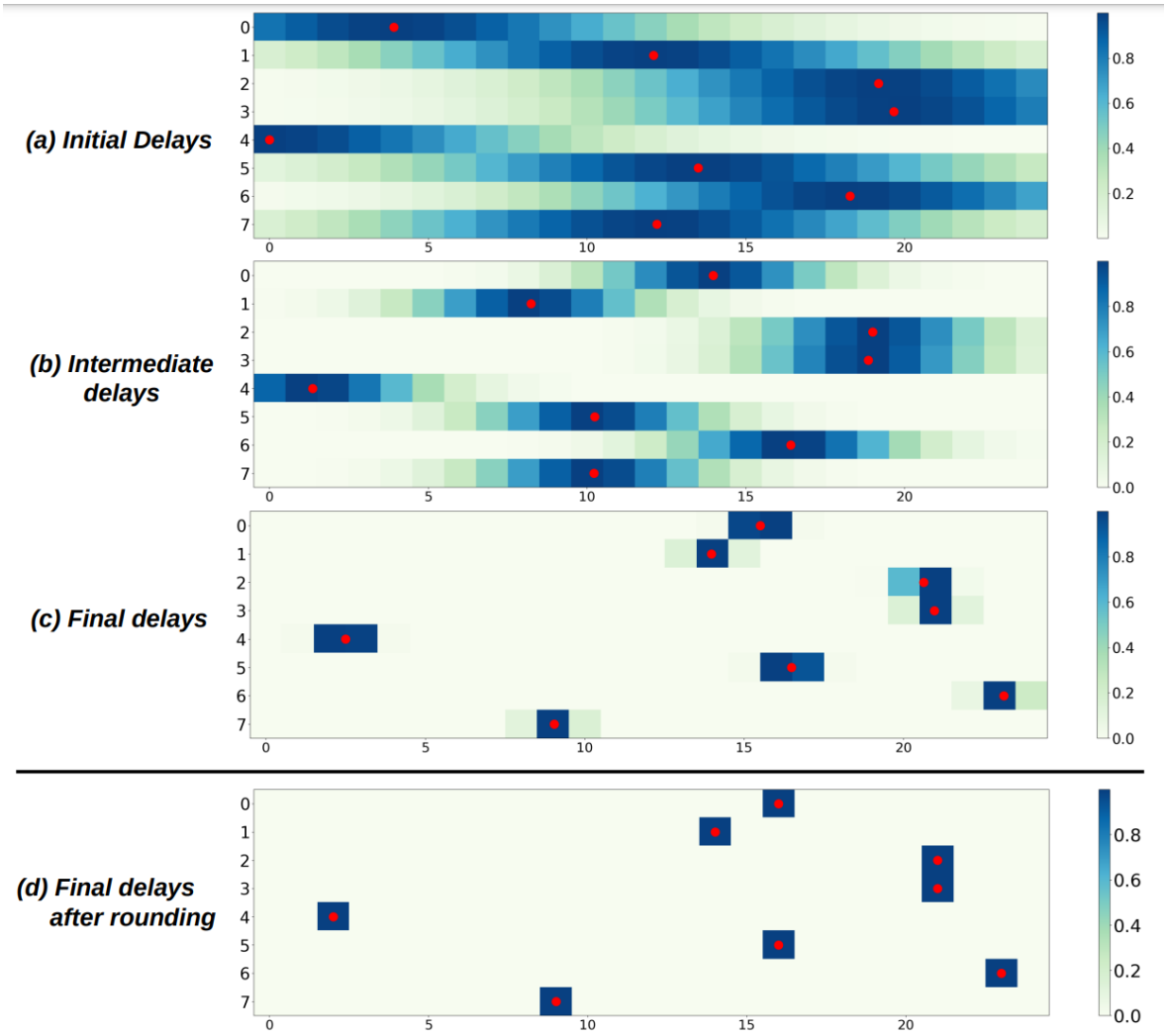

Figure 2.3: Evolution of eight kernels throughout the learning process. The x-axis refers to the delay time with a maximum delay of $T_d = 24$ while the y-axis refers to the synapse ID. (a) corresponds to the initial phase. The standard deviation of the Gaussian σ is large $\frac{T_d}{2}$, which enables the gradient to incorporate long temporal dependencies. (b) corresponds to the intermediate phase, (c) is taken from the final phase where σ is at its minimum value (0.5) and the fine-tuning of the delays is done. (d) represents the kernel after converting to the discrete form with rounded positions. Figure taken from [9].

2.3 PyTorch

PyTorch [17] is an open-source ML framework which is widely used in both academia and industry for building and training neural networks in Python. PyTorch uses tensor datatypes,

which enable accelerated and parallelized operations. It supports automatic differentiation via its 'autograd' system, that simplifies the process of backpropagation by automatically computing gradients.
In listing 2.1 it is demonstrated how to use PyTorch to create and train a model. A model can be constructed by instantiating `torch.nn.Modules` and using them in a forward method to decide what is supposed to happen to an input fed into the model. Its parameters are trained to minimize a loss function. This is done by iterating a dataset in small batches, over which the gradient is averaged. This process is repeated over epochs to gradually approach the optimal parameter values for a given task. When an input is forwarded through the model, a computational graph is created. Afterward, the loss can be calculated, which typically determines the difference between the model's output and some target values which describe the wanted behavior. The gradient is computed by traversing the computational backward from the loss value to each parameter. Those gradients determine the steepest ascend of the loss value in the high dimensional parameter-space, so by subtracting them from our parameters we take a step in the direction which decreases our loss value the most.

```python
import torch
import torch.nn as nn

# Dummy model definition
class SimpleModel(nn.Module):
    def __init__(self):
        super().__init__()
        self.linear = nn.Linear(1, 1)  # A simple linear layer
    def forward(self, x):
        return self.linear(x)
# Instantiate the model
model = SimpleModel()
# Dummy data for training
inputs = torch.tensor([[0.5], [0.1], [0.9]])
targets = torch.tensor([[1.0], [0.2], [0.8]])
learning_rate = 0.01
# Training loop
for epoch in range(100):
    # Forward pass
    outputs = model(inputs)
    # Loss function
    loss = nn.MSELoss(outputs, targets)
    # Backward pass (compute gradients)
    loss.backward()
    # Updating of each parameter
    with torch.no_grad():  # Disable gradient tracking
        for param in model.parameters():
            param -= learning_rate * param.grad
    # Set gradients to zero after updating parameters
    model.zero_grad()
```

Listing 2.1: Code for a simple neural network with one neuron for demonstration purposes. The `torch.no_grad()` method disables gradient tracking during the update step because parameter updating is not supposed to be added to the computational graph. After parameter updating the gradients are reset to zero to ensure that each update step reflects only the current state of the model's performance, not an accumulation of previous updates.

3 The BrainScaleS-2 system

3.1 Overview

The BSS-2 system [18] is a neuromorphic computing platform developed by the Electronic Vision(s) Group in Heidelberg. BSS-2 has 512 neuron circuits distributed across two hemispheres and 131.072 synapses circuits. Analog circuits are used to recreate the Adaptive Exponential Integrate and Fire (AdEx) dynamics [3] and can be interconnected to form multi-compartment neuron morphologies [11] (in this work only the LIF part of the AdEx model is used and also no multi-compartent neurons). The parameters of the LIF model and the synaptic weights can be adjusted digitally via a 10-bit on-chip Digital to Analog Converter (DAC). Therefore, it is possible to calibrate the whole system and compensate for production-induced variations in the hardware. The neuron dynamics are emulated 1000 times faster than biological real-time due to the characteristic time constants of the semiconductor substrate.
To determine the values inside a neuron over the course of an experiment, the user can utilize the Columnar Analog to Digital Converter (CADC) and Membrane Analog to Digital Converter (MADC) on the chip. The CADC can measure the potential in parallel for multiple neurons with a time resolution in the order of 1 MHz and a value resolution of 8 bit. The MADC can only measure the potential of a single neuron, but with a significantly higher resolution (value resolution of 10 bit and sampling frequency of about 29 MHz) [18]. Figure 3.1 depicts the BSS-2 hardware and architecture of the system.

3.2 hxtorch

When developing neuromorphic hardware, it is important to provide software solutions to allow non-expert users to execute their experiments. Indeed, it is important to abstract the hardware through layers to provide an interface for different hardware-close tasks. The BSS-2 system uses multiple abstraction levels, from low-level chip communication to the representation of whole experiments in software. These different layers in the stack are depicted

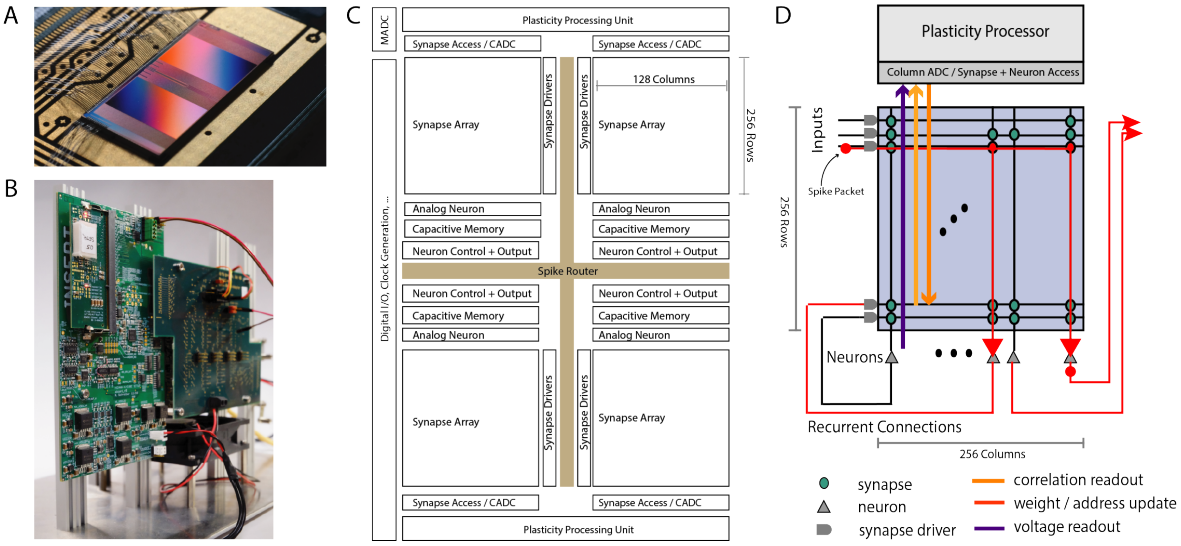

Figure 3.1: Overview of the BSS-2 architecture. (A) A picture of the BSS-2 chip. (B) Test setup with the chip on a carrier board. (C) A schematic plan of the chip with its elements is shown. The MADC is used for accurate voltage readout of a single neuron, while the CADC can read multiple neurons but with less precision. (D) A conceptual view of the signal routing through the synapse array. Figure taken from [18].

in fig. 3.2. For ML tasks the hxtorch [21] framework is used. It integrates the possibility to implicitly execute and model experiments on the BSS-2 hardware into the PyTorch ecosystem.

To perform an experiment on BSS-2 with hxtorch, we first create the elements which our network consists of. These elements are called modules in hxtorch and can either be a populations (nodes) or projections (connections of nodes). All modules that belong to the same experiment register themselves in a shared `Experiment` object.

To establish an order in the network, a placeholder handle is passed down through the modules to create a network graph. The handle is empty before the execution of the experiment and is filled with neuron dynamics data like the spike train and voltage after. The hardware execution is initiated by calling `hxtorch.snn.run`.

When creating a LIF neuron, we can configure its dynamics with various parameters like the threshold potential or the membrane time constant (see section 2.1). To make sure the neuron behaves according to these parameters, the system needs to be calibrated in the beginning.

The first module in the network graph can not be a projection. When trying to inject our input via a synapse module into the network, hxtorch therefore implicitly creates an `InputNeuron`. This module enables external input propagation to the chip and is added at the beginning of the graph (it also can be added explicitly by the user).

hxtorch directly communicates with grenade which is in the next lower level of the stack as can be seen in fig. 3.2. It manages the placing and routing of the modules inside the network

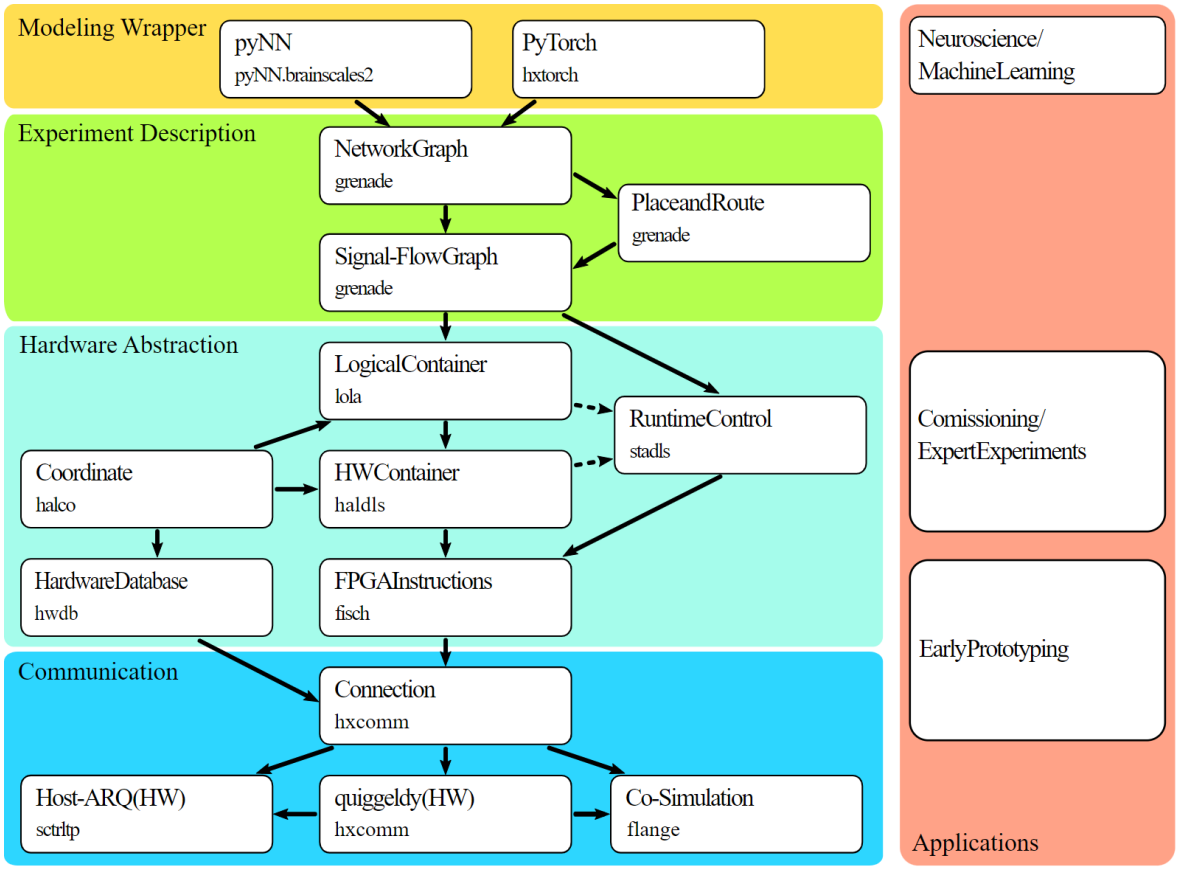

Figure 3.2: The software stack of the BSS-2 platform with distinct conceptual layers, each represented by different background colors. The white squares indicate various modules within the software stack. On the right side, it is shown which layer the programs are intended to be coded on, depending on the specific applications. In the yellow section the hxtorch framework is shown which will be used in our work. Figure taken from [15].

graph. To do so, a grenade graph is created from the network graph when the experiment is executed through `hxtorch.snn.run`. By changing how the grenade graph is created, it is possible to modify the structure of the network on hardware and how the signal is propagated in it — this will later be used to facilitate delays on hardware.

To make backpropagation possible, every module needs to hold a `func` attribute with a PyTorch-differentiable function that represents the module's behavior on hardware, allowing to propagate gradients. Listing 3.1 demonstrates the code for a basic experiment in hxtorch.

```
import hxtorch.snn as hxsnn
import hxtorch.snn.functional as F

# Defining a simple model
class SNN(torch.nn.Module):
    def __init__(self):
        super().__init__()
        # Assigning values to the LIF parameters
        lif_params = F.CalibratedCUBALIFParams(
                tau_mem=5e-6, # membrane time constant
                tau_syn=1e-5, # synaptic time constant
                leak=80,
                reset=80,
                threshold=100,
                ...)
        # Experiment creation
        self.experiment = hxsnn.Experiment(mock=False, dt=1e-6)
        # Creation of the modules
        self.syn = hxsnn.Synapse(3, 2, experiment=self.experiment)
        self.lif = hxsnn.Neuron(
            2,
            experiment=self.experiment,
            func=F.cuba_lif_integration,
            params=lif_params,
            ...)
    def forward(self, inputs):
        x0 = hxsnn.NeuronHandle(inputs)
        x1 = self.syn(x0)
        x2 = self.lif(x1)
        # Execute experiment
        hxsnn.run(self.experiment, 100)
        return x2
```

Listing 3.1: Demonstration of the code for a basic model in `hxtorch`. 3 inputs are sent to 2 LIF neurons. The `lif_params` variable holds the parameters for the LIF neuron. The `mock` argument can be used to numerically simulate the experiment in software. All modules are added to a shared experiment object.

3.2.1 In-the-Loop Learning

In order to use gradient-based learning algorithms to learn parameters in neuromorphic hardware, we need to create a model of the hardware on a conventional computer. It is possible to now train the hardware entirely with this numerical model and then later apply the learned parameters to the hardware; this is called offline learning, since it happens separated from the hardware. The problem with this is that the model and the real hardware are not identical. It is possible that gradients used in software result in parameters, which, while minimizing the loss in software, do lead to a completely different result on hardware.
To solve this, we incorporate the neuromorphic hardware into the loop between forward and backward pass during the learning process (in-the-loop learning [5]) by moving the forward pass from the software model onto the neuromorphic hardware. The learning process now incorporates the deviations between the hardware and the software model, since the loss value is calculated with the behavior of the network on the hardware.

4 Implementation

In this chapter, I will present the implementation of the Gaussian convolution learning algorithm step by step culminating in the successful in-the-loop delay learning on the BSS-2 hardware.
The correct functionality of the algorithm in PyTorch is vital here, since it will be used for the backward pass of our learning process, when we use it with the hardware later on. We therefore start by implementing the delay learning entirely in software, to confirm its learning capabilities and examine its behavior.

4.1 Learning Delays in Software

4.1.1 Integrating Delays into the Synapse

We start by implementing Gaussian delays into the synapse. In order to do this, we create an additional `torch` module called `DelayedSynapse`, which is supposed to extend the standard `Synapse`-type to incorporate synaptic delays. PyTorch's `torch.nn.conv1d` function is used to implement the convolution operation into our new class. This implementation is shown in fig. 4.1 for two different kernels. When creating a vectorized implementation with multiple synapses, every synapse has its own kernel deciding over how a spike that arrives at a certain synapse is being delayed. The code to implement this behavior is shown in listing 4.1.

4.1.2 Creating a Simple Model

We now integrate the `DelayedSynapse` class into a simple SNN. The LIF dynamics are not implemented into PyTorch, so we use a numerical LIF implementation provided by hxtorch. Listing 4.2 shows the code for the model. Figure 4.2 demonstrates a vectorized implementation of the model with three input signals arriving at two output neurons.

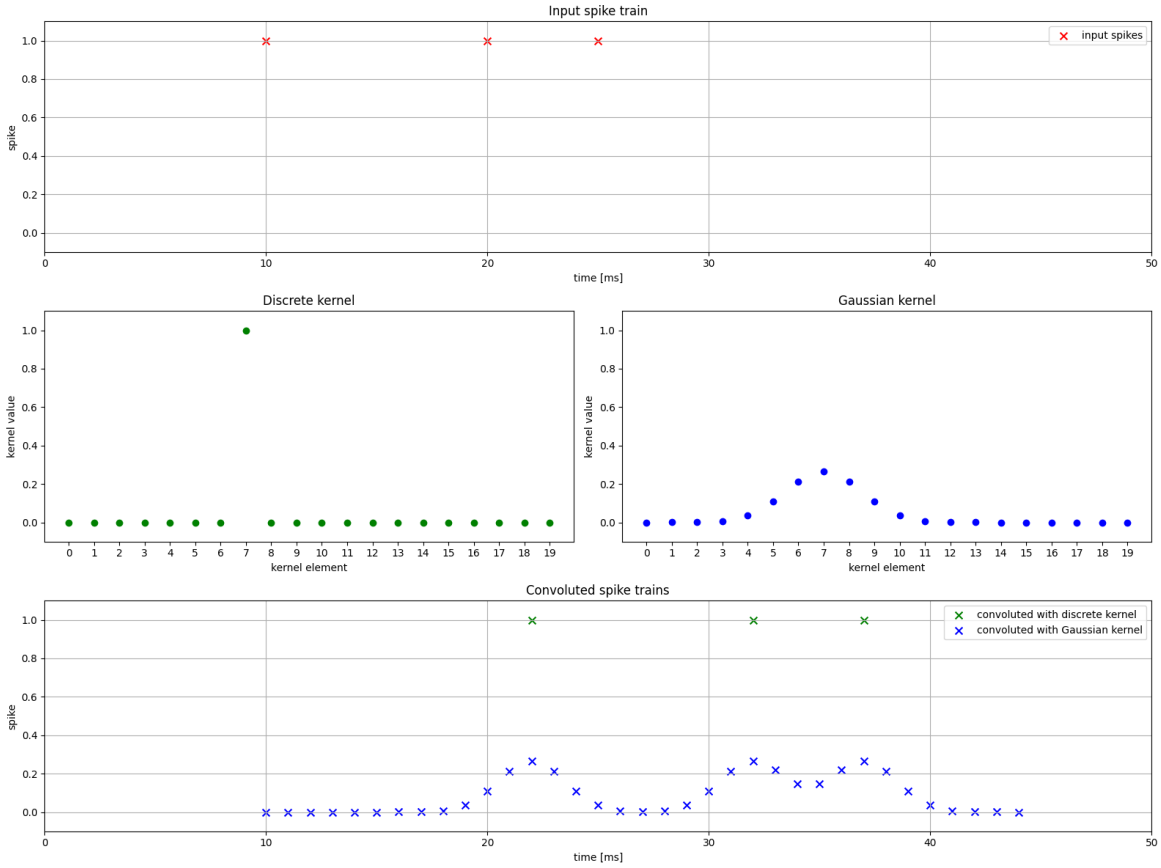

Figure 4.1: This figure demonstrates how a spike train is being delayed by convolving it with a discrete and a Gaussian kernel. The input spike train has spikes after 10, 20 and 25 milliseconds and a resolution of one millisecond. The maximum delay value for the kernel with 20 elements is therefore $d_{\max} = 19\,\mathrm{ms}$. The discrete kernel with value one at position 7 causes a delay in the signal by $d = d_{\max} - 7\,\mathrm{ms} = 12\,\mathrm{ms}$. This delay is also achieved by the Gaussian kernel, but it additionally smears out the signal in the form of a Gaussian with $\sigma = 1.5\,\mathrm{ms}$. It is important to do so in a way that the sum over all the kernel elements is equal to the weight of the synapse. This ensures that the sum of spike values which leave the synapse is the same for both kernels (on a physical level, this means the amount of charge arriving at the post-synaptic neuron is identical).

```
class DelayedSynapse(nn.Module):
    def __init__(self):
        super().__init__()
        self.input_len = 3
        self.output_len = 2
        self.max_delay = 20
        # 3 inputs, 2 outputs
        self.weights = torch.tensor([[1,0], [0,0], [0,0]])
        self.delays = torch.tensor([[7,0], [0,0], [0,0]])
        self.sigma = 1.5
    def forward(self, inputs):
        # Create Gaussian kernel for each synapse
        kernel = torch.zeros(self.output_len, self.input_len, self.
            max_delay)
        for i in range(self.output_len):
            for j in range(self.input_len):
                # Create the Gaussian distribution in the kernel
                kernel[i,j,:] =  torch.exp(-1/2 * (((torch.arange(
                self.max_delay, dtype=torch.float32)
                - self.max_delay + self.delays[j,i] + 1)
                / self.sigma)) ** 2)
                # Normalizing and weight integration
                kernel[i,j,:] /= torch.sum(kernel[i,j,:])
                kernel[i,j,:] *= self.weights[j,i]
        # Input dimensions: (batchs, in_size, in_length)
        # kernel dimensions: (out_size, batchs, kernel_length)
        output = nn.functional.conv1d(nn.functional.pad(inputs.
            transpose(0,1).transpose(0,2), (self.max_delay - 1, 0)),
             kernel)
        return output.transpose(0,1).transpose(0,2)
```

Listing 4.1: The code for our `DelayedSynapse` class. All relevant values for the Gaussian delaying are stored as attributes when creating an instance of `DelayedSynapse`. Inside the `forward` function we first create the Gaussian kernel and then convolve the input with it. Since the `nn.functional.conv1d` function accepts tensors with different dimensions compared to the tensors used in the other modules, we need to transpose them before and after the convolution. We apply a padding before the convolution with `nn.functional.pad`, which ensures that the length of our tensor stays the same after the convolution.

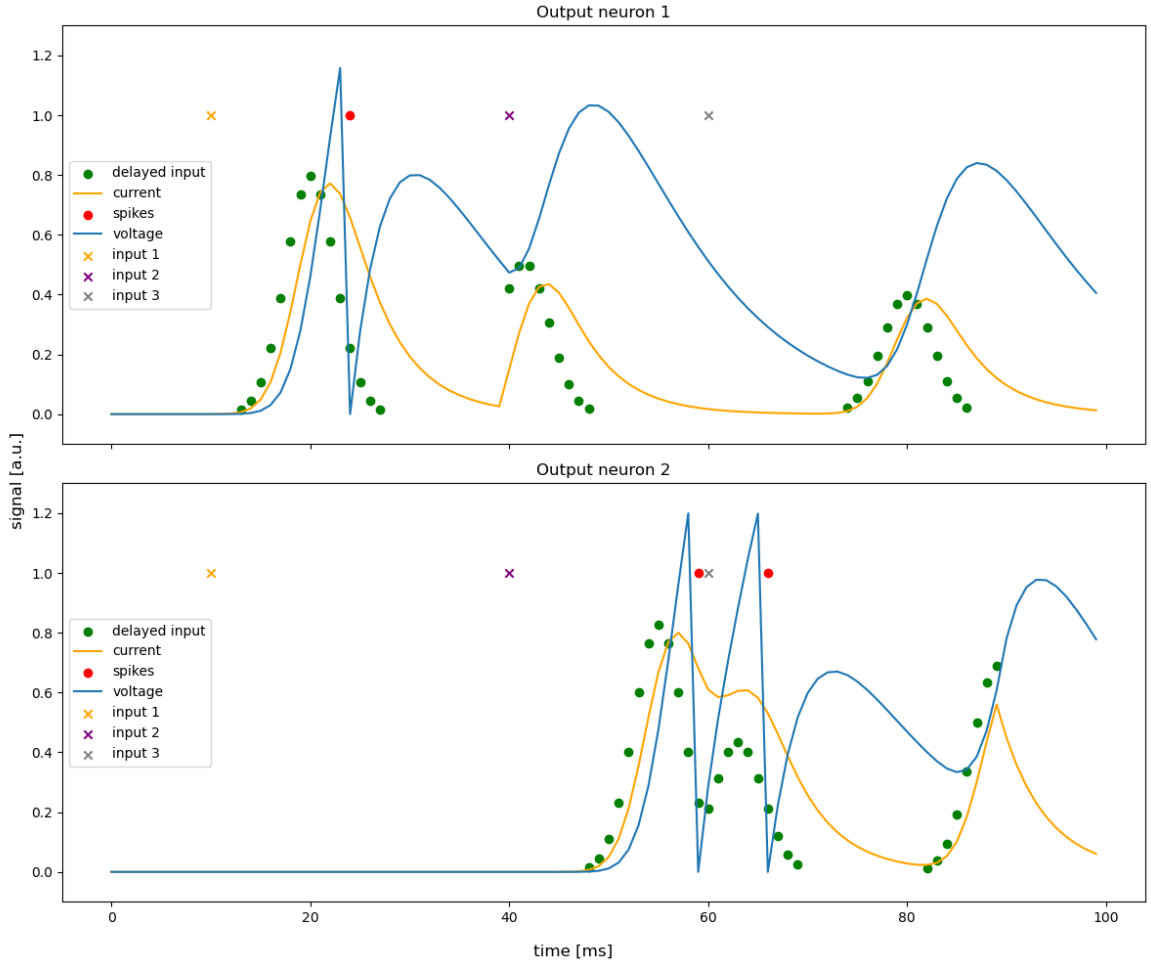

Figure 4.2: Vectorized implementation of a simple SNN. Six delayed synapses connect three inputs to two LIF neurons. The synaptic weights are equal for the synapses which propagate the signal of input 2 (purple) and input 3 (gray) and twice as strong for those of input 1 (yellow). The input spikes occur at time $t_1 = 10\,\mathrm{ms}$, $t_2 = 40\,\mathrm{ms}$ and $t_3 = 60\,\mathrm{ms}$. The first and last arriving spike at output neuron 1 (delay of $10\,\mathrm{ms}$ and $20\,\mathrm{ms}$) illustrate the general influence of the Gaussian on current and voltage in the LIF neuron, where they arrive. Instead of jumping to a high value immediately like at fig. 2.1, the current increases earlier and for longer, but slower, which also causes the voltage to increase more steadily.
The second arriving spike at output neuron 1 and the last arriving spike at output neuron 2 demonstrate what happens if the delay value ($1.5\,\mathrm{ms}$ and $49\,\mathrm{ms}$, respectively) is near the edges of the kernel (zero or the kernel length, which is 50 here). The Gaussian cuts off at a delay of $0\,\mathrm{ms}$ and $49\,\mathrm{ms}$, which causes an increase in the value of the other elements of the Gaussian spike. This happens because of the normalization since the sum of Gaussian spikes needs to equal the synaptic weight in order to not distort the strength of the spike's effect.
Input 1 and 3 at output neuron 2 are being brought together because of the delay values ($45\,\mathrm{ms}$ and $3\,\mathrm{ms}$). The current increases more strongly for longer, which causes the voltage to exceed the threshold two times causing the firing of two spikes in output neuron 2 compared to one spike at output neuron 1, where all inputs arrive spread out in time.

```
class Model(nn.Module):
    def __init__(self):
        super().__init__()
        self.syn = DelayedSynapse()
        self.lif = LIF()
    def forward(self, inputs):
        x1 = self.syn(inputs)
        x2 = self.lif(output)
        return x2
```

Listing 4.2: A simple model using PyTorch, the `LIF` class and our `DelayedSynapse`. This will also later be used for the delay learning.

4.1.3 Learning of the Delay

Since we want to learn the values inside the delay tensor, we turn them into parameters with `torch.nn.Parameters` to enable the gradient flow back to each parameter. The loss L is calculated by summing the squared differences between the spike times $t_{\text{spike},i}$ of spike i and its target times $t_{\text{target},i}$ to test the learning capabilities:

$$L = \sum_{i=1}^{n} (t_{\text{target},i} - t_{\text{spike},i})^2. \tag{4.1}$$

To get the spike time values of our neuron in a way that does not disable the possibility of gradient calculation, the class `ToSpikeTimes` [1] is used. Listing 4.3 shows the code used to achieve delay learning in PyTorch. A demonstration for the learning of two synaptic de-

```
target_times = torch.tensor([15, 80])
learning_rate = 5.
for epoch in range(100):
    spike_times = model(input)
    loss = torch.sum((target_times - spike_times)**2
    loss.backward()
    with torch.no_grad():
        model.syn.delays -= learning_rate *
                                model.syn.delays.grad
        torch.clamp_(model.syn.delays, min=0, max=49)
    model.syn.delays.grad.zero_()
```

Listing 4.3: The PyTorch code for the learning procedure. We update the parameters over 100 epochs. After each epoch we clamp the delay values to ensure they fit in the kernel with length 50. The values used in this code are the same as in fig. 4.3.

lays is shown in fig. 4.3. A thing I considered was that the learning near the minimum and maximum delay values does not work as intended, since the Gaussian cuts off here and be-

comes strongly distorted. The second spike in the lower picture in fig. 4.3 shows that this is not the case and the delay value continues to approach the desired position. One hypothesis for why this works correctly is that only the part of the Gaussian inside the kernel counts anyway, since this is the only direction in which the delay values can change. Values where the cut-off occurs are outside the kernel's range and can therefore not be a delay value to begin with.
Figure 4.4 shows the learning process in detail for a single neuron over 200 epochs. Additionally to what is plotted in fig. 4.4, I also implemented an exponential decrease in the standard deviation of the Gaussian distribution inside the kernel, since this resulted in an improved learning performance in [9]. No consistent difference in the learning behavior could be determined compared to a constant σ value. The influence of this method though can not be determined with the delay learning inside only one neuron and needs to be assessed at a later point of time, when applying it on a real task with a more complex SNN. This also goes for other hyperparameters of the learning process like learning rate, value of standard deviation, number of epochs and loss function.

4.2 Implementing Gaussian Delays in hxtorch

4.2.1 Challenges

To learn delays in-the-loop with hxtorch, we need to implement the forward pass on the BSS-2 hardware, in addition to the numerical representation in software, which is used for the backward pass. The hardware behavior needs to closely resemble the delayed Gaussian distribution in the numerical representation inside the `func` attribute. In theory, the more the numerical representation of a module matches the real hardware behavior, the better the calculated gradient matches the real gradient and the results of the learning process are optimized.
On BSS-2 only delta shaped spikes, without meaningful temporal extent compared to the emulation time, can propagate through the synapses; the shape of these spikes can not be changed.[1] Synaptic delays are not supported and also the delaying of the spike after a neuron (axonal delays) is not supported initially, but will be implemented later on.
One approach to recreating the delayed Gaussian distribution is to use multiple presynaptic neurons, each of which fires a spike at different points in time to the same postsynaptic neuron. By tweaking the synaptic weights in a way that spikes which are fired further away in time from the intended delay value are less pronounced, we can theoretically recreate the

[1] Under certain circumstances, events can be configured to have 6 bit graded values, however, this case will not be considered here.

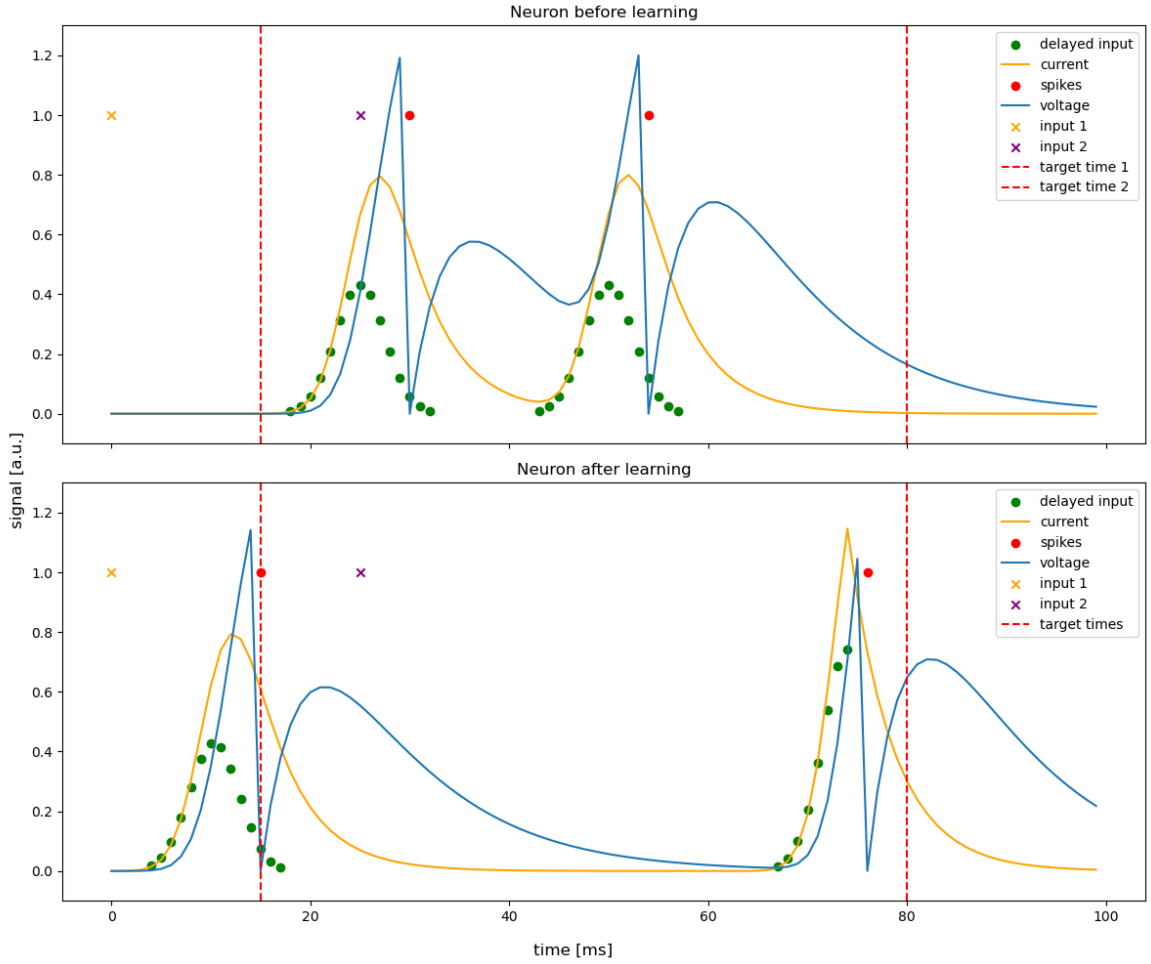

Figure 4.3: This figure shows the dynamics inside a LIF neuron before and after learning has taken place (this is the same setup as in fig. 4.2, but only neuron 1 is plotted and two input signals arrive; all weights are equal). The incoming spikes arrive at $0\,\mathrm{ms}$ and at $25\,\mathrm{ms}$, and are both initially delayed by $25\,\mathrm{ms}$. The kernel has a length of 50.
The top plot shows the dynamics of the neuron before learning and the bottom plot after learning has taken place. Target and spike time now match precisely for the first spike, and the second spike occurs later. This minimizes the loss in both cases, since the $d_{\mathrm{max}} = 49\,\mathrm{ms}$ and the second spike can therefore not further match its target time. The delay values after the learning are $d_1 = 9.537\,\mathrm{ms}$ and $d_2 = 49.0\,\mathrm{ms}$.

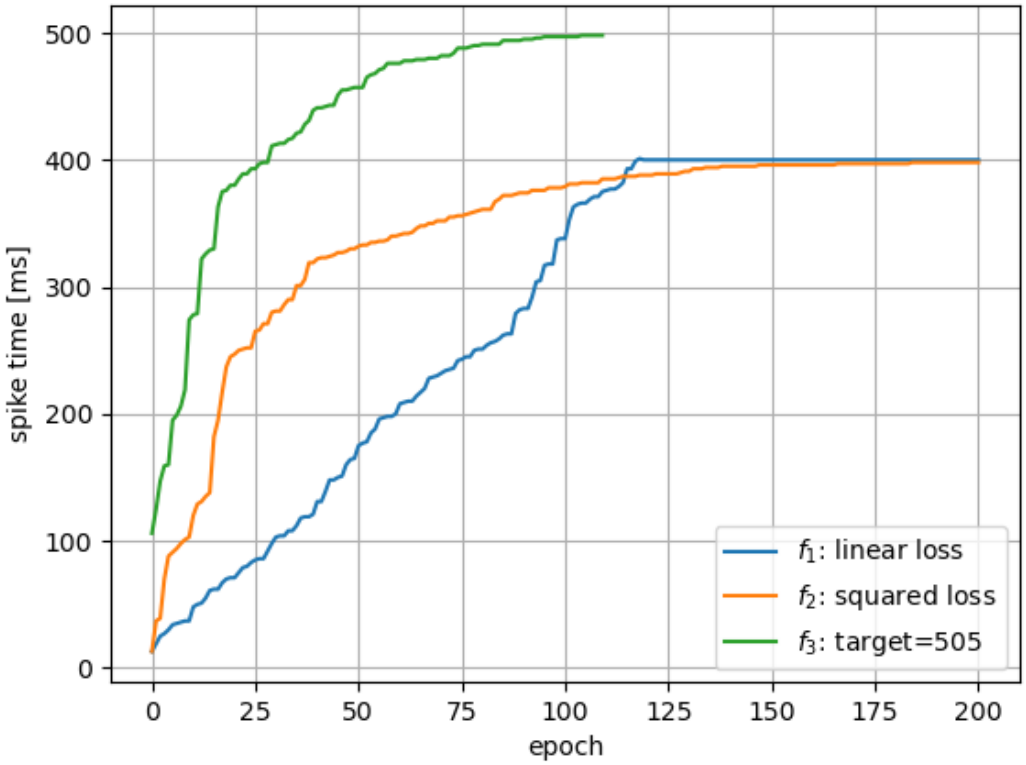

Figure 4.4: Demonstration of the learning behavior over a long period of time ($500\,\mathrm{ms}$). The input time is $0\,\mathrm{ms}$ and the target time is $400\,\mathrm{ms}$ for both f_1 and f_2. For f_3 the input time is $100\,\mathrm{ms}$ and target time is $505\,\mathrm{ms}$. The kernel length is 500 enabling the delay learning over the whole time sequence.
f_2 has the same squared loss function as used in the previous example facilitating a fast learning in the beginning, which becomes slower as the spike time matches more closely with the target time. For f_1, I have taken the absolute value of the difference between spike and target time to calculate the loss, which results in a linear-like approaching and a slight overshooting before it reaches the target time. Furthermore, the learning rate needed to be increased from 1 for f_2 to 200 for f_1 to make the two methods converge after a similar number of epochs.
f_3 demonstrates what happens when the spike time tries to approach a time outside the time period. Since the dynamic inside the neuron is never instantaneous, there always is some delay between incoming spike and emitted spike. Therefore, it is possible that the experiment stops before a spike can be emitted. This results in the problem that loss as well as gradient can not be calculated anymore, preventing the system from correcting itself, making it stuck in that state.

exact same behavior, which results from our Gaussian convolution algorithm. This neuron duplication concept is demonstrated in fig. 4.5.

Each spike that is used for the approximation is routed from an external input neuron through a synapse. On the BSS-2 hardware, each hardware neuron can have 256 synaptic inputs on a single chip instance. When we approximate the Gaussian with k values this number shrinks to $n_{\max} = \lfloor \frac{256}{k} \rfloor$, resulting in a decreased fan-in (means of partitioning the input layer could be applied to achieve larger fan-ins).

The duplication method only enables axonal delays, since the spikes are delayed after the neuron before they are forwarded to the synapses. It would be possible to achieve synaptic delays with neuron duplication, by having each synapse has its own Gaussian approximating input neurons whose delays could be individually tweaked, but this would further decrease the possible network size.

To ensure reasonable possible layer size, I decided to only implement axonal delays and a Gaussian approximation with 3 and 5 values. The time difference of the spikes is the standard deviation σ of the Gaussian in the numerical representation.

Figure 4.6 shows a comparison between the standard approximation with the Gaussian kernel used in the numerical representation of the module and the approximation of it with 3 and 5 spikes. The overall neural dynamics are similar, but since there are fewer spikes for the approximation with 3 and 5 values, the current and therefore also the membrane voltage increase in leaps. The spike time is only identical for the standard approximation and the one using 5 spikes, and $1\,\mathrm{ms}$ earlier for the approximation with 3 values. The curve of the voltage is more similar for the standard Gaussian and the approximation with 5 spikes — especially its strength after the spiking has taken place. These points justify the consideration to approximate the Gaussian with 5 spikes, even though this implies a smaller maximal SNN.

4.2.2 Delaying the Input Signal

We start by implementing the delay and duplication to the input spikes, because this is the more straight-forward task as opposed to interneuron delay and can be used to test the functionality of our neuron duplication concept on BSS-2. We want to integrate this seamlessly into hxtorch. Users should not need to change the input themselves in order to implement delays and neuron duplication, but it should be done automatically. Listing 4.4 demonstrates how we want to integrate our changes into the existing hxtorch Application Programming Interface (API). The shape of the synapses and neurons stays the same as it would be when using the standard `InputNeuron` module. The duplication and delay are done inside our new `DelayedInputNeuron`, for which we provide the necessary information

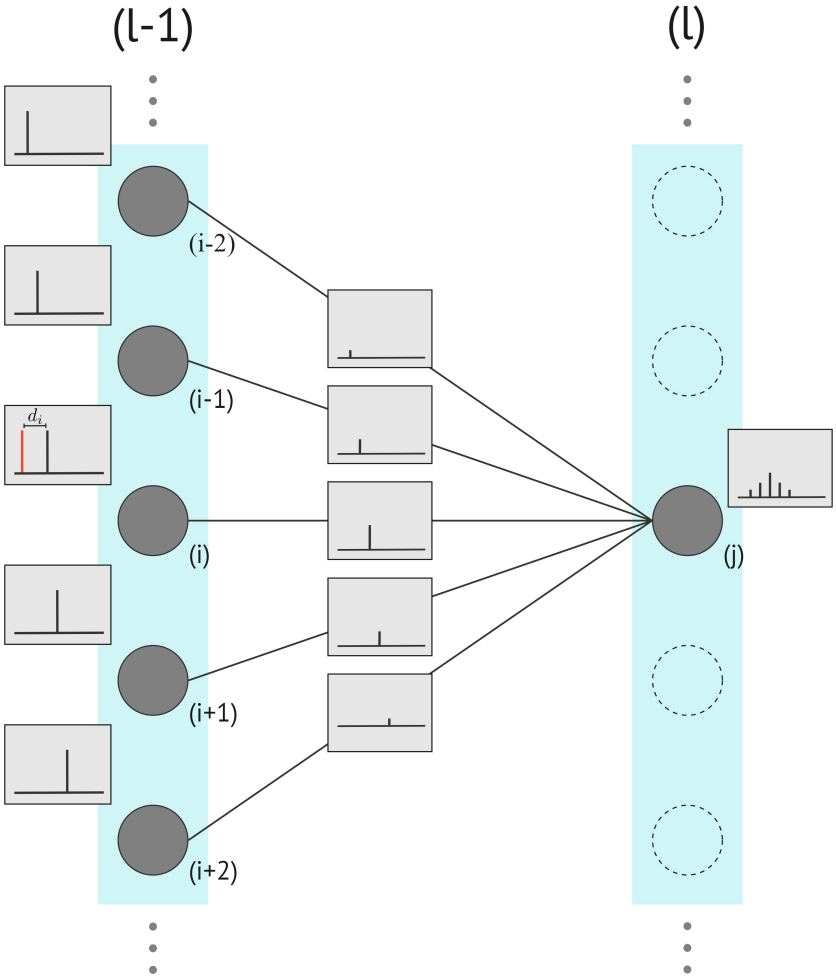

Figure 4.5: Example of the neuron duplication concept to estimate the Gaussian. A spike (red) occurs in neuron i in layer $(l-1)$. The spike is delayed by d_i and duplicated four times. Each new spike is moved to an additional neuron and postponed around the post-delay spike time of neuron i. After that, each spike is routed to the same neuron j in layer (l) through synapses. The synaptic weights are adjusted in a way that results in an approximation of a Gaussian, when the spikes arrive at neuron j.

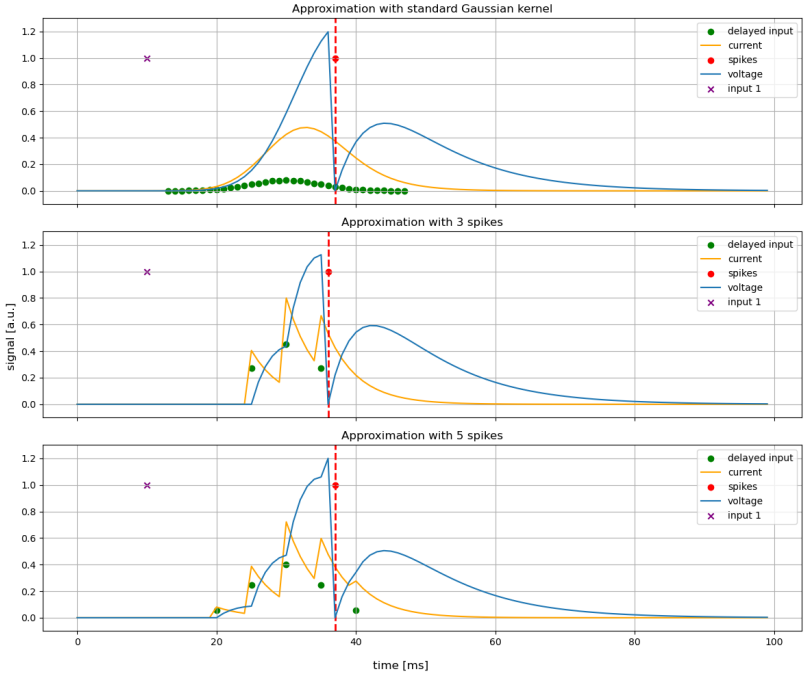

Figure 4.6: An input spike at $10\,\mathrm{ms}$ is being delayed with a Gaussian kernel of length 100 in the top plot. We try to recreate the same dynamics with 3 and 5 values in the middle and bottom plot, respectively. The spike in the top and bottom plot occurs at $37\,\mathrm{ms}$, and $1\,\mathrm{ms}$ earlier for the middle plot. The sum of the 'delayed input' is one in all plots.

```
class DelaySNN(nn.Module):
    def __init__(self):
        super().__init__()
        lif_params = F.CalibratedCUBALIFParams(...)
        self.exp = hxsnn.Experiment(mock=False, dt=1e-6)
        # Creating modules
        self.inp = DelayedInputNeuron(2, experiment=self.exp,
                  delay=[10,20], std=5, hw_dup=3)
        self.syn = Synapse(2, 3, experiment=self.exp)
        self.syn.weight.data = torch.tensor([[63,40],[0,0],[20,0]])
        self.lif = Neuron(
            3,
            experiment=self.exp
            func=F.cuba_lif_integration,
            params=lif_params,
            ...)
    def forward(self, input):
        x0 = hxtorch.snn.NeuronHandle(input)
        x1 = self.inp(x0)
        x2 = self.syn(x1)
        x3 = self.lif(x2)
        hxsnn.run(self.experiment,100)
        return x3
```

Listing 4.4: Code demonstration for the API for our `DelayedInputNeuron` class. Additionally to the arguments, also needed to create an `InputNeuron` object, `DelayedInputNeuron` has arguments which determine the delay values (`delay`), the standard deviation of the Gaussian (`std`), and the amount of spikes used to approximate the Gaussian (`hw_dup`).

like the delay value through new arguments when creating an instance. We do this by making `DelayedInputNeuron` a child class of `InputNeuron` and overwrite the modules' methods that are used to add it to the grenade graph. Inside our new class, we need to change how the input signal is added to the grenade graph, since this decides how the inputs are placed onto the BSS-2 hardware. In detail, we overwrite the `add_to_input_generator` method which takes in the input spikes as a handle, determines the spike events and appends them to grenade's input generator. In the standard `InputNeuron`, this is done by creating a list of all the spike times for each input. We now duplicate the list by the amount of spikes used to approximate the Gaussian and change the spike times inside the new list by multiples of σ. Listing 4.5 shows the code for our overwritten `add_to_input_generator` method. Listing 4.6 shows a comparison of the list object which is used to pass the input spikes to grenade. Since the amount of signals which are supposed to be forwarded onto the hardware changes through the implementations in `add_to_input_generator`, we also need to make sure the grenade population which transports these signals has the same increased

```
class DelayedInputNeuron(InputNeuron):
    def __init__(delay, std, hw_dup, ...):
        self.delays = delay
        self.sigma = std
        self.hw_dup =  hw_dup
        ...
    def add_to_input_generator(
            self, input: NeuronHandle, builder) -> None:
        # Initializing necessary values
        inp = input.spikes.clone()
        delays = self.delays.clone.astype(np.int32)
        sig = int(self.sigma)
        dt = self.experiment.dt / 1e-3
        # Extracting the spike times in the input signal to a list
        spike_times = tensor_to_spike_times(
            input.spikes, dt=dt)[0]
        # Duplicating spike times
        for i, synapse in enumerate(spike_times.copy()):
            spike_timest[i] = [k + delays[i]*dt for k in synapse]
            spike_times.append([k + (delays[i] - sig) * dt for k in
                synapse])
            spike_times.append([k + (delays[i] + sig) * dt for k in
                synapse])
            if self.hw_dup == 5:
                spike_times.append([k + (delays[i] - sig*2) * dt
                   for k in synapse])
                spike_times.append([k + (delays[i] + sig*2) * dt
                   for k in synapse])
        # Adding spike times to grenade
        builder.add(spike_times, self.descriptor)
```

Listing 4.5: Code for the delay and duplication of spikes. The `spike_times` list is extended by additional elements which correspond to spike times of additional input neurons. For this to work the standard deviation as well as the delay values need to be integers.

```
# spike_times list before duplication and delay
[[0.01, 0.02],
 [0.05]]
# spike_times list after duplication and delay
[[0.02, 0.03],
 [0.07],
 [0.018, 0.028], [0.022, 0.032],
 [0.068], [0.072]]
```

Listing 4.6: Example for the `spike_times` list before and after delay and duplication (`std` = 2, `delays` = (10, 20), `hw_dup` = 3).

shape. We therefore also overwrite the method `add_to_network_graph` and increase the size of the grenade population, which is added to the grenade graph:

```
gpopulation = grenade.network.ExternalSourcePopulation(
              self.size * self.hw_dup)
```

Since we now have increased the number of input signals, we need to add additional synapses to forward those signals into the network. We therefore need to change how the synapse is added to the grenade graph by modifying the `add_to_network_graph` method inside the `Synapse` class.

Every synapse module has a weight tensor, whose values decide over the connection strengths of each synaptic connection of the module. More importantly, the shape of this weight tensor also decides over the amount of synapses, meaning the amount of input neurons which this synapse module connects to on-chip neurons. The shape of the weight tensor normally is determined by the first two arguments, when creating the `Synapse` object, after which the weight values can be accessed and assigned through the `weight.data` attribute as can be seen in listing 4.4.

We now simply increase the shape of our weight tensor in order to match its input dimension with the amount of spikes from the `DelayedInputNeuron` module. We do not only accept the additional signals though, but also adjust their strengths to approximate the Gaussian shape. Furthermore, we do this in a way that the effect strength of one spike is distributed in time to the spikes used to approximate the Gaussian to ensure that the same amount of charge arrives at each post synaptic neuron. We implement this by making sure that the sum of the weight values of the post distribution synapses equals the weight of the original synapse w. Furthermore, the weight ratios of the post duplication synapses are the same as the ratio of values of a Gaussian distribution at multiples of the standard deviation σ. The formula to calculate the post distribution weights w_i is the following:

$$w_i = w \cdot \frac{\exp(-\frac{i^2}{2})}{\sum_{k=-n}^{n} \exp(-\frac{k^2}{2})}. \tag{4.2}$$

Each synapse is duplicated to $2n + 1$ to synapses, and n is the number of additonal weights needed for this. The index i declares which of the distributed weights is calculated with w_0 being the biggest weight (at the mean of the Guassian) and w_n the smallest (n multiples of σ away of the mean in the Gaussian). Weight values with $n > 2$ are too small to make an impact with this equation and use up hardware resources unnecessarily, so we did not implement them. To approximate the Gaussian with more than the maximum of 5 values which was implemented here, we would need to implement a different algorithm, but this was not further pursued in this work.

An example for this is shown in listing 4.7. As can be seen, the new input dimension (number of rows) is three times as big because each input is duplicated to 3 `DelayedInputNeurons`. The weight distribution follows the same concept as in listing 4.6 to match the correct weight values with delay values in order to approximate the Gaussian. The first column recreates the strongest weight values, since they belong to the center of the Gaussian and the precise time of the delay values. After that, the next 2 columns for a duplication to 3 spikes (4 columns for a duplication to 5 spikes) belong to the delay times, which are multiples of the standard deviation away from the center. We need to make sure that after the original synaptic weight is distributed onto the additional synapses, the weight of each synapse does not exceed 63, since this is the maximal value supported by the hardware. The maximum weight value before the weight distribution is 139.4229 for a duplication to 3 spikes and 156.4751 for a duplication to 5 spikes.
We integrate this into the standard synapse module of hxtorch, so it is important to only

```
# Weight tensor before weight distribution
[[139.4229, 40.0000],
 [  0.0000,  0.0000],
 [ 20.0000,  0.0000]])
# Weight tensor after weight distribution
[[63.0000, 18.0745, 38.2114, 38.2114, 10.9627, 10.9627],
 [ 0.0000,  0.0000,  0.0000,  0.0000,  0.0000,  0.0000],
 [ 9.0373,  0.0000,  5.4814,  5.4814,  0.0000,  0.0000]]
```

Listing 4.7: Weight tensor before and after weight distribution.

do this weight distribution if the module before the synapse is a `DelayedInputNeuron`. Since we can not access the presynaptic module inside the `add_to_network_graph` method in the standard hxtorch implementation of the synapse, we add it as an argument when calling the method while the experiment is being executed. Listing 4.8 demonstrates the code for the weight distribution inside the `Synapse` class.

Figure 4.7 demonstrates how the signal is delayed and duplicated after a `DelayedInputNeuron` and the dynamics inside the standard hxtorch neuron where the signal arrives.

4.2.3 Delay between Neuron Layers

In the previous section, we could access the spikes through the torch tensor, which was used as the input spike train. By modifying this tensor, we duplicated and delayed the spikes.
After the input is sent to the hardware, we can not access the spike train in any way, and only after the whole experiment is completed the spike information is moved back to the host computer in the form of a torch tensor. Therefore, we need a new approach to duplicate

```
def add_to_network_graph(pre_pop, ...):
    pre_count, post_count = self.weight.data.shape[1], selt.weight.
       data.shape[0]
    if isinstance(pre_pop, DelayedInputNeuron):
        if pre_pop.hw_dup == 3:
            # Make sure each synaptic weights do not exceed 63.
            weight_transformed = torch.min(self.weight.data.clone()
               , torch.tensor(139.4229)).T
            gaussian_extension = torch.zeros(
                                pre_count * 2, post_count)
            for i in range(pre_count):
                gaussian_extension[2 * i] = weight_transformed[i]
                gaussian_extension[2 * i + 1] = weight_transformed[
                   i]
            weight_transformed = (torch.cat((weight_transformed,
                                gaussian_extension * np.exp(-0.5)),
                                dim=0) / (np.exp(-0.5) * 2 + 1).T
        elif pre_pop.hw_dup == 5:
            # weight distribution to five values is analogous
            ...
        else:
            raise Exception("Only 3 and 5 supported")
    # After that, the synapse module is added to grenade
    ...
```

Listing 4.8: The code to increase the number of synapses and distribute the original weights onto them. We check if the presynaptic population `pre_pop` is an instance of the `DelayedInputNeuron` to determine if the weights need to be distributed. If this is the case we determine its `hw_dup` attribute to check how many values need to be added.

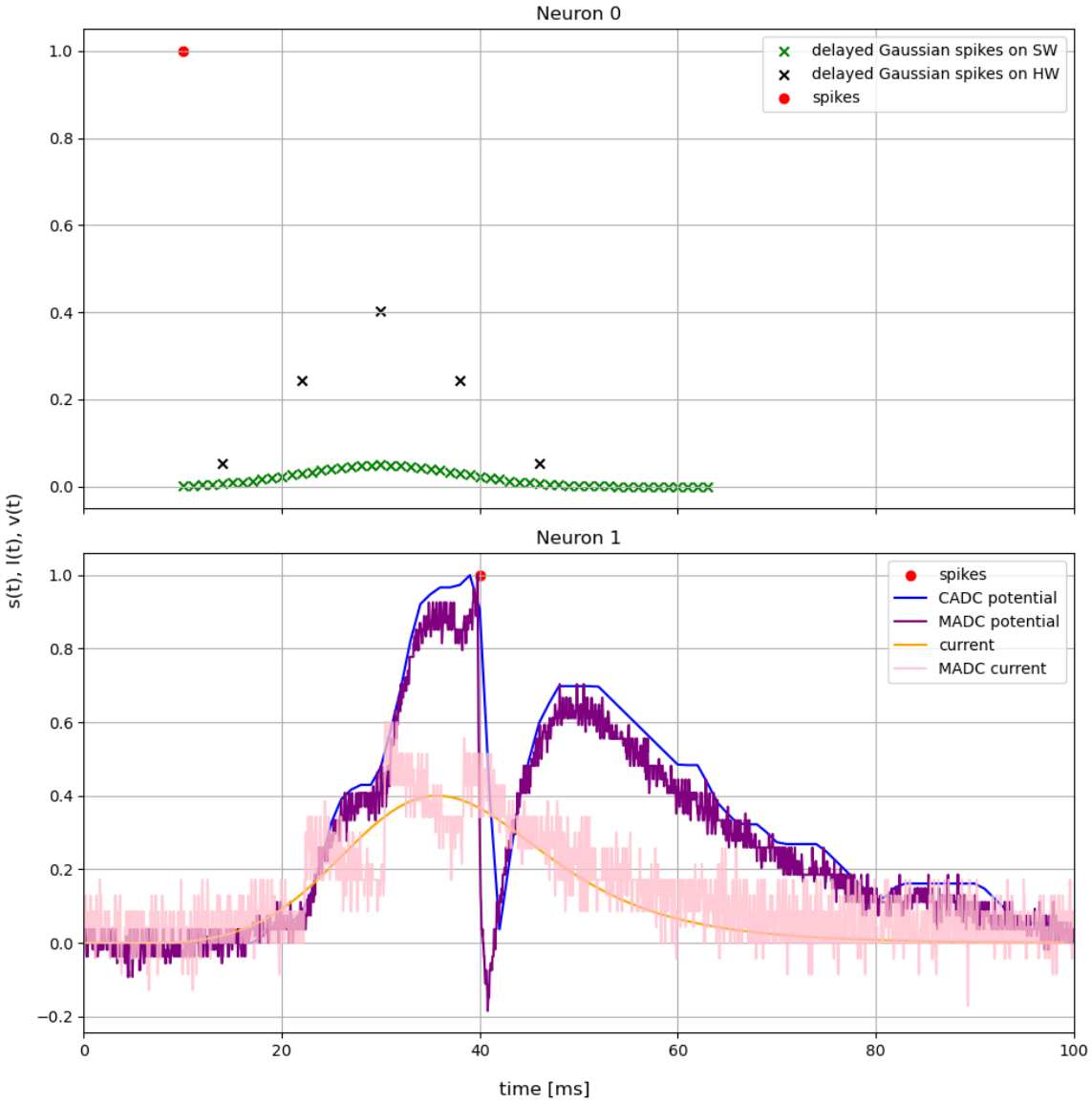

Figure 4.7: Dynamics inside `DelayedInputNeuron` and the hxtorch neuron where the duplicated and delayed spikes arrive. Both MADC and CADC measured potentials are plotted. In addition to the current in the numerical representation (yellow), the current on the hardware is shown (pink). We used a large value for the standard deviation of the Gaussian ($\sigma = 8\,\mathrm{ms}$) to emphasize the implications of the Gaussian approximation.

and delay the spikes when they are on the hardware.
When running an experiment with hxtorch every module is assigned to an execution instance. This is normally done automatically, but can also be done through an argument when creating the module. Only modules with the same execution instance can be executed in one hardware session without host-computer interaction. If the execution instance of a neuron is different to that of the subsequent synapse, it is needed to recreate the neuron, but with the same execution instance as the synapse, in order to further propagate the signal. To transport the spikes from the first execution instance to the second, we route each neuron from the first to one from the second. This routing is normally done one to one, copying the behavior of the first neuron module to the second.
This workaround can be used to route the signal from one neuron to multiple neurons, effectively duplicating the spike train, similarly as done in the previous section, where this was achieved by increasing the size of the input tensor. To approximate the Gaussian distribution of the spikes in time, we additionally need to be able to precisely delay the spike trains. We achieve this by making the projection from the neurons on the first execution instance to the neurons on the second not instantaneous but delayed. Since a controllable inter-execution-instance delay was not supported initially, it needed to first be implemented and tested.
The inter-execution-instance projection is done inside the synapse and realized through a connection array that indicates what neuron from the first execution instance is routed to which from the second. A comparison between the standard connection array and the altered version to duplicate and delay the spikes is shown in listing 4.9. The code to create and correctly handle this array is shown in listing 4.10.
The `DelayedNeuron` is a new child class of the standard neuron in hxtorch and is used to store the values relevant for the delayed Gaussian duplication as well as indicate if and how the connection array needs to be altered. It has the same numerical representation inside the `func` attribute as the `DelayedInputNeuron`. The synapses used to route the new neurons are also identical to the ones we implemented in section 4.2.2. Figure 4.7 shows a demonstration of the signal propagation through a network with a `DelayedInputNeuron` and a `DelayedNeuron`.

4.3 In-the-Loop Delay Learning

In the previous two chapters, we extended the hxtorch framework to support the duplication and delaying of input and inter-neuron spikes on BSS-2. This was done in order to approximate the behavior of the Gaussian distribution, used in the learning algorithm in section 4.1.3 and make it possible to move the forward pass on the hardware.
The code used for the in-the-loop delay learning is the same as used for the delay learning

```
# Standard connection array
[[0, 0, 0, 0],
 [1, 0, 1, 0],
 [2, 0, 2, 0]]

# Duplicated connection array with delays
[[0, 0, 0, 0, 2500],
 [1, 0, 1, 0, 5000],
 [2, 0, 2, 0, 11250],
 [0, 0, 3, 0, 1500],
 [0, 0, 4, 0, 3500],
 [1, 0, 5, 0, 4000],
 [1, 0, 6, 0, 6000],
 [2, 0, 7, 0, 10250],
 [2, 0, 8, 0, 12250]]
```

Listing 4.9: Comparison between standard and altered connection array. The first and third column indicate which neuron on the first execution instance is projected to which on the second. While this projection is one-to-one in the first array, in the second array each neuron is projected to three neurons, tripling the neurons, preparing the approximation of the Gaussian with three values. To additionally delay the projections a fifth column is added which indicates the delay time in FPGA clock cycle units. Since a ms in the accelerated emulated time of the system takes 125 clock cycles, the delay values therefore are $20\,\mathrm{ms}$ for the first, $40\,\mathrm{ms}$ for the second and $90\,\mathrm{ms}$ for the third neuron projection and the standard deviation is $8\,\mathrm{ms}$. The second and fourth column can be used to indicate the compartment in each neuron and are not needed here.

```
...
# Check if execution instances differ
if pre.toExecutionInstanceID() != post.toExecutionInstanceID():
    c_c = 125 # clock cycles per ms of emulated time
    # Check if previous population is DelayedNeuron
    if isinstance(pre_pop, DelayedNeuron):
        # Create input population for second execution instance
        # Size needs to be multiplied by duplication amount
        iei_pre = builder.add(grenade.network.
           ExternalSourcePopulation(
            pre_count*pre_poph.hw_dup), self.execution_instance.ID)
        # Change units in delay and standard deviation from ms to
           clock cylces
        delays = pre_pop.delays.detach().numpy()[0]*c_c
        sig = pre_pop.sigma*c_c
        # Creating connections array
        connections = np.zeros((pre_count*pre_pop.hw_dup,5))
        for i in range(pre_count):
            connections[i] = [i,0,i,0,max(delays[i],0)]
            for j in range(mul-1):
                connections[pre_count + (pre_pop.hw_dup)*i + j] = [
                   i,0,pre_count
                + (pre_pop.hw_dup-1)*i + j,0,max(delays[i]
                + (j // 2 + 1)*sig*(-1)**(j + 1),0)]
    else:
        iei_pre = builder.add(grenade.network.
           ExternalSourcePopulation(
            pre_count), self.execution_instance.ID)
        connections = np.array([[i, 0, i, 0, 0] for i in range(
           pre_count)])
    # Create projection and routing in grenade
    iei_projection = grenade.network.
       InterExecutionInstanceProjection()
    iei_projection.from_numpy(connections, pre, iei_pre)
    builder.add(iei_projection)
    # replace neuron on first execution instance
    pre = iei_pre
...
```

Listing 4.10: The code for the creation of the inter-execution-instance projection. We only want to alter the projection if the previous population is a `DelayedNeuron`. If that is the case, we create an input population with increased size according to our `hw_dup` attribute of the `DelayedNeuron` object. Afterward, we change the units in the delay and standard deviation to clock cycles and use them to create the connection array. In the end, the new routing is added to the grenade graph and the previous population is replaced with the one on the second execution instance.

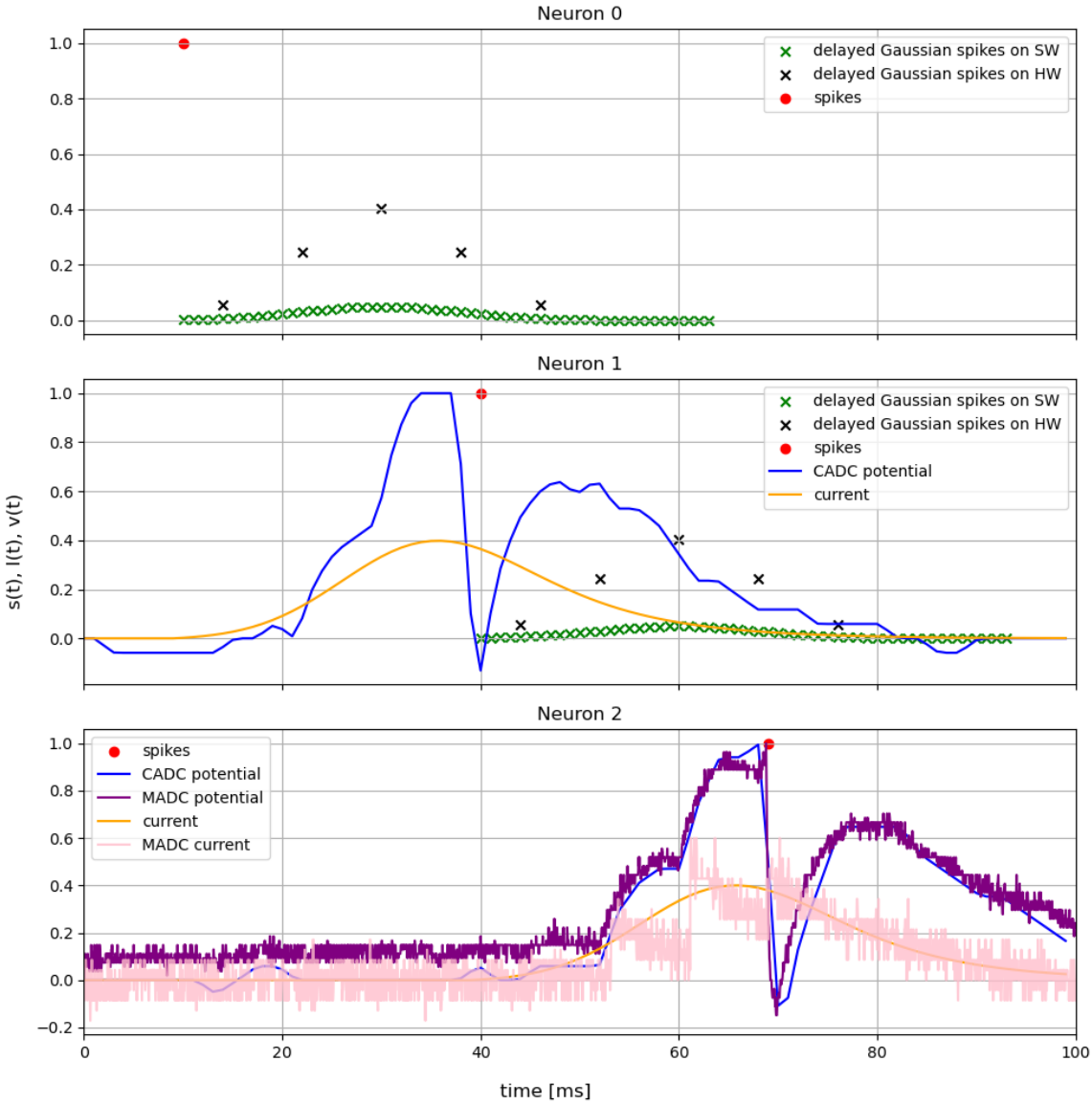

Figure 4.8: Dynamics inside a SNN with both new classes. The input spike arrives at 10 ms through a `DelayedInputNeuron` (neuron 0). It propagates through a `DelayedNeuron` (neuron 1) and then through a standard hxtorch neuron (neuron 2). The delay for both neuron 0 and 1 is 20 ms and the standard deviation value for the Gaussians is 8 ms. This demonstrates the successful implementation of axonal delays and the Gaussian approximation in hxtorch on BSS-2.

in PyTorch (listing 4.3), and also the loss function again is the squared difference between the spike arrival time and a target time. We use a model consisting of a `DelayedInputNeuron`, a `DelayedNeuron` and a normal output neuron to demonstrate the learning capabilities for both new classes. Figure 4.9 shows the simultaneous learning of the delay value in the `DelayedInputNeuron` (d_0) and the `DelayedNeuron` (d_1). Delay d_1 influences the loss more directly compared to d_0 because it is closer to the spike time calculation inside PyTorch's computational graph. Indeed, the gradient of d_1 is more than 4 orders of magnitude bigger than of d_0, and we had to use different learning rates to show the learning of both delay values simultaneously (learning rate for $d_0 = 4 \cdot 10^9$, learning rate for $d_1 = 1 \cdot 10^5$).

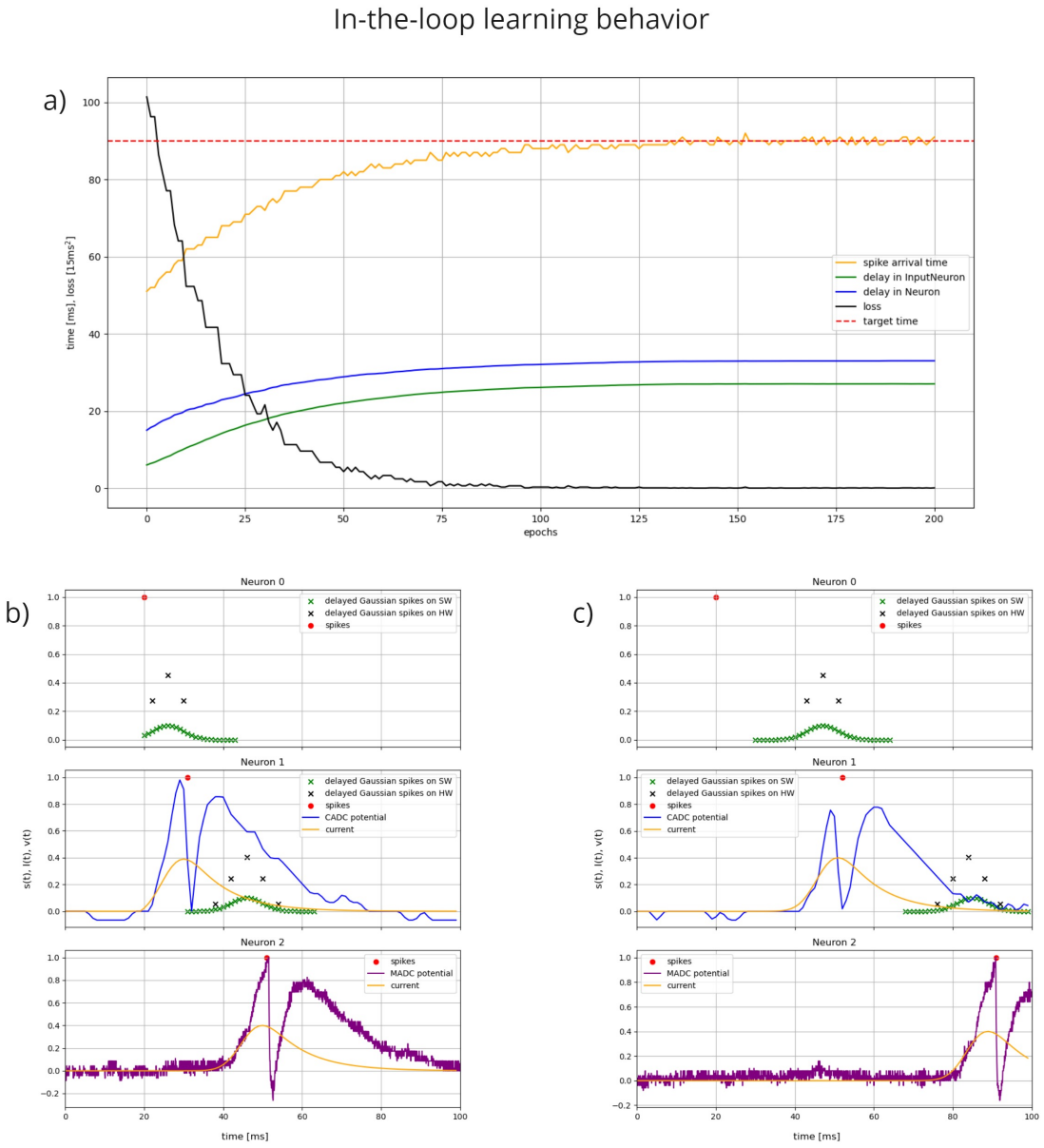

Figure 4.9: Demonstration of in-the-loop learning. A single spike enters the network after 20 ms through a `DelayedInputNeuron` (neuron 0), propagates through a `DelayedNeuron` (neuron 1) and then through a standard hxtorch neuron (neuron 2). The initial delay for neuron 0 and 1 is 6 ms and 15 ms, respectively. Only the spike time of neuron 2 is used for the learning with the target time being 90 ms. (a) Change of delay times, spike arrival times as well as loss value throughout the 200 epochs of the learning process. (b) Dynamic inside the networks' neurons during an experiment before the learning has happened. The spikes are only slightly delayed in each neuron, which results in a spike at 51 ms in neuron 2. (c) Dynamic inside the neurons after the learning process has finished. The learning caused the delays in neuron 0 and 1 to increase which results in a matching of target time and spike time in neuron 2.

5 Discussion

The goal of this work was to enable the training of delays in a BSS-2 software ecosystem. This was done through the implementation of a Gaussian convolution algorithm which delays the spikes and spreads them out in time in the shape of a Gaussian and therefore makes the gradient computation in the backward pass of the learning process possible. This algorithm showed promising results on conventional computers, but was not yet tested on neuromorphic hardware [9].

We started by recreating the algorithm in software with PyTorch to examine and test its behavior and learning capabilities. After that, we moved the forward pass of the learning process onto the BSS-2 system. We did this with the hxtorch framework, which integrates PyTorch with BSS-2. This was a major challenge since it is not possible to directly delay spikes on the BSS-2 chip and also the shape of the spike can not be altered to imitate the spread out spikes. We therefore decided to route the signal from the on-chip neuron back to the host computer, digitally delay it with the intended delay value d, and then send it back to a neuron on the hardware. This delays every spike that leaves a neuron by the same value and therefore implements axonal delays. To approximate the Gaussian additionally to delaying the spikes, we decided to duplicate each spike in the host computer, delay each duplicated spike, distributed around d, and send them over synapses to the same neuron i. The synaptic weights can be set in a way that results in an approximation of the Gaussian in neuron i. With this implementation, we were able to successfully learn delays on the BSS-2 hardware. The Gaussian shape is pivotal for the implementation of the algorithm, but our approximation of this on hardware was not the only possible option. For example, instead of changing the strength of the spikes by routing them through multiple synapses and changing the weight of those synapses, we could have used hardware resources, which are normally used to route the spikes, to change their strengths. This though would limit the options to route the spikes and also would not spread them out in time, like it is needed for the Gaussian approximation, and therefore a different approach would be needed to realize this. Another consideration was to not approximate the Gaussian at all on the hardware and only delay the spikes, suggesting that the difference between the hardware and software behavior can be neglected. We chose not to pursue this approach because maintaining a close similarity between the

software model and the hardware is a crucial factor in machine learning on external systems. However, in-the-loop learning could potentially enable this approach to work for delays.
Also the duplication method we used could have been implemented differently. We approximated the Gaussian with 3 or 5 spikes, with each spike having a time difference of the standard deviation value of the Gaussian, which is used for the backward pass. It is possible that these spikes are not sufficient or should be placed differently to minimize the difference between the behavior of backward and forward pass while not using up too many resources of the hardware.
Before this work, delays between two neurons on the BSS-2 hardware were not implemented. We integrated them in hxtorch into the rerouting of spikes through the host computer. While this was done to implement the Gaussian convolution algorithm, these delays can also be used to test different methods to learn delays in-the-loop.
The implemented changes are done in two new classes: The `DelayedInputNeuron` class is used to learn the delays of input spikes, and the `DelayedNeuron` class is used to learn delays between two neuron layers. These classes are seamlessly integrated in the hxtorch framework and can be used by users to build up on my thesis by using them to create a bigger neural network and train the delay values on a dataset like SHD [6] or YinYang [12]. The algorithm can then be benchmarked against existing algorithms on these datasets. Furthermore, also the exponential decrease of the standard deviation of the Gaussian throughout the learning could be compared to a static standard deviation value, since this resulted in improved performance in [9]. If the in-the-loop delay learning shows promising results in the benchmark, it can be fine-tuned and optimized by testing other implementation for the Gaussian approximation. Further work could make the duplication more generic to easily test many ways to duplicate the spikes and use a parameter sweep to decide which configuration works the best.
SNNs have a strongly increased amount of parameters compared to conventional neural networks. To be able to fully leverage this advantage with neuromorphic hardware, it is needed to be able to train all the parameters. Synaptic weight learning on BSS-2 is already implemented and our work adds the possibility to learn delays. Future work could implement methods to learn the AdEx parameters, as well as multi-compartment morphologies to make the whole parameter space of BSS-2 learnable. This can then be used to assess and test how much each parameter contributes to the network's performance, which is especially important to decide how to improve and design optimized neuromorphic hardware substrates.

Acknowledgements (Danksagung)

Der Weg vom Anfang meines Praktikums bis zur Fertigstellung dieser Bachelorarbeit war eine sehr prägende Zeit für mich. Es hat mich in allen Teilen immer wieder erstaunt, überrascht, begeistert, gefordert und überfordert. Es gab zahlreiche Menschen, die mich beim Meistern dieser Herausforderung immer wieder unterstützt haben.
Zuallererst will ich mich bei meinem Mentor Elias bedanken, dass er mir, obwohl ich keinerlei Erfahrung in Dingen wie Git, Vim, PyTorch oder mit den Grundlagen des neuromorphen Computings hatte, trotzdem mit viel Geduld Schritt für Schritt bei allem geholfen hat und mir immer mit Rat und Tat zur Seite stand. Du hast viel von mir erwartet und hast mich dadurch dazu gebracht, viel zu leisten.
Außerdem will ich Philipp, Jakob, Eric und Elias danken für das Korrekturlesen meiner Bachelorarbeit und dass ihr mir mit eurer tiefgreifenden Expertise zu den Themen und zum Verfassen von wissenschaftlichen Publikaten zu einer runderen und präziseren Arbeit verholfen habt.
Mein Dank geht auch an meinen Lieblingsbüronachbarn Florian, mit dem die Arbeitsatmosphäre immer entspannt und humorvoll war.
Ich will mich bei der ganzen Electronic Vision(s) Gruppe bedanken und bei jeder der zahlreichen netten und hilfsbereiten Personen, die ich in der Zeit kennengelernt habe.
Zudem geht mein Dank an meine Eltern Dorothea und Udo und meine Geschwister Sophie und Charlotte, die immer für mich da sind, wenn ich sie brauche. Danke an Charlotte für das Korrigieren von zahlreichen Grammatikfehlern. Zuletzt will ich Senian danken, dass sie mir Stärke gibt und immer an mich geglaubt hat.

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).

Bibliography

[1] Luca Blessing. “Gradient Estimation With Sparse Observations for Analog Neuromorphic Hardware”. Masterarbeit. Heidelberg University, 2023.

[2] Jeffrey S. Bowers. “Parallel Distributed Processing Theory in the Age of Deep Networks”. In: *Trends in Cognitive Sciences* 21.12 (2017), pp. 950–961. ISSN: 1364-6613. DOI: 10.1016/j.tics.2017.09.013. URL: https://doi.org/10.1016/j.tics.2017.09.013.

[3] 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.

[4] Benjamin Cramer et al. “Surrogate gradients for analog neuromorphic computing”. In: *Proceedings of the National Academy of Sciences* 119.4 (2022). DOI: 10.1073/pnas.2109194119.

[5] Benjamin Cramer et al. “Surrogate gradients for analog neuromorphic computing”. In: *Proceedings of the National Academy of Sciences* 119.4 (2022), e2109194119. DOI: 10.1073/pnas.2109194119. eprint: https://www.pnas.org/doi/pdf/10.1073/pnas.2109194119. URL: https://www.pnas.org/doi/abs/10.1073/pnas.2109194119.

[6] Benjamin Cramer et al. “The Heidelberg spiking datasets for the systematic evaluation of spiking neural networks”. In: *CoRR* abs/1910.07407 (2019). arXiv: 1910.07407. URL: http://arxiv.org/abs/1910.07407.

[7] Wulfram Gerstner et al. *Neuronal Dynamics: From Single Neurons to Networks and Models of Cognition*. Cambridge University Press, 2014.

[8] Wenzhe Guo et al. “Neural Coding in Spiking Neural Networks: A Comparative Study for Robust Neuromorphic Systems”. In: *Frontiers in Neuroscience* 15 (2021). ISSN: 1662-453X. DOI: 10.3389/fnins.2021.638474. URL: https://www.frontiersin.org/journals/neuroscience/articles/10.3389/fnins.2021.638474.

[9] Ilyass Hammouamri, Ismail Khalfaoui-Hassani, and Timothée Masquelier. “Learning Delays in Spiking Neural Networks using Dilated Convolutions with Learnable Spac-

ings". In: *The Twelfth International Conference on Learning Representations*. 2024. URL: https://openreview.net/forum?id=4r2ybzJnmN.

[10] A. L. Hodgkin and A. F. Huxley. "A quantitative description of membrane current and its application to conduction and excitation in nerve". In: *The Journal of Physiology* 116.3 (1951), pp. 499–544. DOI: https://doi.org/9.1113/jphysiol.1952.sp004764. eprint: https://physoc.onlinelibrary.wiley.com/doi/pdf/9.1113/jphysiol.1952.sp004764. URL: https://physoc.onlinelibrary.wiley.com/doi/abs/9.1113/jphysiol.1952.sp004764.

[11] Jakob Kaiser et al. "Emulating dendritic computing paradigms on analog neuromorphic hardware". In: *Neuroscience* 489 (2022), pp. 290–300. ISSN: 0306-4522. DOI: 10.1016/j.neuroscience.2021.08.013. URL: https://www.sciencedirect.com/science/article/pii/S0306452221004218.

[12] Laura Kriener, Julian Göltz, and Mihai A. Petrovici. "The Yin-Yang dataset". In: *arXiv* (2021). URL: https://arxiv.org/abs/2102.08211.

[13] Henry Markram, Wulfram Gerstner, and Per Jesper Sjöström. "Spike-Timing-Dependent Plasticity: A Comprehensive Overview". In: *Frontiers in Synaptic Neuroscience* 4 (2012). ISSN: 1663-3563. DOI: 10.3389/fnsyn.2012.00002. URL: https://www.frontiersin.org/journals/synaptic-neuroscience/articles/10.3389/fnsyn.2012.00002.

[14] Nestor Maslej et al. *Artificial Intelligence Index Report 2024*. 2024. arXiv: 2405.19522 [cs.AI]. URL: https://aiindex.stanford.edu/wp-content/uploads/2024/05/HAI_AI-Index-Report-2024.pdf.

[15] Christian Paul Mauch. "Operating Accelerated Neuromorphic Hardware - A Scalable and Sustainable Approach". PhD thesis. Universität Heidelberg, 2021.

[16] 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.

[17] Adam Paszke et al. "PyTorch: An Imperative Style, High-Performance Deep Learning Library". In: *Advances in Neural Information Processing Systems 32*. Curran Associates, Inc., 2019, pp. 8024–8035. URL: http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf.

[18] Christian Pehle et al. "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.

[19] Samyam Rajbhandari et al. "ZeRO-infinity: breaking the GPU memory wall for extreme scale deep learning". In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. SC '21. St. Louis, Missouri: Association for Computing Machinery, 2021. ISBN: 9781450384421. DOI: 10.1145/3458817.3476205. URL: https://doi.org/10.1145/3458817.3476205.

[20] Santiago Ramón y Cajal. "Estructura de los centros nerviosos de las aves". In: *Revista Trimestral de Histología Normal y Patológica* 1 (1888). Early work using the Golgi method, pp. 1–10.

[21] Philipp Spilger et al. *hxtorch.snn: Machine-learning-inspired Spiking Neural Network Modeling on BrainScaleS-2*. 2022. arXiv: 2212.12210 [cs.NE].

[22] Nishil Talati et al. "mMPU—A Real Processing-in-Memory Architecture to Combat the von Neumann Bottleneck". In: *Applications of Emerging Memory Technology: Beyond Storage*. Ed. by Manan Suri. Singapore: Springer Singapore, 2020, pp. 191–213. ISBN: 978-981-13-8379-3. DOI: 10.1007/978-981-13-8379-3_8. URL: https://doi.org/10.1007/978-981-13-8379-3_8.

[23] Pete Warden. "Speech Commands: A Dataset for Limited-Vocabulary Speech Recognition". In: *CoRR* abs/1804.03209 (2018). arXiv: 1804.03209. URL: http://arxiv.org/abs/1804.03209.

Acronyms

AdEx Adaptive Exponential Integrate and Fire. 11, 40

AI Artificial Intelligence. 1

ANN Artificial Neural Network. 1, 4

API Application Programming Interface. 24, 27

BSS-2 BrainScaleS-2. 2, 11, 12, 16, 21, 24, 27, 33, 36, 39, 40

CADC Columnar Analog to Digital Converter. 11, 12, 32

CPU Central Processing Unit. 1

DAC Digital to Analog Converter. 11

FPGA Field-Programmable Gate Array. 34

GPU Graphical Processing Unit. 1

LIF Leaky Integrate-and-Fire. 3, 5, 11, 12, 14, 16, 19, 22

MADC Membrane Analog to Digital Converter. 11, 12, 32

ML Machine Learning. 1, 2, 4, 8, 12

SHD Spiking Heidelberg Digits. 2, 40

SNN Spiking Neural Network. 1–4, 6, 7, 16, 19, 21, 24, 36, 40

Erklärung:

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

Heidelberg, 05. September 2024 _____

(Unterschrift)