Department of Physics and Astronomy

University of Heidelberg

Bachelor thesis

in Physics

submitted by

Ben Luca Kroehs

born in Berlin

2025

An Intermediate Data Format for In-the-loop Training on Neuromorphic Hardware

This Bachelor thesis has been carried out by

Ben Luca Kroehs

at the

Kirchhoff Institute for Physics

under the supervision of

Prof. Dr. Johannes Schemmel

Ein intermediäres Datenformat für In-the-Loop-Training auf neuromorpher Hardware

Intermediäre Formate, sowohl für Netzwerktopologien als auch für Daten in gepulsten neuronalen Netzen (SNNs) sind entscheidend für die Interoperabilität zwischen Simulatoren und die Austauschbarkeit neuromorpher Hardware-Plattformen. In dieser Arbeit wird ein Datenaustauschformat vorgestellt, das die Neuromorphic Intermediate Representation (NIR) erweitert, indem es die Infrastruktur zur Darstellung und Transformation von Spike-Daten auf Netzwerkknotenebene bereitstellt. Das Format unterstützt sowohl event-basierte als auch zeitdiskretisierte Repräsentationen von Daten und ermöglicht die Konvertierung zwischen beiden.
Um die praktische Relevanz des Formats zu demonstrieren, wurde eine In-the-Loop-Trainingsroutine implementiert, die die Software-Frameworks hxtorch und jaxsnn mit der neuromorphen Hardware BrainScaleS-2 kombiniert. Der Vorwärtspfad wird auf der Hardware ausgeführt, während der Rückwärtspfad in der Simulation unter Verwendung der Hardware-Spikes berechnet wird. Die Trainingsleistung wird als Funktion der zeitlichen Auflösung dt bewertet und zeigt eine starke Abhängigkeit in der Simulation und eine auflösungsinvariante Genauigkeit auf der Hardware jenseits eines bestimmten Schwellenwertes.
Im Vergleich zu existierenden Standards wie ONNX oder PyNN bietet das vorgeschlagene Format eine leichtere und anpassungsfähigere Lösung, die auf Anwendungsfälle des maschinellen Lernens von gepulsten neuronalen Netzen zugeschnitten ist. Es erleichtert simulator-unabhängiges Training und vereinfacht den Zugang zu neuromorphen Substraten.

An Intermediate Data Format for In-the-loop Training on Neuromorphic Hardware

Intermediate representations for both network topologies and observable data in spiking neural networks are crucial for achieving interoperability between simulators and enabling the interchangeability of neuromorphic hardware platforms. This thesis introduces a data exchange format that extends the Neuromorphic Intermediate Representation (NIR) by providing the infrastructure to represent and transform node-wise spike data. The format supports both event-based and time-discretized representations of data and allows conversion between them.
To demonstrate its practical relevance, an in-the-loop training routine is implemented that combines the hxtorch and jaxsnn software frameworks with the BrainScaleS-2 neuromorphic hardware. The forward pass is executed on hardware, while the backward pass is computed in simulation using the hardware spikes. The training performance is evaluated as a function of the temporal resolution dt, showing a strong dependency in simulation and a resolution-invariant accuracy on hardware beyond a certain threshold.
In comparison to existing standards such as ONNX or PyNN, the proposed format offers a more lightweight and adaptable solution tailored for machine learning use cases of spiking neural networks. It facilitates framework-independent training pipelines and simplifies access to neuromorphic substrates.

Contents

1 Introduction

During the decades, various approaches have emerged to address how data is processed and computed, ranging from optimizing conventional methods to exploring entirely novel paradigms. There are many different aspects that can be taken into consideration. On the one hand a major goal is to increase the efficiency of existing algorithms concerning the time and energy needed, since energy consumption of computation is rather high [1]. At the same time, visionary algorithms and methods are being explored to solve the same problems through fundamentally new approaches, like event-based computation [2] or low precision numerics [3].

During the end of the last century, machine learning has emerged to be one of these visionary approaches, though being around since the 19th century [4]. From the the early 2000s on, the popularity has been increasing exponentially, benefiting many other fields [5, 6, 7]. One example is the translation tool DeepL, which outperforms conventional statistical and rule-based models [8].

Nowadays, machine learning as a tool has been adopted in a wide range of domains, itself containing several subfields. A key concept of machine learning is to define computational models and train their parameters by defining a loss with respect to a performance metric to be optimized and using algorithms like gradient descent. Artificial neural networks (ANNs) show a basic concept of machine learning which imitate the brain, being models consisting of abstracted neurons and synapses. There are different approaches which are oriented to biology in varying degrees. While ANNs encode the information in numbers, spiking neural networks (SNNs) encode the information in events over time, also known as spikes, much like the brain. This event-based nature promises to be more energy efficient than conventional computing. As a result, different approaches to optimize these models are emerging and in order to determine the most effective ones, a comparison between these methods is necessary.

The training and inference of these SNNs in software [9, 10, 11, 12, 13] or on dedicated neuromorphic chips [14, 15, 16, 17] is one aspect where these approaches might differ. To train SNNs on neuromorphic hardware, a common approach is so-called hardware in-the-loop (ITL) training [18], in which a simulator interacts with the hardware by receiving the model's inference results and computing the corresponding parameter updates. However it is also possible to train the model in software and port it to hardware for inference. This might possibly perform not as well because it does not take special properties of the hardware into account that might alter the inference.

For hardware ITL training there is typically one simulator framework that is tailored for

the special configurations of the hardware platform. To enlarge the accessibility of such neuromorphic hardware it is desired to enable training and inference of SNNs that are defined in other simulators. This can be simplified by introducing an intermediate representation for network topologies, as this simplifies the implementation of network conversions between different simulators.

There are several approaches for the intermediate representations of network topologies between software frameworks, like ONNX [19] or PyNN [20], where ONNX is solely an exchange format for ANNs and PyNN can also transform topologies of SNNs influenced by neuroscience. Both define a set of nodes that are commonly used in ANNs or SNNs, which form an intermediate representation. A recent development that aims in the same direction is the neuromorphic intermediate representation (NIR) [21], which also defines nodes but focuses on SNNs from a machine-learning perspective.

But additionally to the exchange of topologies, data must be translated between the software frameworks for ITL training. Also the comparison of methods concerning SNNs, as it is done in benchmarks, is simplified when the input data can be converted between the different software frameworks.

In this thesis, a data exchange format is introduced to simplify implementing approaches like hardware ITL training with arbitrary software frameworks. It takes its cue from NIR – a data exchange format for topologies of SNNs. As a first step, the current definition of data formats for observables in software frameworks for SNNs is investigated and a conversion between these formats is implemented. Further, this implementation is then validated in an demonstrator example for ITL training with the BrainScaleS-2 (BSS-2) hardware platform and also with simulation. Finally the performance of the conversion is evaluated.

2 Theoretical Background

2.1 Spiking Neural Networks

During the last years, machine learning as a tool found its way into various fields of application. A basic concept of machine learning are neural networks (NNs) – computational models which are inspired by the behavior observed in the brain. Therefore, they implement a computational graph, consisting of "neurons" connected by "synapses", where the neurons are nodes where the main computation happens and the synapses forward the information and adjust the strength. Artificial neural networks (ANNs) work such that a neuron sums over its input values x_i from incoming synapses and a bias term b, applies an activation function ϕ and returns the output y.

$$y_j = \phi\left(\sum_{i=1}^{n} w_{ji}x_i + b_j\right) \tag{2.1}$$

The synapses can be displayed by a 2-dimensional matrix of weights, where each weight w_{ij} corresponds to the factor by which the output of neuron j is scaled by arriving at neuron i.

In ANNs, the input data is processed in a static, feedforward manner, lacking an inherent time dimension. Spiking neural networks (SNNs) are an extension of ANNs, that have a time dimension by introducing neurons with a state that evolves over time. Their characteristic is that they implement a time dimension in which the data is encoded as discrete events in time, also called spikes. Similar to NNs, the neurons are connected via synapses.

There, an often used spiking neuron model is the leaky-integrate and fire (LIF) neuron [22], where the neuron state consists of a membrane voltage V and the current on the membrane I. For each incoming event, a certain amount of charge is deposited on the membrane, resulting in a membrane voltage V which is decaying over time to the resting potential V_{rest}. If the membrane voltage crosses a threshold θ, the neuron sends out a spike which is transmitted to all connected subsequent neurons and the membrane voltage V is reset to the set potential V_{reset},

$$\text{if } V > \theta : V \to V_{\text{reset}}. \tag{2.2}$$

If there is no spike, the dynamics of the neuron are described by the following differential equation:

$$\tau_{\text{mem}}\frac{dV}{dt} + V = V_{\text{rest}} + \frac{I}{g_l}. \tag{2.3}$$

Here, g_l describes the leak conductance and τ_{mem} the membrane time constant. For an incoming spike from a neuron i, the synaptic input current I is raised by the value of the corresponding weight:

$$I \to I + w_{ji} e_i, \tag{2.4}$$

where e_i is a unitary vector with an `1` at i-th position. Within the LIF model, the synaptic current is modeled as discrete spike events.

The LIF model can be extended by modeling the current I which is decaying over time:

$$\tau_{\text{syn}} \frac{dI}{dt} + I = 0. \tag{2.5}$$

This formula describes current-based synapses with exponential decay, which are used in the current-based leaky-integrate and fire (CubaLIF) neuron.

2.1.1 Grid-based Computation in Spiking Neural Networks

In the several software frameworks like hxtorch [23], Norse [11], or snnTorch [12], which implement SNNs, the spike data is stored in a time-gridded fashion. This means that a temporal resolution dt is introduced and thus the time axis is divided in a set number of time steps. The data, such as the outgoing spikes of a specific neuron, are now represented by a vector which has one entry for every time step. If the neuron spikes at this certain time step, the vector has the value `1` at this position, else it is `0`. A whole batch of spike data for one layer of neurons is represented by a 3-dimensional tensor. Software frameworks now implement their numerical solver of the LIF dynamic equations to find the spike times. Consequently, the numerical solvers used in simulator frameworks operate on this discretized time axis to simulate the dynamics of spiking neurons, such as those described by the LIF model.

The main disadvantage of this approach is concerning the time resolution dt. If the exact timing of a spike is crucial, e.g. for one's learning algorithm, the temporal resolution should be as small as possible. But with smaller temporal resolution also the tensor holding the data gets larger and thus it takes longer to solve the differential equation 2.3. This shows a trade-off between temporal resolution vs. memory and experiment runtime.

2.1.2 Event-based Computation in Spiking Neural Networks

The counterpart of the time-gridded approach is the event-based data representation. The spikes of a neuron are represented by the time stamp at when the spike is emitted as well as some identification key of the neuron that emits the spike. By that, the accuracy of the spike time is no longer limited by a time resolution dt. This representation can especially show advantages for sparse data, e.g. if there is only one spike per neuron: While in the event-based approach there would only be n tuples of spike time and index, where n denotes the number of neurons,

the grid-based data representation would show a tensor consisting of mainly zeros. The size would be dependent on the temporal resolution.

2.2 Description of SNNs in Software and the Implementation of Algorithms for Training

Spiking neural networks but also neural networks in general perform different according to their parameters. In a process called "training" the parameters are varied such that the performance is maximized. Therefore the performance has to be defined in a formula. A common approach is to define a loss, e.g. the mean-squared-error, between a value that is predicted by the model and the real value.

In software frameworks the forward pass is commonly defined by a `model` instance (lst. 1), which can be executed with input data, that matches the model's topology in terms of shape and data format, see section 2.1.1 f.. While computing the forward pass, the model keeps track of the computations that happened. This information is necessary to perform the gradient calculations in the backward pass.

```
model = ...
optimizer = ...

for e in range(n_epochs):
    input_batches, target_batches = dataset(seed=..)
    for i in range(n_batches):
        input_batch = input_batches[i]
        target_batch = target_batches[i]

        output_batch = model(input_batch)  # forward pass

        loss = loss_func(output_batch, target_batch)
        loss.backward()  # backward pass

        optimizer.step()
```

Listing 1: Exemplary training routine of an ANN. After defining the model and a optimizer for training, for each batch of each epoch the model is trained on the dataset by performing the forward pass of the network and then performing backpropagation on the loss.

To achieve a good performance, the most common approach is to use a set number of samples drawn from a dataset split into batches. The model is then applied on all batches iteratively, while the model's parameters are updated after each batch. These processes happen during one

so-called epoch. After one epoch has finished, the dataset is potentially shuffled and the next epoch begins. After several epochs and assuming that the conditions are good enough and the learning rate is decreasing, the accuracy of the model converges to a certain value and reaches a (local) maximum.

2.3 Gradient-based Learning for Spiking Neural Networks

To obtain an optimized set of parameters, a commonly used algorithm is backpropagation. This is achieved by differentiating the loss with respect to the parameters. In a basic case, the gradient descent algorithm would be used to compute an update for the respective parameters. The gradients are scaled with a step size η, which determines how strong the parameters are altered per iteration. In a working training setup, the parameters would converge to a certain value, which denotes a local minimum of the loss. For large step sizes, the gradient descent algorithm can become unstable and thus the parameters would not converge. But there are methods to make the gradient descent algorithms more stable, like the Adam optimizer [24], which uses a floating mean of past gradients (momentum) and is based on stochastic gradient descent.

However spiking neural networks introduce discrete jumps in the dynamics of neurons (eq. (2.2)). With the equations not being differentiable, backpropagation cannot be used for gradient estimation. There are different approaches to handle this issue. The first one is using a differentiable approximation for the threshold function, like a sigmoid function (eq. (2.6)), in the backward pass:

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

Furthermore the computation is performed on a discrete time grid, limiting the resolution of the spike times. This method is called surrogate gradients and was derived by Neftci, Mostafa, and Zenke [25]. This does not alter the forward pass, but the gradients are no longer exact. Nevertheless, this makes it possible to perform backpropagation through time (BPTT).

Wunderlich and Pehle [26] proposed EventProp, an algorithm, where exact gradients are computed using the adjoint method. In the event-based backward pass, the adjoint variables are evolved backwards in time and then sampled at the spike times to calculate the gradients.

2.4 In-the-loop Training

To use neuromorphic hardware like the BrainScaleS-2 (BSS-2) chip, special training routines have to be used, because models trained in software are expected to perform worse on analog hardware, which cannot be simulated perfectly, due to manufacturing defects or fixed-pattern noise. Therefore, backpropagation cannot be performed on the physical dynamics of the neurons

on-chip. To address this problem, Schmitt et al. [27] developed the concept of hardware in-the-loop (ITL) training (fig. 2.1), an approach where the forward pass is performed on hardware and the recording is then used in the backward pass in a software model to compute the gradients and the model parameter updates, instead of evolving the observables in a simulation. This software model only has to be an approximation of the hardware platform since the forward pass is performed on hardware. During this training, several data exchanges are necessary: Every batch has to be transformed from the software framework to hardware and back.

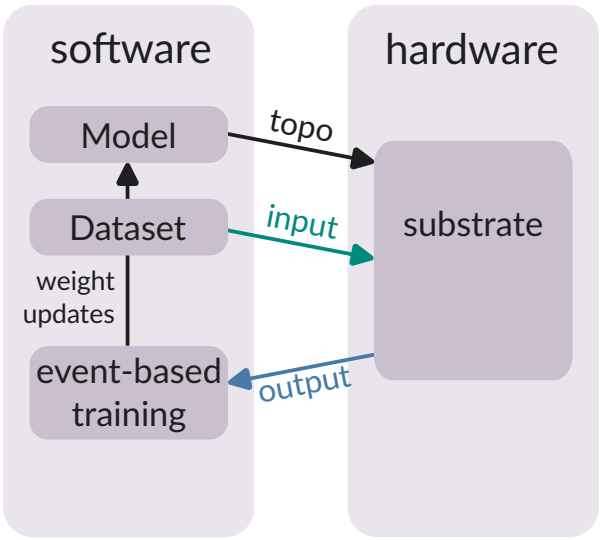


Figure 2.1: Flow graph for ITL training. The model as well as the dataset are defined in the main software framework (left). To perform the inference on the hardware platform, the network topology as well as the batch of input data has to be translated to it. The resulting output data is then transformed back and used for gradient computation and updating the weights.

2.5 BrainScaleS-2 – A Mixed-signal Neuromorphic Hardware

The way in which information is processed by spiking neural networks differs from that of conventional von-Neumann architectures. While these conventional computers have a separation between the storage and the CPU, neuromorphic hardware does not show this separation – just like the brain. Due to the architectural similarity to the executed model, efficiency benefits are expected.

Thus, it is useful to investigate the field of neuromorphic hardware, that is tailored to fit the requirements and needs of spiking neural networks. There are analog as well as digital approaches concerning the type of signal and data processing on such hardware: On digital chips, the single neurons are simulated such that the equations of the desired neuron model are solved in a numerical way. Also digital hardware in general offers advantages such as reduced sensitivity to noise, since calculation inaccuracies are the only significant source of noise. Therefore this

approach is often followed, e.g. by the SpiNNaker architecture using of SpiNNaker-2 chips to emulate spiking neural networks.

In contrast to this, analog hardware implements circuits that emulate the neuron dynamics physically, which aim to be more energy efficient. On analog substrates, the dynamics are more vulnerable to noise on the signal. The BrainScaleS-2 chip [17] is one example for a mixed-signal neuromorphic platform, meaning that there are both, digital and analog parts on the substrate. This chip implements 512 adaptive exponential integrate-and-fire (AdEx) neurons [28], whose parameters, like the threshold potential or the time constants, can be configured and also the topology of the network is flexible. These neurons and the synapses are emulated in an analog fashion, but the processing of the event signals as well as the chip configuration happens digitally. Typical time constants for parameterization of the neuron circuits on the BSS-2 hardware are in the range of microseconds, what facilitates a speed-up factor of 1000 in comparison to biological time constants.

These chips are currently accessible via field-programmable gate arrays (FPGAs), which execute so-called playback programs, describing the experiment protocol in physical time.

The spikes are represented as events, while their time resolution is 8ns – corresponding to the FPGA's 125MHz clock. When measuring the membrane potential of the analog neurons, an analog-to-digital converter (ADC) is needed. The BrainScaleS-2 chip contains two different kinds of ADCs: a slow columnar ADC (CADC) and a faster and thus more precise membrane ADC (MADC). The CADCs are only faster when all neurons have to be read out because they can each read out 256 columns of one row of neurons at parallel, while the MADC can only read out two neurons at one time. Another important information is that the synaptic weights are supplied as 6-bit digital values, while the sign is adjustable for each row of synapses.

This has to be accounted for when deploying a model from software to hardware.

2.6 The YinYang Dataset and Time-To-First-Spike Encoding

The Yin-Yang dataset [29] has been introduced as a task which can be solved by rather small network topologies (30 hidden neurons), though achieving high accuracies (97.6%). These are reasons why it is convenient to be used for prototyping network architectures, new algorithms or substrates. The dataset describes points in a circle, that are categorized in three classes, "Yin", "Yang" and "Dots" (fig. 2.2), where each class occurs equally frequently.

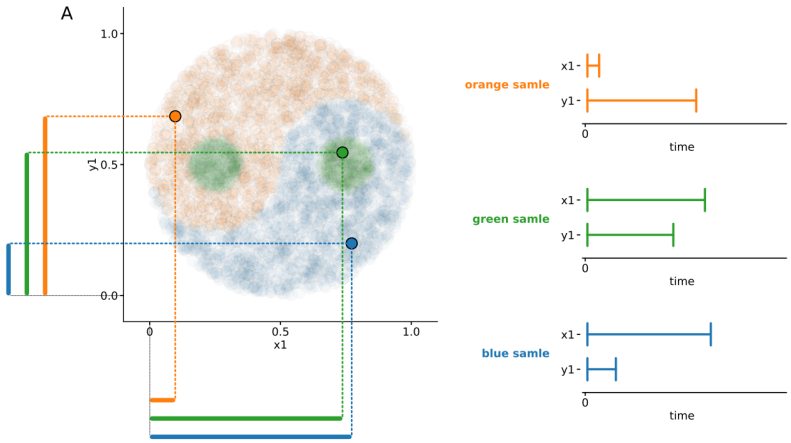


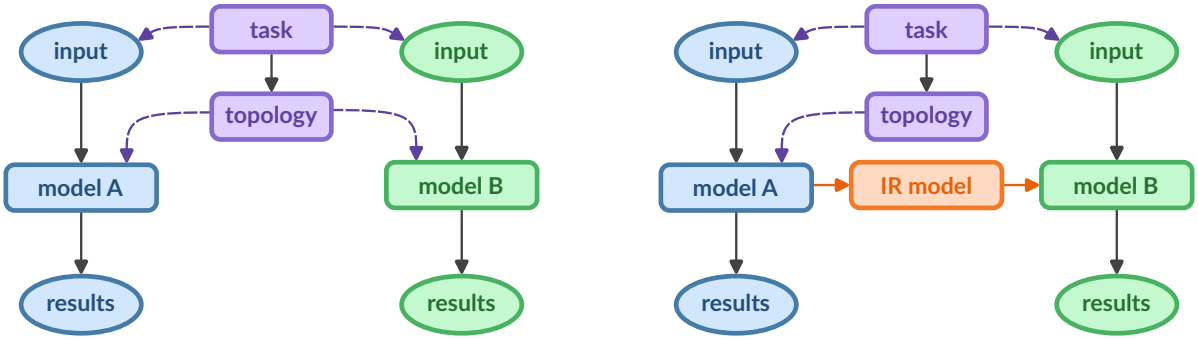
Figure 2.2: Visualization of YinYang dataset and the encoding of coordinates into spike times. Taken from Kriener, Göltz, and Petrovici [29].

For spiking neural networks, each sample is encoded into four input spikes plus one bias spike, that has the same timing for all samples. The spike times of the other four input spikes depend on the sample's coordinates. The coordinates x and y are encoded into spike times such that two spikes correspond to the x and y value while the other two are corresponding to $1-x$ and $1-y$ for symmetry reasons. Each of these four values is then mapped to the range between t_{early} to t_{late}.

Typically, either a non-spiking layer built of leaky-integrator (LI) neurons or a spiking output layer of LIF neurons are used [29, 30]. The spiking output layer requires decoding of the output spikes. Therefore, target spike times are defined for the three output neurons. A possible way is the time-to-first-spike decoding: Here, the loss operates on the difference between the time of the first spike of a neuron and its target time and the classification is based on the first spike occurring across the output neurons.

3 Existing Software

In the previous years, a variety of neuromorphic frameworks with different approaches for network representation came up [23, 11, 12, 13]. Since these software frameworks differ in their implementation, also the numerics might be different. For reasons of comparability it is advantageous to have the possibility to translate the implementation of a spiking neural network from one framework to another. In cases where different implementations are to be compared, the same model description is written down in both frameworks (fig. 3.1a). But with an intermediate representation for model descriptions of SNNs, the topology can be translated from one framework to another, thus erasing the need to write it down twice (fig. 3.1b).



(a) Processing pipeline without an intermediate representation

(b) Processing pipeline with an intermediate representation (IR)

Figure 3.1: Process of applying a task on a software framework by implementing (purple arrows) an encoding of the data into spikes and an implementation of a topology that has previously been the subject of consideration on the basis of the task. The circles describe data while the rectangles describe a model for spiking neural networks. The different colors refer to implementations within different software frameworks. With an intermediate representation (right), the implementation of the topology in one framework suffices and the second model is obtained by conversion via this intermediate representation (orange arrows).

For the case of (hardware) in-the-loop training, where the forward pass is executed on an external software framework (or on a hardware platform), the framework has to support the so-called "Unterjubel" functionality (Ger.: to slip something to someone unnoticed). "Unterjubel" describes the functionality to calculate the gradients of the weights according to the loss and update the weights without performing the forward pass by itself but with externally acquired observables by performing the inference on an external framework. Examples of frameworks

which are currently supporting this are the event-based simulation and training framework jaxsnn [13] as well as the grid-based simulation and training network hxtorch [23]. Both frameworks support the use of the BrainScaleS-2 hardware platform.

3.1 jaxsnn

jaxsnn is an event-driven software framework [13], designed to match with the event-driven nature of neuromorphic substrates just like BSS-2 [17]. It is built upon JAX [31], a library that is built on top of NumPy and designed especially to support high performance computing. This is achieved by delivering key features like just-in-time (JIT) compilation and also efficient and convenient function transformation, improving flexibility. Similar as in PyTorch [32], JAX supports defining a custom backward function and autograd functionality for gradient computation. A new feature is that when generating the backward pass, the forward pass does not have to be performed.

jaxsnn follows an init-apply-approach: The `apply` function implements the network's forward pass and also implicitly the backward pass while the `init` function generates an initial weight setup for the synapse layers. To create deep SNNs, individual `init` and `apply` functions of each layer of neurons are composed in an init-apply-pair, representing the whole network. Currently, jaxsnn does not explicitly represent synapse layers[1]. The `apply` function also depends on the used neuron type. Currently, only CubaLIF neurons are supported in jaxsnn, since this is also a neuron type which is implemented on BrainScaleS-2. The CubaLIF neuron in jaxsnn is implemented in different variations, as detailed in the following:

`LIF` This is the basic CubaLIF neuron. For the forward pass, the neuron dynamics are evolved from spike to spike, while these spikes are found by a analytical formula [30], which only holds for $\tau_{\text{mem}} = \tau_{\text{syn}}$ and $\tau_{\text{mem}} = 2\tau_{\text{syn}}$.

`EventPropLIF` The EventProp version uses the same forward pass but applies the EventProp algorithm for the computation of the backward pass.

`RecurrentLIF` This one allows for layer-wise recurrence by representing the input and the recurrent connection in a single weight matrix, allowing projecting the neuron layer's spikes onto itself.

`RecurrentEventPropLIF` There is also a recurrent version of the `EventPropLIF`, which enables the definition of recurrent network topologies.

`HardwareLIF` If the forward pass should be external, the neuron needs a "Unterjubel" functionality. This is provided by the `HardwareLIF` neuron.

[1] It is planned to implement an explicit representation of synapse layers in jaxsnn [33].

Its backward pass is implemented using the EventProp algorithm.

`HardwareRecurrentLIF` This is a `HardwareLIF` neuron, which can be used for recurrent networks.

In contrast to a time-gridded data representation as it is used in many PyTorch-based neuromorphic frameworks, in jaxsnn every spike is described by a tuple of the time it occurred and the index of the neuron. This tuple is called `Spike`. If the user wants to use the EventProp algorithm for training, also the neurons state, consisting of membrane voltage and current, needs to be known at the spike times. The membrane voltage directly before the spike is the threshold ϑ and is reset to the reset potential V_{reset} directly after the spike. Therefore only the current has to be provided additionally for every spike, what is represented in a class called `EventPropSpike`.

3.2 NIR

The idea of the neuromorphic intermediate representation (NIR) [21] is an approach to implement a simple and universal representation for SNNs in the form of a `NIRGraph`. In addition, its implementation is very close to other machine learning frameworks, which simplifies the integration. Different neuromorphic frameworks can now implement a conversion from their representation of the networks topology to a `NIRGraph` as well as the opposite direction, coming from the `NIRGraph`. An exemplary `NIRGraph` is implemented in lst. 2.

The `NIRGraph` is represented by a dictionary consisting of the graph's nodes plus a list of the edges between the nodes. In advance of this thesis, NIR defines a set of 16 primitive nodes, such as a `Linear` node or a `LIF` node. For convenient graph handling the primitives also include two dummy nodes `Input` and `Output`, that contain the input and output shape of the network. The input and output shape of the remaining nodes are defined implicitly via their attributes: For the `Linear` node the size is determined by the shape of the weight matrix; the shape of the neuron parameters, e.g. `threshold`, specifies the `LIF`'s size.

The intermediate representation is also limited to a description of the networks topology and does not include a representation of e.g. spike data. In addition, NIR itself does not implement the numerics of the nodes but is a pure representation for abstract topologies of spiking neural networks.

Typically, frameworks for SNNs implement a `to_nir` and a `from_nir` function, which represent the interface to NIR. The `to_nir` function takes a graph in the representation of the respective framework that is converted to a `NIRGraph`. Analogously, the `from_nir` function takes a `NIRGraph` and returns the model in the frameworks representation. In some software frameworks, also a conversion configuration has to be supplied. This holds parameters that are not represented in the current version of NIR.

```
one_layer_graph = nir.NIRGraph(
    nodes = {
        "input" : nir.Input(input_type=np.array(5*[1])),
        "linear1" : nir.Linear(weight=...),
        "lif" : nir.LIF(tau=np.array(20*[1]),
                        r=np.array(20*[1]),
                        v_leak=np.array(20*[0]),
                        v_threshold=np.array(20*[1])),
        "linear2" : nir.Linear(weight=...),
        "output" : nir.Output(output_type=np.array(3*[1]))
    },
    edges = [
        ("input", "linear1"),
        ("linear1", "lif"),
        ("lif", "linear2"),
        ("linear2", "output")
    ]
)
```

Listing 2: Example of a `NIRGraph` with 5 input neuron, 20 hidden neurons and 3 output neurons.

3.2.1 NIRTorch

Since many neuromorphic frameworks are based on PyTorch, a helper module NIRTorch [34] is provided as part of NIR. Its aim is to facilitate the implementation of a conversion between PyTorch-based neuromorphic frameworks and NIR. Frameworks like Norse [11] or snnTorch [12] are already using this module. The conversion is implemented such that the nodes are converted iteratively, according to a network graph, which is generated by NIRTorch. Each node is then transformed according to a framework-specific conversion function.

3.3 hxtorch

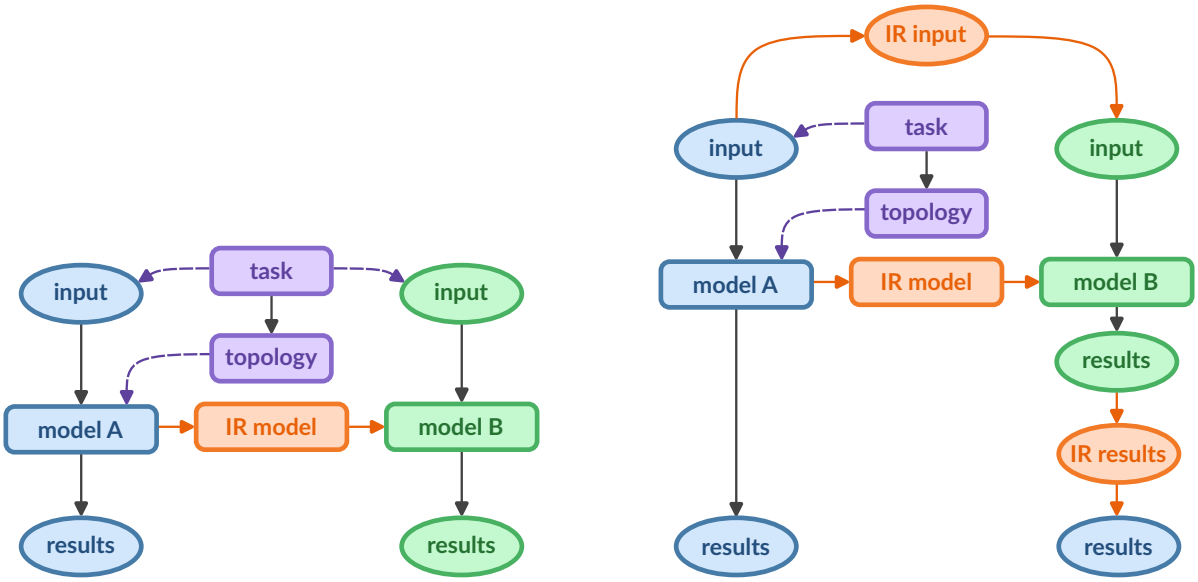
The PyTorch-based neuromorphic simulation and training framework which supports the mixed-signal substrate BSS-2 is hxtorch [23]. This framework supports hardware in-the-loop (ITL) training, meaning that the forward pass is executed on the neuromorphic hardware. The recorded spikes are then used in hxtorch ("Unterjubel") to perform the backward pass by calculating the gradients, e.g. via back-propagation. hxtorch implements different PyTorch modules, like a leaky integrate-and-fire neuron (`Neuron`), a leaky integrator (`ReadoutNeuron`), a synapse (`Synapse`) and also a dummy input neuron (`InputNeuron`). These PyTorch modules

are especially designed to use the BrainScaleS-2 chip. This is shown e.g. by the ability to set which of the analog-digital-converter should record the membrane trace or if the membrane is to be recorded at all. In addition, hxtorch has a `mock` mode, where the neuron traces are simulated in software. This mode is especially useful for testing algorithms or training routines before deploying them on neuromorphic hardware.

Every module in hxtorch has an attribute that takes a `Experiment` object, which represents a experiment in software or hardware. It also contains the information if the experiment is run on hardware or not and also the whole network topology including the associated modules.

4 Implementation

As shown in fig. 3.1, the workflow for deploying the same experiment on different software frameworks is simplified by the introduction of an intermediate representation for topologies, as it eliminates the need to implement the topology twice, once for each framework. In the following, NIR is used as an exchange format for topologies of SNNs. Additionally, the data has to be encoded and transformed to both software frameworks. This can be avoided by introducing a data exchange format, as it is shown in fig. 4.1. With this, spike data between different software frameworks can be transformed easily, no matter if the data is represented grid-based as a `torch.Tensor` or if it is a event-based `jax.Array`.



(a) Processing pipeline without an intermediate data representation

(b) Processing pipeline with an intermediate data representation

Figure 4.1: Process of applying a task on a software framework by implementing (purple arrows) an encoding of the data into spikes and implementation of a topology that has previously been the subject of consideration on the basis of the task. The circles describe data while the rectangles describe a model for spiking neural networks. With NIR the model has only to be implemented in framework A (blue) and the data exchange format enables transforming the data from framework A to framework B (green). Also the output data of both models can be compared easier since they can be transformed between the different software frameworks. Orange arrows show the conversions, which are performed with the NIR interface

Also this format should be oriented on the `NIRGraph`, so that the spike data is organized as a dictionary that connects multiple `NIRNode` objects and the corresponding input or output spikes. It also necessary that the data provided by the software framework is arranged in a dictionary such that the keys correspond to the `NIRGraph`.

```
# transformation: software framework (sf) -> NIR
nir_data = sf.to_nir_data(sf_data, sf_model)

# transformation: NIR -> software framework (sf)
sf_data = sf.from_nir_data(nir_data, sf_model)
```

Listing 3: Conversion interface for conversion of dictionaries of spike data between an arbitrary software framework and NIR. Both conversion functions take the corresponding dictionary of spike data.

The software interface in NIR ought to be one function call per conversion to or from the data exchange format to retain simplicity (lst. 3). An arbitrary software framework has to implement a `from_nir_data` function that converts a `NIRGraphData` object into the frameworks native data representation as well as a `to_nir_data` function. Both functions take just two arguments: the dictionary of data and the model in the software framework's native representation.

4.1 Intermediate Data Exchange Format

This thesis proposes extending the neuromorphic intermediate representation (NIR), which is currently an exchange format for topologies of SNNs, by an intermediate data representation.

4.1.1 TimeGriddedData and EventData

The data representation should be as general as possible – just like NIR itself: It must be very flexible and therefore have the ability to convert effortlessly between event data and time-gridded data. To provide this, the data format must be able to contain time-gridded data as well as event data. For the case that a framework requests the data to be time-gridded but it was delivered as event data and vice versa, there should be an internal conversion between both of the formats.

Therefore, two classes, `EventData` and `TimeGriddedData` that each hold the spike data for a single batch are implemented. Both classes have a function that returns the spike data converted to the respective other class.

In case of the `EventData`, an orientation is the `Spike` class in jaxsnn. `EventData` has two attributes, `time` and `idx`, each being a 2-dimensional `numpy.ndarray` of shape `(batch_size,`

`n_spikes)`, where `n_spikes` is the number of spikes which are allowed per layer. The `time`-array holds the time of the occurring spikes while the `idx`-array contains the indices of the corresponding neurons. If the number of spikes that are allowed per layer `n_spikes` exceeds the number of spikes that really happened, empty events are added, denoted by a `-1` as `idx` at the corresponding place.

```
event_data = EventData(idx_array, time_array)
```

For the `TimeGriddedData` the implementation in Norse or hxtorch is referred. Here the spikes for one batch are stored in a 3-dimensional `numpy.ndarray` of shape `(batch_size, n_time_steps, n_neurons)`. This array is by default filled with zeros. If a spike happens for a specific neuron in a time step, this is denoted by a `1` at the corresponding position along the time dimension in the array. Also the data class contains the temporal resolution *dt* as attribute to interpret the time-gridded data correctly:

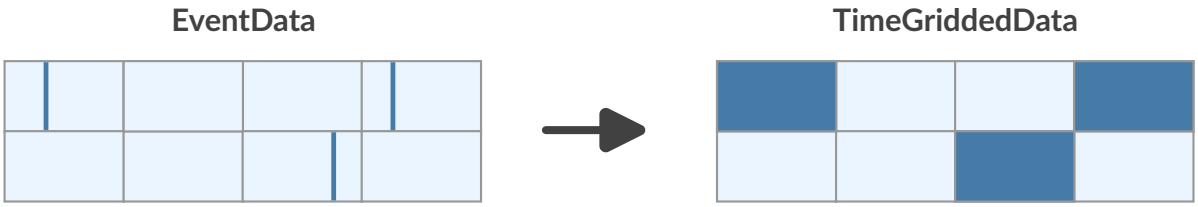
```
time_gridded_data = TimeGriddedData(data, dt=5e-5)
```

Each of both classes also contain a function implementing the conversion to the other data format, meeting the demands of different software frameworks connected via NIR. This reduces the amount of effort that is necessary for a software framework to implement the conversion to the data exchange format and back.

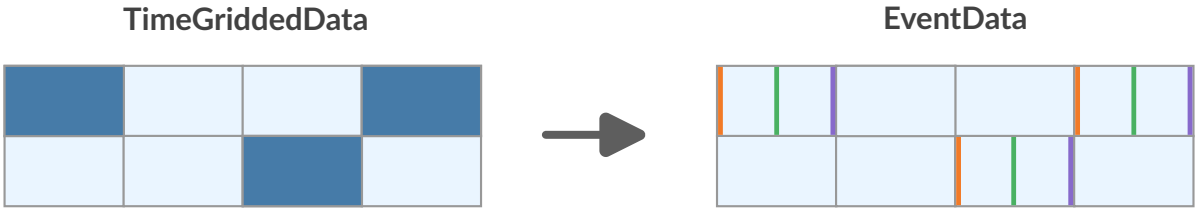
The translation from `EventData` to `TimeGriddedData` is implemented straight-forward by assigning each event-based spike the corresponding time step on the grid (fig. 4.2a). For this conversion one additionally has to provide the number of neurons as well as the number of time steps and the temporal resolution, since these parameters are needed in the grid-based format:

```
time_gridded_data = event_data.to_time_gridded(n_neurons=50,
                                               n_time_steps=100,
                                               dt=5e-5)
```

The opposite direction, converting grid-based data to event-based data is not as trivial. Since the number of spikes is not easily known, the maximum number of spikes has to be provided as an argument to get the shape of the event-based data format. But for a spike occurring in on certain time step it is not known at which discrete point it time the threshold was exceeded (fig. 4.2b). Some implementations in neuromorphic software frameworks might differ in the way of numerically solving the neurons dynamics. Thus it is useful to implement the conversion from `TimeGriddedData` to `EventData` such that the user can choose, where the spike should be located on the time axis.



(a) Conversion from `EventData` to `TimeGriddedData`



(b) Conversion from `TimeGriddedData` to `EventData`

Figure 4.2: Example showcasing the conversion between `EventData` and `TimeGriddedData`. While the conversion from the event-based format to the time-gridded format is ambiguous, the way vice versa is not clear. The introduced parameter `time_shift` resolves this ambiguity in the translation from `TimeGriddedData` to `EventData`. This is illustrated by three exemplary spikes (`time_shift = 0.0, 0.5 dt, dt`) in one time step.

The `time_shift` parameter, introduced in the `time_gridded_to_event` conversion, describes the temporal offset from the start of a time step, enabling more flexibility for the implementation of data format conversions:

```
event_data = time_gridded_data.to_event(n_spikes=100, time_shift=0.5*dt)
```

The range of this parameter is limited from `0` to dt ensuring the event-based spike stays in the corresponding time step. For most cases this offset will either be `0` (fig. 4.2b, orange spike) or dt (purple spike), but also a value like $0.5 \cdot dt$ (green spike) could be desirable.

Currently it is not supported to change the temporal resolution dt when converting from the data exchange format to a software framework. If the temporal resolution has to be changed, the data could first be translated to an `EventData` object and then back to a `TimeGriddedData` object.

4.1.2 From NIRNodeData to NIRGraphData

In an SNN, every node, e.g. a LIF layer, comes with a set of observables. Some of these are value-less observables and just contain a time stamp of an event, like spike times. Other observables like the membrane trace are valued data, because there is a value for each time stamp. Also for different training approaches, the set of necessary observables varies. The

proposal is to have a `NIRNodeData` object, which contains a dictionary of observables for a specific `NIRNode`. Currently, the conversion only supports one entry, `"spikes"`, which can either be `TimeGriddedData` or `EventData`, but this would be a possible interface for future extensions. This `NIRNodeData` object also has an attribute `t_max`, which denotes the maximum simulation time. This is necessary for the conversion from `EventData` to `TimeGriddedData`, to ensure that the resulting time-gridded arrays do not differ in size along the time axis. This would happen if the last spike time would be taken as `t_max`.

```
lif_data = NIRNodeData(
                observables = {
                    "spikes": event_data,
                    # "membrane": not_yet_implemented
                },
                t_max = ..
            )
```

Listing 4: Exemplary use of a `NIRNodeData` object, containing the event-based data of a LIF layer. As of now, only spikes are supported as observables.

A possible application of this data exchange can now be the transformation of recorded spikes on a hardware platform back to a software framework for training. For example, one could be interested in the spikes of the output layer but also of a hidden layer. The different instances of `NIRNodeData` are organized in a dictionary `nodes` with the corresponding node key, according to the `NIRGraph`. The `nodes` dictionary is the only member of the `NIRGraphData` class, which finally represents one batch of data for the desired nodes (fig. 4.3 and lst. 5).

```
graph_data = NIRGraphData(
                nodes = {
                    "lif": lif_data,
                    ..
                }
            )
```

Listing 5: Exemplary use of a `NIRGraphData` object, containing a `NIRNodeData` object `lif_data` to the node with key `"lif"`.

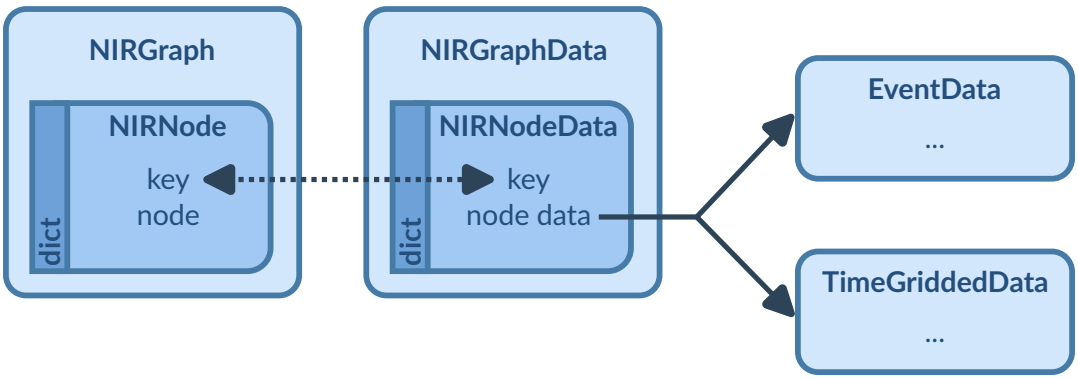


Figure 4.3: Class diagram of the data exchange format. The books display dictionaries. The keys of the dictionary which contains the `NIRNodeData` objects coincide with the keys in the `nodes` dictionary of the corresponding `NIRGraph`, retaining interoperability between NIR and the data exchange format. The data is stored in a `NIRNodeData` object, which is currently restricted to spike data, thus being either of type `EventData` or `TimeGriddedData`.

Similar to the topology conversion, the different software frameworks now have to implement the conversion from their data format to the data exchange format and back. It is expected that a software framework converts to the type of data that is more suitable, such that a grid-based framework like Norse also uses the `TimeGriddedData` class for the conversion to NIR. For the opposite direction, coming from the data exchange format, the framework should first check of which type the spike data is and then use the internal conversion function if the other format is desired.

4.2 Conversion between jaxsnn and NIR

In previous work by the author [35] the conversion of network topologies from NIR to jaxsnn has been implemented for the set of nodes which are currently supported by jaxsnn. This was done by an iterative approach translating node-by-node. Also a separate `ConversionConfig` object had to be implemented to provide additional attributes that are not represented in NIR such as the maximum simulation time $t_{\texttt{max}}$, which in jaxsnn is required for constructing the neuron layers. At the moment, the set of subclasses of `NIRNode` that is supported for translation to jaxsnn is limited, as it only includes the `CubaLIF` and `Linear`. See lst. 6 for the conversion interface from NIR to jaxsnn.

This configuration is now being extended to also support use cases where the forward pass is not performed by jaxsnn itself but on an external framework like hxtorch using the BSS-2 hardware. Therefore, a boolean argument `external` is added, which determines which layer type to use. For external spikes, the neuron type is changed to a version which does not perform

the forward pass itself but receives the output spikes from the inference platform.

```
nir_model = nir.NIRGraph(...)

jaxsnn_cfg = jaxsnn.ConversionConfig(v_reset = ..,
                                     n_spikes = {'layer_1': .., 'layer_2': ..}
                                     t_max = ..,
                                     external = ..)
jaxsnn_model = jaxsnn.from_nir(nir_model, jaxsnn_cfg)
```

Listing 6: Conversion of a `NIRGraph` to jaxsnn; `v_reset`: reset potential of neuron layers, `n_spikes`: dictionary of neuron layers that contains the maximum number of spikes allowed per layer, `t_max`: maximum simulation time of the experiment, `external`: indication if the forward pass is processed on an external framework. The `jaxsnn_model` consists of of the `init` and the `apply` function.

In jaxsnn the spike data is represented in an event-based fashion. See lst. 7 or a depiction of the interface. Thus the translation from jaxsnn to the data exchange format simply extracts the indices and spike time from the `Spike` object and then injects them in the new `EventData` object. This conversion expects the data coming from jaxsnn to be an `EventPropSpike` with global indexing of neurons. This global index determines the layer the neuron belongs to. The opposite direction for the conversion takes a `NIRGraphData` object and transforms it to a `Spike`.

```
# transformation jaxsnn -> NIR
nir_data = jaxsnn.to_nir_data(jaxsnn_data, jaxsnn_model)

# transformation NIR -> jaxsnn
jaxsnn_data = jaxsnn.from_nir_data(nir_data, jaxsnn_model)
```

Listing 7: Conversion interface for conversion of spike data between jaxsnn and NIR. The `to_nir_data` conversion takes the spike data as `EventPropSpike` and the neuron model in jaxsnn to infer which spikes belong to which layer. Analogously, `from_nir_data` takes a `NIRGraphData` object and converts it to an `EventPropSpike`.

4.3 Conversion between hxtorch and NIR

Similar to the conversion between jaxsnn and NIR, hxtorch, a software framework with access to the BSS-2 hardware, is to be connected to NIR. This involves the conversion of network topologies on the one hand and on the other hand the data conversion to the new data exchange

format.

4.3.1 NIRGraph Conversion

Since hxtorch is based on PyTorch, the first approach is using NIRTorch for the translation of topologies between hxtorch and NIR. The part of the NIRTorch module that converts a node from a PyTorch framework to NIR or vice versa is very intuitive and without any constraints, but the challenge is to extract the graph information, meaning which nodes are connected via synapses. This part already contains various edge cases for the different software frameworks using NIRTorch, what makes it less practical to use. And in addition, the `Experiment` instance introduced in hxtorch (Sec. 3.3) would have to be passed through the conversion since it is an attribute for all hxtorch nodes. To avoid these difficulties during implementation, NIRTorch is just referenced for the implementation for the general parts but the graph building is performed in hxtorch itself.

Since a graph of the network topology is already needed in hxtorch for the mapping on hardware, this existing functionality is used to generate the network graph. Graph means in this case that for every node, the output node is stored. This sufficiently describes the network topology. This graph is processed node-by-node to build the `NIRGraph` and the `NIRNode` objects it consists of. The topology conversion at this moment supports the hxtorch modules `Neuron`, `ReadoutNeuron`, `Synapse` and `InputNeuron`, which are translated to `CubaLIF`, `CubaLI`, `Linear` and `Input` – subclasses of `NIRNode` – and vice versa.

For the opposite direction from NIR to hxtorch, a custom hxtorch module `SNN` is constructed, which describes the whole network. The nodes are transformed iteratively and are represented in a dictionary `hxnodes`, which also contains the node keys. The forward pass is constructed by iterating through the edges and connecting the output of a node to the input of a following node, while the input and output of the whole model are handled separately. The construction of the module `SNN` takes a `NIRGraph` and also a `ConversionConfig`, similar to the implementation in jaxsnn, which is needed to supply attributes that are not represented is the `NIRGraph` (lst. 8). This includes some parameters that complete the description of the network state by the `NIRGraph`, like the temporal resolution `dt`, which is important for solving the dynamics in software and as measuring grid of hardware, or the reset potential `v_reset` of the spiking neurons. Furthermore, the `torch.device` can be specified, which is the device the computation for model simulation is executed on, e.g. the CPU or the GPU. In addition, a set of hardware parameters has to be supplied. This includes for example the scaling of membrane traces and weights from the software model to the model that is executed on the neuromorphic chip. This

is due to limited ranges of e.g. weights and traces on the chip.

```
# define topology in NIR
nir_model = nir.NIRGraph(...)

# transform NIR topology to hxtorch and jaxsnn
hxtorch_cfg = hxtorch.ConversionConfig(dt = ..,
                                       v_reset = ..,
                                       device = ..,
                                       hardware_params = ..)
hxtorch_model = hxtorch.from_nir(nir_model, hxtorch_cfg)
```

Listing 8: Conversion of a `NIRGraph` to hxtorch. The necessary parameters for conversion are the temporal resolution `dt`, the reset potential `v_reset` and `device`, the `torch.device` to perform the computation on. When emulating the SNN on the BSS-2 hardware platform, additional hardware parameters have to be supplied.

4.3.2 Conversion to data exchange format

Since the grid-based intermediate data representation is inspired by the one in hxtorch, the conversion between both formats is rather simple. The only differences are that the data exchange format uses a `numpy.ndarray` instead of a `torch.Tensor` for data handling and also the axes of the data are permuted. The interface is shown in lst. 9 The `from_nir_data` function introduces the conversion functionality of a `NIRGraphData` object to a dictionary of `torch.Tensor`, on which hxtorch can operate. The keys of this dictionary correspond to the keys in the `nodes` dictionary of the `NIRGraph`. Analogously to this, the `to_nir_data` describes the conversion from a dictionary of `torch.Tensor` to a `NIRGraphData` object. Both directions of the conversion take the corresponding data that is to be converted as well as the hxtorch model, which holds information like the temporal resolution dt.

```
# transformation hxtorch -> NIR
nir_data = hxtorch.to_nir_data(hxtorch_data, hxtorch_model)

# transformation NIR -> hxtorch
hxtorch_data = hxtorch.from_nir_data(nir_data, hxtorch_model)
```

Listing 9: Conversion interface for conversion of dictionaries of spike data between hxtorch and the data exchange format.

4.4 Other Contributions to NIR

NIR follows a very general approach of representing spiking neural networks without too many restrictions. In previous work by the author [35] it was discovered that the lack of internal checks, e.g. for matching input or output sizes, leads to the construction of faulty `NIRGraph` instances. This issue is addressed by calling an already existing internal function, `_check_graph`, during the initialization of `NIRGraph`. If now a faulty `NIRGraph` is built by hand or by a conversion from another software framework, an error is raised. This ensures that instances of `NIRGraph` are built in the way which is specified by NIR, hardening the definition.

In addition to this, the NIR module is extended by a 17th primitive: a current-based leaky-integrator (CubaLI) neuron. This is done because CubaLI neurons are regularly used in hxtorch models in the output layer. The only difference from the CubaLIF neuron already existing in NIR is that the CubaLI does not have a spike mechanism.

As described in section 2.1, dynamics of a neuron are completely described by its differential equations and thus the neuron's parameters. For spiking neuron models, like the leaky-integrate and fire (LIF) neuron or its current-based version, the reset potential V_{reset} is an essential parameter in most software frameworks. In contrast to resetting the potential to the reset potential, NIR implements spiking neuron models such that when exceeding the threshold ϑ, the threshold is subtracted from the membrane voltage V_{mem}. In event-based simulation, this corresponds to the case where the reset potential $V_{\text{reset}} = 0$. But equality does not apply for grid-based simulators like Norse: It can happen that the membrane voltage V_{mem} is larger than the threshold ϑ when resetting, resulting in a membrane voltage $V_{\text{mem}} > 0$, caused by the finite temporal resolution of the point in time where the spike occurs.

Given the influential role of this parameter, it is proposed to introduce the attribute `v_reset` as a configurable option for all spiking neurons. To ensure backward compatibility, `v_reset=0` is kept as the default value.

5 Experiments

In the following, the previously described developed data exchange format is demonstrated. To this end, an ITL training example is set up, to show how the data exchange format enables interchangeability of training substrates – independent of the used software framework. ITL training is a good showcasing example, because frequent exchange of data and topology is necessary: First, the input spikes need to be translated to the substrate. Then, the spike data of the forward pass has to be translated to the software framework after each batch of training. Subsequently, the model description with the updated weights is translated from the training frontend to the inference backend.

For reasons of comparability, the forward pass of the network is performed via hxtorch on the neuromorphic substrate BSS-2 and also in simulation, while the resulting spikes are translated to the jaxsnn framework and then used to perform the backward pass, which yields the gradients and weight updates (fig. 5.1).

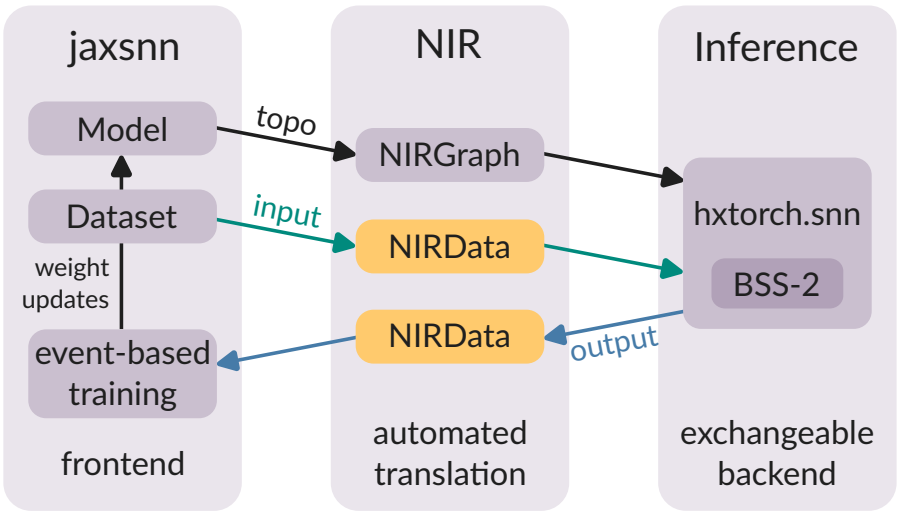


Figure 5.1: Flow graph for ITL training. The model as well as the dataset are defined in the jaxsnn. To perform the inference on the hardware platform BSS-2, the network topology as well as the batch of input data has to be translated to it via NIR. The resulting output data is then transformed back and used for gradient computation and updating the weights.

Since hxtorch uses time-gridded data, it is expected that the temporal resolution dt influences the accuracy which is achieved. To investigate this, the experiment is run for different values of dt over three orders of magnitude. In fig. 5.5, the accuracies are compared. In the next section,

the training setup as well as the performed routine explained in detail.

5.1 In-the-loop Training via NIR – One Loop to Rule Them All

This hardware ITL training example is performed to validate the functionality of the implemented data conversions on the one hand and on the other hand to build an example which can easily be adapted to work for other software frameworks. The general concept of ITL training is explained in section 2.4.

5.1.1 Experiment Setup

The YinYang dataset (section 2.6) is selected to perform the training on, since small network topologies suffice to solve this task with high accuracies of approximately 97% [29]. The SNN which is trained consists of a single hidden layer with 100 CubaLIF neurons. The input size of five neurons and the three output neurons are given by the task. For the emulation on hardware, each input neuron is quintupled, making 25 input neurons in total. A reason for this is that this increases the effective resolution of the synaptic weights on hardware. Additionally, with less synaptic input, neurons might not spike on hardware due to insufficient input strength. For computing the gradients in jaxsnn we rely on the EventProp algorithm, which is the reason for using a spiking output layer consisting of CubaLIF neurons. The neuron parameters of the hidden and the output layer as well as the initializations for the synapse layers and the dataset parameters are depicted in table 5.1. The synaptic weights are initialized randomly using a Gaussian distribution with mean μ_i and standard deviation σ_i, where i describes the index of the synapse layer.

Table 5.1: Model parameters for the software simulation model as well as the hardware emulation model: The means μ_i and standard deviations σ_i of the Gaussian initializations of the synapse layers, the time constants τ_{mem} (membrane) and τ_{syn} (synaptic), the voltages V_{th} (threshold), V_{leak} (leak) and V_{reset} (reset) and the target times t_{target} for the correct class and $t_{\text{no target}}$ for the wrong classes. The input spikes happen between t_{early} and t_{late}.

(a) Simulation Parameters

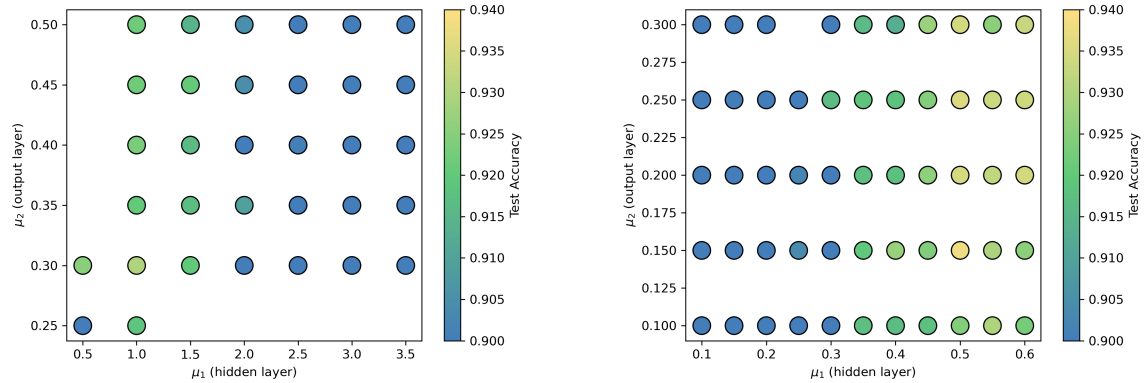
parameter	value
μ_1	1.0
μ_2	0.3
σ_1	1.6
σ_2	0.8
τ_{mem}	$10 \cdot 10^{-3}$ s
τ_{syn}	$5 \cdot 10^{-3}$ s
V_{th}	1.0 a.u.
V_{leak}	0.0 a.u.
V_{reset}	-1000.0 a.u.
t_{target}	$4.5 \cdot 10^{-3}$ s
$t_{\text{no target}}$	$5.5 \cdot 10^{-3}$ s
t_{early}	0 s
t_{late}	$10 \cdot 10^{-3}$ s

(b) Hardware Parameters

parameter	value
μ_1	0.5
μ_2	0.15
σ_1	0.4
σ_2	0.2
τ_{mem}	$12 \cdot 10^{-6}$ s
τ_{syn}	$6 \cdot 10^{-6}$ s
V_{th}	1.0 a.u.
V_{leak}	0.0 a.u.
V_{reset}	0.0 a.u.
t_{target}	$5.4 \cdot 10^{-6}$ s
$t_{\text{no target}}$	$6.6 \cdot 10^{-6}$ s
t_{early}	0 s
t_{late}	$12 \cdot 10^{-6}$ s

The parameters for the weight initialization are found by starting at the values used by Göltz et al. [30] and then validating them by grid sweeps for the means μ_1 and μ_2, both in software and on hardware (fig. 5.2). Both plots show the test accuracy after 50 epochs of training with the training procedure that is described in section 5.1.2. As expected, it can be seen that the accuracy differs for different initializations of the synapse layer weights. For the software runs, the combination of $\mu_1 = 1.0$, $\mu_2 = 0.3$ shows the highest accuracy within the investigated range, while on hardware the this is achieved with $\mu_1 = 0.5$, $\mu_2 = 0.15$. Also both plots appear continuous within the measured range, without abrupt jumps or irregularities.

Figure 5.2: Grid sweeps for the means μ_1 and μ_2 of the Gaussian initializations of the synapse layers. Each data point in the plot represents the mean of three individual measurements.



(a) Grid sweep for the means μ_1 and μ_2 in software

(b) Grid sweep for the means μ_1 and μ_2 on hardware

The training parameters, such as the sizes of the datasets and the learning rate are depicted in table 5.2.

Table 5.2: Training Parameters

parameter	value
$N_{\text{train samples}}$	5000
$N_{\text{test samples}}$	3000
batch size	64
learning rate	0.005
learning rate decay	0.99

5.1.2 Training Procedure

In lst. 10, an example depicts how the ITL training looks like (fig. 5.3). First the network topology is defined in jaxsnn, the framework for the backward pass, and then converted to hxtorch, the framework for the forward pass. For the conversion from NIR to hxtorch, a separate `ConversionConfig` is delivered.

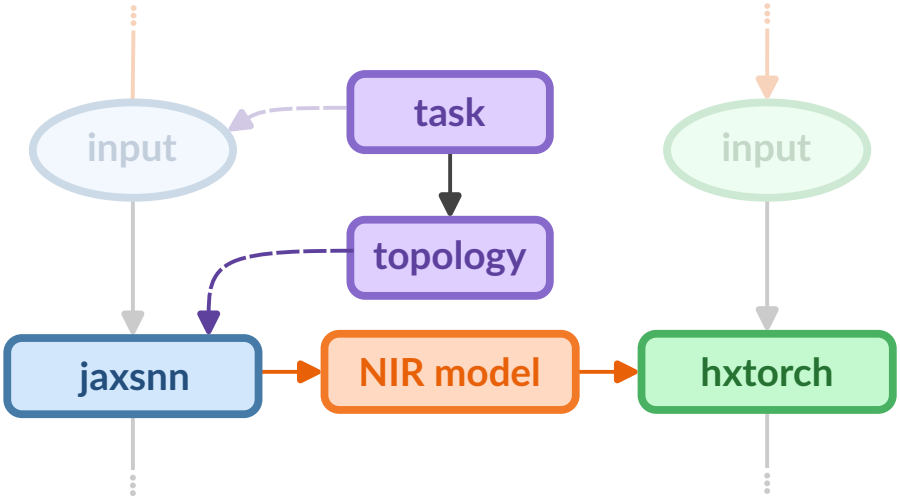


Figure 5.3: Ideal model conversion during ITL training. Process of implementation (purple arrows) of a topology that has previously been the subject of consideration on the basis of a task. The rectangles describe a model for spiking neural networks. With NIR, the model has only to be implemented in jaxsnn (blue) and can be transformed to hxtorch (green).

```
# Generate trainset and testset in a jaxsnn representation
trainset = yinyang_dataset(...)
testset = yinyang_dataset(...)

# Define topology in jaxsnn
jaxsnn_model = ...

# Transform the jaxsnn model via NIR to hxtorch
nir_model = jaxsnn.to_nir(jaxsnn_model)
hxtorch_cfg = hxtorch.ConversionConfig(...)
hxtorch_model = hxtorch.from_nir(nir_model, hxtorch_cfg)
```

Listing 10: Conversion of the models for the ITL training example. The network topology is defined in jaxsnn and then converted via NIR to hxtorch.

With the models set and the datasets generated, a training routine is performed for each batch (lst. 11). The data is drawn randomly and batched from a trainset, which is defined in jaxsnn. Each input batch gets converted from jaxsnn via the data exchange format to hxtorch. After execution of the inference, the resulting spikes of the hidden layer as well as the output layer are converted back to jaxsnn to compute the weight updates with the backward pass.

With the new weights, also the hxtorch model is updated.

```
for e in epochs:
    # Draw data from trainset
    trainset = # ... Yin-Yang ...

    for jaxsnn_input, jaxsnn_target in trainset:
        # input transformation: jaxsnn -> NIR -> hxtorch
        nir_input = jaxsnn.to_nir_data(jaxsnn_input, jaxsnn_model)
        hxtorch_input = hxtorch.from_nir_data(nir_input, hxtorch_model)

        # inference: hxtorch
        hxtorch_output = hxtorch_model(hxtorch_input)

        # output transformation: hxtorch -> NIR -> jaxsnn
        nir_output = hxtorch.to_nir_data(hxtorch_output, hxtorch_model)
        jaxsnn_output = jaxsnn.from_nir_data(nir_output, jaxsnn_model)

        # update weights in jaxsnn model
        # gradient calculations using outputs from hxtorch inference
        jaxsnn_model = jaxsnn.grad_and_update(
            jaxsnn_model,
            loss_fn, optimizer,
            jaxsnn_input, jaxsnn_target, jaxsnn_output)

        # model transformation: jaxsnn -> NIR -> hxtorch
        nir_model = jaxsnn.to_nir(jaxsnn_model, ...)
        hxtorch_model = hxtorch.from_nir(nir_model, ...)
```

Listing 11: ITL training procedure. First the input spikes are converted from the jaxsnn framework to hxtorch, where the forward pass is executed. The resulting output spikes are converted from hxtorch via NIR and then used to perform the backward pass and update the weights. The updated model is finally converted from jaxsnn to hxtorch for the next loop entry.

5.1.3 Deviations from the Ideal Experiment Procedure

The model conversion from jaxsnn to NIR is not implemented yet, thus the weight updates are inserted directly into the hxtorch model and also initially the network topology is defined in NIR and can then be transformed to jaxsnn and hxtorch, instead of implementing the jaxsnn topology first and converting it to hxtorch.

But due to jaxsnn-internal bugs, only the recurrent adjoint dynamics are computing the correct gradients, so that the `HardwareRecurrentLIF` has to be used in the jaxsnn representation of the network. And since the conversion from NIR to jaxsnn currently does not support the

conversion of recurrent spiking neural networks (RSNNs), the jaxsnn model is implemented by hand.

The model conversion as it is performed in the ITL training is depicted in fig. 5.4.

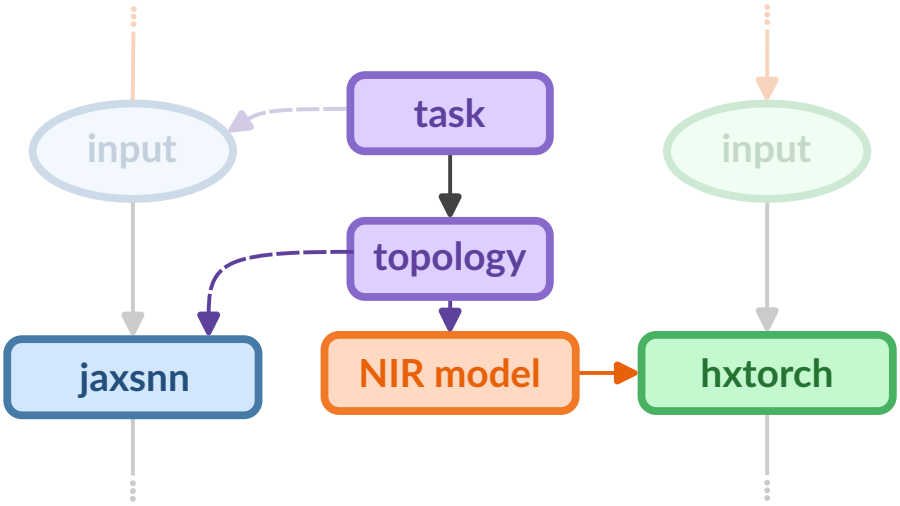


Figure 5.4: Deviations from the ideal ITL model conversion shown in fig. 5.3. Process of implementation (purple arrows) of a topology that has previously been the subject of consideration on the basis of a task. The rectangles describe a model for spiking neural networks. In contrast to the ideal model transformation, the model is implemented both in jaxsnn and in NIR and then converted from NIR to hxtorch.

5.1.4 Experiment Results

The training is performed for a sweep of temporal resolutions over three orders of magnitude. For each temporal resolution, the mean and the standard deviation of the test accuracy of five independent runs are determined. The resulting plot in fig. 5.5 shows that the test accuracy for the software runs strongly correlates with the temporal resolution dt: Training with a temporal resolution of $0.1\tau_{\text{syn}}$ only achieves an accuracy of $(45 \pm 10)\%$ in simulation, while for smaller resolutions, the accuracy saturates at about 95%. The temporal resolution $dt = 0.001\tau_{\text{syn}}$ achieved the highest accuracy, with a value of $(95.2 \pm 0.8)\%$. The bad performance for large temporal resolutions is explained by discretization of the time grid the neuron dynamics are evolved on. Additionally, the resolution of the time grid spikes are represented on is reduced. The resolution of the spike time is reduced. Especially looking at the difference between the target times of the correct and wrong neurons, which only differ by $0.2\tau_{\text{syn}} = 2\ dt$, it is clear why the model performs worse for this temporal resolution.

When emulating the experiment on hardware, again the largest dt corresponds to the worst accuracy $(87.1 \pm 2.5)\%$. But the difference to the highest accuracy $(93.0 \pm 0.9)\%$, achieved for a temporal resolution of $0.01\tau_{\text{syn}}$, is not as significant as in simulation. Since when using the

neuromorphic hardware, the neural dynamics are not influenced by dt, the worse accuracy in simulation is most likely caused by the discretization of the numerical computation of the neural dynamics.

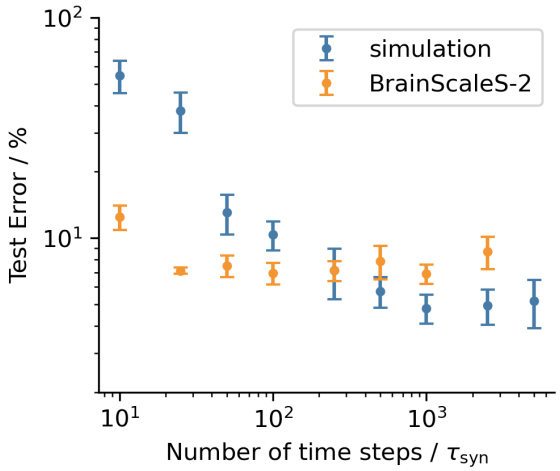


Figure 5.5: Test accuracy after ITL training on software (blue) and hardware (orange) in dependence of the number of time steps N per τ_{syn}: $N = \tau_{\text{syn}}/dt$. For software, the temporal resolution is used for both the data discretization as well as the neuron dynamics. In comparison, the dynamics on hardware are not discretized since they are emulated physically – only the spike times are discretized onto the time grid. This is expected to lead to the test accuracy for the hardware runs to be more independent of the temporal resolution.

5.2 Runtime Performance Evaluation of In-the-loop Training

The runtime performance of ITL training plays a major role for the feasibility of such a training routine. It is expected, that the runtime performance is worse than the training within a single software framework where no conversion are necessary to be performed. Nevertheless, the order of magnitude of these runtime performance losses is highly relevant: Ideally, the conversion of observables only takes up a minor part in the total runtime.

To investigate this, the runtimes of the different components of the training routine are recorded and averaged over 50 epochs. Therefore four training runs are conducted: two with inference on BSS-2 and two in pure simulation, each with a small $dt_1 = 0.02\tau_{\text{syn}}$ and a large $dt_2 = 0.0004\tau_{\text{syn}}$. The measured runtime is divided among the training routines' key components, including the input and output transformation of data that was performed via the developed intermediate data format. The setup used to evaluate runtime performance consists of two AMD EPYC 7543 32-core processors (128 virtual cores in total) and 1000 GB of RAM. Figure 5.5 shows the measurement results.

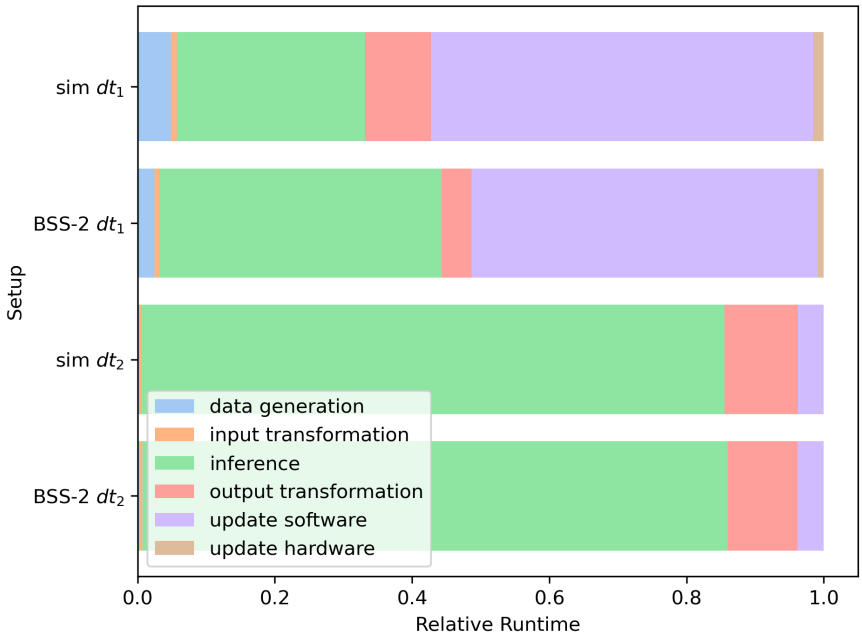


Figure 5.6: Experiment runtimes divided into the key components of the training routine. The experiment is performed in simulation (sim) and emulated on BrainScaleS-2 (BSS-2). Also for each execution platform, an experiment run with a large temporal resolution $dt_1 = 0.02\tau_{\text{syn}}$ and a small one $dt_2 = 0.0004\tau_{\text{syn}}$ is performed.

Table 5.3: Mean runtime per batch for different training setups

setup	sim dt_1	BSS-2 dt_1	sim dt_2	BSS-2 dt_2
total runtime per batch (s)	15.4 ± 2.9	38 ± 5	308 ± 4	364 ± 7

As expected, the runtime of the experiment with larger temporal resolution dt_1 exceeds the one with smaller temporal resolution dt_2 by far, as depicted in table 5.3. When comparing the inference durations, the experiment with a temporal resolution that is 50 times larger than in the short experiment also shows a similar increase in simulation inference time, with a factor of approximately 62. On BSS-2, this factor is only 20. This can be explained by the higher overhead associated with hardware execution at small temporal resolutions, which leads to longer runtimes compared to simulation. In contrast, the simulation of neuron dynamics becomes increasingly expensive as the temporal resolution increases. The duration of the data conversion also scales with the temporal resolution, but not as much as for the inference. Furthermore, it can be seen that the relative experiment runtimes do not differ much between using BSS-2 or simulation but between the different temporal resolutions. Most importantly, independent of the experiment setup investigated, the data conversion only requires up to 10% of the total runtime, which is reasonable.

6 Discussion

In this bachelor thesis, a data exchange format is proposed as extension of the neuromorphic intermediate representation (NIR), which is designed for topologies of spiking neural networks (SNNs). NIR describes network topologies using a `NIRGraph`, which contains neurons and synapses represented as `NIRNode` objects. The developed data exchange format stores the observables or input data of a SNN node-wise, matching the `NIRNode` objects of the corresponding `NIRGraph`. A flexible interface supports the definition of arbitrary observables per node. Although the data format is only implemented for spike data at the moment, extending this to meet the additional needs of software frameworks is possible, see chapter 7.

Among existing software frameworks, two different representations of spike data are used: an event-based and a time-gridded one. In order to provide a handy interface for these, the data exchange format implements their native formats for spike data – as well as a conversion between them.

As part of this thesis, the model conversion between the hxtorch software framework and NIR is implemented. Additionally, the data conversion between the data exchange format and the software frameworks jaxsnn and hxtorch is developed.

This facilitates machine-learning-inspired experiments where flexible data exchange is necessary, like hardware in-the-loop (ITL) training: There, for each epoch, the input spikes have to be transformed from the software framework to the hardware while the data from the inference on hardware has to be transformed back into the software framework. In combination, NIR and the data exchange format enable training of a network without requiring detailed knowledge of how topologies and spike data are represented within the specific software framework. Also, accessing hardware substrates via NIR is simplified by removing the constraint to directly using software frameworks which are connected to the hardware. In general the data exchange format has been designed in such a way that it is interoperable with NIR. Thus it can be used in experiments where interchangeability between software frameworks is necessary.

For demonstration of the above, an in-the-loop training example is performed. In this example, the forward pass is executed on the BrainScaleS-2 (BSS-2) hardware platform via hxtorch, while the backward pass is performed using jaxsnn. The performance of this is evaluated to investigate the impact of the temporal resolution dt on training when using the EventProp algorithm for gradient calculation. For this purpose, the test error is compared for a sweep over different temporal resolutions dt in simulation and also using the neuromorphic hardware.

It is observed that the test accuracy is strongly anticorrelated with the temporal resolution dt,

while on hardware, from a certain dt on, the accuracy is independent of the temporal resolution, achieving its highest accuracy $(93.0 \pm 0.9)\%$ for $dt = 0.01\tau_{\text{syn}}$. The simulation achieves the highest test accuracy of $95.2 \pm 0.8)\%$ for temporal resolutions $dt = 0.001\tau_{\text{syn}}$.

This section compares the newly implemented data exchange format to existing data representations used in intermediate formats for machine learning. In the case of artificial neural networks (ANNs), the data format is generally consistent across software frameworks, as the data is typically represented simply as a tensor of numbers. This is why ONNX [19], an example for an intermediate representation for ANNs, does not include an exchange format for data. In contrast, SNNs rely on more diverse representations of spiking activity, leading to greater variation between frameworks. PyNN [20] serves as an exchange format for SNNs, supporting not only their topologies but also including an intermediate data format. The main disadvantage is that this data format is difficult to handle for machine-learning purposes such as training, since PyNN is targeting the field of computational neuroscience and the corresponding data format was explicitly designed to represent complex biological recordings. The data exchange format designed and developed in this thesis differs from PyNN in that it can be translated into other software frameworks' native formats. In particular, this directly enables training for the respective frameworks using the transformed observables.

7 Outlook

The data exchange format that is developed in this thesis is designed to be extensible. Therefore, an obvious extension is the support for arbitrary observables, like membrane traces. This can be achieved by adding `ValuedEventData` next to the existing `EventData` class. For spike data, the only information to be represented is when a spike happens (time) and from which neuron it originates (idx). But for other observables, a value is to be recorded in addition. Sticking to the example of the membrane recording, the conversion from event-based data to time-gridded data has to implement some kind of configurable interpolation functionality because there has to be a value for every time step, not only when an event occurred:

```
valued_event_data = ValuedEventData(idx_array, time_array, value_array,
↪  interpolation_method)
```

The `TimeGriddedData` format can also be extended by an additional conversion which returns the data in a time-gridded format but with different temporal resolution. To implement this, it is necessary to decide how to proceed with more that one events in one time step or, when converting to a smaller temporal resolution dt, which time step holds the event. The latter issue is similar to the one occurring in the conversion of time-gridded data to event-based data, where a `time_shift` parameter was introduced (fig. 4.2b).

A valid intermediate format for topologies and data is key to providing a user-friendly experience. To achieve this, an internal check is now performed during initialization of `NIRGraph` objects, ensuring it was built properly. A similar approach can be followed for verifying `NIRGraphData` objects. Two exemplary aspects that should be tested for are: first, that the used keys of `NIRNodeData` also corresponds to a `NIRNode` in the `NIRGraph`; and second, that the data matches the corresponding node in shape.

A goal of development that is not directly connected to the data exchange format is the implementation of the `jaxsnn.to_nir` function to convert jaxsnn models to the NIR format. This will for example enable a more inherent implementation of hardware ITL training, where the network topology is initially defined in jaxsnn and then transformed via NIR to hxtorch and also the update of the hxtorch model during training can be performed via NIR.

To take full advantage of the capabilities of the BSS-2 hardware platform, the time-gridding performed in hxtorch should be omitted for accessing the hardware. This would increase efficiency by eliminating an unnecessary conversion step and could also improve performance by returning precise spike times instead of discretized ones.

8 Acknowledgments

First of all, I would like to thank Prof. Johannes Schemmel for giving me the opportunity to write my Bachelor's thesis in the Electronic Visions group.

From the very beginning, I felt warmly welcomed by the entire group. A major part of this positive experience is thanks to my supervisors Elias, Philipp, and Eric, who not only helped me keep sight of the bigger picture during my thesis but also supported me with debugging when I was stuck and couldn't find a way forward. Working with them was both enjoyable and inspiring — our interactions made research feel exciting and accessible, and they definitely contributed to my growing interest in scientific work. Equally important were my office mates, Florian and Lennart. Our office had a perfect balance of casual conversations and focused coding sessions, which I truly appreciated. I would also like to thank the rest of the group, who were always friendly, approachable, and supportive.

Furthermore, I want to thank Elias, Philipp, Florian, and Jakob for taking the time to proofread my thesis. Your feedback and suggestions were a great help.

Last but not least, I want to thank my parents, my siblings, and my family in general for their continuous support throughout this time.

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

Acronyms

ADC analog-to-digital converter

AdEx adaptive exponential integrate-and-fire

ANN artificial neural network

BPTT backpropagation through time

BSS-2 BrainScaleS-2

CADC columnar ADC

CubaLI current-based leaky-integrator

CubaLIF current-based leaky-integrate and fire

FPGA field-programmable gate array

ITL in-the-loop

JIT just-in-time

LI leaky-integrator

LIF leaky-integrate and fire

MADC membrane ADC

NIR neuromorphic intermediate representation

NN neural network

RSNN recurrent spiking neural network

SNN spiking neural network

9 References

[1] Susan Rakov and Abigail Ham. *Fact file: Computing is using more energy than ever*. https://frontiergroup.org/resources/fact-file-computing-is-using-more-energy-than-ever/. Accessed: 2025-04-10. Oct. 2023.

[2] Christopher Willuweit, Carsten Bockelmann, and Armin Dekorsy. "Energy and Bandwidth Efficiency of Event-Based Communication". In: *IEEE 97th Vehicular Technology Conference (VTC Spring)*. IEEE. 2023, pp. 1–5. DOI: 10.1109/VTC2023-Spring.2023.10181234.

[3] Nicolas Alder and Ralf Herbrich. "Energy-Efficient Gaussian Processes Using Low-Precision Arithmetic". In: *Proceedings of the 41st International Conference on Machine Learning*. Ed. by Ruslan Salakhutdinov, Zico Kolter, Katherine Heller, Adrian Weller, Nuria Oliver, Jonathan Scarlett, and Felix Berkenkamp. Vol. 235. Proceedings of Machine Learning Research. PMLR, July 2024, pp. 955–975. URL: https://proceedings.mlr.press/v235/alder24a.html.

[4] Justyna Zwolak. *Ada Lovelace: The World's First Computer Programmer Who Predicted Artificial Intelligence*. Accessed: 2025-04-10. Mar. 2023. URL: https://www.nist.gov/blogs/taking-measure/ada-lovelace-worlds-first-computer-programmer-who-predicted-artificial.

[5] Giuseppe Carleo, Ignacio Cirac, Kyle Cranmer, Laurent Daudet, Maria Schuld, Naftali Tishby, Leslie Vogt-Maranto, and Lenka Zdeborová. "Machine Learning and the Physical Sciences". In: *Reviews of Modern Physics* 91.4 (2019), p. 045002. DOI: 10.1103/RevModPhys.91.045002. arXiv: 1903.10563 [physics.comp-ph]. URL: https://arxiv.org/abs/1903.10563.

[6] John A. Keith, Valentin Vassilev-Galindo, Bingqing Cheng, Stefan Chmiela, Michael Gastegger, Klaus-Robert Müller, and Alexandre Tkatchenko. "Combining Machine Learning and Computational Chemistry for Predictive Insights Into Chemical Systems". In: *Chemical Reviews* 121.16 (2021), pp. 9816–9872. DOI: 10.1021/acs.chemrev.1c00107. arXiv: 2102.06321 [physics.chem-ph]. URL: https://arxiv.org/abs/2102.06321.

[7] Mohammad Shehab, Laith Abualigah, Qusai Shambour, Muhannad A. Abu-Hashem, Mohd Khaled Yousef Shambour, Ahmed Izzat Alsalibi, and Amir H. Gandomi. "Machine learning in medical applications: A review of state-of-the-art methods". In: *Computers in Biology and Medicine* 145 (June 2022). Epub 2022 Mar 28, p. 105458. DOI: 10.1016/j.compbiomed.2022.105458. URL: https://pubmed.ncbi.nlm.nih.gov/35364311/.

[8] Mohamad Kamaluddin, Moch Rasyid, Fourus Abqoriyyah, and Andang Saehu. "Accuracy Analysis of DeepL: Breakthroughs in Machine Translation Technology". In: *Journal of English Education Forum (JEEF)* 4 (June 2024), pp. 122–126. DOI: 10.29303/jeef.v4i2.681.

[9] Marc-Oliver Gewaltig and Markus Diesmann. "NEST (NEural Simulation Tool)". In: *Scholarpedia* 2.4 (2007), p. 1430. DOI: 10.4249/scholarpedia.1430.

[10] Marcel Stimberg, Romain Brette, and Dan Fm Goodman. "Brian 2, an intuitive and efficient neural simulator". In: *eLife* 8 (Aug. 2019). DOI: 10.7554/eLife.47314.

[11] 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.

[12] Jason K Eshraghian, Max Ward, Emre 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: *arXiv preprint* (2021). arXiv: 2109.12894 [cs.NE].

[13] 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].

[14] Eric Müller, Sebastian Schmitt, Christian Mauch, Hartmut Schmidt, José Montes, Joscha Ilmberger, Johann Klähn, Felix Passenberg, Christoph Koke, Mitja Kleider, Sebastian Jeltsch, Maurice Güttler, Dan Husmann, Sebastian Billaudelle, Paul Müller, Andreas Grübl, Jakob Kaiser, Jonas Weidner, Bernhard Vogginger, Johannes Partzsch, Christian Mayr, and Johannes Schemmel. "The Operating System of the Neuromorphic BrainScaleS-1 System". In: *arXiv preprint* (Mar. 2020). submitted to Neurocomputing OSP. arXiv: 2003.13749 [cs.NE]. URL: http://arxiv.org/abs/2003.13749.

[15] Mike Davies, Narayan Srinivasa, Tsung-Han Lin, Gautham Chinya, Yongqiang Cao, Sri Harsha Choday, Georgios Dimou, Prasad Joshi, Nabil Imam, Shweta Jain, et al. "Loihi: A neuromorphic manycore processor with on-chip learning". In: *IEEE Micro* 38.1 (2018), pp. 82–99. DOI: 10.1109/MM.2018.112130359.

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

[17] 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.

[18] Sebastian Schmitt, Johann Klähn, Guillaume Bellec, Andreas Grübl, Maurice Güttler, Andreas Hartel, Stephan Hartmann, Dan Husmann, Kai Husmann, Sebastian Jeltsch, Mitja Kleider, Christoph Koke, Alexander Kononov, Christian Mauch, Eric Müller, Paul Müller, Johannes Partzsch, Mihai A. Petrovici, Bernhard Vogginger, Stefan Schiefer, Stefan Scholze, Vasilis Thanasoulis, Johannes Schemmel, Robert Legenstein, Wolfgang Maass, Christian Mayr, and Karlheinz Meier. "Neuromorphic Hardware In The Loop: Training a Deep Spiking Network on the BrainScaleS Wafer-Scale System". In: *Proceedings of the 2017 IEEE International Joint Conference on Neural Networks (IJCNN)* (2017), pp. 2227–2234. DOI: 10.1109/IJCNN.2017.7966125. URL: http://ieeexplore.ieee.org/document/7966125/.

[19] ONNX Contributors. *ONNX: Open Neural Network Exchange*. https://github.com/onnx/onnx. Accessed: 2025-04-13.

[20] Andrew P. Davison, Daniel Brüderle, Jochen Eppler, Jens Kremkow, Eilif Muller, Dejan Pecevski, Laurent Perrinet, and Pierre Yger. "PyNN: a common interface for neuronal network simulators". In: *Front. Neuroinform.* 2.11 (2009). DOI: 10.3389/neuro.11.011.2008.

[21] Jens E. Pedersen, Steven Abreu, Matthias Jobst, Gregor Lenz, Vittorio Fra, Felix C. Bauer, Dylan R. Muir, Peng Zhou, Bernhard Vogginger, Kade Heckel, Gianvito Urgese, Sadasivan Shankar, Terrence C. Stewart, Jason K. Eshraghian, and Sadique Sheik. "Neuromorphic Intermediate Representation: A Unified Instruction Set for Interoperable Brain-Inspired Computing". In: 2023. DOI: 10.48550/arXiv.2311.14641.

[22] 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.

[23] Philipp Spilger, Eric Müller, Arne Emmel, Aron Leibfried, Christian Mauch, Christian Pehle, Johannes Weis, Oliver Breitwieser, Sebastian Billaudelle, Sebastian Schmitt, Timo C. Wunderlich, Yannik Stradmann, and Johannes Schemmel. "hxtorch: PyTorch for BrainScaleS-2 — Perceptrons on Analog Neuromorphic Hardware". In: *IoT Streams for Data-Driven Predictive Maintenance and IoT, Edge, and Mobile for Embedded Machine Learning*. Cham: Springer International Publishing, 2020, pp. 189–200. ISBN: 978-3-030-66770-2. DOI: 10.1007/978-3-030-66770-2_14.

[24] Diederik P. Kingma and Jimmy Ba. "Adam: A Method for Stochastic Optimization". In: *International Conference on Learning Representations* (2014). arXiv: 1412.6980 [cs.LG].

[25] 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.

[26] 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.

[27] Sebastian Schmitt, Johann Klähn, Guillaume Bellec, Andreas Grübl, Maurice Güttler, Andreas Hartel, Stephan Hartmann, Dan Husmann, Kai Husmann, Sebastian Jeltsch, Mitja Kleider, Christoph Koke, Alexander Kononov, Christian Mauch, Eric Müller, Paul Müller, Johannes Partzsch, Mihai A. Petrovici, Bernhard Vogginger, Stefan Schiefer, Stefan Scholze, Vasilis Thanasoulis, Johannes Schemmel, Robert Legenstein, Wolfgang Maass, Christian Mayr, and Karlheinz Meier. "Neuromorphic Hardware In The Loop: Training a Deep Spiking Network on the BrainScaleS Wafer-Scale System". In: *Proceedings of the 2017 IEEE International Joint Conference on Neural Networks (IJCNN)* (2017), pp. 2227–2234. DOI: 10.1109/IJCNN.2017.7966125. URL: http://ieeexplore.ieee.org/document/7966125/.

[28] 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.

[29] 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.

[30] 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.

[31] 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.

[32] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. "PyTorch:

An Imperative Style, High-Performance Deep Learning Library". In: *Advances in Neural Information Processing Systems 32*. Ed. by H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett. Curran Associates, Inc., 2019, pp. 8024–8035. URL: http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf.

[33] Florian Fischer. *Event-based Learning of Synaptic Delays and Arbitrary Topologies*. Bachelor's thesis. 2025.

[34] Neuromorphs. *NIRTorch: PyTorch helper module to translate to and from NIR*. https://github.com/neuromorphs/NIRTorch. Accessed: 2025-03-31. 2025.

[35] Ben Kroehs. *Initial Steps Towards a Translation from NIR to jaxsnn*. Internship Report. Department of Physics and Astronomy, Heidelberg University, Jan. 2025.

A Software State

Table A.1: Apptainer container used for all experiments.

key	value
path	`/containers/stable/2025-02-19_1.img`
fingerprint	`64d91623-6dcd-43b5-886a-2d5eca67eff6`
app	dls

Table A.2: Software state used in experiments.

repository	git hash	change set
jax-snn	`75383a7f2e52e6e389b5e50a07dbfcdd7a699b4d`	21824
code-format	`09f3a985a6f264359b10a6a129dd6dce7e55c9e8`	
hxtorch	`701abf3104d8c4dfe2c749edab4e50c2da91eafa`	24718
hxcomm	`0182b27465ca5f468fbf2c5f4849b5f6934be56a`	
haldls	`d8d62550b2cd4183e6d9ed08c4f39ca2be9a634c`	
grenade	`10f67d92fff203348e7d897379903e91dc6bdd09`	
hate	`35b3cb211cabbbc5c01036ae7878a73e338166c4`	
calix	`a706868c6ba285b1f8fd7cdef1a19d7328e02912`	
sctrltp	`1d854f953f7e8c8ead44406a22bb80421ca3857c`	
rant	`53199ee94cae1e1c2e4db10e88d570a761b14e0f`	
hwdb	`28a72d4d635aa6c0f1522616b1dacfaa817baeef`	
logger	`73dadb3ce413c521845ef7d36f818073eee4fefa`	
visions-slurm	`8f41ea4f5bd1573d8f4623e9ed698a29f30036a3`	
flange	`28e729d59df3b4ff380f84351c40d4da3086bed8`	
lib-rcf	`000185eb11db4d54cb6b12b09af54cf742741036`	
bss-hw-params	`b7be7827b51536804f0bda76f8ba4be693df23a8`	
halco	`a97040a732ab1ba954e077616303a18acf623092`	
fisch	`a67fc99215f038f09a33fd09ff85c0bb594f9f8c`	
libnux	`edfac92188ad2d78c68c7f19f606e45b8cb3b316`	
pywrap	`5e2af30e9593882b471d3cd02df00b93f13ff479`	
ztl	`c2d4faee05f497010ee55e35bf9c9607eecbf884`	
lib-boost-patches	`136c5b41cb046afe2c726aa4646928bf5190622e`	

Erklärung:

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

Heidelberg, den 16.04.2025 .....