Initial Steps Towards a Translation from NIR to jaxsnn

Internship report

Ben Luca Kroehs

Supervisors: Elias Arnold, Philipp Spilger and Eric Müller

Department of Physics and Astronomy

Heidelberg University

January 16th 2025

Abstract

Spiking neural networks can either be simulated in traditional computing devices or be emulated on dedicated neuromorphic hardware. There is a variety of neuromorphic software frameworks that all have different implementations of spiking neural networks. Currently, translating between those representations of neural networks is rather a manual thing to do. Recently, the approach was proposed to create an intermediate representation which should enable an easier way of building a network on one platform and translate it to another one. In this internship the translation from NIR to jaxsnn is partly developed. To validate the translation, a model is translated from Norse to NIR to jaxsnn and its performance in compared and validated.

Contents

1 Introduction

Especially in the last few years, neural networks are getting increasingly more public attention. Spiking neural networks follow an approach of transmitting information via discrete signals, so called spikes. These networks are can be trained and simulated by various neuromorphic frameworks, e.g. jaxsnn [1], Norse [2] or hxtorch [3]. There are also some neuromorphic hardware platforms like BrainScaleS-2 [4] or SpiNNaker [5], which are able to emulate those networks. To compare these frameworks and their performance, or even to compare them with the performance of a hardware platform, it is necessary to translate these networks between the different representations, since the only alternative is to rewrite the whole network. The Neuromorphic Intermediate Representation [6] aims to provide a representation to and from which other frameworks can translate their models. This intended to simplify the translation process.

In this report we explore the first parts of the conversion of spiking neural networks from the neuromorphic intermediate representation to the event-driven framework jaxsnn. We implement the translation of linear feed-forward networks and are only supporting `nir.CubaLIF` and `nir.Linear` nodes. To validate this process we implement an example network in Norse which is trained on the YinYang dataset [7]. Then the network is translaten to NIR and finally to jaxsnn. The test accuracy is 80% in Norse and in jaxsnn it is 79%. We can conclude that the conversion works since both the accuracies match.

A conversion from NIR to jaxsnn will make This is especially interesting for BrainScaleS-2 since this makes it possible that spiking neural networks defined in different software frameworks can easily be trained and emulated on hardware.

2 Methods

2.1 Spiking Neural Networks

Spiking neural networks (SNNs) describe a subgroup of neural networks (NNs). Their main feature is that the information is encoded in temporal pulses - also known as spikes. Another crucial feature is that they evolve a state in time and have temporal dynamics.

This is something inherited from biology. Also the brain uses these spikes between neurons to communicate. There are different ways to implement such a spiking neural network. A commonly used neuron model is the leaky-integrate-and-fire (LIF) model, where we assume all spike to have the same shape.

2.2 Leaky-Integrate-And-Fire Model

The leaky-integrate-and-fire model describes the temporal dynamics of the membrane potential V_{mem} of the neuron. Am electric current or incoming spikes cause the membrane potential to rise. If it reaches the threshold V_{th}, a spike is sent out and the membrane potential is reset to V_{reset} for the refractory time t_{ref}. The membrane potential also continuously decays with the time constant τ_{mem} to the resting potential $V_{\mathtt{rest}}$. The following differential equation represents the dynamics:

$$\tau_{\mathrm{mem}} \frac{dV_{\mathrm{mem}}}{dt} + V_{\mathrm{mem}}(t) = V_{\mathrm{rest}} + \frac{I}{g_l}. \tag{1}$$

Here g_l describes the leak conductance. Commonly the current $I(t)$ follows an exponential function. This is then described as current-based LIF (CubaLIF).

2.3 Grid-based and Event-based Simulation

On some neuromorphic frameworks, spiking neural networks operate on tensor-like data. In the discrete way, spike data is represented in a matrix where "0" indicates that there is no spike and "1" indicates there is a spike in the corresponding time step with size dt. This matrix is very sparse which can make the data-handling very inefficient. An alternative way is to represent the spikes by a list of spike times. This is a dense data representation and thus a more memory efficient data handling can be possible.

2.4 jaxsnn: A Framework for Efficient SNN Simulation

The platform jaxsnn aims to be an efficient SNN simulator and training tool with support for BrainScaleS-2 - a mixed-signal neuromorphic system.

SNNs in jaxsnn are represented by two functions: the `init()` and the `apply()` function. The `init()` function which initiates the weight matrices in the right shape and also the neuron parameters if needed, while the `apply()` function represents the forward pass through the model and also implicitly the backward pass. jaxsnn is based on JAX [8], a library based on NumPy which includes just-in-time compilation and is focused on array calculations. JAX also includes efficient and convenient function transformation. This is particularly useful when a function is to be applied to each sample in a batch. The efficiency of just-in-time compilation combined with the flexibility of function transformation is why JAX is used.

2.4.1 Current TTFS Solver in jaxsnn

When a SNN is used for a classification problem, there are several ways to determine the label from the spike times of the output layers. A label can be determined by the neuron that is spiking most frequently in a set time or by the neuron that spikes first. For the forward pass through the network we use a time-to-first-spike classifier. For finding the next spiking event starting from a neuron state at any time the current jaxsnn library supports an analytical solver, which was derived and implemented in PyTorch by Göltz et al. [9], referred to here as the "original implementation", and then implemented in jaxsnn by Althaus [10], the "jaxsnn implementation". This applies to the case that $\tau_{\text{mem}} = 2\tau_{\text{syn}}$. The neuron state includes the current I and the membrane voltage V. The time to the next spike time T is then calculated by

$$T = \tau_{\text{mem}} \cdot \log\left(\frac{2a_1}{a_2 + \sqrt{a_2^2 - 4a_1 V_{\text{th}}}}\right) \tag{2}$$

where $a_1 = I$, $a_2 = I + V$ and V_{th} describes the voltage threshold and a conductance $g_l = 1$ is assumed.

2.5 Neuromorphic Intermediate Representation

Currently there are many different software frameworks and hardware platforms for simulating and emulating SNNs. On the one hand there are simulating and learning frameworks like Norse, hxtorch and jaxsnn, whereby all of them implement time-stepped simulation and jaxsnn also provides event-stepped simulation. Both hxtorch and jaxsnn are also used to emulate SNNs on neuromorphic hardware like BrainScaleS.

If a model is designed in Norse and now should be used on neuromorphic hardware, it has to be translated. That is where the Neuromorphic Intermediate Representation (NIR) comes in: The idea is that every framework establishes a translation for models from a so-called NIR graph data structure to a network representation runable on the given framework and vice versa. Without this intermediate representation, much more translation would need to be implemented. An SNN implemented for one hardware system or described in one software framework would have to be reimplemented from scratch for another hardware systems or framework. This also facilitates the evaluation of SNNs in a single software framework on different neuromorphic systems.

```python
import numpy as np
import nir

weight1 = np.arange(1, 3).reshape(2, 1)
weight2 = np.arange(0, 4).reshape(2, 2)

one_layer_graph = nir.NIRGraph(
    nodes = {
        "input" : nir.Input(input_type=np.array([1])),
        "linear1" : nir.Linear(weight=weight1),
        "lif" : nir.LIF(tau=np.array(2*[1]),
                        r=np.array(2*[1]),
                        v_leak=np.array(2*[0]),
                        v_threshold=np.array(2*[1])),
        "linear2" : nir.Linear(weight=weight2),
        "output": nir.Output(output_type=np.array(2*[1]))
    },
    edges = [
        ("input", "linear1"),
        ("linear1", "lif"),
        ("lif", "linear2"),
        ("linear2", "output")
    ]
)

```

Listing 1: Example of a NIRGraph. This graph represents a network with two LIF layers, consisting of two neurons each. The layers are fully connected by two synapse layers.

In NIR, neural networks are graphs, represented as a dictionary (`nodes`) and an array of tuples (`edges`). An example of such a graph is given in listing 1. The `edges` defines the connections between the layers while the `nodes` contains all used layers, e.g. a Leaky-Integrator and a CubaLIF. They are represented as `NIRNode`. There always has to be an `Input` and an `Output` node. NIR currently defines a set of 16 `NIRNode`'s: nodes of which neural networks should be built of. The documentation [11] also states that the size of parameter arrays equals the size of the layer. But in Norse, the size of neuron layers is given by the output size of the previous synapse layer. SpiNNaker on the other side uses the `input_type` attribute of the `NIRNode`, as it is intended by NIR.

3 Results

3.1 From NIR to jaxsnn: The `from_nir()` Function

First the translation from NIR to jaxsnn is established. For the beginning not all different types of layers have to be implemented. Therefore we chose to implement the CubaLIF layer, since it's representing LIF neurons in BrainScaleS-2 in most experiments conducted so far. Also we need synapse layers to connect the neuron layers and set the weights. The mock example we choose is set in NIR first (lst. 1). Therefore the nodes and the edges have to be defined.

Algorithm 1 Processing of each node in the `from_nir()` function

```
for node in node_list do
    if node is instance of nir.CubaLIF then
        extract neuron parameters from node
        create init() and apply()
        append init() and apply() to init_apply_list
    else if node is instance of nir.Linear then
        extract weights from node
        append weights to weights_list
    else if node is instance of nir.Input or nir.Output then
        pass
    else
        raise NotImplementedError
    end if
end for
```

The first thing is to build a graph out of the edges. With that graph now every node can be processed iteratively (alg. 1, starting with the `Input` node. Since the NIR package does not come with much internal functionality to ensure that objects are declared as expected, these things have to be handled by the `from_nir()` function. Two examples of such cases are

1. `edges` only contains tuples of length two. This means that one edge connects exactly two nodes.
2. These tuples only refer to nodes which are previously defined in the `nodes` dictionary. This avoids getting undefined nodes.

When getting to a synapse layer, like a `Linear` or an `Affine` node, the weights are saved in a list. If the current layer is a neuron layer, like a `CubaLIF` node, the parameters are read out and put into the jaxsnn pendant `LIF`. There are some parameters like the maximum simulation time $t_{\max}$, which have to be saved in a separate configuration object since they are not represented in NIR. The `LIF` function gives us the `apply()` function, which gets spiking input data and returns spiking output data. If there are several neuron layers, all the apply functions are saved in a list.

If the loop reaches the end of the node list, the final `init_fn()` function is defined such that it returns the list of weights and the `apply()` functions get composed to one single `apply_fn()`. These two functions now represent the model in jaxsnn.

```python
from jaxsnn.event.from_nir import from_nir, ConversionConfig

cfg = ConversionConfig()
init_fn, apply_fn = from_nir(one_layer_graph, cfg)
```

Listing 2: example for the translation of a NIRGraph into a jaxsnn representation, consisting of an `init_fn()` and an `apply_fn()`. The `ConversionConfig` objects carries simulation parameters like t_{max}

3.2 Validating the Translation from NIR to jaxsnn

To validate that the translation of models from NIR to jaxsnn works we train a SNN in Norse, convert it to NIR by the already implemented `norse.to_nir()` function and finally convert it to jaxsnn where its accuracy can be determined. If both the test accuracies match, we can assume that the conversion works just fine. This experiment is performed on the YinYang dataset. This one is a classification task which is nice for prototyping since it can be solved with small networks.

3.2.1 Training a Norse SNN on the YinYang Dataset

The Norse model is build such that the used modules are convertible to NIR, since not all modules provided by Norse can be translated to NIR. That means we use module `norse.LIFCell` which corresponds to `nir.CubaLIF` and we also use `torch.nn.Linear` as synapse layers (lst. 3). This model is then trained on the YinYang dataset. Since the test accuracy is very unstable, we cannot demand $\tau_{\text{mem}} = 2\tau_{\text{syn}}$. Instead a configuration with $\tau_{\text{mem}} = \tau_{\text{syn}}$ is chosen. Through multiple iterations and experimentation, the optimal parameters for the Norse model were determined (1). After a training with 2048 samples over 40 epochs we reach a accuracy of 80%. The training was done with surrogate gradients.

```python
model = norse.SequentialState(
    torch.nn.Linear(input_size, hidden_size, bias=False),
    norse.LIFCell(params, dt=dt),
    torch.nn.Linear(hidden_size, output_size, bias=False),
    norse.LIFCell(params, dt=dt)
    )
```

Listing 3: Norse model of SNN which performs the YinYang task

Table 1: CubaLIF parameters (`params`) of the hidden layer and output layer. The time constants are given in seconds and the voltages in volt

parameter	value
τ_{mem}	$2 \cdot 10^{-5}$
τ_{syn}	$2 \cdot 10^{-5}$
V_{th}	1.0
V_{leak}	0.0
V_{reset}	0.0
t_{target}	$2.6 \cdot 10^{-5}$
$t_{\text{no target}}$	$3.6 \cdot 10^{-5}$

3.2.2 Analytical TTFS Solver for $\tau_{\text{mem}} = \tau_{\text{syn}}$

To get the next spike, meaning the next time when the membrane potential exceeds the threshold, a solver is needed. The most general approach would be a numerical solver such like a forward-Newton-solver. For two cases there are analytical solutions proposed by Göltz et al. [9]. If both time constants are the same, the first-spike time T is given by

$$T = \tau_{\text{mem}} \left\{ \frac{b}{a_1} - \mathcal{W}\left[-\frac{g_l V_{th}}{a_1} \exp\left(\frac{b}{a_1}\right) \right] \right\}, \tag{3}$$

where $\mathcal{W}[x]$ describes the Lambert $\mathcal{W}$ Function.

In the original implementation the time-to-first-spike is calculated starting from $t_0 = 0$ where also $V_0 = I_0 = 0$ but the jaxsnn implementation can start from any point in time t_i and takes the corresponding amplitudes $V(t_i)$ and $I(t_i)$ for calculations. By that, previous spike times do not have to be remembered. Instead the neurons state, consisting of current and voltage, is evolved and used for calculation of the TTFS. Because of this difference in the two implementations using the original expression for the variable b is less convenient and a new expression must be derived, whereas a_1 is already defined in the jaxsnn implementation.

This can be solved as follows[1]: The differential equation

$$0 = \tau_{\text{syn}} \dot{V} + V - \frac{1}{g_l} I \tag{4}$$

was also used by Althaus [10] to derive the closed form for the $\tau_{\text{mem}} = 2\tau_{\text{syn}}$ case. Now we set $t_i = 0$ and name $V_0' = V(t_i)$ and $I_0' = I(t_i)$, since we are only interested in the time difference to the next spike. We expect $V(t)$ and $I(t)$ to look like

$$V(t) = \frac{t}{g_l \tau_{\text{syn}}} a_1 \exp\left(-\frac{t}{\tau_{\text{syn}}}\right) - \frac{1}{g_l} b \exp\left(-\frac{t}{\tau_{\text{syn}}}\right). \tag{5}$$

$$I(t) = I_0' \exp\left(-\frac{t}{\tau_{\text{syn}}}\right) \tag{6}$$

[1]First derived and proposed by Florian Fischer

Inserting $V(t)$ and $I(t)$ into equation eq. (4) gives us

$$a_1 = I'_0. \tag{7}$$

With V'_0 we then get

$$b = -g_l V'_0. \tag{8}$$

By applying the boundary condition $V(T) = V_{\text{th}}$, where T is the time to the next spike, this gives

$$0 = V_{\text{th}} \exp\left(\frac{T}{\tau_{\text{syn}}}\right) - g_l V'_0 - I'_0 \frac{T}{\tau_{\text{syn}}}. \tag{9}$$

This expression is found very similar in the derivation of Göltz et al. [9]. Following their derivation and applying it to our case we get

$$T = \tau_{\text{mem}} \left\{ -\frac{g_l V'_0}{I'_0} - \mathcal{W}\left[-\frac{g_l V_{th}}{I'_0} \exp\left(-\frac{g_l V'_0}{I'_0} \right) \right] \right\} \tag{10}$$

the new solver can be implemented. It is important to tackle all edge-cases with `jax.lax.cond()` to make sure that the solver does not return `nan`'s or `inf`'s. One condition is that the argument of the Lambert $\mathcal{W}$ Function is equal or larger than $-e^{-1}$, since the real branch of the function is not defined for smaller values. The other condition is that the spike time T is positive. By that, the solver is able to find the next spike time for the case $\tau_{\text{mem}} = \tau_{\text{syn}}$.

3.2.3 Validating the NIR-to-jaxsnn Transformation

With the TTFS solver and the NIR-model set it can now be translated. This translation process takes place in the new `from_nir()` function, which returns the functions `init_fn()` and `apply_fn()`. Both must now be used to test for the networks accuracy at performing the task. Testing the network in jaxsnn can done by using the `loss_and_acc()` function. The test set is generated by `yinyang_dataset()` generator in jaxsnn. The loss is determined by the MSE between the predicted and the target spike time. This finally yields an accuracy of 79%, which is very close to the test accuracy in Norse.

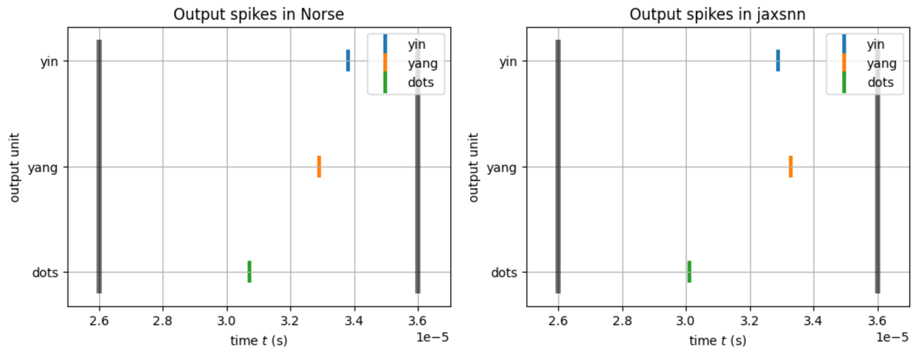

Figure 1: Comparison between the spike times in Norse (left) and in jaxsnn (right). The gray lines indicate the time for the right class (left) and the time for the wrong class (right). On the y-axis are the three different output neurons.

We can also look at the spike times in the output layer of a single example (fig. 1). By looking at the first spike times we see little differences between for both models. But the decisive spike - the first one over all - has a big difference in both models, so both models will classify similar in this case.

4 Summary

During this internship, part of the translation of spiking neural networks from NIR to jaxsnn was implemented. It is now possible to translate linear feed-forward models from NIR to jaxsnn, as long as they only consist of `nir.CubaLIF` and `nir.Linear` nodes. This translation is achieved by looping over nodes of the NIRGraph and converting them one by one to a representation in jaxsnn. In the end, the list of jaxsnn representations of nodes is composed into a `init()` and a `apply()` function for the whole network.

The functionality of the translation process was demonstrated through the application of an Norse-to-jaxsnn example. Therefore an SNN was defined in Norse to solve the YinYang task. After training in Norse the model achieved a test accuracy of 80%. After translating this model via NIR to jaxsnn the test accuracy yielded 79%. Both values match approximately. The little difference is assumed to be caused by insufficient precision of the numerical solver used in Norse since the spike times in jaxsnn are computed exactly via analytical expressions. Compared to previous experiments [1], the accuracy in Norse remains below expectations. It is not fully understood if the relatively low accuracy and also its significant fluctuations during training are connected to issues in our Norse training procedure.

The corresponding TTFS solver needed for testing in jaxsnn has also been implemented. This extends the functionality of jaxsnn by supporting the case $\tau_{\text{mem}} = \tau_{\text{syn}}$ in jaxsnn, as derived by Göltz et al. [9].

As part of this work, the architecture of NIR was analyzed and we looked at its strengths and weaknesses. While being flexible and versatile, e.g. by not requiring special data types for weight matrices, this can easily become a weakness if different frameworks adopt varying approaches. For this reason, it may be considerable to specify the data type of variables or to include internal checks to ensure that attributes such as matching input and output shapes are consistent.

5 Outlook

Now as basic principles of the translation are completed, we can progress with the conversion of more complex and also more diverse models. One of the next steps will be implementing the translation for more types of nodes, like the `nir.LIF` or the `nir.Affine`. This may be a little less straightforward since there is no exact counterpart for the nodes on the BrainScaleS-2 hardware. Also we want to enable the conversion for more different topologies besides the simple linear networks, such as recursive networks. In addition to the `from_nir()` function we plan to implement a `to_nir()` function, to also allow exporting models defined in jaxsnn to be trained or tested on other platforms. It is also planned to annotate NIR nodes with jaxsnn functions. By this, NIR would be extended by a numerical backend and thus be able to do the forward or backward pass itself.

References

[1] Eric Müller et al. "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].

[2] 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.

[3] Philipp Spilger et al. "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.

[4] Christian Pehle et al. "The BrainScaleS-2 Accelerated Neuromorphic System with Hybrid Plasticity". In: *Front. Neurosci.* 16 (2022). ISSN: 1662-453X. DOI: 10.3389/fnins.2022.795876. arXiv: 2201.11063 [cs.NE]. URL: https://www.frontiersin.org/articles/10.3389/fnins.2022.795876.

[5] 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).

[6] Jens E. Pedersen et al. "Neuromorphic Intermediate Representation: A Unified Instruction Set for Interoperable Brain-Inspired Computing". In: 2023. DOI: 10.48550/arXiv.2311.14641.

[7] 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.

[8] James Bradbury et al. *JAX: composable transformations of Python + NumPy programs*. Version 0.3.25. 2022. URL: http://github.com/google/jax.

[9] Julian Göltz et al. "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.

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

[11] Steven Abreu et al. *NIR - Neuromorphic Intermediate Representation*. URL: https://neuroir.org/docs.