Generalizing and Refactoring jaxsnn

Internship report

Florian Fischer
Supervisors: Elias Arnold, Philipp Spilger and Eric Müller
Department of Physics and Astronomy
Ruprecht-Karls-University Heidelberg

October 2, 2024

Abstract

The main objective of this internship was to become familiar with the Python library *jaxsnn* and make improvements to it. *Jaxsnn* is a state-of-the art python framework for machine learning with spiking neural networks (SNNs). It stands out because its event-based simulation of SNNs differs from most other libraries which follow a time-grid-based approach. Moreover, it supports the execution of simulations and in-the-loop training on the neuromorphic *BrainScales-2* system. The modifications made to *jaxsnn* included generalizations in terms of network composability as well as abstractions and simplifications of the code.

Contents

1 Introduction **1**

2 Theory **2**
2.1 Spiking Neural Networks 2
2.2 JAX 4
2.3 jaxsnn 4

3 Methods **6**

4 Summary and Outlook **8**

1 Introduction

While machine learning based on artificial neural networks has gained significant attention in recent years, machine learning with spiking neural networks (SNNs) has also become an increasingly important research topic. One of the advantages of SNNs is that they are more biologically realistic and thus might also help us understand how the brain works. Moreover, SNNs are well-suited for execution on hardware, which promises energy-efficient execution by implementing bio-inspired dynamics [11]. Additionally, advancements in training algorithms of SNNs like the EventProp algorithm [12] and backpropagation through time have made them an attractive option. This underlines that working on neuromorphic hardware as well as SNN-inspired machine learning are very important fields of research. jaxsnn [7] sits at the intersection of these two. While it is a software library for simulating and training of SNNs, it also allows users to run experiments on neuromorphic hardware and perform in-the-loop training of network parameters. The event-based simulation has also some advantages over a time-grid-based one because the times of spikes can be continuous and don't have to be discretized into time bins as can be seen in figure 1.

First, the underlying theory that jaxsnn builds upon will be discussed. Then, the improvements made to jaxsnn during this internship will be explained and finally a short outlook on future work will be provided.

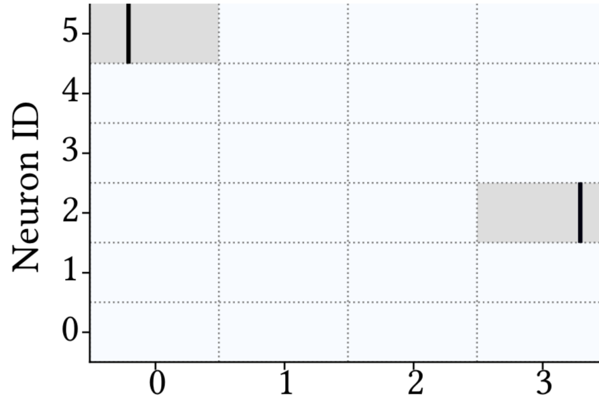

Figure 1: Continuous spike times vs discrete spike times. Taken from [7].

2 Theory

2.1 Spiking Neural Networks

Spiking Neural Networks (SNNs) are a special type of artificial neural networks (ANNs). In contrast to traditional ANNs, they process information using spike events in a way that aims to replicate the functioning of the brain. When it comes to the simulation of SNNs, one of the most important choices to consider is the choice of the neuron model. A common example is the Leaky Integrate-and-Fire neuron which is a very simple yet powerful mathematical model to describe neuronal dynamics. It goes back to Louis Lapicque who introduced a mathematical description in 1907 [6].

The Leaky Integrate-and-Fire Neuron

The following explanations about LIFs are inspired by Chapter 1.3 of the book *Neuronal Dynamics* by Wulfram Gerstner [3]. The basic idea of LIFs is that starting from an initial membrane state the incoming current is integrated over time, leading to a change in the neurons membrane potential. However, the membrane potential naturally "leaks" over time and converges towards a resting potential in the absence of inputs. This temporal evolution of the membrane potential can be represented by a parallel circuit consisting of a capacitor with capacitance C and a resistor with resistance R. By using some basic equations for the current $I(t)$ and the voltage $u(t)$ one can derive the following linear differential equation for the circuit

$$\tau_{\mathrm{m}} \frac{du}{dt} = -[u(t) - u_{\mathrm{rest}}] + RI(t). \tag{1}$$

In the biological context $I(t)$ corresponds to the synaptic current caused by pre-synaptic neurons and $u(t)$ to the membrane potential of the neuron. Moreover,

u_{rest} refers to the resting membrane potential and $\tau_{\mathrm{m}} = RC$ to the membrane time constant.

When the membrane potential reaches a certain threshold ϑ due to the accumulated input, the neuron fires an action potential (spike) at the firing time $t^{(f)}$. The sequence of firing times for one neuron i forms the spike train $S_i(t)$ which is given by

$$S_i(t) = \sum_f \delta(t - t_i^{(f)}), \qquad \text{where} \qquad t^{(f)}: \quad u(t^{(f)}) = \vartheta. \tag{2}$$

After the neuron fires, the membrane potential is reset to a potential u_{r} which is typically below the resting potential

$$\lim_{\delta \to 0; \delta > 0} u(t^{(f)} + \delta) = u_{\mathrm{r}}. \tag{3}$$

In the human brain, neurons are connected to each other via synapses to allow the propagation of action potentials. Similarly, a single LIF neuron can be extended into a network of LIF neurons by linking the spikes of a presynaptic neuron to the synaptic current of the postsynaptic neuron. The method of linking spikes to synaptic currents depends on the model, but one common approach is given by the following equation:

$$I(t) = \sum_i w_i [\epsilon * S_i(t)], \qquad \text{where} \qquad \epsilon(t) = \theta(t) \exp\left(-\frac{t}{\tau_{\mathrm{s}}}\right). \tag{4}$$

The synaptic weights w_i represent the strength of the connection between individual neurons and the synaptic kernel ϵ determines the shape of the postsynaptic current caused by a single spike. Moreover, τ_{s} is called the synaptic time constant.

Machine Learning with SNNs

In order to use SNNs for machine learning, the input has to be encoded into spikes. The most popular encoding techniques are rate coding and temporal coding. While rate coding uses the spike frequency to encode information, temporal encoding conveys information via the precise timing of individual spikes [2]. After the simulation of the SNN the spike output also has to be decoded. One way to do this is by using Leaky Integrators as output neurons. This adapted model of the LIF neuron doesn't fire spikes and just records the membrane potential which can then be used to make predictions by taking the maximum voltages for example. But spikes can also be used for instance by using the spike times of the individual output neurons to make predictions.

However, optimizing the weights based on a loss function turns out to be rather difficult. Standard backpropagation through time and temporal discretization of the SNN to compute the gradients of the loss function with respect to the network's parameters is not directly applicable. This is because the activation functions that produce discrete spikes are not differentiable. One solution

to this is using surrogate gradients [8] which replace the activation function for the backward pass with a smooth and differentiable approximation like a sigmoid function

$$\sigma(x) = \frac{1}{1 + e^{-x}} \tag{5}$$

Exact gradients

One way to compute exact gradients was presented by Göltz et al. in [4]. There they derive an analytical solution for the time of the next threshold crossing for the two specific cases $\tau_\mathrm{m} = \tau_\mathrm{s}$ and $\tau_\mathrm{m} = 2\tau_\mathrm{s}$. Because the derived formulas are differentiable they can be used to calculate exact gradients when the loss depends on the first spike times of the output neurons.

Another more general method is the EventProp algorithm [12]. For that method a set of adjoint equations has to be evolved backwards in time which can then be used to calculate exact gradients.

2.2 JAX

JAX [1] is a Python library built on top of NumPy for high-performance numerical computing, developed by Google. But it also provides composable function transformations for efficient and parallelized computation.

JAX uses XLA (Accelerated Linear Algebra) to optimize the code. It generates machine code that can run on CPUs, GPUs or TPUs to speed up performance. In order to achieve this, the code is traced during execution. Then for subsequent calls, JAX checks the cache, and the function only has to be recompiled if the input shapes or types have changed. This makes it necessary to pay special attention to implementing the control flow in a way that is compatible with the tracing. For this reason, JAX provides built-int functions like `jax.lax.cond` or `jax.lax.scan` which are substitutions for if statements and for-loops respectively. While library calls get just-in-time (JIT) compiled by default, for custom Python functions `jax.jit` can be used to compile them into XLA-optimized kernels.

Furthermore, automatic vectorization of functions is supported via the `jax.vmap` transformation. This allows defining functions on single elements and then to perform automatic batching on them. Going one step further with `jax.pmap`, functions can also be parallelized across multiple devices.

Additionally, JAX supports automatic differentiation allowing the differentiation of multi variable vector-valued functions as well as higher order differentiation. It is also possible to define operations with custom forward and backward functions.

2.3 jaxsnn

jaxsnn [7] is a Python library for simulating SNNs and optimizing for bio-inspired machine learning. Moreover, it is designed to work with event-driven neuromorphic hardware and perform simulations on hardware as well as in the

loop training. It is split up into two parts: On one hand there is a discrete time step based implementation which was inspired by Norse [10] and on the other hand there is an event-based implementation. In the following the event-based implementation will be explained.

The layers of a given network are simulated sequentially. For each layer a simulation step is performed for a certain number of times which has to be set when the network is defined. This number of steps per layer corresponds to the number of events (input/internal spike or no spike) that are simulated for a given layer.

At each step, the next event is found first. For this, the time of the next spike in the layer is calculated analytically using the solutions from [4] or numerically using a root solver. Then this time is compared to the time of the next input spike and the next event is chosen. If both times are bigger than a predefined time t_{late} no event will be simulated. Next, the discrete transition dynamics are applied to the neuron states based on what type of event occurred. At the end of the step the event is returned. The output of one layer then becomes the input of the next layer. The data flow of the step function is also visualized in figure 2.

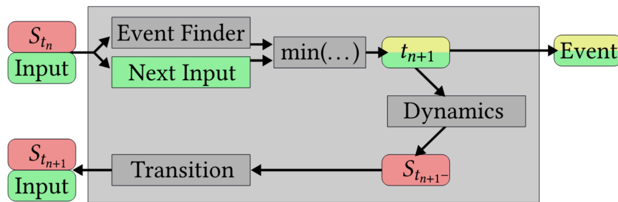

Figure 2: Step function data flow. Taken from [7].

For training the weights a loss function, for example based on the output spike times, has to be defined. If inside the step function the analytical solution was used one can use automatic differentiation to calculate the gradients. However, for the general case backpropagation with EventProp can be used. To achieve this, after the forward pass the adjoint dynamics are simulated backwards in time. This is done using a backwards step function which works similarly to the normal one.

jaxsnn features multiple examples for training on the Yin-Yang dataset [5]. Moreover there is also an example for training on the BBS-2 System [9]. In [7] high accuracies on the Yin-Yang dataset with EventProp backpropagation are reported with an accuracy of $(98.2 \pm 0.2)\%$ in simulation and $(94.8 \pm 0.2)\%$ on BBS-2 with in-the-loop training.

3 Methods

Part of the internship was spent understanding the necessary theory from the previous section and becoming familiar with the tools. Following that, the main objective was to improve the code base of jaxsnn. While jaxsnn was mostly in a functional state, some parts were a little unstructured. At first, some refactoring of the code was done to improve its maintainability and accessibility to people unfamiliar with the library. Then also some generalizations and improvements were made. While not every minor change will be mentioned, the more significant changes are explained below.

Generalization of the serial function

The serial function lets the user compose layers of neurons to networks by creating one init/apply pair. Previously, there were separate serial functions for discrete simulation, normal event-based simulation and event-based simulation with known spike times (needed when the spike times from a hardware run should be used). This functionality was unified to one single serial function for all the aforementioned cases. In order to achieve this, some adjustments to the code for the discrete simulation were made to sequentially simulate the layers instead of simulating the whole network at each time step. Moreover, for the apply function a `carry` value was introduced which allows the apply functions to carry values between layers. This is useful for keeping track of the layer index and in the case of the event-based simulation the starting index of neurons in the layer (because absolute indexing is used). Additionally, an `external` value was added which lets the user pass in the known spikes if necessary which can then be handled by the apply functions of the layers.

Another goal was to make the `serial` function composable, allowing for the definition of smaller models which then can be used to build more complex models. To achieve this nesting, the signature of the apply function which is returned by `serial` had to be matched with the signature of apply functions from individual layers. This way apply functions of entire networks can be passed into `serial` and handled the same way as layer apply functions. Moreover, after some consideration, the lists of parameters (weights/recording) which are used by the apply functions were forced to be flat to allow uniformity between equivalent models that are build only from single layers and models which already use smaller models as building blocks. Moreover, this simplifies the handling of results after the training because nested lists of unknown depth don't have to be handled. The adjusted `serial` function can be seen in figure 3.

Filtering of wrong input spikes

In the previous implementation in the scan of the step function input spikes as well as non-spiking events where added to the output. This led to the problem that those input spikes could stay in the input queue for all following layers.

```python
def serial(*layers: SingleInitApply) -> InitApply:
    init_fns, apply_fns = zip(*layers)

    def init_fn(
        rng: jax.random.KeyArray,
        input_size: int,
    ) -> Tuple[int, List[Weight]]:
        ...
        return input_size, weights

    def apply_fn(
        weights: List[Weight],
        spikes: EventPropSpike,
        external: Optional[Any] = None,
        carry: Optional[Any] = None,
    ) -> Tuple[Any, List[Weight], EventPropSpike, List[EventPropSpike]]:
        ...
        return carry, future_weights, spikes, recording

    return init_fn, apply_fn
```

Figure 3: Adjusted `serial` function.

While this was technically already solved by checking in the `transition` function if the input spikes are really from the previous layer, this is still undesirable because, with increasing numbers of layers, a lot of simulation steps just become unnecessary events where nothing happens. To solve this issue, a new function was implemented which filters out wrong spikes before the scan of the `step` function (figure 4).

Unifying dataset functionality and separating encoding

Another objective was to unify the dataset functionality for the discrete and event-based case. To achieve this, the previous, similar implementations of a circle, constant, linear and Yin-Yang dataset from the event and discrete module were combined together. For the event module the encoding used to be inside the dataset functionality. Therefore, based on the encoding implementation inside the dataset functionality, new encoding functions were implemented to allow spatio-temporal encoding of inputs and also the temporal encoding of targets. This encoding functionality was also abstracted to work for arbitrary dimensions. For the discrete case, encoding was part of the network definition with `serial`. This was changed to ensure that, for both event-based and discrete simulations the dataset setup and encoding are handled separately from the SNN.

Additionally, a data loader was implemented which uses `JAX Pytree` func-

```python
def apply_fn(...):
    ...
    # Filter out input spikes which are not from previous layer
    input_spikes = filter_spikes(input_spikes, layer_start - input_size)

    step_state = StepState(
        neuron_state=initial_state,
        time=0.0,
        input_queue=InputQueue(input_spikes),
    )
    _, spikes = jax.lax.scan(
        step_fn,
        (step_state, this_layer_weights, layer_start),
        np.arange(n_spikes)
    )
    ...
    return spikes, ...
```

Figure 4: Filtering of input spikes.

tionalities to permute the samples randomly and split them into batches of desired size. It works for samples which are stored in arrays as well as more complex pytree structures like `EventPropSpikes`.

4 Summary and Outlook

Overall this internship was very helpful to gain an understanding of the theory and the libraries like *JAX* and *jaxsnn*. Many of the changes were chore tasks to improve jaxsnn's maintainability in the future. But also some important improvements were made like generalizing the `serial` function and adapting it to support composability. And also the filtering of wrong input spikes is critical when thinking about scaling up the network sizes. All the changes were also verified using the existing examples and tests. In the following bachelor thesis jaxsnn will be extended to allow the use of a wider range of network topologies and also delays will be incorporated. This is important to make jaxsnn a more flexible library for simulating SNNs and ML-inspired training of SNNs.

References

[1] James Bradbury, Roy Frostig, Peter Hawkins, Matthew James Johnson, Chris Leary, Dougal Maclaurin, George Necula, Adam Paszke, Jake VanderPlas, Skye Wanderman-Milne, and Qiao Zhang. JAX: composable transformations of Python+NumPy programs, 2018.

[2] 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. *Proceedings of the IEEE*, 111(9):1531–1556, September 2023.

[3] Wulfram Gerstner, Werner M. Kistler, Richard Naud, and Liam Paninski. *Neuronal Dynamics: From Single Neurons to Networks and Models of Cognition*. Cambridge University Press, Cambridge, UK, 2014.

[4] J. Göltz, L. Kriener, A. Baumbach, S. Billaudelle, O. Breitwieser, B. Cramer, D. Dold, A. F. Kungl, W. Senn, J. Schemmel, K. Meier, and M. A. Petrovici. Fast and energy-efficient neuromorphic deep learning with first-spike times. *Nature Machine Intelligence*, 3(9):823–835, September 2021.

[5] Laura Kriener, Julian Göltz, and Mihai A. Petrovici. The yin-yang dataset. In *Neuro-Inspired Computational Elements Conference*, NICE 2022, page 107–111. ACM, March 2022.

[6] Louis Lapicque. Recherches quantitatives sur l'excitation électrique des nerfs traitée comme une polarisation. *Journal de Physiologie et de Pathologie Générale*, 9:620–635, 1907.

[7] Eric Müller, Moritz Althaus, Elias Arnold, Philipp Spilger, Christian Pehle, and Johannes Schemmel. jaxsnn: Event-driven gradient estimation for analog neuromorphic hardware. In *2024 Neuro Inspired Computational Elements Conference (NICE)*, volume 2022, page 1–6. IEEE, April 2024.

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

[9] Christian Pehle, Sebastian Billaudelle, Benjamin Cramer, Jakob Kaiser, Korbinian Schreiber, Yannik Stradmann, Johannes Weis, Aron Leibfried, Eric Müller, and Johannes Schemmel. The brainscales-2 accelerated neuromorphic system with hybrid plasticity. *Frontiers in Neuroscience*, 16, February 2022.

[10] Christian Pehle and Jens Egholm Pedersen. Norse - A deep learning library for spiking neural networks, January 2021. Documentation: https://norse.ai/docs/.

[11] Amirhossein Tavanaei, Masoud Ghodrati, Saeed Reza Kheradpisheh, Timothée Masquelier, and Anthony Maida. Deep learning in spiking neural networks. *Neural Networks*, 111:47–63, March 2019.

[12] Timo C. Wunderlich and Christian Pehle. Event-based backpropagation can compute exact gradients for spiking neural networks. *Scientific Reports*, 11(1), June 2021.