Introducing trainable delays in JaxSNN (in a Minimal Example and the Yin-Yang dataset)

Internship Report

Rabea Fasel
Supervisors: Elias Arnold, Eric Müller

Chair of Neuromorphic Computing Architecture (ZITI),
European Institute for Neuromorphic Computing (EINC)

Department of Physics and Astronomy
Heidelberg University

April 30, 2026

Abstract

Spike transmission delays are a promising but still new field of research in the domain of Spiking Neural Networks, especially with focus on energy consumption and efficiency as well as modeling and training. In this internship report, the main goal is to get familiar with the concept and implementation of delays and the machine learning framework jaxsnn which will be used. For that we introduce delays in a minimal example. This implementation is then extended to the Yin-Yang dataset.
The results produced show proper functioning and trainability of the delays, with only a slight decease in accuracy. Further steps will be a optimization of the parameters of the network for smaller sizes (e.g. the parameters used in the DelGrad paper), at which better results for a implementation with delays in comparison to one without delays are expected. In addition accounting for recurrence as well as more stable sorting methods are planned.

Contents

1. Introduction

With the introduction of Large Language Models (LLMs) to the general public, neural networks have gained significantly in attention in the last years. While the main focus right now lies on Artificial Neural Networks (ANNs), the research on Spiking Neural Networks (SNNs) as a more biology oriented and energy efficient alternative becomes increasingly important[1, Sec. 1].
In this report, the focus will lie on adding trainable delays to a SNN. In the long run the aim is to introduce them in more training algorithms and establish them as a possible way to increase the expressivity of the SNNs. This in Turn might decrease the number of parameters and hence potentially its energy footprint. In the scope of this report we will implement delays, using the python library jaxsnn, and test them on a minimal example and the Yin-Yang dataset.

1.1. Theoretical background

1.1.1. Spiking Neural Networks

Spiking Neural Networks (SNNs) are an alternative approach to classical ANNs. Operating in a continuous time setting, neurons in SNNs transport information via discrete electrical signals, so called spikes. Inherently, SNNs have temporal dynamics. These dynamics are closely related to those in biological Neural Networks, as the mammalian brain uses a similar event scheme for inter-neuron communication.

Event-based Simulation

Classically SNNs are set up in a time continuous context. But especially in biology the occurrence of spikes is oftentimes sparse, meaning: most of the times nothing happens. Consequently, reducing the actual computing from a time grid where after every δt the neuron state is calculated, to a setting where only actual events (spikes) are taken into account, can make data handling more efficient. The data representation is reduced from a very large and sparse matrix the size of $n \times n$ where $n = \delta t \cdot$ [duration of simulation] to a list of only the spike times.

There are several models that mimic such dynamics. For reasons of its wide spread usage and its simplicity, as well as the model existence of analytical solutions with respect to the spike times, in this work, the Leaky-Integrate-And-Fire (LIF) will be the model of choice

1.1.2. Leaky-Integrated-And-Fire Model

The Leaky-Integrate-and-Fire (LIF) models central differential equation, that describes it sub-threshold dynamics, is the following:

$$\tau_m \frac{dV(t)}{dt} + V(t) = V_{\text{rest}} + \frac{I(t)}{g_\ell}. \tag{1.1}$$

$V(t)$ describes the membrane potential of the neuron. Through integrating over the currents I induced by incoming spikes, it rises. Because of the leak term, if no incoming spikes occur for a longer time, it falls. When the membrane potential reaches a certain threshold V_{th}, it gets reset to V_{rest} the resting potential for a refractory time t_{ref} and the neuron "fires", meaning it emits a spike. In addition, the membrane potential decays with the time constant τ_m (it is "leaky").
g_ℓ is the so called leak conductance, $I(t)$ is often modeled as a exponential function, which is also what will be used here:

$$I(t) = \sum_i \Theta(t - t_i)\, \omega_i \, \exp\left(-\frac{t - t_i}{\tau_{\text{syn}}}\right) \tag{1.2}$$

Θ is the Heaviside step function, ω_i are the weights of the synapse that receives a spike at time t_i and τ_{syn} is the synaptic time constant [1, Sec. 4.2.1].

1.1.3. Exact Computation of the Time of the Next Spike and Gradients

The response function of a LIF neuron maps from a sequence of input to a sequence of output spikes. Respectively when knowing the set of the input spike times $\{t_i\}$, and the input weights $\{\omega_i\}$, it is possible to express the time of the next spike T as a function of t_i and ω_i, given the neurons state V and I. Further, under certain conditions for the synaptic time constants τ_m and τ_{syn} this function becomes analytical [2].
It should be noted that the closed-form solution of this function is what makes backpropagation and exact gradient calculation possible in the first place. Here we will focus on the case $\tau_m = 2\,\tau_{syn}$, which yields:

$$T = 2\,\tau_{syn}\,ln\left[\frac{2\,a_1}{a_2 + \sqrt{a_2^2 - 4a_1 g_l V_{th}}}\right] \tag{1.3}$$

Where g_l is the leak conductance of the membrane, V_{th} the spike threshold and a_1 and a_2 explicitly only depend on previous spike times and weights and the current neuron state I, V.

To train such a network using gradient descent, we need to define a loss $\mathcal{L}$ that is dependent on all the networks parameters θ. Then we can update the parameters θ by using the loss functions gradient $\frac{\partial \mathcal{L}}{\partial \theta}$ which by the chain rule is iteratively made up of $\frac{\partial T}{\partial \theta}$ and $\frac{\partial T}{\partial t_i}$. Because this allows to relate a change in the output of a neuron to the change of the input that is influenced by a synaptic weight, backpropagation of errors is possible in the first place. These derivatives can be computed only with the knowledge of the spike time and parameters of the network. Effectively leading to a forward and backwards pass that does not need any discretizations in time and is purely event based [3, Sec. 2].

1.1.4. Introducing Delays

Within equation 1.3, spike transmission delays can be introduced fairly easy. There are 3 types of delays: dendritic delays d_{den} on a neurons input, synaptic delays d_{syn} that act on the individual connections between 2 neurons and axonal delays d_{axo} that affect each neurons output. But effectively we only need to account for two, because we can combine d_{den} and d_{axo} since the dynamics in a neuron are invariant to shifts in time. Wether a spike arrives delayed to the input or the output spike is sent out delayed results in the same dynamic.

In the following, the focus is going to be put on axional delays. With the in 1.1.3 described definitions such delays can simply be introduced as additive constant to the original spike time t_i, that we treat as continuous variables:

$$t_i^d = t_i + d_i \tag{1.4}$$

The gradients with respect to the delays can be computed using:

$$\frac{\partial t_i^d}{\partial t_i} = 1 \quad \text{and} \quad \frac{\partial t_i}{\partial d_i} = 1 \tag{1.5}$$

Allowing full precision delays and exact training of the delays [3, Sec. 2].

1.1.5. jaxsnn

When talking about efficient data handling, it is important to think about the machine-learning framework. While previous frameworks such as hxtorch were more focused on processing dense data structures, the python library jaxsnn uses a new approach, better suited for asynchronous data like spike trains. It is built upon `JAX`[4], a numpy based library that provides an automatic numerical differentiation tool, efficient loop handling in form of vectorization and parallelization as well as function transformation, just in time compilation (JITing) and offloading to hardware accelerators such as graphics processing units (GPUs)[5, Sec. 1].
Generally jaxsnn simulates strongly connected components of a network sequentially, meaning for each strongly connected component(SCC) a simulation step is performed a certain predefined amount of times

(that is to make use of the `JAX` dynamics).

A **strongly connected component** is a set of nodes in a directed graph (like a SNN is), where from every node in the set, you can reach every other node, and where there is no larger set with the same property. In a simple feed forward net this would mean that a SCC is only one neuron.

A simulation step here means one execution of the so-called **step function**. It takes the SNN simulation from a state S_t to a state S_{t+1} while generating new events:
This next event can either be an input spike, a internal spike or a non-event event. A event is the latter if both next events, internal or input spike are non existent or too far ahead in time. To determine which of those cases happens, the time for the next internal spike has to be computed and compared to the time of the next input spike. For obtaining the next internal spike time, the so-called Time to first Spike function (TTFS-function) can be implemented in different ways, e.g. via equation 1.3, which will be used in the following examples. Which of those two events occurs first is then the next event that gets processed in the neuron, meaning the jumps in the current at input spike time, or jumps in the membrane at output spike times (resets), get applied. The outputs produced in that process will be the inputs for the next layer.

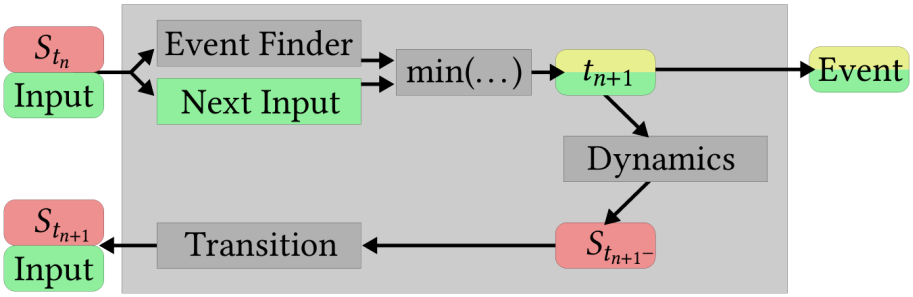


Figure 1.1.: Data flow in the step function, taken from [5]. In a state S_{t_n} the next event, be it a internal or a input spike gets chosen, on this basis a output spike gets generated if it was a internal spike and the dynamics and transitions described in section 1.1.2 get applied, so that the SNN then is in the next state $S_{t_{n+1}}$.

2. Methods

2.1. Building a simple SNN

To build a SNN with jaxsnn a few steps are necessary. In the following, the most important ones will be described briefly, with the purpose to explain what and where to alter, to introduce delays.

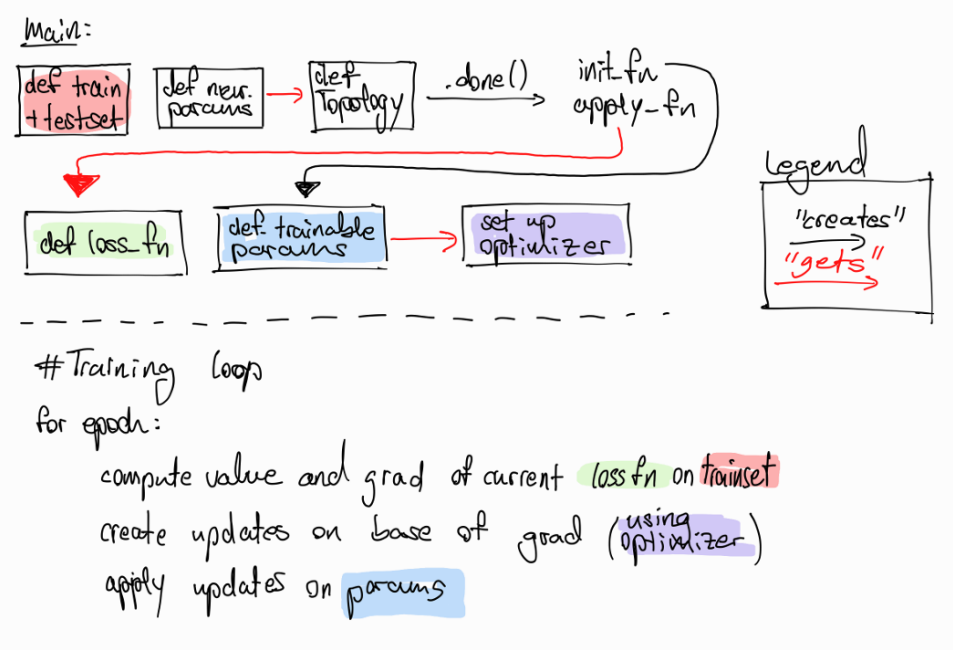


Figure 2.1.: Main steps to set up a SNN in the given jaxsnn framework and train it on a dataset: First the dataset for training and validation is defined. Afterwards parameters for the network size and properties get input into the topology function that then creates a init and a apply function. The apply function resembles a forward pass of the network and will be used in the loss function, the init function initializes the parameters of the network that actually get trained (weights, delays, etc.), who then get input into the optimizer. In a training loop then repeatedly the gradients of the loss with respect to the parameters get calculated and on base of these the optimizer creates updated parameters, that then get applied before the steps get repeated.

Topology

When defining a Topology object, we set the time window of the simulation and create and connect the modules, i.e. the layers, that the SNN will be made of. That is, a dictionary containing keys with the associated objects, each representing a layer of neurons or synapses. These are `Projection` objects that contain a generator for the layer and its previously defined parameters. The input layer is represented by `SourcePopulation` object. On the base of this info the `.done()` function of `Topology` then returns a `init_fn` function to initialize the weights of the layers and a `apply_fn` function to apply the topology to inputs, i.e. forward execute the model.

During the forward pass in `topology` the `apply function` uses the so-called `trajectory` function. It executes the defined amount of simulation steps and propagates spikes through all layers of a Strongly Connected Component. This becomes relevant when recurrence occurs in a network. In a linear feed forward net that e.g. follows `inp` → `syn1` → `lif1` each layer is its own SCC. In this case a propagation of the spikes through all layers of the SCC means only a propagation to one layer.

Loss Function

The obtained `apply_fn` is then used in the next step, to perform a forward pass, and compare its results with the target data set. This is done through a loss function. This loss can be defined in several ways, here it was opted for the Mean Squared Error (MSE) loss, that is defined by [6]:

$$\mathcal{L}_{\text{mse}} = \sum_i \left(\frac{\min(t_i^{\text{spike}}, 2t_{\max}) - t_i^{\text{target}}}{\tau_{\text{mem}}} \right)^2 \tag{2.1}$$

Where for a neuron with index i, t_i^{spike} is the actual first spike time and t_i^{target} is the target spike time. τ_{mem} is the membrane time constant (normalization factor). The spike times are clipped to a maximum of $2\tau_{\text{mem}}$ before computing the error, and the normalized differences are squared and summed across all output neurons.
For the first minimal example, a simpler version of this loss without clipping and normalization by τ_{mem} was used. Relevant is, that the loss is differentiable with respect to weights and parameters, since the spike times t_i^{target} are differentiable.

Optimizer and training

With the earlier obtained `init_fn`, trainable parameters are defined and a optimizer is set up. There are several ones you can choose from, here for both the minimal example and co-training of delays and weights Adam is chosen.
The SNN is then trained iterating over a defined amount of epochs where in each epoch the loss and its gradient with respect to the parameter that is getting trained get computed and applied.

2.1.1. Code Example

In the following a code example is shown that implements the above explained steps.

```
# 1) Define train and testset and neuron params
params = LIFParameters(v_reset, v_th, tau_syn, tau_mem)
train_batch = ({"inp": sample_input_spikes()}, sample_target_times())

# 2) Build network topology
builder = Topology(t_max)
builder.add({
    "inp": Source(inplayer_size),
    "syn": Linear(weight_mean),
    "lif": LIF(hidden_size, n_spikes, params=params),
})
builder.connect([("inp", "syn"), ("syn", "lif")])
init_fn, (apply_fn, _) = builder.done()
params = init_fn(rng)

# 4) Define loss function
def loss_fn(params, batch):
    inputs, targets = batch
    outputs = apply_fn(inputs, params)
    spike_times = first_spike(outputs["lif"].event, n_outputs=3)
    return jnp.sum((spike_times - targets[0])**2)

# 5) Set up optimizer
optimizer = optax.adam(learning_rate=2e-5)
opt_state = optimizer.init(params)

# 6) Training loop
for epoch in range(200):
    loss, grad = jax.value_and_grad(loss_fn)(params, train_batch)
    updates, opt_state = optimizer.update(grad, opt_state)
    params = optax.apply_updates(params, updates)
```

Listing 2.1: Simplified training example illustrating the principle of optimizing a spiking neural network by using topology to set up a neural network with all parameters and the defined shape, defining a loss function, setting up an optimizer and training the network with said optimizer in a training loop.

When defining the dataset we train on, we encode our data in **spikes**. Those spikes are objects that contain info about the spike time, the index of the neuron that spiked, the associated current and the index of the layer where the spike occurred and wether it was a internal spike.

`LIF()`creates a LIF layer. It does that by returning a `Population` object, that has the generator function that builds the layer based on input connections, delays, and chosen backpropagation strategy in addition to some other parameters needed.

The **`generator`** that is called by `.done()` generates the initialization (init) and apply functions for a neuron layer among others. It builds layer-specific step functions and forward/backward dynamics depending on the topology and backpropagation method.

3. Results and Discussion

3.1. Implementing Delays

This section will outline the changes implemented in jaxsnn in order to incorporate learnable delays. Mainly the focus is on writing a alternative method to the `LIF()` method.

3.1.1. `DLIF`

To introduce delays, first of all we have to give each neuron i in the layer a additional parameter d_i. Where beforehand the init function returned `None` because a neuron had no trainable parameters, it will now initialize and return trainable delays. This is done by using a Random Number Generator (RNG) that produces delays with desired mean and standard deviation. In addition the delays are clipped to be non negative, since a spike can not be emitted before it was caused.

3.1.2. `dstep`

Now that we have a variable to store and alter our delays, we also need to apply them in our forward pass. This is done in the step function, that `dlifs` `generator` function will return. For this another module `dstep` was written. The step function will determine the next event at t_{dyn} by comparing the times of the next input event and the next internal event computed by the TTFS function (eq. 1.3) and update the neurons state accordingly. That means to integrate the differential equations from the current time to t_{dyn}. Additionally a list of output spikes gets created that the neuron emits when the threshold voltage gets surpassed.
The delays get applied by creating a `t_dyn_delayed` where t_{dyn} for no event or an input event stays the same, but t_{dyn} for a spike happening gets altered to $t_{dyn} + max(0, d_i)$. The `max()` function is used to bottom out the delay at 0, since negative delays make no sense. (A event can never happen before its cause.) It is not yet sure wether that is the best way to avoid having negative delays, since some delays might end up being stuck at zero because of gradients being computed with momentum still being negative.
When now returning the output spike objects, we return our `t_dyn_delayed` times as the times of the output spikes of that layer.

3.1.3. Sorting the Spikes by Time

Another effect we have to consider is, that when different neurons in a layer have different delays $d_1 < d_2$, two spikes s_a and s_b that might have been input at times $t_{s_a} < t_{s_b}$ and thus get processed in that same order, might end up producing output spikes with times $t_{out;s_a} > t_{out;s_b}$ so when storing the spikes in the order of them getting processed, it no longer necessarily means that they are in the correct order with respect to their times, which is assumed for further processing in later layers.
Consequently sorting the output spikes before they get handed over to the next layer is necessary. This happens in the `trajectory` function. The sorting is applied once after all spikes of a layer have been processed, by using the `Spike.sort()` method that already existed.

3.2. Setup of the Minimal Example

To test wether the above alterations are sufficient to simulate and train a SNN with delays, first a minimal example with one layer containing 3 neurons is tested.
The weights are initialized with a value that ensures that all neurons produce a output spike if they receive a input spike, since the interest lies on the delays whose behavior can only be observed when spikes happen.

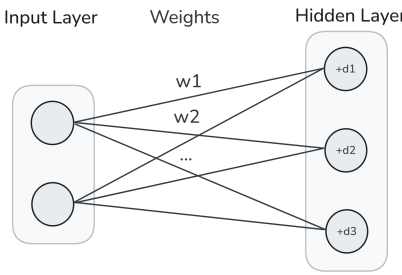


Figure 3.1.: Architecture of the SNN used for the minimal example

```python
# Topology
builder = Topology(t_max)
builder.add({
    "inp": Source(inplayer_size=2),
    "syn": Linear(weight_mean=3,weight_std=0.5),
    "lif": LIF(hidden_size=3, n_spikes=10, params=
        params),
})
builder.connect([("inp", "syn"), ("syn", "lif")])
init_fn, (apply_fn, _) = builder.done()
params = init_fn(rng)

# Loss function
def loss_fn(params, batch):
    inputs, targets = batch
    outputs = apply_fn(inputs, params)
    spike_times = first_spike(outputs["lif"].event
        , n_outputs=3)
    return jnp.sum((spike_times - targets[0])**2)

# 4) Set up optimizer
optimizer = optax.adam(learning_rate=2e-5)
opt_state = optimizer.init(params)
```

Listing 3.1: Code example for the setup of the minimal example.

The goal here was to see that the implemented delays actually changed the output spike times, as well as that they are trainable. For that purpose a training set containing only 2 input spikes was defined. In addition some target times larger than the original spike times were introduced and the SNN was trained by only optimizing the delays and keeping the weights constant.
It is expected to deserve delays in the spike times and after training it is expected for them to be close to the values of the training sets delays.

3.3. Analytical Yin-Yang with delays

After confirming the general functioning of the implemented delays, they were introduced in the preexisting example of training on the YinYang dataset with the analytical analytical approach, meaning the usage of the analytical form for the TTFS solver. It builds and trains a SNN on the Yin-Yang dataset [7], using 2 layers of lif neurons, one hidden layer with variable size and one output layer containing 3 neurons. The Yin-Yang dataset classifies data in 3 categories, Yin, Yang or the "eyes". The dataset gets generated via a already existing generator `yinyang_dataset()` from `jaxsnn.base.dataset` as x and y data that then get encoded into spikes, we will train with. The principle of the classification is that the net only has 3 output neurons that each represent one group of data, and to whichever neuron spikes first, the data point gets classified. Previous experiments in jaxsnn were conducted with a layer size of around 100 hidden neurons.

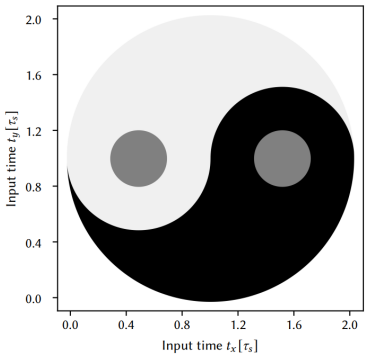


Figure 3.2.: Visualization of the 2D Yin-Yang dataset. [5, fig. 2]

Due to the modularity of the whole system, to add delays, only few changes had to be made. The dataset was trained and set up on the same scheme as described in 2.1. The main change was: when setting up the topology all `lif` layers were replaced by `dlif` layers. In addition, we have to implement the co training of delays and weights. This is done by `optax.multi_transform()`.

3.4. Training Delays in the Minimal Example

The results of traces we obtained for our Input data (fig. 3.3) show delayed spiking behavior in the forward pass and in addition that it is possible to train delays the same way it is done with weights. Since in this example weights were initialized with the property to spike if they receive a input spike but otherwise were left untouched, and only delays were trained, the resulting drop of the loss to 0 (fig. 3.4a) as well as the spike times converging to their target values [fig. 3.4b) is a clear outcome of delay adaptation.

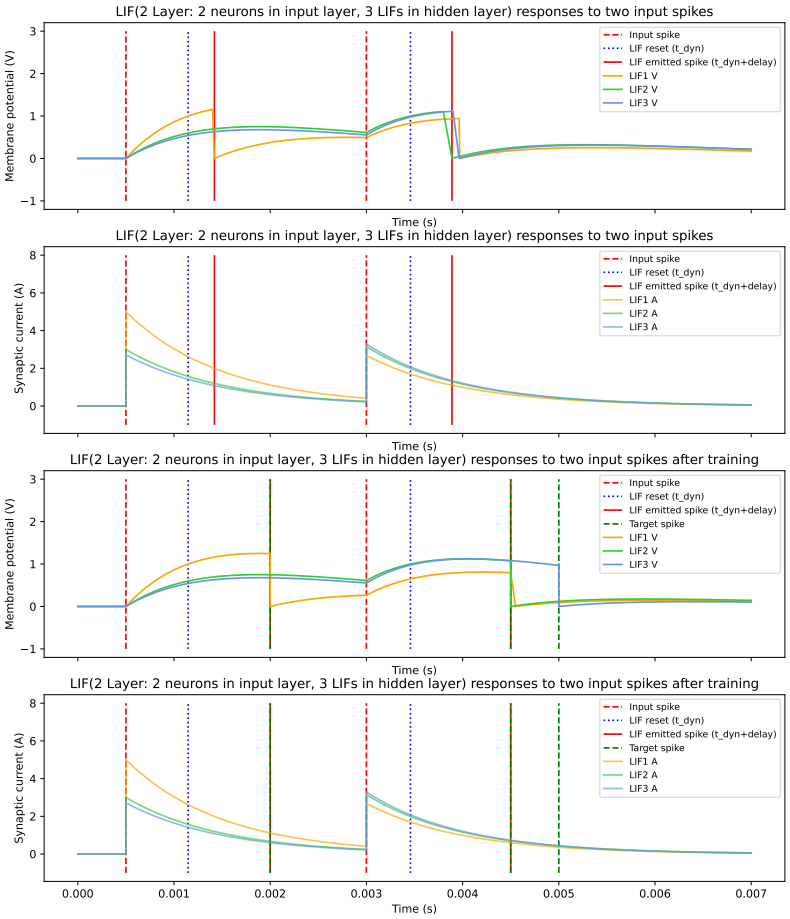
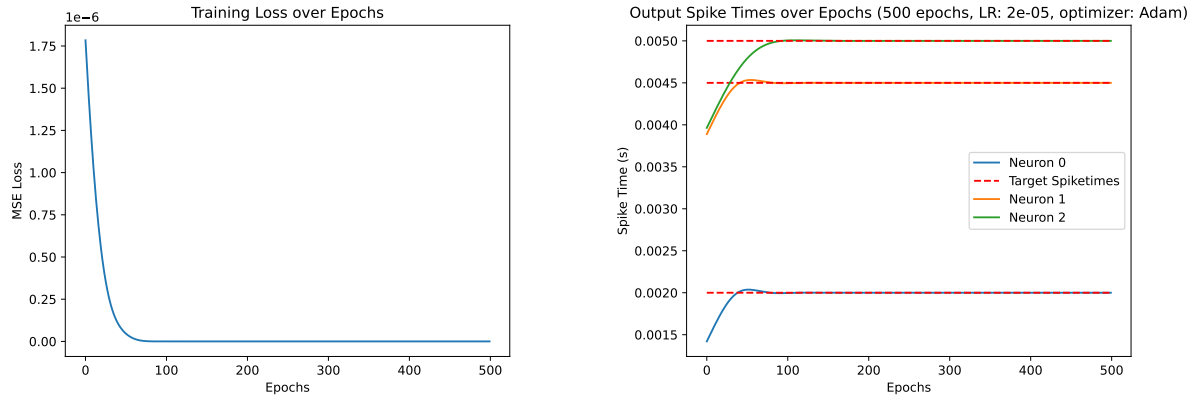


Figure 3.3.: Traces of each neurons synaptic current and Membrane potential. Upper 2 graphs: before training; Lower 2 graphs: after training. Input spikes, the reset of the potential and resulting spikes are represented as horizontal lines, where training happens, this is also the case for the target time we train the net to spike at.

It is visible that the reset of the membrane potential in all cases does not happen at the blue dotted line, which indicates the reset point of the membrane. Instead the membrane potential rises over the threshold. This is not because of false assumptions, but due to visualization. The inbuilt `record` function records traces in the position of a external viewer or in other words the next neuron. The next neuron after this will only see the resulting input spike it gets from the neuron and assume that the membrane reset happens at that exact point in time. All neuron dynamics are assumed to behave accordingly, when in reality in the neuron the potential was reset at the blue dotted line. With the given time limitations and

this fact in mind, this recording function will be used never the less. Potentially it will get fixed in the future.



(a) Loss **(b)** Spike times

Figure 3.4.: Training results: (a) loss over epochs, converging to zero; and (b) output spike times over epochs with dotted lines indicating the target value.

3.5. Applying Delays to the yinyang_analytical algorithm

The SNN with delays (yinyang_analytical_delays) was ran for different sizes of the hidden layer. For 100 neurons, which is the value used in earlier experiments of yinyang_analytical, the accuracy achieved was around 95% percent and the loss converged to a value quite close to zero. In comparison the results obtained without delays (Fig. A.2) yielded a loss jittering a lot more, but a higher final accuracy of around 97%.
Possibly this is because of the parameters not being tuned for small network sizes yet.

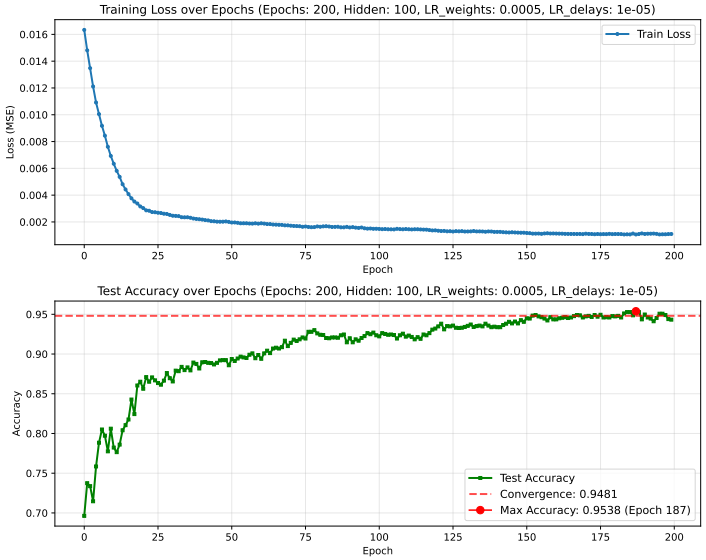


Figure 3.5.: Loss (top) and accuracy (bottom) over epochs for Yin-Yang with delays (100 hidden neurons)

When looking at the evolution of weights and delays (fig. 3.6), all weights seem to evolve as expected (similar to the ones in Fig. A.1). The delays of layer 1 converge to a non zero value as well, but the delays of the output layer eventually all converge to zero. When looking at a sample of 10 of the delay traces in the first neuron layer, it is visible that some delays drop to zero or a small value below. This is most likely due to not yet optimized handling of negative delays. Currently a delay that reaches zero

stays zero forever. This might also be a reason for a less good performance than for training without delays.

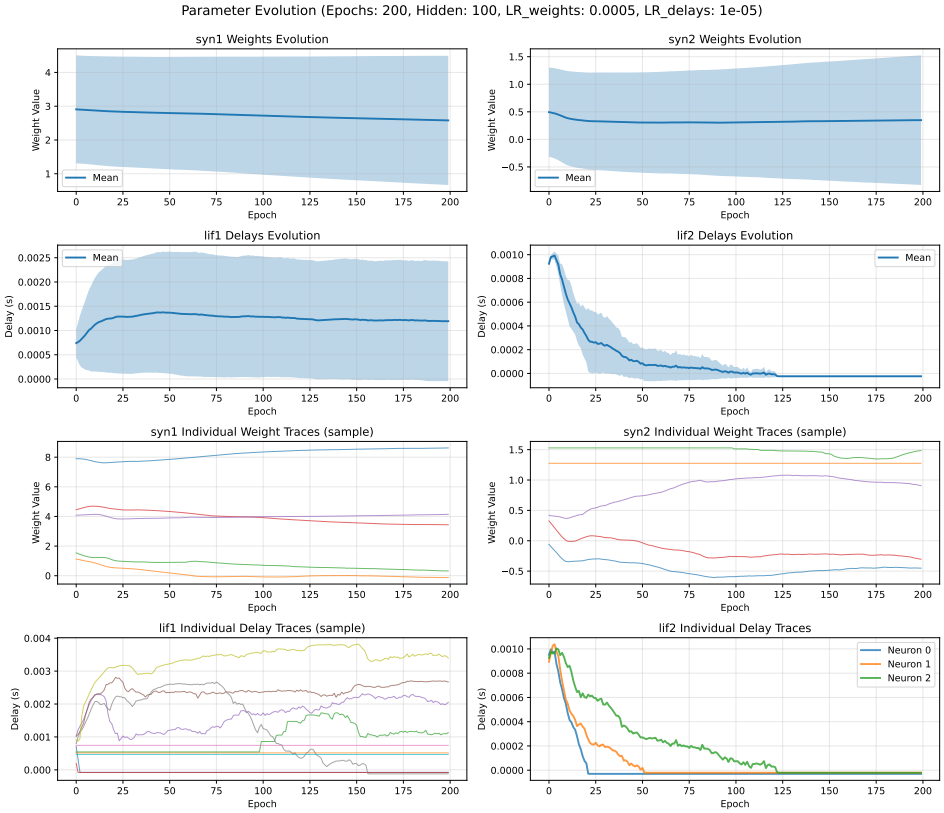


Figure 3.6.: Yin-Yang (100 hidden neurons) evolution of the weights and delays (standard deviation and mean per epoch) and evolution of a sample of 10, or if less present, this amount of weights or delays in each layer of the SNN

As it is proposed in DelGrad [3] that delays especially improve the performance of small networks, networks with smaller amounts of hidden neurons were also observed (e.g. for 50 hidden neurons: fig. A.3 and fig. A.4). They show similar behavior in weights and delays as described before, but the difference in achieved accuracy becomes even larger and is in favor of the implementation without delays. These results are not expected and also do not agree with those of Göltz et al. [3]. In their paper they present results of a SNN so small (less than 5 hidden neurons), that if in our example we only copy the size and leave the parameter the same, the performance collapses entirely (e.g. for 7 hidden neurons fig. A.5, fig. A.6).
Essentially what was done different in this internship was that the parameters were not yet optimized. Next steps would be to find out the optimized parameters by varying them, for instance for smaller network sizes less activity is expected, which requires to tune the weight initialization.
Or as a first step use theirs, and compare the performance with and without delays again.

4. Conclusion and Outlook

In this internship delays were started to get introduced in the jaxsnn API. The trainability of those introduced delays using the `JAX` auto grad engine, corresponding to implicit DelGrad [3] was verified on a minimal example and on the Yin-Yang Dataset.

While the accuracies of 96% obtained for the Yin-Yang example show that the addition of trainable delays to SNNs is possible in general, which was the main goal of this report, a significant increase in accuracy for a version with delays is yet to be seen.
The assumption is that this is not due to the delays themselves, but due to the parameters not yet being initialized correctly. The strength of delays especially shows for smaller networks (see [3]), but the network used in the Yin-Yang example initialized parameters for network sizes of 100 neurons in the hidden layer.

The next steps would be to optimize the parameters of the Yin-Yang example to be more fit for smaller network sizes and compare the results.
In terms of computational efficiency, it might be sensible to introduce a sorted data structure such as a heap to store the spikes ordered in time, instead of sorting after every layer.
Another aspect not yet examined in detail is the clipping of delays that become negative in the training process. Introducing an approach where delays are clipped to a small constant might cause less delays to drop to 0 and stay there. In general a more thorough examination of the gradient behavior would be sensible, but was not possible due to the time scope.

What can be concluded is that the fascinating topic of delays has not yet been exhausted and holds a wealth of potential! The inclusion of delays is only the first step in a path that will require a lot of optimization, observation, and research . Thus, it is important to create an infrastructure within which delays can be included in training. Consequently, further steps will be the inclusion of delays in the Application Programming Interface (API) of jaxsnn, as well as an examination of the inclusion of delays into other algorithms such as event prop [3] in jaxsnn.

Bibliography

[1] K. Yamazaki, V.-K. Vo-Ho, D. Bulsara, and N. Le. "Spiking Neural Networks and Their Applications: A Review". In: *Brain Sciences* 12.7 (2022), p. 863. DOI: 10.3390/brainsci12070863. URL: https://www.mdpi.com/2076-3425/12/7/863.

[2] 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". In: *Nature Machine Intelligence* 3.9 (2021), pp. 823–835. DOI: 10.1038/s42256-021-00388-x.

[3] J. Göltz, J. Weber, L. Kriener, S. Billaudelle, P. Lake, J. Schemmel, M. Payvand, and M. A. Petrovici. "DelGrad: Exact Event-Based Gradients for Training Delays and Weights on Spiking Neuromorphic Hardware". In: *Nature Communications* 16.1 (2025), p. 8245. DOI: 10.1038/s41467-025-63120-y. URL: https://doi.org/10.1038/s41467-025-63120-y.

[4] J. Bradbury, R. Frostig, P. Hawkins, M. J. Johnson, C. Leary, D. Maclaurin, G. Necula, A. Paszke, J. VanderPlas, S. Wanderman-Milne, and Q. Zhang. *JAX: composable transformations of Python+NumPy programs*. Version 0.3.13. 2018. URL: http://github.com/jax-ml/jax.

[5] E. Müller, M. Althaus, E. Arnold, P. Spilger, C. Pehle, and J. Schemmel. *jaxsnn: Event-Driven Gradient Estimation for Analog Neuromorphic Hardware*. 2024. arXiv: 2401.16841 [cs.NE]. URL: https://arxiv.org/abs/2401.16841.

[6] M. Althaus. "Efficient Software for Event-based Optimization on Neuromorphic Hardware". Master's thesis. Heidelberg, Germany: Kirchhoff-Institute for Physics, Heidelberg University, 2023.

[7] L. Kriener, J. Göltz, and M. A. Petrovici. *The Yin-Yang dataset*. 2022. arXiv: 2102.08211 [cs.AI]. URL: https://arxiv.org/abs/2102.08211.

A. Additional material

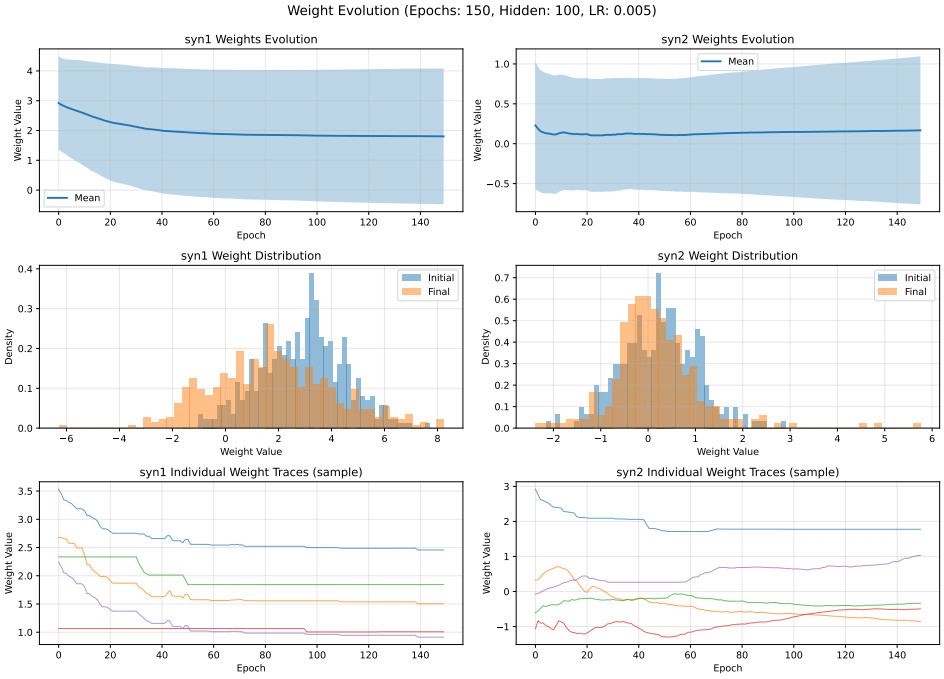


Figure A.1.: Parameter evolution of Yin-Yang without delays (100 hidden neurons).

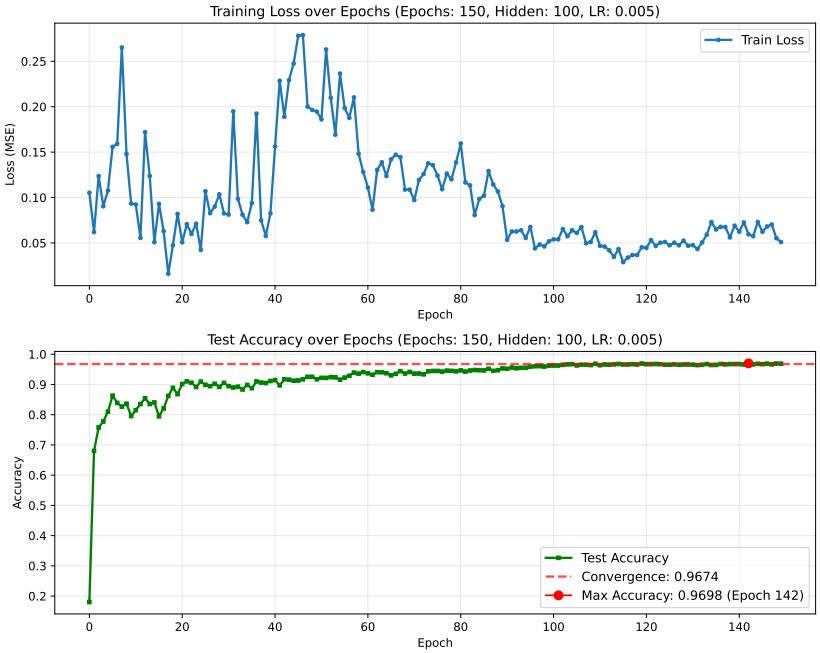


Figure A.2.: Training metrics of Yin-Yang without delays (100 hidden neurons).

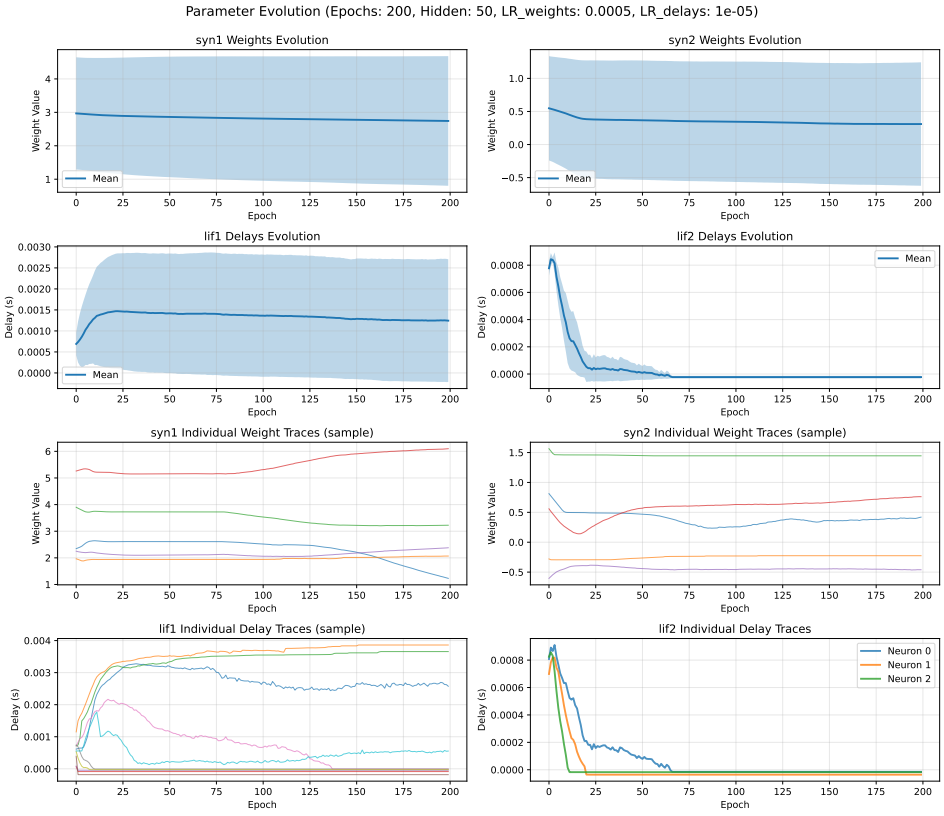


Figure A.3.: Yin-Yang parameter evolution (50 hidden neurons).

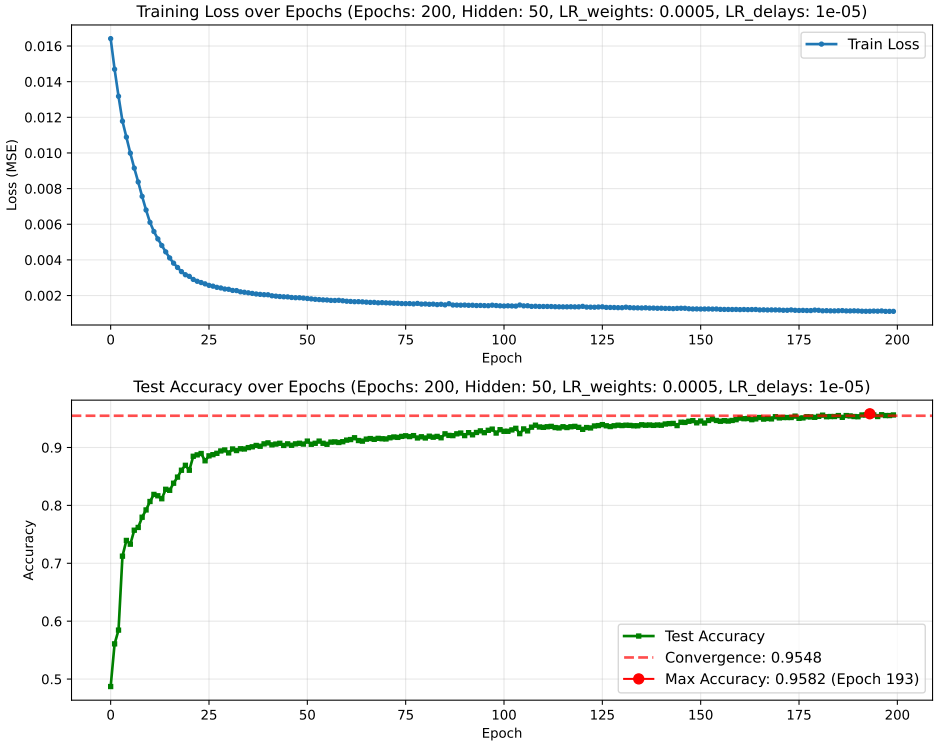


Figure A.4.: Yin-Yang training metrics (50 hidden neurons).

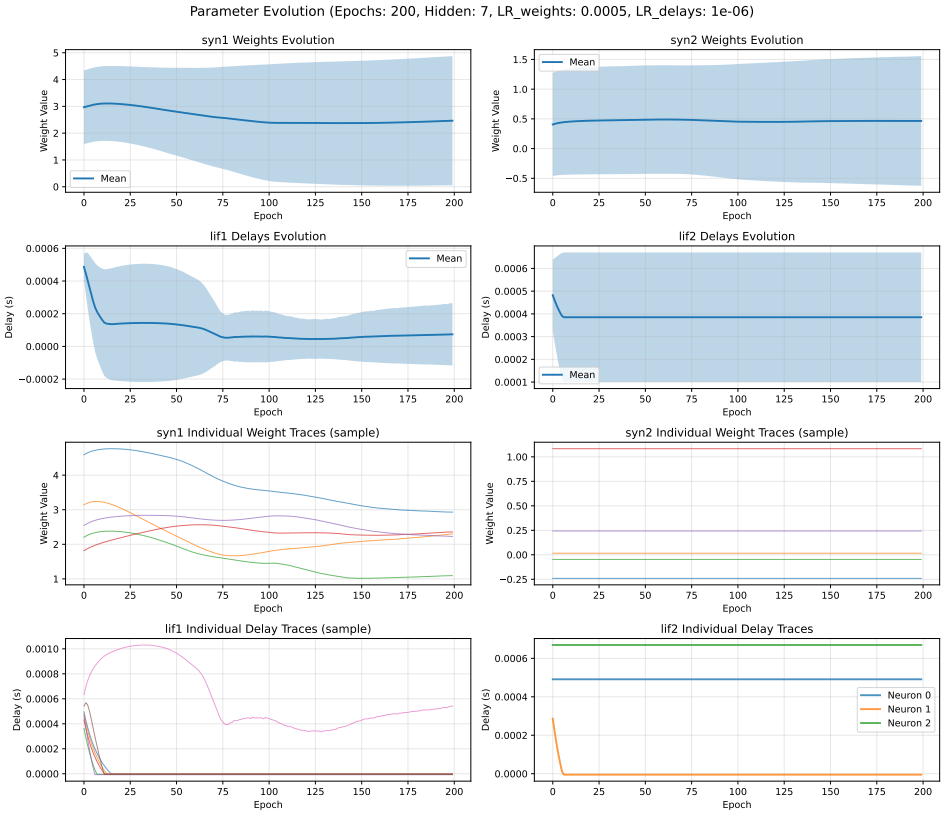


Figure A.5.: Parameter evolution of Yin-Yang with delays (7 hidden neurons).

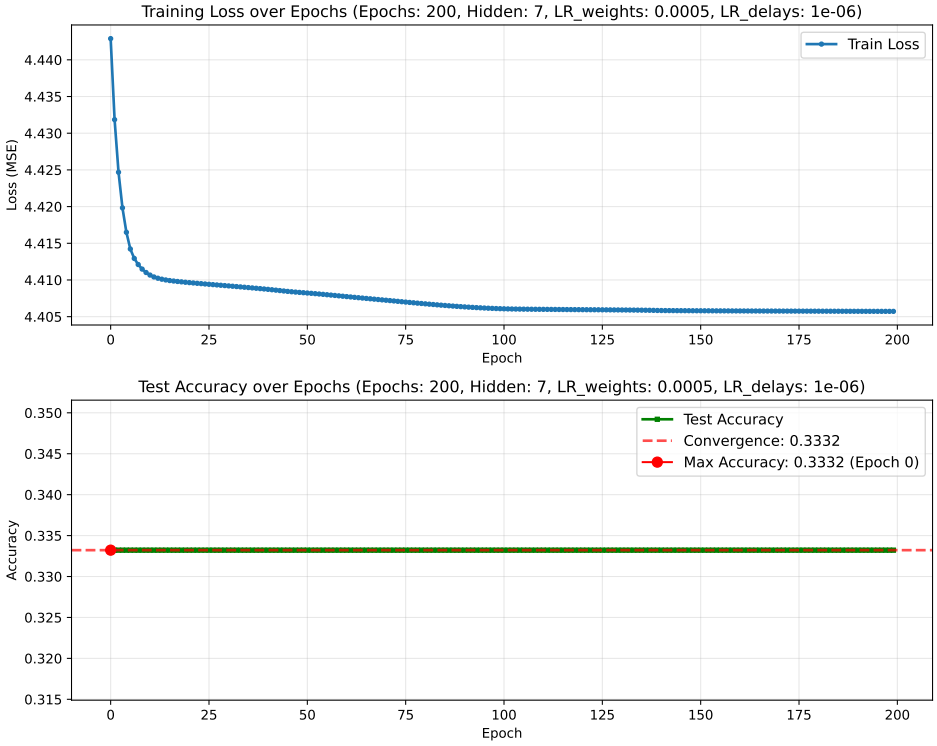


Figure A.6.: Training metrics of Yin-Yang with delays (7 hidden neurons). The accuracy persistently stays on 0.333 meaning that the network is not better than just randomly picking the label of a point

A.0.1. Parameters used

Parameter	**Fig.3.6; Fig.3.5** 20260127132654	**A.1; A.2** 20260127141017	**A.3; A.4** 20260127131249	**A.5; A.6** 20260114154925
Seed	42	0	42	42
Hidden size	100	100	50	7
Hidden spikes	50	50	50	50
Output spikes	53	53	53	53
Epochs	200	150	200	200
Batch size	64	64	64	64
Learning rate (weights)	5×10^{-4}	5×10^{-3}	5×10^{-4}	5×10^{-4}
Learning rate (delays)	1×10^{-5}	–	1×10^{-5}	1×10^{-6}
Learning rate decay	0.99	0.99	0.99	0.99
Membr. time const. τ_{mem}	0.01	0.01	0.01	0.01
Syn.c time const. τ_{syn}	0.005	0.005	0.005	0.005
Threshold	1.0	1.0	1.0	1.0
T_{late}	0.01	0.01	0.01	0.01
Correct target time	0.0045	0.0045	0.0045	0.0045
Wrong target time	0.0055	0.0055	0.0055	0.0055
Training set size	4992	4992	4992	4992
Test set size	2944	2944	2944	2944

Table A.1.: Training parameters of experiments shown in this report. Highlighted rows indicate parameters that differ between experiments.