

LUDWIG-MAXIMILIANS-UNIVERSITÄT AT MÜNCHEN
Department “Institut für Mathematik”
Lehr- und Forschungseinheit Mathematische Grundlagen der künstlichen Intelligenz
Prof. Dr. Gitta Kutyniok



Bachelor's Thesis

Understanding the Representational Power of Recurrent Spiking Neural Networks in Discrete Time

Valentin Herrmann
plus+official@valentin-herrmann.de

Bearbeitungszeitraum: 15.8.2025 bis 31.10.2025
Betreuer: Adalbert Fono
Verantw. Hochschullehrer: Prof. Dr. Gitta Kutyniok

Contents

1	Introduction	1
2	Definition of r. LIF-SNN	2
2.1	Notations and Conventions	2
2.2	Motivation	2
2.3	Definitions	5
2.4	Basic properties	7
3	Structure of computations in r. LIF-SNNs	12
4	Complexity of input partitions	30
5	Experimental results	40
5.1	Computing the number of regions	40
5.2	Visualization of landscape	42
6	Conclusion	45
	Bibliography	46
7	Appendix	47
7.1	dt-lif-snn-compute-regions-depthsearch	47
7.2	dt-lif-snn-compute-regions	57
7.3	dt-lif-snn-visualizer	85

1 Introduction

The capabilities of AI have improved significantly in recent years. Especially large language models have found their ways into most people’s lives. But even though LLMs (see [Jones and Bergen, 2025]) already pass a simple Turing Test, it is still unclear whether they are fundamentally able of reasoning. While they are able to solve many problems with lots of compute, the process of finding a solution quite often involves far more try and error than it does for a human (see [Collins et al., 2023]) and obtains the correct result for the wrong reasons (see [Mondorf and Plank, 2024]). Human intuition on the other hand does not seem to require as many attempts and quite often leads to a better solution (see [Collins et al., 2023]). Another limitation are hallucinations that appear to be an intrinsic property of LLMs (see [Huang et al., 2025]).

Another problem with current models is their incredible energy inefficiency compared to a human brain. While the human brain operates only on 20W (see [Kováč, 2009]), the training by itself of LLM consumes huge amounts of energy: The training of GPT3 consumed 1GWh and inference of a short query using e.g. GPT-4o uses 0.4Wh (see [Jegham et al., 2025]).

While LLMs might eventually overcome some of those problems, especially the extreme energy-inefficiency seems to be an inherent property of the current training process. We conclude therefore that other models might be better fitted to the task of reasoning. Since many human technological advances have been inspired by the longest ongoing optimization progress in history — *the evolution of life* — like airplanes or sonars, we propose that neural networks with more similar inner workings to human brains might be smarter and more efficient. In particular we will investigate a spiking neural network with recursive connections.

While the idea behind spiking neural networks is quite old, they have not been researched as much, since finding a good training algorithm seems to be harder in comparison. Therefore there still remain a lot of open questions about these networks. In this paper we shall extend on the work done in [Nguyen et al., 2025]. We will extend the model of the networks by adding a decaying factor to the input of the neurons and allowing recursive connections between neurons in a layer.

We will roughly follow the structure of [Nguyen et al., 2025]: In *Section 2*, we will motivate and formally introduce recurrent discrete time leaky-integrate-and-fire SNN, in short r. LIF-SNNs. In *Section 3* we will show that some continuously differentiable functions can be approximated arbitrarily well by r. LIF-SNNs. While this has already been show in [Nguyen et al., 2025] more generally for continuous functions, we present a construction using far less neurons, using the internal linear structure of r. LIF-SNNs. In the following, in *Section 4*, we analyze the landscape of a r. LIF-SNN regarding the shape of constant output regions. While we were not able to generalize the upper bound on the number of constant regions of d.t. LIF-SNN that [Nguyen et al., 2025] established to r. LIF-SNN, we show some promising experimental results in *Section 5*. We additionally explain how we implemented the algorithms

2 Definition of r. LIF-SNN

2.1 Notations and Conventions

First a few words about the notation we will use in the paper:

We write $\{a, \dots, b\} := \{a, a+1, \dots, b\}$ for the range of integers from a to b and in particular $[n] := \{1, \dots, n\}$ for the integers from 1 to n and $[n]_0 := \{0, \dots, n\}$ for the integers from 0 to n . Moreover \vee is used to mean the logical “or”, \wedge to mean the logical “and”.

We write $[x, y) := \{z \in \mathbb{R} \mid x \leq z < y\}$ for the half-open interval between $x \in \mathbb{R}$ and $y \in \mathbb{R}$ and further use $\llbracket x, y \rrbracket$ with $x, y \in \mathbb{R}^n$ to write half-open cuboids $\prod_{i=1}^n [x_i, y_i)$. Similarly $\llbracket x, y \rrbracket$ is the closed cuboid $\prod_{i=1}^n [x_i, y_i]$. A cube is a cuboid such that all sides have the same length. We further take $[x, y)$ to be empty for $x, y \in \mathbb{R}$ with $x \geq y$, and $[-\infty, x]$ to mean $(-\infty, x)$. This means in particular, that $\llbracket x, y \rrbracket = \emptyset$ holds exactly when $y_i \leq x_i$ for a $i \in [n]$.

We further use $\text{diam}_p(U) := \sup_{x, y \in U} \|x - y\|_p$ to mean the diameter of $U \subset \mathbb{R}^m$ regarding the p -norm $\|\cdot\|_p$.

Continuing, $e_i := (0, \dots, 1, \dots, 0) \in \mathbb{R}^n$ is the i -th standard basis vector, $\mathbf{0}_n := (0, \dots, 0) \in \mathbb{R}^n$, $\mathbf{1}_n := (1, \dots, 1) \in \mathbb{R}^n$ the vectors containing 0 and 1s in every component, respectively and $I_n \in \mathbb{R}^{n \times n}$ the identity matrix of size $n \times n$. Is moreover $W \in \mathbb{R}^{n \times m}$ a matrix, then we will write $w_i \in \mathbb{R}^{1 \times n}$ for the i -th row vector and $W_{i,j} \in \mathbb{R}$ for the value of the cell (i, j) . We add an ordering \leq to vectors: $x \leq y$ holds exactly for $x, y \in \mathbb{R}^n$ if $\forall_{i \in [n]} x_i \leq y_i$. Further $x < y$ is canonically defined, so $x < y$ holds exactly if $x \leq y$ but not $x = y$.

Is further $f: U \rightarrow \mathbb{R}^n$ a function with arbitrary domain U , then $f_i := \pi_i \circ f$ is the i -th component function. $\pi_i: \mathbb{R}^n \rightarrow \mathbb{R}$ is the projection to the i -th component. We also write f^{-1} for the inverse of f and $f^{-1}(W) := \{x \in U \mid f(x) \in W\}$ for the preimage of W under f . The set of continuous functions of type $U \rightarrow W$ will be written by $\mathcal{C}^0(U, W)$. Correspondingly, the set of k -times continuously differentiable functions will be written $\mathcal{C}^k(U, W)$. The total derivative of a differentiable function $f: U \rightarrow W$ is written as $df: U \rightarrow \text{Hom}_{\mathbb{R}}(W, W)$.

The norms $\|\cdot\|_p$ either are the p -norm on \mathbb{R}^n or the operator norm with regard to the p -norm $\|\cdot\|_p$ in input and output space. Further, $\|f\|_{\infty, p} := \sup_{x \in U} \|f(x)\|_p$ for a function $f: U \rightarrow \mathbb{R}^n$.

Some further notation regarding geometry: \bar{A} is the closure of $A \subset \mathbb{R}^n$, A° the interior; we further write $B_{\varepsilon, p}(x) := \{y \mid \|y - x\|_p < \varepsilon\}$ for $\varepsilon > 0$, $p \geq 1$ and $x \in \mathbb{R}^n$.

We moreover write $\max U := (\max_{u \in U} u_i)_{i \in [n]}$ with $U \subset (\mathbb{R} \cup \{\pm\infty\})^n$ for the maximum by component of U , we similarly define $\min U$, $\inf U$ and $\sup U$. In addition we use the conventions $\inf(\emptyset) = \infty$ and $\sup(\emptyset) = -\infty$. We will also use \wedge and \vee to write the supremum and infimum respectively when convenient.

We finally write

$$\chi_M(x) := \begin{cases} 1 & x \in M \\ 0 & x \notin M \end{cases}$$

for the characteristic function of a set M and

$$1_A := \begin{cases} 1 & A \\ 0 & \neg A \end{cases}$$

for a formula A , such that in particular $\chi_M(x) = 1_{(x \in M)}$.

2.2 Motivation

A neuron as in Fig. 2.1 has been described in [Gerstner et al., 2014] by the following:

¹Source: By BruceBlaus - Own work, CC BY 3.0, <https://commons.wikimedia.org/w/index.php?curid=28761830>

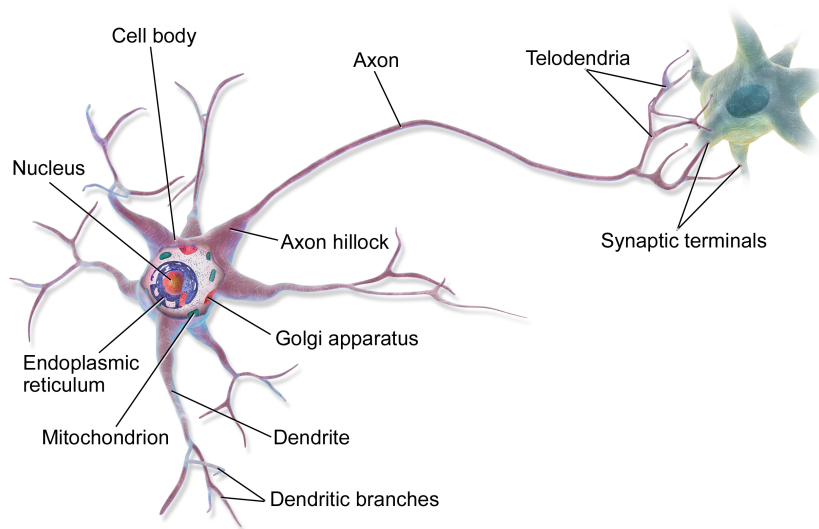


Figure 2.1: Structure of a neuron.¹

A typical neuron can be divided into three functionally distinct parts, called dendrites, soma [also called cell body], and axon; [...]. Roughly speaking, the dendrites play the role of the ‘input device’ that collects signals from other neurons and transmits them to the soma. The soma is the ‘central processing unit’ that performs an important non-linear processing step: If the total input arriving at the soma exceeds a certain threshold, then an output signal is generated. The output signal is taken over by the ‘output device’, the axon, which delivers the signal to other neurons.

The nature of the signals traveling between neurons is described in a following paragraph.

The neuronal signals consist of short electrical pulses [...]. The form of the pulse does not change as the action potential propagates along the axon. A chain of action potentials emitted by a single neuron is called a spike train – a sequence of stereotyped events which occur at regular or irregular intervals; [...]. Since isolated spikes of a given neuron look alike, the form of the action potential does not carry any information. Rather, it is the number and the timing of spikes which matter.

Finally, the effect of signals on the state of the cell body is outlined in the last quote.

The effect of a spike on the postsynaptic neuron can be recorded with an intracellular electrode which measures the potential difference $u(t)$ between the interior of the cell and its surroundings. This potential difference is called the membrane potential. Without any input, the neuron is at rest corresponding to a constant membrane potential u_{rest} . After the arrival of a spike, the potential changes and finally decays back to the resting potential, [...].

To summarize: A neuron in the human brain has connections to other neurons, which may be weaker or stronger, as well as an internal membrane potential that is increased or decreased by the input from other neurons. If that membrane potential reaches a threshold, the neuron will spike and the subscribed neurons will receive an update. If it does not reach the threshold, the potential just decays to the rest potential. Updates from other neurons are delivered by chemical transmitters or electric signals. We will not model these signals as a single binary spike, such that in particular the input immediately drops down to the rest state, but will instead let it decay over time after the spike, e.g. because the chemical transmitters have some chance of taking a longer path to the next neuron.

From these biologic observations we can derive the following differential equations for a single neuron input $I : \mathbb{R} \rightarrow \mathbb{R}$ and membrane potential $U : \mathbb{R} \rightarrow \mathbb{R}$:

$$\begin{aligned} I'(t) &:= -\tau_\alpha I(t) + \sum_{j=1}^n w_j s_j(t - \Delta t) \\ U'(t) &:= -\tau_\beta U(t) + I(t) + b - \vartheta s(t) \end{aligned}$$

Here $s_j : \mathbb{R} \rightarrow \{0, 1\}$ represents the j -th connection of the given neuron spiking at time t ; the variables $w_j \in \mathbb{R}$ represent the weights of the connections. The $\vartheta s(t)$ resets the potential $U(t)$ after a spike. Since we might get unresolvable dependencies between neurons otherwise, the connections need to have some latency $\Delta t \in \mathbb{R}$. The variables $\tau_\alpha, \tau_\beta > 0$ specify the decay rate with which I and U respectively decay. We also include the bias $b \in \mathbb{R}$ in the differential equation of U to simplify constructions of r. LIF-SNN later.

Since we are in an analog scenario, it is reasonable to assume that I and U describe smoothly differentiable functions. We can therefore use the first-order exponential integrator method to obtain a discretization of I and U from the differential equations. Let $t_0, h \in \mathbb{R}$ be arbitrary and $t_{n+1} := t_n + h$. By using the fundamental theorem of calculus we get

$$\begin{aligned} &e^{\tau_\alpha t_{n+1}} I(t_{n+1}) - e^{\tau_\alpha t_n} I(t_n) \\ &= \int_{t_n}^{t_{n+1}} \frac{d}{dt} (e^{\tau_\alpha t} I(t)) dt \\ &= \int_{t_n}^{t_{n+1}} e^{\tau_\alpha t} (I'(t) + \tau_\alpha I(t)) dt \\ &= \int_{t_n}^{t_{n+1}} e^{\tau_\alpha t} \left(\sum_{j=1}^n w_j s_j(t - \Delta t) \right) dt \end{aligned}$$

If we assume the input from the spikes of other neurons to be constant during $[t_n, t_{n+1}]$, we get

$$\begin{aligned} &= \frac{1}{\tau_\alpha} (e^{\tau_\alpha t_{n+1}} - e^{\tau_\alpha t_n}) \sum_{j=1}^n w_j s_j(t - \Delta t) \\ &= \frac{1}{\tau_\alpha} e^{\tau_\alpha t_{n+1}} (1 - e^{-\tau_\alpha h}) \sum_{j=1}^n w_j s_j(t - \Delta t) \end{aligned}$$

So we get

$$I(t_{n+1}) = e^{-\tau_\alpha h} I(t_n) + \frac{1}{\tau_\alpha} (1 - e^{-\tau_\alpha h}) \sum_{j=1}^n w_j s_j(t - \Delta t)$$

Let us now use $h = 1$ and absorb $\frac{1 - e^{-\tau_\alpha}}{\tau_\alpha}$ into the weights $(w_i)_{i \in [n]}$, such that we have

$$I(t_{n+1}) = \alpha I(t_n) + \sum_{j=1}^n w_j s_j(t - \Delta t) \quad (1)$$

by writing $\alpha := e^{-\tau_\alpha}$. We similarly obtain

$$U(t_{n+1}) = \beta U(t_n) + I(t) + b - \vartheta s(t) \quad (2)$$

by using the first-order exponential integrator method, defining $\beta := e^{-\tau_\beta}$ and absorbing $\frac{1 - \beta}{\tau_\beta}$ into $I(0)$, $(w_i)_{i \in [n]}$, b and ϑ . Note that we have $\alpha, \beta \in [0, 1]$ by construction.

We will now arrange those neurons into layers like shown in Fig. 2.2, such that neurons are only connected to neurons from the previous layer or their own layer. Of course, since we allow recurrent connections we can always just merge all layers into the first one, but the layers do give

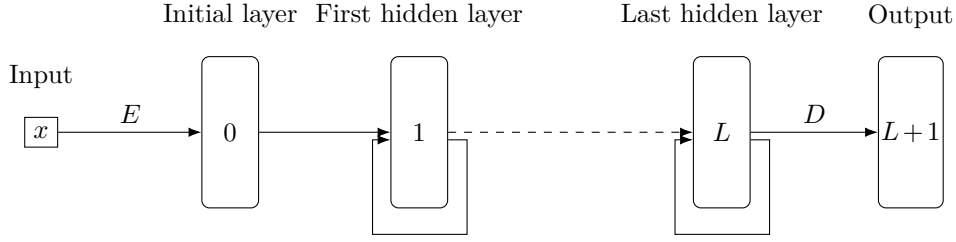


Figure 2.2: High-level network layout

us more control over the network. For connections between different layers, we can use $\Delta t = 0$ in (1) since there can be no interdependent connections between neurons in different layers. For connections between neurons in the same layer we use $\Delta t > 0$ to prevent those interdependencies. Since we have previously chosen $h = 1$, it is natural to further choose $\Delta t = 1$.

Since we want to work with arbitrary data and not just spike trains, we further need to encode our data to spikes trains and decode it from spike trains. Like [Nguyen et al., 2025] we choose a simple direct encoding and membrane potential outputs.

2.3 Definitions

Our type of SNN should be thought of as a composition of an initial input layer, a number of hidden spiking layers with internal state and an affine-linear layer mapping spikes activations over time, so called spike trains, to the value of the output layer.

We first define the structure of the hidden layers:

Definition 2.1. The **input vector** $i^{[l]}(t) \in \{0, 1\}^{n_l}$, the **spike vector** $s^{[l]}(t) \in \{0, 1\}^{n_l}$, the **pre-spike membrane potential vector** $p^{[l]}(t)$ and the **post-spike membrane potential vector** $u^{[l]}(t)$ of a hidden layer $\lambda = (W^{[l]}, b^{[l]}, u^{[l]}(0), i^{[l]}(0), \alpha^{[l]}, \beta^{[l]}, \vartheta^{[l]})$ with index $l \in [L]$, are recursively defined as

$$i^{[l]}(t) := \alpha^{[l]} i^{[l]}(t-1) + W^{[l]} s^{[l-1]}(t) + V^{[l]} s^{[l]}(t-1), \quad (3)$$

$$p^{[l]}(t) := \beta^{[l]} u^{[l]}(t-1) + i^{[l]}(t) + b^{[l]}, \quad (4)$$

$$s^{[l]}(t) := H(p^{[l]}(t) - \vartheta \mathbf{1}_{n_l}), \quad (5)$$

$$u^{[l]}(t) := p^{[l]}(t) - \vartheta s^{[l]}(t), \quad (6)$$

with $\forall_{l \in [L]} s^{[l]}(0) = 0$ and given

- **initial membrane potential:** $u^{[l]}(0) \in \mathbb{R}^{n_l}$,
- **initial input:** $i^{[l]}(0) \in \mathbb{R}^{n_l}$,
- **weight matrices:** $W^{[l]} \in \mathbb{R}^{n_l \times n_{l-1}}$, $V^{[l]} \in \mathbb{R}^{n_l \times n_l}$,
- **bias vectors:** $b^{[l]} \in \mathbb{R}^{n_l}$,
- **leaky terms:** $\alpha^{[l]}, \beta^{[l]} \in [0, 1]$,
- **threshold:** $\vartheta^{[l]} \in (0, \infty)$,

where $H := \mathbb{1}_{[0, \infty)}$ is a step function, $T \in \mathbb{N}$ is the number of **simulated time steps** and $L \in \mathbb{N}$ the total **number of hidden layers**.

Remark 2.1. While it is suppressed in the notation, $i^{[l]}$, $p^{[l]}$, $s^{[l]}$ and $u^{[l]}$ clearly not only depend on t , but by recursion also on $s^{[0]}$. Further, $s^{[l]}$ can be represented as an element of $\{0, 1\}^{n_l \times T}$, which will become useful later to quantify over spike trains. In particular, we will write $\sigma(t)$ for $\sigma \in \{0, 1\}^{n_l \times T}$ and $1 \in [T]$ to mean the t -th column vector of σ .

We further define recurrent d.t. LIF-SNN and the function the network realizes:

Definition 2.2. A **recurrent discrete-time LIF-SNN**, also called r. LIF-SNN, of **depth** L with **layer-widths** (n_0, \dots, n_{L+1}) and $T \in \mathbb{N}$ time-steps is given by

$$\Phi := ((W^{[l]}, b^{[l]}, V^{[l]}, u^{[l]}(0), i^{[l]}(0), \alpha^{[l]}, \beta^{[l]}, \vartheta^{[l]})_{l \in [L]}, T, (E, D))$$

where the **input encoder** $E: \mathbb{R}^{n_0} \rightarrow \mathbb{R}^{n_0 \times T}$ maps a vector $x \in \mathbb{R}^{n_0}$ to a corresponding first layer spike activation $\forall_{t \in [T]} s^{[0]}(t) = E(x)(t)$ and the **output decoder** $D: \{0, 1\}^{n_L \times T} \rightarrow \mathbb{R}^{n_{L+1}}$ maps the spike activations of the last hidden layer to real values.

Definition 2.3. A recurrent discrete-time LIF-SNN ϕ **realizes** the function $R(\Phi): \mathbb{R}^{n_0} \rightarrow \mathbb{R}^{n_{L+1}}$:

$$R(\Phi)(x) = D((s^{[L]}(t))_{t \in [T]}) \quad \text{with } s^{[0]} := E(x)$$

Definition 2.4. A recurrent discrete-time LIF-SNN employs **direct encoding** if we have

$$\forall_{t \in [T]} E(x)(t) = x$$

for the input encoder and has **membrane potential outputs** if the output decoder can be written as

$$D((s(t))_{t \in [T]}) = \sum_{t=1}^T a_t (W^{[L+1]} s^{[L]}(t) + b^{[L+1]})$$

for some $(a_t)_{t \in [T]} \in \mathbb{R}^T$, $b^{[L+1]} \in \mathbb{R}^{n_{L+1}}$ and $W^{[L+1]} \in \mathbb{R}^{n_{L+1} \times n_L}$.

Remark 2.2. We will only consider recurrent discrete-time LIF-SNN with direct encoding and membrane potential outputs. In fact, we will use “recurrent discrete-time LIF-SNN” to mean “r. LIF-SNN with direct encoding and membrane potential”.

Remark 2.3. Our definition of r. LIF-SNN breaks now down to the definition d.t. LIF-SNN of [Nguyen et al., 2025] if we require $\alpha^{[l]} := 0$ and $V^{[l]}$ for all layers $l \in [L]$. So r. LIF-SNNs can have recursive dependencies inside of the layers and decaying input vectors in contrast to the more simpler d.t. LIF-SNNs.

Let us now take a look at some simple examples of r. LIF-SNN:

Example 2.1. Let $T, L \in \mathbb{N}$. Then there exists a r. LIF-SNN with $\forall_{t \in [T]} s^{[L]}(t) = s^{[0]}(t)$ for any $s^{[0]} \in \{0, 1\}^{n_0 \times T}$.

Let us use constant width $n_l = n$, weights $W^{[l]} = I_n$, $V^{[l]} = \mathbf{0}_{n \times n}$, biases $b^{[l]} = 0$, initial input $i^{[l]}(0) = 0$, initial membrane potential $u^{[l]}(0) = 0$, leaky terms $\alpha^{[l]} = \beta^{[l]} = 0$ and threshold $\vartheta^{[l]} = 1$ for all $l \in [L]$.

It follows from the definitions that $i^{[l]}(t) = s^{[l-1]}(t)$ and therefore further

$$s^{[l]}(t) = H(s^{[l-1]}(t) - \mathbf{1}_{n_l}) = s^{[l-1]}(t).$$

So by induction we indeed get $\forall_{t \in [T]} s^{[L]}(t) = s^{[0]}(t)$.

Example 2.2. Let $T, L \in \mathbb{N}$. Then there exists a r. LIF-SNN with $\forall_{t \in T, i \in [n]} s_i^{[L]}(t) = 1 \Leftrightarrow \exists_{t' \in [t-1]} (s_i^{[0]}(t')) = 1$ for any $s^{[0]} \in \{0, 1\}^{n_0 \times T}$, i.e. an output neuron switches on once the corresponding input neurons fires.

Let us use constant width $n_l = n$, weights $W^{[l]} = I_n$, $V^{[l]} = I_n$, biases $b^{[l]} = 0$, initial input $i^{[l]}(0) = 0$, initial membrane potential $u^{[l]}(0) = 0$, leaky terms $\alpha^{[l]} = 0$, $\beta^{[l]} = 0$ and threshold $\vartheta^{[l]} = 1$ for all $l \in [L]$.

We then get by definition

$$i^{[l]}(t) = s^{[l-1]}(t) + s^{[l]}(t-1)$$

and therefore

$$s^{[l]}(t) = H(s^{[l-1]}(t) + s^{[l]}(t-1) - \mathbf{1}_{n_l}).$$

So if and only if $s_i^{[l-1]}(t) = 1$ or $s_i^{[l]}(t-1) = 1$, then $s_i^{[l]}(t) = 1$. We therefore have $\forall_{t \in T} s_i^{[l]}(t) = 1 \Leftrightarrow \exists_{t' \in [t]} (s_i^{[l-1]}(t')) = 1$ for all $i \in [n], l \in [L]$. Thus $\forall_{t \in T, i \in [n]} s_i^{[L]}(t) = 1 \Leftrightarrow \exists_{t' \in [t-1]} (s_i^{[0]}(t')) = 1$ by induction.

There also is a d.t. LIF-SNN construction achieving the same behavior: We use the same parameters as before, but with $V^{[l]} = \mathbf{0}_{n \times n}$, $W^{[l]} = T \cdot I_n$ and $\beta^{[l]} = 1$ for all layers $l \in [L]$.

We then get by definition

$$i^{[l]}(t) = T s^{[l-1]}(t)$$

and therefore

$$\begin{aligned} s^{[l]}(t) &= H(p^{[l]}(t) - \mathbf{1}_{n_l}) \\ p^{[l]}(t) &= p^{[l]}(t-1) + T s^{[l-1]}(t) - s^{[l]}(t-1) \end{aligned}$$

By induction over t (see also [Lemma 2.1](#)) we obtain

$$p^{[l]}(t) = \sum_{k=1}^t (T s^{[l-1]}(k) - s^{[l]}(k-1))$$

Let now $i \in [n]$. Since $s^{[l]}(0) = 0$, we have $\sum_{k=1}^t s_i^{[l]}(k-1) \leq t-1$. Now if and only if there is any $t_0 \in [T]$ with $s_i^{[l-1]}(t_0) = 1$, we get

$$p_i^{[l]}(t') = T \sum_{k=1}^{t'} s_i^{[l-1]}(k) - \sum_{k=1}^{t'} s_i^{[l]}(k-1) \geq 1$$

for $t' \geq t_0$ and hence $s_i^{[l]}(t') = 1$. Just as before we now get the required property for the whole network by induction over the layers.

For a clearer construction of our networks we will additionally define neurons in our network:

Definition 2.5. The i -th neuron of a hidden layer $\lambda = (W^{[l]}, b^{[l]}, V^{[l]}, u^{[l]}(0), i^{[l]}(0), \alpha^{[l]}, \beta^{[l]}, \vartheta^{[l]})$ of a r. LIF-SNN is a tuple (w, b, v, u_0, i_0) with $w \in \mathbb{R}^{n_{l-1}}$, $v \in \mathbb{R}^{n_l}$ and $b, u_0, i_0 \in \mathbb{R}$, such that b, u_0, i_0 are the i -th component of $b^{[l]}, u^{[l]}(0), i^{[l]}(0)$ respectively and w, v are the i -th row vector of $W^{[l]}, V^{[l]}$ respectively.

2.4 Basic properties

In the following we present some technical but helpful notations and lemmas for writing proofs about SNN. Of particular importance are [Definition 2.6](#) and [Lemma 2.1](#), which introduce non-recursive formulas for the defining equations of r. LIF-SNNs.

Definition 2.6. Let $t \in [T]$, $l \in [L]$ and spike train families $\sigma = (\sigma^{[l']})_{l' \in [l]_0}$, $\sigma' = (\sigma'^{[l']})_{l' \in [l]_0}$ be given, such that $\forall_{l \in [l-1]_0} \sigma^{[l']} \in \{0, 1\}^{n_{l'} \times t}$ and $\sigma^{[l]} \in \{0, 1\}^{n_l \times t}$ as well as $\forall_{l' \in [l]_0} \sigma'^{[l']} \in \{0, 1\}^{n_{l'} \times t}$, i.e. σ, σ' have to be chosen such that the terms in the following definitions are well-defined. We define

$$i^{[l]}(t; \sigma) := (\alpha^{[l]})^t i^{[l]}(0) + \sum_{k=1}^t (\alpha^{[l]})^{t-k} (W^{[l]} \sigma^{[l-1]}(k) + V^{[l]} \sigma^{[l]}(k-1)), \quad (7)$$

$$p^{[l]}(t; \sigma) := (\beta^{[l]})^t u^{[l]}(0) + \sum_{k=1}^t (\beta^{[l]})^{t-k} (i^{[l]}(k; \sigma) + b^{[l]}) - \vartheta \sum_{k=1}^{t-1} (\beta^{[l]})^{t-k} \sigma^{[l]}(k), \quad (8)$$

$$s^{[l]}(t; \sigma) := H(p^{[l]}(t; \sigma) - \vartheta \mathbf{1}_{n_l}), \quad (9)$$

$$u^{[l]}(t; \sigma') := (\beta^{[l]})^t u^{[l]}(0) + \sum_{k=1}^t (\beta^{[l]})^{t-k} (i^{[l]}(k; \sigma') + b^{[l]} - \vartheta \sigma'^{[l]}(k)). \quad (10)$$

Remark 2.4. The spike train families σ, σ' in [Definition 2.6](#) are chosen such that they only include the data that is actually needed in the definition of $i^{[l]}, p^{[l]}, s^{[l]}, u^{[l]}$. This is also the reason why we have to use σ' for $u^{[l]}$; its definition requires in contrast to the definitions of $i^{[l]}, p^{[l]}, s^{[l]}$ the newest spikes from the current layer layer.

We will also allow using $i^{[l]}, p^{[l]}, s^{[l]}, u^{[l]}$ with extensions of the by the definition necessary spike train families, in particular we will use the functions with “complete” spike train families $(\sigma^{[l]})_{l \in [L]}$ with $\sigma^{[l]} \in \{0, 1\}^{n_l \times T}$.

The notation for the previous definitions is justified due to

Lemma 2.1. *The non-recursive formulas from [Definition 2.6](#) are equivalent to the recursive definitions for $l \in [L]$, $t \in [T]$ assuming previous spikes are equal: $\forall_{l' \in [l-1]_0} \sigma^{[l']} = s^{[l']}$ and $\forall_{t' \in [t-1]} \sigma^{[l]}(t') = s^{[l]}(t')$ as well as $\forall_{l' \in [l]_0} \sigma'^{[l']} = s^{[l']}$.*

Proof. We first proof $\forall_{t \in [T]} i^{[l]}(t; \sigma) = i^{[l]}(t)$ and $\forall_{t \in [T]} u^{[l]}(t; \sigma) = u^{[l]}(t)$ for $\sigma^{[l]} = s^{[l]}$ by induction: Let $t = 1$. We have

$$\begin{aligned} i^{[l]}(1; \sigma) &= \alpha^{[l]} i^{[l]}(0) + W^{[l]} \sigma^{[l-1]}(1) + V^{[l]} \sigma^{[l]}(0) \\ &= \alpha^{[l]} i^{[l]}(0) + W^{[l]} s^{[l-1]}(1) + V^{[l]} s^{[l]}(0) \\ &= i^{[l]}(1) \end{aligned}$$

and

$$\begin{aligned} u^{[l]}(1; \sigma') &= \beta^{[l]} u^{[l]}(0) + i^{[l]}(1; \sigma') + b^{[l]} - \vartheta \sigma'^{[l]}(1) \\ &= \beta^{[l]} u^{[l]}(0) + i^{[l]}(1) + b^{[l]} - \vartheta s^{[l]}(1) \\ &= p^{[l]}(1) - \vartheta s^{[l]}(1) \\ &= u^{[l]}(1) \end{aligned}$$

by using the definitions and using our assumption $\sigma^{[l]} = s^{[l]}$. We have $i^{[l]}(1; \sigma') = i^{[l]}(1)$, since σ' is an “extension” of σ .

Let further $t > 1$. We may assume $i^{[l]}(t-1; \sigma) = i^{[l]}(t-1)$ and $u^{[l]}(t-1; \sigma') = u^{[l]}(t-1)$ and obtain

$$\begin{aligned} i^{[l]}(t; \sigma) &= (\alpha^{[l]})^t i^{[l]}(0) + \sum_{k=1}^t (\alpha^{[l]})^{t-k} (W^{[l]} \sigma^{[l-1]}(k) + V^{[l]} \sigma^{[l]}(k-1)) \\ &= \alpha^{[l]} i^{[l]}(t-1; \sigma) + (W^{[l]} \sigma^{[l-1]}(t) + V^{[l]} \sigma^{[l]}(t-1)) \\ &= \alpha^{[l]} i^{[l]}(t-1) + W^{[l]} s^{[l-1]}(t) + V^{[l]} s^{[l]}(t-1) \\ &= i^{[l]}(t) \end{aligned}$$

and similarly

$$\begin{aligned} u^{[l]}(t; \sigma') &= (\beta^{[l]})^t u^{[l]}(0) + \sum_{k=1}^t (\beta^{[l]})^{t-k} (i^{[l]}(k; \sigma') + b^{[l]} - \vartheta \sigma'^{[l]}(k)) \\ &= \beta^{[l]} u^{[l]}(t-1; \sigma') + (i^{[l]}(t; \sigma') + b^{[l]} - \vartheta \sigma'^{[l]}(t)) \\ &= \beta^{[l]} u^{[l]}(t-1) + i^{[l]}(t) + b^{[l]} - \vartheta s^{[l]}(t) \\ &= p^{[l]}(t) - \vartheta s^{[l]}(t) \\ &= u^{[l]}(t). \end{aligned}$$

By substituting $u^{[l]}$ in $p^{[l]}$ we further obtain

$$\begin{aligned} p^{[l]}(t; \sigma) &= (\beta^{[l]})^t u^{[l]}(0) + \sum_{k=1}^t (\beta^{[l]})^{t-k} (i^{[l]}(k; \sigma) + b^{[l]}) - \vartheta \sum_{k=1}^{t-1} (\beta^{[l]})^{t-k} \sigma^{[l]}(k), \\ &= \beta^{[l]} u^{[l]}(t-1; \sigma) + (i^{[l]}(t; \sigma) + b^{[l]}) \\ &= \beta^{[l]} u^{[l]}(t-1) + i^{[l]}(t) + b^{[l]} \\ &= p^{[l]}(t). \end{aligned}$$

By using the above we obtain

$$s^{[l]}(t; \sigma) = H(p^{[l]}(t; \sigma) - \vartheta \mathbf{1}_{n_l}) = H(p^{[l]}(t) - \vartheta \mathbf{1}_{n_l}) = s^{[l]}(t).$$

□

Lemma 2.2. Let $a, x \in \mathbb{R}$, $a \neq 0$ and $b \in \mathbb{Z}$. We then have $\lfloor \frac{x}{a} \rfloor = b \Leftrightarrow 0 \leq x - ab < a$.

Proof. $0 \leq x - ab < a$ is equivalent to $b \leq \frac{x}{a} < b+1$, which is yet again equivalent to $\lfloor \frac{x}{a} \rfloor = b$ by definition of $\lfloor \cdot \rfloor$. □

Lemma 2.3. Let $t_0, t_\omega \in [T]$ and $i \in [n_l]$ for an $l \in [L]$ such that $t_0 \leq t_\omega$. If $\beta = 1$, then $0 \leq u_i^{[l]}(t_\omega) < \vartheta$ holds if and only if

$$\left\lfloor \frac{1}{\vartheta} \left(u_i^{[l]}(t_0 - 1) + \sum_{t=t_0}^{t_\omega} (i_i^{[l]}(t) + b_i^{[l]}) \right) \right\rfloor = \sum_{t=t_0}^{t_\omega} s_i^{[l]}(t). \quad (11)$$

Proof. We have

$$u_i^{[l]}(t_\omega) = u_i^{[l]}(t_0 - 1) + \sum_{k=t_0}^{t_\omega} (i_i^{[l]}(k) + b_i^{[l]} - \vartheta s_i^{[l]}(k))$$

by Lemma 2.1. So (11) is equal to $0 \leq u_i^{[l]}(t_\omega) < \vartheta$ by Lemma 2.2. □

Lemma 2.4. Let $t \in [T]$, $l \in [L]$ and $i \in [n_l]$. We then have $u_i^{[l]}(t) \geq 0 \Leftrightarrow p_i^{[l]}(t) \geq 0$.

Proof. Let $u_i^{[l]}(t) \geq 0$. Then $p_i^{[l]}(t) = u_i^{[l]}(t) + \vartheta s_i^{[l]}(t) \geq 0$.

If we know $p_i^{[l]}(t) \geq 0$ instead, then suppose $u_i^{[l]}(t) < 0$. Since $p_i^{[l]}(t) \neq u_i^{[l]}(t)$, $s_i^{[l]}(t) = 1$ and therefore $p_i^{[l]}(t) \geq \vartheta$. But this means $u_i^{[l]}(t) = p_i^{[l]}(t) - \vartheta \geq 0$. □

Lemma 2.5. Let $t_0, t_\omega \in [T]$ and $i \in [n_l]$ for an $l \in [L]$ such that $t_0 \leq t_\omega$. If $\forall_{t \in \{t_0+1, \dots, t_\omega\}} i_i^{[l]}(t) + b_i^{[l]} \geq 0$ and $u_i^{[l]}(t_0) \geq 0$, then $u_i^{[l]}(t_\omega) \geq 0$.

Proof. Suppose there is a t , $t_0 \leq t \leq t_\omega$ with $u_i^{[l]}(t) < 0$. W.l.o.g. we can assume t to be minimal. Clearly $t \neq t_0$, since this contradicts our assumption. So we have $u_i^{[l]}(t-1) \geq 0$ and $i_i^{[l]}(t) + b_i^{[l]} \geq 0$. So from

$$0 > u_i^{[l]}(t) = p_i^{[l]}(t) - \vartheta s_i^{[l]}(t) = \beta^{[l]} u_i^{[l]}(t-1) + \beta^{[l]} (i_i^{[l]}(t) + b_i^{[l]}) - \vartheta s_i^{[l]}(t).$$

we conclude $s_i^{[l]}(t) = 1$ and $p_i^{[l]}(t) \geq \vartheta$. But this means $u_i^{[l]}(t) \geq 0$ by Lemma 2.4. □

Lemma 2.6. Let $t_0, t_\omega \in [T]$ and $i \in [n_l]$ for an $l \in [L]$ such that $t_0 \leq t_\omega$. If $\forall_{t \in \{t_0, \dots, t_\omega\}} i_i^{[l]}(t) + b_i^{[l]} \leq \vartheta$ and $u_i^{[l]}(t_0 - 1) < \vartheta$, then

$$\forall_{t \in \{t_0-1, \dots, t_\omega\}} u_i^{[l]}(t) < \vartheta. \quad (12)$$

Proof. We proof (12) by induction over $t \in \{t_0 - 1, \dots, t_\omega\}$. The base case is given by assumption. Let further $t \in \{t_0 \dots t_\omega\}$. By definition of $p^{[l]}$, the given assumptions and the induction hypothesis we get

$$p_i^{[l]}(t) = \beta^{[l]} u_i^{[l]}(t-1) + i_i^{[l]}(t) + b_i^{[l]} \leq u_i^{[l]}(t-1) + i_i^{[l]}(t) + b_i^{[l]} < 2\vartheta$$

We further get $u_i^{[l]}(t) < \vartheta$ by definition of $u^{[l]}$ and $s^{[l]}$. \square

Lemma 2.7. *Let $t_0, t_\omega \in [T]$ and $i \in [n_l]$ for an $l \in [L]$ such that $t_0 \leq t_\omega$. If $\forall_{t \in \{t_0+1 \dots t_\omega\}} i_i^{[l]}(t) + b_i^{[l]} \leq 0$ and $u_i^{[l]}(t_\omega) \geq \vartheta$, then $\forall_{t \in \{t_0 \dots t_\omega\}} u_i^{[l]}(t) \geq \vartheta$.*

Proof. Suppose there is a t , $t_0 \leq t \leq t_\omega$, with $u_i^{[l]}(t) < \vartheta$. Let t be maximal with this property. Clearly $t \neq t_\omega$ by assumption. So we have a contradiction by

$$\vartheta \leq u_i^{[l]}(t+1) - \beta^{[l]}(i_i^{[l]}(t+1) + b_i^{[l]}) + \vartheta s_i^{[l]}(t) = u_i^{[l]}(t).$$

\square

Lemma 2.8. *Let $t_0, t_\omega \in [T]$ and $i \in [n_l]$ for an $l \in [L]$ such that $t_0 \leq t_\omega$. If both*

$$0 \leq (\beta^{[l]})^{t_\omega - t_0} u_i^{[l]}(t_0) + \sum_{t=t_0+1}^{t_\omega} (\beta^{[l]})^{t_\omega - t} (i_i^{[l]}(t) + b_i^{[l]}) \quad (13)$$

and $\forall_{t \in \{t_0+1, \dots, t_\omega\}} s_i^{[l]}(t) = 0$ or $\forall_{t \in \{t'+1, \dots, t_\omega\}} i_i^{[l]}(t) + b_i^{[l]} \geq 0$, where t' is the maximal time $\leq t_\omega$ such that $s_i^{[l]}(t') = 1$, then $u_i^{[l]}(t_\omega) \geq 0$.

Proof. If $\forall_{t \in \{t_0 \dots t_\omega\}} s_i^{[l]}(t) = 0$, we get by assumption

$$u_i^{[l]}(t_\omega) = u_i^{[l]}(t_\omega) + \sum_{t=t_0+1}^{t_\omega} (\beta^{[l]})^{t_\omega - t} s_i^{[l]}(t) = (\beta^{[l]})^{t_\omega - t_0} u_i^{[l]}(t_0) + \sum_{t=t_0+1}^{t_\omega} (\beta^{[l]})^{t_\omega - t} (i_i^{[l]}(t) + b_i^{[l]}) \geq 0.$$

Is there on the other hand a t' with $s_i^{[l]}(t') = 1$, then let t' be maximal $\leq t_\omega$. By assumption we now have $\forall_{t \in \{t'+1 \dots t_\omega\}} i_i^{[l]}(t) + b_i^{[l]} \geq 0$. Since $s_i^{[l]}(t') = 1$ we further have $p_i^{[l]}(t') \geq \vartheta$, so by Lemma 2.4 $u_i^{[l]}(t') \geq 0$ such that we can conclude by Lemma 2.5. \square

The following two propositions show that a neuron in a r. LIF-SNN has an internal “linear” structure.

Proposition 2.1. *Let $\beta = 1$, $t_0, t_\omega \in [T]$ and $i \in [n_l]$ for an $l \in [L]$ such that $t_0 \leq t_\omega$. Suppose $u_i^{[l]}(t_0 - 1) < \vartheta$ and further $\forall_{t \in \{t_0, \dots, t_\omega\}} i_i^{[l]}(t) + b_i^{[l]} \leq \vartheta$. If moreover both*

$$0 \leq u_i^{[l]}(t_0 - 1) + \sum_{t=t_0}^{t_\omega} (i_i^{[l]}(t) + b_i^{[l]})$$

and $\forall_{t \in \{t_0, \dots, t_\omega\}} s_i^{[l]}(t) = 0$ or $\forall_{t \in \{t'+1, \dots, t_\omega\}} i_i^{[l]}(t) + b_i^{[l]} \geq 0$, where t' is the last time $\leq t_\omega$ such that $s_i^{[l]}(t') = 1$, then

$$\sum_{t=t_0}^{t_\omega} s_i^{[l]}(t) = \left\lfloor \frac{1}{\vartheta} \left(u_i^{[l]}(t_0 - 1) + \sum_{t=t_0}^{t_\omega} (i_i^{[l]}(t) + b_i^{[l]}) \right) \right\rfloor.$$

Proof. It suffices to show $0 \leq u_i^{[l]}(t_\omega) < \vartheta$ due to Lemma 2.3. We get $0 \leq u_i^{[l]}(t_\omega)$ due to Lemma 2.8 and $u_i^{[l]}(t_\omega) < \vartheta$ due to Lemma 2.6. \square

Proposition 2.2. *Let $\beta = 1$, $t_0, t_m, t_\omega \in [T]$ and $i \in [n_l]$ for an $l \in [L]$ such that $t_0 \leq t_m \leq t_\omega$. If $\forall_{t \in \{t_m+1, \dots, t_\omega\}} i_i^{[l]}(t) + b_i^{[l]} = 0$, $u_i^{[l]}(t_m) \geq 0$ and*

$$0 \leq u_i^{[l]}(t_0 - 1) + \sum_{t=t_0}^{t_m} (i_i^{[l]}(t) + b_i^{[l]}) \leq \vartheta(t_\omega - t_m + 1),$$

then

$$\sum_{t=t_0}^{t_\omega} s_i^{[l]}(t) = \left\lfloor \frac{1}{\vartheta} \left(u_i^{[l]}(t_0 - 1) + \sum_{t=t_0}^{t_m} (i_i^{[l]}(t) + b_i^{[l]}) \right) \right\rfloor.$$

Proof. It suffices to show $0 \leq u_i^{[l]}(t_\omega) < \vartheta$ due to [Lemma 2.3](#). We get $u_i^{[l]}(t_\omega) \geq 0$ from [Lemma 2.5](#) using our assumptions, in particular $u_i^{[l]}(t_m) \geq 0$. Furthermore, $u_i^{[l]}(t_\omega) < \vartheta$; otherwise, if $u_i^{[l]}(t_\omega) \geq \vartheta$, then we had $\forall_{t \in \{t_m \dots t_\omega\}} u_i^{[l]}(t) \geq \vartheta$ by [Lemma 2.7](#) and therefore in particular $\forall_{t \in \{t_m \dots t_\omega\}} s_i^{[l]}(t) = 1$, from which we get

$$\begin{aligned} u_i^{[l]}(t_\omega) &= u_i^{[l]}(t_0 - 1) + \sum_{k=t_0}^{t_\omega} (i_i^{[l]}(k) + b_i^{[l]} - \vartheta s_i^{[l]}(k)) \\ &= u_i^{[l]}(t_0 - 1) + \sum_{k=t_0}^{t_m} (i_i^{[l]}(k) + b_i^{[l]}) - \vartheta \sum_{k=t_0}^{t_\omega} s_i^{[l]}(k) \\ &\leq \vartheta(t_\omega - t_m + 1) - \vartheta(t_\omega - t_m + 1) - \vartheta \sum_{k=t_0}^{t_m-1} s_i^{[l]}(k) \\ &\leq 0 \end{aligned}$$

contradicting $u_i^{[l]}(t_\omega) \geq \vartheta$. □

3 Structure of computations in r. LIF-SNNs

When working with neural networks a fundamental question is how well they are able to approximate functions. Towards that end the following theorem was proved in [Nguyen et al., 2025].

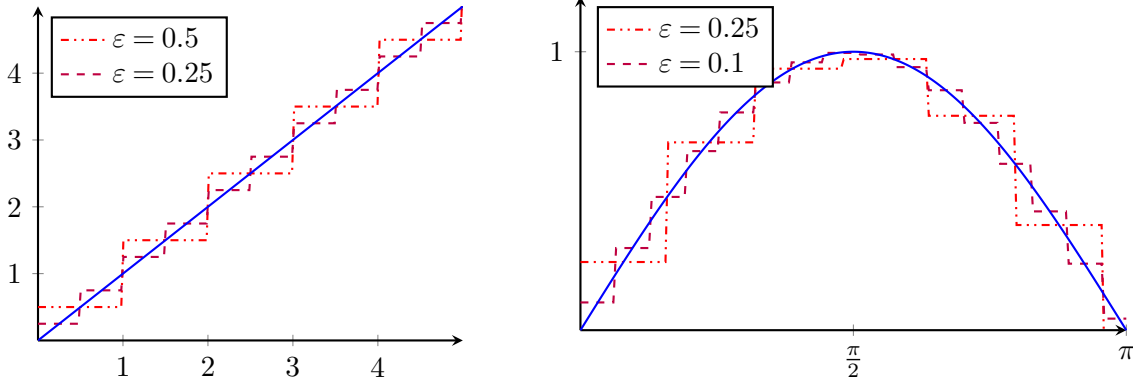
Theorem 3.1. *Let f be a continuous function on a compact set $\Omega \subset \mathbb{R}^{n_0}$. For all $\varepsilon > 0$, there exists a d.t. LIF-SNN Φ with direct encoding, membrane potential output, $L = 2$ and $T = 1$ such that*

$$\|(R(\Phi) - f)|_{\Omega}\|_{\infty} \leq \varepsilon$$

Moreover, if f is Γ -Lipschitz, then Φ can be chosen with width parameters (n_1, n_2) given by

$$n_1 = \left(\max \left\{ \left\lceil \frac{\text{diam}_{\infty}(\Omega)}{\varepsilon} \Gamma \right\rceil, 1 \right\} + 1 \right) n_0,$$

$$n_2 = \max \left\{ \left\lceil \frac{\text{diam}_{\infty}(\Omega)}{\varepsilon} \Gamma \right\rceil^{n_0}, 1 \right\}.$$



(a) A d.t. LIF-SNN approximating the identity

(b) d.t. LIF-SNNs approximating a sinus wave

The proof of Theorem 3.1 first shows that a continuous function can be arbitrarily approximated by step functions, in particular by step functions constant on hypercubes in Ω . Then a d.t. LIF-SNN is constructed by using the first layer to partition the input space along hyperplanes into cubes and the second layer to assign values to the hypercubes.

While quite simple, this construction does not use the unique feature of d.t. LIF-SNNs/r. LIF-SNNs, the ability of neurons to accumulate state over time. It therefore needs quite a lot more neurons than actually needed for many functions with (almost) linear segments, like a sinus wave. E.g. in Fig. 3.1b a neuron is needed for every constant region of the graphs in the first and second layer each.

We will now show a more efficient construction for r. LIF-SNN using the fact that r. LIF-SNN can quite efficiently approximate linear segments. The general intuition behind it is to use piece-wise linear functions to approximate continuously differentiable functions and then construct a r. LIF-SNN approximating the piece-wise linear function by discretizing the input dimensions into spike trains in the first layer that are consumed by groups of neurons in the second layer, one group for each almost linear segment.

To state our theorem we first need to define the notions of “modulus of uniform continuity” and “generalized inverse of a modulus of uniform continuity” and proof some simple properties:

Definition 3.1. Let M, N be metric spaces. A modulus of uniform continuity of a uniformly continuous function $f : M \rightarrow N$ is a function $\omega : [0, \infty] \rightarrow [0, \infty]$, such that it vanishes at 0, i.e. $\lim_{x \rightarrow 0} \omega(x) = 0$, and

$$\forall_{x, y \in M} d_N(f(x), f(y)) \leq \omega(d_M(x, y)).$$

The generalized inverse of ω is defined as

$$\omega^\dagger(s) := \inf\{t \in [0, \infty] \mid \omega(t) > s\}.$$

Lemma 3.1. *Let $\omega : [0, \infty] \rightarrow [0, \infty]$ be a modulus of uniform continuity of a uniformly continuous function $f : M \rightarrow N$, where M, N are metric spaces.*

We have the following properties

1. $\forall_{x,y \in M, s \in [0, \infty]} d_M(x, y) \leq \omega^\dagger(s) \Rightarrow d_N(f(x), f(y)) \leq s.$
2. $\forall_{s \in [0, \infty]} s = 0 \Leftrightarrow \omega^\dagger(s) = 0.$

Proof.

1. Let $x, y \in M$ and $s \in [0, \infty]$ be given such that $d_M(x, y) \leq \omega^\dagger(s)$. By definition of ω^\dagger , this means $\omega(d_M(x, y)) \leq s$. Since ω is a modulus of uniform continuity of f , we have $d_N(f(x), f(y)) \leq \omega(d_M(x, y))$ and therefore overall $d_N(f(x), f(y)) \leq s$.
2. Since ω is a modulus of uniform continuity, it is by definition continuous at 0. Let us choose an arbitrary sequence $(t_n)_{n \in \mathbb{N}}$ with $t_n \rightarrow 0$. Then $\omega(t_n) \rightarrow 0$ and therefore $\omega^\dagger(0) \leq \inf_{n \in \mathbb{N}} t_n = 0$.

Is on the other hand $\omega^\dagger(s) = 0$, then there is a sequence $(t_n)_{n \in \mathbb{N}}$ with $t_n \rightarrow 0$, such that $\omega(t_n) > s$ by definition of ω^\dagger . But since $\omega(t_n) \rightarrow 0$ by definition of ω , we get $s = 0$.

□

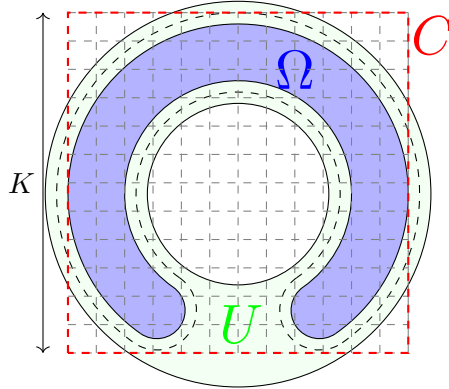


Figure 3.2: A possible configuration of the sets in Theorem 3.2

Let us now state our theorem:

Theorem 3.2. *Let a $f \in \mathcal{C}^0(U, \mathbb{R}^n)$ be a continuous function, such that $f|_{U^\circ} \in \mathcal{C}^1(U^\circ, \mathbb{R}^m)$ is continuously differentiable with bounded differential, i.e. $\|d(f|_{U^\circ})\|_{\infty, 2} < \infty$. Let further $\emptyset \neq \Omega \subset U$ be an arbitrary non-empty subset and $C \subset \mathbb{R}^n$ an half-open cube such that $\Omega \subset C$ and $(\Omega + B_{\rho, 2}(0)) \cap C^\circ \subset U^\circ$ for a $\rho > 0$.*

For all $\varepsilon, \mu, \nu > 0$, $\varepsilon = \mu + \nu$, there exists a r . LIF-SNN Φ with $L = 2$ and

$$\begin{aligned} T &= (K(\mu) + 1)T_r(\nu) + 2 \\ n_1 &= n + 1 \\ n_2 &= K(\mu)^n(n + 1) + 3 \end{aligned}$$

such that

$$\|R(\Phi)|_\Omega - f\|_{\infty, 2} \leq \varepsilon.$$

Where we use

$$T_r := T_r(\nu) := \max \left(2, \left\lceil \sqrt{n} \frac{\text{diam}_\infty(C)}{K} \frac{\|d(f|_{U^\circ})\|_{\infty,2}}{\nu} \right\rceil \right),$$

$$K := K(\mu) := \min_{\substack{\xi, \theta > 0 \\ \xi\theta = \mu}} \left\{ \left\lceil \frac{\text{diam}_\infty(C)}{\frac{2}{\sqrt{n}} \min(\omega^\dagger(\xi), \theta, \frac{\rho}{2})} \right\rceil \right\}.$$

Here ω^\dagger is the generalized inverse of a modulus of uniform continuity with regard to $\|\cdot\|_2$ of the total derivative $d(f|_{U^\circ})$. $d(f|_{U^\circ})$ is uniformly continuous since it is bounded. Since $\xi \neq 0$, we have $\omega^\dagger(\xi) > 0$ by [Lemma 3.1](#). Further there are ξ, θ such that the minimum in the definition of K^n is obtained, since we the minimum is taken over the set of natural numbers, so K is well-defined. Further $\min(\omega^\dagger(\xi), \theta) < \infty$ as well as $\text{diam}_\infty(C) \neq 0$, since $\Omega \subset C$ and $\Omega \neq \emptyset$. We therefore have $K \neq 0$, such that T_r is well-defined.

Remark 3.1. Let $f : C \rightarrow \mathbb{R}^n$ be defined on a half-open cube C and $\Omega = C$. We can then choose an arbitrarily big ρ , so we can drop $\frac{\rho}{2}$ from the definition of K ,

$$K = \min_{\substack{\xi, \theta > 0 \\ \xi\theta = \mu}} \left\{ \left\lceil \frac{\text{diam}_\infty(C)}{\frac{2}{\sqrt{n}} \min(\omega^\dagger(\xi), \theta)} \right\rceil \right\}.$$

Remark 3.2. If the differential $d(f|_{U^\circ})$ is L -Lipschitz in addition to the conditions in [Theorem 3.2](#), we can simplify the definition of K . Let us choose the canonical modulus of uniform continuity $\omega(x) := Lx$, such that $\omega^\dagger(x) = \frac{1}{L}x$ (with $\frac{1}{L} = \infty$ for $L = 0$). Now for $L > 0$ we have

$$\max_{\substack{\xi, \theta > 0 \\ \xi\theta = \mu}} \min(\omega^\dagger(\xi), \theta) = \max_{\substack{\xi, \theta > 0 \\ \xi\theta = \mu}} \min\left(\frac{1}{L}\xi, \frac{\mu}{\xi}\right) = \sqrt{\frac{\mu}{L}}$$

since $\frac{1}{L}\xi$ is monotonically increasing in ξ , $\frac{\mu}{\xi}$ monotonically decreasing in ξ and $\frac{1}{L}\xi = \frac{\mu}{\xi}$ equivalent to $\xi = \sqrt{\mu L}$ for $\xi > 0$. So we have

$$K = \left\lceil \frac{\text{diam}_\infty(C)}{\frac{2}{\sqrt{n}} \min(\sqrt{\frac{\mu}{L}}, \frac{\rho}{2})} \right\rceil.$$

For $L = 0$ we have $\min(\omega^\dagger(\xi), \theta) = \theta$, so $K = 1$. We therefore get

$$K = 1 \vee \left\lceil \frac{\text{diam}_\infty(C)}{\frac{2}{\sqrt{n}} \min(\sqrt{\frac{\mu}{L}}, \frac{\rho}{2})} \right\rceil$$

for $L \geq 0$ arbitrary.

Remark 3.3. The conditions of the theorem regarding U, Ω, C might seem unnecessarily complex at first, but they are indeed necessary to approximate some functions efficiently. For our construction of the r. LIF-SNN we need piece-wise affine-linear approximations of f , defined on subcubes of a cube $C \subset \mathbb{R}^n$. Suppose, the theorem just required f to be defined on that cube C instead and further $\Omega = C$.

Now picture a function f like the one shown in [Fig. 3.3](#). If we want to approximate the whole complete input space Ω of f , we need to put a half-open square C around the arc due to our construction. To further use the theorem with $U = \Omega = C$ we would need to have differentiability on the whole of C° , but f is only defined on an arc inside of C . This problem can be handled by replacing f with an extension of f , defined on all of \mathbb{R}^n , but now we have the problem that we

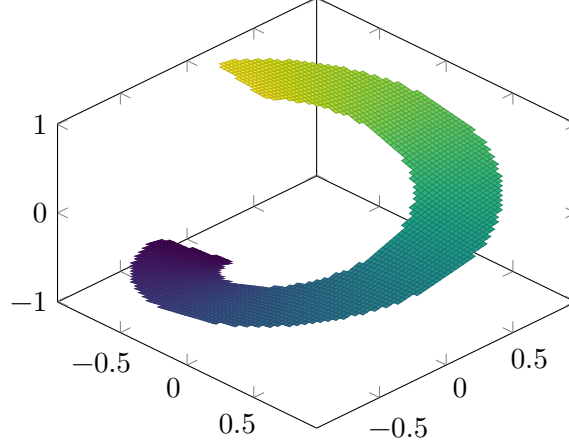


Figure 3.3: A spiral staircase function

might get a very steep graph around $(-0.5, 0.0)$, such that $\|d(f|_{C^\circ})\|_{\infty, 2}$ is unnecessarily large and changes unnecessarily quick. So T_r and K would get excessively large.

On the other hand, [Theorem 3.2](#) only uses the values of df on a thin region around Ω to determine how large K and T_r need to be.

Remark 3.4. In our construction μ and ν determine whether to optimize the number of neurons or the number of time-steps. As we see later, K^n corresponds to the number of subcubes we will split C into such that f is almost linear on each of them. From the definition it is clear that K and therefore the number of neurons only depend on f through $\omega^\dagger(\xi)$, which is essentially a measure of how strongly the slope of f is changing and therefore into how many subcubes we need to split C to get sufficiently almost affine linear regions of f .

We will further see that T_r corresponds to the number of constant intervals with which we approximate f on the almost affine linear regions. It is therefore to be expected that T_r depends on the width $\frac{\text{diam}_\infty(C)}{K}$ of the subcubes and the maximal slope $\|d(f|_{U^\circ})\|_{\infty, 2}$.

We first proof that continuous differentiable functions with uniform continuous differential can be efficiently approximated by piece-wise linear function. Compare e.g. [Fig. 3.4](#) to [Fig. 3.1b](#)

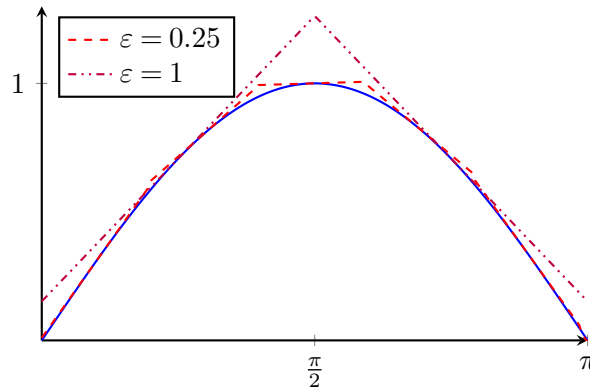


Figure 3.4: Piece-wise affine linear functions approximating the sinus

Lemma 3.2. *Let a $f \in \mathcal{C}^0(U, \mathbb{R}^n)$ be a continuous function, such that $f|_{U^\circ} \in \mathcal{C}^1(U^\circ, \mathbb{R}^m)$ is continuously differentiable with a uniform continuous differential $d(f|_{U^\circ})$. Let further $\emptyset \neq \Omega \subset U$ be an arbitrary non-empty subset and $C \subset \mathbb{R}^n$ an half-open cube such that $\Omega \subset C$ and $(\Omega + B_{\rho, 2}(0)) \cap C^\circ \subset U^\circ$ for a $\rho > 0$. Let further K be defined as in [Theorem 3.2](#).*

For every $\mu > 0$ we can compose C into K^n half-open subcubes $(C^{(j)})_{j \in [K^n]}$ such that affine

linear functions $g^{(j)} : C^{(j)} \rightarrow \mathbb{R}^m$ exist with $\|d(g^{(j)})\|_{\infty,2} \leq \|d(f|_{U^\circ})\|_{\infty,2}$ and $\|(f-g)|_\Omega\|_{\infty,2} < \mu$, where $g := \sum_{i=1}^m g^{(j)} \chi_{C^{(j)}}$.

Proof of Lemma 3.2. Let $\mu > 0$ be given. Let ω be a modulus of uniform continuity of $df|_{U^\circ}$. We will now partition C in K^n half-open subcubes with K defined as in Theorem 3.2. Let ξ, Θ be given, such that the minimum in the definition of K is attained. Then the subcubes have width

$$w := \frac{\text{diam}_\infty(C)}{K} \leq \frac{2}{\sqrt{n}} \min(\omega^\dagger(\xi), \theta, \frac{\rho}{2}).$$

Let now $c^{(j)}$ be the center of $C^{(j)}$, such that in particular for all $x \in C^{(j)}$

$$\|x - c^{(j)}\|_2 = \sqrt{\sum_{i=1}^n |(x - c^{(j)})_i|^2} \leq \frac{w}{2} \sqrt{n} \leq \min(\omega^\dagger(\xi), \theta, \frac{\rho}{2}). \quad (14)$$

Let us further define $g^{(j)} : C^{(j)} \rightarrow \mathbb{R}^m$ by

$$g^{(j)} := \begin{cases} x \mapsto f(c^{(j)}) + df_{c^{(j)}}(x - c^{(j)}), & C^{(j)} \cap \Omega \neq \emptyset \\ x \mapsto 0, & C^{(j)} \cap \Omega = \emptyset \end{cases}.$$

The first case is well-defined, since $C^{(j)} \cap \Omega \neq \emptyset$ implies that there exists a $x \in \mathbb{R}^n$ with $x \in \Omega$ and $\forall_{y \in C^{(j)}} \|x - y\|_2 \leq \rho$ by (14), so $C^{(j)} \subset (\Omega + \overline{B}_{\rho,2}(0)) \cap C$ and therefore $(C^{(j)})^\circ \subset (\Omega + \overline{B}_{\rho,2}(0)) \cap C^\circ \subset U^\circ$. Now by definition of $g^{(j)}$ we already have $\|d(g^{(j)})\|_{\infty,2} = \|df_{c^{(j)}}\| \leq \|d(f|_{U^\circ})\|_{\infty,2}$ for $C^{(j)} \cap \Omega \neq \emptyset$ and otherwise $0 = \|d(g^{(j)})\|_{\infty,2} \leq \|d(f|_{U^\circ})\|_{\infty,2}$.

It only remains to show $\|(f - g^{(j)})|_{\Omega \cap C^{(j)}}\|_\infty < \mu$. This is clearly trivial for $C^{(j)} \cap \Omega = \emptyset$, so suppose $C^{(j)} \cap \Omega \neq \emptyset$. Let $x \in C^{(j)}$ and $h(t) := f(t(x - c^{(j)}) + c^{(j)})$. We then have

$$h'(t) = (df_{(x-c^{(j)})t+c^{(j)}} \circ d(t \mapsto t(x - c^{(j)}) + c^{(j)}))_t(1) = df_{(x-c^{(j)})t+c^{(j)}}(x - c^{(j)}).$$

Since $(C^{(j)})^\circ \subset U^\circ$, we obtain by the fundamental theorem of calculus

$$\begin{aligned} \|f(x) - g^{(j)}(x)\|_2 &= \|f(x) - f(c^{(j)}) - df_{c^{(j)}}(x - c^{(j)})\|_2 \\ &= \|h(1) - h(0) - df_{c^{(j)}}(x - c^{(j)})\|_2 \\ &= \left\| \int_0^1 df_{(x-c^{(j)})t+c^{(j)}}(x - c^{(j)}) dt - df_{c^{(j)}}(x - c^{(j)}) \right\|_2. \end{aligned}$$

Now, due to the generalized Minkowski-Inequality we can move the norm inside the integral:

$$\begin{aligned} &\leq \int_0^1 \|df_{(x-c^{(j)})t+c^{(j)}}(x - c^{(j)}) - df_{c^{(j)}}(x - c^{(j)})\|_2 dt \\ &= \int_0^1 \|(df_{(x-c^{(j)})t+c^{(j)}} - df_{c^{(j)}})(x - c^{(j)})\|_2 dt \\ &\leq \int_0^1 \|df_{(x-c^{(j)})t+c^{(j)}} - df_{c^{(j)}}\|_2 \|x - c^{(j)}\|_2 dt \\ &\leq \int_0^1 \xi \|x - c^{(j)}\|_2 dt \\ &= \xi \|x - c^{(j)}\|_2 \\ &\leq \xi \theta \\ &= \mu \end{aligned}$$

In the fourth step we use $\|df_{(x-c^{(j)})t+c^{(j)}} - df_{c^{(j)}}\| \leq \xi$, which holds due to $\forall_{t \in [0,1]} (x - c^{(j)})t + c^{(j)} \in C^{(j)}$, (14) and Lemma 3.1. \square

Proof of Theorem 3.2.. Let there be $\varepsilon, \mu, \nu > 0$ with $\varepsilon = \mu + \nu$. By Lemma 3.2 we have a composition K^n of C into half-open subcubes $(C^{(j)})_{j=1..K^n}$ and linear functions $g^{(j)} : C^{(j)} \rightarrow \mathbb{R}^m$, such that $\|d(g^{(j)})\|_{\infty,2} \leq \|df\|_{\infty,2}$ and $\|f - g\|_{\infty} < \mu$ for $g := \sum_{j=1}^m g^{(j)} \chi_{C^{(j)}}$.

We will now define a r. LIF-SNN Φ with direct input encoding and membrane-potential outputs such that $\|R(\Phi)|_C - g\|_{\infty} < \nu$.

Let us first set the basic parameters $i^{[l]}(0) = 0$, $\alpha^{[l]} = 0$ and $\beta^{[l]} = \vartheta^{[l]} = 1$ for all layers.

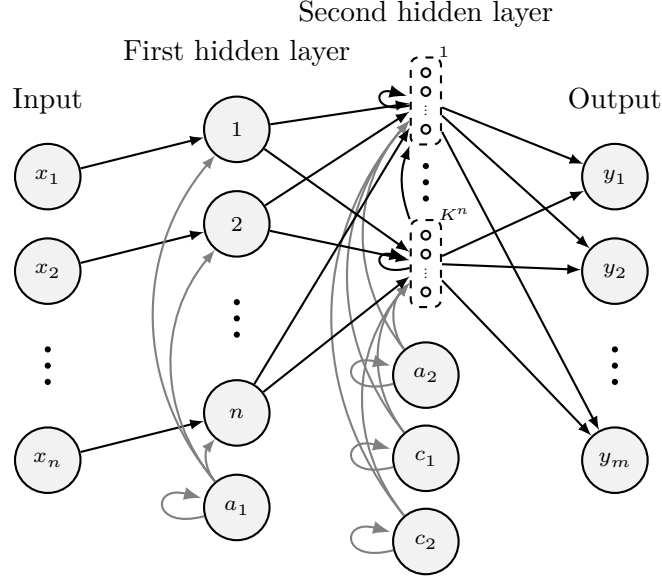


Figure 3.5: Structure of the whole network

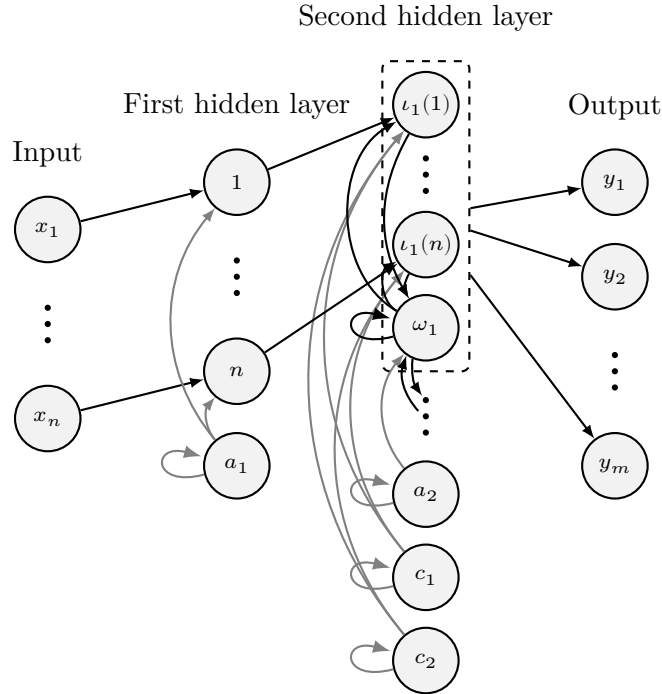


Figure 3.6: Structure of the network, focused on the j -th group of the second layer

For our construction and proof we use the following five phases. We define

$$\begin{aligned} T_1 &:= \{1, \dots, KT_r\}, & T_2 &:= \{KT_r\}, & T_3 &:= \{KT_r + 1\}, \\ T_4 &:= \{KT_r + 2\}, & T_5 &:= \{KT_r + 3, \dots, T\}. \end{aligned}$$

Note that T_2 overlaps with T_1 . For ease of notation, we will also use T_2, T_3, T_4 as if they were numbers.

During the following proof it will be helpful to look at Fig. 3.5 and Fig. 3.6 to get a visual feeling for the constructed network. For understanding the procedure of the network it will be useful to compare the results with the timelines in Fig. 3.7 and Fig. 3.8.

The higher level idea is that we accumulate the state during T_1 , compute the subregion $C^{(j)}$ of the input during T_2, \dots, T_4 and flush out the position of the input inside of $C^{(j)}$ during T_5 .

The constructed first layer is only active during the first phase, T_1 . It is composed of $n + 1$ neurons, where the first n neurons convert the input vector regarding its position in C into spike trains. The last neuron, the “alarm clock”, shuts down the first layer after T_1 ends.

The second layer only accumulates state without spike during $T_1 \setminus T_2$. Then during $T_2 \cup T_3 \cup T_4$ it is decided in which region $C^{(j)}$ the input x is located. Further during $T_4 \cup T_5$ the location in $C^{(j)}$ is encoded through spikes.

For each region $C^{(j)}$ we have $n + 1$ neurons in the second layer. Each of the first n neurons encodes a component of the linear part of $g^{(j)}$. They are also used to inform the $n + 1$ -th neuron of the group if the x has at least as big as the base point of $C^{(j)}$. The $n + 1$ -th deactivates all other neurons of regions with smaller base point and encodes the constant part of $g^{(j)}$. The last 3 neurons act as “clock neurons” enabling and disabling the other ones.

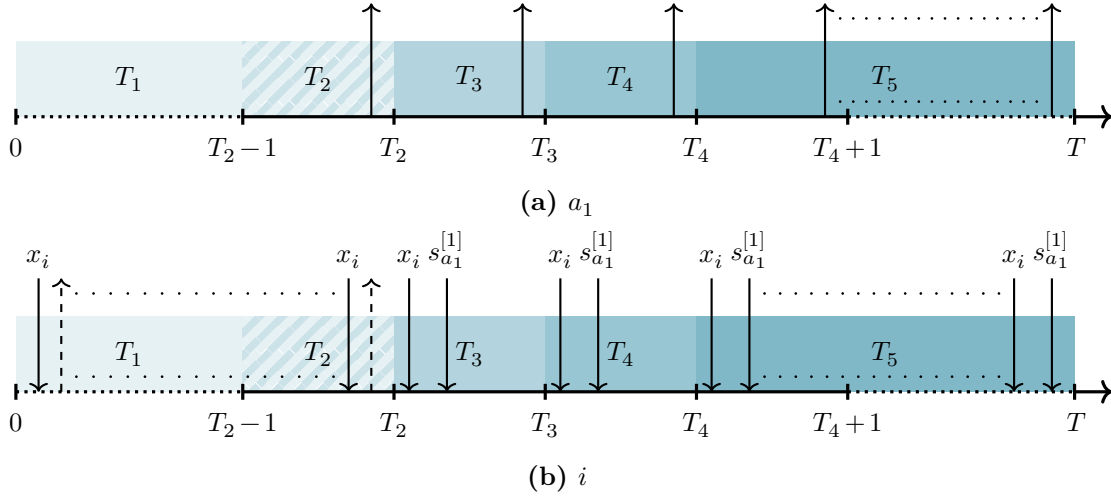


Figure 3.7: Timelines of first layer neurons

To obtain the normalized location of a value in C we will often use

$$o_i(z) = \frac{z_i - x_i^C}{y_i^C - x_i^C}$$

with $i \in [n]$ in the following.

1. **First layer:** We define the i -th neuron of the n neurons of the first layer by parameters

$$w = \frac{1}{y_i^C - x_i^C} e_i, \quad b = -\frac{x_i^C}{y_i^C - x_i^C}, \quad v = -e_{a_1}, \quad u_0 = 0, \quad i_0 = 0. \quad (i)$$

The “alarm neuron” of the first layer, with index $a_1 := n + 1$, is defined by:

$$w = 0, \quad b = \frac{1}{T_2}, \quad v = e_{a_1}, \quad u_0 = 0, \quad i_0 = 0. \quad (a_1)$$

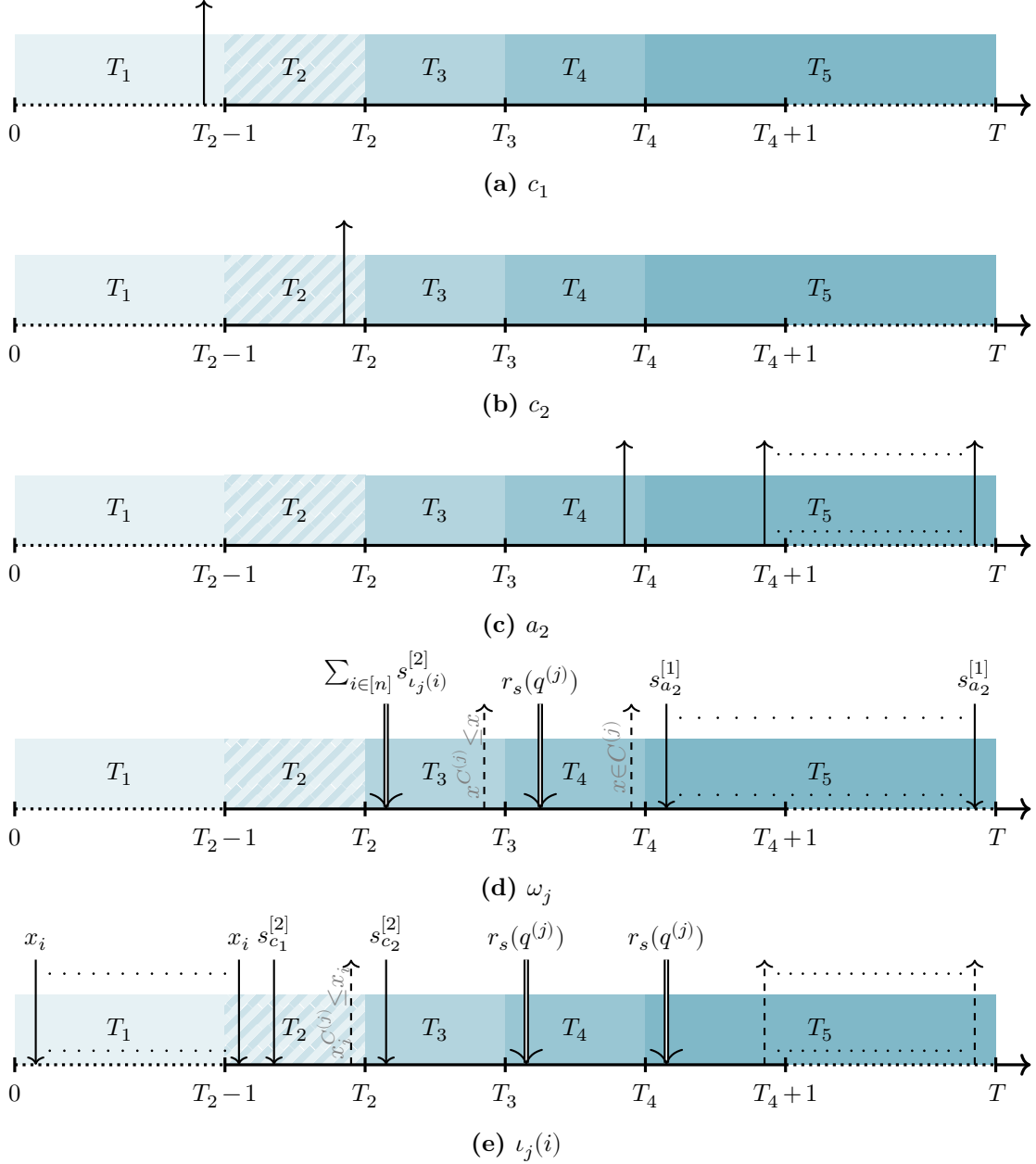


Figure 3.8: Timelines of second layer neurons

2. **Second layer:** Let us now construct the second layer in the following way: For each of the K^n subcubes in C we define $n+1$ neurons like so: Let $C^{(j)} = \llbracket x^{C^{(j)}}, y^{C^{(j)}} \rrbracket$ be one such subcube with position $q^{(j)} \in ([K-1]_0)^n$ in C , i.e.

$$\forall_{i \in [n]} q_i^{(j)} = K o_i(x^{C^{(j)}}).$$

We will write $\iota_j(i) := j(n+1) + i$ to index the first n neurons in the layer and $\omega_j := (j+1)(n+1)$ to index the last neuron of each group.

The i -th neuron of the first n neurons (of the j -th group), with index $\iota_j(i)$ in the second layer, has the parameters

$$\begin{aligned} w &= e_i, \quad b = 0, \quad v = T(e_{c_1} - 2e_{c_2} + r(q^{(j)})), \\ u_0 &= -q_i^{(j)}T_r - T + 1, \quad i_0 = 0. \end{aligned} \tag{\iota_j(i)}$$

where “the switch” is

$$r(q) := e_{\omega_{j(q)}} - \sum_{\substack{q' \in ([K-1]_0)^n \\ q < q'}} e_{\omega_{j(q')}}.$$

with the index $j(q)$ of the subcube at position q . We further define the applied variant

$$r_s(q; t) := \langle r(q), s^{[2]}(t) \rangle = s_{\omega_{j(q)}}^{[2]}(t) - \sum_{\substack{q' \in ([K-1]_0)^n \\ q < q'}} s_{\omega_{j(q')}}^{[2]}(t).$$

The **final neuron** of the group, with index ω_j in its layer, has the parameters

$$w = 0, \quad b = 0, \quad v = \frac{1}{n} \sum_{i=1}^n e_{\iota_j(i)} - 2e_{a_2} + r(q^{(j)}), \quad u_0 = 0, \quad i_0 = 0. \quad (\omega_j)$$

We also define the two “**clock neurons**”, with index $c_1 := (n+1)K^n + 1$ and $c_2 := (n+1)K^n + 2$ with parameters:

$$w = 0, \quad b = b_{c_i}, \quad v = -(T-1)e_{c_i}, \quad u_0 = 0, \quad i_0 = 0. \quad (c_1, c_2)$$

where $b_{c_1} = \frac{1}{T_2-1}$ and $b_{c_2} = \frac{1}{T_2}$. We further define the “**alarm neuron**”, with index $a_2 := (j+1)K^n(\mu) + 3$, by

$$w = 0, \quad b = \frac{1}{T_4}, \quad v = e_{a_2}, \quad u_0 = 0, \quad i_0 = 0. \quad (a_2)$$

3. Output decoder: We further define the parameters of the output decoder by $a_t = 0$, for $t \leq T_3$ and otherwise $a_t = 1$. We further set $b^{[L+1]} = 0$ and

$$W_{k, \iota_j(i)}^{[L+1]} = d(g^{(j)})_k ((y_i^{C^{(j)}} - x_i^{C^{(j)}}) \frac{1}{T_r} e_i)$$

for $k \in [m]$, $j \in K^n$ and $i \in [n]$. Here $d(g^{(j)})$ is not only the total derivative, but also the linear part of $g^{(j)}$, i.e. $g^{(j)}(x) = d(g^{(j)})(x - x^{C^{(j)}}) + g(x^{C^{(j)}})$ for all $x \in C^{(j)}$. We further set

$$W_{k, \omega_j}^{[L+1]} = g_k(x^{C^{(j)}})$$

for $k \in [m]$ and $j \in K^n$. We finally define $W_{k, c_i}^{[L+1]} = 0$ for $i \in \{2..4\}$.

We will now proof that this construction indeed approximates g well enough. It will be helpful to consider the following, by choice of $i^{[l]}, \alpha^{[l]}, \beta^{[l]}, \vartheta^{[l]}$, simplified equations:

$$\begin{aligned} i^{[l]}(t) &= W^{[l]} s^{[l-1]}(t) + V^{[l]} s^{[l]}(t-1) \\ p^{[l]}(t) &= u^{[l]}(t-1) + i^{[l]}(t) + b^{[l]} \\ s^{[l]}(t) &= H(p^{[l]}(t) - \mathbf{1}_{n_l}) \\ u^{[l]}(t) &= p^{[l]}(t) - s^{[l]}(t) \end{aligned}$$

and in particular by [Lemma 2.1](#)

$$p^{[l]}(t) = u^{[l]}(0) + \sum_{k=1}^t (i^{[l]}(k) + b^{[l]}) - \sum_{k=1}^{t-1} s^{[l]}(k).$$

Let now $s^{[0]}(t) = x \in C$. We will proof $\|R(\Phi)(x) - g(x)\|_{\infty, 2} \leq \nu$ in steps, by first characterizing the behavior of the first layer:

3 STRUCTURE OF COMPUTATIONS IN R. LIF-SNNS

1. Characterization of the “alarm neuron” a_1 :

Let us first regard the neuron in the first layer: By choice of parameters we get:

$$p_{a_1}^{[1]}(t) = \frac{t}{T_2} + \sum_{k=1}^t s_{a_1}^{[1]}(k-1) - \sum_{k=1}^{t-1} s_{a_1}^{[1]}(k) = \frac{t}{T_2}$$

So $s_{a_1}^{[1]}(t) = 1 \Leftrightarrow t \geq T_2$.

2. Characterization of i -th neuron, $i \in [n]$:

We have

$$i_i^{[1]}(t) + b_i^{[1]} = \frac{x_i - x_i^C}{y_i^C - x_i^C} - s_{a_1}^{[1]}(t-1) = o_i(x) - s_{a_1}^{[1]}(t-1).$$

So in particular

$$0 \leq i_i^{[1]}(t) + b_i^{[1]} = o_i(x) \leq 1$$

for $t \in T_1$, since $x_i^C \leq x_i < y_i^C$. Because further $0 \leq u_i^{[1]}(0) = 0 < 1$ we can use [Proposition 2.1](#) with $t_0 := 0$ and $t_\omega := T_2$ to obtain

$$\lfloor T_2 o_i(x) \rfloor = \left\lfloor u_i^{[1]}(0) + \sum_{t=1}^{T_2} (i_i^{[1]}(t) + b_i^{[1]}) \right\rfloor = \sum_{t=1}^{T_2} s_i^{[1]}(t).$$

We further have

$$i_i^{[1]}(t) + b_i^{[1]} = o_i(x) - s_{a_1}^{[1]}(t-1) = o_i(x) - 1 < 0$$

for $t > T_2$, so we get

$$p_i^{[1]}(t) = u_i^{[1]}(T_2) + \sum_{k=T_3}^t (i_i^{[1]}(k) + b_i^{[1]}) - \sum_{k=T_3}^{t-1} s_i^{[1]}(k) < 1$$

and therefore $s_i^{[1]}(t) = 0$ for $t > T_2$.

We will now continue with characterizing the second layer. We start with “clock neurons” and the “alarm” neuron:

1. Characterization of the “clock neurons”:

In contrast to the alarm neuron of the first layer, the two “clock” neurons only fire once:

$$p_{c_i}^{[2]}(t) = tb_{c_i} - (T-1) \sum_{k=1}^t s_{c_i}^{[2]}(k-1) - \sum_{k=1}^{t-1} s_{c_i}^{[2]}(k) = tb_{c_i} - T \sum_{k=1}^{t-1} s_{c_i}^{[2]}(k)$$

Let us first consider c_1 : We clearly have $p_{c_1}^{[2]}(t) < 1$ for $t < T_2 - 1$, but $p_{c_1}^{[2]}(T_2 - 1) = 1$. Since by definition $K \geq 1$ and $T_r \geq 2$, so $T_2 = KT_r \geq 2$, we have $t \leq T \leq T(T_2 - 1)$. Therefore $p_{c_1}^{[2]}(t) < 1$ for $t > T_2 - 1$ due to $s_{c_1}^{[2]}(T_2 - 1) = 1$. So $\forall_{t \in [T]} s_{c_1}^{[2]}(t) = \chi_{\{T_2-1\}}(t)$.

We similarly obtain $\forall_{t \in [T]} s_{c_2}^{[2]}(t) = \chi_{T_2}(t)$.

2. Characterization of the “alarm neuron”:

Just like for a_1 , we also get $s_{a_2}^{[1]}(t) = 1 \Leftrightarrow t \geq T_4$.

We now proof the behavior of the remaining neurons in the second layer step by step throughout the phases.

1. Phase 1

We will show $\forall_{i \in [n]} s_{\iota_j(i)}^{[2]}(t) = 0$ and $s_{\omega_j}^{[2]}(t) = 0$ for all $j \in [K^n]$ and $t \in [T_2 - 1]_0$ by induction over t . Let $t = 0$. We then get $s_{\iota_j(i)}^{[2]}(0) = s_{\omega_j}^{[2]}(0) = 0$ by definition. Let further $1 \leq t \leq T_2 - 1$. First notice that by induction hypothesis, we have $\forall_{i \in [n]} s_{\iota_j(i)}^{[2]}(k) = 0$ and $r_s(q^{(j)}; k) = 0$ for $k < t$. It follows that

$$\begin{aligned} i_{\iota_j(i)}^{[2]}(k) + b_{\iota_j(i)}^{[2]} &= s_i^{[1]}(k) + T(s_{c_1}^{[2]}(k-1) - 2s_{c_2}^{[2]}(k-1) + r_s(q^{(j)}; k-1)) = s_i^{[1]}(k), \\ i_{\omega_j}^{[2]}(k) + b_{\omega_j}^{[2]} &= \frac{1}{n} \sum_{i=1}^n s_{\iota_j(i)}^{[2]}(k-1) - 2s_{a_2}^{[2]}(k-1) + r_s(q^{(j)}; k-1) = 0. \end{aligned}$$

for all $k \leq t$. Thus, we further get

$$\begin{aligned} p_{\iota_j(i)}^{[2]}(t) &= -q_i^{(j)}T_r - T + 1 + \sum_{k=1}^t s_i^{[1]}(k) - \sum_{k=1}^{t-1} s_{\iota_j(i)}^{[2]}(k) = -q_i^{(j)}T_r - T + 1 + \sum_{k=1}^t s_i^{[1]}(k), \\ p_{\omega_j}^{[2]}(t) &= \sum_{k=1}^t 0 - \sum_{k=1}^{t-1} s_{\omega_j}^{[2]}(k) = 0. \end{aligned}$$

Since $\sum_{k=1}^t s_i^{[1]}(k) \leq T_2 - 1 < T - 1$ and $s_{c_1}^{[2]}(t) = \chi_{\{T_2-1\}}(t)$, we in particular get $p_{\iota_j(i)}^{[2]}(t) \leq 0$. So we have proven $\forall_{i \in [n]} s_{\iota_j(i)}^{[2]}(t) = 0$, $s_{\omega_j}^{[2]}(t) = 0$.

2. Phase 2

Just as in phase 1, we have

$$i_{\omega_j}^{[2]}(T_2) + b_{\omega_j}^{[2]} = \frac{1}{n} \sum_{i=1}^n s_{\iota_j(i)}^{[2]}(T_2 - 1) - 2s_{a_2}^{[2]}(T_2 - 1) + r_s(q^{(j)}; T_2 - 1) = 0.$$

So we also get $p_{\omega_j}^{[2]}(T_2) = 0$ and $s_{\omega_j}^{[2]}(T_2) = 0$. It is different for $\iota_j(i)$ due to the dependence on c_1 . Let $j \in [K^n]$ and $i \in [n]$ be given. The neuron with index $\iota_j(i)$ fires exactly then at T_2 , if $x_i \geq x_i^{C(j)}$: First notice that

$$i_{\iota_j(i)}^{[2]}(T_2) + b_{\iota_j(i)}^{[2]} = s_i^{[1]}(T_2) + T(s_{c_1}^{[2]}(T_2 - 1) - 2s_{c_2}^{[2]}(T_2 - 1) + r_s(q^{(j)}; T_2 - 1)) = s_i^{[1]}(T_2) + T,$$

thus, we conclude that

$$\begin{aligned} p_{\iota_j(i)}^{[2]}(T_2) &= p_{\iota_j(i)}^{[2]}(T_2 - 1) + i_{\iota_j(i)}^{[2]}(T_2) + b_{\iota_j(i)}^{[2]} - s_{\iota_j(i)}^{[2]}(T_2 - 1) \\ &= -q_i^{(j)}T_r + 1 + \sum_{k=1}^{T_2} s_i^{[1]}(k) \\ &= -q_i^{(j)}T_r + 1 + \lfloor T_2 o_i(x) \rfloor \end{aligned}$$

holds using the characterization of layer 1. Therefore we have $s_{\iota_j(i)}^{[2]}(T_2) = 1$ exactly if $q_i^{(j)}T_r \leq T_2 o_i(x)$, which is equivalent to $\frac{q_i^{(j)}}{K}(y_i^C - x_i^C) + x_i^C \leq x_i$ by definition of o_i . Further $\frac{q_i^{(j)}}{K}(y_i^C - x_i^C) + x_i^C$ is equal to $x_i^{C(j)}$ by definition of $q^{(j)}$. So $s_{\iota_j(i)}^{[2]}(T_2) = 1$ holds exactly if $x_i^{C(j)} \leq x_i$.

3. Phase 3

The ‘‘Activator neuron’’ ω_j fires at T_3 if and only if $x^{C(j)} \leq x$: First notice

$$i_{\omega_j}^{[2]}(T_3) + b_{\omega_j}^{[2]} = \frac{1}{n} \sum_{i=1}^n s_{\iota_j(i)}^{[2]}(T_2) - 2s_{a_2}^{[2]}(T_2) + r_s(q^{(j)}; T_2) = \frac{1}{n} \sum_{i=1}^n s_{\iota_j(i)}^{[2]}(T_2).$$

from which we derive

$$p_{\omega_j}^{[2]}(T_3) = p_{\omega_j}^{[2]}(T_2) + i_{\omega_j}^{[2]}(T_3) + b_{\omega_j}^{[2]} - s_{\omega_j}^{[2]}(T_2) = \frac{1}{n} \sum_{i=1}^n s_{\iota_j(i)}^{[2]}(T_2).$$

So $0 \leq p_{\omega_j}^{[2]}(T_3) \leq 1$ and we get $p_{\omega_j}^{[2]}(T_3) = 1$, as well as $s_{\omega_j}^{[2]}(T_3) = 1$ exactly if $\forall_{i \in [n]} x_i^{C(j)} \leq x_i$, so if $x^{C(j)} \leq x$.

Let further $i \in [n]$. We get

$$i_{\iota_j(i)}^{[2]}(T_3) + b_{\iota_j(i)}^{[2]} = s_i^{[1]}(T_3) + T(s_{c_1}^{[2]}(T_2) - 2s_{c_2}^{[2]}(T_2) + r_s(q^{(j)}; T_2)) = -2T$$

and therefore

$$\begin{aligned} p_{\iota_j(i)}^{[2]}(T_3) &= p_{\iota_j(i)}^{[2]}(T_2) + i_{\iota_j(i)}^{[2]}(T_3) + b_{\iota_j(i)}^{[2]} - s_{\iota_j(i)}^{[2]}(T_2) \\ &= -q_i^{(j)}T_r + 1 + \lfloor T_2 o_i(x) \rfloor - 2T - s_{\iota_j(i)}^{[2]}(T_2). \end{aligned}$$

So $p_{\iota_j(i)}^{[2]}(T_3) \leq -T$ and $s_{\iota_j(i)}^{[2]}(T_3) = 0$, since $\lfloor T_2 o_i(x) \rfloor \leq T_2 \leq T - 1$.

4. Phase 4

Let $i \in [n]$. The neuron $\iota_j(i)$ stays inactive at T_4 , since

$$i_{\iota_j(i)}^{[2]}(T_4) + b_{\iota_j(i)}^{[2]} = s_i^{[1]}(T_4) + T(s_{c_1}^{[2]}(T_3) - 2s_{c_2}^{[2]}(T_3) + r_s(q^{(j)}; T_3)) = Tr_s(q^{(j)}; T_3)$$

and hence

$$\begin{aligned} p_{\iota_j(i)}^{[2]}(T_4) &= p_{\iota_j(i)}^{[2]}(T_3) + i_{\iota_j(i)}^{[2]}(T_4) + b_{\iota_j(i)}^{[2]} - s_{\iota_j(i)}^{[2]}(T_3) \\ &= p_{\iota_j(i)}^{[2]}(T_3) + Tr_s(q^{(j)}; T_3). \end{aligned}$$

Since $p_{\iota_j(i)}^{[2]}(T_3) \leq -T$, we conclude $p_{\iota_j(i)}^{[2]}(T_4) \leq 0$ and $s_{\iota_j(i)}^{[2]}(T_4) = 0$.

Further the “activator neuron” ω_j fires at T_4 exactly if $x \in C^{(j)}$: First notice

$$i_{\omega_j}^{[2]}(T_4) + b_{\omega_j}^{[2]} = \frac{1}{n} \sum_{i=1}^n s_{\iota_j(i)}^{[2]}(T_3) - 2s_{a_2}^{[2]}(T_3) + r_s(q^{(j)}; T_3) = r_s(q^{(j)}; T_3).$$

So we get

$$\begin{aligned} p_{\omega_j}^{[2]}(T_4) &= p_{\omega_j}^{[2]}(T_3) + i_{\omega_j}^{[2]}(T_4) + b_{\omega_j}^{[2]} - s_{\omega_j}^{[2]}(T_3) \\ &= p_{\omega_j}^{[2]}(T_3) + r_s(q^{(j)}; T_3) - s_{\omega_j}^{[2]}(T_3). \end{aligned}$$

By definition of q , $q_i^{(j)} < q_i^{(j')}$ holds exactly if $x_i^{C(j)} < x_i^{C(j')}$ holds for all $i \in [n]$ and $j, j' \in [K^n]$, so $\forall_{j, j' \in [K^n]} q^{(j)} < q^{(j')} \Leftrightarrow x^{C(j)} < x^{C(j')}$. We further have shown $\forall_{j' \in K^n} s_{\omega_{j'}}^{[2]}(T_3) = 1 \Leftrightarrow x^{C(j')} \leq x$ before.

$$\begin{aligned} r_s(q^{(j)}; T_3) &= s_{\omega_j}^{[2]}(T_3) - \sum_{\substack{q' \in ([K-1]_0)^n \\ q^{(j)} < q'}} s_{\omega_{j(q')}}^{[2]}(T_3) \\ &= s_{\omega_j}^{[2]}(T_3) - \sum_{\substack{j' \in [K^n] \\ x^{C(j)} < x^{C(j')} \leq x}} 1 \end{aligned}$$

Now if $x \in C^{(j)}$, i.e. $x^{C^{(j)}} \leq x < y^{C^{(j)}}$, then $s_{\omega_j}^{[2]}(T_3) = 1$ and if there was a $s_{\omega_j}^{[2]}(T_3) = 1$ with $x^{C^{(j)}} < x^{C^{(j')}} \leq x$, we would further get $x^{C^{(j)}} < x^{C^{(j(q'))}} < y^{C^{(j)}}$ due to $x < y^{C^{(j)}}$. But this contradicts the construction of the subcubes $(C^{(j)})_{j \in [K^n]}$.

We have also shown $p_{\omega_j}^{[2]}(T_3) = 1$ and $s_{\omega_j}^{[2]}(T_3) = 1$ for this case, so we can conclude $p_{\omega_j}^{[2]}(T_4) = 1$ as well as $s_{\omega_j}^{[2]}(T_4) = 1$.

Now suppose $x \notin C^{(j)}$. Since we assumed $x \in C$ and constructed the subcubes $(C^{(j)})_{j \in [K^n]}$ as a partition of C , we have $x \in C^{(j')}$, $j \neq j'$. Now if $x^{C^{(j)}} < x^{C^{(j')}}$, then $r_s(q^{(j)}; T_3) \leq 0$. We further have $p_{\omega_j}^{[2]}(T_3) = 1$ and $s_{\omega_j}^{[2]}(T_3) = 1$ in this case and therefore $p_{\omega_j}^{[2]}(T_4) \leq 0$ and $s_{\omega_j}^{[2]}(T_4) = 0$.

Is on the other hand $x^{C^{(j)}} \not< x^{C^{(j')}}$, then there is a component $i \in [n]$, such that $x_i^{C^{(j')}} < x_i^{C^{(j)}}$ and therefore $x_i < y_i^{C^{(j')}} \leq x_i^{C^{(j)}}$. This implies $x^{C^{(j)}} \not\leq x$ and we also get $r_s(q^{(j)}; T_3) \leq 0$. We further have $p_{\omega_j}^{[2]}(T_3) < 1$ and $s_{\omega_j}^{[2]}(T_3) = 0$ in this case and therefore $p_{\omega_j}^{[2]}(T_4) < 1$ and $s_{\omega_j}^{[2]}(T_4) = 0$.

To summarize, we $s_{\omega_j}^{[2]}(T_4) = 1$ exactly if $x \in C^{(j)}$ just as we claimed, as well as $p_{\omega_j}^{[2]}(T_4) \leq 1$ in general.

5. Phase 5

The “activator neuron” ω_j is inactive during T_5 , since for $t > T_4$

$$i_{\omega_j}^{[2]}(t) + b_{\omega_j}^{[2]} = \frac{1}{n} \sum_{i=1}^n s_{\iota_j(i)}^{[2]}(t-1) - 2s_{a_2}^{[2]}(t-1) + r_s(q^{(j)}; t-1) \leq 0.$$

and

$$\begin{aligned} p_{\omega_j}^{[2]}(t) &= u_{\omega_j}^{[2]}(T_4) + \sum_{k=T_4+1}^t (i_{\omega_j}^{[2]}(k) + b_{\omega_j}^{[2]}) - \sum_{k=T_4+1}^{t-1} s_{\omega_j}^{[2]}(k) \\ &\leq u_{\omega_j}^{[2]}(T_4). \end{aligned}$$

Further $u_{\omega_j}^{[2]}(T_4) < 1$, since $p_{\omega_j}^{[2]}(T_4) \leq 1 < 2$. So $\forall_{t > T_4} s_{\omega_j}^{[2]}(t) = 0$. In particular the “switch” $r_s(q^{(j)}; t) = 0$ is off for all $j \in [K^n]$ and $t > T_4$.

Let $i \in [n]$. During T_5 the neuron $\iota_j(i)$ captures the position of x in $C^{(j)}$ regarding the i -th dimension if $x \in C^{(j)}$ and stays inactive otherwise.

Let us first assume $x \notin C^{(j)}$. We have previously shown $s_{\omega_j}^{[2]}(T_4) = 0$ and just now $\forall_{t > T_4} s_{\omega_j}^{[2]}(t) = 0$. So $\forall_{t \geq T_4} r_s(q^{(j)}; t) = 0$ and therefore for all $t > T_4$

$$i_{\iota_j(i)}^{[2]}(t) + b_{\iota_j(i)}^{[2]} = s_i^{[1]}(t) + T(s_{c_1}^{[2]}(t-1) - 2s_{c_2}^{[2]}(t-1) + r_s(q^{(j)}; t-1)) \leq 0$$

as well as

$$\begin{aligned} p_{\iota_j(i)}^{[2]}(t) &= p_{\iota_j(i)}^{[2]}(T_4) + \sum_{k=T_4+1}^t (i_{\iota_j(i)}^{[2]}(k) + b_{\iota_j(i)}^{[2]} - s_{\iota_j(i)}^{[2]}(k-1)) \\ &= p_{\iota_j(i)}^{[2]}(T_4) + \sum_{k=T_4+1}^t (i_{\iota_j(i)}^{[2]}(k) + b_{\iota_j(i)}^{[2]} - s_{\iota_j(i)}^{[2]}(k-1)) \\ &\leq p_{\iota_j(i)}^{[2]}(T_4) \\ &\leq 0. \end{aligned}$$

So in particular $\forall_{t>T_4} s_{\iota_j(i)}^{[2]}(t) = 0$.

Suppose now $x \in C^{(j)}$.

Let j be given with $x \in C^{(j)}$. As we have seen before, we have $r_s(q^{(j)}; T_4) = 1$ and $\forall_{t>T_4} r_s(q^{(j)}; t) = 0$. We therefore get

$$i_{\iota_j(i)}^{[2]}(T_4 + 1) + b_{\iota_j(i)}^{[2]} = s_i^{[1]}(T_4 + 1) + T(s_{c_1}^{[2]}(T_4) - 2s_{c_2}^{[2]}(T_4) + r_s(q^{(j)}; T_4)) = T$$

and for all $t > T_4 + 1$

$$i_{\iota_j(i)}^{[2]}(t) + b_{\iota_j(i)}^{[2]} = s_i^{[1]}(t) + T(s_{c_1}^{[2]}(t-1) - 2s_{c_2}^{[2]}(t-1) + r_s(q^{(j)}; t-1)) = 0.$$

Using previous results and in particular $s_{\iota_j(i)}^{[2]}(T_2) = 1 \Leftrightarrow x_i^{C^{(j)}} \leq x_i$, we obtain

$$\begin{aligned} p_{\iota_j(i)}^{[2]}(T_4 + 1) &= p_{\iota_j(i)}^{[2]}(T_4) + i_{\iota_j(i)}^{[2]}(T_4 + 1) + b_{\iota_j(i)}^{[2]} - s_{\iota_j(i)}^{[2]}(T_4) \\ &= p_{\iota_j(i)}^{[2]}(T_3) + Tr_s(q^{(j)}; T_3) + T \\ &= -q_i^{(j)}T_r + 1 + \lfloor T_2 o_i(x) \rfloor - 2T - s_{\iota_j(i)}^{[2]}(T_2) + 2T \\ &= -KT_r o_i(x^{C^{(j)}}) + \lfloor KT_r o_i(x) \rfloor. \end{aligned}$$

We now have $KT_r o_i(x^{C^{(j)}}) \leq KT_r o_i(x)$, since $x_i^{C^{(j)}} \leq x_i$, so $p_{\iota_j(i)}^{[2]}(T_4 + 1) \geq 0$.

We further have

$$o_i(y^{C^{(j)}}) - o_i(x^{C^{(j)}}) = \frac{y_i^{C^{(j)}} - x_i^{C^{(j)}}}{y_i^C - x_i^C} = \frac{1}{K}.$$

and $o_i(x) < o_i(y^{C^{(j)}}) = o_i(x^{C^{(j)}}) + \frac{1}{K}$ for all $x \in C^{(j)}$. We can deduce

$$\begin{aligned} p_{\iota_j(i)}^{[2]}(T_4 + 1) &= -KT_r o_i(x^{C^{(j)}}) + \lfloor KT_r o_i(x) \rfloor \\ &\leq -KT_r o_i(x^{C^{(j)}}) + KT_r o_i(y^{C^{(j)}}) \\ &\leq -KT_r o_i(x^{C^{(j)}}) + T_r + KT_r o_i(x^{C^{(j)}}) \\ &\leq T_r. \end{aligned}$$

So we have shown

$$0 \leq p_{\iota_j(i)}^{[2]}(T_4 + 1) = u_{\iota_j(i)}^{[2]}(T_4) + i_{\iota_j(i)}^{[2]}(T_4 + 1) + b_{\iota_j(i)}^{[2]} \leq T_r = T - (T_4 + 1).$$

By [Lemma 2.8](#) we now get $u_j^{[2]}(T_4 + 1) \geq 0$. We can therefore use [Proposition 2.2](#) with $t_0 := T_4 + 1$, $t_m := T_4 + 1$ and $t_\omega := T$, such that we obtain

$$\begin{aligned} \sum_{t=T_4+1}^T s_{\iota_j(i)}^{[2]}(t) &= \lfloor u_{\iota_j(i)}^{[2]}(T_4) + i_{\iota_j(i)}^{[2]}(T_4 + 1) + b_{\iota_j(i)}^{[2]} \rfloor \\ &= -KT_r o_i(x^{C^{(j)}}) + \lfloor KT_r o_i(x) \rfloor. \end{aligned}$$

We have now reached the final step, where we will show that the spikes actually approximate g . Let us consolidate our results. For j with $x \in C^{(j)}$ we have

$$\begin{aligned} \sum_{t=T_4}^T s_{\iota_j(i)}^{[2]}(t) &= \sum_{t=T_4+1}^T s_{\iota_j(i)}^{[2]}(t) = -KT_r o_i(x^{C^{(j)}}) + \lfloor KT_r o_i(x) \rfloor, \\ \sum_{t=T_4}^T s_{\omega_j}^{[2]}(t) &= s_{\omega_j}^{[2]}(T_4) = 1. \end{aligned}$$

And therefore for all $k \in [m]$

$$\begin{aligned} W_{k,\omega_j}^{[L+1]} \sum_{t=T_4}^T s_{\omega_j}^{[2]}(t) &= g_k(x^{C(j)}), \\ W_{k,\iota_{j(i)}}^{[L+1]} \sum_{t=T_4}^T s_{\iota_{j(i)}}^{[2]}(t) &= \left(-KT_r o_i(x^{C(j)}) + \lfloor KT_r o_i(x) \rfloor\right) d(g^{(j)})_k (y_i^{C(j)} - x_i^{C(j)}) \frac{1}{T_r} e_i. \end{aligned}$$

Let now j be such that $x \notin C^{(j)}$, we have shown

$$\sum_{t=T_4}^T s_{\iota_{j(i)}}^{[2]}(t) = 0, \quad \sum_{t=T_4}^T s_{\omega_j}^{[2]}(t) = 0.$$

So this group of neurons does not contribute to the output of the network:

$$W_{k,\omega_j}^{[L+1]} \sum_{t=T_4}^T s_{\omega_j}^{[2]}(t) = 0, \quad W_{k,\iota_{j(i)}}^{[L+1]} \sum_{t=T_4}^T s_{\iota_{j(i)}}^{[2]}(t) = 0.$$

Finally note that by choice of $W^{[L+1]}$, the neurons c_1, c_2, a_2 don't contribute to the output as well. For any $j \in \{c_1, c_2, a_2\}$

$$W_{k,j}^{[L+1]} \sum_{t=T_4}^T s_j^{[2]}(t) = 0.$$

Let now $j \in [K^n]$ be such that $x \in C_j$ again. We can consequently have

$$R(\Phi)_k(x) = \sum_{t=T_4}^T (W^{[L+1]} s^{[2]}(t))_k = W_{k,\omega_j}^{[L+1]} \sum_{t=T_4}^T s_{\omega_j}^{[2]}(t) + W_{k,\iota_{j(i)}}^{[L+1]} \sum_{t=T_4}^T s_{\iota_{j(i)}}^{[2]}(t).$$

Which is further equal to

$$\begin{aligned} &g_k(x^{C(j)}) + \sum_{i \in [n]} \left(d(g^{(j)})_k (y_i^{C(j)} - x_i^{C(j)}) \frac{1}{T_r} e_i \right) \left(-KT_r o_i(x^{C(j)}) + \lfloor KT_r o_i(x) \rfloor \right) \\ &= g_k(x^{C(j)}) + \underbrace{\sum_{i \in [n]} \frac{y_i^{C(j)} - x_i^{C(j)}}{T_r} \left(-KT_r o_i(x^{C(j)}) + \lfloor KT_r o_i(x) \rfloor \right) e_i}_{=: x'} \end{aligned}$$

Now by assumption, we have

$$T_r = \sqrt{n} \frac{y_i^C - x_i^C}{K} \frac{\|df\|_{\infty,2}}{2\nu} \geq \sqrt{n} (y_i^{C(j)} - x_i^{C(j)}) \frac{\|d(g^{(j)})\|_{\infty,2}}{2\nu}$$

for any $i \in [n]$ and hence

$$\xi_i := \frac{1}{T_r} (\lfloor KT_r o_i(x) \rfloor - KT_r o_i(x)) \leq \frac{1}{\sqrt{n} (y_i^{C(j)} - x_i^{C(j)}) \|d(g^{(j)})\|_{\infty,2}} \frac{2\nu}{2\nu}.$$

We now get the following inequality by definition of ξ_i and $(y_i^{C(j)} - x_i^{C(j)})K = (y_i^C - x_i^C)$

$$\begin{aligned}
 \|x' - x\|_2^2 &= \sum_{i \in [n]} (x' - x_i)^2 \\
 &= \sum_{i \in [n]} \left((y_i^{C(j)} - x_i^{C(j)}) \left(\frac{1}{T_r} \lfloor KT_r o_i(x) \rfloor - K o_i(x^{C(j)}) \right) - (x_i - x_i^{C(j)}) \right)^2 \\
 &= \sum_{i \in [n]} \left((y_i^{C(j)} - x_i^{C(j)}) \left(\xi_i + K \frac{x_i - x_i^{C(j)}}{y_i^C - x_i^C} \right) - (x_i - x_i^{C(j)}) \right)^2 \\
 &= \sum_{i \in [n]} \left((\xi_i (y_i^{C(j)} - x_i^{C(j)}) + (x_i - x_i^{C(j)})) - (x_i - x_i^{C(j)}) \right)^2 \\
 &= \sum_{i \in [n]} \xi_i^2 (y_i^{C(j)} - x_i^{C(j)})^2 \\
 &\leq \frac{\nu^2}{\|d(g^{(j)})\|_{\infty,2}^2}.
 \end{aligned}$$

So we can conclude

$$\|g(x) - R(\Phi)_k(x)\|_2 = \|g(x) - g(x')\|_2 = \|d(g^{(j)})(x - x')\|_2 \leq \|d(g^{(j)})\|_{\infty,2} \|x - x'\|_2 \leq \nu$$

□

Sadly the size of the network in this construction is not always smaller than the one from [Theorem 3.1](#). A concrete counter example is a sinus wave with high frequency and small amplitude, like $f(x) := \frac{\sin(nx)}{n}$ with $n \in \mathbb{N}$ on $C = U = [0, 3\pi)$ and $\Omega = [0, 2\pi]$. Since $f'(x) = \cos(nx)$ and $\|f'\|_{(0,3\pi)} = 1$, $\Gamma := 1$ is the optimal Lipschitz-constant for f . At the same time, since $f''(x) = n \sin(nx)$ and $\|f''\|_{(0,3\pi)} = n$, the biggest possible modulus of uniform continuity on $(0, 3\pi)$ for f' is $\delta(\varepsilon) := \frac{\varepsilon}{n}$. So we get

$$\max_{\substack{\xi, \theta > 0 \\ \xi \theta = \varepsilon}} \min(\delta(\xi), \theta) = \max_{\xi > 0} \min\left(\frac{\xi}{n}, \frac{\varepsilon}{\xi}\right) = \sqrt{\frac{\varepsilon}{n}}.$$

So we get $K(\varepsilon) := \lfloor \pi \sqrt{\frac{n}{\varepsilon}} \rfloor$. We therefore get for the layer sizes:

<i>Theorem 3.1</i>	<i>Theorem 3.2</i>
$n_1 = \lceil \frac{2\pi}{\varepsilon} \rceil + 1$	$n_1 = 2$
$n_2 = \lceil \frac{2\pi}{\varepsilon} \rceil$	$n_2 = 2 \lfloor \pi \sqrt{\frac{n}{\varepsilon}} \rfloor + 3$

While the first and second layer of [Theorem 3.2](#) are clearly arbitrarily smaller than the first layer of the other construction for small ε , the second layer of [Theorem 3.2](#) is arbitrarily bad for $n \rightarrow \infty$ compared to the second layer of [Theorem 3.1](#).

On the other hand, even for large $n \in \mathbb{N}$, the second layer of [Theorem 3.2](#) is arbitrarily more efficient for $\varepsilon \rightarrow 0$, since the size of the second layer of [Theorem 3.2](#) only grows proportionally to $\frac{1}{\sqrt{\varepsilon}}$ and not $\frac{1}{\varepsilon}$.

We generalize this observation with the following theorem.

Theorem 3.3. *Let the function f and the sets U, C, Ω be given as in [Theorem 3.2](#). Let further Ω be compact, assume that f is Γ' -Lipschitz, and $d(f|_{U^o})$ is Γ -Lipschitz.*

We then have

$$\lim_{\varepsilon \rightarrow 0} \frac{n_2(\varepsilon)}{n_2'(\varepsilon)} = 0.$$

3 STRUCTURE OF COMPUTATIONS IN R. LIF-SNNS

Where $n'_2(\varepsilon)$ is $n_2(\varepsilon)$ as defined in [Theorem 3.1](#) for Ω , $f|_\Omega$ with Lipschitz-constant Γ' and ε ; and where $n_2(\varepsilon)$ is as defined in [Theorem 3.2](#) for sets U , C , Ω and function f with modulus of uniform continuity $\omega(x) := \Gamma x$ of $d(f|_U)$ and approximation control $\mu := \frac{\varepsilon}{2}$.

Proof. First notice, that since

$$n'_2(\varepsilon) = \max \left\{ \left\lceil \frac{\text{diam}_\infty(\Omega)}{\varepsilon} \Gamma' \right\rceil^n, 1 \right\}$$

we have

$$\lim_{\varepsilon \rightarrow 0} n'_2(\varepsilon) \varepsilon = (\text{diam}_\infty(\Omega) \Gamma')^n.$$

We further have (compare [Remark 3.2](#))

$$n_2(\varepsilon) = \left(1 \vee \left\lceil \frac{\text{diam}_\infty(C)}{\frac{2}{\sqrt{n}} \min(\sqrt{\frac{\varepsilon}{2\Gamma}}, \frac{\rho}{2})} \right\rceil \right) (n+1) + 3$$

such that we get

$$\lim_{\varepsilon \rightarrow 0} n_2(\varepsilon) \sqrt{\varepsilon} = \sqrt{\frac{n\Gamma}{2}} \text{diam}_\infty(C) (n+1).$$

We can conclude

$$\lim_{\varepsilon \rightarrow 0} \frac{n_2(\varepsilon)}{n'_2(\varepsilon)} \frac{1}{\sqrt{\varepsilon}} = \lim_{\varepsilon \rightarrow 0} \frac{n_2(\varepsilon) \varepsilon}{n'_2(\varepsilon) \sqrt{\varepsilon}} = \frac{\sqrt{\frac{n\Gamma}{2}} \text{diam}_\infty(C) (n+1)}{\text{diam}_\infty(\Omega) \Gamma'} \in \mathbb{R}$$

So we can deduce

$$\lim_{\varepsilon \rightarrow 0} \frac{n_2(\varepsilon)}{n'_2(\varepsilon)} = 0.$$

□

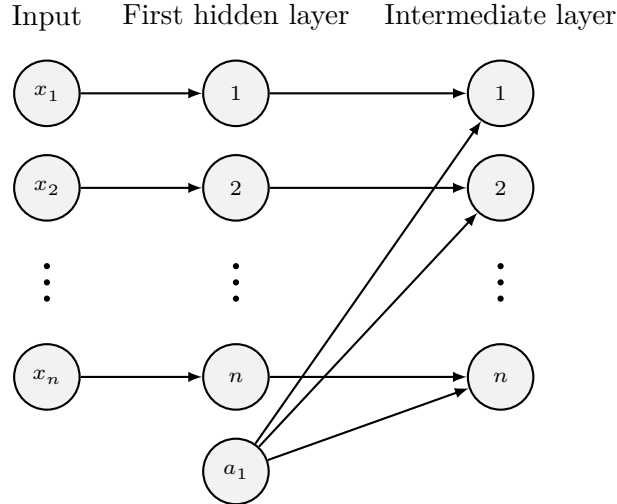


Figure 3.9: Replacing the first hidden layer by non-recurrent structure

To conclude this section, we shall give one last remark. While we used r. LIF-SNN for our construction in [Theorem 3.2](#), we do think that it is not too hard too remove the interdependencies inside of the layers too obtain a d.t. LIF-SNN by replicating layers as often as needed and connecting where necessary. E.g. we can remove the interdependency of a_1 on itself in the construction of [Theorem 3.2](#) by using $u_0 = -T_2$ and $b = 1$ to obtain the “alarm clock” behavior. Further its connections to the other neurons in the first layer can be removed by adding an

3 STRUCTURE OF COMPUTATIONS IN R. LIF-SNNS

intermediate layer between first and second hidden layer with n neurons that just forward the input from their respective neuron in the first layer (see also [Example 2.1](#)) and also take in an input from a_1 , compare [Fig. 3.9](#). The reason why we have still chosen to go ahead with r. LIF-SNN in our proof is that the network complexity is smaller.

4 Complexity of input partitions

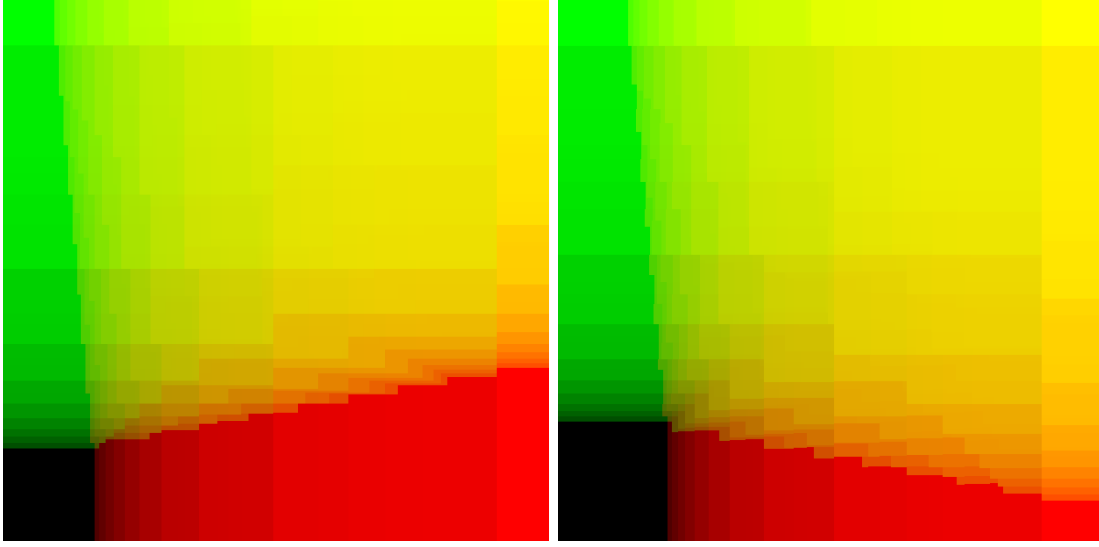


Figure 4.1: The landscape of the first layer of two different r. LIF-SNN

In the following we will analyze what shape the graph of the realized function $R(\Phi)$ of a r. LIF-SNN Φ has. Of particular interest to us is the question how many different values it can obtain. Since the number of different values of the following layers only depends on the spike trains of the first hidden layer, we can get an upper bound on that number by analyzing how many different spike trains the first hidden layer can produce. We will therefore only study the output landscape of the first hidden layer. We will further generally assume $W^{[1]} = I_{n_1}$. Let Φ be a r. LIF-SNN with $W^{[1]}$ arbitrary and Φ' the same one, but with $W^{[1]} = I_{n_1}$. We then have $R(\Phi)(x) = R(\Phi')(W^{[1]}x)$ since we are using direct encoding. So $W^{[1]}$ just corresponds to a pre-transformation on the input vector and can therefore (1) only decrease the number of different output values and further the key to understanding the output landscape of a general r. LIF-SNN is to understand the landscape of one with $W^{[1]} = I_{n_1}$.

We can further simplify the notation in this section by writing $W, b, V, \alpha, \beta, \vartheta, i, u, s, n$ for $W^{[1]}, b^{[1]}, V^{[1]}, \alpha^{[1]}, \beta^{[1]}, \vartheta^{[1]}, i^{[1]}, u^{[1]}, s^{[1]}, n_1$ respectively, since we will only be working on the first layer anyways. Further write $x := s^{[0]}$. Since we are using direct encoding, we may assume $\forall_{t \in [T]} s^{[0]}(t) = x$. We now get the following simplified defining equations:

$$i(t) = \alpha i(t-1) + Wx + Vs(t-1) \quad (15)$$

$$p(t) = \beta u(t-1) + i(t) + b \quad (16)$$

$$s(t) = H(p(t) - \vartheta \mathbf{1}_n) \quad (17)$$

$$u(t) = p(t) - \vartheta s(t) \quad (18)$$

Since the definitions recursively depend on x , we will sometimes also write $i(t; x)$, $p(t; x)$, $s(t; x)$ and $u(t; x)$ to make the dependency explicit. It will further be helpful to write e.g. $i(\cdot; x)$ for $(i(t; x))_{t \in [T]}$.

By using the simplified notation, we obtain the following definitions from [Definition 2.6](#) for

the input vector $x \in \mathbb{R}^{n_0}$ and the first hidden layer spike-train σ :

$$i(t; x; \sigma) = \alpha^t i(0) + \sum_{k=1}^t \alpha^{t-k} (Wx + V\sigma(k-1)), \quad (19)$$

$$p(t; x; \sigma) = \beta^t u(0) + \sum_{k=1}^t \beta^{t-k} (i(k; x; \sigma) + b) - \vartheta \sum_{k=1}^{t-1} \beta^{t-k} \sigma(k), \quad (20)$$

$$s(t; x; \sigma) = H(p(t; x; \sigma) - \vartheta \mathbf{1}_{n_1}) \quad (21)$$

$$u(t; x; \sigma) = \beta^t u(0) + \sum_{k=1}^t \beta^{t-k} (i(k; x; \sigma) + b - \vartheta \sigma(k)), \quad (22)$$

Let us first introduce some preliminary definitions.

Definition 4.1. The set of constant regions of a r. LIF-SNN Φ is defined as the partition

$$C_\Phi := \{R(\Phi)^{-1}(\{y\}) \mid y \in \text{im}(R(\Phi))\}$$

of \mathbb{R}^{n_0} . A constant region with spike train $s' \in \{0, 1\}^{n_1 \times T}$, of the first layer of a r. LIF-SNN Φ is defined as

$$C_{s'} := \{x \in \mathbb{R}^{n_0} \mid \forall_{t \in [T]} s(t; x) = s'(t)\}$$

We further notate the set of such non-empty regions by $C_{\Phi, 1} := \{C_{s'} \mid s' \in \{0, 1\}^{n_1 \times T}, C_{s'} \neq \emptyset\}$.

While quite technical, g is the key to proper rigorous proofs about the landscape of r. LIF-SNNs.

Lemma 4.1. *There is a function $g: \bigcup_{t \in [T]} \{\sigma \mid \sigma \in \{0, 1\}^{n_1 \times (t-1)}\} \rightarrow \mathbb{R}$ such that $\forall_{i \in [n_1]} s_i(t; x; \sigma) = 1 \Leftrightarrow \langle w_i, x \rangle \geq g_i(t; \sigma)$ defined by*

$$g(t; \sigma) := - \frac{\sum_{k=1}^t \beta^{t-k} (\alpha^k i(0) + b + \sum_{l=1}^k \alpha^{k-l} V\sigma(l-1)) + \beta^t u(0) - \vartheta (1 + \sum_{k=1}^{t-1} \beta^{t-k} \sigma(k))}{\sum_{k=1}^t \beta^{t-k} \sum_{l=1}^k \alpha^{k-l}}$$

Where w_i denotes the i -th row vector of W .

Proof of Lemma 4.1. We compute for $i \in [n_1]$ using Lemma 2.1:

$$\begin{aligned} & H(p_i(t; x; \sigma) - \vartheta) \\ &= H \left(\underbrace{\sum_{k=1}^t \beta^{t-k} (i_i(k; x; \sigma) + b_i) + \beta^t u_i(0) - \vartheta \left(1 + \sum_{k=1}^{t-1} \beta^{t-k} \sigma_i(k) \right)}_{(*)} \right) \\ &= H \left(\sum_{k=1}^t \beta^{t-k} \left(\alpha^k i_i(0) + \sum_{l=1}^k \alpha^{k-l} (\langle w_i, x \rangle + (V\sigma(l-1))_i) + b_i \right) + (*) \right) \\ &= H \left(\sum_{k=1}^t \beta^{t-k} \sum_{l=1}^k \alpha^{k-l} \langle w_i, x \rangle + \sum_{k=1}^t \beta^{t-k} \left(\alpha^k i_i(0) + b_i + \sum_{l=1}^k \alpha^{k-l} (V\sigma(l-1))_i \right) + (*) \right) \\ &= H \left(\langle w_i, x \rangle + \frac{\sum_{k=1}^t \beta^{t-k} (\alpha^k i_i(0) + b_i + \sum_{l=1}^k \alpha^{k-l} (V\sigma(l-1))_i) + (*)}{\sum_{k=1}^t \beta^{t-k} \sum_{l=1}^k \alpha^{k-l}} \right) \\ &= H(\langle w_i, x \rangle - g_i(t; \sigma)) \end{aligned}$$

□

Remark 4.1. Like with $i(t; x; \sigma)$, $p(t; x; \sigma)$, etc. we will allow supplying g with a extension of the spiketrain that is required. In that case the value of g should be understood as the value at the prefix of that spiketrain.

Proposition 4.1. *The constant regions of the first layer of a r. LIF-SNN Φ (with $W = I_{n_1}$) are half-open cuboids. In particular, let $s' \in \{0, 1\}^{n_1 \times T}$ be a spike train and $C_{s'} = \llbracket x^{s'}, y^{s'} \rrbracket$ be the corresponding constant region. Then*

$$x_i^{s'} = \sup_{\substack{t \in [T] \\ s'_i(t)=1}} g_i(t; s') \quad y_i^{s'} = \inf_{\substack{t \in [T] \\ s'_i(t)=0}} g_i(t; s')$$

using the conventions $\inf \emptyset = \infty$ and $\sup \emptyset = -\infty$ here.

Remark 4.2. Note that by [Proposition 4.1](#), we have $x_i^{s'} = -\infty$ exactly if $\forall_{t \in [T]} s'_i(t) = 0$; and $y_i^{s'} = \infty$ exactly if $\forall_{t \in [T]} s'_i(t) = 1$. So in particular we cannot have s' with an component $i \in [n_1]$, such that both $x_i^{s'} = -\infty$ and $y_i^{s'} = \infty$.

Lemma 4.2. *For every $s' \in \{0, 1\}^{n_1 \times T}$ we have*

$$C_{s'} = \left(\bigcap_{\substack{i \in [n_1], t \in [T] \\ s'_i(t)=0}} \pi_i^{-1}([-\infty, g_i(t; s')]) \right) \cap \left(\bigcap_{\substack{i \in [n_1], t \in [T] \\ s'_i(t)=1}} \pi_i^{-1}([g_i(t; s'), \infty]) \right).$$

Proof of Lemma 4.2. Let $s' \in \{0, 1\}^{n_1 \times T}$ be given. We then get

$$\begin{aligned} C_{s'} &= \bigcap_{i \in [n_1], t \in [T]} \{x \mid s'_i(t) = s_i(t; x)\} \\ &= \bigcap_{i \in [n_1], t \in [T]} \{x \mid s'_i(t) = s_i(t; x; s')\} \\ &= \left(\bigcap_{\substack{i \in [n_1], t \in [T] \\ s'_i(t)=0}} \{x \mid s_i(t; x; s') = 0\} \right) \cap \left(\bigcap_{\substack{i \in [n_1], t \in [T] \\ s'_i(t)=1}} \{x \mid s_i(t; x; s') = 1\} \right) \\ &= \left(\bigcap_{\substack{i \in [n_1], t \in [T] \\ s'_i(t)=0}} \pi_i^{-1}([-\infty, g_i(t; s')]) \right) \cap \left(\bigcap_{\substack{i \in [n_1], t \in [T] \\ s'_i(t)=1}} \pi_i^{-1}([g_i(t; s'), \infty]) \right) \end{aligned}$$

by [Lemma 2.1](#) for the second equality and [Lemma 4.1](#) for the last one. \square

Lemma 4.3. $\bigcap_{j \in J} \prod_{i \in [n]} M_{i,j} = \prod_{i \in [n]} \bigcap_{j \in J} M_{i,j}$ for an index set J and sets $(M_{i,j})_{i \in [n], j \in J}$.

Proof of Lemma 4.3. For every $x = (x_i)_{i \in [n]} \in M := \prod_{i \in [n]} \bigcup_{j \in J} M_{i,j}$ holds:

$$x \in \bigcap_{j \in J} \prod_{i \in [n]} M_{i,j} \Leftrightarrow \left(\forall_{j \in J} x \in \prod_{i \in [n]} M_{i,j} \right) \Leftrightarrow \forall_{j \in J} \forall_{i \in [n]} x_i \in M_{i,j}.$$

On the other hand, we have

$$x \in \prod_{i \in [n]} \bigcap_{j \in J} M_{i,j} \Leftrightarrow \left(\forall_{i \in [n]} x_i \in \bigcap_{j \in J} M_{i,j} \right) \Leftrightarrow \forall_{i \in [n]} \forall_{j \in J} x_i \in M_{i,j}.$$

In total we get

$$\bigcap_{j \in J} \prod_{i \in [n]} M_{i,j} = M \cap \bigcap_{j \in J} \prod_{i \in [n]} M_{i,j} = M \cap \prod_{i \in [n]} \bigcap_{j \in J} M_{i,j} = \prod_{i \in [n]} \bigcap_{j \in J} M_{i,j}.$$

\square

Lemma 4.4. *The intersection $C_1 \cap C_2$ of half-open cuboids $C_1, C_2 \subset \mathbb{R}^n$ is a half-open cuboid. In particular, if $C_i := \prod_{j \in [n]} [c_{i,j}, d_{i,j})$, then*

$$C_1 \cap C_2 = \prod_{j \in [n]} ([\sup(c_{1,j}, c_{2,j}), \inf(d_{1,j}, d_{2,j}))$$

We use \inf/\sup instead of \min/\max to highlight that the components of the vertices might be $\pm\infty$.

Proof of Lemma 4.4. Let us first regard $n = 1$. For $c_1, c_2, d_1, d_2 \in \mathbb{R} \cup \{\pm\infty\}$ we get

$$\begin{aligned} x &\in [c_1, d_1) \cap [c_2, d_2) \\ &\Leftrightarrow c_1, c_2 \leq x < d_1, d_2 \\ &\Leftrightarrow x \in [\sup(c_{1,j}, c_{2,j}), \inf(d_{1,j}, d_{2,j})) \end{aligned}$$

for $x \in \mathbb{R}$ and therefore

$$C_1 \cap C_2 = \prod_{j \in [n]} ([c_{1,j}, d_{1,j}) \cap [c_{2,j}, d_{2,j})) = \prod_{j \in [n]} [\sup(c_{1,j}, c_{2,j}), \inf(d_{1,j}, d_{2,j}))$$

by Lemma 4.3. □

Proof of Proposition 4.1. Let $C_{s'} \in C_{\Phi,1}$ be a constant region. We then get

$$\begin{aligned} C_{s'} &= \left(\bigcap_{\substack{i \in [n_1], t \in [T] \\ s'_i(t)=0}} \pi_i^{-1}([-\infty, g_i(t; s')]) \right) \cap \left(\bigcap_{\substack{i \in [n_1], t \in [T] \\ s'_i(t)=1}} \pi_i^{-1}([g_i(t; s'), \infty)) \right) \\ &= \prod_{i \in [n]} [\sup_{t \in [T], s'_i(t)=1} g_i(t; s'), \inf_{t \in [T], s'_i(t)=0} g_i(t; s')] \end{aligned}$$

by Lemma 4.2 and Lemma 4.4, since

$$\pi_i^{-1}([c, d]) = [-\infty, \infty) \times \dots \times [c, d) \times \dots \times [-\infty, \infty).$$

□

We will now add an ordering to spike trains based on lexicographical ordering.

Definition 4.2. Let $s', s'' \in \{0, 1\}^{n_1 \times T}$, we then have $s' \leq_l s''$ exactly if either $s' = s''$ or there exists a $t \in [T]$ such that $s'(t) < s''(t)$ and $\forall_{t' < t} s'(t') = s''(t')$. We further define $s' <_l s''$ as $s' \leq_l s''$ and $s' \neq s''$.

Lemma 4.5. *Let $x, y \in \mathbb{R}^{n_0}$. We then have $x \leq y \Rightarrow s(\cdot; x) \leq_l s(\cdot; y)$.*

Is on the other hand $s(\cdot; x) <_l s(\cdot; y)$ and $i \in [n_1]$ such that $s_i(t; x) \neq s_i(t; y)$ holds at the minimal time t at which $s(\cdot; x)$ and $s(\cdot; y)$ are different, then $x_i < y_i$ holds.

Proof. First notice that for a fixed σ the function $i_i(t; x; \sigma)$ is growing monotonically in x_i for all $i \in [n]$, since $\alpha \geq 0$. We similarly get that $p_i(t; x; \sigma)$ and $s_i(t; x; \sigma)$ are growing monotonically in x_i , since $H = \chi_{[0, \infty)}$ is monotonically growing.

Let now $x \leq y$ and suppose $s(\cdot; x) \neq s(\cdot; y)$. We then have a minimal $t \in [T]$ such $s(t; x) \neq s(t; y)$. Now due to $\forall_{t' < t} s(t'; x) = s(t'; y)$ and $x \leq y$ we have $\forall_{i \in [n_1]} s_i(t; x) \leq s_i(t; y)$ using the previous remark with $\sigma(t) := s(t; x)$. We therefore get $s(t; x) <_l s(t; y)$.

Let on the other hand $s(\cdot; x) <_l s(\cdot; y)$. Then there is a smallest time t such that $\exists_{i \in [n_1]} s_i(t; x) \neq s_i(t; y)$. By definition of the ordering on spike trains we get $s_i(t; x) <_l s_i(t; y)$. Now $s_i(t; x)$ is growing monotonically in x_i with $\sigma(t) := s(t; x)$ by choice of t . We therefore have $x_i < y_i$. □

4 COMPLEXITY OF INPUT PARTITIONS

We will further proof some theorems about specific kinds of spike trains. We first have to define them.

Definition 4.3. We call a spike train $s' \in \{0,1\}^{n_1 \times T}$ constant in component $i \in [n_1]$, if $\forall_{t,t' \in [T]} s'_i(t) = s'_i(t')$. A spike train s' is therefore constant in every component, if $\forall_{i \in [n_1]} \forall_{t,t' \in [T]} s'_i(t) = s'_i(t')$. We further call s' non-constant in every component, if $\forall_{i \in [n_1]} \exists_{t,t' \in [T]} s'_i(t) \neq s'_i(t')$.

We also need some geometric definitions.

Definition 4.4. We call a point $p \in (\mathbb{R} \cup \{\pm\infty\})^n$ vertex of a non-empty region $\llbracket x, y \rrbracket$ or (x, y) , if $\forall_{i \in [n]} p_i \in \{x_i, y_i\}$. We call a vertex p finite, if $p \in \mathbb{R}^n$.

We will continue by characterizing regions with constant spike trains in none/some/all components.

Lemma 4.6. Let $s' \in \{0,1\}^{n_1 \times T}$ be a spike train that is constant in every component. Then $C_{s'}$ is non-empty.

Proof. Suppose s' is constant in every component. Let further $i \in [n_1]$. We now either have $\forall_{t \in [T]} s'_i(t) = 0$ or $\forall_{t \in [T]} s'_i(t) = 1$. In the first case, we get $x_i^{s'} = -\infty$ and $y_i^{s'} \in \mathbb{R}$; in the second case $x_i^{s'} \in \mathbb{R}$ and $y_i^{s'} = \infty$ by construction. So $C_{s'}$ is clearly non-empty. \square

Lemma 4.7. Let $s' \in \{0,1\}^{n_1 \times T}$ be a spike train such that $C_{s'} \neq \emptyset$ and s' is constant in k components. Then $C_{s'}$ has 2^{n_1-k} finite vertices.

Remark 4.3. So in particular, every region $C_{s'} \in C_{\Phi,1}$ has at least one finite vertex.

Proof. First note that if s' is constant in component $i \in [n_1]$, then either $x_i^{s'} = -\infty$ and $y_i^{s'} \in \mathbb{R}$; or $x_i^{s'} \in \mathbb{R}$ and $y_i^{s'} = \infty$. In either case $|\mathbb{R} \cap \{x_i^{s'}, y_i^{s'}\}| = 1$. Is on the other hand s' non-constant in i , then $x_i^{s'}, y_i^{s'} \in \mathbb{R}$. We therefore get

$$\left| \mathbb{R}^{n_1} \cap \prod_{j \in [n_1]} \{x_j^{s'}, y_j^{s'}\} \right| = \left| \prod_{j \in [n_1]} \mathbb{R} \cap \{x_j^{s'}, y_j^{s'}\} \right| = 2^{n_1-k}$$

using [Lemma 4.3](#) for the set of finite vertices, $\mathbb{R}^{n_1} \cap \prod_{j \in [n_1]} \{x_j^{s'}, y_j^{s'}\}$. \square

Lemma 4.8. Let $s' \in \{0,1\}^{n_1 \times T}$ be a spike train such that $C_{s'} \neq \emptyset$. Then the following conditions,

- (a) s' is constant in at least one component,
- (b) $C_{s'}$ has $\leq 2^{n_1-1}$ finite vertices,
- (c) $C_{s'}$ has a non-finite vertex,
- (d) $C_{s'}$ is unbounded,

are equivalent.

Proof. Let first $x^{s'}, y^{s'}$ be defined as in [Proposition 4.1](#), such that we have $C_{s'} = \llbracket x^{s'}, y^{s'} \rrbracket$.

- (a) \Leftrightarrow (b): Follows directly by [Lemma 4.7](#).
- (b) \Rightarrow (c): A non-empty cube $\llbracket x^{s'}, y^{s'} \rrbracket$ has

$$\left| \prod_{j \in [n_1]} \{x_j^{s'}, y_j^{s'}\} \right| = 2^{n_1}$$

vertices, so assuming $C_{s'}$ has $\leq 2^{n_1-1}$ finite vertices, it must have a non-finite one.

- (c) \Rightarrow (b): If $C_{s'}$ has a non-finite vertex, then the number of finite vertices is smaller than 2^{n_1} . On the other hand the number of finite vertices must be a power of 2 by [Lemma 4.7](#), so the number must be smaller or equal 2^{n_1-1} .
- (c) \Leftrightarrow (d): If all vertices are finite, then in particular $x^{s'}, y^{s'} \in \mathbb{R}^{n_1}$ and $\llbracket x^{s'}, y^{s'} \rrbracket$ is clearly bounded. Is one vertex non-finite, then either $x^{s'}$ or $y^{s'}$ has a component that is $\pm\infty$. Suppose we have $x_i^{s'} = -\infty$. Since $C_{s'} \neq \emptyset$ we further have $z \in C_{s'}$. By definition of $\llbracket x^{s'}, y^{s'} \rrbracket$, we have now $z' \in \llbracket x^{s'}, y^{s'} \rrbracket$ with $\forall_{i \neq j} z'_j = z_j$ and $z'_i \in [-\infty, z_i]$ arbitrary. So $C_{s'}$ is not bounded. The proof for $y_i^{s'} = \infty$ is similar.

□

In the following, final proposition of this chapter we show that we can find a tight boundary around all finite vertices.

Proposition 4.2. *Let P be the set of all finite vertices of regions $C_{s'} \in C_{\Phi,1}$.*

We then have $P \subset \llbracket x^C, y^C \rrbracket$ for

$$x_i^C := \min_{\substack{t \in [T], \sigma \in \{0,1\}^{n_1 \times t} \\ \forall_{t \in [T]} \sigma_i(t) = 0}} g(t; \sigma),$$

$$y_i^C := \max_{\substack{t \in [T], \sigma \in \{0,1\}^{n_1 \times t} \\ \forall_{t \in [T]} \sigma_i(t) = 1}} g(t; \sigma).$$

In fact every component in x_i^C is maximal and every component in y_i^C minimal with $P \subset \llbracket x^C, y^C \rrbracket$. We further have in particular $x_i^C = \min_{t \in [T]} g_i(t; s^{x_i^C})$ and $y_i^C = \max_{t \in [T]} g_i(t; s^{y_i^C})$, where

$$s_j^{x_i^C}(t) := \begin{cases} 1 & (i \neq j) \wedge (v_{ij} > 0) \\ 0 & (i = j) \vee (v_{ij} \leq 0) \end{cases}$$

$$s_j^{y_i^C}(t) := \begin{cases} 1 & (i = j) \vee (v_{ij} < 0) \\ 0 & (i \neq j) \wedge (v_{ij} \geq 0) \end{cases}$$

Remark 4.4. From [Proposition 4.2](#) we obtain in particular that $C_{s'} \subset \llbracket x^C, y^C \rrbracket$ for all regions $C_{s'} \in C_{\Phi,1}$ with non-constant spiketrain s' : By [Lemma 4.8](#) all vertices of $C_{s'} = \llbracket x^{s'}, y^{s'} \rrbracket$ are finite, so $x^{s'}, y^{s'} \in P \subset \llbracket x^C, y^C \rrbracket$ and therefore $\llbracket x^{s'}, y^{s'} \rrbracket \subset \llbracket x^C, y^C \rrbracket$.

Remark 4.5. [Proposition 4.2](#) is in particular useful, when computing $|C_{\Phi,1}|$: We know that every region $C_{s'} \in C_{\Phi,1}$ has at least one finite vertex by [Lemma 4.7](#) and by [Proposition 4.2](#) we know where to search for those vertices.

Proof. We will first proof $P \subset \llbracket x^C, y^C \rrbracket$: Let $v \in P$ be a finite vertex of region $C_{s'} = \llbracket x^{s'}, y^{s'} \rrbracket$. Now $\forall_{i \in [n_1]} v_i \in \{x_i^{s'}, y_i^{s'}\} \cap \mathbb{R}$, so in particular in the case of $v_i = x_i^{s'}$ for a $i \in [n_1]$ there is time-step $t_1 \in [T]$ such that $s'(t_1) = 1$; instead $v_i = y_i^{s'}$ then there is time-step $t_0 \in [T]$ such that $s'(t_0) = 0$. Let now $i \in [n_1]$ and $v_i = x_i^{s'}$ be given. Let further t_1 be minimal with $s'(t_1) = 1$. Let us now define $s^1 \in \{0,1\}^{n_1 \times t_1}$ by $s^0(t) := \mathbf{0}_{n_1}$. Now by choice of t_1 , we have $\forall_{t < t_1} s_i^1(t) = 0$. Further, since $g(t; \sigma)$ only consumes the first $t-1$ time-steps of the given spike train σ (see [Remark 4.1](#)), we have $g_i(t_1; s^0) = g_i(t_1; s')$. We therefore get

$$x_i^C = \min_{\substack{t \in [T], \sigma \in \{0,1\}^{n_1 \times t} \\ \forall_{t \in [T]} \sigma_i(t) = 0}} g_i(t; \sigma) \leq \sup_{\substack{t \in [T] \\ s'_i(t) = 1}} g_i(t; s')$$

since $g_i(t_1; s^1)$ appears on the left hand and $g_i(t_1; s')$ on the right hand. We can similarly construct s^0 and obtain

$$y_i^C = \max_{\substack{t \in [T], \sigma \in \{0,1\}^{n_1 \times t} \\ \forall_{t \in [T]} \sigma_i(t) = 1}} g_i(t; \sigma) \geq \inf_{\substack{t \in [T] \\ s'_i(t) = 0}} g_i(t; s').$$

4 COMPLEXITY OF INPUT PARTITIONS

So we get $v \in \llbracket x^C, y^C \rrbracket$.

Further x^C, y^C are in fact chosen optimal: Let $i \in [n_1]$. Now the spike-train $s' := s^{x_i^C}$ is constant in every component, so $C_{s'} = \llbracket x^{s'}, y^{s'} \rrbracket$ is non-empty by [Lemma 4.6](#) and has a finite vertex $v \in \mathbb{R}^{n_1}$ (exactly one) by [Lemma 4.7](#). This vertex must have $v_i = y_i^{s'}$, since $\forall_{t \in [T]} s'_i(t; x) = 0$ by definition and therefore $x_i^{s'} = -\infty$. Further

$$v_i = y_i^{s'} = \inf_{\substack{t \in [T] \\ s'_i(t) = 0}} g_i(t; s') = \min_{t \in [T]} g_i(t; s') = x_i^C$$

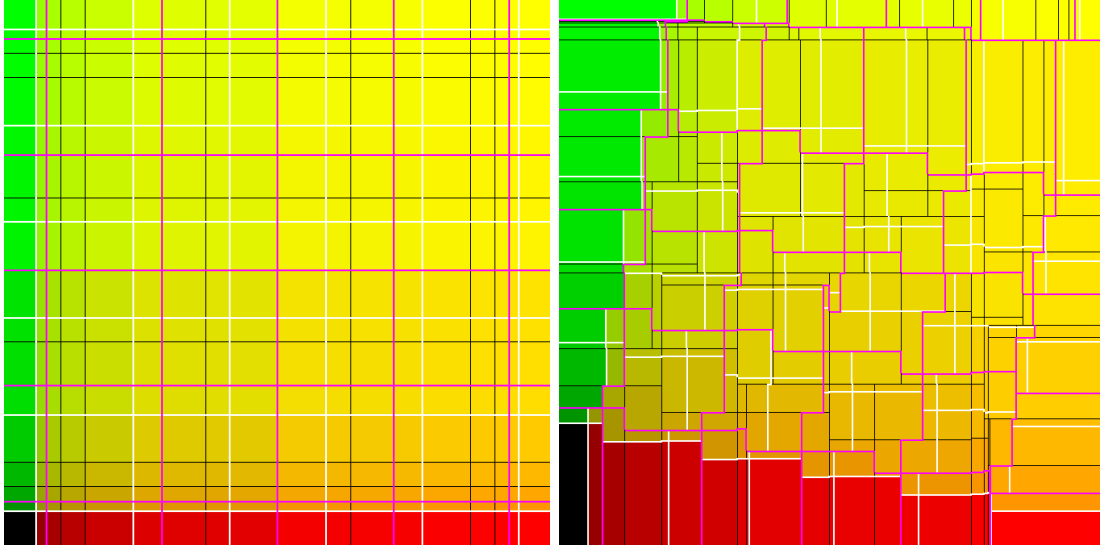
by definition. We similarly get a vertex $v \in P$ with $v_i = x_i^{s^{y_i^C}} = y_i^C$.

Let us now show that it suffices to compute g on $s_j^{x_i^C}$ and $s_j^{y_i^C}$ to obtain the boundaries. For that purpose we have to take another look at the definition of g . Since the definition is quite clunky we have split it up into the following functions

$$g_i^r(t; \sigma) := - \frac{\sum_{k=1}^t \beta^{t-k} (\alpha^k i_i(0) + b + \sum_{l=1}^k \alpha^{k-l} \langle v_i, \sigma(l-1) \rangle) + \beta^t u_i(0)}{\sum_{k=1}^t \beta^{t-k} \sum_{l=1}^k \alpha^{k-l}},$$

$$g_i^\vartheta(t; \sigma) := \vartheta \frac{1 + \sum_{k=1}^{t-1} \beta^{t-k} \sigma_i(k)}{\sum_{k=1}^t \beta^{t-k} \sum_{l=1}^k \alpha^{k-l}}$$

such that $g(t; \sigma) = g^r(t; \sigma) + g^\vartheta(t; \sigma)$. Now fixing t and requiring $\forall_{t \in [T]} \sigma_i(t) = 0$, the function $g_i^\vartheta(t; \sigma)$ is constant; further $g_i^r(t; \sigma)$ is minimal for $\forall_{i \neq j} \sigma_j(t) = 1_{((v_i)_j > 0)}$, so in particular for $\sigma = s^{x_i^C}$. Hence we indeed have $x_i^C = \min_{t \in [T]} g_i(t; s^{x_i^C})$. We similarly get $x_i^C = \min_{t \in [T]} g_i(t; s^{y_i^C})$. \square



(a) The landscape of a d.t. LIF-SNN.

(b) The landscape of a r. LIF-SNN.

Figure 4.2: Comparison of d.t. LIF-SNN vs. r. LIF-SNN; White lines separate regions with a different spike count; Pink lines separate regions with a different spike count on the previous time step

Theorem 4.3 of [\[Nguyen et al., 2025\]](#) states the following:

Theorem 4.1. *Consider a discrete-time LIF-SNN Φ with T time steps, input dimension n_0 and n_1 neurons in the first hidden layer. Then the number of constant regions is bounded by*

$$|C_\Phi| \leq |C_{\Phi,1}| \leq \begin{cases} \sum_{i=0}^{n_0} \left(\frac{T^2+T}{2} \right)^i \binom{n_1}{i} & n_1 > n_0, \\ \left(\frac{T^2+T+2}{2} \right)^{n_1} & \text{otherwise.} \end{cases} \quad (23)$$

While we were able to get promising empirical results in [Section 5](#) that seem to confirm that [Theorem 4.1](#) also holds for r. LIF-SNN, we were not able to prove them. We will therefore first give a sketch of a proof for the case of $W = I_{n_1}$ and $V = \mathbf{0}_{n_1 \times n_1}$ and then showcase why the generalization to arbitrary V fails. Let us first state the stripped down version:

Theorem 4.2. *Consider a r. LIF-SNN Φ with T time steps, input dimension n_0 , trivial weights, i.e. $W = I_{n_1}$ and no recurrent connections, i.e. $V = \mathbf{0}_{n_1 \times n_1}$. Then the number of constant regions is bounded by*

$$|C_\Phi| \leq |C_{\Phi,1}| \leq \left(\frac{T^2 + T + 2}{2} \right)^{n_1} \quad (24)$$

Sketch. As we have stated before, $|C_\Phi| \leq |C_{\Phi,1}|$ follows directly from the fact that the output of the whole network only depends on the binary spike trains of the first layer.

Note that the separating lines in [Fig. 4.2a](#) are real lines instead of line segments like in [Fig. 4.2b](#). This shows the independence of the two neurons in the first hidden layer of the used d.t. LIF-SNN. In fact this property is inherent to d.t. LIF-SNN, since we get the following equations for $W = I_{n_1}$ and $V = \mathbf{0}_{n_1 \times n_1}$

$$i(t) = \alpha i(t-1) + x, \quad (25)$$

$$p(t) = \beta u(t-1) + i(t) + b, \quad (26)$$

$$s(t) = H(p(t) - \vartheta \mathbf{1}_n), \quad (27)$$

$$u(t) = p(t) - \vartheta s(t) \quad (28)$$

and in particular

$$i(t; x) = \alpha^t i(0) + x \sum_{k=1}^t \alpha^{t-k}, \quad (29)$$

$$p(t; x) = \beta^t u(0) + \sum_{k=1}^t \beta^{t-k} (i(k; x) + b) - \vartheta \sum_{k=1}^{t-1} \beta^{t-k} s(k). \quad (30)$$

Notice that the components i_i, p_i, s_i, u_i are computed in parallel, without affecting each other, in particular due to $V = 0$. Further the result of i_i, p_i, s_i, u_i only depends on x_i since further $W = I_{n_1}$.

So it suffices to analyze the possible change of the output of s_i depending on different values for x_i . Let us first define $\Phi_{t,i}$ as the restriction of Φ on the i -th neuron in the first hidden layer, using $t \in [T]$ for $T_{\Phi_{t,i}}$. Due to the previous analysis, the s_i of Φ has the same functionality as s_1 of $\Phi_{t,i}$.

We will now proof that $(s(t'))_{t' \in [t]}$ can take on $\frac{t^2+t+2}{2}$ many different values for $t \in [T]$ by induction on t . This suffices to proof the theorem since we then have n_1 independent neurons in the first layer that each can take on a maximal number of $\frac{T^2+T+2}{2}$ many values. Let $t = 1$. This case is obvious, since $s_i(1) \in \{0, 1\}$. Suppose on the other hand now $t > 1$. From now on, all notations like s, α, β, \dots will refer to $\Phi_{t,i}$, so in particular $s(t) \in \{0, 1\}$. By induction hypothesis, we have $\frac{t^2-t+2}{2}$ possible values for $(s(t'))_{t' \in [t-1]}$. Let us further categorize $(s(t'))_{t' \in [t-1]}$ by the number of spikes it contains. The number must be at least 0 and can be at most $t-1$, so we have t categories.

Let us further regard the definition of g for $\alpha = 0$ and $V = 0$,

$$g(t; \sigma) := - \frac{\sum_{k=1}^t \beta^{t-k} (\alpha^k i(0) + b) + \beta^t u(0) - \vartheta \left(1 + \sum_{k=1}^{t-1} \beta^{t-k} \sigma(k) \right)}{\sum_{k=1}^t \beta^{t-k} \sum_{l=1}^k \alpha^{k-l}}.$$

Clearly $g(t; \sigma)$ only changes in the subterm $\sum_{k=1}^{t-1} \beta^{t-k} \sigma(k)$ in σ . Suppose $x, x' \in \mathbb{R}$ are given with $x < x'$ and $s(\cdot; x) \neq s(\cdot; x')$. Then there appears a spike earlier in $s(\cdot; x)$ than in $s(\cdot; x')$ due

to Lemma 4.5. In fact, if both $s(\cdot; x)$ and $s(\cdot; x')$ are located in the same category, every spike in $s(\cdot; x)$ appears earlier than the corresponding spike in $s(\cdot; x')$, though we won't proof this here. So $\sum_{k=1}^{t-1} \beta^{t-k} s(k; x) \geq \sum_{k=1}^{t-1} \beta^{t-k} s(k; x')$ and therefore $g(t; s(\cdot; x)) \geq g(t; s(\cdot; x'))$. Notice that the ordering has been reversed.

Further, due to Proposition 4.1, a region $C_{s'}$ with $s' \in \{0, 1\}^{t-1}$ is only split at time-step t if $g(t; s') \in C^{s'}$. But since the regions are half-open cuboids, so half-open intervals in this case, given a category of regions $\kappa \in [t-1]_0$, we have that only one region of those with spike count κ can be split, since the ordering of $g(t; s(\cdot; x))$ is reversed to $x \in \mathbb{R}$ inside of category κ . So we get at a maximum, as many new regions as categories. So we obtain $t + \frac{t^2-t+2}{2} = \frac{t^2+t+2}{2}$ regions in total. \square

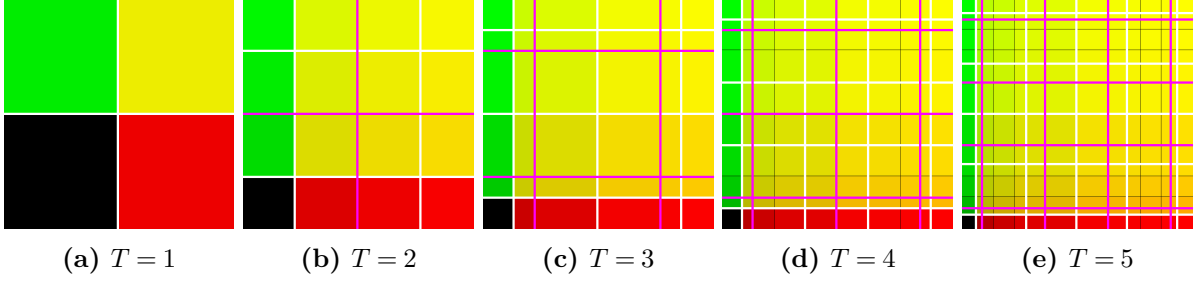


Figure 4.3: Development of landscape of a d.t. LIF-SNN through time; White lines separate regions with a different spike count; Pink lines separate regions with a different spike count on the previous time step

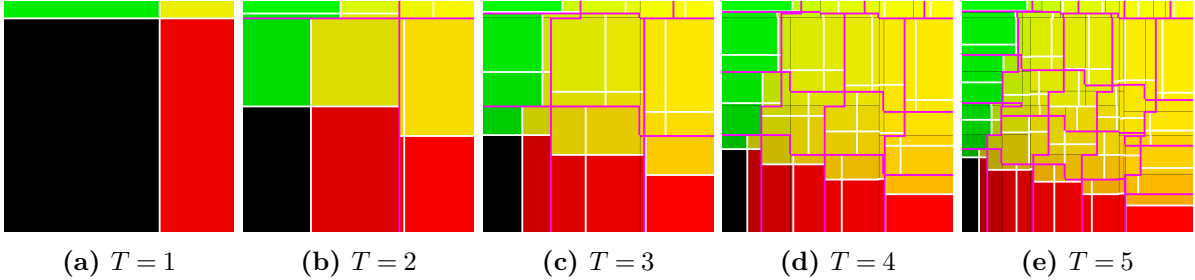


Figure 4.4: Development of landscape of a r. LIF-SNN through time; White lines separate regions with a different spike count; Pink lines separate regions with a different spike count on the previous time step

While we have not properly proven it, one interesting fact is that the introduction of α did not affect the theorem, only V creates problems. Let us now analyze we that is the case. One of the major properties the proof relies on is the neurons being independent of each other. While it is easily derived from the equations that this is not the case for $V \neq \mathbf{0}_{n_1 \times n_1}$, it might be more instructive to consider looking at Fig. 4.2b. To get out of this problem and “control” the chaos that the neurons might inflict on each other, one might consider it helpful to do the same categorization trick as in the previous proof, but applied on all neurons on the same time, i.e. we consider regions of equal number of spikes in each respective component one by one. Compare Fig. 4.4 and Fig. 4.4. The pink lines are the white lines of the respective previous time-step and gray lines were at some point white/pink, since every separating line was at some point a new line and every new line separates regions with different spike count (the regions around a new line differ in exactly one spike).

We further see that the plane is partitioned by the pink lines into similar shaped regions, that each contain a (potentially degenerated) white cross. Those white lines of the crosses correspond to $g_i(T; s(\cdot; x))$. The white lines are straight, even on different regions $C_{s'}$ inside of one category,

since we use $\beta = 1$ in both figures: In that case g_i is constant on spike trains with the same spike count. See e.g. Fig. 4.5 for $\beta \neq 1$. Now for $\beta = 1$ it is not hard to prove that inside of a given category, there exists only this single cross without multiple parallel lines, which is quite a promising result. One would be forgiven for thinking that we can just finish the proof similarly to before, after all we now know how many new lines are getting added in time-step t . But the devil being in the details, a considerable problem arises when considering how many regions will be added at a maximum. This can be deduced by finding the maximal number of regions from the previous time-step located horizontally/vertically next to each other in a category. In other words: How many regions can a straight line (that is parallel to a coordinate axis) touch inside of a category.

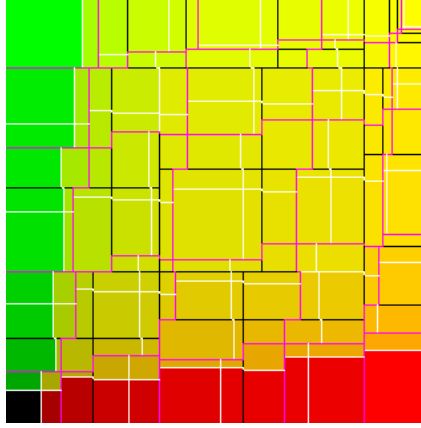


Figure 4.5: For $\beta \neq 1$ and $V \neq \mathbf{0}_{n_1 \times n_1}$, the white crosses are not made of straight lines

This question is very much non-trivial, especially since this question cannot be reduced to looking at the maximal number of regions that can be touched by a straight line in the whole plane like for $V = \mathbf{0}_{n_1 \times n_1}$, since the categories as well as the crosses might be shifted in relation to each other, as can be seen in e.g. Fig. 4.4.

5 Experimental results

To better understand the landscape of the input of a r. LIF-SNN with the goal of proving [Theorem 4.1](#) in mind, we have created the following programs.

5.1 Computing the number of regions

We use $W = I_n$ yet again in the following section, since it simplifies the following algorithm considerably and we hope to quite easily proof the situation for arbitrary W , once we have done it for $W = I_n$.

Algorithm 1 Compute regions similar to [Proposition 4.1](#)

```

1: function SUBREGIONS( $\Phi, t, st, x^C, y^C$ )
2:    $c \leftarrow g_\Phi(t + 1; st)$ 
3:    $subRegions \leftarrow \{\}$ 
4:   for  $sp \in \{0, 1\}^{n_1}$  do
5:      $x' \leftarrow (\text{if } sp_i = 0 \text{ then } x_i^C \text{ else } \max(x_i^C, c_i))_{i \in [(n_\Phi)_1]}$ 
6:      $y' \leftarrow (\text{if } sp_i = 1 \text{ then } y_i^C \text{ else } \min(y_i^C, c_i))_{i \in [(n_\Phi)_1]}$ 
7:     if  $x' < y'$  then
8:        $subRegions \leftarrow subRegions \cup \{(x', y', st \mathrel{++} [sp])\}$ 
9:     end if
10:  end for
11:  return  $subRegions$ 
12: end function
13: function REGIONSWITHST( $\Phi, t, st, x^C, y^C$ )
14:  if  $t \geq T_\Phi$  then
15:    return  $\{(st, x^C, y^C)\}$ 
16:  end if
17:   $regions \leftarrow \{\}$ 
18:  for  $(x', y', st) \in \text{SUBREGIONS}(\Phi, t, st, x^C, y^C)$  do
19:     $newRegions \leftarrow \text{REGIONSWITHST}(\Phi, t + 1, st, x', y')$ 
20:     $regions \leftarrow regions \cup newRegions$ 
21:  end for
22:  return  $regions$ 
23: end function
24: function COMPREGIONS( $\Phi$ )
25:  return  $\text{REGIONSWITHST}(\Phi, 0, [(0, \dots, 0)], (-\infty, \dots, -\infty), (\infty, \dots, \infty))$ 
26: end function

```

[Algorithm 1](#) is motivated by [Proposition 4.1](#), but instead of iterating over all spike trains $s' \in \{0, 1\}^{n_1 \times T}$ and checking whether $C_{s'} \neq \emptyset$ by evaluating $x_i^{s'} < y_i^{s'}$, we use a more efficient algorithm.

A few words on notation and representation in [Algorithm 1](#): We represent spike trains as lists, including the initial and also trivial spike $s^{[l]}(0) = \mathbf{0}_{n_l}$. So e.g. $s' \in \{0, 1\}^{n_1 \times T}$ with $\forall_{t \in [T]} s'_1(t) = 1$ and $\forall_{t \in [T]} s'_i(t) = 1$ for all $i \neq 1$ is represented by $st = [(0, \dots, 0), (1, 0, \dots, 0), \dots, (1, 0, \dots, 0)]$. Further a region $C_{s'} = \llbracket x^{s'}, y^{s'} \rrbracket$ is represented as a tuple $(st, x^{s'}, y^{s'})$, where st is the list corresponding to s' . We also use g_Φ for g as defined in [Lemma 4.1](#), used parameters from Φ . Similarly T_Φ is T from Φ .

Lemma 5.1. *Let Φ be a r. LIF-SNN. With $W^{[1]} = I_{n_1}$. The algorithm $\text{CompRegions}(\Phi)$ from [Algorithm 1](#) computes $C_{\Phi, 1}$.*

Proof. Let us write $st[t]$ for the t -th element of st , where st is indexed starting with 0.

In the following we will use Φ_τ to mean the r. LIF-SNN that only differ from Φ by having $T_{\Phi_\tau} = \tau$.

Let us now show that given $C_{s'} = \llbracket x^{s'}, y^{s'} \rrbracket \in C_{\Phi_{t-1},1}$, the algorithm $\text{SubRegions}(\Phi, t, s', x^{C_{s'}}, y^{C_{s'}})$ computes all regions $C_{s''} = \llbracket x^{s''}, y^{s''} \rrbracket \in C_{\Phi_t,1}$, where s'' is an extension of s' with non-empty region $C_{s''}$, i.e. $\forall_{t' \in [t-1]} s'(t') = s''(t')$ and $x^{C_{s''}} < y^{C_{s''}}$.

Notice first, that by [Proposition 4.1](#), we have $x_i^{s''} = \sup(x_i^{s'}, g_i(t; s''))$ and $y_i^{s''} = y_i^{s'}$ if $s''_i(t) = 1$ as well as $x_i^{s''} = x_i^{s'}$ and $y_i^{s''} = \inf(y_i^{s'}, g_i(t; s''))$ otherwise.

So we have

$$\begin{aligned} C_{s''} &= \prod_{i \in [n_1]} [x_i^{s''}, y_i^{s''}] \\ &= \prod_{i \in [n_1]} \begin{cases} [x_i^{s'}, \inf(y_i^{s'}, g_i(t; s''))] & s''(t) = 0 \\ [\sup(x_i^{s'}, g_i(t; s''))_i, y_i^{s'}] & s''(t) = 1 \end{cases} \end{aligned}$$

Comparing with the code of **SubRegions**, it is clear that given a region $C_{s'} \in C_{\Phi_{t-1},1}$, the algorithm computes the set of all extensions s'' of s' such that $C_{s''}$ is non-empty.

We similar get that **SubRegions** $(\Phi, 0, [(0, \dots, 0)], (-\infty, \dots, -\infty), (\infty, \dots, \infty))$ computes all regions $C_{s''} \in C_{\Phi_1,1}$ with spikes $s'' \in \{0, 1\}^{n_1 \times 1}$. First note that again by [Proposition 4.1](#), $x_i^{s''} = g_i(t; s'')$ and $y_i^{s''} = \infty$ if $s''_i(t) = 1$ as well as $x_i^{s''} = -\infty$ and $y_i^{s''} = g_i(t; s'')$ otherwise. So we have

$$C_{s''} = \prod_{i \in [n_1]} \begin{cases} [-\infty, g_i(t; s'')] & s''(t) = 0 \\ [g_i(t; s'')_i, \infty] & s''(t) = 1 \end{cases}$$

Which is clearly what **SubRegions** $(\Phi, 0, [(0, \dots, 0)], (-\infty, \dots, -\infty), (\infty, \dots, \infty))$ will compute.

Since **RegionsWithST** just repeatedly maps **SubRegions** over an initial value of $\{(\Phi, 0, [(0, \dots, 0)], (-\infty, \dots, -\infty), (\infty, \dots, \infty))\}$, and since **SubRegions** computes the regions of $C_{\Phi_\tau,1}$ from $C_{\Phi_{\tau-1},1}$; as well as $C_{\Phi_1,1}$ from the initial value, **RegionsWithST** computes $C_{\Phi,1}$. \square

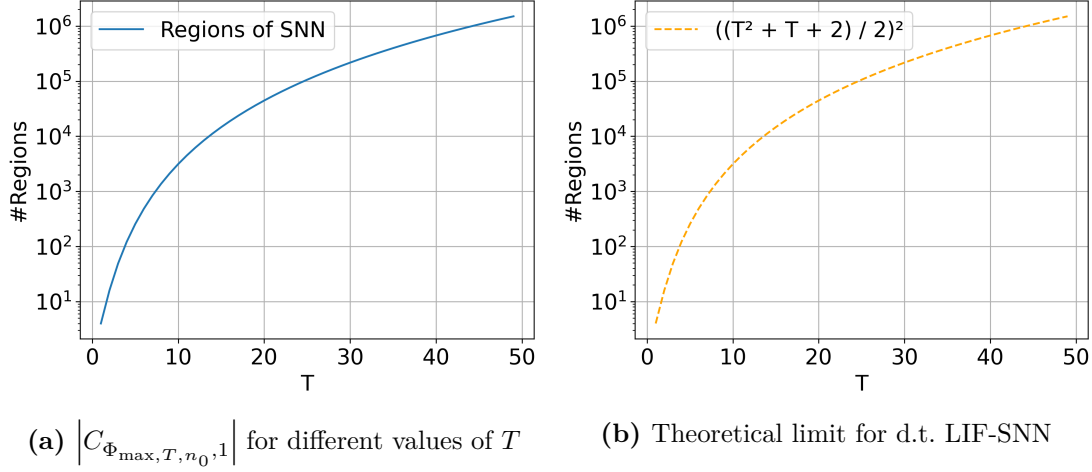
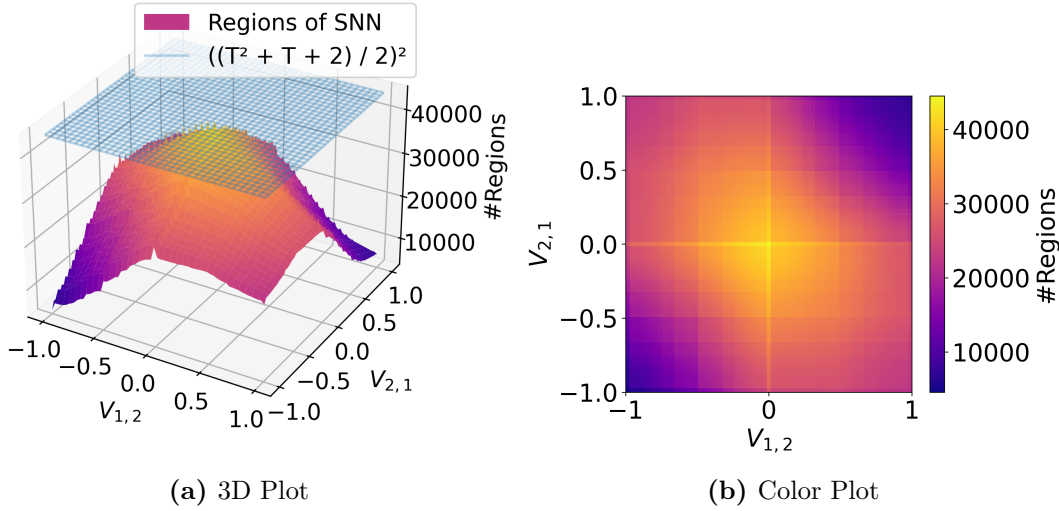
You can find two implementations of [Algorithm 1](#) in [Section 7.1](#). The first is a simple, straight-forward implementation in Python; the other is a more efficient version in C++. Due to the usage of floating point numbers, the results are only accurate up to a certain degree. In particular the C++ and Python implementations sometimes differ slightly, even for a low iteration number. The problem lies in the numerical instability of the algorithm. The landscape quite often includes empty regions $C_{s'} = \llbracket x^{s'}, y^{s'} \rrbracket$, i.e. $\exists_i x_i^{s'} = y_i^{s'}$, such that an implementation of g might just coincidentally return a slightly lower value for $x_i^{s'}$ than for $y_i^{s'}$.

Just skipping very slim regions does not seem correct either, since proper regions might just as well be very slim. Which brings us to the second issue of the implementations: Undercounting of regions. In Theorem B.15 of [\[Nguyen et al., 2025\]](#) it was proven, that the bound in [Theorem 4.1](#) is tight, i.e. for every $T \in \mathbb{N}$ and $n_0 = n_1 \in \mathbb{N}$ there exists a d.t. LIF-SNN Φ_{\max, T, n_0} such that the bound $(\frac{T^2 + T + 2}{2})^{n_1}$ is reached. Suppose we compute the number of regions for this d.t. LIF-SNN. In our algorithm we used 64-bit doubles for each vector component, so we can represent at most 2^{64} values. So at the very latest at $T \approx 2^{32} \sqrt{2}$ the algorithm will start to undercount the regions, though we have reason to think that the problem might appear far earlier.

Now Φ_{\max, T, n_0} is essentially just a d.t. LIF-SNN with canonical parameters in the first layer, so $\beta = \vartheta = 1$, $W = I_{n_1}$ and $b = i(0) = \mathbf{0}_{n_1}$, but with a small positive number for every component $u_i(0) > 0$ of $u(0)$.

Let us consider such a d.t. LIF-SNN and take a look at the maximum number of regions in [Fig. 5.1](#), computed by [Algorithm 1](#).

Clearly the plot in [Fig. 5.1a](#) of the number of regions of Φ_{\max, T, n_0} corresponds to the theoretical limit in [Fig. 5.1b](#).

Figure 5.1: Optimal Φ vs. theoretical limitFigure 5.2: $|C_{\Phi,1}|$ for $T = 20$ and different values of $V_{1,2}$ and $V_{2,1}$

Let us now consider a r. LIF-SNN $\Phi_{\alpha,V}$ that only differs in α or V from Φ_{\max,T,n_0} . Changing $V_{1,2}$ or $V_{2,1}$ only decreases $|C_{\Phi_{0,V},1}|$ as can be seen in Fig. 5.2. If we instead just change α or even β , we also only find a decrease in the number of regions, as one can see in Fig. 5.3.

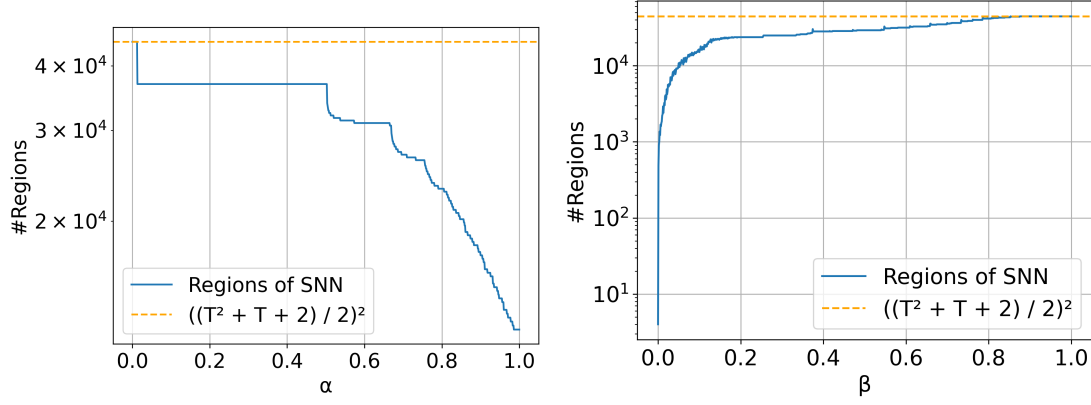
We have furthermore found, that in our experience even changes in multiple parameters do not increase the number of regions past the threshold.

Still, our hypothesis might be wrong, since our algorithms are insufficient and unable to compute the number of regions for high enough iterations efficiently; or the usage of floating point numbers might lead to too much undercounting of regions.

The only change that consistently pushes the number of regions above the upper bound is increasing β past its limit of 1, see Fig. 5.4 in contrast to Fig. 5.3b. This is also consistent with the proof of Theorem 4.1 in [Nguyen et al., 2025], since $\beta \in [0,1]$ is a critical condition of the proof. Of course this bothers us not too much, since exponential growth instead of decay would not make a good model of the neurological realities.

5.2 Visualization of landscape

While we can use the previous algorithm to compute all regions for a 2-neuron network and draw them on the screen, the algorithm is quite slow for higher iterations due to its exponential



(a) Changing α from 0 decreases the number of regions (b) Changing β from 1 decreases the number of regions

Figure 5.3: No parameter seems to be able to push the number of regions beyond the upper bound for $T = 20$

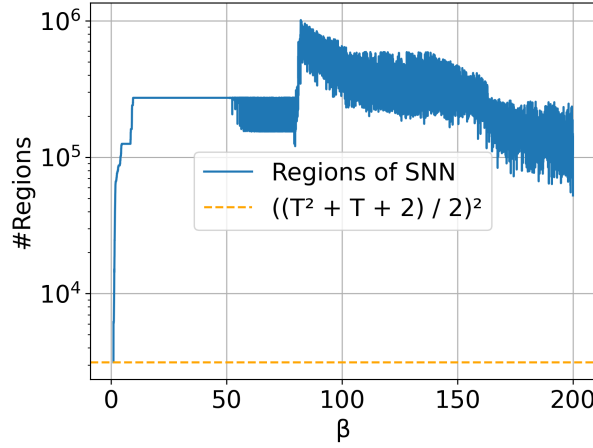


Figure 5.4: Changing β past the limit pushes the number of regions above the threshold for $T = 20$

nature. A previous iteration of our algorithms does not have that limitation:

We just compute $s^{[1]}$ on a grid in the input space and then compute the set of unique spike trains. Executed on the CPU this algorithm would be quite slow, however it is natural to implement it on the gpu instead. You can find an implementation in [Section 7.2](#).

But this has following disadvantages: Since there are no dynamically-sized lists in GLSL, the GPU programming language that we implemented the algorithm in, we used integers as fixed-sized lists of bits to represent the spike trains. Furthermore since there are no arbitrary precision integers on the gpu, we have chosen to use two 32-bit integers to represent spike trains with a length up to 64 time-steps.

Moreover, since we just evaluate $s^{[1]}$ on a grid, we might miss very slim regions located between nodes. Finally GLSL is also not flexible enough to allow arbitrary sized matrices/vectors. We have therefore fixed ourselves to the simple and easily visualizable case of 2 neurons in the first layer.

Another limitation is the storage requirement of the algorithm: If we want to make sure not to miss regions with width of smaller than $\text{diam}_\Omega(C) \cdot 0.00025 \approx \text{diam}_\Omega(C) \cdot 2^{-12}$ in a direction, the grid needs to have at least a width of 2^{12} , so in total it needs 2^{24} nodes. For each of those we compute 8 bytes (64 bits), so we need $2^{27}\text{B} = 128\text{MiB}$ of storage. Further, since we are using

2 neurons in layer one, the storage requirement grows quadratically in the width of the grid, so we quite quickly reach the limits of consumer hardware in storage.

On the other hand, this approach has the big advantage of allowing us to easily create visualizations of the landscape of a r. LIF-SNN by represent the pixels of an image with the grid. It is furthermore quite simple to port the program into the web browser using WebGL. See snn.valentin-herrmann.de for an instance of the program and [Section 7.3](#) for the code. This allowed us to create a simple user interface for changing the parameters of the r. LIF-SNN, the coloring algorithm, etc. without too much fuss, which allows users to obtain a much better intuition for the problem at hand through its interactive visualization.

We also implemented region counting for this type of algorithm, but sadly due to the limited architecture of WebGL, we were not able to implement it on the web. The concrete problem is that we are not able to read out data from gpu buffers. This will change with the new WebGPU API, but sadly that API is not yet completely supported in most browsers at the time of writing.

We have instead implemented the region counting in python. The obvious way to implement it, to just read out the buffer of spike-trains from the gpu and use a standard-algorithm to determine the unique elements has proved to be a major bottleneck. As before, we probably want to be able to quickly compute our result for a width of 2^{12} for the grid, so for 2^{24} nodes in total. It is further well-known, that algorithms filtering out the unique elements of a list are $\Theta(n \log(n))$, compare e.g. [Ben-Or, 1983].

We have therefore implemented a different algorithm for finding the unique elements in this particular case, that is just $\Theta(n)$. We will assume $W = I_n$, though it should be possible to generalize it. Since we know due to [Proposition 4.1](#) that all regions in $C_{\Phi,1}$ are half-open rectangles, it suffices to count all lower-left corners. So we just count all pixels such that the lower and left neighboring pixels are different (or don't exist since the pixel is located at the left/lower border of the grid).

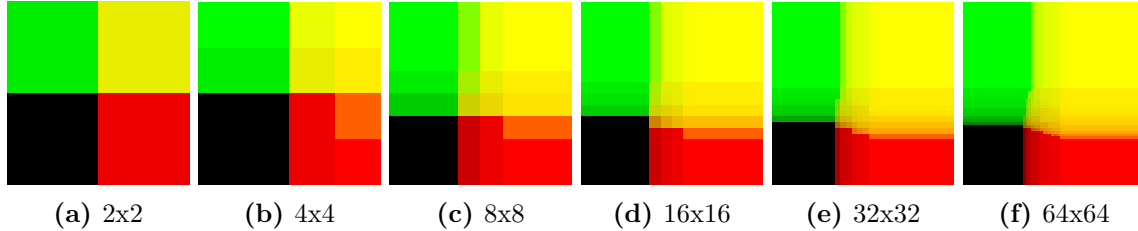


Figure 5.5: Doubling the grid repeatedly to improve the accuracy

We further utilize the regions having rectangular shapes by first executing the algorithm on a narrow grid and then doubling the size of the grid repeatedly. During the next iteration we can then just use the previous result if it is the same on the surrounding four nodes from the previous iteration. A nice side-effect is that we are additionally getting results for smaller grid sizes, such that we can get some feel for how much our algorithm is undercounting the regions due to too narrow grids.

6 Conclusion

To summarize, we have examined the properties of R. LIF-SNN, in particular in comparison to d.t. LIF-SNN: In [Section 2](#) we motivated our definitions, introduced lots of helpful notation and basic, technical lemmas. In [Section 3](#) we showed that r. LIF-SNN allow more compact and efficient approximation of some class of well-behaved functions than d.t. LIF-SNN.

Later in [Section 4](#) we showed that the graph of R. LIF-SNN consists of a finite number of constant regions, which are half-open cuboids for trivial $W^{[1]}$. We also showed that r. LIF-SNN with trivial $W^{[1]}$ can only really fit data inside of a square. While we were not able to prove it in [Section 4](#), we are hopeful to prove in the future that the number of different values a R. LIF-SNN with trivial $W^{[1]}$, given T and n_1 can obtain is bound by $(\frac{T^2+T+2}{2})^{n_1}$.

For that purpose we continued our analysis in [Section 5](#), where we explained a few programs which we wrote to generate empirical data as well as visualizations to obtain a better intuitive understanding of the problem at hand. While the data supports the hypothesis of the upper bound being correct, the data might be too specific, since we were not able to simulate many neurons or many time-steps.

To conclude: We were only able to scratch the surface of the theory of r. LIF-SNN. Much more research is needed to answer the question if r. LIF-SNN are indeed better suited for machine learning and in particular reasoning than current approaches. The next step would be to analyze more of the practicality of this model, since we focused mostly on theoretical aspects; for example how well r. LIF-SNN are able to fit basic data and how the output landscape changes during training.

Bibliography

References

- [Ben-Or, 1983] Ben-Or, M. (1983). Lower bounds for algebraic computation trees. In *Proceedings of the Fifteenth Annual ACM Symposium on Theory of Computing*, STOC '83, page 80–86, New York, NY, USA. Association for Computing Machinery.
- [Collins et al., 2023] Collins, K. M., Jiang, A. Q., Frieder, S., Wong, L., Zilka, M., Bhatt, U., Lukasiewicz, T., Wu, Y., Tenenbaum, J. B., Hart, W., Gowers, T., Li, W., Weller, A., and Jamnik, M. (2023). Evaluating language models for mathematics through interactions.
- [Gerstner et al., 2014] Gerstner, W., Kistler, W. M., Naud, R., and Paninski, L. (2014). *Neuronal Dynamics: From Single Neurons to Networks and Models of Cognition*. Cambridge University Press.
- [Herrmann, 2025a] Herrmann, V. (2025a). dt-lif-snn-compute-regions - compute constant regions of r. lif-snn. GitHub repository.
- [Herrmann, 2025b] Herrmann, V. (2025b). dt-lif-snn-compute-regions-depthsearch - compute constant regions of r. lif-snn. GitHub repository.
- [Herrmann, 2025c] Herrmann, V. (2025c). dt-lif-snn-visualizer - visualize the constant regions of r. lif-snn. GitHub repository.
- [Huang et al., 2025] Huang, L., Yu, W., Ma, W., Zhong, W., Feng, Z., Wang, H., Chen, Q., Peng, W., Feng, X., Qin, B., and Liu, T. (2025). A survey on hallucination in large language models: Principles, taxonomy, challenges, and open questions. *ACM Trans. Inf. Syst.*, 43(2).
- [Jegham et al., 2025] Jegham, N., Abdelatti, M., Elmoubarki, L., and Hendawi, A. (2025). How hungry is ai? benchmarking energy, water, and carbon footprint of llm inference.
- [Jones and Bergen, 2025] Jones, C. R. and Bergen, B. K. (2025). Large language models pass the turing test.
- [Kováč, 2009] Kováč, L. (2009). The 20 w sleep-walkers.
- [Love, 2024] Love, F. (2024). Nntikz - tikz diagrams for deep learning and neural networks. GitHub repository.
- [Mondorf and Plank, 2024] Mondorf, P. and Plank, B. (2024). Comparing inferential strategies of humans and large language models in deductive reasoning.
- [Nguyen et al., 2025] Nguyen, D. A., Araya, E., Fono, A., and Kutyniok, G. (2025). Time to spike? understanding the representational power of spiking neural networks in discrete time.
- [Ou, 2019] Ou, C. (2019). lmu-thesis-latex - lmu thesis latex template. GitHub repository.

7 Appendix

Find the source code for our programs of [Section 5](#) below.

7.1 dt-lif-snn-compute-regions-depthsearch

See [\[Herrmann, 2025b\]](#) for the corresponding git repository.

```

1 import argparse
2 from datatypes import SNN
3 import numpy as np
4
5
6 def parse_array(arg):
7     """Convert a comma-separated string into a numpy array or 2D array."""
8     if ";" in arg: # 2D array
9         rows = arg.split(";")
10        return np.array([list(map(float, row.split(","))) for row in rows])
11    else: # 1D array
12        return np.array(list(map(float, arg.split(","))))
13
14
15 def parseCMDLine():
16     # Default values
17     default_V = np.array([[0.0, 0.0], [0.0, 0.0]])
18     default_T = 10
19     default_i0 = np.array([0.0, 0.0])
20     default_u0 = np.array([0.0, 0.0])
21     default_b = np.array([0.0, 0.0])
22     default_theta = 1.0
23     default_alpha = 0.0
24     default_beta = 1.0
25     default_show = False
26     default_use_cpp = False
27
28     # Argument parser
29     parser = argparse.ArgumentParser(
30         description="Read neural network parameters from command line."
31     )
32     parser.add_argument(
33         "-V",
34         type=parse_array,
35         default=default_V,
36         help="matrix, rows separated by ';', elements by ',',",
37     )
38     parser.add_argument("-T", type=int, default=default_T, help="Simulation steps")
39     parser.add_argument(
40         "-i0",
41         type=parse_array,
42         default=default_i0,
43         help="Initial input, elements separated by ',',",
44     )
45     parser.add_argument(
46         "-u0",
47         type=parse_array,
48         default=default_u0,
49         help="Initial membrane potential, elements separated by ',',",
50     )

```

```

51     parser.add_argument(
52         "-b",
53         type=parse_array,
54         default=default_b,
55         help="Bias vector, elements separated by ','",
56     )
57     parser.add_argument(
58         "-theta", type=float, default=default_theta, help="Threshold value"
59     )
60     parser.add_argument(
61         "-alpha", type=float, default=default_alpha, help="Input decay rate"
62     )
63     parser.add_argument(
64         "-beta", type=float, default=default_beta, help="Membrane potential
65         decay rate"
66     )
67     parser.add_argument(
68         "--show",
69         action="store_true",
70         default=default_show,
71         help="Show visualization of regions",
72     )
73     parser.add_argument(
74         "--use-cpp",
75         action="store_true",
76         default=default_use_cpp,
77         help="Use quick C++ implementation",
78     )
79     args = parser.parse_args()
80
81     neurons_n = len(args.u0)
82
83     assert (
84         args.i0.shape[0] == neurons_n
85     ), f"i0 length {args.i0.shape[0]} does not match neurons_n {neurons_n}"
86     assert (
87         args.b.shape[0] == neurons_n
88     ), f"b length {args.b.shape[0]} does not match neurons_n {neurons_n}"
89     assert args.V.shape == (
90         neurons_n,
91         neurons_n,
92     ), f"V shape {args.V.shape} does not match (neurons_n, neurons_n) = ({
93         neurons_n},{neurons_n})"
94     assert (
95         (neurons_n == 2) if args.show else True
96     ), "Visualization only supported for 2 neurons"
97     assert (
98         args.show and args.use_cpp
99     ) == False, "C++ implementation does not support visualization"
100
101     snn = SNN(
102         i0=args.i0,
103         u0=args.u0,
104         V=args.V,
105         b=args.b,
106         theta=args.theta,

```

```

106     alpha=args.alpha,
107     beta=args.beta,
108     T=args.T,
109     neurons_n=neurons_n,
110 )
111 return snn, args.show, args.use_cpp

```

Code 1: src/args.py

```

1 import numpy as np
2 import math
3 from helper import all_subsets
4
5
6 def g(snn, t, st):
7     i0, u0, V, b, theta, alpha, beta = (
8         snn.i0,
9         snn.u0,
10        snn.V,
11        snn.b,
12        snn.theta,
13        snn.alpha,
14        snn.beta,
15    )
16    return -(
17        sum(
18            beta ** (t - k)
19            * (
20                alpha**k * i0
21                + b
22                + sum(alpha ** (k - l) * V @ st[l - 1] for l in range(1, k
23            + 1))
24        )
25        for k in range(1, t + 1)
26    )
27    + beta**t * u0
28    - theta * (1 + sum(beta ** (t - k) * st[k] for k in range(1, t)))
29    ) / sum(
30        beta ** (t - k) * sum(alpha ** (k - l) for l in range(1, k + 1))
31        for k in range(1, t + 1)
32    )
33
34 def get_finite_vertex_bounds(snn):
35     lB = ()
36     uB = ()
37     for i in range(snn.neurons_n):
38         sLower = (np.full_like(snn.u0, False),) + tuple(
39             np.array([i != j and snn.V[i][j] > 0 for j in range(snn.
40 neurons_n)])
41             for t in range(1, snn.T + 1)
42         )
43         sUpper = (np.full_like(snn.u0, False),) + tuple(
44             np.array([i == j or snn.V[i][j] < 0 for j in range(snn.
45 neurons_n)])
46             for t in range(1, snn.T + 1)
47         )
48         lB += (min([g(snn, t, sLower)[i] for t in range(1, snn.T + 1)]),)

```

```

47         uB += (max([g(snn, t, sUpper)[i] for t in range(1, snn.T + 1)]),)
48     return np.array(lB), np.array(uB)
49
50
51 def compute_regions_starting_with(snn, st, lowerBound, upperBound):
52     t = len(st)
53     if t > snn.T:
54         return [(st, lowerBound, upperBound)]
55     x = g(snn, t, st)
56     swappingNeurons = ((lowerBound < x) & (x < upperBound)).nonzero()[0]
57     alwaysActiveNeurons = x <= lowerBound
58     regions_n = []
59     for nowActive in all_subsets(swappingNeurons):
60         inactive = np.setdiff1d(swappingNeurons, nowActive)
61         activeNeurons = alwaysActiveNeurons.copy()
62         activeNeurons[list(nowActive)] = True
63
64         newSt = st + (activeNeurons,)
65         newLowerBound = lowerBound.copy()
66         newLowerBound[list(nowActive)] = x[list(nowActive)]
67         newUpperBound = upperBound.copy()
68         newUpperBound[inactive] = x[inactive]
69         regions_n += compute_regions_starting_with(
70             snn, newSt, newLowerBound, newUpperBound
71         )
72     return regions_n
73
74
75 def compute_regions(snn):
76     return compute_regions_starting_with(
77         snn,
78         (np.full_like(snn.u0, False),),
79         np.full_like(snn.u0, -math.inf),
80         np.full_like(snn.u0, math.inf),
81     )

```

Code 2: src/compute_regions_py.py

```

1  from dataclasses import dataclass
2  import numpy as np
3
4
5  @dataclass
6  class SNN:
7      i0: np.ndarray
8      u0: np.ndarray
9      V: np.ndarray
10     b: np.ndarray
11     theta: float
12     alpha: float
13     beta: float
14     T: int
15     neurons_n: int

```

Code 3: src/datatypes.py

```

1  from itertools import chain, combinations
2

```

```

3
4 def geometric_sum(ratio, n):
5     if ratio == 1:
6         return n + 1
7     return (1 - ratio ** (n + 1)) / (1 - ratio)
8
9
10 def all_subsets(iterable):
11     items = list(iterable)
12     return chain.from_iterable(combinations(items, r) for r in range(len(
        items) + 1))

```

Code 4: src/helper.py

```

1 import numpy as np
2 import math
3 import matplotlib.pyplot as plt
4 from matplotlib.patches import Rectangle
5 import matplotlib.colors as mcolors
6 import compute_regions_cpp
7 import compute_regions_py
8 from args import parseCMDLine
9
10
11 def check_res(res):
12     sts = [tuple(tuple(s) for s in st) for st, _, _ in res]
13     assert len(sts) == len(set(sts)), "Duplicate states found"
14
15
16 def visualize(snn, res):
17     def repl_inf(it):
18         return tuple(
19             (-1e10 if np.isneginf(x) else (1e10 if np.isposinf(x) else x))
20             for x in it
21         )
22
23     def uint2colScal(x):
24         n = 4
25         return (1 / n) * math.log(1 + (math.exp(n) - 1) * x / math.pow(2,
26             snn.T))
27
28     fig, ax = plt.subplots()
29
30     for st, lowerBound, upperBound in res:
31         lowerBound = repl_inf(lowerBound)
32         upperBound = repl_inf(upperBound)
33         width = upperBound[0] - lowerBound[0]
34         height = upperBound[1] - lowerBound[1]
35         if width < 0.0001 or height < 0.0001:
36             print("highlighting small region", st, lowerBound, upperBound)
37             ax.add_patch(
38                 Rectangle(
39                     lowerBound,
40                     max(width, 0.01),
41                     max(height, 0.01),
42                     alpha=0.8,
43                     facecolor="white",
44                     zorder=10,

```

```

43         )
44     )
45     else:
46         num_st = map(
47             lambda sti: sti.dot(1 << np.arange(sti.size)[::-1]),
48             np.array(st).transpose(),
49         )
50         hex_color = mcolors.to_hex(tuple(uint2colScal(x) for x in
num_st) + (0,))
51         ax.add_patch(
52             Rectangle(
53                 lowerBound,
54                 width,
55                 height,
56                 alpha=0.7,
57                 facecolor=hex_color,
58                 edgecolor="black",
59                 zorder=1,
60             )
61         )
62
63     lB, uB = compute_regions_py.get_finite_vertex_bounds(snn)
64     import matplotlib.patches as patches
65
66     rect = patches.Rectangle(
67         lB, uB[0] - lB[0], uB[1] - lB[1], edgecolor="blue", facecolor="none
68     )
69     ax.add_patch(rect)
70     ax.set_xlim(lB[0] - 0.5, uB[0] + 0.5)
71     ax.set_ylim(lB[1] - 0.5, uB[1] + 0.5)
72     ax.set_aspect("equal", adjustable="box")
73     plt.show()
74
75
76 if __name__ == "__main__":
77     snn, show, use_cpp = parseCMDLine()
78     if use_cpp:
79         snnConfig = compute_regions_cpp.SNNConfig(
80             snn.i0,
81             snn.u0,
82             snn.V,
83             snn.b,
84             snn.theta,
85             snn.alpha,
86             snn.beta,
87             snn.T,
88             snn.neurons_n,
89         )
90         n = compute_regions_cpp.compute_regions(snnConfig)
91         print(f"Regions: {n}")
92     else:
93         res = compute_regions_py.compute_regions(snn)
94
95         check_res(res)
96
97         print(f"Regions: {len(res)}")
98

```

```

99         if show:
100             visualize(snn, res)

```

Code 5: src/main.py

```

1  import numpy as np
2
3
4  def first_layer(u0, , W, b, V, , x):
5      u = u0
6      s = np.zeros_like(u0)
7      while True:
8          u = * u + W @ x + b + V @ s - * s
9          s = (u >= ).astype(float)
10         yield u, s

```

Code 6: src/regions.py

```

1  from setuptools import setup, Extension
2  from setuptools.command.build_ext import build_ext
3  import pybind11
4
5
6  class BuildExt(build_ext):
7      """Custom build extension to set compiler flags."""
8
9      def build_extensions(self):
10         c = self.compiler.compiler_type
11         opts = []
12         if c == "unix":
13             opts = [
14                 "-O3",
15                 "-march=native",
16                 "-flto",
17                 "-fno-math-errno",
18                 "-fPIC",
19                 "-DDEBUG",
20             ]
21         for ext in self.extensions:
22             ext.extra_compile_args = opts
23         build_ext.build_extensions(self)
24
25
26  print([pybind11.get_include()])
27  ext_modules = [
28      Extension(
29          "compute_regions_cpp",
30          ["compute_regions_cpp.cpp"],
31          include_dirs=[pybind11.get_include()],
32          language="c++",
33      )
34  ]
35
36  setup(
37      name="compute_regions_cpp",
38      version="0.1",
39      ext_modules=ext_modules,
40      cmdclass={"build_ext": BuildExt},

```

```

41 )
42
43 # build with
44 # python setup.py build_ext --inplace

```

Code 7: src/setup.py

```

1  #define EIGEN_NO_DEBUG
2  #include <Eigen/Dense>
3  #include <iomanip>
4  #include <iostream>
5  #include <limits>
6  #include <pybind11/eigen.h>
7  #include <pybind11/pybind11.h>
8  #include <pybind11/stl.h>
9
10 namespace py = pybind11;
11
12 using std::vector;
13 namespace E = Eigen;
14
15 struct SNNConfig {
16     E::VectorXd i0;
17     E::VectorXd u0;
18     E::MatrixXd V;
19     E::VectorXd b;
20     double theta;
21     double alpha;
22     double beta;
23     size_t T;
24     size_t neurons_n;
25 };
26
27 const double pos_inf = std::numeric_limits<double>::infinity();
28 const double neg_inf = -std::numeric_limits<double>::infinity();
29
30 double geometric_series(double r, size_t n) {
31     if (r == 1) {
32         return n + 1;
33     } else {
34         return (1 - pow(r, n + 1)) / (1 - r);
35     }
36 }
37
38 template <typename Derived> std::vector<int> where(const Eigen::ArrayBase<
    Derived> &mask) {
39     static_assert(std::is_same<typename Derived::Scalar, bool>::value,
40         "where() requires a boolean Array");
41
42     std::vector<int> indices;
43     indices.reserve(mask.size());
44
45     for (int i = 0; i < mask.size(); ++i) {
46         if (mask(i)) {
47             indices.push_back(i);
48         }
49     }
50     return indices;

```

```

51 }
52
53 size_t __compute_regions(const SNNConfig &conf) {
54     const vector<double> gDiv = ([&] {
55         vector<double> gDiv;
56         gDiv.push_back(-1);
57         for (size_t t = 1; t <= conf.T; ++t) {
58             double sum = 0;
59             for (size_t k = 1; k <= t; ++k) {
60                 double innerSum = 0;
61                 for (size_t l = 1; l <= k; ++l) {
62                     innerSum += pow(conf.alpha, k - l);
63                 }
64                 sum += pow(conf.beta, t - k) * innerSum;
65             }
66             gDiv.push_back(-1 / sum);
67         }
68         return gDiv;
69     })();
70     const vector<E::VectorXd> tDep = ([&] {
71         vector<E::VectorXd> tDep;
72         for (size_t t = 0; t <= conf.T; ++t) {
73             tDep.push_back(pow(conf.beta, t) * conf.u0 - conf.theta * E::VectorXd
74                 ::Ones(conf.neurons_n));
75         }
76         return tDep;
77     })();
78     auto run = [&](this auto self, const SNNConfig &conf, vector<E::VectorXd>
79         &st,
80         const E::VectorXd &inputOffset, const E::VectorXd &
81         inputVOffset,
82         const E::VectorXd &spikeSum, const E::VectorXd &lowerBound
83         ,
84         const E::VectorXd &upperBound) -> size_t {
85         const auto t = st.size();
86         if (t == 0) {
87             throw std::invalid_argument("Spike time vector is empty.");
88         }
89         if (t > conf.T) {
90             return 1;
91         }
92         const auto x = (inputOffset + tDep[t] - (conf.theta * conf.beta) *
93             spikeSum) * gDiv[t];
94         const auto swappingNeurons =
95             (lowerBound.array() < x.array()) && (x.array() < upperBound.array()
96             );
97         const auto swappingNeuronsIndices = where(swappingNeurons);
98         const E::ArrayXd alwaysActiveNeurons = (x.array() <= lowerBound.array()
99             ).cast<double>();
100         auto regions_n = 0;
101         if (swappingNeuronsIndices.size() >= sizeof(unsigned long long) * 8) {
102             throw std::runtime_error("Too many swapping neurons, cannot compute

```

```

regions.");
102     }
103     for (unsigned long long i = 0; i < (1ull << swappingNeuronsIndices.size()) ++i) {
104         E::VectorXd newLowerBound = lowerBound;
105         E::VectorXd newUpperBound = upperBound;
106         E::VectorXd newSpike = alwaysActiveNeurons.matrix();
107
108         for (size_t j = 0; j < swappingNeuronsIndices.size(); ++j) {
109             auto sjI = swappingNeuronsIndices[j];
110             if ((i >> j) & 1) {
111                 newLowerBound(sjI) = x(sjI);
112                 newSpike(sjI) = 1;
113             } else {
114                 newUpperBound(sjI) = x(sjI);
115             }
116         }
117
118         // TODO: compute after branch instead
119         const E::VectorXd newSpikeSum = conf.beta * spikeSum + newSpike;
120         const E::VectorXd newInputVOffset = conf.alpha * inputVOffset + conf.
V * newSpike;
121         const E::VectorXd newInputOffset =
122             conf.beta * inputOffset + pow(conf.alpha, t + 1) * conf.i0 + conf.
.b + newInputVOffset;
123         st.push_back(std::move(newSpike));
124         regions_n += self(conf, st, newInputOffset, newInputVOffset,
newSpikeSum, newLowerBound,
125             newUpperBound);
126         st.pop_back();
127     }
128
129     return regions_n;
130 };
131
132 const E::VectorXd initSpikeSum = E::VectorXd::Zero(conf.neurons_n);
133 const E::VectorXd initInputOffset = conf.alpha * conf.i0 + conf.b;
134 const E::VectorXd initInputVOffset = E::VectorXd::Zero(conf.neurons_n);
135 vector<E::VectorXd> initSt = {E::VectorXd::Zero(conf.neurons_n)};
136 return run(conf, initSt, initInputOffset, initInputVOffset, initSpikeSum,
137     E::VectorXd::Constant(conf.neurons_n, neg_inf),
138     E::VectorXd::Constant(conf.neurons_n, pos_inf));
139 }
140
141 size_t compute_regions(const SNNConfig &conf) {
142     if (conf.V.rows() != conf.neurons_n || conf.V.rows() != conf.V.cols()) {
143         throw std::invalid_argument("Malformed matrix V");
144     }
145
146     if (conf.u0.size() != conf.neurons_n) {
147         throw std::invalid_argument("Malformed vector u0");
148     }
149
150     vector<E::VectorXd> initSt = {E::VectorXd::Zero(conf.neurons_n)};
151     return __compute_regions(conf);
152 }
153
154 PYBIND11_MODULE(compute_regions_cpp, m) {

```

```

155 m.doc() = "Module for computing regions of dt lif snns ";
156 m.def("compute_regions", &compute_regions, py::arg("conf"),
157       "Compute number of regions of r dt lif snns.");
158 py::class_<SNNConfig>(m, "SNNConfig")
159     .def(py::init<E::VectorXd, E::VectorXd, E::MatrixXd, E::VectorXd,
160           double, double, double,
161           size_t, size_t>())
162     .def_readwrite("i0", &SNNConfig::i0)
163     .def_readwrite("u0", &SNNConfig::u0)
164     .def_readwrite("V", &SNNConfig::V)
165     .def_readwrite("b", &SNNConfig::b)
166     .def_readwrite("theta", &SNNConfig::theta)
167     .def_readwrite("alpha", &SNNConfig::alpha)
168     .def_readwrite("beta", &SNNConfig::beta)
169     .def_readwrite("T", &SNNConfig::T)
170     .def_readwrite("neurons_n", &SNNConfig::neurons_n);
171 }

```

Code 8: src/compute_regions_cpp.cpp

7.2 dt-lif-snn-compute-regions

See [Herrmann, 2025a] for the corresponding git repository.

```

1 import matplotlib.pyplot as plt
2 import numpy as np
3 import glob
4
5 X = np.arange(32)
6
7 plt.rcParams.update({"font.size": 14})
8
9 for file_path in glob.glob("output4096.csv"):
10     data = np.loadtxt(file_path, delimiter=",")
11     # plt.plot(X, data[:32], label="#Counted Regions")
12
13 for i in range(4, 7):
14     plt.plot(X, data / X**i, label=f"Regions / T^{i}")
15
16 # plt.plot(X, (((X**2 + X + 2) / 2) ** 2), label=f"(T^2 + T + 2)^2 / 2^2")
17
18 plt.title("")
19 # plt.yscale("log")
20 plt.xlabel("T")
21 plt.ylabel("#Regions")
22 plt.legend(loc="upper left")
23 # plt.ylim(0, 3)
24 plt.grid(True)
25 plt.show()

```

Code 9: src/analyze.py

```

1 from contextlib import contextmanager
2 import time
3 import moderngl
4 from pathlib import Path
5
6

```

```

7 @contextmanager
8 def measure_perf(label):
9     start = time.perf_counter()
10    try:
11        yield ()
12    finally:
13        end = time.perf_counter()
14        print(f"Elapsed time ({label}): {end - start:.6f} seconds")
15
16 @contextmanager
17 def measure_perf_gpu(ctx, label):
18     timer = ctx.query(time=True)
19     try:
20         with timer:
21             yield ()
22     finally:
23         elapsed_ns = timer.elapsed
24         print(f"Elapsed time ({label}): {elapsed_ns/10**9:.6f} seconds")
25
26
27 def mkdirp(path):
28     Path(path).mkdir(parents=True, exist_ok=True)
29
30
31 def chunked_iterable(iterable, n):
32     """Yield successive n-sized chunks from iterable."""
33     for i in range(0, len(iterable), n):
34         yield iterable[i : i + n]
35
36
37 def getUniformsDict(shader):
38     return {
39         name: shader[name].value
40         for name in shader
41         if isinstance(shader[name], moderngl.Uniform)
42     }
43
44
45 def compileCompShaderFile(ctx, filename):
46     """Compile a compute shader from a file."""
47     with open(filename, "r") as f:
48         shader_code = f.read()
49     return ctx.compute_shader(shader_code)
50
51
52 def getUniformsDictSpliced(shader):
53     res = {}
54     for name in shader:
55         uniform = shader[name]
56         if isinstance(uniform, moderngl.Uniform):
57             for i in range(uniform.array_length):
58                 for j in range(uniform.dimension):
59                     cpName = f"{name}"
60                     value = uniform.value
61                     if uniform.array_length > 1:
62                         cpName += f"[{i}]"
63                     value = value[i]

```

```

65         if uniform.dimension > 1:
66             cpName += f"[{j}]"
67             value = value[j]
68             res[cpName] = value
69     return res

```

Code 10: src/common.py

```

1  from abc import ABC, abstractmethod
2
3
4  class CountRegions(ABC):
5      def setUniforms(self, **kwargs):
6          """Set uniforms for the shader."""
7          for key, value in kwargs.items():
8              if key in self.shader:
9                  self.shader[key] = value
10             else:
11                 raise KeyError(f"Uniform '{key}' not found in shader.")
12
13     @abstractmethod
14     def run(self, iterationsR, uOR, betaR, bR, WR, VR, thetaR, scale,
15            offset):
16         pass

```

Code 11: src/count_regions.py

```

1  import moderngl
2  import numpy as np
3  from common import getUniformsDict, getUniformsDictSpliced,
4      compileCompShaderFile
5  from count_regions import CountRegions
6  import itertools
7
8  class CountRegionsEfficient(CountRegions):
9      def __init__(self, imagep, spikeTrp, maxSizePot2):
10         # Create context (offscreen)
11         self.imagep = imagep
12         self.spikeTrp = spikeTrp
13         self.ctx = moderngl.create_standalone_context(backend="egl")
14         self.shader = compileCompShaderFile(self.ctx, "shaders/
15         uintEfficient.glsl")
16         self.shaderCorners = compileCompShaderFile(self.ctx, "shaders/
17         corners.glsl")
18
19         self.resSize = 2
20         assert maxSizePot2 >= 6
21         self.maxSizeI = maxSizePot2 - 6
22         self.initialSize = 64 # 2**6
23         self.scalePrev = 2
24
25         self.bufData = []
26         self.bufColor = []
27         self.bufPrev = []
28         self.bufRegionsN = self.ctx.buffer(reserve=4)
29         self.bufRegionsN.bind_to_storage_buffer(binding=3)
30         for i in range(0, self.maxSizeI + 1):

```

```

29         size = self.initialSize * self.scalePrev**i
30         self.bufData.append(self.ctx.buffer(reserve=size * size * self.
resSize * 4))
31         self.bufColor.append(self.ctx.buffer(reserve=size * size * 4))
32         self.bufPrev.append(self.ctx.buffer(reserve=size * size * self.
resSize * 4))
33
34     def run(self, iterationsR, uOR, betaR, bR, WR, VR, thetaR, scale,
offset):
35         ctx = self.ctx
36         shader = self.shader
37
38         for iterations, u0, beta, b, W, V, theta in itertools.product(
39             iterationsR, uOR, betaR, bR, WR, VR, thetaR
40         ):
41             for i in range(0, self.maxSizeI + 1):
42                 prevSize = self.initialSize * self.scalePrev ** (i - 1)
43                 size = self.initialSize * self.scalePrev**i
44                 print(size)
45
46                 self.bufData[i].bind_to_storage_buffer(binding=0)
47                 self.bufColor[i].bind_to_storage_buffer(binding=1)
48                 self.bufPrev[i].bind_to_storage_buffer(binding=2)
49
50                 initialRun = i == 0
51                 self.setUniforms(
52                     iResolution=(size, size),
53                     iterations=iterations,
54                     u0=u0,
55                     beta=beta,
56                     # b=b,
57                     # W=W,
58                     V=V,
59                     theta=theta,
60                     scale=scale,
61                     offset=offset,
62                     initialRun=initialRun,
63                     scalePrev=self.scalePrev,
64                     imagep=self.imagep,
65                 )
66
67                 assert size % 8 == 0, "Size must be a multiple of 8"
68                 shader.run(group_x=size // 8, group_y=size // 8)
69
70                 assert self.resSize == 2
71                 self.bufRegionsN.write(np.array([0], dtype=np.uint32))
72                 self.shaderCorners["iResolution"] = size, size
73                 assert size % 8 == 0, "Size must be a multiple of 8"
74                 self.shaderCorners.run(group_x=size // 8, group_y=size //
8)
75
76                 regions_n = np.frombuffer(self.bufRegionsN.read(), dtype=np
.uint32)[0]
77
78                 print(f"Regions for {size}:", regions_n)
79
80                 if i != self.maxSizeI:
81                     ctx.copy_buffer(self.bufPrev[i + 1], self.bufData[i])

```

```

82         imageArray = None
83         if self.imagep:
84             imageArray = np.frombuffer(
85                 self.bufColor[i].read()[0 : size * size * 4], dtype
86                 =np.byte
87                 ).reshape((size, size, 4))
88
89         uniforms = getUniformsDict(shader)
90         uniformsSpliced = getUniformsDictSpliced(shader)
91         yield regions_n, imageArray, uniforms, uniformsSpliced

```

Code 12: src/count_regions_efficient.py

```

1  import moderngl
2  import numpy as np
3  import unique_bytes
4  import itertools
5  from common import getUniformsDict, getUniformsDictSpliced,
6  compileCompShaderFile
7  from count_regions import CountRegions
8
9  class CountRegionsSimple(CountRegions):
10     def __init__(self, imagep, spikeTrp, sizeR):
11         # Create context (offscreen)
12         self.imagep = imagep
13         self.spikeTrp = spikeTrp
14         self.ctx = moderngl.create_standalone_context(backend="egl")
15         self.shader = compileCompShaderFile(self.ctx, "shaders/uint.glsl")
16         self.sizeR = sizeR
17
18     def setUniforms(self, **kwargs):
19         """Set uniforms for the shader."""
20         for key, value in kwargs.items():
21             if key in self.shader:
22                 self.shader[key] = value
23             else:
24                 raise KeyError(f"Uniform '{key}' not found in shader.")
25
26     def getSpiketrains(self, bufData, size, resSize):
27         """Count unique chunks in the data buffer."""
28         return unique_bytes.unique_regions(bufData.read(), size, resSize *
29     4)
30
31     def printSpikeTrains(self, regionsSt, resSize, iterations):
32         """Print spiketrains from the data buffer."""
33         for spiketrain in regionsSt:
34             rs = resSize
35             # fmt: off
36             print("x:", "".join(f"{byte:08b}"[::-1] for byte in spiketrain
37     [0 : rs * 2])[:iterations][::-1],
38             "y:", "".join(f"{byte:08b}"[::-1] for byte in spiketrain[
39     rs * 2 : rs * 4])[:iterations][::-1])
40             # fmt: on
41
42     def countUniqueChunks(self, bufData, size, resSize):
43         """Count unique chunks in the data buffer."""
44         return len(self.getSpiketrains(bufData, size, resSize))

```

```

42
43     def run(self, iterationsR, uOR, betaR, bR, WR, VR, thetaR, scale,
44         offset):
45         ctx = self.ctx
46         shader = self.shader
47         for size in self.sizeR:
48             resSize = 4
49
50             # size * size * resSize many uints
51             bufData = ctx.buffer(reserve=size * size * resSize * 4)
52             bufColor = ctx.buffer(reserve=size * size * 4)
53
54             # Bind buffer to binding=0
55             bufData.bind_to_storage_buffer(binding=0)
56             bufColor.bind_to_storage_buffer(binding=1)
57
58             for iterations, u0, beta, b, W, V, theta in itertools.product(
59                 iterationsR, uOR, betaR, bR, WR, VR, thetaR
60             ):
61                 self.setUniforms(
62                     iResolution=(size, size),
63                     iterations=iterations,
64                     u0=u0,
65                     beta=beta,
66                     b=b,
67                     W=W,
68                     V=V,
69                     theta=theta,
70                     scale=scale,
71                     offset=offset,
72                 )
73
74                 assert size % 32 == 0, "Size must be a multiple of 32"
75                 shader.run(group_x=size // 32, group_y=size // 32)
76
77                 regions_n = self.countUniqueChunks(bufData, size, resSize)
78
79                 imageArray = None
80                 if self.imagep:
81                     imageArray = np.frombuffer(bufColor.read(), dtype=np.
82                         byte).reshape(
83                             (size, size, 4)
84                         )
85
86                 uniforms = getUniformsDict(shader)
87                 uniformsSpliced = getUniformsDictSpliced(shader)
88                 yield regions_n, imageArray, uniforms, uniformsSpliced

```

Code 13: src/count_regions_simple.py

```

1 import numpy as np
2 from PIL import Image
3 import sys
4 import csv
5 import itertools
6 from count_regions_simple import CountRegionsSimple
7 from count_regions_efficient import CountRegionsEfficient
8

```

```

9  ## args
10 efficientp = "--efficient" in sys.argv[1:]
11 imagep = "--image" in sys.argv[1:]
12 spikeTrp = "--spiketrain" in sys.argv[1:]
13
14
15 def once(v):
16     yield v
17
18
19 ls = np.linspace
20 itpr = itertools.product
21 iterationsR = range(10, 20)
22 uOR = itpr(ls(0.0123456789, 1, 5), ls(0.0123456789, 1, 5))
23 betaR = ls(0.5, 1, 10)
24 WR = once((1.0, 0.0, 0.0, 1.0)) # doesn't increase number of regions if
    changed
25 bR = once((0.0, 0.0)) # doesn't increase number of regions if changed
26 VR = itpr(ls(-1, 1, 10), ls(-1, 1, 10), ls(-1, 1, 10), ls(-1, 1, 10))
27 thetaR = ls(0, 1, 10)
28 scale = 2
29 offset = -0.5, -0.5
30 maxSizePot2 = 12 # size = 2**maxSizePot2
31
32 if efficientp:
33     counter = CountRegionsEfficient(
34         imagep=imagep,
35         spikeTrp=spikeTrp,
36         maxSizePot2=maxSizePot2,
37     )
38 else:
39     sizeR = map(lambda x: 2**x, range(6, maxSizePot2 + 1))
40     counter = CountRegionsSimple(imagep=imagep, spikeTrp=spikeTrp, sizeR=
        sizeR)
41
42 counterGen = counter.run(iterationsR, uOR, betaR, bR, WR, VR, thetaR, scale
    , offset)
43
44 regions = []
45 for regions_n, imageArray, uniforms, uniformsSpliced in counterGen:
46     iResolution = uniforms["iResolution"]
47     iResolution = (int(iResolution[0]), int(iResolution[1]))
48     iterations = uniforms["iterations"]
49     print(f"iResolution: {iResolution}, iterations: {iterations}")
50     if imagep:
51         image = Image.frombytes(
52             "RGBA",
53             iResolution,
54             imageArray,
55         )
56         image = image.transpose(Image.FLIP_TOP_BOTTOM)
57         image.save(
58             f"images/output_size{iResolution[0]:04}_{iResolution[1]:04}
                _iteration{iterations:02}.png"
59         )
60
61     regions.append({"regions_n": regions_n} | uniformsSpliced)
62     print(f"Regions for {uniformsSpliced}:\n{regions_n}")

```

```

63
64
65 with open("output.csv", "w", newline="") as f:
66     writer = csv.DictWriter(f, fieldnames=["regions_n"] + list(
        uniformsSpliced.keys()))
67     writer.writeheader()
68     writer.writerows(regions)

```

Code 14: src/main.py

```

1  import numpy as np
2  from itertools import islice
3  import matplotlib.pyplot as plt
4
5
6  def get_nth_result(generator, n):
7      return next(islice(generator, n, n + 1))
8
9
10 def first_layer(u0, , W, b, V, , x):
11     u = u0
12     s = np.zeros_like(u0)
13     while True:
14         u = * u + W @ x + b + V @ s - * * s
15         s = (u >= ).astype(float)
16         yield u, s
17
18
19 def run_with(V, x, W=[[1, 0], [0, 1]], b=None, =1, u0=None, =1):
20     if u0 is None:
21         u0 = np.zeros_like(x)
22     if b is None:
23         b = np.zeros_like(x)
24     if isinstance(W, list):
25         W = np.array(W)
26     if isinstance(V, list):
27         V = np.array(V)
28     if isinstance(b, list):
29         b = np.array(b)
30     if isinstance(x, list):
31         x = np.array(x)
32     if isinstance(u0, list):
33         u0 = np.array(u0)
34
35     for u, s in first_layer(u0, , W, b, V, , x):
36         print("Membrane potential:", u)
37         print("Spikes:", s)
38         input()
39
40
41 def spiketrain_to_nextspike(spiketrain, V, =1, u0=None):
42     # we assume b=0 and W =
43     if u0 is None:
44         u0 = np.zeros_like(V[0])
45     if isinstance(u0, list):
46         u0 = np.array(u0)
47     lower_bounds = [[] for _ in range(len(u0))]
48     upper_bounds = [[] for _ in range(len(u0))]

```

```

49     stW0 = np.append([[0, 0]], spiketrain, axis=0)
50     one_n = np.ones_like(u0)
51     for t, spike in enumerate(stW0):
52         if t == 0:
53             continue
54         # we compute x, so that u(t) =
55         stSum = np.sum(stW0[:t], axis=0)
56         xCut = ( * one_n - u0 - (V @ stSum - * stSum)) / t
57         for i, x in enumerate(xCut):
58             if spike[i] == 1.0:
59                 lower_bounds[i].append(x)
60             else:
61                 upper_bounds[i].append(x)
62         # print("Lower bounds:", lower_bounds)
63         # print("Upper bounds:", upper_bounds)
64         infimums = np.array([max(lb) if lb else -np.inf for lb in lower_bounds
65 ])
66         # print("Highest lower bound:", infimums)
67         supremums = np.array([min(ub) if ub else np.inf for ub in upper_bounds
68 ])
69         # print("Lowest upper bound:", supremums)
70
71     def get_continuation_us(x):
72         return list(
73             islice(
74                 first_layer(
75                     u0=u0, =1, W=[[1, 0], [0, 1]], b=np.zeros_like(u0), V=V
76                     , =, x=x
77                 ),
78                 0,
79                 len(spiketrain) + 1,
80             )
81         )
82
83     check_if_spikes_fit = lambda stRes: spiketrain == list(
84         map(lambda x: list(x[1]), stRes[:-1])
85     )
86
87     # The spike trains at infimums might not be correct,
88     # since we can't properly compute with  $\omega^-$ .
89     # Also the supremums are probably not correct,
90     # since they should be the highest value that just produces another
91     # spike train.
92     def find_actual_possible(x, other):
93         y = np.copy(x)
94         while np.all((x < other) == (y < other)) and np.all((other < x) ==
95 (other < y)):
96             stRes = get_continuation_us(x)
97             if check_if_spikes_fit(stRes):
98                 return stRes
99             x = np.nextafter(x, other)
100             print(f"trying next {x}")
101         else:
102             raise ValueError(f"no possible next value for {x} with other {
103 other}")
104
105     # print(f"checking spikes for infimums")
106     # FIXME: wrong for negative weights in V

```

```

101     stInf = find_actual_possible(infimums, supremums)
102     uInf, sInf = stInf[-1]
103
104     # print(f"checking spikes for supremums")
105     stSup = find_actual_possible(supremums, infimums)
106     uSup, sSup = stSup[-1]
107
108     assert check_if_spikes_fit(stSup)
109     if np.array_equal(sInf, sSup):
110         # print(f"next spikes are fixed to {sInf}")
111         return {"next": [sInf], "infs": infimums, "sups": supremums}
112     else:
113         # print(f"next spikes might be {sInf} or {sSup}")
114         return {"next": [sInf, sSup], "infs": infimums, "sups": supremums}
115
116
117 def play(startSpikes, V, , u0):
118     spiketrain = [startSpikes]
119     forcedp = [False]
120     while True:
121         result = spiketrain_to_nextspike(spiketrain, V, , u0)
122         forcedp.append(len(result["next"]) == 1)
123         print("Infimums:", result["infs"])
124         print("Supremums:", result["sups"])
125         print(
126             "Spikes:\n"
127             + "\n".join(
128                 map(
129                     lambda n: "".join(map(lambda x: "1" if x == 1 else "0",
n)),
130                     zip(*spiketrain),
131                 )
132             )
133             + "\n"
134             + "".join(["^" if f else " " for f in forcedp])
135         )
136         print("Possible next spikes:", result["next"])
137         choice = input("Choose next spikes (or 'q' to quit): ")
138         if choice.lower() == "q":
139             break
140         try:
141             next_spikes = result["next"][int(choice)]
142             spiketrain.append(list(next_spikes))
143         except (ValueError, IndexError):
144             print("Invalid input, please enter q or an integer.")
145
146
147 def plot(V, n, lower=-1, upper=1, steps=100):
148     x = np.linspace(lower, upper, steps)
149     y = np.linspace(lower, upper, steps)
150
151     X, Y = np.meshgrid(x, y)
152
153     points = np.stack([X.ravel(), Y.ravel()], axis=1)
154     g = lambda p: list(
155         islice(
156             first_layer(
157                 u0=np.zeros_like(p),

```

```

158         =1,
159         W=[[1, 0], [0, 1]],
160         b=np.zeros_like(p),
161         V=V,
162         =1,
163         x=p,
164     ),
165     0,
166     n,
167 )
168 )
169 Z = np.array([g(p) for p in points])
170 Z = Z.reshape(X.shape)
171
172 plt.figure(figsize=(6, 6))
173 plt.pcolormesh(X, Y, Z, cmap="Set2", shading="auto")
174 plt.colorbar(label="Group")
175 plt.title("Grouped Grid Function Output")
176 plt.xlabel("x")
177 plt.ylabel("y")
178 plt.axis("equal")
179 plt.show()

```

Code 15: src/regions.py

```

1 from setuptools import setup, Extension
2 from setuptools.command.build_ext import build_ext
3 import pybind11
4
5
6 class BuildExt(build_ext):
7     """Custom build extension to set compiler flags."""
8
9     def build_extensions(self):
10         c = self.compiler.compiler_type
11         opts = []
12         if c == "unix":
13             opts = ["-O3", "-fno-math-errno", "-fPIC", "-DDEBUG"]
14             for ext in self.extensions:
15                 ext.extra_compile_args = opts
16             build_ext.build_extensions(self)
17
18
19 print([pybind11.get_include()])
20 ext_modules = [
21     Extension(
22         "unique_bytes",
23         ["unique_bytes.cpp"],
24         include_dirs=[pybind11.get_include()],
25         language="c++",
26     )
27 ]
28
29 setup(
30     name="unique_bytes",
31     version="0.1",
32     ext_modules=ext_modules,
33     cmdclass={"build_ext": BuildExt},

```

```

34 )
35
36 # build with
37 # python setup.py build_ext --inplace

```

Code 16: src/setup.py

```

1  import moderngl
2  import time
3  from contextlib import contextmanager
4
5  resSize = 4
6
7
8  @contextmanager
9  def measure_perf(label):
10     start = time.perf_counter()
11     try:
12         yield ()
13     finally:
14         end = time.perf_counter()
15         print(f"Elapsed time ({label}): {end - start:.6f} seconds")
16
17
18  def is_power_of_two(n):
19     return n > 0 and (n & (n - 1)) == 0
20
21
22  # given buffer has to be bound to binding=0
23  def count_unique(buffer, ctx=None):
24     copyBuffer = False
25     if ctx is None:
26         copyBuffer = True
27         ctx = moderngl.create_standalone_context()
28     with measure_perf(f"Count regions inner"):
29         valSize = resSize * 4 # resSize many uints
30         dataSize = buffer.size
31         print("DataSize:", dataSize)
32         if dataSize % valSize != 0:
33             raise ValueError("Data length must be a multiple of 4.")
34         if not is_power_of_two(dataSize // valSize):
35             raise ValueError("Data length must be a power of two.")
36         if copyBuffer:
37             with measure_perf(f"reserve buffer"):
38                 bufData = ctx.buffer(buffer.read())
39                 # bufData = ctx.buffer(reserve=buffer.size)
40                 # with measure_perf(f"copy buffer"):
41                 # ctx.copy_buffer(bufData, buffer)
42             with measure_perf(f"bind buffer"):
43                 bufData.bind_to_storage_buffer(binding=0)
44         else:
45             bufData = buffer
46         with measure_perf(f"compute shader"):
47             with open("shaders/uniques.glsl", "r") as f:
48                 shader = ctx.compute_shader(f.read())
49
50         with measure_perf(f"shader2"):
51             resNum = ctx.buffer(reserve=4) # one uint

```

```

52         resNum.bind_to_storage_buffer(binding=1)
53
54         with measure_perf(f"run"):
55             i = 1
56             while 2**i * valSize < dataSize:
57                 prevBatchSize = 2**i
58                 shader["prevBatchSize"] = prevBatchSize
59
60                 batchSize = 2 * prevBatchSize
61                 shader.run(group_x=dataSize // (valSize * batchSize))
62                 i += 1
63
64         with measure_perf(f"read"):
65             res = int.from_bytes(resNum.read(), byteorder="little", signed=
False)
66             resNum.release()
67             if copyBuffer:
68                 bufData.release()
69             return res

```

Code 17: src/uniques.py

```

1  #version 330
2  out vec4 fragColor;
3  uniform vec2 iResolution;
4  uniform int iterations;
5  uniform vec2 u0;
6  uniform float beta;
7  uniform vec2 b;
8  uniform mat2 W;
9  uniform mat2 V;
10 uniform float theta;
11 uniform vec2 offset;
12 uniform float scale;
13
14 vec2 first_layer(vec2 x, int iterations, vec2 u0, float beta, mat2 W, vec2
b, mat2 V, float theta)
15 {
16     vec2 u = u0;
17     vec2 s = vec2(0,0);
18     vec2 col = vec2(0,0);
19     for (int i = 0; i < iterations; ++i) {
20         u = beta * u + W * x + b + V * s - theta * beta * s;
21         s = step(theta,u);
22         col = col + pow(2.,float(-i))*s;
23     }
24     return col;
25 }
26
27
28 void main()
29 {
30     // Normalized pixel coordinates (from 0 to scale)
31     vec2 nuv = gl_FragCoord.xy/iResolution.xy;
32     vec2 uv = scale*nuv+offset;
33
34     // Time varying pixel color
35

```

```

36     // int iterations = 1;
37     // vec2 u0 = vec2(0,0);
38     // float beta = 1.;
39     // vec2 b = vec2(0,0);
40     // mat2 W = mat2(1,0,0,1);
41     // mat2 V = mat2(0,0,0,0);
42     // float theta = 1.;
43     vec2 res = first_layer(uv, iterations, u0, beta, W, b, V, theta);
44     vec2 col = log(res+1.)*2.5;
45
46     // Output to screen
47     fragColor = vec4(col, 0.0,1.0);
48 }

```

Code 18: shaders/color.glsl

```

1  #version 330
2  out vec4 fragColor;
3  uniform vec2 iResolution;
4  uniform int iterations;
5  uniform vec2 u0;
6  uniform float beta;
7  uniform vec2 b;
8  uniform mat2 W;
9  uniform mat2 V;
10 uniform float theta;
11 uniform vec2 offset;
12 uniform float scale;
13
14 vec2[2] first_layer(vec2 x, int iterations, vec2 u0, float beta, mat2 W,
15     vec2 b, mat2 V, float theta)
16 {
17     vec2 u = u0;
18     vec2 s = vec2(0,0);
19     vec2 col = vec2(0,0);
20     vec2 col2 = vec2(0,0);
21     for (int i = 0; i < iterations; ++i) {
22         u = beta * u + W * x + b + V * s - theta * beta * s;
23         s = step(theta,u);
24         col = col + pow(2.,float(-i))*s;
25         col2 = col2 + pow(2.,float(i-iterations+1))*s;
26     }
27     return vec2[2](col, col2);
28 }
29
30 void main()
31 {
32     // Normalized pixel coordinates (from 0 to scale)
33     vec2 nuv = gl_FragCoord.xy/iResolution.xy;
34     vec2 uv = scale*nuv+offset;
35
36     // Time varying pixel color
37
38     // int iterations = 1;
39     // vec2 u0 = vec2(0,0);
40     // float beta = 1.;
41     // vec2 b = vec2(0,0);

```

```

42     // mat2 W = mat2(1,0,0,1);
43     // mat2 V = mat2(0,0,0,0);
44     // float theta = 1.;
45     vec2[2] res = first_layer(uv, iterations, u0, beta, W, b, V, theta);
46
47     // Output to screen
48     fragColor = vec4(res[0], res[1]);
49 }

```

Code 19: shaders/color2.glsl

```

1  #version 460
2
3  uniform ivec2 iResolution;
4
5  layout (local_size_x = 8, local_size_y = 8, local_size_z = 1) in;
6
7  layout(std430, binding = 0) buffer GridData {
8      uint[2] data[];
9  };
10 layout(std430, binding = 3) buffer RegionsN {
11     uint regions_n;
12 };
13 // layout(binding = 3) uniform atomic_uint regions_n;
14
15 void main()
16 {
17     int i = int(gl_GlobalInvocationID.x);
18     int j = int(gl_GlobalInvocationID.y);
19
20     // Check if the value at the current position is different
21     // from the value to the left and below it.
22     bool leftIsDiff = i == 0 || (data[(i-1) * iResolution.x + j] != data[i
23 * iResolution.x + j]);
24     bool belowIsDiff = j == 0 || (data[i * iResolution.x + (j - 1)] != data
25 [i * iResolution.x + j]);
26
27     // If it is different indeed, the current position is a ll corner of
28     // a region.
29     // atomicCounterIncrement(regions_n);
30     if (leftIsDiff && belowIsDiff) {
31         // regions_n += 1;
32         atomicAdd(regions_n, 1);
33     }
34 }

```

Code 20: shaders/corners.glsl

```

1  #version 460
2  uniform dvec2 iResolution;
3  uniform int iterations;
4  uniform dvec2 u0;
5  uniform double beta;
6  uniform dvec2 b;
7  uniform mat2 W;
8  uniform mat2 V;
9  uniform double theta;
10 uniform dvec2 offset;

```

```

11 uniform double scale;
12
13 layout (local_size_x = 32, local_size_y = 32, local_size_z = 1) in;
14
15 layout(std430, binding = 0) buffer GridData {
16     uint[4] data[];
17 };
18 layout(std430, binding = 1) buffer GridColor {
19     uint color[];
20 };
21
22 uint[4] first_layer(dvec2 x, int iterations, dvec2 u0, double beta, mat2 W,
23     dvec2 b, mat2 V, double theta)
24 {
25     dvec2 u = u0;
26     uvec2 s = uvec2(0,0);
27     uvec2 spiketrLow = uvec2(0,0);
28     uvec2 spiketrUp = uvec2(0,0);
29     for (int i = 0; i < iterations; ++i) {
30         u = beta * (u - theta * s) + W * x + b + V * s;
31         s = uvec2(step(theta,u));
32         spiketrUp = (spiketrUp << 1) + (spiketrLow >> 31);
33         spiketrLow = (spiketrLow << 1) + s;
34     }
35     return uint[4](spiketrLow.x, spiketrUp.x, spiketrLow.y, spiketrUp.y);
36 }
37
38 dvec2 posToUV(uvec2 pos)
39 {
40     dvec2 nuv = dvec2(pos)/iResolution.xy;
41     return scale*nuv+offset;
42 }
43
44 uint uint2colScal(uint high, uint low)
45 {
46     float n = 10.;
47     float x = float(high) * pow(2., 32.) + float(low);
48     double a = ((1. / n) * log(1 + (exp(n) - 1) * x / pow(2., float(
49         iterations))));
50     // a should be in [0,1] by definition. But just to make sure:
51     return min(255, uint(floor(255*a)));
52 }
53
54 uint uint2col(uint[4] res)
55 {
56     return uint2colScal(res[1], res[0]) | (uint2colScal(res[3], res[2]) <<
57         8) | uint(0xFF000000);
58 }
59
60 uint uint2colBorder(uint[4] res)
61 {
62     dvec2 aBitRightPos = posToUV(gl_GlobalInvocationID.xy+uvec2(1,0));
63     dvec2 aBitUpPos = posToUV(gl_GlobalInvocationID.xy+uvec2(0,1));
64     uint[4] aBitRight = first_layer(aBitRightPos, iterations, u0, beta, W,
65         b, V, theta);
66     uint[4] aBitUp = first_layer(aBitUpPos, iterations, u0, beta, W, b, V,
67         theta);
68     if (res != aBitRight || res != aBitUp) {

```

```

64         return 0xFFFFFFFF; // white
65     } else {
66         return uint2col(res);
67     }
68 }
69
70 void main()
71 {
72     dvec2 uv = posToUV(gl_GlobalInvocationID.xy);
73     uint index = uint(gl_GlobalInvocationID.x + gl_GlobalInvocationID.y *
74         iResolution.x);
75
76     uint[4] res = first_layer(uv, iterations, u0, beta, W, b, V, theta);
77
78     data[index] = res;
79     color[index] = uint2colBorder(res);
80 }

```

Code 21: shaders/uint.glsl

```

1  #version 460
2
3  uniform dvec2 iResolution;
4  uniform int iterations;
5  uniform dvec2 u0;
6  uniform double beta;
7  // W and b can't be used with this optimization
8  // uniform dvec2 b;
9  dvec2 b = dvec2(0.0, 0.0);
10 // uniform mat2 W;
11 mat2 W = mat2(1.0, 0.0, 0.0, 1.0);
12 uniform mat2 V;
13 uniform double theta;
14 uniform dvec2 offset;
15 uniform double scale;
16 uniform int scalePrev;
17 uniform bool initialRun;
18 uniform bool imagep;
19
20 layout (local_size_x = 8, local_size_y = 8, local_size_z = 1) in;
21
22 layout(std430, binding = 0) buffer GridData {
23     uint[2] data[];
24 };
25 layout(std430, binding = 1) buffer GridColor {
26     uint color[];
27 };
28
29 // data from previous run with 1/scalePrev of size unless INITIAL_RUN is
30 // set
31 layout(std430, binding = 2) buffer GridPrevData {
32     uint[2] prevData[];
33 };
34
35 uint[2] first_layer(dvec2 x, int iterations, dvec2 u0, double beta, mat2 V,
36     double theta)
37 {
38     dvec2 u = u0;

```

```

37     uvec2 s = uvec2(0,0);
38     uvec2 spiketr = uvec2(0,0);
39     for (int i = 0; i < iterations; ++i) {
40         u = beta * (u - theta * s) + W * x + b + V * s;
41         s = uvec2(step(theta,u));
42         spiketr = (spiketr << 1) | s;
43     }
44     return uint[2](spiketr.x, spiketr.y);
45 }
46
47 uint uint2col(uint x)
48 {
49     float n = 10.;
50     float a = ((1. / n) * log(1 + (exp(n) - 1) * x / pow(2., float(
51     iterations)))));
52     // a should be in [0,1] by definition. But just to make sure:
53     return min(255, uint(floor(255*a)));
54 }
55
56 void main()
57 {
58     dvec2 nuv = gl_GlobalInvocationID.xy/iResolution.xy;
59     uint index = uint(gl_GlobalInvocationID.x + gl_GlobalInvocationID.y *
60     iResolution.x);
61     dvec2 uv = scale*nuv+offset;
62
63     uint[2] res;
64     ivec2 LDCorner = ivec2(gl_GlobalInvocationID.x / scalePrev,
65     gl_GlobalInvocationID.y / scalePrev);
66     uint LDCIndex = uint(LDCorner.x + LDCorner.y * (iResolution.x /
67     scalePrev));
68     ivec2 RUCorner = ivec2(LDCorner.x + 1, LDCorner.y + 1);
69     uint RUCIndex = uint(RUCorner.x + RUCorner.y * (iResolution.x /
70     scalePrev));
71
72     bool test = initialRun || (RUCorner.x >= iResolution.x / scalePrev ||
73     RUCorner.y >= iResolution.y / scalePrev) || (prevData[LDCIndex] !=
74     prevData[RUCIndex]);
75     if (test) {
76         res = first_layer(uv, iterations, u0, beta, V, theta);
77     } else {
78         res = prevData[LDCIndex];
79     }
80
81     data[index] = res;
82     if (imagep) {
83         color[index] = uint2col(res[0]) | (uint2col(res[1]) << 8) | uint(0
84     xFF000000);
85     }
86 }

```

Code 22: shaders/uintEfficient.glsl

```

1 uniform dvec2 iResolution;
2 uniform int iterations;
3 uniform dvec2 u0;
4 uniform double beta;
5 // W and b can't be used with this optimization

```

```

6 // uniform dvec2 b;
7 dvec2 b = dvec2(0.0, 0.0);
8 // uniform mat2 W;
9 mat2 W = mat2(1.0, 0.0, 0.0, 1.0);
10 uniform mat2 V;
11 uniform double theta;
12 uniform dvec2 offset;
13 uniform double scale;
14 uniform int scalePrev;
15
16 layout (local_size_x = 1, local_size_y = 1, local_size_z = 1) in;
17
18 layout(std430, binding = 0) buffer GridData {
19     uint[2] data[];
20 };
21 layout(std430, binding = 1) buffer GridColor {
22     uint color[];
23 };
24
25 // data from previous run with 1/scalePrev of size unless INITIAL_RUN is
    set
26 layout(std430, binding = 2) buffer GridPrevData {
27     uint[2] prevData[];
28 };
29
30 uint[2] first_layer(dvec2 x, int iterations, dvec2 u0, double beta, mat2 V,
    double theta)
31 {
32     dvec2 u = u0;
33     uvec2 s = uvec2(0,0);
34     uvec2 spiketr = uvec2(0,0);
35     for (int i = 0; i < iterations; ++i) {
36         u = beta * u + W * x + b + V * s - theta * beta * s;
37         s = uvec2(step(theta,u));
38         spiketr = (spiketr << 1) + s;
39     }
40     return uint[2](spiketr.x, spiketr.y);
41 }
42
43 uint uint2col(uint x)
44 {
45     float n = 10.;
46     float a = ((1. / n) * log(1 + (exp(n) - 1) * x / pow(2., float(
    iterations))));
47     // a should be in [0,1] by definition. But just to make sure:
48     return min(255, uint(floor(255*a)));
49 }
50
51 void main()
52 {
53     dvec2 nuv = gl_GlobalInvocationID.xy/iResolution.xy;
54     uint index = uint(gl_GlobalInvocationID.x + gl_GlobalInvocationID.y *
    iResolution.x);
55     dvec2 uv = scale*nuv+offset;
56
57     uint[2] res;
58     #ifdef INITIAL_RUN
59     res = first_layer(uv, iterations, u0, beta, V, theta);

```

```

60     #else
61     // points of corners
62     ivec2 LDCorner = ivec2(gl_GlobalInvocationID.x / scalePrev,
63     gl_GlobalInvocationID.y / scalePrev);
64     uint LDCIndex = uint(LDCorner.x + LDCorner.y * (iResolution.x /
65     scalePrev));
66     ivec2 RUCorner = ivec2(LDCorner.x + 1, LDCorner.y + 1);
67     uint RUCIndex = uint(RUCorner.x + RUCorner.y * (iResolution.x /
68     scalePrev));
69
70     bool test = (RUCorner.x >= iResolution.x / scalePrev || RUCorner.y >=
71     iResolution.y / scalePrev) || (prevData[LDCIndex] != prevData[RUCIndex])
72     ;
73
74     res = test ? first_layer(uv, iterations, u0, beta, V, theta) : prevData
75     [LDCIndex];
76     // if (RUCorner.x >= iResolution.x / scalePrev || RUCorner.y >=
77     iResolution.y / scalePrev) {
78     //     // out of bounds, compute new data
79     //     res = first_layer(uv, iterations, u0, beta, V, theta);
80     // } else if (prevData[LDCIndex] == prevData[RUCIndex]) {
81     //     // due to convexity and since the borders are on the axis
82     //     res = prevData[LDCIndex];
83     // } else {
84     //     res = first_layer(uv, iterations, u0, beta, V, theta);
85     // }
86     #endif
87
88     data[index] = res;
89     color[index] = uint2col(res[0]) | (uint2col(res[1]) << 8) | uint(0
90     xFF000000);
91 }

```

Code 23: shaders/uintEfficient.old.glsl

```

1  struct SNNConfig {
2      int iterations;
3      vec2 u0;
4      float beta;
5      mat2 W;
6      vec2 b;
7      mat2 V;
8      float theta;
9      vec2 offset;
10     float scale;
11 };
12
13 uint[4] first_layer(vec2 x, SNNConfig conf)
14 {
15     vec2 u = conf.u0;
16     uvec2 s = uvec2(0,0);
17     uvec2 spiketrLow = uvec2(0,0);
18     uvec2 spiketrUp = uvec2(0,0);
19     for (int i = 0; i < conf.iterations; ++i) {
20         u = conf.beta * u + conf.W * x + conf.b + conf.V * vec2(s) - conf.
21         theta * conf.beta * vec2(s);
22         s = uvec2(step(conf.theta,u));
23         spiketrUp = (spiketrUp << 1) + (spiketrLow >> 31);

```

```

23     spiketrLow = (spiketrLow << 1) + s;
24 }
25 return uint[4](spiketrLow.x, spiketrUp.x, spiketrLow.y, spiketrUp.y);
26 }
27
28 vec2 posToUV(vec2 pos, SNNConfig conf)
29 {
30     vec2 nuv = pos/iResolution.xy;
31     return conf.scale*nuv+conf.offset;
32 }
33
34 float uint2colScal(uint high, uint low, int iterations)
35 {
36     float n = 10.;
37     float x = float(high) * pow(2., 32.) + float(low);
38     float a = ((1. / n) * log(1. + (exp(n) - 1.) * x / pow(2., float(
iterations)))));
39     return a;
40 }
41
42 float uint2colGap(uint high, uint low, int iterations)
43 {
44     int n = 0;
45     int nMax = 0;
46     for (int i = 0; i < iterations; ++i) {
47         if ((low & 1u) == 1u) {
48             n++;
49             nMax = max(nMax, n);
50         } else {
51             n = 0;
52         }
53         low = (low >> 1) + (high << 31);
54         high = (high >> 1);
55     }
56     return (float(nMax) / float(iterations));
57 }
58
59 vec4 uint2col(uint[4] res, int iterations)
60 {
61     return vec4(uint2colGap(res[1], res[0], iterations),
62                 uint2colGap(res[3], res[2], iterations),
63                 0.0, 1.0);
64 }
65
66 vec4 uint2colBorder(vec2 fragCoord, uint[4] res, SNNConfig conf)
67 {
68     vec2 aBitRightPos = posToUV(fragCoord.xy+vec2(1,0), conf);
69     vec2 aBitUpPos     = posToUV(fragCoord.xy+vec2(0,1), conf);
70     uint[4] aBitRight = first_layer(aBitRightPos, conf);
71     uint[4] aBitUp    = first_layer(aBitUpPos, conf);
72     if (res != aBitRight || res != aBitUp) {
73         return vec4(1.0); // white
74     } else {
75         return uint2col(res, conf.iterations);
76     }
77 }
78
79 void mainImage(out vec4 fragColor, in vec2 fragCoord)

```

```

80 {
81     int iterations = 20;
82     vec2 u0 = vec2(0.0, 0.0);
83     float beta = 0.5;
84     vec2 b = vec2(0.0, 0.0);
85     mat2 W = mat2(1.0, 0.0, 0.0, 1.0);
86     mat2 V = mat2(1.0, 0.0, 0.0, 1.0);
87     float theta = 1.;
88     vec2 offset = vec2(0.0, 0.0);
89     float scale = 2.0;
90     SNNConfig conf = SNNConfig(iterations, u0, beta, W, b, V, theta, offset
    , scale);
91
92     vec2 uv = posToUV(fragCoord.xy, conf);
93     uint index = uint(fragCoord.x + fragCoord.y * iResolution.x);
94
95     uint[4] res = first_layer(uv, conf);
96
97     fragColor = uint2colBorder(fragCoord, res, conf);
98 }

```

Code 24: shaders/uintShaderToy.glsl

```

1  #version 460
2  uniform int prevBatchSize;
3
4  layout (local_size_x = 1, local_size_y = 1, local_size_z = 1) in;
5
6  layout(std430, binding = 0) buffer Data {
7      uint[4] data[];
8  };
9
10 layout(std430, binding = 1) buffer ResNum {
11     uint resNum[];
12 };
13
14 uint chunkSize = 4;
15
16 void main()
17 {
18     uint batchSize = 2*prevBatchSize;
19     uint batchIndex = uint(gl_GlobalInvocationID.x) * batchSize;
20
21     // we merge the batch of uniques at batchIndex with the one at
    batchIndex + prevBatchSize
22     uint readPointer = batchIndex + prevBatchSize;
23     while (true) {
24         uint writePointer = batchIndex;
25         while (true) {
26             // check if value already exists in lower batch
27             if (data[writePointer] == data[readPointer]) {
28                 break;
29             }
30             writePointer += 1;
31
32             // we have reached the end of the uniques of the lower batch
33             // (either because we have reached a marker (two equal values)
    or because we have reached the end of the batch)

```

```

34         if ((data[writePointer-1] == data[writePointer]) || (
writePointer - batchSize >= prevBatchSize)) {
35             data[writePointer] = data[readPointer];
36             // mark the end of the next unique values
37             data[writePointer+1] = data[readPointer];
38             break;
39         }
40     }
41
42     readPointer += 1;
43     // we have reached the end of the uniques of the upper batch
44     // (either because we have reached a marker (two equal values) or
because we have reached the end of the batch)
45     if ((data[readPointer-1] == data[readPointer]) || readPointer ==
batchSize + batchSize) {
46         break;
47     }
48 }
49
50 uint pointer = batchSize;
51 while (true) {
52     if ((data[pointer] == data[pointer+1])) {
53         resNum[0] = pointer - batchSize + 1;
54         break;
55     }
56     if (pointer - batchSize >= prevBatchSize) {
57         resNum[0] = pointer - batchSize;
58         break;
59     }
60
61     pointer += 1;
62 }
63 }

```

Code 25: shaders/uniques.glsl

```

1  #include <iostream>
2  #include <pybind11/pybind11.h>
3  #include <pybind11/stl.h>
4  #include <string>
5  #include <unordered_set>
6
7  namespace py = pybind11;
8
9  // Function that counts unique byte sequences (chunks) of length '
chunk_size'
10 // in a binary string provided as a py::bytes object.
11 size_t count_unique_chunks(py::bytes data, size_t chunk_size) {
12     // Convert the Python bytes object to a std::string.
13     std::string s = data;
14     size_t data_size = s.size();
15
16     if (chunk_size <= 0 || static_cast<size_t>(chunk_size) > data_size) {
17         throw std::invalid_argument("Invalid chunk size: " +
std::to_string(chunk_size));
18     }
19 }
20
21 if (data_size % chunk_size != 0) {

```

```

22         throw std::invalid_argument(
23             "Data size must be a multiple of chunk size");
24     }
25
26     std::unordered_set<std::string> unique;
27     for (size_t i = 0; i < data_size; i += chunk_size) {
28         unique.insert(s.substr(i, chunk_size));
29     }
30
31     return unique.size();
32 }
33
34 size_t _get_index_of(size_t size, int chunk_size,
35                     std::pair<size_t, size_t> point) {
36     return chunk_size * (point.first * size + point.second);
37 }
38
39 // We regard data as a grid of width and height size, where each node has a
40 // value of with bytes chunk_size.
41 // This function should always return the same value as count_unique_chunks
42 // This function should be much more efficient than count_unique_chunks,
43 // since it uses the fact that regions are cuboids.
44 namespace count_regions {
45 void _count_subregion(std::unordered_set<std::string> &unique,
46                      const std::string &s, std::pair<size_t, size_t>
47                      corner,
48                      size_t subSize, size_t size, size_t chunk_size) {
49     size_t startIndex = _get_index_of(size, chunk_size, corner);
50     if (subSize <= 2) {
51         for (size_t i = 0; i < subSize; ++i) {
52             for (size_t j = 0; j < subSize; ++j) {
53                 size_t index = _get_index_of(
54                     size, chunk_size, {corner.first + i, corner.second + j
55 });
56                 unique.insert(s.substr(index, chunk_size));
57             }
58         }
59         // Base case: add the single chunk at the corner to the set.
60         unique.insert(s.substr(startIndex, chunk_size));
61         return;
62     } else {
63         size_t endIndex = _get_index_of(
64             size, chunk_size,
65             {corner.first + subSize - 1, corner.second + subSize - 1});
66         if (s.substr(startIndex, chunk_size) ==
67             s.substr(endIndex, chunk_size)) {
68             unique.insert(s.substr(startIndex, chunk_size));
69         } else {
70             // Recursive case: divide the region into four quadrants.
71             size_t halfSize = subSize / 2;
72             _count_subregion(unique, s, {corner.first, corner.second},
73                             halfSize,
74                             size, chunk_size);
75             _count_subregion(unique, s,
76                             {corner.first, corner.second + halfSize},
77                             halfSize,
78                             size, chunk_size);

```

```

75         _count_subregion(unique, s,
76                             {corner.first + halfSize, corner.second},
77                             halfSize,
78                             size, chunk_size);
79         _count_subregion(
80             unique, s, {corner.first + halfSize, corner.second +
81             halfSize},
82             halfSize, size, chunk_size);
83     }
84 }
85 bool _is_power_of_two(int x) { return x > 0 && (x & (x - 1)) == 0; }
86
87 size_t count_regions(py::bytes data, size_t size, size_t chunk_size) {
88     std::string s = data;
89     size_t data_size = s.size();
90
91     if (chunk_size <= 0) {
92         throw std::invalid_argument("Invalid chunk size: " +
93                                     std::to_string(chunk_size));
94     }
95
96     if (size <= 0) {
97         throw std::invalid_argument("Invalid size: " +
98                                     std::to_string(chunk_size));
99     }
100
101     if (data_size != size * size * chunk_size) {
102         throw std::invalid_argument(
103             "Data byte count must be equal to size * size * chunk_size");
104     }
105
106     if (!_is_power_of_two(size)) {
107         throw std::invalid_argument("Size must be a power of two");
108     }
109
110     std::unordered_set<std::string> unique;
111     _count_subregion(unique, s, {0, 0}, size, size, chunk_size);
112     return unique.size();
113 }
114 } // namespace count_regions
115
116 // search last element with same value as start until end in s
117 // values are have size chunk_size
118 size_t search_steps(std::string &s, size_t chunk_size, size_t start, size_t
119                     end,
120                     size_t stepSize) {
121     size_t d = 0;
122     size_t n = 0;
123     while (true) {
124         size_t nextD = d + (1 << n);
125         size_t nextIndex = start + nextD * stepSize;
126         if ((nextIndex < end) &&
127             (std::memcmp(s.data() + nextIndex, s.data() + start, chunk_size
128 ) ==
129                 0)) {
130             d = nextD;

```

```

129         n++;
130     } else {
131         // if n == 0 we found the last node with same value as
132         // (i,j)
133         if (n == 0) {
134             break;
135         } else {
136             n--;
137         }
138     }
139 }
140 return d;
141 }
142
143 std::vector<py::bytes> unique_regions(py::bytes data, size_t size,
144                                     size_t chunk_size) {
145     std::string s = data;
146     size_t data_size = s.size();
147
148     if (chunk_size <= 0) {
149         throw std::invalid_argument("Invalid chunk size: " +
150                                     std::to_string(chunk_size));
151     }
152
153     if (size <= 0) {
154         throw std::invalid_argument("Invalid size: " +
155                                     std::to_string(chunk_size));
156     }
157
158     if (data_size != size * size * chunk_size) {
159         throw std::invalid_argument(
160             "Data byte count must be equal to size * size * chunk_size");
161     }
162
163     // valueMap captures if the value of a certain element in s
164     // has already been added to values.
165     // If that is the case, valueMap has the index of the value in values.
166     std::vector<int> valueMap(s.size() / chunk_size, -1);
167     std::vector<py::bytes> values{};
168
169     for (size_t i = 0; i < size; ++i) {
170         for (size_t j = 0; j < size; ++j) {
171             size_t indexVM = i * size + j;
172             // check if value has already been added to values
173             if (valueMap[indexVM] == -1) {
174                 // find the width dx and height dy of the region with the
175                 // same value
176                 size_t dx =
177                     1 + search_steps(s, chunk_size,
178                                     _get_index_of(size, chunk_size, {i, j
179
180                                     _get_index_of(size, chunk_size, {size,
181                                     j}),
182                                     size * chunk_size);
183
184                 size_t dy =
185                     1 + search_steps(s, chunk_size,
186                                     _get_index_of(size, chunk_size, {i, j
187                                     j}),
188                                     size * chunk_size);

```

```

184         _get_index_of(size, chunk_size, {i,
size})),
185         chunk_size);
186     values.push_back(s.substr(
187         _get_index_of(size, chunk_size, {i, j}), chunk_size));
188     int valueIndex = values.size();
189     for (size_t k = 0; k < dx; ++k) {
190         for (size_t l = 0; l < dy; ++l) {
191             valueMap[(i + k) * size + (j + l)] = valueIndex;
192         }
193     }
194 }
195 }
196 }
197
198 return values;
199 }
200
201 std::vector<py::bytes> llcorner_values(py::bytes data, size_t size,
202                                       size_t chunk_size) {
203     std::string s = data;
204     size_t data_size = s.size();
205
206     if (chunk_size <= 0) {
207         throw std::invalid_argument("Invalid chunk size: " +
208                                     std::to_string(chunk_size));
209     }
210
211     if (size <= 0) {
212         throw std::invalid_argument("Invalid size: " +
213                                     std::to_string(chunk_size));
214     }
215
216     if (data_size != size * size * chunk_size) {
217         throw std::invalid_argument(
218             "Data byte count must be equal to size * size * chunk_size");
219     }
220
221     std::vector<py::bytes> values{};
222
223     for (size_t i = 0; i < size; ++i) {
224         for (size_t j = 0; j < size; ++j) {
225             // Check if the value at the current position is different
226             // from the value to the left and below it.
227             bool leftIsDiff =
228                 i == 0 ||
229                 (std::memcmp(s.data() + _get_index_of(size, chunk_size, {i,
j})),
230                 s.data() +
231                 _get_index_of(size, chunk_size, {i - 1, j
chunk_size) != 0);
232             bool belowIsDiff =
233                 j == 0 ||
234                 (std::memcmp(s.data() + _get_index_of(size, chunk_size, {i,
j})),
235                 s.data() +
236                 _get_index_of(size, chunk_size, {i, j -

```

```

1}),
238                                     chunk_size) != 0);
239         // If it is different indeed, the current position is a ll
corner of
240         // a region.
241         if (leftIsDiff && belowIsDiff) {
242             values.push_back(s.substr(
243                 _get_index_of(size, chunk_size, {i, j}), chunk_size));
244         }
245     }
246 }
247
248 return values;
249 }
250
251 int llcorner_uniques(py::bytes data, size_t size, size_t chunk_size) {
252     std::string s = data;
253     size_t data_size = s.size();
254
255     if (chunk_size <= 0) {
256         throw std::invalid_argument("Invalid chunk size: " +
257                                     std::to_string(chunk_size));
258     }
259
260     if (size <= 0) {
261         throw std::invalid_argument("Invalid size: " +
262                                     std::to_string(chunk_size));
263     }
264
265     if (data_size != size * size * chunk_size) {
266         throw std::invalid_argument(
267             "Data byte count must be equal to size * size * chunk_size");
268     }
269
270     int valuesN = 0;
271
272     for (size_t i = 0; i < size; ++i) {
273         for (size_t j = 0; j < size; ++j) {
274             // Check if the value at the current position is different
275             // from the value to the left and below it.
276             bool leftIsDiff =
277                 i == 0 ||
278                 (std::memcmp(s.data() + _get_index_of(size, chunk_size, {i,
279 j}),
279                             s.data() +
280                             _get_index_of(size, chunk_size, {i - 1, j
281 chunk_size) != 0);
282             bool belowIsDiff =
283                 j == 0 ||
284                 (std::memcmp(s.data() + _get_index_of(size, chunk_size, {i,
285 j}),
286                             s.data() +
287                             _get_index_of(size, chunk_size, {i, j -
288 1}),
289                                     chunk_size) != 0);
288             // If it is different indeed, the current position is a ll
corner of

```

```

289         // a region.
290         if (leftIsDiff && belowIsDiff) {
291             valuesN++;
292         }
293     }
294 }
295
296 return valuesN;
297 }
298
299 PYBIND11_MODULE(unique_bytes, m) {
300     m.doc() = "Module for counting unique fixed-length byte sequences from
a "
301             "binary string";
302     m.def("count_unique_chunks", &count_unique_chunks, py::arg("data"),
303           py::arg("chunk_size"),
304           "Count unique chunks in the given bytes object with the specified
"
305           "chunk_size");
306     m.def("count_regions", &count_regions::count_regions, py::arg("data"),
307           py::arg("size"), py::arg("chunk_size"),
308           "Count unique regions in the given bytes object with the "
309           "specified size and chunk_size");
310     m.def("unique_regions", &unique_regions, py::arg("data"), py::arg("size
"),
311           py::arg("chunk_size"),
312           "Get unique regions in the given bytes object with the "
313           "specified size and chunk_size");
314     m.def("llcorner_values", &llcorner_values, py::arg("data"), py::arg("
size"),
315           py::arg("chunk_size"),
316           "Get the values of the lower-left corners of unique regions in
the "
317           "given bytes object with the specified size and chunk_size"
318           "Only works for cuboids regions.");
319     m.def("llcorner_uniques", &llcorner_uniques, py::arg("data"),
320           py::arg("size"), py::arg("chunk_size"),
321           "Count the number of unique lower-left corners in the given bytes
"
322           "object with the specified size and chunk_size"
323           "Only works for cuboids regions.");
324 }

```

Code 26: src/unique_bytes.cpp

7.3 dt-lif-snn-visualizer

See [Herrmann, 2025c] for the corresponding git repository.

```

1 import { baseScale } from "./Globals";
2 import { throttle } from "./Helpers";
3 import { Mat2D, Vec2D } from "./Types";
4
5 const PROXY_TAG = Symbol("isProxy");
6 const PROXY_FN_TAG = Symbol("prxxyFn");
7
8 const makeReactiveObj = <T extends object>(obj: T, onChange: () => void) =>
    {

```

```

9   const proxy = new Proxy(obj, {
10     set(target, prop, value) {
11       if (prop === PROXY_TAG || prop === PROXY_FN_TAG) {
12         (target as any)[prop as any] = value;
13         return true;
14       }
15       if (
16         typeof value === "object" &&
17         value !== null &&
18         (!(value as any)[PROXY_TAG] || (value as any)[PROXY_FN_TAG] !==
19           onChange)
20       ) {
21         value = makeReactiveObj(value, onChange);
22       }
23       (target as any)[prop as any] = value;
24       onChange();
25       return true;
26     },
27     deleteProperty(target, prop) {
28       delete (target as any)[prop as any];
29       onChange();
30       return true;
31     }
32   });
33
34   // Wrap initial nested objects
35   for (const key in obj) {
36     const value = (obj as any)[key];
37     if (
38       typeof value === "object" &&
39       value !== null &&
40       (!(value as any)[PROXY_TAG] || (value as any)[PROXY_FN_TAG] !==
41         onChange)
42     ) {
43       (obj as any)[key] = makeReactiveObj(value, onChange);
44     }
45   }
46
47   (proxy as any)[PROXY_TAG] = true;
48   (proxy as any)[PROXY_FN_TAG] = onChange;
49   return proxy;
50 };
51
52 export const backgroundColorAlgs = [
53   "countSpikes",
54   "spikeTrain",
55   "spikeTrainSimple",
56   "u",
57   "p",
58 ] as const;
59
60 export const borderTypes = [
61   "newBorder",
62   "differentCountPrev",
63   "differentCount",
64   "other",
65 ];

```

```

65 export const colorWithAlgs = [
66   "spikeTrain",
67   "spikeTrainPrev",
68   "spikeCount",
69   "spikeCountPrev",
70 ];
71
72 export type Conf = {
73   iterations: number;
74   u0: Vec2D;
75   i0: Vec2D;
76   alpha: number;
77   beta: number;
78   b: Vec2D;
79   W: Mat2D;
80   V: Mat2D;
81   theta: number;
82   offset: Vec2D;
83   scale: number;
84   showBorders: boolean;
85   colorWithAlgSpikes: [number, number];
86   colorWithAlg: number; // corresponds to index in `colorWithAlg`
87   showTooltip: boolean;
88   showBoundary: boolean;
89   tooltipPorU: "u" | "p";
90   backgroundColorAlg: number; // corresponds to index in `
    backgroundColorAlgs`
91   borderTypeColors: [number, number, number, number];
92   borderTypeOrder: [number, number, number, number]; // coresponds to
    indices in `borderTypes`
93 };
94
95 const saveConf = throttle(() => {
96   const params = new URLSearchParams();
97   for (const k in conf) {
98     params.set(k, JSON.stringify((conf as any)[k]));
99   }
100   window.history.replaceState(
101     {},
102     "",
103     `${window.location.pathname}?${params.toString()}`,
104   );
105 }, 500);
106
107 const loadConf = () => {
108   const params = new URLSearchParams(window.location.search);
109
110   for (const key in conf) {
111     if (params.has(key)) {
112       try {
113         (conf as any)[key] = JSON.parse(params.get(key)!);
114       } catch {
115         // fallback: treat as plain string if not valid JSON
116         (conf as any)[key] = params.get(key);
117       }
118     }
119   }
120 };

```

```

121
122 export const conf = makeReactiveObj<Conf>(
123   {
124     iterations: 10,
125     u0: [0, 0],
126     i0: [0, 0],
127     alpha: 0,
128     beta: 1,
129     b: [0, 0],
130     W: [1, 0, 0, 1],
131     V: [0, 0, 0, 0],
132     theta: 1,
133     offset: [0, 0],
134     scale: baseScale,
135     showBorders: true,
136     colorWithAlgSpikes: [-1, -1],
137     colorWithAlg: 0,
138     showTooltip: true,
139     showBoundary: false,
140     tooltipPorU: "u",
141     backgroundColorAlg: 1,
142     borderTypeColors: [0xffffffff, 0x0000ffff, 0xaaaaaaff, 0x080808ff],
143     borderTypeOrder: [0, 1, 2, 3],
144   },
145   saveConf,
146 );
147
148 window.addEventListener("load", loadConf);

```

Code 27: src/Conf.ts

```

1 export const baseScale: number = 3;
2
3 export const ParamSliderVals = {
4   a: 0,
5   b: 0,
6   c: 0,
7 };
8 export const frontConf = {
9   autoShowLeft: false,
10  autoShowRight: false,
11 };

```

Code 28: src/Globals.ts

```

1 import { Conf } from "../Conf";
2 import { Point2D, Vec2D } from "../Types";
3
4 export function posToUV(pos: Point2D, conf: Conf): Point2D {
5   const minSize = Math.min(window.innerWidth, window.innerHeight);
6   const uv: Point2D = [
7     conf.scale * (pos[0] / minSize - 0.5) + conf.offset[0],
8     conf.scale * (pos[1] / minSize - 0.5) + conf.offset[1],
9   ];
10  return uv;
11 }
12
13 export const clientCordConv = (

```

```

14   clientX: number,
15   clientY: number,
16   conf: Conf,
17 ) => {
18   return posToUV([clientX, window.innerHeight - clientY - 1], conf);
19 };
20
21 export function firstLayer(
22   x: Point2D,
23   conf: Conf,
24 ): [[number[], number[]], Vec2D, string[]] {
25   let i: [number, number] = [0, 0];
26   let p: [number, number] = conf.u0.slice() as Vec2D;
27   let u: [number, number] = p.slice() as Vec2D;
28   let uhistory: [number[], number[]] = [[u[0]], [u[1]]];
29   let s: [number, number] = [0, 0];
30   let spiketr: [string, string] = ["", ""];
31
32   const alpha = conf.alpha;
33   const beta = conf.beta;
34   const b = conf.b;
35   const theta = conf.theta;
36   const W = conf.W;
37   const V = conf.V;
38   const xWb = [W[0] * x[0] + W[2] * x[1], W[1] * x[0] + W[3] * x[1]];
39   const useU = conf.tooltipPorU == "u"; // fix value during loop
40   for (let l = 0; l < conf.iterations; l++) {
41     const prevS = s.slice() as Vec2D;
42     for (let j = 0; j < 2; j++) {
43       i[j] = alpha * i[j] + xWb[j] + (V[j] * prevS[0] + V[j + 2] * prevS
44         [1]);
45       p[j] = beta * u[j] + i[j] + b[j];
46       s[j] = p[j] >= theta ? 1 : 0;
47       u[j] = p[j] - theta * s[j];
48
49       spiketr[j] = spiketr[j] + (s[j] === 1 ? "1" : "0");
50     }
51     if (useU) {
52       uhistory[0].push(u[0]);
53       uhistory[1].push(u[1]);
54     } else {
55       uhistory[0].push(p[0]);
56       uhistory[1].push(p[1]);
57     }
58   }
59   return [uhistory, u, spiketr];
60 }
61
62 export function countChar(str: string, char: string): number {
63   let count = 0;
64   for (let i = 0; i < str.length; i++) {
65     if (str[i] === char) {
66       count++;
67     }
68   }
69   return count;
70 }

```

```

71
72 export function range1ToN(n: number): number[] {
73   return Array.from({ length: n }, (_, i) => i + 1);
74 }
75
76 export function throttle<T extends (...args: any[]) => void>(
77   fn: T,
78   limit: number,
79 ) {
80   let inThrottle: boolean = false;
81   let calledDuringThrottle: boolean = false;
82   return function (this: ThisParameterType<T>, ...args: Parameters<T>) {
83     if (!inThrottle) {
84       const run: () => void = () => {
85         fn.apply(this, args);
86         inThrottle = true;
87         setTimeout(() => {
88           if (calledDuringThrottle) {
89             calledDuringThrottle = false;
90             run();
91           } else inThrottle = false;
92         }, limit);
93       };
94       run();
95     } else {
96       calledDuringThrottle = true;
97     }
98   };
99 }

```

Code 29: src/Helpers.ts

```

1 export let rerender: { fn: null | (() => void) } = { fn: null };
2 let rerenderRequested = false;
3 export function requestRerender() {
4   if (rerender.fn && !rerenderRequested) {
5     rerenderRequested = true;
6     window.requestAnimationFrame(() => {
7       rerender.fn && rerender.fn();
8       rerenderRequested = false;
9     });
10  }
11 }

```

Code 30: src/Rerender.ts

```

1 // TODO: switch to three.js types
2 export type Vec2D = [number, number];
3
4 export type Point2D = Vec2D;
5
6 export type Mat2D = [number, number, number, number];

```

Code 31: src/Types.ts

```

1 declare module "*.glsl" {
2   const value: string;
3   export default value;

```

```
4 }
```

Code 32: src/custom.d.ts

```
1  /// <reference types="vite/client" />
2  declare module "*.glsl" {
3      const src: string;
4      export default src;
5  }
```

Code 33: vite-env.d.ts

```
1  import { defineConfig } from "vite";
2  import react from "@vitejs/plugin-react";
3  import checker from "vite-plugin-checker";
4  import glsl from "vite-plugin-glsl";
5
6  export default defineConfig({
7      plugins: [
8          glsl(),
9          react(),
10         checker({
11             typescript: true,
12         }),
13     ],
14 });
```

Code 34: vite.config.ts

```
1  import React, { useState } from "react";
2  import { Checkbox } from "@mui/material";
3  import { ParameterInput } from "../ParameterInput";
4  export function CheckboxInput({
5      paramName,
6      defaultValue,
7      onChange,
8  }: {
9      paramName: string;
10     defaultValue: boolean;
11     onChange: (checked: boolean) => void;
12 }) {
13     const [checked, setChecked] = useState(defaultValue);
14
15     const handlechange = (event: any) => {
16         setChecked(event.target.checked);
17         onChange(event.target.checked);
18     };
19
20     return (
21         <ParameterInput paramName={paramName}>
22             <Checkbox checked={checked} onChange={handlechange} />
23         </ParameterInput>
24     );
25 }
```

Code 35: src/CheckboxInput.tsx

```

1  import React, { useState } from "react";
2  import { Box, TextField, Typography } from "@mui/material";
3  import { requestRerender } from "../Rerender";
4
5  const uintToRGBA = (value: number) => {
6    const r = (value >> 24) & 0xff;
7    const g = (value >> 16) & 0xff;
8    const b = (value >> 8) & 0xff;
9    const a = value & 0xff;
10   return { r, g, b, a };
11 };
12
13 const RGBAtoUint = (r: number, g: number, b: number, a: number) => {
14   return (
15     ((r & 0xff) << 24) | ((g & 0xff) << 16) | ((b & 0xff) << 8) | (a & 0xff)
16   );
17 };
18
19 export function ColorInput({
20   defaultValue,
21   onChange,
22 }: {
23   defaultValue: number;
24   onChange: (value: number) => void;
25 }) {
26   const [value, setValue] = useState<{
27     r: number | null;
28     g: number | null;
29     b: number | null;
30     a: number | null;
31   }>(uintToRGBA(defaultValue));
32   const handleChange = (cp: "r" | "g" | "b") => (event: any) => {
33     const str = event.target.value;
34     if (str === "") {
35       setValue((v) => ({ ...v, [cp]: null }));
36       return;
37     }
38     const newValue = Number(str);
39     if (!isNaN(newValue)) {
40       setValue((v) => {
41         const updated = { ...v, [cp]: Math.max(0, Math.min(255, newValue))
42       };
43       onChange(
44         RGBAtoUint(updated.r ?? 0, updated.g ?? 0, updated.b ?? 0, 255),
45       );
46       requestRerender();
47       return updated;
48     } else {
49       setValue((v) => ({ ...v, [cp]: null }));
50       console.error("Invalid input:", str);
51     }
52   };
53
54   return (
55     <Box sx={{ display: "flex", gap: 1 }}>

```

```

56     <Typography sx={{ fontWeight: "bold" }}>#</Typography>
57     <TextField
58       size="small"
59       sx={{ width: 60 }}
60       value={value.r ?? ""}
61       onChange={handleChange("r")}
62     />
63     <TextField
64       size="small"
65       sx={{ width: 60 }}
66       value={value.g ?? ""}
67       onChange={handleChange("g")}
68     />
69     <TextField
70       size="small"
71       sx={{ width: 60 }}
72       value={value.b ?? ""}
73       onChange={handleChange("b")}
74     />
75   </Box>
76 );
77 }

```

Code 36: src/ColorInput.tsx

```

1  import React from "react";
2  import { Grid } from "@mui/material";
3  import { Mat2D } from "../Types";
4  import { ParameterInput } from "../ParameterInput";
5  import { NumberInput } from "../NumberInput";
6
7  export function Mat2DInput({
8    paramName,
9    defaultValue,
10   setValue,
11 }): {
12   paramName: string;
13   defaultValue: Mat2D;
14   setValue: (index: number, value: number) => void;
15 } {
16   return (
17     <ParameterInput paramName={paramName}>
18       <Grid
19         container
20         sx={{ px: 2 }}
21         rowSpacing={1}
22         columnSpacing={{ xs: 1, sm: 2, md: 3 }}
23       >
24         <Grid size={6}>
25           <NumberInput
26             defaultValue={defaultValue[0]}
27             setValue={(v) => setValue(0, v)}
28           />
29         </Grid>
30         <Grid size={6}>
31           <NumberInput
32             defaultValue={defaultValue[2]}
33             setValue={(v) => setValue(2, v)}

```

```

34     />
35   </Grid>
36   <Grid size={6}>
37     <NumberInput
38       defaultValue={defaultValue[1]}
39       setValue={(v) => setValue(1, v)}
40     />
41   </Grid>
42   <Grid size={6}>
43     <NumberInput
44       defaultValue={defaultValue[3]}
45       setValue={(v) => setValue(3, v)}
46     />
47   </Grid>
48 </Grid>
49 </ParameterInput>
50 );
51 }

```

Code 37: src/Mat2DInput.tsx

```

1  import React from "react";
2  import {
3    Box,
4    Drawer,
5    Grid,
6    IconButton,
7    Stack,
8    Typography,
9  } from "@mui/material";
10 import MenuIcon from "@mui/icons-material/Menu";
11 import { ParamSlider } from "../ParamSlider";
12 import { frontConf, ParamSliderVals } from "../Globals";
13 import { backgroundColorAlgs, borderTypes, colorWithAlgs, conf } from "../
    Conf";
14 import { NumberInputStandAlone } from "../NumberInputStandAlone";
15 import { Mat2DInput } from "../Mat2DInput";
16 import { Vec2DInput } from "../Vec2DInput";
17 import { requestRerender } from "../Rerender";
18 import { CheckboxInput } from "../CheckboxInput";
19 import { SelectInput } from "../SelectInput";
20 import { ColorInput } from "../ColorInput";
21
22 export function Menu() {
23   const [open, setOpen] = React.useState(false);
24   const drawerWidth = 400;
25
26   const handleDrawerFlip = () => {
27     setOpen((b) => !b);
28   };
29
30   return (
31     <Box>
32       <IconButton
33         color="inherit"
34         onClick={handleDrawerFlip}
35         edge="start"
36         sx={

```

```

37     {
38       position: "absolute",
39       top: 10,
40       mr: 2,
41       right: open ? drawerWidth : 0,
42       transition: "right 0.2s",
43       zIndex: 1201, // above Drawer zIndex (default is 1200)
44     },
45   ]}
46 >
47   <MenuIcon />
48 </IconButton>
49 <Drawer
50   sx={{
51     width: drawerWidth,
52     flexShrink: 0,
53     "& .MuiDrawer-paper": {
54       width: drawerWidth,
55       boxSizing: "border-box",
56     },
57   }}
58   variant="persistent"
59   anchor="right"
60   open={open}
61 >
62   <Typography variant="h5" sx={{ p: 2 }}>
63     Parameters
64   </Typography>
65   <ParamSlider
66     paramName="a"
67     defaultValue={ParamSliderVals.a}
68     min={0}
69     max={1}
70     step={0.01}
71     setValue={(x) => (ParamSliderVals.a = x)}
72   />
73   <ParamSlider
74     paramName="b"
75     defaultValue={ParamSliderVals.b}
76     min={0}
77     max={1}
78     step={0.01}
79     setValue={(x) => (ParamSliderVals.b = x)}
80   />
81   <ParamSlider
82     paramName="c"
83     defaultValue={ParamSliderVals.c}
84     min={0}
85     max={1}
86     step={0.01}
87     setValue={(x) => (ParamSliderVals.c = x)}
88   />
89   <NumberInputStandAlone
90     paramName="T"
91     defaultValue={conf.iterations}
92     setValue={(x) => (conf.iterations = Math.floor(x))}
93   />
94   <NumberInputStandAlone

```

```

95     paramName=" "
96     defaultValue={conf.alpha}
97     setValue={(x) => (conf.alpha = x)}
98   />
99   <NumberInputStandAlone
100     paramName=" "
101     defaultValue={conf.beta}
102     setValue={(x) => (conf.beta = x)}
103   />
104   <NumberInputStandAlone
105     paramName=" "
106     defaultValue={conf.theta}
107     setValue={(x) => (conf.theta = x)}
108   />
109   <Mat2DInput
110     paramName="W"
111     defaultValue={conf.W}
112     setValue={(i, v) => (conf.W[i] = v)}
113   />
114   <Vec2DInput
115     paramName="b"
116     defaultValue={conf.b}
117     setValue={(i, v) => (conf.b[i] = v)}
118   />
119   <Vec2DInput
120     paramName="u(0)"
121     defaultValue={conf.u0}
122     setValue={(i, v) => (conf.u0[i] = v)}
123   />
124   <Mat2DInput
125     paramName="V"
126     defaultValue={conf.V}
127     setValue={(i, v) => (conf.V[i] = v)}
128   />
129   <SelectInput
130     paramName="Background color algorithm"
131     defaultValue={backgroundColorAlgs[conf.backgroundColorAlg]}
132     values={backgroundColorAlgs.slice()}
133     onChange={(v) => {
134       conf.backgroundColorAlg = backgroundColorAlgs.indexOf(v as any)
135     }}
136   />
137   <CheckboxInput
138     paramName="Show borders"
139     defaultValue={conf.showBorders}
140     onChange={(c) => {
141       conf.showBorders = c;
142       requestRerender();
143     }}
144   />
145   <Stack sx={{ mb: 1, p: 1 }}>
146     <Typography variant="h6" sx={{ px: 2 }}>
147       Border config
148     </Typography>
149     <Grid container spacing={2}>
150       {[...conf.borderTypeColors.keys()].map((i) => {

```

```

152         return (
153             <>
154                 <Grid key={`$${i} 2`} size={5}>
155                     <Typography sx={{ px: 2 }}>{borderTypes[i]}: </
Typography>
156                 </Grid>
157                 <Grid key={`$${i} 3`} size={7}>
158                     <ColorInput
159                         defaultValue={conf.borderTypeColors[i]}
160                         onChange={(v) => (conf.borderTypeColors[i] = v)}
161                     />
162                 </Grid>
163             </>
164         );
165     })}
166 </Grid>
167 </Stack>
168 <CheckboxInput
169     paramName="Show tooltip"
170     defaultValue={conf.showTooltip}
171     onChange={(c) => {
172         conf.showTooltip = c;
173     }}
174 />
175 <CheckboxInput
176     paramName="Show boundary"
177     defaultValue={conf.showBoundary}
178     onChange={(c) => {
179         conf.showBoundary = c;
180         requestRerender();
181     }}
182 />
183 <CheckboxInput
184     paramName="Use post-spike potential for tooltip"
185     defaultValue={conf.tooltipPorU === "u"}
186     onChange={(c) => {
187         conf.tooltipPorU = c ? "u" : "p";
188         requestRerender();
189     }}
190 />
191 <CheckboxInput
192     paramName="Show same left"
193     defaultValue={frontConf.autoShowLeft}
194     onChange={(c) => {
195         if (!c && frontConf.autoShowLeft) {
196             conf.colorWithAlgSpikes[0] = -1;
197             requestRerender();
198         }
199         frontConf.autoShowLeft = c;
200     }}
201 />
202 <CheckboxInput
203     paramName="Show same right"
204     defaultValue={frontConf.autoShowRight}
205     onChange={(c) => {
206         if (!c && frontConf.autoShowRight) {
207             conf.colorWithAlgSpikes[1] = -1;
208             requestRerender();

```

```

209         }
210         frontConf.autoShowRight = c;
211     }}
212     />
213     <SelectInput
214         paramName="Color same by"
215         defaultValue={colorWithAlgs[conf.colorWithAlg]}
216         values={colorWithAlgs.slice()}
217         onChange={(v) => {
218             conf.colorWithAlg = colorWithAlgs.indexOf(v as any);
219             requestRerender();
220         }}
221     />
222 </Drawer>
223 </Box>
224 );
225 }

```

Code 38: src/Menu.tsx

```

1  import React from "react";
2  import { TextField } from "@mui/material";
3  import { ParamSliderVals } from "../Globals";
4  import { requestRerender } from "../Rerender";
5
6  export function NumberInput({
7      defaultValue,
8      setValue,
9  }: {
10     defaultValue: number;
11     step?: number;
12     setValue: (value: number) => void;
13 }) {
14     const [valueStr, setValueStr] = React.useState<string>(
15         defaultValue.toString(),
16     );
17     const [intervalId, setIntervalId] = React.useState<number | null>(null);
18     const handleInputChange = (event: any) => {
19         const str = event.target.value;
20         intervalId && clearInterval(intervalId);
21         setValueStr(str);
22         if (str === "") {
23             return;
24         }
25         const newValue = Number(str);
26         if (!isNaN(newValue)) {
27             setValue(newValue);
28         } else {
29             try {
30                 const f = eval(`(x,a,b,c,sin,cos,exp) => ${str}`);
31                 let currentValue: number = NaN;
32                 setIntervalId(
33                     setInterval(() => {
34                         try {
35                             const evaluatedValue = f(
36                                 Date.now() / 1000,
37                                 ParamSliderVals.a,
38                                 ParamSliderVals.b,

```

```

39         ParamSliderVals.c,
40         Math.sin,
41         Math.cos,
42         Math.exp,
43     );
44     if (isNaN(evaluatedValue)) {
45         throw new Error("Evaluated value is NaN");
46     }
47     if (currentValue !== evaluatedValue) {
48         setValue(evaluatedValue);
49         currentValue = evaluatedValue;
50         requestRerender();
51     }
52     } catch (e) {
53         intervalId && clearInterval(intervalId);
54         console.error("Error evaluating expression:", e);
55     }
56     }, 1000 / 20),
57 );
58 } catch (e) {
59     console.error("Invalid input:", str);
60 }
61 }
62 requestRerender();
63 };
64
65 return (
66     <TextField
67         value={valueStr}
68         onChange={handleInputChange}
69         variant="standard"
70     />
71 );
72 }

```

Code 39: src/NumberInput.tsx

```

1  import React from "react";
2  import { NumberInput } from "./NumberInput";
3  import { ParameterInput } from "./ParameterInput";
4  export function NumberInputStandAlone({
5      paramName,
6      defaultValue,
7      setValue,
8  }): {
9      paramName: string;
10     defaultValue: number;
11     setValue: (value: number) => void;
12 } {
13     return (
14         <ParameterInput paramName={paramName}>
15             <NumberInput defaultValue={defaultValue} setValue={setValue} />
16         </ParameterInput>
17     );
18 }

```

Code 40: src/NumberInputStandAlone.tsx

```

1 import React, { useCallback, useState } from "react";
2 import Slider from "@mui/material/Slider";
3 import { Typography } from "@mui/material";
4 import { ParameterInput } from "../ParameterInput";
5 import { requestRerender } from "../Rerender";
6
7 export function ParamSlider({
8   paramName,
9   defaultValue,
10  min,
11  max,
12  step = 0.01,
13  setValue,
14  transformValue = (v: number) => v,
15 }): {
16   paramName: string;
17   defaultValue: number;
18   min: number;
19   max: number;
20   step?: number;
21   setValue: (value: number) => void;
22   transformValue?: (v: number) => number;
23 } {
24   const [value, setValueIntern] = useState(defaultValue);
25   const handleChange = useCallback(
26     (_: Event, newValue: number) => {
27       const transformedValue = transformValue(newValue);
28       setValue(transformedValue);
29       setValueIntern(transformedValue);
30       requestRerender();
31     },
32     [setValue, setValueIntern, transformValue],
33   );
34   return (
35     <ParameterInput paramName={paramName}>
36       <Slider
37         size="small"
38         sx={{
39           px: 1,
40           py: 2,
41         }}
42         min={min}
43         max={max}
44         step={step}
45         defaultValue={defaultValue}
46         onChange={handleChange}
47       />
48       <Typography variant="h6" sx={{ flexGrow: 1, px: 2 }}>
49         {value.toFixed(2)}
50       </Typography>
51     </ParameterInput>
52   );
53 }

```

Code 41: src/ParamSlider.tsx

```

1 import React from "react";
2 import { Stack, Typography } from "@mui/material";

```

```

3
4 export function ParameterInput({
5   paramName,
6   children,
7 }: {
8   paramName: string;
9   children: React.ReactNode;
10 }) {
11   return (
12     <Stack
13       spacing={2}
14       direction="row"
15       sx={{ alignItems: "center", mb: 1, p: 1 }}
16     >
17       <Typography variant="h6" sx={{ px: 2 }}>
18         {paramName}
19       </Typography>
20       {children}
21     </Stack>
22   );
23 }

```

Code 42: src/ParameterInput.tsx

```

1 import React, { useState } from "react";
2 import { MenuItem, Select } from "@mui/material";
3 import { ParameterInput } from "../ParameterInput";
4 export function SelectInput({
5   paramName,
6   values,
7   defaultValue,
8   onChange,
9 }: {
10   paramName: string;
11   values: string[];
12   defaultValue: string;
13   onChange: (value: string) => void;
14 }) {
15   const [value, setValue] = useState(defaultValue);
16
17   const handleChange = (event: any) => {
18     setValue(event.target.value);
19     onChange(event.target.value);
20   };
21
22   return (
23     <ParameterInput paramName={paramName}>
24       <Select value={value} onChange={handleChange}>
25         {values.map((v) => (
26           <MenuItem key={v} value={v}>
27             {v}
28           </MenuItem>
29         ))}
30       </Select>
31     </ParameterInput>
32   );
33 }

```

Code 43: src/SelectInput.tsx

```

1 import { Point2D, Vec2D } from "../Types";
2 import { clientCordConv, countChar, firstLayer, range1ToN } from "../Helpers";
3 import { frontConf } from "../Globals";
4 import { conf } from "../Conf";
5 import { requestRerender } from "../Rerender";
6 import { Box, createTheme, ThemeProvider } from "@mui/material";
7 import { useEffect, useState } from "react";
8 import {
9   LineChart,
10  MarkElement,
11  MarkElementProps,
12 } from "@mui/x-charts/LineChart";
13 import { ChartsReferenceLine } from "@mui/x-charts/ChartsReferenceLine";
14
15 const darkTheme = createTheme({
16   palette: { mode: "dark" },
17 });
18
19 const CustomMark = (props: MarkElementProps) => (
20   <MarkElement {...props} classes={{ root: "customMarkElement" }} />
21 );
22
23 const uLineChart = (uhistory: [number[], number[]]) => {
24   return (
25     <LineChart
26       xAxis={
27         [
28           {
29             data: range1ToN(uhistory[0].length),
30             domainLimit: "strict",
31             tickMaxStep: 2,
32           },
33         ]
34       }
35       yAxis={
36         [
37           { tickMaxStep: 1 }
38         ]
39       }
40       series={
41         [
42           {
43             data: uhistory[0],
44             curve: "linear",
45             disableHighlight: true,
46           },
47           {
48             data: uhistory[1],
49             curve: "linear",
50             disableHighlight: true,
51           },
52         ]
53       }
54       slots={{ mark: CustomMark }}
55       sx={{
56         "& .customMarkElement": {
57           scale: "0.5", // Shrinks the dot to 50% of its original size
58         },
59       }}
60       height={70}
61       skipAnimation={true}
62       disableAxisListener={true}

```

```

55     disableLineItemHighlight={true}
56     margin={{ top: 5, right: 5, bottom: 0, left: -20 }}
57   >
58     <ChartsReferenceLine
59       y={0}
60       lineStyle={{ stroke: "gray", strokeWidth: 1 }}
61     />
62     <ChartsReferenceLine
63       y={1}
64       lineStyle={{ stroke: "gray", strokeWidth: 1 }}
65     />
66   </LineChart>
67 );
68 };
69
70 export const Tooltip = () => {
71   const [display, setDisplay] = useState(true);
72   const [pos, setPos] = useState<[number, number] | null>(null);
73   const [res, setRes] = useState<
74     [[number[], number[]], Vec2D, string[]] | null
75   >(null);
76
77   useEffect(() => {
78     window.addEventListener("mouseout", () => setDisplay(false));
79     window.addEventListener(
80       "mousemove",
81       (event) => {
82         if ((event.target as any).tagName.toLowerCase() === "canvas") {
83           setPos([event.clientX, event.clientY]);
84           setDisplay(true);
85           const uv: Point2D = clientCordConv(
86             event.clientX,
87             event.clientY,
88             conf,
89           );
90           window.requestAnimationFrame(() => {
91             const [uhistory, u, st] = firstLayer(uv, conf);
92             setRes([uhistory, u, st]);
93
94             if (frontConf.autoShowLeft) {
95               conf.colorWithAlgSpikes[0] = Number("0b" + st[0]);
96             }
97             if (frontConf.autoShowRight) {
98               conf.colorWithAlgSpikes[1] = Number("0b" + st[1]);
99             }
100             if (frontConf.autoShowLeft || frontConf.autoShowRight) {
101               requestRerender();
102             }
103           });
104         } else {
105         }
106       },
107       false,
108     );
109   }, []);
110
111   if (!display || !pos || !res || !conf.showTooltip) {
112     return <></>;

```

```

113 }
114
115 const [uhistory, u, st] = res;
116
117 const uv: Point2D = clientCordConv(pos[0], pos[1], conf);
118 return (
119   <Box
120     sx={{
121       position: "absolute",
122       background: "rgba(0,0,0,0.7)",
123       color: "white",
124       padding: "5px 10px",
125       pointerEvents: "none",
126       borderRadius: "5px",
127       left: pos[0] + 10 + "px",
128       top: pos[1] + 10 + "px",
129     }}
130   >
131     Pos: ({uv[0].toFixed(3)},{uv[1].toFixed(3)})
132     <br />
133     Potential: {u[0].toFixed(3)}:{u[1].toFixed(3)}
134     <ThemeProvider theme={darkTheme}>{uLineChart(uhistory)}</
135     ThemeProvider>
136     Spikes: {st[0]}:{st[1]}
137     <br />
138     Spike count: {countChar(st[0], "1")}:{countChar(st[1], "1")}
139   </Box>
140 );
};

```

Code 44: src/Tooltip.tsx

```

1 import React from "react";
2 import { Grid } from "@mui/material";
3 import { ParameterInput } from "../ParameterInput";
4 import { NumberInput } from "../NumberInput";
5 import { Vec2D } from "../Types";
6
7 export function Vec2DInput({
8   paramName,
9   defaultValue,
10   setValue,
11 }: {
12   paramName: string;
13   defaultValue: Vec2D;
14   setValue: (index: number, value: number) => void;
15 }) {
16   return (
17     <ParameterInput paramName={paramName}>
18       <Grid
19         container
20         sx={{ px: 2 }}
21         rowSpacing={1}
22         columnSpacing={{ xs: 1, sm: 2, md: 3 }}
23       >
24         <Grid size={12}>
25           <NumberInput
26             defaultValue={defaultValue[0]}

```

```

27         setValue={v => setValue(0, v)}
28       />
29     </Grid>
30     <Grid size={12}>
31       <NumberInput
32         defaultValue={defaultValue[1]}
33         setValue={v => setValue(1, v)}
34       />
35     </Grid>
36   </Grid>
37 </ParameterInput>
38 );
39 }

```

Code 45: src/Vec2DInput.tsx

```

1  import * as THREE from "three";
2  import fragmentShader from "../shaders/fragment.glsl";
3  import { baseScale } from "../Globals";
4  import { conf } from "../Conf";
5  import { requestRerender, rerender } from "../Rerender";
6  import { Point2D } from "../Types";
7  import { clientCordConv } from "../Helpers";
8  import ReactDOM from "react-dom/client";
9  import React from "react";
10 import { Menu } from "../Menu";
11 import { Tooltip } from "../Tooltip";
12
13 const init = async () => {
14   const scene = new THREE.Scene();
15   const camera = new THREE.Camera();
16   const renderer = new THREE.WebGLRenderer();
17   document.body.appendChild(renderer.domElement);
18   const canvas = renderer.domElement;
19   canvas.addEventListener("mousedown", (e) => e.preventDefault());
20   canvas.addEventListener("mousemove", (e) => e.preventDefault());
21   canvas.addEventListener("mouseup", (e) => e.preventDefault());
22   canvas.addEventListener("touchstart", (e) => e.preventDefault());
23   canvas.addEventListener("touchmove", (e) => e.preventDefault());
24   canvas.addEventListener("touchend", (e) => e.preventDefault());
25
26   // Fullscreen plane geometry
27   const geometry = new THREE.PlaneGeometry(2, 2);
28
29   // Simple fragment shader
30   const material = new THREE.ShaderMaterial({
31     fragmentShader: fragmentShader,
32     uniforms: {
33       iResolution: { value: [window.innerWidth, window.innerHeight] },
34       conf: { value: conf },
35     },
36   });
37
38   const mesh = new THREE.Mesh(geometry, material);
39   scene.add(mesh);
40
41   // const animate = (time) => {
42   //   // material.uniforms.uTime.value = time * 0.001;

```

```

43 //   requestAnimationFrame(animate);
44 // };
45
46 rerender.fn = () => {
47   renderer.setSize(window.innerWidth, window.innerHeight);
48   renderer.render(scene, camera);
49 };
50
51 window.addEventListener("resize", () => {
52   requestRerender();
53 });
54 rerender.fn();
55
56 // allow moving the plane
57 var anchorPoint: null | Point2D = null;
58 var anchorPointUV: null | Point2D = null;
59 window.addEventListener(
60   "mousedown",
61   (event) => {
62     if ((event.target as any).tagName.toLowerCase() === "canvas") {
63       anchorPoint = [event.clientX, event.clientY];
64       anchorPointUV = clientCordConv(event.clientX, event.clientY, conf);
65     }
66   },
67   false,
68 );
69 window.addEventListener("mousemove", (event) => {
70   if (anchorPoint !== null && anchorPointUV !== null) {
71     const eventPos = clientCordConv(event.clientX, event.clientY, conf);
72
73     conf.offset = [
74       conf.offset[0] + anchorPointUV[0] - eventPos[0],
75       conf.offset[1] + anchorPointUV[1] - eventPos[1],
76     ];
77     requestRerender();
78   }
79 });
80 window.addEventListener("mouseup", () => (anchorPoint = null), false);
81
82 // allow zooming in/out of the plane
83 let currentScale = 0;
84 window.addEventListener("wheel", (event) => {
85   if ((event.target as any).tagName.toLowerCase() === "canvas") {
86     event.preventDefault(); // Prevent default scroll
87
88     const delta = Math.sign(event.deltaY);
89     currentScale += delta === 1 ? 1 : -1;
90     conf.scale = baseScale * Math.pow(2, currentScale / 20);
91     requestRerender();
92   }
93 });
94
95 // animate();
96 ReactDOM.createRoot(document.getElementById("root")!).render(
97   <React.StrictMode>
98     <Tooltip />
99     <Menu />
100   </React.StrictMode>,

```

```

101     );
102 };
103
104 window.addEventListener("load", () => {
105     init();
106 });

```

Code 46: src/main.tsx

```

1  struct SNNConfig {
2      int iterations;
3      vec2 u0;
4      vec2 i0;
5      float alpha;
6      float beta;
7      mat2 W;
8      vec2 b;
9      mat2 V;
10     float theta;
11     vec2 offset;
12     float scale;
13     bool showBorders;
14     bool autoShowPrev;
15     bool showBoundary;
16     ivec2 colorWithAlgSpikes;
17     uint colorWithAlg;
18     uint backgroundColorAlg;
19     uint[4] borderTypeColors;
20     uint[4] borderTypeOrder;
21 };
22
23 uniform vec2 iResolution;
24 uniform SNNConfig conf;
25
26 struct ST
27 {
28     uint l0;
29     uint h0;
30     uint l1;
31     uint h1;
32 };
33
34 ST shiftRightST(ST st, int n)
35 {
36     ST stRest = st;
37     if (n <= 0) {
38         return stRest;
39     } else if (n <= 32) {
40         stRest.l0 = (stRest.l0 >> n) | (stRest.h0 << (32 - n));
41         stRest.h0 = (stRest.h0 >> n);
42         stRest.l1 = (stRest.l1 >> n) | (stRest.h1 << (32 - n));
43         stRest.h1 = (stRest.h1 >> n);
44     } else if (n <= 64) {
45         stRest.l0 = stRest.h0 >> (n - 32);
46         stRest.h0 = 0u;
47         stRest.l1 = stRest.h1 >> (n - 32);
48         stRest.h1 = 0u;
49     } else {

```

```

50     stRest = ST(0u, 0u, 0u, 0u);
51 }
52 return stRest;
53 }
54
55 vec4 unpackColor(uint c) {
56     return vec4(
57         float((c >> 24) & 0xFFu), // R
58         float((c >> 16) & 0xFFu), // G
59         float((c >> 8) & 0xFFu),  // B
60         float(c & 0xFFu)          // A
61     ) / 255.0; // normalize to [0,1]
62 }
63
64 struct LayerRes
65 {
66     ST spikeTrain;
67     vec2 uRes;
68 };
69
70
71 LayerRes first_layer(vec2 x, SNNConfig conf)
72 {
73     vec2 i = conf.i0;
74     vec2 p = vec2(0.0);
75     uvec2 s = uvec2(0,0);
76     vec2 u = conf.u0;
77     uvec2 spiketrLow = uvec2(0,0);
78     uvec2 spiketrHigh = uvec2(0,0);
79     for (int l = 0; l < conf.iterations; ++l) {
80         i = conf.alpha * i + conf.W * x + conf.V * vec2(s);
81         p = conf.beta * u + i + conf.b;
82         s = uvec2(step(conf.theta,p));
83         u = p-conf.theta*vec2(s);
84         spiketrHigh = (spiketrHigh << 1) + (spiketrLow >> 31);
85         spiketrLow = (spiketrLow << 1) + s;
86     }
87     return LayerRes(ST(spiketrLow.x, spiketrHigh.x, spiketrLow.y,
88         spiketrHigh.y), u);
89 }
90
91 vec2 posToUV(vec2 pos, SNNConfig conf)
92 {
93     float resMin = min(iResolution.x, iResolution.y);
94     vec2 nuv = pos/resMin- vec2(0.5);
95     return conf.scale*nuv+conf.offset;
96 }
97
98 uint bitCount(uint x) {
99     uint count = 0u;
100     for (int i = 0; i < 32; ++i) {
101         count += (x >> uint(i)) & uint(1);
102     }
103     return count;
104 }
105
106 uvec2 bitCountST(ST st) {
107     return uvec2(bitCount(st.l0)+ bitCount(st.h0), bitCount(st.l1)+

```

```

    bitCount(st.h1));
107 }
108
109 float uint2colScalSimple(uint high, uint low, int iterations)
110 {
111     float x = float(high) * pow(2., 32.) + float(low);
112     return x / pow(2., float(iterations));
113 }
114
115 float uint2colScal(uint high, uint low, int iterations)
116 {
117     float n = 10.;
118     float a = ((1. / n) * log(1. + (exp(n) - 1.) * uint2colScalSimple(high,
119     low, iterations)));
119     return a;
120 }
121
122 vec4 uint2colSimple(ST res, int iterations)
123 {
124     return vec4(uint2colScalSimple(res.h0, res.l0, iterations),
125     uint2colScalSimple(res.h1, res.l1, iterations),
126     0.0, 1.0);
127 }
128
129 vec4 uint2col(ST res, int iterations)
130 {
131     return vec4(uint2colScal(res.h0, res.l0, iterations),
132     uint2colScal(res.h1, res.l1, iterations),
133     0.0, 1.0);
134 }
135
136 bool[2] newestSpikes(ST res) {
137     return bool[2]((res.l0 & 1u) == 1u, (res.l1 & 1u) == 1u);
138 }
139
140 bool vec2smallerEQ(vec2 a, vec2 b) {
141     return (a.x <= b.x) && (a.y <= b.y);
142 }
143
144 bool bool2smallerEQ(bool[2] a, bool[2] b) {
145     return (uint(a[0]) <= uint(b[0])) && (uint(a[1]) <= uint(b[1]));
146 }
147
148 bool stSmallerEQ(ST a, ST b) {
149     return ((a.h0 < b.h0) || ((a.h0 == b.h0) && (a.l0 <= b.l0))) &&
150     ((a.h1 < b.h1) || ((a.h1 == b.h1) && (a.l1 <= b.l1)));
151 }
152
153 // majority of change is in [0,2]
154 float Rto01(float x) {
155     return atan((x-1.)*2.)*(1./3.15) + 0.5; // atan((x - 1)*2)/pi + 0.5
156 }
157
158 // majority of change is in [0,1]
159 float Rto01Norm(float x) {
160     return atan((x-.5)*3.)*(1./3.15) + 0.5; // atan((x - 0.5)*3)/pi + 0.5
161 }
162

```

```

163 vec4 colU(LayerRes res, float theta, float blue) {
164     return vec4(Rto01(res.uRes.x / theta) , Rto01(res.uRes.y / theta),
165               blue, 1.0);
166 }
167 vec4 colSpikeCount(LayerRes res, int iterations, float blue) {
168     return vec4(Rto01Norm(float(bitCount(res.spikeTrain.l0))/float(
169               iterations)),
170               Rto01Norm(float(bitCount(res.spikeTrain.l1))/float(
171               iterations)),
172               blue, 1.0);
173 }
174 vec4 colUNorm(LayerRes res, float theta, float blue) {
175     return vec4(Rto01Norm(res.uRes.x / theta) ,
176               Rto01Norm(res.uRes.y / theta) ,
177               blue,
178               1.0);
179 }
180 vec4 colPNorm(LayerRes res, float theta, float blue) {
181     return vec4(Rto01Norm((res.uRes.x + float(res.spikeTrain.l0 & 1u))
182               / theta) ,
183               Rto01Norm((res.uRes.y + float(res.spikeTrain.l1 & 1u))
184               / theta) ,
185               blue,
186               1.0);
187 }
188 uint firstDifferentBit(ST x, ST y) {
189     if (x == y) {
190         return ~0u;
191     }
192     for (uint i = 0u; i<32u; ++i) {
193         uint mask = 1u << i;
194         if (((x.l0 & mask) != (y.l0 & mask)) ||
195             ((x.l1 & mask) != (y.l1 & mask))) {
196             return i;
197         }
198     }
199     for (uint i = 0u; i<32u; ++i) {
200         uint mask = 1u << i;
201         if (((x.h0 & mask) != (y.h0 & mask)) ||
202             ((x.h1 & mask) != (y.h1 & mask))) {
203             return i+32u;
204         }
205     }
206     return ~0u; // IMPOSSIBLE
207 }
208 uint lastDifferentBit(ST x, ST y) {
209     if (x == y) {
210         return ~0u;
211     }
212     for (int i = 31; i>=0; --i) {
213         uint mask = 1u << uint(i);
214         if (((x.h0 & mask) != (y.h0 & mask)) ||
215             ((x.h1 & mask) != (y.h1 & mask))) {

```

```

216         return uint(i)+32u;
217     }
218 }
219 for (int i = 31; i>=0; --i) {
220     uint mask = 1u << uint(i);
221     if (((x.l0 & mask) != (y.l0 & mask)) ||
222         ((x.l1 & mask) != (y.l1 & mask))) {
223         return uint(i);
224     }
225 }
226 return ~0u; // IMPOSSIBLE
227 }
228
229
230 float geomSeries(float r, int n)
231 {
232     if (n < 0) {
233         return 0.0;
234     } else if (r == 1.0) {
235         return float(n+1);
236     } else {
237         return (1.0 - pow(r, float(n+1))) / (1.0 - r);
238     }
239 }
240
241 // demonominator of g
242 float doubleGeomSeries(float beta, float alpha, int n)
243 {
244     float sum = 0.0;
245     if (conf.beta == 0.0) {
246         sum += geomSeries(conf.alpha, n-1);
247     } else {
248         for (int k = 1; k <= n; ++k) {
249             sum += pow(beta, float(n-k)) * geomSeries(alpha, k-1);
250         }
251     }
252     return sum;
253 }
254
255 void mainImage(out vec4 fragColor, in vec2 fragCoord)
256 {
257     vec2 uv = posToUV(fragCoord.xy, conf);
258     uint index = uint(fragCoord.x + fragCoord.y * iResolution.x);
259
260     LayerRes res = first_layer(uv, conf);
261     ST st = res.spikeTrain;
262     bool isBorder = false;
263
264     if (conf.showBoundary) {
265         vec2 maxV = vec2(max(0.0, conf.V[1][0]), max(0.0, conf.V[0][1]));
266         mat2 minV = mat2(conf.V[0][0], min(conf.V[0][1], 0.0), min(conf.V
[1][0], 0.0), conf.V[1][1]);
267         float inf = 1.0 / 0.0;
268
269         vec2 lowerBound = vec2(inf);
270         vec2 inputOffset = vec2(0.0);
271         for (int t = 1; t <= conf.iterations; ++t) {
272             inputOffset = conf.beta * inputOffset + pow(conf.alpha, float(t

```

```

)) * conf.i0 + conf.b + (geomSeries(conf.alpha, t-2))*maxV;
273     lowerBound = min(lowerBound,
274                       - (inputOffset + pow(conf.beta, float(t))*
conf.u0-vec2(conf.theta))
275                       / doubleGeomSeries(conf.beta, conf.alpha, t))
;
276 }
277
278 vec2 upperBound = vec2(-inf);
279 inputOffset = vec2(0);
280 for (int t = 1; t <= conf.iterations; ++t) {
281     inputOffset = conf.beta * inputOffset + pow(conf.alpha, float(t
)) * conf.i0 + conf.b + (geomSeries(conf.alpha, t-2))*minV*vec2(1.0);
282     upperBound = max(upperBound,
283                     - (inputOffset + pow(conf.beta, float(t))*
conf.u0-conf.theta*geomSeries(conf.beta, t-1))
284                     / doubleGeomSeries(conf.beta, conf.alpha, t))
;
285 }
286
287 float eps = 1./min(iResolution.x, iResolution.y)*conf.scale;
288 if (any(lessThan(abs(uv - lowerBound), vec2(2.*eps))) || any(
289 lessThan(abs(uv - upperBound), vec2(2.*eps)))) {
290     fragColor = vec4(0.0, 0.0, 1.0, 1.0);
291     return;
292 }
293 }
294
295 if (conf.backgroundColorAlg == 0u) {
296     fragColor = colSpikeCount(res, conf.iterations, 0.0);
297 } else if (conf.backgroundColorAlg == 1u) {
298     fragColor = uint2col(st, conf.iterations);
299 } else if (conf.backgroundColorAlg == 2u) {
300     fragColor = uint2colSimple(st, conf.iterations);
301 } else if (conf.backgroundColorAlg == 3u) {
302     fragColor = colUNorm(res, conf.theta, 0.0);
303 } else if (conf.backgroundColorAlg == 4u) {
304     fragColor = colPNorm(res, conf.theta, 0.0);
305 }
306
307 if (conf.showBorders) {
308     // check values to right side and to upper side
309     vec2 aBitRightPos = posToUV(fragCoord.xy+vec2(1,0), conf);
310     vec2 aBitUpPos     = posToUV(fragCoord.xy+vec2(0,1), conf);
311     LayerRes aBitRightRes = first_layer(aBitRightPos, conf);
312     LayerRes aBitUpRes   = first_layer(aBitUpPos, conf);
313     ST aBitRightSt = aBitRightRes.spikeTrain;
314     ST aBitUpSt   = aBitUpRes.spikeTrain;
315
316     ST st = res.spikeTrain;
317     if (st != aBitRightSt || st != aBitUpSt) {
318         isBorder = true;
319         for (int i = 0; i < 4; ++i) {
320             if (conf.borderTypeOrder[i] == 0u) {
321                 uint lastDiffBit = min(lastDifferentBit(st, aBitRightSt
), lastDifferentBit(st, aBitUpSt));

```

```

323         if (lastDiffBit == 0u) {
324             fragColor = unpackColor(conf.borderTypeColors[0]);
325             break;
326         }
327     } else if (conf.borderTypeOrder[i] == 1u) {
328         uvec2 resCountPrev = bitCountST(shiftRightST(st, 1));
329         uvec2 aBitRightCountPrev = bitCountST(shiftRightST(
aBitRightSt, 1));
330         uvec2 aBitUpCountPrev = bitCountST(shiftRightST(
aBitUpSt, 1));
331         if (resCountPrev != aBitRightCountPrev || resCountPrev
!= aBitUpCountPrev) {
332             fragColor = unpackColor(conf.borderTypeColors[1]);
333             break;
334         }
335     } else if (conf.borderTypeOrder[i] == 2u) {
336         uvec2 resCount = bitCountST(st);
337         uvec2 aBitRightCount = bitCountST(aBitRightSt);
338         uvec2 aBitUpCount = bitCountST(aBitUpSt);
339
340         if (resCount != aBitRightCount || resCount !=
aBitUpCount) {
341             fragColor = unpackColor(conf.borderTypeColors[2]);
342             break;
343         }
344     } else if (conf.borderTypeOrder[i] == 3u) {
345         // default
346         fragColor = unpackColor(conf.borderTypeColors[3]);
347         break;
348     }
349 }
350 }
351 }
352
353 if (!conf.showBorders || !isBorder) {
354     uint shiftN = conf.autoShowPrev ? 1u : 0u;
355
356     if (conf.colorWithAlg == 0u) {
357         if ((st.h0==0u && (int(st.l0)) == conf.colorWithAlgSpikes[0])
||
358             (st.h1==0u && (int(st.l1)) == conf.colorWithAlgSpikes[1]))
359     {
360         fragColor.z = 1.;
361     }
362     } else if (conf.colorWithAlg == 1u) {
363         if ((st.h0==0u && ((st.l0 & 0x80000000u) != 1u) && int(st.l0 >>
1) == (conf.colorWithAlgSpikes[0] >> 1)) ||
364             (st.h1==0u && ((st.l1 & 0x80000000u) != 1u) && int(st.l1 >>
1) == (conf.colorWithAlgSpikes[1] >> 1))) {
365             fragColor.z = 1.;
366         }
367     } else if (conf.colorWithAlg == 2u) {
368         if ((conf.colorWithAlgSpikes[0] != -1 &&
369             (bitCount(st.l0) + bitCount(st.h0)) == bitCount(uint(conf.
colorWithAlgSpikes[0]))) ||
370             (conf.colorWithAlgSpikes[1] != -1 &&
371             (bitCount(st.l1) + bitCount(st.h1)) == bitCount(uint(conf.
colorWithAlgSpikes[1])))) {

```

```

371         fragColor.z = 1.;
372     }
373     } else if (conf.colorWithAlg == 3u) {
374         if ((conf.colorWithAlgSpikes[0] != -1 &&
375             (bitCount(st.l0 >> 1) + bitCount(st.h0)) == bitCount(uint(
376             conf.colorWithAlgSpikes[0]) >> 1)) ||
377             (conf.colorWithAlgSpikes[1] != -1 &&
378             (bitCount(st.l1 >> 1) + bitCount(st.h1)) == bitCount(uint(
379             conf.colorWithAlgSpikes[1]) >> 1))) {
380             fragColor.z = 1.;
381         }
382     }
383 }
384 void main()
385 {
386     vec2 fragCoord = gl_FragCoord.xy;
387     vec4 fragColor;
388     mainImage(fragColor, fragCoord);
389     gl_FragColor = fragColor;
390 }

```

Code 47: src/shaders/fragment.glsl