Ludwig-Maximilians-Universität at München
Department “Institut für Mathematik”
Chair of Artificial Intelligence and Machine Learning
Supervisor: Prof. Dr. Gitta Kutyniok

Bachelor’s Thesis

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

Valentin Herrmann - 12623552
plus+official@valentin-herrmann.de

Ich erkläre hiermit, dass ich die vorliegende Arbeit selbstständig angefertigt, alle Zitate als solche kenntlich gemacht, sowie alle benutzten Quellen und Hilfsmittel angegeben habe.

München, October 31, 2025 ....

Contents

1 Introduction

The capabilities of AI have improved significantly in recent years. Especially large language models have found their way into most people's lives. But even though LLMs already pass a simple Turing Test [Jones and Bergen, 2025], it is still unclear whether they are fundamentally capable of reasoning. While they are able to solve many problems using lots of compute, the process of finding a solution often involves far more trial and error than it does for a human [Collins et al., 2023] and reaches the correct result for the wrong reasons [Mondorf and Plank, 2024]. Human intuition on the other hand seems to require less attempts and can often lead to better solutions [Collins et al., 2023]. Another limitation are hallucinations that appear to be an intrinsic property of LLMs [Huang et al., 2025].

Furthermore, current models are extremely energy inefficient compared to the human brain. While the human brain operates only on around 20W [Kováč, 2009], the usage of LLMs consumes huge amounts of energy: The training of GPT3 consumed ca. 1GWh and inference of a short query using GPT-4o uses around 0.4Wh [Jegham et al., 2025].

While LLMs might eventually overcome some of those problems, especially the energy inefficiency seems to be an inherent property of the current training process. Thus it seems to be reasonable that other models might be better fitted to the task of reasoning. Many technological advances — like airplanes or sonars — have been inspired by the longest ongoing optimization progress in history, the evolution of life[1]. We therefore propose that neural networks more similar to the human brain might be smarter and more efficient. In this thesis we will investigate spiking neural networks with recursive connections.

While the idea behind spiking neural networks is quite old, they have not been researched as much since finding an efficient training algorithm seems to be harder in comparison. Therefore, a lot of open questions about these networks still remain. In this paper we will extend the work done in [Nguyen et al., 2025], by adding a decaying factor to the input of the neurons and allowing recursive connections between neurons of the same layer. This type of network will be called recurrent discrete time leaky-integrate-and-fire SNN, in short R-LIF-SNNs.

This thesis will roughly follow the structure of [Nguyen et al., 2025]: In Section 2 we motivate and formally introduce R-LIF-SNNs. In Section 3 we will show that a certain class of continuously differentiable functions can be approximated arbitrarily well by R-LIF-SNNs. While this has already been shown more generally for continuous functions in [Nguyen et al., 2025], we present a construction using far less neurons, using the internal linear structure of R-LIF-SNNs. After this, in Section 4, we analyze the output 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 present some promising experimental results in Section 5 and explain how exactly the algorithms are implemented and why they compute the correct result.

[1] https://xkcd.com/1605/

2 Definition of R-LIF-SNN

2.1 Notations and Conventions

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

We write $\{a,...,b\} := \{a, a+1, ..., b\}$ for the range of integers from a to b and in particular $[n] := \{1,...,n\}$ and $[n]_0 := \{0,...,n\}$ for the integers from 1 to n and from 0 to n respectively. Moreover $\vee$ is used to mean the logical "or" and $\wedge$ 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 $[\![x,y)\!)$ with $x,y \in \mathbb{R}^n$ to denote half-open cuboids $\prod_{i=1}^{n}[x_i, y_i)$. Similarly $[\![x,y]\!]$ 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 $(-\infty, x)$. Thus, $[\![x,y)\!) = \emptyset$ holds exactly when $y_i \leq x_i$ for a $i \in [n]$.

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

Continuing, $e_i := (0,...,1,...,0) \in \mathbb{R}^n$ is the i-th standard basis vector; $\mathbf{0}_n := (0,...,0) \in \mathbb{R}^n$ and $\mathbf{1}_n := (1,...,1) \in \mathbb{R}^n$ are the vector containing 0s and 1s in every component respectively and $I_n \in \mathbb{R}^{n\times n}$ is the identity matrix of size $n \times n$. Moreover, for a matrix $W \in \mathbb{R}^{n\times m}$ we 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 component (i,j). We add an ordering $\leq$ to vectors via the following: For all $x,y \in \mathbb{R}^n$, $x \leq y$ holds exactly if $\forall_{i\in[n]} x_i \leq y_i$. Furthermore, $x < y$ is canonically defined to hold if $x \leq y$ but not $x = y$.

If $f : U \to \mathbb{R}^n$ is a function with arbitrary domain U, then $f_i := \pi_i \circ f$ is the i-th component function, where $\pi_i : \mathbb{R}^n \to \mathbb{R}$ 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 from U to W will be written as $\mathcal{C}^0(U,W)$. Analogously, the set of k-times continuously differentiable functions will be written $\mathcal{C}^k(U,W)$. The total derivative of a differentiable function $f : U \to W$ is written as $df : U \to \operatorname{Hom}_{\mathbb{R}}(W,W)$.

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

Some additional notation regarding geometry: $\overline{A}$ is the closure of $A \subset \mathbb{R}^n$, A° is the interior; the open balls in $\mathbb{R}^n$ regarding p-Norm are notated by $B_{\varepsilon,p}(x) := \{y \mid \|y-x\|_p < \varepsilon\} \subset \mathbb{R}^n$ for $\varepsilon > 0$, $p \geq 1$ and $x \in \mathbb{R}^n$.

We 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, and similarly define $\min U$, $\inf U$ and $\sup U$. In addition we use the conventions $\inf(\emptyset) = \infty$ and $\sup(\emptyset) = -\infty$.

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

In [Gerstner et al., 2014] neurons are described as follows (Fig. 2.1):

[2]Source: By BruceBlaus - Own work, CC BY 3.0, `https://commons.wikimedia.org/w/index.php?curid=28761830`

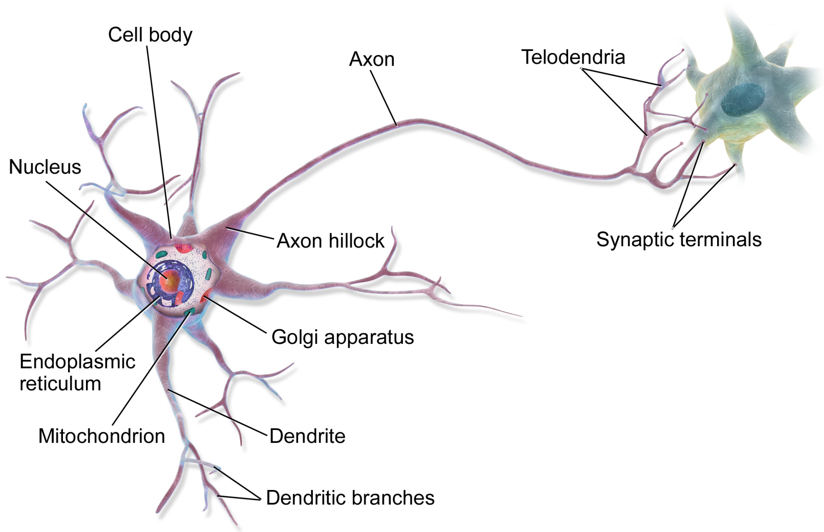


Figure 2.1: Structure of a neuron.[2]

> 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 book further describes that the neuron signals consist of short electrical pulses which we will call spikes. Since the pulses of a neuron look alike and since the pulses never overlap we can regard the output of a neuron as a list of zeros and ones, as a list of bits. We will call such lists spike trains.

Those spike trains update the cell-body by changing an internal variable, the membrane potential of the neuron. It is increased or decreased by incoming spikes depending on the neuron of origin. If the membrane potential reaches a threshold, the neuron will reset the potential to its resting potential and spike such that the subscribed neurons receive an update. If it does not reach the threshold, the potential just decays over time to the resting potential.

We will model the update of subscribed neurons with a decay factor: Instead of having the input to a neuron drop down to the resting state immediately after receiving a spike, the input will decay over time.

From these biologic observations one can derive the following system of differential equations for the input $I : \mathbb{R} \to \mathbb{R}$ and membrane potential $U : \mathbb{R} \to \mathbb{R}$ of a single neuron (c.f. [Gerstner et al., 2014]):

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

Here $s_i : \mathbb{R} \to \{0,1\}$ represents the j-th connection of the given neuron spiking at time t, the variables $w_i \in \mathbb{R}$ represent the weights of the connections to other neuron and $\vartheta s(t)$ resets the potential $U(t)$ after a spike. To avoid unresolvable dependencies between neurons, the connections need to have some latency $\Delta t \in \mathbb{R}$. The variables $\tau_\alpha, \tau_\beta > 0$ specify the rate with which I and U decay respectively. 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 are 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 have

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

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

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

which yields

$$I(t_{n+1}) = e^{-\tau_\alpha h} I(t_n) + \frac{1}{\tau_\alpha} (1 - e^{\tau_\alpha h}) \sum_{i=1}^{n} w_i s_i (t - \Delta t).$$

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

$$I(t_{n+1}) = \alpha I(t_n) + \sum_{i=1}^{n} w_i s_i (t - \Delta t) \tag{1}$$

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

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

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 $\alpha, \beta \in [0, 1]$ by construction.

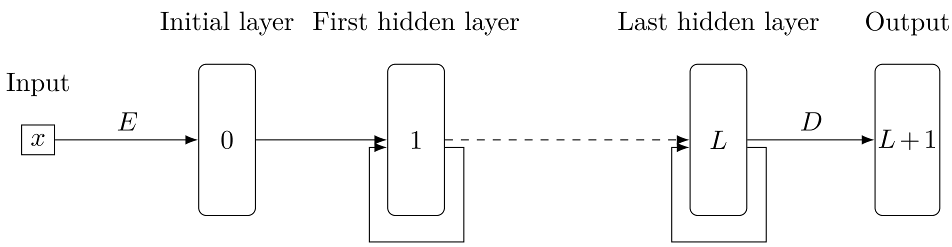


Figure 2.2: High-level network layout

We will now arrange those neurons into layers as 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 inside of layers we can always just merge all layers into the first one, but as we will see in Section 4, the layers allow us to analyze the networks more easily. 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 and decode our data to and from spikes trains. Like [Nguyen et al., 2025] we choose a simple direct encoding for the encoder E and membrane potential outputs for the decoder D.

2.3 Definitions

Our type of SNN should be thought of as a composition of an initial input layer realizing the encoder E, a number of hidden spiking layers with internal state and an affine-linear layer realizing the decoder D.

We first define the structure of the neurons of a hidden layer. Note that for any given neuron $i \in [n_l]$ in a layer l, the functions $i_i^{[l]}$ and $u_i^{[l]}$ correspond to I and U from the previous section.

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

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

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

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

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

with $s^{[l]}(0) = 0$ for all $l \in [l]$ 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 $\alpha^{[l]}, \beta^{[l]} \in [0,1]$ and
- threshold $\vartheta^{[l]} \in (0,\infty)$

where $H := \chi_{[0,\infty)}$ 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 $t \in [T]$ 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** $(n_0, \ldots, n_{L+1})$ 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** $E : \mathbb{R}^{n_0} \to \mathbb{R}^{n_0 \times T}$ 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** $D : \{0,1\}^{n_L \times T} \to \mathbb{R}^{n_{L+1}}$ maps the spike activations of the last hidden layer to values in $\mathbb{R}^{n_{L+1}}$.

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

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

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

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

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 R-LIF-SNN to mean "R-LIF-SNN with direct encoding and membrane potential".

Remark 2.3. Our definition of R-LIF-SNN breaks down to the one of d.t. LIF-SNN in [Nguyen et al., 2025] if we require $\alpha^{[l]} := 0$ and $V^{[l]} := \mathbf{0}_{n_l \times n_l}$ 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 $p^{[l]}(t) = i^{[l]}(t) = s^{[l-1]}(t)$ and therefore further

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

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

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

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

and therefore

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

Hence $s_i^{[l]}(t) = 1$ if and only if $s_i^{[l-1]}(t) = 1$ or $s_i^{[l]}(t-1) = 1$. 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.

Even though this construction is more natural defined using R-LIF-SNN, there is also a d.t. LIF-SNN construction achieving the same behavior using 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 $l \in [L]$.

By definition

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

and therefore

$$p^{[l]}(t) = p^{[l]}(t-1) + Ts^{[l-1]}(t) - s^{[l]}(t-1).$$

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

$$p^{[l]}(t) = \sum_{k=1}^{t} \left(Ts^{[l-1]}(k) - s^{[l]}(k-1)\right).$$

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

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

for $t' \ge t_0$ 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 easier construction of our networks we will additionally define allow building networks up from single networks using the following definition.

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 on the behavior of R-LIF-SNNs. 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 $\sigma^{[l]} \in \{0,1\}^{n_l \times (t-1)}$ 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 following terms are well-defined. We define

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

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

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

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

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' in $u^{[l]}$, in contrast to the definitions of $i^{[l]}, p^{[l]}, s^{[l]}$ the definition of $u^{[l]}$ requires the latest spikes from the current layer layer.

We will also allow using $i^{[l]}, p^{[l]}, s^{[l]}, u^{[l]}$ with extensions of the necessary spike trains. In particular, we will allow using the functions with "complete" spike trains $(\sigma^{[l]})_{l \in [L]}$ 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 even have $i^{[l]}(1;\sigma') = i^{[l]}(1)$, since σ' is an "extension" of σ.

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

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

and similarly

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

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

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

By using the above we obtain

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

$\square$

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

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*

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

Proof. We have

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

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 $i \in [n_l]$. Then $u_i^{[l]}(t) \geq 0 \Leftrightarrow p_i^{[l]}(t) \geq 0$.*

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

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 $\forall_{t \in \{t_0+1 \ldots t_\omega\}} i_i^{[l]}(t) + b_i^{[l]} \geq 0$ and $u_i^{[l]}(t_0) \geq 0$, then $u_i^{[l]}(t_\omega) \geq 0$.*

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

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

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 since $\vartheta > 0$ by definition. □

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 $\forall_{t \in \{t_0, \ldots, t_\omega\}} i_i^{[l]}(t) + b_i^{[l]} \leq \vartheta$ and $u_i^{[l]}(t_0 - 1) < \vartheta$, then*

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

Proof. We proof (12) by induction over $t \in \{t_0 - 1, \ldots, t_\omega\}$. The base case is given by assumption. Let further $t \in \{t_0 \ldots t_\omega\}$. 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 $s^{[l]}$. □

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 $\forall_{t \in \{t_0+1 \ldots t_\omega\}} i_i^{[l]}(t) + b_i^{[l]} \leq 0$ and $u_i^{[l]}(t_\omega) \geq \vartheta$, then $\forall_{t \in \{t_0 \ldots t_\omega\}} u_i^{[l]}(t) \geq \vartheta$.*

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

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

□

Lemma 2.8. *Let $t_0, t_\omega \in [T]$ and $i \in [n_l]$ for an $l \in [L]$ such that $t_0 \leq t_\omega$. Then $u_i^{[l]}(t_\omega) \geq 0$ if both*

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

and $\forall_{t \in \{t_0+1,\dots,t_\omega\}} s_i^{[l]}(t) = 0$ hold 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 $s_i^{[l]}(t') = 1$, .

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

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

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 $p_i^{[l]}(t') \geq \vartheta > 0$, so by Lemma 2.4 $u_i^{[l]}(t') \geq 0$ such that we can conclude by Lemma 2.5. □

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*

$$0 \leq u_i^{[l]}(t_0 - 1) + \sum_{t=t_0}^{t_\omega} \left(i_i^{[l]}(t) + b_i^{[l]} \right) \tag{14}$$

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

$$\sum_{t=t_0}^{t_\omega} s_i^{[l]}(t) = \left\lfloor \frac{1}{\vartheta} \left(u_i^{[l]}(t_0 - 1) + \sum_{t=t_0}^{t_m} \left(i_i^{[l]}(t) + b_i^{[l]} \right) \right) \right\rfloor. \tag{15}$$

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. □

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*

$$0 \leq u_i^{[l]}(t_0 - 1) + \sum_{t=t_0}^{t_m} \left(i_i^{[l]}(t) + b_i^{[l]} \right) \leq \vartheta(t_\omega - t_m + 1), \tag{16}$$

then

$$\sum_{t=t_0}^{t_\omega} s_i^{[l]}(t) = \left\lfloor \frac{1}{\vartheta} \left(u_i^{[l]}(t_0 - 1) + \sum_{t=t_0}^{t_m} \left(i_i^{[l]}(t) + b_i^{[l]} \right) \right) \right\rfloor. \tag{17}$$

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 have $\forall_{t \in \{t_m \dots t_\omega\}} u_i^{[l]}(t) \geq \vartheta$ by Lemma 2.7 and in particular $\forall_{t \in \{t_m \dots t_\omega\}} s_i^{[l]}(t) =$

1. This then yields

$$
\begin{aligned}
u_i^{[l]}(t_\omega) &= u_i^{[l]}(t_0-1) + \sum_{k=t_0}^{t_\omega} \left(i^{[l]}(k) + b^{[l]} - \vartheta s^{[l]}(k)\right) \\
&= u_i^{[l]}(t_0-1) + \sum_{k=t_0}^{t_m} \left(i^{[l]}(k) + b^{[l]}\right) - \vartheta \sum_{k=t_0}^{t_\omega} s^{[l]}(k) \\
&\leq \vartheta(t_\omega - t_m + 1) - \vartheta(t_\omega - t_m + 1) - \vartheta \sum_{k=t_0}^{t_m-1} s^{[l]}(k) \\
&\leq 0,
\end{aligned}
$$

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

3 Structure of computations in R-LIF-SNNs

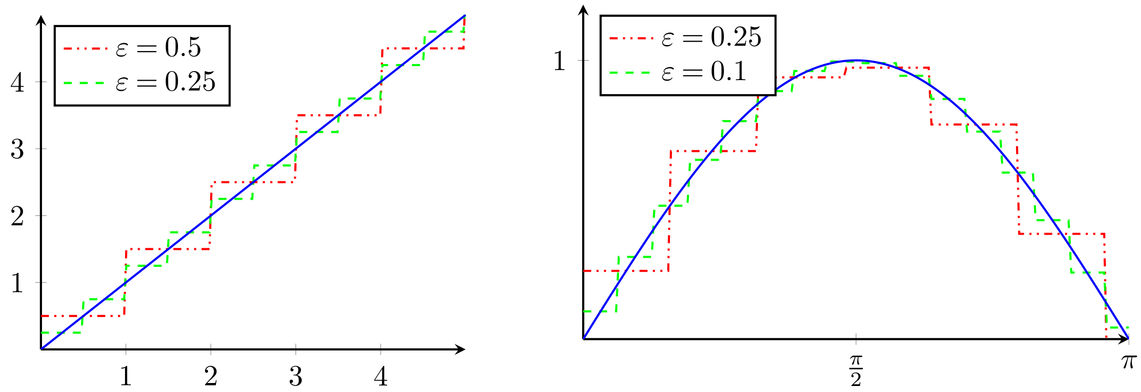
When working with neural networks a fundamental question is how well they are able to approximate functions. Towards that end the following theorem was proved in [Nguyen et al., 2025].

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

$$n_1 = \left(\max\left\{\left\lceil \frac{\operatorname{diam}_{\infty}(\Omega)}{\varepsilon}\Gamma \right\rceil, 1\right\} + 1\right) n_0,$$
$$n_2 = \max\left\{\left\lceil \frac{\operatorname{diam}_{\infty}(\Omega)}{\varepsilon}\Gamma \right\rceil^{n_0}, 1\right\}.$$



(a) D.t. LIF-SNNs approximating the identity **(b)** D.t. LIF-SNNs approximating a sinus wave

Figure 3.1: D.t. LIF-SNNs approximating functions

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 fitting 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 show a more efficient construction for R-LIF-SNN using the fact that R-LIF-SNN can efficiently approximate linear regions. The general intuition is to use piece-wise linear functions to approximate continuously differentiable functions and to 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 region.

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 prove some simple properties:

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

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

The generalized inverse of ω is defined as

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

Lemma 3.1. *Let $\omega : [0,\infty] \to [0,\infty]$ be a modulus of uniform continuity of a uniformly continuous function $f : M \to N$, 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. $\forall_{s\in[0,\infty]} w^\dagger(s) = 0 \Rightarrow s = 0$.

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

□

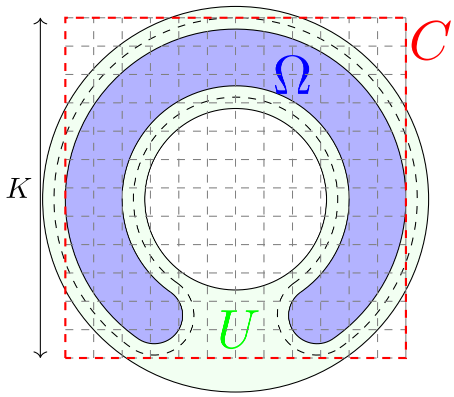


Figure 3.2: A possible configuration of the sets in Theorem 3.2

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 $C \subset \mathbb{R}^n$ a half-open cube such that $\Omega \subset C$ and $(\Omega + B_{\rho,2}(0)) \cap C^\circ \subset U^\circ$ for a $\rho > 0$.*

Then for all $\varepsilon, \mu, \nu > 0$ with $\varepsilon = \mu + \nu$ 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

$$T_r := T_r(\nu) := \max\left(2, \left\lceil \sqrt{n}\frac{\operatorname{diam}_\infty(C)}{K}\frac{\|d(f|_{U^\circ})\|_{\infty,2}}{\nu}\right\rceil\right) \in \mathbb{N},$$
$$K := K(\mu) := \min_{\substack{\xi,\theta>0\\ \xi\theta=\mu}}\left\{\left\lceil \frac{\operatorname{diam}_\infty(C)}{\frac{2}{\sqrt{n}}\min(\omega^\dagger(\xi),\theta,\frac{\rho}{2})}\right\rceil\right\} \in \mathbb{N}.$$

Here $\omega^\dagger$ is the generalized inverse of a modulus of uniform continuity of the total derivative $d(f|_{U^\circ})$ with regard to $\|\cdot\|_2$. The differential $d(f|_{U^\circ})$ is uniformly continuous because it is bounded. Due to $\xi \neq 0$, we have $\omega^\dagger(\xi) > 0$ by Lemma 3.1. Furthermore, K is well-defined, since there are ξ, θ such that the minimum in the definition of K^n is obtained, because the minimum is taken over the set of natural numbers. Notice that further $K \neq 0$, because $\min(\omega^\dagger(\xi),\theta,\frac{\rho}{2}) < \infty$ as well as $\operatorname{diam}_\infty(C) \neq 0$, since $\Omega \subset C$ and $\Omega \neq \emptyset$. Hence $T_r \in \mathbb{N}$ is well-defined.

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

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

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

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

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

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

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

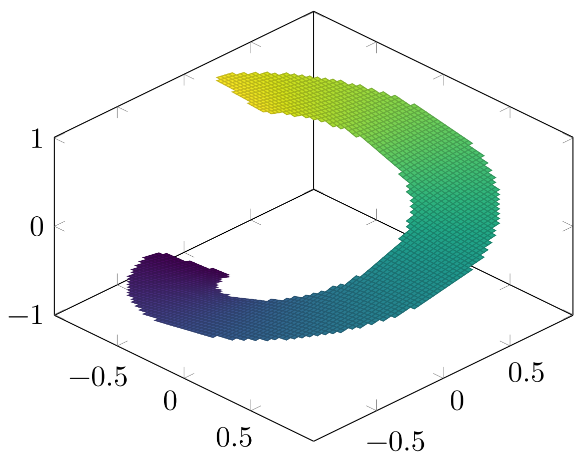


Figure 3.3: A spiral staircase function

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 avoided by extending f to all of $\mathbb{R}^n$. However, this might introduce a very steep gradient around $(-0.5, 0.0)$, such that $\|d(f|_{C^\circ})\|_{\infty,2}$ is very large and changes unnecessarily quick, making T_r and K excessively large.

On the other hand, Theorem 3.2 only uses the values of df on a thin region around Ω to determine the network size.

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 is 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 $\omega^\dagger(\xi)$. Furthermore, $\omega^\dagger(\xi)$ is a measure of how strongly the slope of f is changing and thus into how many subcubes we need to split C such that f is sufficiently almost linear on each subcube.

We will moreover 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 $\frac{\operatorname{diam}_\infty(C)}{K}$ of the subcubes and the maximal slope $\|d(f|_{U^\circ})\|_{\infty,2}$.

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

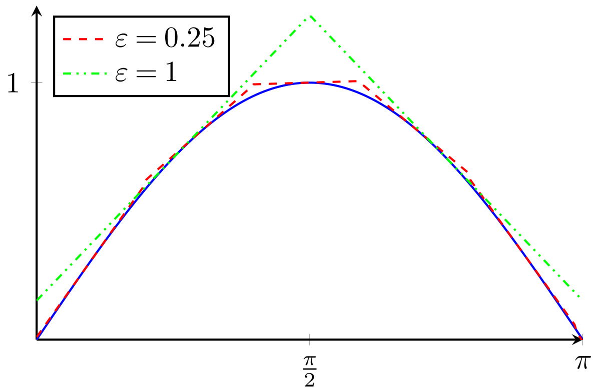


Figure 3.4: Piece-wise affine linear functions approximating the sinus.

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 uniformly continuous differential $d(f|_{U^\circ})$. Let further $\emptyset \neq \Omega \subset U$ be an arbitrary non-empty subset and $C \subset \mathbb{R}^n$ a half-open cube such that $\Omega \subset C$ and $(\Omega + B_{\rho,2}(0)) \cap C^\circ \subset U^\circ$ for a $\rho > 0$. Furthermore, let K be defined as in Theorem 3.2.*

For every $\mu > 0$ we can then compose C into K^n half-open subcubes $(C^{(j)})_{j\in[K^n]}$ such that there exist affine linear functions $g^{(j)} : C^{(j)} \to \mathbb{R}^m$ with $\|d(g^{(j)})\|_{\infty,2} \leq \|d(f|_{U^\circ})\|_{\infty,2}$ and $\|(f-g)|_\Omega\|_{\infty,2} \leq \mu$ for $g := \sum_{i=1}^m g^{(j)}\chi_{C^{(j)}}$.

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

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

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

$$\left\|x - c^{(j)}\right\|_2 = \sqrt{\sum_{i=1}^{n} \left|(x - c^{(j)})_i\right|^2} \leq \frac{w}{2}\sqrt{n} \leq \min(\omega^\dagger(\xi), \theta, \frac{\rho}{2}). \tag{18}$$

Let us further define $g^{(j)} : C^{(j)} \to \mathbb{R}^m$ 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 (18), 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 $\left\|d(g^{(j)})\right\|_{\infty,2} = \|df_{c^{(j)}}\| \leq \|d(f|_{U^\circ})\|_{\infty,2}$ for $C^{(j)} \cap \Omega \neq \emptyset$ and otherwise $0 = \left\|d(g^{(j)})\right\|_{\infty,2} \leq \|d(f|_{U^\circ})\|_{\infty,2}$.

It only remains to show $\left\|(f - g^{(j)})|_{\Omega \cap C^{(j)}}\right\|_\infty < \mu$. 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

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

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

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

In the fourth step we use $\left\|df_{(x-c^{(j)})t+c^{(j)}} - df_{c^{(j)}}\right\| \leq \xi$, which holds due to $\forall_{t \in [0,1]} (x - c^{(j)})t + c^{(j)} \in C^{(j)}$, (18) and Lemma 3.1. □

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 $(C^{(j)})_{j \in [K^n]}$ and linear functions $g^{(j)} : C^{(j)} \to \mathbb{R}^m$, such that $\left\|d(g^{(j)})\right\|_{\infty,2} \leq \|df\|_{\infty,2}$ and $\|(f-g)|_\Omega\|_\infty < \mu$ for $g := \sum_{i=1}^m g^{(j)} \chi_{C^{(j)}}$.

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.

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

$$T_1 := \{1, ..., KT_r\}, \qquad T_2 := \{KT_r\}, \qquad T_3 := \{KT_r + 1\},$$
$$T_4 := \{KT_r + 2\}, \qquad T_5 := \{KT_r + 3, ..., T\}.$$

Note that T_2 overlaps with T_1. For ease of notation, we will also use T_2, T_3, T_4 as numbers for their respective element, e.g. T_2 for KT_r.

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.

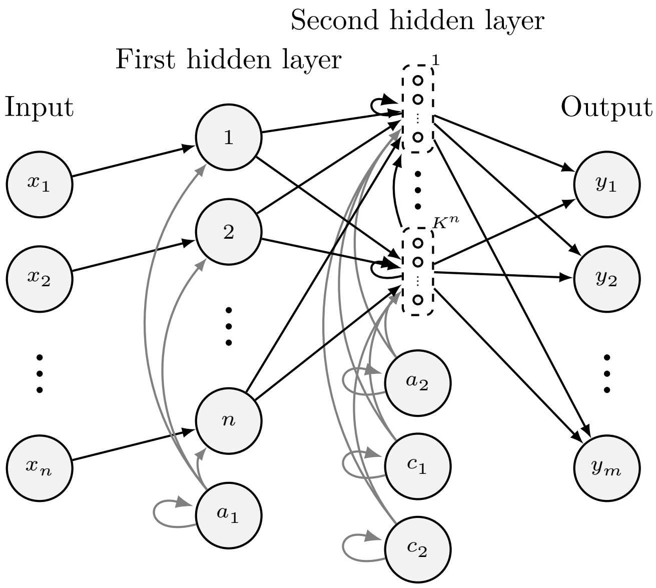


Figure 3.5: Structure of the whole network

The higher level idea is that we accumulate the state during T_1, compute the subregion $C^{(j)}$ of the input during $T_2, ..., T_4$ 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 neuron 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.

During $T_1 \setminus T_2$ the second layer only accumulates state without spiking. Then during $T_2 \cup T_3 \cup T_4$ the network decides in which region $C^{(j)}$ the input x is located; during $T_4 \cup T_5$ the location of x in $C^{(j)}$ as well as x being located in $C^{(j)}$ is encoded as spike trains.

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 is at least as big as the base point of $C^{(j)}$. The $n+1$-th neuron deactivates all other neurons of regions with smaller base point and encodes the constant part of $g^{(j)}$. Apart from the neuron groups, the last 3 neurons in the second layer act as "clock neurons" enabling and disabling the other ones.

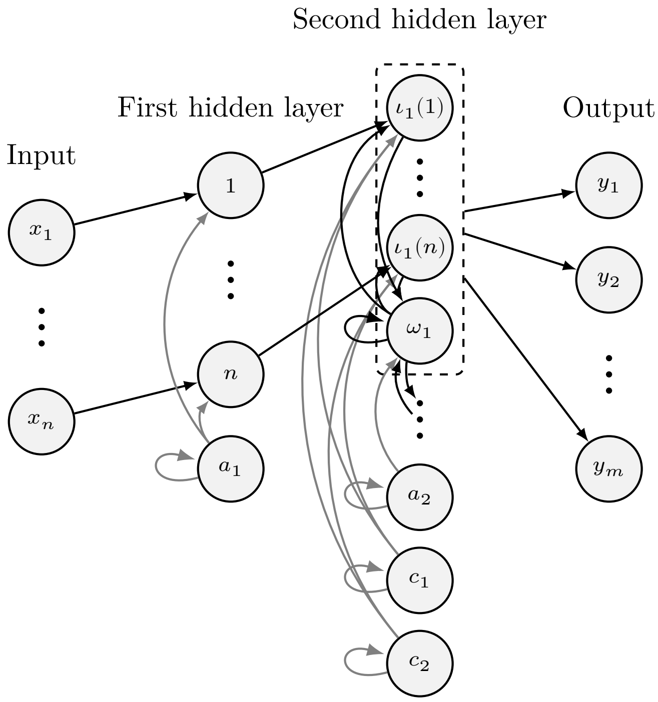


Figure 3.6: Structure of the network, focused on the j-th group of the second layer

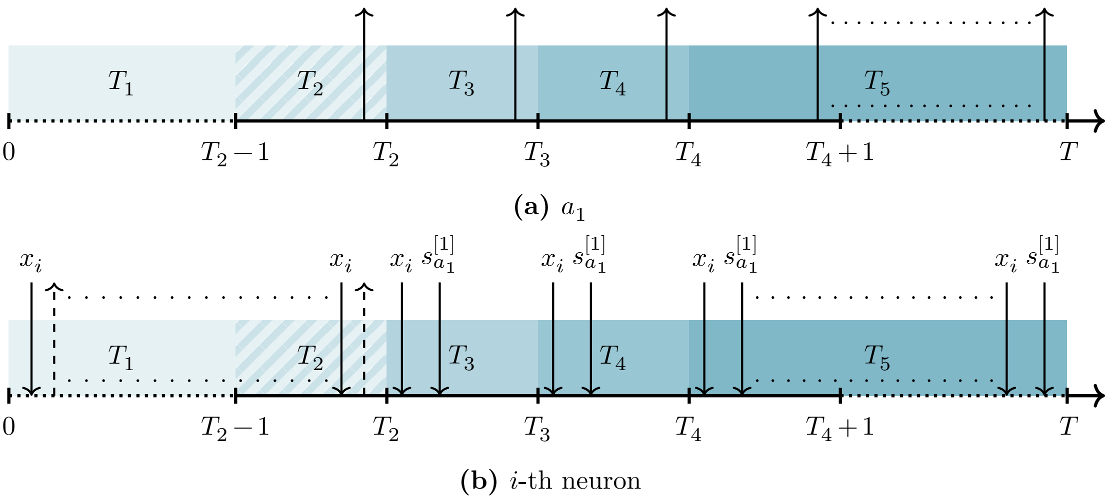


Figure 3.7: Timelines of first layer neurons. Upward arrows depict outgoing spikes, downward arrows depict ingoing spikes of the given neuron. Solid arrows depict certainly occurring spikes, dashed arrows depict possibly occurring spikes.

To obtain the normalized location of a value in C we will often write $o_i(z)$ for

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

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

Consider now the following rigorous construction.

1. **First layer**: We define the ***i*-th neuron** of the n neurons of the first layer with parameters using the notation from Definition 2.5:

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

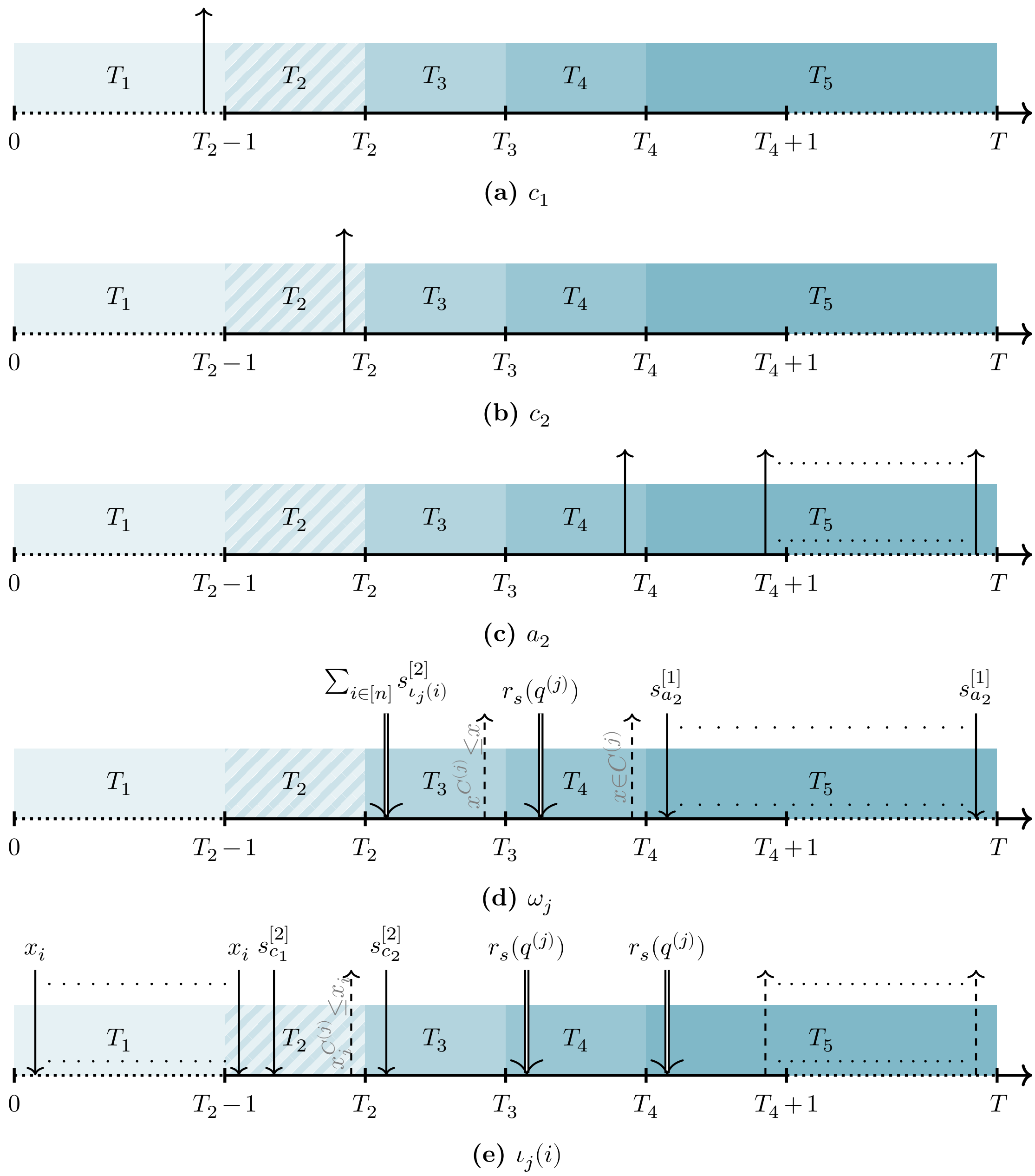


Figure 3.8: Timelines of second layer neurons. A few possible outgoing spikes are labeled with their respective condition in gray. Thick arrows depict possible input spikes from multiple neurons.

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

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

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 $C^{(j)} = [\![x^{C^{(j)}}, y^{C^{(j)}})\!)$ be one such subcube with position $q^{(j)} \in ([K-1]_0)^n$ in C, i.e.

$$q_i^{(j)} = Ko_i(x^{C^{(j)}}), \quad q_i^{(j)} + 1 = Ko_i(y^{C^{(j)}}) \quad \text{for all } i \in [n].$$

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

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

is "the switch" and $j(q)$ is the index of the subcube at position q. We further define the applied variant

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

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

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

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

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

where $b_{c_1} = \frac{1}{T_2 - 1}$ and $b_{c_2} = \frac{1}{T_2}$. Finally, the **"alarm neuron"**, with index $a_2 := (j+1)K^n(\mu) + 3$, is defined by

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

3. Output decoder: We 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

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

for $k \in [m]$, $j \in K^n$ and $i \in [n]$. Note that $d(g^{(j)})$ is in particular 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 moreover set

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

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

We will now prove that this construction indeed approximates g up to a precision of ν. It will be helpful to consider the following simplified equations, which follow by choice of $i^{[l]}, \alpha^{[l]}, \beta^{[l]}, \vartheta^{[l]}$:

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

and in particular by Lemma 2.1

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

Let now $s^{[0]}(t) = x \in C$. We will prove $\|R(\Phi)(x) - g(x)\|_{\infty,2} \leq \nu$ in four steps, beginning with a characterization of the first layer.

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 $x_i^C \leq x_i < y_i^C$. Due to $0 \leq u_i^{[1]}(0) = 0 < 1$ 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

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

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

We will now continue with characterizing the second layer. We start with the "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 $T_r \geq 2$ and in particular $T_2 = KT_r \geq 2$, 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 $s^{[1]}_{a_2}(t) = 1 \Leftrightarrow t \geq T_4$.

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

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

$$i^{[2]}_{\iota_j(i)}(k) + b^{[2]}_{\iota_j(i)} = s^{[1]}_i(k) + T(s^{[2]}_{c_1}(k-1) - 2s^{[2]}_{c_2}(k-1) + r_s(q^{(j)}; k-1)) = s^{[1]}_i(k),$$
$$i^{[2]}_{\omega_j}(k) + b^{[2]}_{\omega_j} = \frac{1}{n}\sum_{i=1}^{n} s^{[2]}_{\iota_j(i)}(k-1) - 2s^{[2]}_{a_2}(k-1) + r_s(q^{(j)}; k-1) = 0$$

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

$$p^{[2]}_{\iota_j(i)}(t) = -q^{(j)}_i T_r - T + 1 + \sum_{k=1}^{t} s^{[1]}_i(k) - \sum_{k=1}^{t-1} s^{[2]}_{\iota_j(i)}(k) = -q^{(j)}_i T_r - T + 1 + \sum_{k=1}^{t} s^{[1]}_i(k),$$
$$p^{[2]}_{\omega_j}(t) = \sum_{k=1}^{t} 0 - \sum_{k=1}^{t-1} s^{[2]}_{\omega_j}(k) = 0.$$

   Since $\sum_{k=1}^{t} s^{[1]}_i(k) \leq T_2 - 1 < T - 1$ and $s^{[2]}_{c_1}(t) = \chi_{\{T_2-1\}}(t)$, we get $p^{[2]}_{\iota_j(i)}(t) \leq 0$. Thus $\forall_{i\in[n]} s^{[2]}_{\iota_j(i)}(t) = 0$ and $s^{[2]}_{\omega_j}(t) = 0$.

2. Phase 2
   Just as in phase 1, we have

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

   Thus we also get $p^{[2]}_{\omega_j}(T_2) = 0$ and $s^{[2]}_{\omega_j}(T_2) = 0$. This differs 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 $\iota_j(i)$ then fires exactly at T_2, if $x_i \geq x^{C(j)}_i$: First notice that

$$i^{[2]}_{\iota_j(i)}(T_2) + b^{[2]}_{\iota_j(i)} = s^{[1]}_i(T_2) + T(s^{[2]}_{c_1}(T_2 - 1) - 2s^{[2]}_{c_2}(T_2 - 1) + r_s(q^{(j)}; T_2 - 1)) = s^{[1]}_i(T_2) + T,$$

   and conclude that

$$\begin{aligned} p^{[2]}_{\iota_j(i)}(T_2) &= p^{[2]}_{\iota_j(i)}(T_2 - 1) + i^{[2]}_{\iota_j(i)}(T_2) + b^{[2]}_{\iota_j(i)} - s^{[2]}_{\iota_j(i)}(T_2 - 1) \\ &= -q^{(j)}_i T_r + 1 + \sum_{k=1}^{T_2} s^{[1]}_i(k) \\ &= -q^{(j)}_i T_r + 1 + \lfloor T_2 o_i(x) \rfloor \end{aligned}$$

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

3. Phase 3
The "activator neuron" ω_j fires at T_3 if and only if $x^{C^{(j)}} \leq x$: First, notice
$$i^{[2]}_{\omega_j}(T_3)+b^{[2]}_{\omega_j}=\frac{1}{n}\sum_{i=1}^{n}s^{[2]}_{\iota_j(i)}(T_2)-2s^{[2]}_{a_2}(T_2)+r_s(q^{(j)};T_2)=\frac{1}{n}\sum_{i=1}^{n}s^{[2]}_{\iota_j(i)}(T_2)$$
from which we derive
$$p^{[2]}_{\omega_j}(T_3)=p^{[2]}_{\omega_j}(T_2)+i^{[2]}_{\omega_j}(T_3)+b^{[2]}_{\omega_j}-s^{[2]}_{\omega_j}(T_2)=\frac{1}{n}\sum_{i=1}^{n}s^{[2]}_{\iota_j(i)}(T_2).$$
So $0\leq p^{[2]}_{\omega_j}(T_3)\leq 1$ and we get $p^{[2]}_{\omega_j}(T_3)=1$, as well as $s^{[2]}_{\omega_j}(T_3)=1$ exactly if $\forall_{i\in[n]}x_i^{C^{(j)}}\leq x_i$, i.e. if $x^{C^{(j)}}\leq x$.
Let further $i\in[n]$. We then have
$$i^{[2]}_{\iota_j(i)}(T_3)+b^{[2]}_{\iota_j(i)}=s^{[1]}_i(T_3)+T(s^{[2]}_{c_1}(T_2)-2s^{[2]}_{c_2}(T_2)+r_s(q^{(j)};T_2))=-2T$$
and therefore
$$\begin{aligned}p^{[2]}_{\iota_j(i)}(T_3)&=p^{[2]}_{\iota_j(i)}(T_2)+i^{[2]}_{\iota_j(i)}(T_3)+b^{[2]}_{\iota_j(i)}-s^{[2]}_{\iota_j(i)}(T_2)\\&=-q_i^{(j)}T_r+1+\lfloor T_2o_i(x)\rfloor-2T-s^{[2]}_{\iota_j(i)}(T_2).\end{aligned}$$
Thus $p^{[2]}_{\iota_j(i)}(T_3)\leq -T$ and $s^{[2]}_{\iota_j(i)}(T_3)=0$, since $\lfloor T_2o_i(x)\rfloor\leq T_2\leq T-1$.

4. Phase 4
Let $i\in[n]$. The neuron $\iota_j(i)$ stays inactive at T_4, since
$$i^{[2]}_{\iota_j(i)}(T_4)+b^{[2]}_{\iota_j(i)}=s^{[1]}_i(T_4)+T(s^{[2]}_{c_1}(T_3)-2s^{[2]}_{c_2}(T_3)+r_s(q^{(j)};T_3))=Tr_s(q^{(j)};T_3)$$
and hence
$$\begin{aligned}p^{[2]}_{\iota_j(i)}(T_4)&=p^{[2]}_{\iota_j(i)}(T_3)+i^{[2]}_{\iota_j(i)}(T_4)+b^{[2]}_{\iota_j(i)}-s^{[2]}_{\iota_j(i)}(T_3)\\&=p^{[2]}_{\iota_j(i)}(T_3)+Tr_s(q^{(j)};T_3).\end{aligned}$$
Since $p^{[2]}_{\iota_j(i)}(T_3)\leq -T$, we conclude $p^{[2]}_{\iota_j(i)}(T_4)\leq 0$ and $s^{[2]}_{\iota_j(i)}(T_4)=0$.
Further the "activator neuron" ω_j fires at T_4 exactly if $x\in C^{(j)}$: First, notice that
$$i^{[2]}_{\omega_j}(T_4)+b^{[2]}_{\omega_j}=\frac{1}{n}\sum_{i=1}^{n}s^{[2]}_{\iota_j(i)}(T_3)-2s^{[2]}_{a_2}(T_3)+r_s(q^{(j)};T_3)=r_s(q^{(j)};T_3).$$
We thus have
$$\begin{aligned}p^{[2]}_{\omega_j}(T_4)&=p^{[2]}_{\omega_j}(T_3)+i^{[2]}_{\iota_j(i)}(T_4)+b^{[2]}_{\iota_j(i)}-s^{[2]}_{\iota_j(i)}(T_3)\\&=p^{[2]}_{\omega_j}(T_3)+r_s(q^{(j)};T_3)-s^{[2]}_{\iota_j(i)}(T_3).\end{aligned}$$
By definition of q, $q_i^{(j)}<q_i^{(j')}$ holds exactly if $x_i^{C^{(j)}}<x_i^{C^{(j')}}$ for all $i\in[n]$ and $j,j'\in[K^n]$. We thus conclude $\forall_{j,j'\in[K^n]}q^{(j)}<q^{(j')}\Leftrightarrow x^{C^{(j)}}<x^{C^{(j')}}$. Note that we have shown $\forall_{j'\in K^n}s^{[2]}_{\omega_{j'}}(T_3)=1\Leftrightarrow x^{C^{(j')}}\leq x$ before. We therefore have
$$\begin{aligned}r_s(q^{(j)};T_3)&=s^{[2]}_{\omega_j}(T_3)-\sum_{\substack{q'\in([K-1]_0)^n\\q^{(j)}<q'}}s^{[2]}_{\omega_{j(q')}}(T_3)\\&=s^{[2]}_{\omega_j}(T_3)-\sum_{\substack{j'\in[K^n]\\x^{C^{(j)}}<x^{C^{(j')}}\leq x}}1.\end{aligned}$$

Now if $x \in C^{(j)}$, i.e. $x^{C^{(j)}} \leq x < y^{C^{(j)}}$, then $s^{[2]}_{\omega_j}(T_3) = 1$. If there also was a j' with $s^{[2]}_{\omega_{j'}}(T_3) = 1$ and $x^{C^{(j)}} < x^{C^{(j')}} \leq x$, 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 $p^{[2]}_{\omega_j}(T_3) = 1$ and $s^{[2]}_{\omega_j}(T_3) = 1$ for this case, so we can conclude $p^{[2]}_{\omega_j}(T_4) = 1$ as well as $s^{[2]}_{\omega_j}(T_4) = 1$.

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 $x \in C^{(j')}$ for some $j' \neq j$.

Now if $x^{C^{(j)}} < x^{C^{(j')}}$, then $r_s(q^{(j)};T_3) \leq 0$. In this case we also have $p^{[2]}_{\omega_j}(T_3) = 1$ and $s^{[2]}_{\omega_j}(T_3) = 1$ such that $p^{[2]}_{\omega_j}(T_4) \leq 0$ and $s^{[2]}_{\omega_j}(T_4) = 0$.

Is on the other hand $x^{C^{(j)}} \not< x^{C^{(j')}}$, then there is an index $i \in [n]$ 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 get in particular $r_s(q^{(j)};T_3) \leq 0$. We thus have $p^{[2]}_{\omega_j}(T_3) < 1$ and $s^{[2]}_{\omega_j}(T_3) = 0$ such that $p^{[2]}_{\omega_j}(T_4) < 1$ and $s^{[2]}_{\omega_j}(T_4) = 0$.

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

5. Phase 5
   The “activator neuron” ω_j is inactive during T_5, since for $t > T_4$

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

and

$$\begin{aligned} p^{[2]}_{\omega_j}(t) &= u^{[2]}_{\omega_j}(T_4) + \sum_{k=T_4+1}^{t}\left(i^{[2]}_{\omega_j}(k) + b^{[2]}_{\omega_j}\right) - \sum_{k=T_4+1}^{t-1} s^{[2]}_{\omega_j}(k) \\ &\leq u^{[2]}_{\omega_j}(T_4). \end{aligned}$$

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

Let $i \in [n]$. We show next that during T_5 the neuron $\iota_j(i)$ captures the position of x_i in $[x_i^{C^{(j)}}, y_i^{C^{(j)}})$ if $x \in C^{(j)}$ and stays inactive otherwise.

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

$$i^{[2]}_{\iota_j(i)}(t) + b^{[2]}_{\iota_j(i)} = s^{[1]}_i(t) + T(s^{[2]}_{c_1}(t-1) - 2s^{[2]}_{c_2}(t-1) + r_s(q^{(j)};t-1)) \leq 0$$

as well as

$$\begin{aligned} p^{[2]}_{\iota_j(i)}(t) &= p^{[2]}_{\iota_j(i)}(T_4) + \sum_{k=T_4+1}^{t}\left(i^{[2]}_{\iota_j(i)}(k) + b^{[2]}_{\iota_j(i)} - s^{[2]}_{\iota_j(i)}(k-1)\right) \\ &= p^{[2]}_{\iota_j(i)}(T_4) + \sum_{k=T_4+1}^{t}\left(i^{[2]}_{\iota_j(i)}(k) + b^{[2]}_{\iota_j(i)} - s^{[2]}_{\iota_j(i)}(k-1)\right) \\ &\leq p^{[2]}_{\iota_j(i)}(T_4) \\ &\leq 0. \end{aligned}$$

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

Suppose now $x \in C^{(j)}$. As we have seen before, we have $r_s(q^{(j)};T_4) = 1$ and $\forall_{t>T_4} r_s(q^{(j)};t) = 0$. Hence

$$i^{[2]}_{\iota_j(i)}(T_4+1) + b^{[2]}_{\iota_j(i)} = s^{[1]}_i(T_4+1) + T(s^{[2]}_{c_1}(T_4) - 2s^{[2]}_{c_2}(T_4) + r_s(q^{(j)};T_4)) = T$$

and for all $t > T_4+1$

$$i^{[2]}_{\iota_j(i)}(t) + b^{[2]}_{\iota_j(i)} = s^{[1]}_i(t) + T(s^{[2]}_{c_1}(t-1) - 2s^{[2]}_{c_2}(t-1) + r_s(q^{(j)};t-1)) = 0.$$

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

$$\begin{aligned}
p^{[2]}_{\iota_j(i)}(T_4+1) &= p^{[2]}_{i_j(i)}(T_4) + i^{[2]}_{\iota_j(i)}(T_4+1) + b^{[2]}_{\iota_j(i)} - s^{[2]}_{\iota_j(i)}(T_4) \\
&= p^{[2]}_{\iota_j(i)}(T_3) + Tr_s(q^{(j)};T_3) + T \\
&= -q^{(j)}_i T_r + 1 + \lfloor T_2 o_i(x) \rfloor - 2T - s^{[2]}_{\iota_j(i)}(T_2) + 2T \\
&= -KT_r o_i(x^{C^{(j)}}) + \lfloor KT_r o_i(x) \rfloor .
\end{aligned}$$

We now have $KT_r o_i(x^{C^{(j)}}) \leq KT_r o_i(x)$ due to $x^{C^{(j)}}_i \leq x_i$ and thus $p^{[2]}_{\iota_j(i)}(T_4+1) \geq 0$.

Furthermore,

$$o_i(y^{C^{(j)}}) - o_i(x^{C^{(j)}}) = \frac{y^{C^{(j)}}_i - x^{C^{(j)}}_i}{y^C_i - x^C_i} = \frac{1}{K}$$

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

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

and have thus shown

$$0 \leq p^{[2]}_{\iota_j(i)}(T_4+1) = u^{[2]}_{i_j(i)}(T_4) + i^{[2]}_{\iota_j(i)}(T_4+1) + b^{[2]}_{\iota_j(i)} \leq T_r = T - (T_4+1).$$

By Lemma 2.8 we get $u^{[2]}_j(T_4+1) \geq 0$ and can therefore use Proposition 2.2 with $t_0 := T_4+1$, $t_m := T_4+1$ and $t_\omega := T$ to obtain

$$\begin{aligned}
\sum_{t=T_4+1}^{T} s^{[2]}_{\iota_{j(i)}}(t) &= \left\lfloor u^{[2]}_{\iota_{j(i)}}(T_4) + i^{[2]}_{\iota_{j(i)}}(T_4+1) + b^{[2]}_{\iota_{j(i)}} \right\rfloor \\
&= -KT_r o_i(x^{C^{(j)}}) + \lfloor KT_r o_i(x) \rfloor .
\end{aligned}$$

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

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

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

$$W^{[L+1]}_{k,\omega_j} \sum_{t=T_4}^{T} s^{[2]}_{\omega_j}(t) = g_k(x^{C^{(j)}}),$$
$$W^{[L+1]}_{k,\iota_{j(i)}} \sum_{t=T_4}^{T} s^{[2]}_{\iota_{j(i)}}(t) = \left(-KT_r o_i(x^{C^{(j)}}) + \lfloor KT_r o_i(x) \rfloor\right) d(g^{(j)})_k((y_i^{C^{(j)}} - x_i^{C^{(j)}})\tfrac{1}{T_r} e_i).$$

Let now $j \in [K^n]$ be such that $x \notin C^{(j)}$, in that case

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

Thus the group of neurons with index j does not contribute to the output of the network,

$$W^{[L+1]}_{k,\omega_j} \sum_{t=T_4}^{T} s^{[2]}_{\omega_j}(t) = 0, \qquad W^{[L+1]}_{k,\iota_{j(i)}} \sum_{t=T_4}^{T} s^{[2]}_{\iota_{j(i)}}(t) = 0.$$

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 $j \in \{c_1, c_2, a_2\}$

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

Let now $j \in [K^n]$ be such that $x \in C^{(j)}$. We consequently have

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

which is further equal to

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

Now by assumption, we have

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

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

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

Hence, by definition of ξ_i and $(y_i^{C(j)} - x_i^{C(j)})K = (y_i^C - x_i^C)$ we have

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

and thus conclude

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

□

Unfortunately 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 $\left\|f'|_{(0,3\pi)}\right\|_\infty = 1$, $\Gamma := 1$ is the optimal Lipschitz-constant for f. At the same time, since $f''(x) = n\sin(nx)$ and $\left\|f''|_{(0,3\pi)}\right\|_\infty = n$, the biggest possible modulus of uniform continuity on $(0, 3\pi)$ for f' is $\delta(\varepsilon) := \frac{\varepsilon}{n}$. Thus

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

Hence $K(\varepsilon) := \lfloor \pi\sqrt{\frac{n}{\varepsilon}}\rfloor$. We therefore get the following network sizes:

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

While the first layer of Theorem 3.2 is clearly arbitrarily smaller than the first layer of the other construction for $\varepsilon \to 0$, the second layer of Theorem 3.2 is arbitrarily worse than the second layer of Theorem 3.1 for $n \to \infty$.

On the other hand, even for large $n \in \mathbb{N}$, the second layer of Theorem 3.2 is arbitrarily more efficient for $\varepsilon \to 0$, 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 to arbitrary functions 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 and assume that f and $d(f|_{U^\circ})$ are Γ,Γ'-Lipschitz respectively.*

We then have

$$\lim_{\varepsilon\to 0} \frac{n_2(\varepsilon)}{n_2'(\varepsilon)} = 0,$$

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 defined as in Theorem 3.2 for sets U, C, Ω and function f with modulus of uniform continuity $\omega(x) := \Gamma x$ of $d(f|_{U^\circ})$ and approximation parameter $\mu := \frac{\varepsilon}{2}$.

Proof. First notice, that since

$$n_2'(\varepsilon) = \max\left\{\left\lceil \frac{\operatorname{diam}_\infty(\Omega)}{\varepsilon}\Gamma' \right\rceil^n, 1\right\}$$

we have

$$\lim_{\varepsilon\to 0} n_2'(\varepsilon)\varepsilon = (\operatorname{diam}_\infty(\Omega)\Gamma')^n.$$

We further have (c.f. Remark 3.2)

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

such that we get

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

We therefore have

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

and hence conclude

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

□

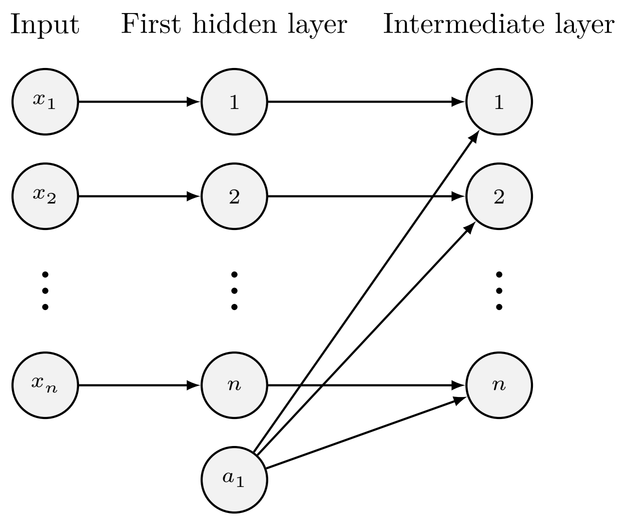


Figure 3.9: Replacing the first hidden layer by non-recurrent structure

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 to remove the interdependencies inside of the layers to obtain a d.t. LIF-SNN by replicating layers as often as needed and connecting where necessary. For example 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 intermediate layer with n neurons between the first and second hidden layer that just forward the input from their respective neuron in the first layer [3] and also take in an input from

[3] see also Example 2.1

a_1, compare Fig. 3.9. However, compared to our construction, this would significantly increase the network complexity.

4 Complexity of input partitions

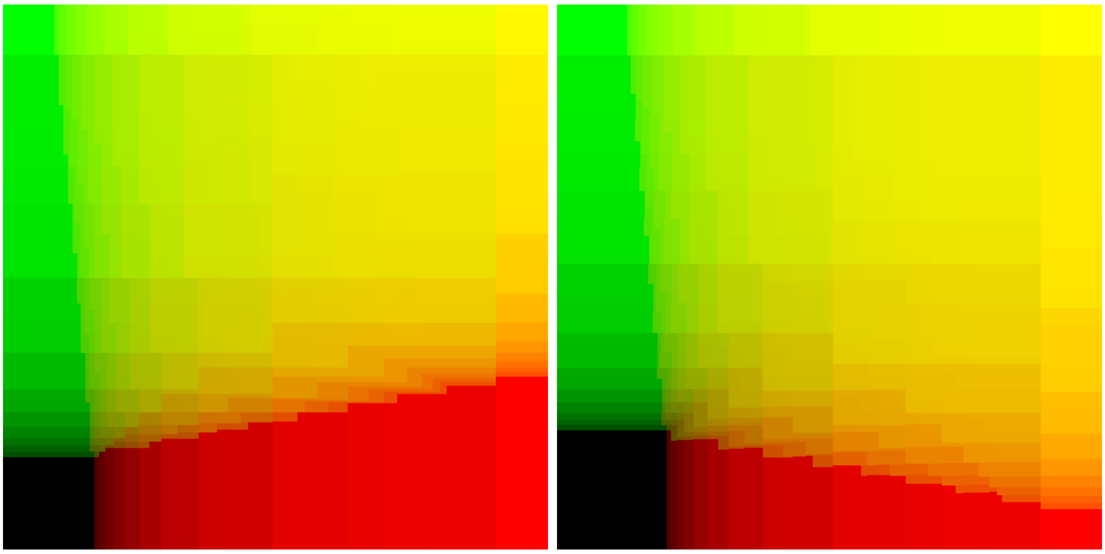


Figure 4.1: The landscape of the first layer of two different R-LIF-SNN

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 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 assume $W^{[1]} = I_{n_1}$. Let Φ be a R-LIF-SNN with $W^{[1]}$ arbitrary and Φ' be a copy of Φ, but with trivial $W^{[1]} = I_{n_1}$. 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 only decrease the number of different output values. Hence, 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 will moreover 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. We further write $x := s^{[0]}$. Since we are using direct encoding, we have $\forall_{t\in[T]} s^{[0]}(t) = x$ and hence obtain the following simplified defining equations:

$$i(t) = \alpha i(t-1) + Wx + Vs(t-1), \tag{19}$$

$$p(t) = \beta u(t-1) + i(t) + b, \tag{20}$$

$$s(t) = H(p(t) - \vartheta \mathbf{1}_n), \tag{21}$$

$$u(t) = p(t) - \vartheta s(t). \tag{22}$$

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 also be helpful to write e.g. $i(\cdot;x)$ for $(i(t;x))_{t\in[T]}$.

By using this simplified notation, we obtain the following equations from Definition 2.6 for the input vector $x \in \mathbb{R}^{n_0}$ and the first hidden layer spike train σ:

$$\begin{aligned}
i(t;x;\sigma) &= \alpha^t i(0) + \sum_{k=1}^{t} \alpha^{t-k} \left(Wx + V\sigma(k-1)\right), \\
p(t;x;\sigma) &= \beta^t u(0) + \sum_{k=1}^{t} \beta^{t-k} \left(i(k;x;\sigma) + b\right) - \vartheta \sum_{k=1}^{t-1} \beta^{t-k} \sigma(k), \\
s(t;x;\sigma) &= H(p(t;x;\sigma) - \vartheta \mathbf{1}_{n_1}), \\
u(t;x;\sigma) &= \beta^t u(0) + \sum_{k=1}^{t} \beta^{t-k} \left(i(k;x;\sigma) + b - \vartheta\sigma(k)\right).
\end{aligned}$$

Let us now introduce some preliminary definitions.

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

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

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 $C_{\Phi,1} := \{C_{s'} \mid s' \in \{0,1\}^{n_1 \times T}, C_{s'} \neq \emptyset\}$.

Note that we have argued in the beginning of this chapter that in fact $|C_\Phi| \leq |C_{\Phi,1}|$.

While quite technical the following lemma is the key for writing rigorous proofs about the landscape of R-LIF-SNNs.

Lemma 4.1. *Consider the function $g : \dot{\bigcup}_{t \in [T]} \{\sigma \mid \sigma \in \{0,1\}^{n_1 \times (t-1)}\} \to \mathbb{R}$ defined by*

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

Then g satisfies $\forall_{i \in [n_1]} s_i(t;x;\sigma) = 1 \Leftrightarrow \langle w_i, x\rangle \geq g_i(t;\sigma)$. Here w_i denotes the i-th row vector of W.

Proof. We compute for $i \in [n_1]$ using Lemma 2.1,

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

□

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

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 $C_{s'} = [\![x^{s'}, y^{s'}\rangle\!\rangle$ be the corresponding constant region. Then*

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

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 a 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*

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

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

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

where we use Lemma 2.1 for the second equality and Lemma 4.1 for the last one. □

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

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

We thus conclude with

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

□

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*

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

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. □

We are now ready to proof Proposition 4.1.

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

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

□

We will order spike trains using lexicographical ordering.

Definition 4.2. Let $s', s'' \in \{0, 1\}^{n_1 \times T}$, we then define $s' \leq_l s''$ to be true exactly in the case that either $s' = s''$ or that there exists a $t \in [T]$ such that $s'(t) < s''(t)$ and $\forall_{t'<t} s'(t') = s''(t')$. We further define $s' <_l s''$ to be true if and only if $s' \leq_l s''$ but not $s' = s''$.

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 $x_i < y_i$.

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 hence have $s(t; x) <_l s(t; y)$.

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 $x_i < y_i$. □

We will further prove some theorems about specific classes of spike trains.

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. A point $p \in (\mathbb{R} \cup \{\pm\infty\})^n$ as a vertex of a non-empty region $[\![x,y)\!)$, $[\![x,y]\!]$ or $(\!(x,y)\!)$, if $\forall_{i \in [n]} p_i \in \{x_i, y_i\}$. We call the 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. In both cases, $C_{s'}$ is clearly non-empty. □

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. 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 $\left|\mathbb{R} \cap \{x_i^{s'}, y_i^{s'}\}\right| = 1$. 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, $\mathbb{R}^{n_1} \cap \prod_{j \in [n_1]} \{x_j^{s'}, y_j^{s'}\}$. □

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 are equivalent:*

(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.

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

- $(a) \Leftrightarrow (b)$: Follows directly by Lemma 4.7.
- $(b) \Rightarrow (c)$: A non-empty cube $[\![x^{s'}, y^{s'})\!)$ 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 at least one 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 $[\![x^{s'}, y^{s'})\!)$ 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 have a $z \in C_{s'}$. By definition of $[\![x^{s'}, y^{s'})\!)$, we have now $z' \in [\![x^{s'}, y^{s'})\!)$ 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 analogous.

□

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 $P \subset [\![x^C, y^C]\!]$ 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 $P \subset [\![x^C, y^C]\!]$. We further have $x_i^C = \min_{t \in [T]} g_i(t; s^{x_i^C})$ 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 that $C_{s'} \subset [\![x^C, y^C)\!)$ for all regions $C_{s'} \in C_{\Phi,1}$ with non-constant spike train s': By Lemma 4.8 all vertices of $C_{s'} = [\![x^{s'}, y^{s'})\!)$ are finite, so $x^{s'}, y^{s'} \in P \subset [\![x^C, y^C]\!]$ and therefore $[\![x^{s'}, y^{s'})\!) \subset [\![x^C, y^C)\!)$.

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 look for those vertices.

Proof. We will first proof $P \subset [\![x^C, y^C)\!)$: Let $v \in P$ be a finite vertex of region $C_{s'} = [\![x^{s'}, y^{s'})\!)$. 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 $s'(t_1) = 1$; is 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 and t_1 be minimal with $s'(t_1) = 1$ and define $s^1 \in \{0,1\}^{n_1 \times t_1}$ by $s^0(t) := \mathbf{0}_{n_1}$. By choice of t_1, we then 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 $g_i(t_1; s^0) = g_i(t_1; s')$. Thus

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

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

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

Hence $v \in [\![x^C, y^C]\!]$.

In fact, x^C, y^C are chosen optimal: Let $i \in [n_1]$. Now the spike train $s' := s^{x_i^C}$ is constant in every component, so $C_{s'} = [\![x^{s'}, y^{s'}\rangle\!\rangle$ 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

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

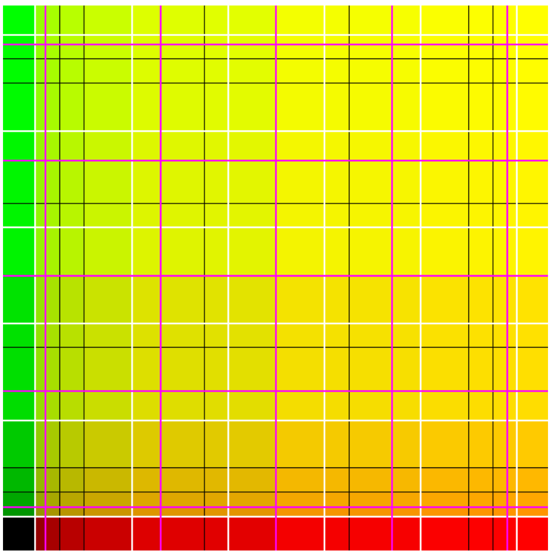
by definition. We similarly get a vertex $v \in P$ with $v_i = x_i^{s^{y_i^C}} = y_i^C$.

Finally, it suffices to compute g on $s_j^{x_i^C}$ and $s_j^{y_i^C}$ to obtain the boundaries. Consider the definition of g again. Since it is quite unwieldy we have split it up into the following functions,

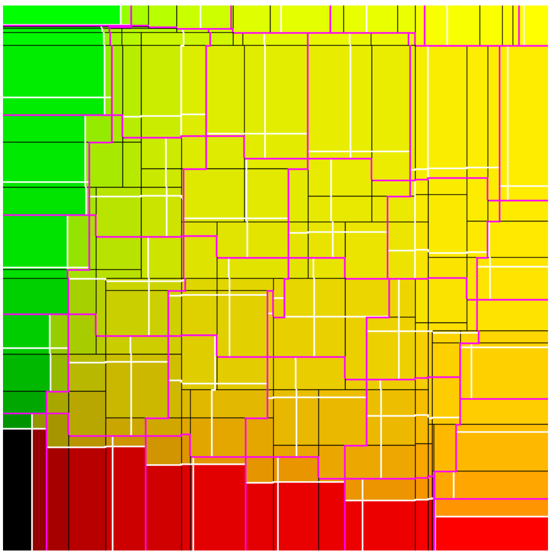
$$g_i^r(t;\sigma) := -\frac{\sum_{k=1}^t \beta^{t-k}\left(\alpha^k i_i(0) + b + \sum_{l=1}^k \alpha^{k-l}\langle v_i, \sigma(l-1)\rangle\right) + \beta^t u_i(0)}{\sum_{k=1}^t \beta^{t-k}\sum_{l=1}^k \alpha^{k-l}},$$

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

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 $x_i^C = \min_{t\in[T]} g_i(t; s^{y_i^C})$. □



(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; Black lines are borders between regions with different spike trains that are not yet colored white or pink.

Theorem 4.3 of [Nguyen et al., 2025] states the following:

Theorem 4.1. *Consider a d.t. 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*

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

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 proof 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 reduced 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*

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

Sketch. As we have stated before, $|C_\Phi| \le |C_{\Phi,1}|$ 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 actual lines instead of line segments like in Fig. 4.2b. This shows the independence of the two neurons in the first hidden layer to 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 $V = \mathbf{0}_{n_1 \times n_1}$,

$$\begin{aligned} i(t) &= \alpha i(t-1) + x, \\ p(t) &= \beta u(t-1) + i(t) + b, \\ s(t) &= H(p(t) - \vartheta \mathbf{1}_n), \\ u(t) &= p(t) - \vartheta s(t) \end{aligned}$$

and in particular

$$i(t;x) = \alpha^t i(0) + x \sum_{k=1}^{t} \alpha^{t-k}, \tag{25}$$

$$p(t;x) = \beta^t u(0) + \sum_{k=1}^{t} \beta^{t-k}(i(k;x) + b) - \vartheta \sum_{k=1}^{t-1} \beta^{t-k} s(k). \tag{26}$$

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 $W = I_{n_1}$. It thus suffices to analyze the possible change of the output of s_i depending on different values for x_i.

Let us now define $\Phi_{t,i}$ as the restriction of Φ to 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 $\Phi_{t,i}$.

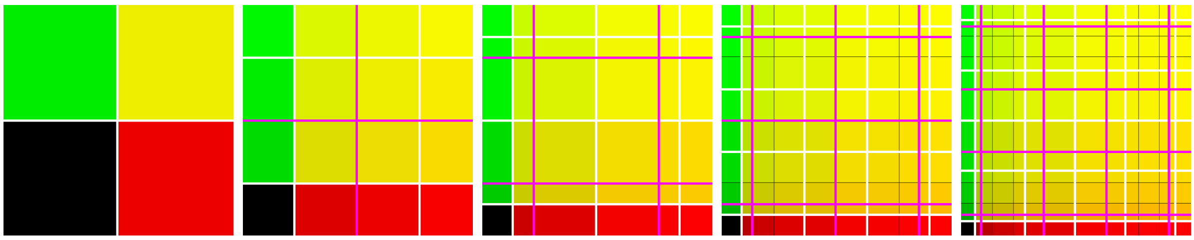
We will now proof that $(s(t'))_{t' \in [t]}$ can take on $\frac{t^2+t+2}{2}$ many different values for $t \in [T]$ by induction on t. This suffices to proof the theorem since we then have n_1 independent neurons in the first layer that each can take on a maximal number of $\frac{T^2+T+2}{2}$ many values. 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 $s, \alpha, \beta, \ldots$ 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 $V = 0$,

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

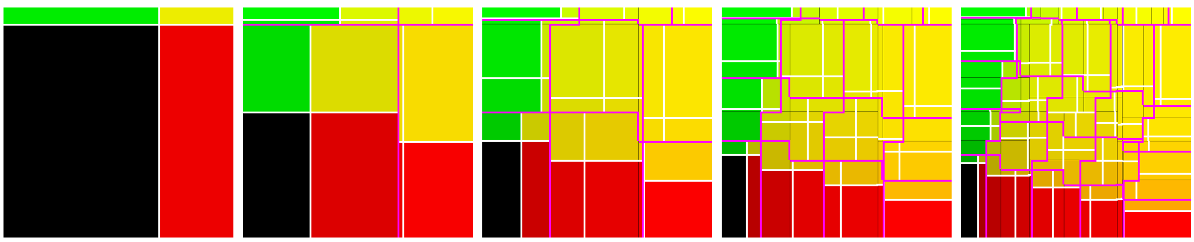
Clearly $g(t;\sigma)$ only changes in the subterm $\sum_{k=1}^{t-1} \beta^{t-k}\sigma(k)$ in σ. Suppose $x, x' \in \mathbb{R}$ are given with $x < x'$ and $s(\cdot;x) \neq s(\cdot;x')$. Then there appears a spike earlier in $s(\cdot;x)$ than in $s(\cdot;x')$ due to Lemma 4.5. In fact, if both $s(\cdot;x)$ 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. Thus we get at a maximum as many new regions as categories. We therefore obtain $t + \frac{t^2-t+2}{2} = \frac{t^2+t+2}{2}$ regions in total. □



(a) $T=1$ (b) $T=2$ (c) $T=3$ (d) $T=4$ (e) $T=5$

Figure 4.3: Development of landscape of a d.t. LIF-SNN through time; Development of landscape of R-LIF-SNNs 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; Black lines are borders between regions with different spike trains that are not yet colored white or pink.



(a) $T=1$ (b) $T=2$ (c) $T=3$ (d) $T=4$ (e) $T=5$

Figure 4.4: Development of landscape of a R-LIF-SNN through time; See Fig. 4.3 for further description.

While we have not proven it for the general case, note that the introduction of α did not affect the theorem, only V prevents the proof. Let us now analyze why this 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 look at Fig. 4.2b.

To reduce the impact of the interconnectedness on our proof 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 to all neurons simultaneously, i.e. we consider regions of equal number of spikes in each respective component one by one.

Compare the different time-steps in Fig. 4.4. The pink lines are the white lines of the respective previous time-step and all black lines are older boundaries that were white and pink before, since every separating line was at some point a new line and every new line separates regions with different spike count[4].

We further see that the plane is partitioned by the pink lines into similarly 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. This is not the case for $\beta \neq 1$, see e.g. Fig. 4.5 for a R-LIF-SNN with $\beta \neq 1$.

Now for $\beta = 1$ it is not hard to prove that inside of a given category, there is only a single cross with no other parallel lines, which is quite a promising result. One might think that we can just finish the proof as before since we now know how many new lines get added in time-step t. But a considerable problem arises when considering how many regions exactly will be added at a maximum during this time-step. That number 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.

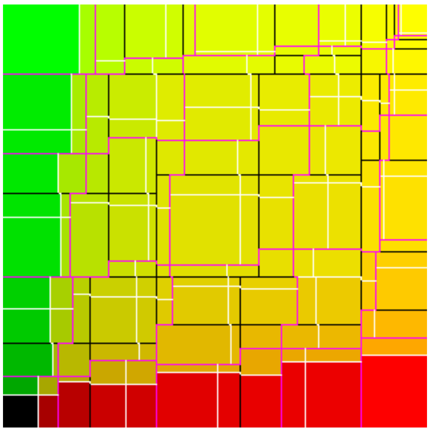


Figure 4.5: For $\beta \neq 1$ and $V \neq \mathbf{0}_{n_1 \times n_1}$, the white crosses inside of categories are not made of straight lines

This question is very much non-trivial, in particular because in contrast to the case of $V = \mathbf{0}_{n_1 \times n_1}$, it cannot be reduced to looking at the maximal number of regions that can be touched by a straight line in the whole plane. This is due to the categories as well as the crosses being potentially shifted in relation to each other, as can be seen in e.g. Fig. 4.4.

[4]The regions around a new line differ in exactly one spike

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 expect be able to then extend our result to the more general case.

Algorithm 1 `CompRegions`

function $\textsc{SubRegions}(\Phi, t, st, x^C, y^C)$
  $c \leftarrow g_\Phi(t+1; st)$
  $subRegions \leftarrow \{\}$
  for $sp \in \{0,1\}^{n_1}$ **do**
    $x' \leftarrow (\text{if } sp_i = 0 \text{ then } x_i^C \text{ else } \max(x_i^C, c_i))_{i \in [(n_\Phi)_1]}$
    $y' \leftarrow (\text{if } sp_i = 1 \text{ then } y_i^C \text{ else } \min(y_i^C, c_i))_{i \in [(n_\Phi)_1]}$
    if $x' < y'$ **then**
      $subRegions \leftarrow subRegions \cup \{(x', y', st +\!\!+ [sp])\}$
    end if
  end for
  return $subRegions$
end function
function $\textsc{RegionsWithST}(\Phi, t, st, x^C, y^C)$
  if $t \geq T_\Phi$ **then**
    return $\{(st, x^C, y^C)\}$
  end if
  $regions \leftarrow \{\}$
  for $(x', y', st) \in \textsc{SubRegions}(\Phi, t, st, x^C, y^C)$ **do**
    $newRegions \leftarrow \textsc{RegionsWithST}(\Phi, t+1, st, x', y')$
    $regions \leftarrow regions \cup newRegions$
  end for
  return $regions$
end function
function $\textsc{CompRegions}(\Phi)$
  return $\textsc{RegionsWithST}(\Phi, 0, [(0, \ldots, 0)], (-\infty, \ldots, -\infty), (\infty, \ldots, \infty))$
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 $\forall_{t \in [T]} s_i'(t) = 0$ for all $i \neq 1$ is represented by $st = [(0, \ldots, 0), (1, 0, \ldots, 0), \ldots, (1, 0, \ldots, 0)]$. Further a region $C_{s'} = [\![x^{s'}, y^{s'})\!)$ 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, using parameters from Φ. Similarly T_Φ is the total latency T of Φ.

Lemma 5.1. *Let Φ be a R-LIF-SNN with $W^{[1]} = I_{n_1}$. The algorithm* `CompRegions`(Φ) *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 write Φ_τ for the R-LIF-SNN that only differ from Φ by having a restricted number of iterations, i.e. $T_{\Phi_\tau} = \tau$.

Let us now show that given $C_{s'} = [\![x^{s'}, y^{s'})\!) \in C_{\Phi_{t-1},1}$ with $s' \in \{0,1\}^{n_1 \times (t-1)}$, the algorithm $\texttt{SubRegions}(\Phi, t, s', x^{C_{s'}}, y^{C_{s'}})$ computes all regions $C_{s''} = [\![x^{s''}, y^{s''})\!) \in C_{\Phi_t,1}$, where $s'' \in \{0,1\}^{n_1 \times t}$ is an extension by one time-step 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 $\texttt{SubRegions}$, it is clear that given a region $C_{s'} \in C_{\Phi_{t-1},1}$, the algorithm iterates over all possible extensions s'' of s' by one time-step, computes $C_{s''}$ and checks if $C_{s''}$ is non-empty. In that case, $C_{s''}$ is appended to the output.

We similar get that $\texttt{SubRegions}(\Phi, 0, [(0, \dots, 0)], (-\infty, \dots, -\infty), (\infty, \dots, \infty))$ 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. We thus have

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

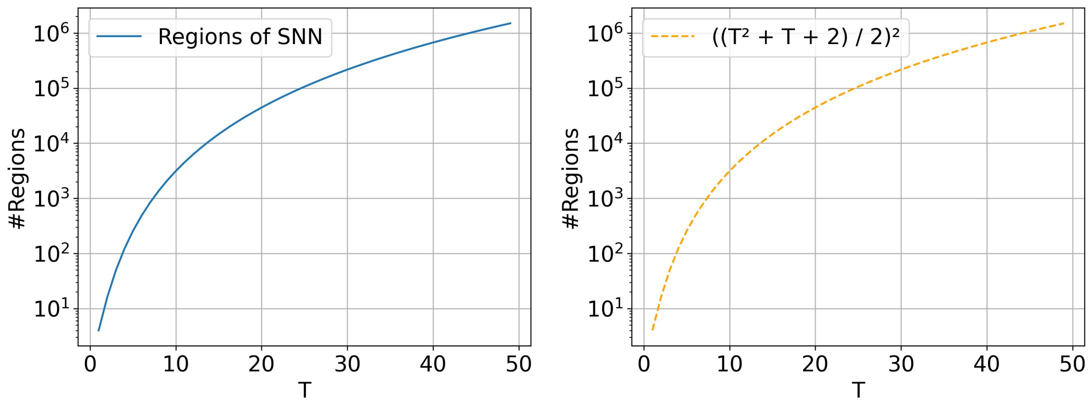
which is clearly what $\texttt{SubRegions}(\Phi, 0, [(0, \dots, 0)], (-\infty, \dots, -\infty), (\infty, \dots, \infty))$ will compute.

Since $\texttt{RegionsWithST}$ just repeatedly maps $\texttt{SubRegions}$ over an initial value of $\{(\Phi, 0, [(0, \dots, 0)], (-\infty, \dots, -\infty), (\infty, \dots, \infty))\}$, and since further $\texttt{SubRegions}$ 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, $\texttt{RegionsWithST}$ 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 one 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 $C_{s'} = [\![x^{s'}, y^{s'})\!)$, 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 to avoid the previous problem does not seem correct either, since properly non-empty 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 $\Phi_{\max,T,n_0}$ 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, hence we can represent at most 2^{64} values. Thus at the very latest at $T \approx 2^{32}\sqrt{2}$ the algorithm will start to undercount the regions. Of course this will happen far earlier, since the borders of the regions are not fitted to the floating point number representatives.

Now $\Phi_{\max,T,n_0}$ is constructed in [Nguyen et al., 2025] as 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 value in every component $u_i(0) > 0$ of $u(0)$.

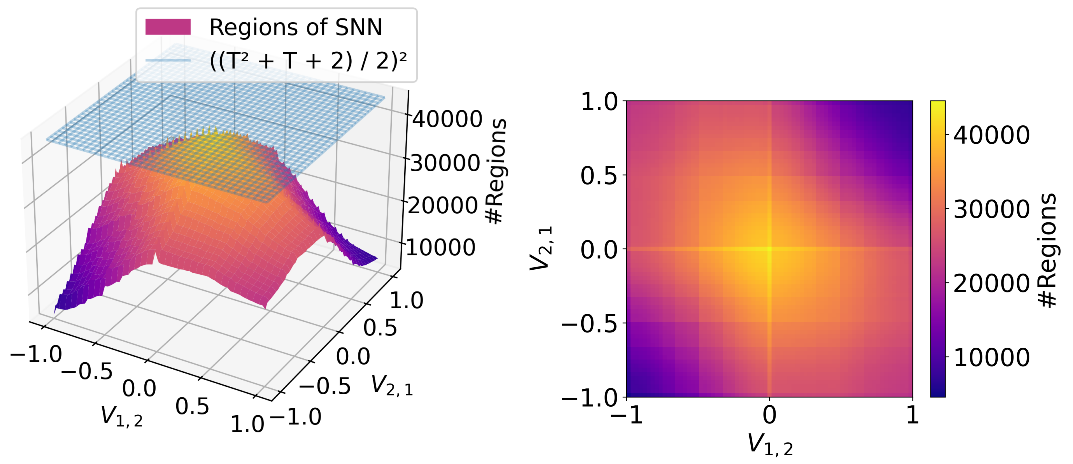


(a) $\left|C_{\Phi_{\max,T,n_0},1}\right|$ for different values of T **(b)** Theoretical limit for d.t. LIF-SNN

Figure 5.1: Optimal Φ vs. theoretical limit

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 $\Phi_{\max,T,n_0}$ corresponds to the theoretical limit in Fig. 5.1b.



(a) 3D Plot **(b)** Color Plot

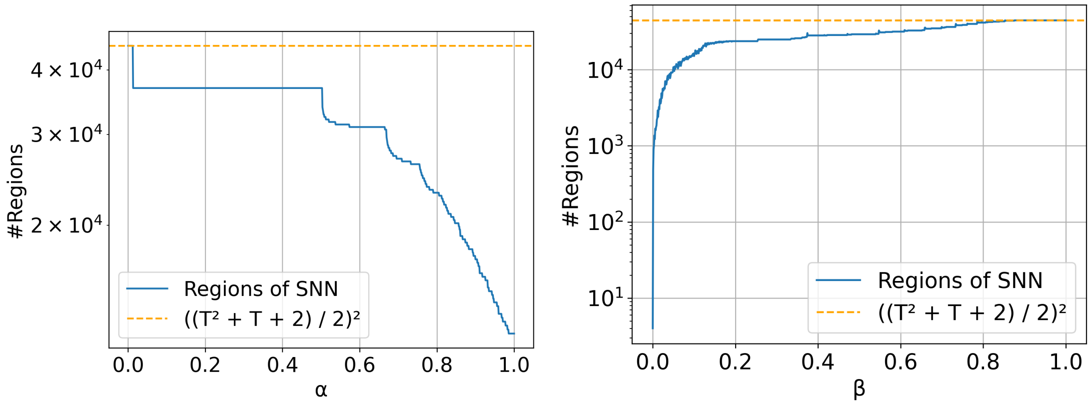
Figure 5.2: $\left|C_{\Phi,1}\right|$ for $T = 20$ and different values of $V_{1,2}$ and $V_{2,1}$

Let us now consider a R-LIF-SNN $\Phi_{\alpha,V}$ that only differs in α or V from $\Phi_{\max,T,n_0}$. Changing $V_{1,2}$ or $V_{2,1}$ only decreases $\left|C_{\Phi_{0,V},1}\right|$ 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 not efficient enough to compute the number of regions for a high number of iterations efficiently; also 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. But since $\beta > 1$ leads to exponential growth of the membrane potential instead of decay, this is not too much of a problem for our model, because exponential growth does not fit



(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 $T = 20$

neurological realities.

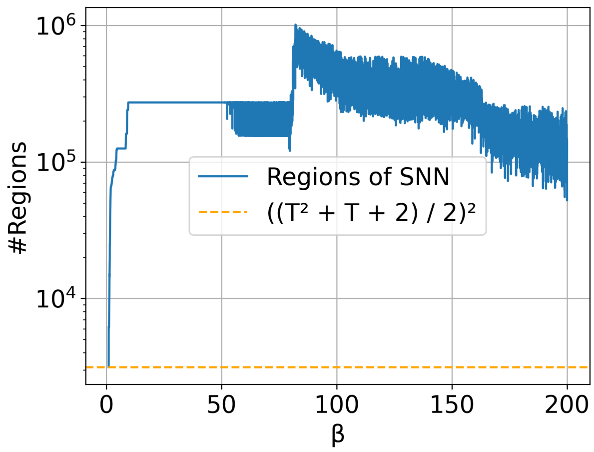


Figure 5.4: Changing β past the limit pushes the number of regions above the threshold for $T = 20$

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

The algorithm just computes $s^{[1]}$ on a grid in the input space and then determines 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 the following disadvantages: Since there are no dynamically-sized lists in the GPU programming language GLSL, 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.

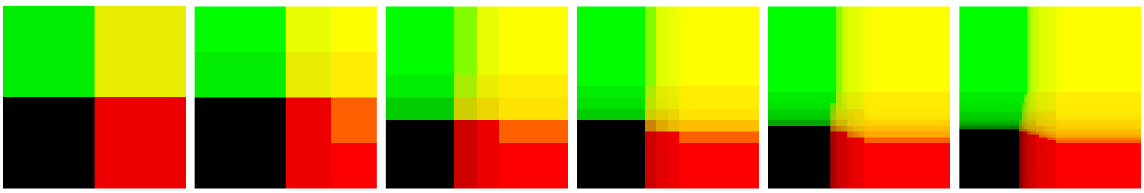
Moreover, since we just evaluate $s^{[1]}$ on a grid, we might miss very slim regions located between nodes. Finally, GLSL is also not flexible enough to allow arbitrarily sized matrices/vectors. We have therefore fixed ourselves to the simple and easily visualizable case of 2 neurons in the first layer.

Another limitation is the storage requirement of the algorithm: If we want to make sure not to miss regions with width of smaller than $\mathrm{diam}_\Omega(C) \cdot 0.00025 \approx \mathrm{diam}_\Omega(C) \cdot 2^{-12}$ 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 $2^{27}\mathrm{B} = 128\mathrm{MiB}$ 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 current hardware limits for working memory.

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 representing 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 overhead. Thus the interface 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. Instead, we have implemented the region counting in Python and GLSL. 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 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 $O(n)$. 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[5].



(a) 2x2 **(b)** 4x4 **(c)** 8x8 **(d)** 16x16 **(e)** 32x32 **(f)** 64x64

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 positive side-effect is that we are additionally getting results for smaller grid sizes, which is a good indication for how much our algorithm is undercounting the regions due to too narrow grids.

[5] Or don't exist since the pixel is located at the left/lower border of the grid.

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 helpful notation and a few technical lemmas. In Section 3 we showed that R-LIF-SNN allow more compact and efficient approximation than d.t. LIF-SNN of a certain class of differentiable continuous functions.

In Section 4 we showed that the graph of R-LIF-SNNs 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 fit data inside of a square. While we were unable to prove it in Section 4, we are expecting to eventually find a proof 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 reason we continued our analysis in Section 5, where we described two algorithms to generate empirical data as well as visualizations to obtain an 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 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 if R-LIF-SNN are indeed better suited for machine learning and in particular for reasoning than current approaches. Since this thesis focused on theoretic aspects, further research investigating the practicality of the model is in particular needed.

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.

```
import argparse
from datatypes import SNN
import numpy as np


def parse_array(arg):
    """Convert a comma-separated string into a numpy array or 2D array."""
    if ";" in arg:  # 2D array
        rows = arg.split(";")
        return np.array([list(map(float, row.split(","))) for row in rows])
    else:  # 1D array
        return np.array(list(map(float, arg.split(","))))


def parseCMDLine():
    # Default values
    default_V = np.array([[0.0, 0.0], [0.0, 0.0]])
    default_T = 10
    default_i0 = np.array([0.0, 0.0])
    default_u0 = np.array([0.0, 0.0])
    default_b = np.array([0.0, 0.0])
    default_theta = 1.0
    default_alpha = 0.0
    default_beta = 1.0
    default_show = False
    default_use_cpp = False

    # Argument parser
    parser = argparse.ArgumentParser(
        description="Read neural network parameters from command line."
    )
    parser.add_argument(
        "--V",
        type=parse_array,
        default=default_V,
        help="matrix, rows separated by ';', elements by ','",
    )
    parser.add_argument("--T", type=int, default=default_T, help="Simulation steps")
    parser.add_argument(
        "--i0",
        type=parse_array,
        default=default_i0,
        help="Initial input, elements separated by ','",
    )
    parser.add_argument(
        "--u0",
        type=parse_array,
        default=default_u0,
        help="Initial membrane potential, elements separated by ','",
    )
```

```
    parser.add_argument(
        "-b",
        type=parse_array,
        default=default_b,
        help="Bias vector, elements separated by ','",
    )
    parser.add_argument(
        "-theta", type=float, default=default_theta, help="Threshold value"
    )
    parser.add_argument(
        "-alpha", type=float, default=default_alpha, help="Input decay rate"
    )
    parser.add_argument(
        "-beta", type=float, default=default_beta, help="Membrane potential decay rate"
    )
    parser.add_argument(
        "--show",
        action="store_true",
        default=default_show,
        help="Show visualization of regions",
    )
    parser.add_argument(
        "--use-cpp",
        action="store_true",
        default=default_use_cpp,
        help="Use quick C++ implementation",
    )

    args = parser.parse_args()

    neurons_n = len(args.u0)

    assert (
        args.i0.shape[0] == neurons_n
    ), f"i0 length {args.i0.shape[0]} does not match neurons_n {neurons_n}"
    assert (
        args.b.shape[0] == neurons_n
    ), f"b length {args.b.shape[0]} does not match neurons_n {neurons_n}"
    assert args.V.shape == (
        neurons_n,
        neurons_n,
    ), f"V shape {args.V.shape} does not match (neurons_n, neurons_n) = ({neurons_n},{neurons_n})"
    assert (
        (neurons_n == 2) if args.show else True
    ), "Visualization only supported for 2 neurons"
    assert (
        args.show and args.use_cpp
    ) == False, "C++ implementation does not support visualization"

    snn = SNN(
        i0=args.i0,
        u0=args.u0,
        V=args.V,
        b=args.b,
        theta=args.theta,
```

```
        alpha=args.alpha,
        beta=args.beta,
        T=args.T,
        neurons_n=neurons_n,
    )
    return snn, args.show, args.use_cpp
```

Code 1: src/args.py

```
import numpy as np
import math
from helper import all_subsets


def g(snn, t, st):
    i0, u0, V, b, theta, alpha, beta = (
        snn.i0,
        snn.u0,
        snn.V,
        snn.b,
        snn.theta,
        snn.alpha,
        snn.beta,
    )
    return -(
        sum(
            beta ** (t - k)
            * (
                alpha**k * i0
                + b
                + sum(alpha ** (k - l) * V @ st[l - 1] for l in range(1, k + 1))
            )
            for k in range(1, t + 1)
        )
        + beta**t * u0
        - theta * (1 + sum(beta ** (t - k) * st[k] for k in range(1, t)))
    ) / sum(
        beta ** (t - k) * sum(alpha ** (k - l) for l in range(1, k + 1))
        for k in range(1, t + 1)
    )


def get_finite_vertex_bounds(snn):
    lB = ()
    uB = ()
    for i in range(snn.neurons_n):
        sLower = (np.full_like(snn.u0, False),) + tuple(
            np.array([i != j and snn.V[i][j] > 0 for j in range(snn.neurons_n)])
            for t in range(1, snn.T + 1)
        )
        sUpper = (np.full_like(snn.u0, False),) + tuple(
            np.array([i == j or snn.V[i][j] < 0 for j in range(snn.neurons_n)])
            for t in range(1, snn.T + 1)
        )
        lB += (min([g(snn, t, sLower)[i] for t in range(1, snn.T + 1)]),)
```

```
        uB += (max([g(snn, t, sUpper)[i] for t in range(1, snn.T + 1)]),)
    return np.array(lB), np.array(uB)


def compute_regions_starting_with(snn, st, lowerBound, upperBound):
    t = len(st)
    if t > snn.T:
        return [(st, lowerBound, upperBound)]
    x = g(snn, t, st)
    swappingNeurons = ((lowerBound < x) & (x < upperBound)).nonzero()[0]
    alwaysActiveNeurons = x <= lowerBound
    regions_n = []
    for nowActive in all_subsets(swappingNeurons):
        inactive = np.setdiff1d(swappingNeurons, nowActive)
        activeNeurons = alwaysActiveNeurons.copy()
        activeNeurons[list(nowActive)] = True

        newSt = st + (activeNeurons,)
        newLowerBound = lowerBound.copy()
        newLowerBound[list(nowActive)] = x[list(nowActive)]
        newUpperBound = upperBound.copy()
        newUpperBound[inactive] = x[inactive]
        regions_n += compute_regions_starting_with(
            snn, newSt, newLowerBound, newUpperBound
        )
    return regions_n


def compute_regions(snn):
    return compute_regions_starting_with(
        snn,
        (np.full_like(snn.u0, False),),
        np.full_like(snn.u0, -math.inf),
        np.full_like(snn.u0, math.inf),
    )
```

Code 2: src/compute_regions_py.py

```
from dataclasses import dataclass
import numpy as np


@dataclass
class SNN:
    i0: np.ndarray
    u0: np.ndarray
    V: np.ndarray
    b: np.ndarray
    theta: float
    alpha: float
    beta: float
    T: int
    neurons_n: int
```

Code 3: src/datatypes.py

```
from itertools import chain, combinations

```

```python

def geometric_sum(ratio, n):
    if ratio == 1:
        return n + 1
    return (1 - ratio ** (n + 1)) / (1 - ratio)


def all_subsets(iterable):
    items = list(iterable)
    return chain.from_iterable(combinations(items, r) for r in range(len(
items) + 1))
```

Code 4: src/helper.py

```python
import numpy as np
import math
import matplotlib.pyplot as plt
from matplotlib.patches import Rectangle
import matplotlib.colors as mcolors
import compute_regions_cpp
import compute_regions_py
from args import parseCMDLine


def check_res(res):
    sts = [tuple(tuple(s) for s in st) for st, _, _ in res]
    assert len(sts) == len(set(sts)), "Duplicate states found"


def visualize(snn, res):
    def repl_inf(it):
        return tuple(
            (-1e10 if np.isneginf(x) else (1e10 if np.isposinf(x) else x))
for x in it
        )

    def uint2colScal(x):
        n = 4
        return (1 / n) * math.log(1 + (math.exp(n) - 1) * x / math.pow(2,
snn.T))

    fig, ax = plt.subplots()

    for st, lowerBound, upperBound in res:
        lowerBound = repl_inf(lowerBound)
        upperBound = repl_inf(upperBound)
        width = upperBound[0] - lowerBound[0]
        height = upperBound[1] - lowerBound[1]
        if width < 0.0001 or height < 0.0001:
            print("highlighting small region", st, lowerBound, upperBound)
            ax.add_patch(
                Rectangle(
                    lowerBound,
                    max(width, 0.01),
                    max(height, 0.01),
                    alpha=0.8,
                    facecolor="white",
                    zorder=10,
```

```
                )
            )
        else:
            num_st = map(
                lambda sti: sti.dot(1 << np.arange(sti.size)[::-1]),
                np.array(st).transpose(),
            )
            hex_color = mcolors.to_hex(tuple(uint2colScal(x) for x in num_st) + (0,))
            ax.add_patch(
                Rectangle(
                    lowerBound,
                    width,
                    height,
                    alpha=0.7,
                    facecolor=hex_color,
                    edgecolor="black",
                    zorder=1,
                )
            )

    lB, uB = compute_regions_py.get_finite_vertex_bounds(snn)
    import matplotlib.patches as patches

    rect = patches.Rectangle(
        lB, uB[0] - lB[0], uB[1] - lB[1], edgecolor="blue", facecolor="none"
    )
    ax.add_patch(rect)
    ax.set_xlim(lB[0] - 0.5, uB[0] + 0.5)
    ax.set_ylim(lB[1] - 0.5, uB[1] + 0.5)
    ax.set_aspect("equal", adjustable="box")
    plt.show()


if __name__ == "__main__":
    snn, show, use_cpp = parseCMDLine()
    if use_cpp:
        snnConfig = compute_regions_cpp.SNNConfig(
            snn.i0,
            snn.u0,
            snn.V,
            snn.b,
            snn.theta,
            snn.alpha,
            snn.beta,
            snn.T,
            snn.neurons_n,
        )
        n = compute_regions_cpp.compute_regions(snnConfig)
        print(f"Regions: {n}")
    else:
        res = compute_regions_py.compute_regions(snn)

        check_res(res)

        print(f"Regions: {len(res)}")

```

```
        if show:
            visualize(snn, res)
```

Code 5: src/main.py

```
import numpy as np


def first_layer(u0,  , W, b, V,  , x):
    u = u0
    s = np.zeros_like(u0)
    while True:
        u =   * u + W @ x + b + V @ s -   * s
        s = (u >=  ).astype(float)
        yield u, s
```

Code 6: src/regions.py

```
from setuptools import setup, Extension
from setuptools.command.build_ext import build_ext
import pybind11


class BuildExt(build_ext):
    """Custom build extension to set compiler flags."""

    def build_extensions(self):
        c = self.compiler.compiler_type
        opts = []
        if c == "unix":
            opts = [
                "-O3",
                "-march=native",
                "-flto",
                "-fno-math-errno",
                "-fPIC",
                "-DNDEBUG",
            ]
            for ext in self.extensions:
                ext.extra_compile_args = opts
        build_ext.build_extensions(self)


print([pybind11.get_include()])
ext_modules = [
    Extension(
        "compute_regions_cpp",
        ["compute_regions_cpp.cpp"],
        include_dirs=[pybind11.get_include()],
        language="c++",
    )
]

setup(
    name="compute_regions_cpp",
    version="0.1",
    ext_modules=ext_modules,
    cmdclass={"build_ext": BuildExt},
```

```
)

# build with
# python setup.py build_ext --inplace
```

Code 7: src/setup.py

```
#define EIGEN_NO_DEBUG
#include <Eigen/Dense>
#include <iomanip>
#include <iostream>
#include <limits>
#include <pybind11/eigen.h>
#include <pybind11/pybind11.h>
#include <pybind11/stl.h>

namespace py = pybind11;

using std::vector;
namespace E = Eigen;

struct SNNConfig {
  E::VectorXd i0;
  E::VectorXd u0;
  E::MatrixXd V;
  E::VectorXd b;
  double theta;
  double alpha;
  double beta;
  size_t T;
  size_t neurons_n;
};

const double pos_inf = std::numeric_limits<double>::infinity();
const double neg_inf = -std::numeric_limits<double>::infinity();

double geometric_series(double r, size_t n) {
  if (r == 1) {
    return n + 1;
  } else {
    return (1 - pow(r, n + 1)) / (1 - r);
  }
}

template <typename Derived> std::vector<int> where(const Eigen::ArrayBase<
    Derived> &mask) {
  static_assert(std::is_same<typename Derived::Scalar, bool>::value,
                "where() requires a boolean Array");

  std::vector<int> indices;
  indices.reserve(mask.size());

  for (int i = 0; i < mask.size(); ++i) {
    if (mask(i)) {
      indices.push_back(i);
    }
  }
  return indices;
```

```cpp
}

size_t __compute_regions(const SNNConfig &conf) {
  const vector<double> gDiv = ([&] {
    vector<double> gDiv;
    gDiv.push_back(-1);
    for (size_t t = 1; t <= conf.T; ++t) {
      double sum = 0;
      for (size_t k = 1; k <= t; ++k) {
        double innerSum = 0;
        for (size_t l = 1; l <= k; ++l) {
          innerSum += pow(conf.alpha, k - l);
        }
        sum += pow(conf.beta, t - k) * innerSum;
      }
      gDiv.push_back(-1 / sum);
    }
    return gDiv;
  })();
  const vector<E::VectorXd> tDep = ([&] {
    vector<E::VectorXd> tDep;
    for (size_t t = 0; t <= conf.T; ++t) {
      tDep.push_back(pow(conf.beta, t) * conf.u0 - conf.theta * E::VectorXd::Ones(conf.neurons_n));
    }
    return tDep;
  })();

  auto run = [&](this auto self, const SNNConfig &conf, vector<E::VectorXd> &st,
                 const E::VectorXd &inputOffset, const E::VectorXd &inputVOffset,
                 const E::VectorXd &spikeSum, const E::VectorXd &lowerBound,
                 const E::VectorXd &upperBound) -> size_t {
    const auto t = st.size();
    if (t == 0) {
      throw std::invalid_argument("Spike time vector is empty.");
    }

    if (t > conf.T) {
      return 1;
    }

    const auto x = (inputOffset + tDep[t] - (conf.theta * conf.beta) * spikeSum) * gDiv[t];

    const auto swappingNeurons =
        (lowerBound.array() < x.array()) && (x.array() < upperBound.array());
    const auto swappingNeuronsIndices = where(swappingNeurons);
    const E::ArrayXd alwaysActiveNeurons = (x.array() <= lowerBound.array()).cast<double>();

    auto regions_n = 0;

    if (swappingNeuronsIndices.size() >= sizeof(unsigned long long) * 8) {
      throw std::runtime_error("Too many swapping neurons, cannot compute
```

```
regions.");
    }
    for (unsigned long long i = 0; i < (1ull << swappingNeuronsIndices.size()); ++i) {
      E::VectorXd newLowerBound = lowerBound;
      E::VectorXd newUpperBound = upperBound;
      E::VectorXd newSpike = alwaysActiveNeurons.matrix();

      for (size_t j = 0; j < swappingNeuronsIndices.size(); ++j) {
        auto sjI = swappingNeuronsIndices[j];
        if ((i >> j) & 1) {
          newLowerBound(sjI) = x(sjI);
          newSpike(sjI) = 1;
        } else {
          newUpperBound(sjI) = x(sjI);
        }
      }

      // TODO: compute after branch instead
      const E::VectorXd newSpikeSum = conf.beta * spikeSum + newSpike;
      const E::VectorXd newInputVOffset = conf.alpha * inputVOffset + conf.V * newSpike;
      const E::VectorXd newInputOffset =
          conf.beta * inputOffset + pow(conf.alpha, t + 1) * conf.i0 + conf.b + newInputVOffset;
      st.push_back(std::move(newSpike));
      regions_n += self(conf, st, newInputOffset, newInputVOffset, newSpikeSum, newLowerBound,
                        newUpperBound);
      st.pop_back();
    }

    return regions_n;
  };

  const E::VectorXd initSpikeSum = E::VectorXd::Zero(conf.neurons_n);
  const E::VectorXd initInputOffset = conf.alpha * conf.i0 + conf.b;
  const E::VectorXd initInputVOffset = E::VectorXd::Zero(conf.neurons_n);
  vector<E::VectorXd> initSt = {E::VectorXd::Zero(conf.neurons_n)};
  return run(conf, initSt, initInputOffset, initInputVOffset, initSpikeSum,
             E::VectorXd::Constant(conf.neurons_n, neg_inf),
             E::VectorXd::Constant(conf.neurons_n, pos_inf));
}

size_t compute_regions(const SNNConfig &conf) {
  if (conf.V.rows() != conf.neurons_n || conf.V.rows() != conf.V.cols()) {
    throw std::invalid_argument("Malformed matrix V");
  }

  if (conf.u0.size() != conf.neurons_n) {
    throw std::invalid_argument("Malformed vector u0");
  }

  vector<E::VectorXd> initSt = {E::VectorXd::Zero(conf.neurons_n)};
  return __compute_regions(conf);
}

PYBIND11_MODULE(compute_regions_cpp, m) {
```

```
  m.doc() = "Module for computing regions of dt lif snns ";
  m.def("compute_regions", &compute_regions, py::arg("conf"),
        "Compute number of regions of r dt lif snns.");
  py::class_<SNNConfig>(m, "SNNConfig")
      .def(py::init<E::VectorXd, E::VectorXd, E::MatrixXd, E::VectorXd,
   double, double, double,
                    size_t, size_t>())
      .def_readwrite("i0", &SNNConfig::i0)
      .def_readwrite("u0", &SNNConfig::u0)
      .def_readwrite("V", &SNNConfig::V)
      .def_readwrite("b", &SNNConfig::b)
      .def_readwrite("theta", &SNNConfig::theta)
      .def_readwrite("alpha", &SNNConfig::alpha)
      .def_readwrite("beta", &SNNConfig::beta)
      .def_readwrite("T", &SNNConfig::T)
      .def_readwrite("neurons_n", &SNNConfig::neurons_n);
}
```

Code 8: src/compute_regions_cpp.cpp

7.2 dt-lif-snn-compute-regions

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

```
from contextlib import contextmanager
import time
import moderngl
from pathlib import Path


@contextmanager
def measure_perf(label):
    start = time.perf_counter()
    try:
        yield ()
    finally:
        end = time.perf_counter()
        print(f"Elapsed time ({label}): {end - start:.6f} seconds")


@contextmanager
def measure_perf_gpu(ctx, label):
    timer = ctx.query(time=True)
    try:
        with timer:
            yield ()
    finally:
        elapsed_ns = timer.elapsed
        print(f"Elapsed time ({label}): {elapsed_ns/10**9:.6f} seconds")


def mkdirp(path):
    Path(path).mkdir(parents=True, exist_ok=True)


def chunked_iterable(iterable, n):
    """Yield successive n-sized chunks from iterable."""
    for i in range(0, len(iterable), n):
```

```python
        yield iterable[i : i + n]


def getUniformsDict(shader):
    return {
        name: shader[name].value
        for name in shader
        if isinstance(shader[name], moderngl.Uniform)
    }


def compileCompShaderFile(ctx, filename):
    """Compile a compute shader from a file."""
    with open(filename, "r") as f:
        shader_code = f.read()
    return ctx.compute_shader(shader_code)


def getUniformsDictSpliced(shader):
    res = {}
    for name in shader:
        uniform = shader[name]
        if isinstance(uniform, moderngl.Uniform):
            for i in range(uniform.array_length):
                for j in range(uniform.dimension):
                    cpName = f"{name}"
                    value = uniform.value
                    if uniform.array_length > 1:
                        cpName += f"[{i}]"
                        value = value[i]
                    if uniform.dimension > 1:
                        cpName += f"[{j}]"
                        value = value[j]
                    res[cpName] = value
    return res
```

Code 9: src/common.py

```python
from abc import ABC, abstractmethod


class CountRegions(ABC):
    def setUniforms(self, **kwargs):
        """Set uniforms for the shader."""
        for key, value in kwargs.items():
            if key in self.shader:
                self.shader[key] = value
            else:
                raise KeyError(f"Uniform '{key}' not found in shader.")

    @abstractmethod
    def run(self, iterationsR, u0R, betaR, bR, WR, VR, thetaR, scale, offset):
        pass
```

Code 10: src/count_regions.py

```python
import numpy as np
```

```
from itertools import islice
import matplotlib.pyplot as plt


def get_nth_result(generator, n):
    return next(islice(generator, n, n + 1))


def first_layer(u0,  , W, b, V,  , x):
    u = u0
    s = np.zeros_like(u0)
    while True:
        u =  * u + W @ x + b + V @ s -  *  * s
        s = (u >=  ).astype(float)
        yield u, s


def run_with(V, x, W=[[1, 0], [0, 1]], b=None,  =1, u0=None,  =1):
    if u0 is None:
        u0 = np.zeros_like(x)
    if b is None:
        b = np.zeros_like(x)
    if isinstance(W, list):
        W = np.array(W)
    if isinstance(V, list):
        V = np.array(V)
    if isinstance(b, list):
        b = np.array(b)
    if isinstance(x, list):
        x = np.array(x)
    if isinstance(u0, list):
        u0 = np.array(u0)

    for u, s in first_layer(u0,  , W, b, V,  , x):
        print("Membrane potential:", u)
        print("Spikes:", s)
        input()


def spiketrain_to_nextspike(spiketrain, V,  =1, u0=None):
    # we assume b=0 and W =
    if u0 is None:
        u0 = np.zeros_like(V[0])
    if isinstance(u0, list):
        u0 = np.array(u0)
    lower_bounds = [[] for _ in range(len(u0))]
    upper_bounds = [[] for _ in range(len(u0))]
    stW0 = np.append([[0, 0]], spiketrain, axis=0)
    one_n = np.ones_like(u0)
    for t, spike in enumerate(stW0):
        if t == 0:
            continue
        # we compute x, so that u(t) =
        stSum = np.sum(stW0[:t], axis=0)
        xCut = ( * one_n - u0 - (V @ stSum -  * stSum)) / t
        for i, x in enumerate(xCut):
            if spike[i] == 1.0:
                lower_bounds[i].append(x)
```

```
        else:
            upper_bounds[i].append(x)
# print("Lower bounds:", lower_bounds)
# print("Upper bounds:", upper_bounds)
infimums = np.array([max(lb) if lb else -np.inf for lb in lower_bounds])
# print("Highest lower bound:", infimums)
supremums = np.array([min(ub) if ub else np.inf for ub in upper_bounds])
# print("Lowest upper bound:", supremums)

def get_continuation_us(x):
    return list(
        islice(
            first_layer(
                u0=u0, =1, W=[[1, 0], [0, 1]], b=np.zeros_like(u0), V=V, =, x=x
            ),
            0,
            len(spiketrain) + 1,
        )
    )

check_if_spikes_fit = lambda stRes: spiketrain == list(
    map(lambda x: list(x[1]), stRes[:-1])
)

# The spike trains at infimums might not be correct,
# since we can't properly compute with ∞-.
# Also the supremums are probably not correct,
# since they should be the highest value that just produces another spike train.
def find_actual_possible(x, other):
    y = np.copy(x)
    while np.all((x < other) == (y < other)) and np.all((other < x) == (other < y)):
        stRes = get_continuation_us(x)
        if check_if_spikes_fit(stRes):
            return stRes
        x = np.nextafter(x, other)
        print(f"trying next {x}")
    else:
        raise ValueError(f"no possible next value for {x} with other {other}")

# print(f"checking spikes for infimums")
# FIXME: wrong for negative weights in V
stInf = find_actual_possible(infimums, supremums)
uInf, sInf = stInf[-1]

# print(f"checking spikes for supremums")
stSup = find_actual_possible(supremums, infimums)
uSup, sSup = stSup[-1]

assert check_if_spikes_fit(stSup)
if np.array_equal(sInf, sSup):
    # print(f"next spikes are fixed to {sInf}")
    return {"next": [sInf], "infs": infimums, "sups": supremums}
```

```
    else:
        # print(f"next spikes might be {sInf} or {sSup}")
        return {"next": [sInf, sSup], "infs": infimums, "sups": supremums}


def play(startSpikes, V,  , u0):
    spiketrain = [startSpikes]
    forcedp = [False]
    while True:
        result = spiketrain_to_nextspike(spiketrain, V,  , u0)
        forcedp.append(len(result["next"]) == 1)
        print("Infimums:", result["infs"])
        print("Supremums:", result["sups"])
        print(
            "Spikes:\n"
            + "\n".join(
                map(
                    lambda n: "".join(map(lambda x: "1" if x == 1 else "0", n)),
                    zip(*spiketrain),
                )
            )
            + "\n"
            + "".join(["^" if f else " " for f in forcedp])
        )
        print("Possible next spikes:", result["next"])
        choice = input("Choose next spikes (or 'q' to quit): ")
        if choice.lower() == "q":
            break
        try:
            next_spikes = result["next"][int(choice)]
            spiketrain.append(list(next_spikes))
        except (ValueError, IndexError):
            print("Invalid input, please enter q or an integer.")


def plot(V, n, lower=-1, upper=1, steps=100):
    x = np.linspace(lower, upper, steps)
    y = np.linspace(lower, upper, steps)

    X, Y = np.meshgrid(x, y)

    points = np.stack([X.ravel(), Y.ravel()], axis=1)
    g = lambda p: list(
        islice(
            first_layer(
                u0=np.zeros_like(p),
                 =1,
                W=[[1, 0], [0, 1]],
                b=np.zeros_like(p),
                V=V,
                 =1,
                x=p,
            ),
            0,
            n,
        )
    )
```

```
    Z = np.array([g(p) for p in points])
    Z = Z.reshape(X.shape)

    plt.figure(figsize=(6, 6))
    plt.pcolormesh(X, Y, Z, cmap="Set2", shading="auto")
    plt.colorbar(label="Group")
    plt.title("Grouped Grid Function Output")
    plt.xlabel("x")
    plt.ylabel("y")
    plt.axis("equal")
    plt.show()
```

Code 11: src/regions.py

```
from setuptools import setup, Extension
from setuptools.command.build_ext import build_ext
import pybind11


class BuildExt(build_ext):
    """Custom build extension to set compiler flags."""

    def build_extensions(self):
        c = self.compiler.compiler_type
        opts = []
        if c == "unix":
            opts = ["-O3", "-fno-math-errno", "-fPIC", "-DNDEBUG"]
            for ext in self.extensions:
                ext.extra_compile_args = opts
        build_ext.build_extensions(self)


print([pybind11.get_include()])
ext_modules = [
    Extension(
        "unique_bytes",
        ["unique_bytes.cpp"],
        include_dirs=[pybind11.get_include()],
        language="c++",
    )
]

setup(
    name="unique_bytes",
    version="0.1",
    ext_modules=ext_modules,
    cmdclass={"build_ext": BuildExt},
)

# build with
# python setup.py build_ext --inplace
```

Code 12: src/setup.py

```
import moderngl
import time
from contextlib import contextmanager

```

```python
resSize = 4


@contextmanager
def measure_perf(label):
    start = time.perf_counter()
    try:
        yield ()
    finally:
        end = time.perf_counter()
        print(f"Elapsed time ({label}): {end - start:.6f} seconds")


def is_power_of_two(n):
    return n > 0 and (n & (n - 1)) == 0


# gven buffer has to be bound to binding=0
def count_unique(buffer, ctx=None):
    copyBuffer = False
    if ctx is None:
        copyBuffer = True
        ctx = moderngl.create_standalone_context()
    with measure_perf(f"Count regions inner"):
        valSize = resSize * 4  # resSize many uints
        dataSize = buffer.size
        print("DataSize:", dataSize)
        if dataSize % valSize != 0:
            raise ValueError("Data length must be a multiple of 4.")
        if not is_power_of_two(dataSize // valSize):
            raise ValueError("Data length must be a power of two.")
        if copyBuffer:
            with measure_perf(f"reserve buffer"):
                bufData = ctx.buffer(buffer.read())
                # bufData = ctx.buffer(reserve=buffer.size)
            # with measure_perf(f"copy buffer"):
            # ctx.copy_buffer(bufData, buffer)
            with measure_perf(f"bind buffer"):
                bufData.bind_to_storage_buffer(binding=0)
        else:
            bufData = buffer
        with measure_perf(f"compute shader"):
            with open("shaders/uniques.glsl", "r") as f:
                shader = ctx.compute_shader(f.read())

        with measure_perf(f"shader2"):
            resNum = ctx.buffer(reserve=4)  # one uint
            resNum.bind_to_storage_buffer(binding=1)

        with measure_perf(f"run"):
            i = 1
            while 2**i * valSize < dataSize:
                prevBatchSize = 2**i
                shader["prevBatchSize"] = prevBatchSize

                batchSize = 2 * prevBatchSize
                shader.run(group_x=dataSize // (valSize * batchSize))
                i += 1
```

```python

        with measure_perf(f"read"):
            res = int.from_bytes(resNum.read(), byteorder="little", signed=False)
            resNum.release()
            if copyBuffer:
                bufData.release()
            return res
```

Code 13: src/uniques.py

```python
import matplotlib.pyplot as plt
import numpy as np
import glob

X = np.arange(32)

plt.rcParams.update({"font.size": 14})

for file_path in glob.glob("/tmp/output.csv"):
    data = np.loadtxt(file_path, delimiter=",")
    # plt.plot(X, data[:32], label="#Counted Regions")

for i in range(4, 7):
    plt.plot(X, data / X**i, label=f"Regions / T^{i}")

# plt.plot(X, (((X**2 + X + 2) / 2) ** 2), label=f"(T² + T + 2)² / 2²")

plt.title("")
# plt.yscale("log")
plt.xlabel("T")
plt.ylabel("#Regions")
plt.legend(loc="upper left")
# plt.ylim(0, 3)
plt.grid(True)
plt.show()
```

Code 14: src/analyze.py

```python
import moderngl
import numpy as np
from common import getUniformsDict, getUniformsDictSpliced, compileCompShaderFile
from count_regions import CountRegions
import itertools
import unique_bytes


class CountRegionsEfficient(CountRegions):
    def __init__(self, imagep, spikeTrp, maxSizePot2):
        # Create context (offscreen)
        self.imagep = imagep
        self.spikeTrp = spikeTrp
        self.ctx = moderngl.create_standalone_context(backend="egl")
        self.shader = compileCompShaderFile(self.ctx, "shaders/uintEfficient.glsl")
        self.shaderCorners = compileCompShaderFile(self.ctx, "shaders/corners.glsl")
```

```

        self.resSize = 4
        assert maxSizePot2 >= 5
        self.maxSizeI = maxSizePot2 - 5
        self.initialSize = 2**5
        self.scalePrev = 2

        self.bufData = []
        self.bufColor = []
        self.bufPrev = []
        self.bufRegionsN = self.ctx.buffer(reserve=4)
        self.bufRegionsN.bind_to_storage_buffer(binding=3)
        for i in range(0, self.maxSizeI + 1):
            size = self.initialSize * self.scalePrev**i
            self.bufData.append(self.ctx.buffer(reserve=size * size * self.resSize * 4))
            self.bufColor.append(self.ctx.buffer(reserve=size * size * 4))
            prevSize = self.initialSize * self.scalePrev ** (i - 1) if i > 0 else 1
            self.bufPrev.append(
                self.ctx.buffer(reserve=prevSize * prevSize * self.resSize * 4)
            )

    def run(self, iterationsR, u0R, betaR, bR, WR, VR, thetaR, scale, offset):
        ctx = self.ctx
        shader = self.shader

        for iterations, u0, beta, b, W, V, theta in itertools.product(
            iterationsR, u0R, betaR, bR, WR, VR, thetaR
        ):
            for i in range(0, self.maxSizeI + 1):
                size = self.initialSize * self.scalePrev**i
                print(size)

                self.bufData[i].bind_to_storage_buffer(binding=0)
                self.bufColor[i].bind_to_storage_buffer(binding=1)
                self.bufPrev[i].bind_to_storage_buffer(binding=2)

                initialRun = True  # i == 0
                self.setUniforms(
                    iResolution=(size, size),
                    iterations=iterations,
                    u0=u0,
                    beta=beta,
                    # b=b,
                    # W=W,
                    V=V,
                    theta=theta,
                    scale=scale,
                    offset=offset,
                    initialRun=initialRun,
                    scalePrev=self.scalePrev,
                    imagep=self.imagep,
                )

                assert (
```

```python
                    size % 8 == 0
                ), "size must be multiple of 8; if you want to lower this, you need to amend the shader"
                groupSize = size // 8
                shader.run(group_x=groupSize, group_y=groupSize)

                assert self.resSize == 4
                self.bufRegionsN.write(np.array([0], dtype=np.uint32))
                self.shaderCorners["iResolution"] = size, size
                self.shaderCorners.run(group_x=groupSize, group_y=groupSize)
                regions_n = np.frombuffer(self.bufRegionsN.read(), dtype=np.uint32)[0]

                print(f"Regions for {size}:", regions_n)

                if i != self.maxSizeI:
                    ctx.copy_buffer(self.bufPrev[i + 1], self.bufData[i])

                if self.spikeTrp:
                    spikeTrains = unique_bytes.unique_regions(
                        self.bufData[i].read(), size, self.resSize * 4
                    )

                    for spiketrain in spikeTrains:
                        rs = self.resSize
                        # fmt: off
                        print("x:", "".join(f"{byte:08b}"[::-1] for byte in spiketrain[0 : rs * 2])[:iterations][::-1],
                              "y:", "".join(f"{byte:08b}"[::-1] for byte in spiketrain[rs * 2 : rs * 4])[:iterations][::-1])
                        # fmt: on

                imageArray = None
                if self.imagep:
                    imageArray = np.frombuffer(
                        self.bufColor[i].read()[0 : size * size * 4], dtype=np.byte
                    ).reshape((size, size, 4))

                uniforms = getUniformsDict(shader)
                uniformsSpliced = getUniformsDictSpliced(shader)
                yield regions_n, imageArray, uniforms, uniformsSpliced
```

Code 15: src/count_regions_efficient.py

```python
import moderngl
import numpy as np
import unique_bytes
import itertools
from common import getUniformsDict, getUniformsDictSpliced, compileCompShaderFile
from count_regions import CountRegions


class CountRegionsSimple(CountRegions):
    def __init__(self, imagep, spikeTrp, sizeR):
        # Create context (offscreen)
```

```
        self.imagep = imagep
        self.spikeTrp = spikeTrp
        self.ctx = moderngl.create_standalone_context(backend="egl")
        self.shader = compileCompShaderFile(self.ctx, "shaders/uint.glsl")
        self.sizeR = sizeR

    def setUniforms(self, **kwargs):
        """Set uniforms for the shader."""
        for key, value in kwargs.items():
            if key in self.shader:
                self.shader[key] = value
            else:
                raise KeyError(f"Uniform '{key}' not found in shader.")

    def getSpiketrains(self, bufData, size, resSize):
        """Count unique chunks in the data buffer."""
        return unique_bytes.unique_regions(bufData.read(), size, resSize * 4)

    def printSpikeTrains(self, regionsSt, resSize, iterations):
        """Print spiketrains from the data buffer."""
        for spiketrain in regionsSt:
            rs = resSize
            # fmt: off
            print("x:", "".join(f"{byte:08b}"[::-1] for byte in spiketrain[0      : rs * 2])[:iterations][::-1],
                  "y:", "".join(f"{byte:08b}"[::-1] for byte in spiketrain[rs * 2 : rs * 4])[:iterations][::-1])
            # fmt: on

    def countUniqueChunks(self, bufData, size, resSize):
        """Count unique chunks in the data buffer."""
        return len(self.getSpiketrains(bufData, size, resSize))

    def run(self, iterationsR, u0R, betaR, bR, WR, VR, thetaR, scale, offset):
        ctx = self.ctx
        shader = self.shader
        for size in self.sizeR:
            resSize = 4

            # size * size * resSize many uints
            bufData = ctx.buffer(reserve=size * size * resSize * 4)
            bufColor = ctx.buffer(reserve=size * size * 4)

            # Bind buffer to binding=0
            bufData.bind_to_storage_buffer(binding=0)
            bufColor.bind_to_storage_buffer(binding=1)

            for iterations, u0, beta, b, W, V, theta in itertools.product(
                iterationsR, u0R, betaR, bR, WR, VR, thetaR
            ):
                self.setUniforms(
                    iResolution=(size, size),
                    iterations=iterations,
                    u0=u0,
                    beta=beta,
                    b=b,
```

```
                    W=W,
                    V=V,
                    theta=theta,
                    scale=scale,
                    offset=offset,
                )

                assert size % 32 == 0, "Size must be a multiple of 32"
                shader.run(group_x=size // 32, group_y=size // 32)

                regions_n = self.countUniqueChunks(bufData, size, resSize)
                if self.spikeTrp:
                    regionsSt = self.getSpiketrains(bufData, size, resSize)
                    if len(regionsSt) > 0:
                        self.printSpikeTrains(regionsSt, resSize, iterations)

                imageArray = None
                if self.imagep:
                    imageArray = np.frombuffer(bufColor.read(), dtype=np.byte).reshape(
                        (size, size, 4)
                    )

                uniforms = getUniformsDict(shader)
                uniformsSpliced = getUniformsDictSpliced(shader)
                yield regions_n, imageArray, uniforms, uniformsSpliced
```

Code 16: src/count_regions_simple.py

```
import numpy as np
from PIL import Image
import sys
import csv
import itertools
from count_regions_simple import CountRegionsSimple
from count_regions_efficient import CountRegionsEfficient

## args
efficientp = "--efficient" in sys.argv[1:]
imagep = "--image" in sys.argv[1:]
spikeTrp = "--spiketrain" in sys.argv[1:]


def once(v):
    yield v


ls = np.linspace
itpr = itertools.product
# iterationsR = range(10, 20, 3)
# uOR = itpr(ls(0.0123456789, 1, 3), ls(0.0123456789, 1, 3))
# betaR = ls(0.9, 1, 10)
# WR = once((1.0, 0.0, 0.0, 1.0))  # doesn't increase number of regions if changed
# bR = once((0.0, 0.0))  # doesn't increase number of regions if changed
# VR = itpr(once(0), ls(0.1, 0.5, 3), ls(-0.5, 0.5, 5), once(0))
# thetaR = once(1)
```

```
iterationsR = once(10)
u0R = once((0, 0))
betaR = once(0.9)
WR = once((1.0, 0.0, 0.0, 1.0))  # doesn't increase number of regions if changed
bR = once((0.0, 0.0))  # doesn't increase number of regions if changed
VR = once((0.0, 0.3, -0.2, 0.0))
thetaR = once(1)
scale = 4
offset = -1.11263, -1.11263
maxSizePot2 = 12  # size = 2**maxSizePot2


if efficientp:
    counter = CountRegionsEfficient(
        imagep=imagep,
        spikeTrp=spikeTrp,
        maxSizePot2=maxSizePot2,
    )
else:
    sizeR = map(lambda x: 2**x, range(6, maxSizePot2 + 1))
    counter = CountRegionsSimple(imagep=imagep, spikeTrp=spikeTrp, sizeR=sizeR)

counterGen = counter.run(iterationsR, u0R, betaR, bR, WR, VR, thetaR, scale, offset)

regions = []
for regions_n, imageArray, uniforms, uniformsSpliced in counterGen:
    iResolution = uniforms["iResolution"]
    iResolution = (int(iResolution[0]), int(iResolution[1]))
    iterations = uniforms["iterations"]
    print(f"iResolution: {iResolution}, iterations: {iterations}")
    if imagep:
        image = Image.frombytes(
            "RGBA",
            iResolution,
            imageArray,
        )
        image = image.transpose(Image.FLIP_TOP_BOTTOM)
        image.save(
            f"images/output_size{iResolution[0]:04}_{iResolution[1]:04}_iteration{iterations:02}.png"
        )

    regions.append({"regions_n": regions_n} | uniformsSpliced)
    print(f"Regions for {uniformsSpliced}:\n{regions_n}")


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

Code 17: src/main.py

```
#version 330
```

```
out vec4 fragColor;
uniform vec2 iResolution;
uniform int iterations;
uniform vec2 u0;
uniform float beta;
uniform vec2 b;
uniform mat2 W;
uniform mat2 V;
uniform float theta;
uniform vec2 offset;
uniform float scale;

vec2 first_layer(vec2 x, int iterations, vec2 u0, float beta, mat2 W, vec2
    b, mat2 V, float theta)
{
    vec2 u = u0;
    vec2 s = vec2(0,0);
    vec2 col = vec2(0,0);
    for (int i = 0; i < iterations; ++i) {
        u = beta * u + W * x + b + V * s - theta * beta * s;
        s = step(theta,u);
        col = col + pow(2.,float(-i))*s;
    }
    return col;
}


void main()
{
    // Normalized pixel coordinates (from 0 to scale)
    vec2 nuv = gl_FragCoord.xy/iResolution.xy;
    vec2 uv = scale*nuv+offset;

    // Time varying pixel color

    // int iterations = 1;
    // vec2 u0 = vec2(0,0);
    // float beta = 1.;
    // vec2 b = vec2(0,0);
    // mat2 W = mat2(1,0,0,1);
    // mat2 V = mat2(0,0,0,0);
    // float theta = 1.;
    vec2 res = first_layer(uv, iterations, u0, beta, W, b, V, theta);
    vec2 col = log(res+1.)*2.5;

    // Output to screen
    fragColor = vec4(col, 0.0,1.0);
}
```

Code 18: shaders/color.glsl

```
#version 330
out vec4 fragColor;
uniform vec2 iResolution;
uniform int iterations;
uniform vec2 u0;
uniform float beta;
uniform vec2 b;
```

```
uniform mat2 W;
uniform mat2 V;
uniform float theta;
uniform vec2 offset;
uniform float scale;

vec2[2] first_layer(vec2 x, int iterations, vec2 u0, float beta, mat2 W,
    vec2 b, mat2 V, float theta)
{
    vec2 u = u0;
    vec2 s = vec2(0,0);
    vec2 col = vec2(0,0);
    vec2 col2 = vec2(0,0);
    for (int i = 0; i < iterations; ++i) {
        u = beta * u + W * x + b + V * s - theta * beta * s;
        s = step(theta,u);
        col = col + pow(2.,float(-i))*s;
        col2 = col2 + pow(2.,float(i-iterations+1))*s;
    }
    return vec2[2](col, col2);
}


void main()
{
    // Normalized pixel coordinates (from 0 to scale)
    vec2 nuv = gl_FragCoord.xy/iResolution.xy;
    vec2 uv = scale*nuv+offset;

    // Time varying pixel color

    // int iterations = 1;
    // vec2 u0 = vec2(0,0);
    // float beta = 1.;
    // vec2 b = vec2(0,0);
    // mat2 W = mat2(1,0,0,1);
    // mat2 V = mat2(0,0,0,0);
    // float theta = 1.;
    vec2[2] res = first_layer(uv, iterations, u0, beta, W, b, V, theta);

    // Output to screen
    fragColor = vec4(res[0], res[1]);
}
```

Code 19: shaders/color2.glsl

```
#version 460

uniform ivec2 iResolution;

layout (local_size_x = 8, local_size_y = 8, local_size_z = 1) in;

layout(std430, binding = 0) buffer GridData {
    uint[2] data[];
};
layout(std430, binding = 3) buffer RegionsN {
    uint regions_n;
};
```

```
// layout(binding = 3) uniform atomic_uint regions_n;

void main()
{
    int i = int(gl_GlobalInvocationID.x);
    int j = int(gl_GlobalInvocationID.y);

    // Check if the value at the current position is different
    // from the value to the left and below it.
    bool leftIsDiff = i == 0 || (data[(i-1) * iResolution.x + j] != data[i
    * iResolution.x + j]);
    bool belowIsDiff = j == 0 || (data[i * iResolution.x + (j - 1)] != data
    [i * iResolution.x + j]);

    // If it is different indeed, the current position is a ll corner of
    // a region.
    // atomicCounterIncrement(regions_n);
    if (leftIsDiff && belowIsDiff) {
        // regions_n += 1;
        atomicAdd(regions_n, 1);
    }
}
```

Code 20: shaders/corners.glsl

```
#version 460
uniform dvec2 iResolution;
uniform int iterations;
uniform dvec2 u0;
uniform double beta;
uniform dvec2 b;
uniform mat2 W;
uniform mat2 V;
uniform double theta;
uniform dvec2 offset;
uniform double scale;

layout (local_size_x = 32, local_size_y = 32, local_size_z = 1) in;

layout(std430, binding = 0) buffer GridData {
    uint[4] data[];
};
layout(std430, binding = 1) buffer GridColor {
    uint color[];
};

uint[4] first_layer(dvec2 x, int iterations, dvec2 u0, double beta, mat2 W,
     dvec2 b, mat2 V, double theta)
{
    dvec2 u = u0;
    uvec2 s = uvec2(0,0);
    uvec2 spiketrLow = uvec2(0,0);
    uvec2 spiketrUp = uvec2(0,0);
    for (int i = 0; i < iterations; ++i) {
        u = beta * (u - theta * s) + W * x + b + V * s;
        s = uvec2(step(theta,u));
        spiketrUp = (spiketrUp << 1) + (spiketrLow >> 31);
        spiketrLow = (spiketrLow << 1) + s;
```

```
    }
    return uint[4](spiketrLow.x, spiketrUp.x, spiketrLow.y, spiketrUp.y);
}

dvec2 posToUV(uvec2 pos)
{
    dvec2 nuv = dvec2(pos)/iResolution.xy;
    return scale*nuv+offset;
}

uint uint2colScal(uint high, uint low)
{
    float n = 10.;
    float x = float(high) * pow(2., 32.) + float(low);
    double a = ((1. / n) * log(1 + (exp(n) - 1) * x / pow(2., float(
   iterations))));
    // a should be in [0,1] by definition. But just to make sure:
    return min(255, uint(floor(255*a)));
}

uint uint2col(uint[4] res)
{
    return uint2colScal(res[1], res[0]) | (uint2colScal(res[3], res[2]) <<
   8) | uint(0xFF000000);
}

uint uint2colBorder(uint[4] res)
{
    dvec2 aBitRightPos = posToUV(gl_GlobalInvocationID.xy+uvec2(1,0));
    dvec2 aBitUpPos    = posToUV(gl_GlobalInvocationID.xy+uvec2(0,1));
    uint[4] aBitRight = first_layer(aBitRightPos, iterations, u0, beta, W,
   b, V, theta);
    uint[4] aBitUp = first_layer(aBitUpPos, iterations, u0, beta, W, b, V,
   theta);
    if (res != aBitRight || res != aBitUp) {
        return 0xFFFFFFFF; // white
    } else {
        return uint2col(res);
    }
}

void main()
{
    dvec2 uv = posToUV(gl_GlobalInvocationID.xy);
    uint index = uint(gl_GlobalInvocationID.x + gl_GlobalInvocationID.y *
   iResolution.x);

    uint[4] res = first_layer(uv, iterations, u0, beta, W, b, V, theta);

    data[index] = res;
    color[index] = uint2colBorder(res);
}
```

Code 21: shaders/uint.glsl

```
uniform dvec2 iResolution;
uniform int iterations;
uniform dvec2 u0;
```

```
uniform double beta;
// W and b can't be used with this optimization
// uniform dvec2 b;
dvec2 b = dvec2(0.0, 0.0);
// uniform mat2 W;
mat2 W = mat2(1.0, 0.0, 0.0, 1.0);
uniform mat2 V;
uniform double theta;
uniform dvec2 offset;
uniform double scale;
uniform int scalePrev;

layout (local_size_x = 1, local_size_y = 1, local_size_z = 1) in;

layout(std430, binding = 0) buffer GridData {
    uint[2] data[];
};
layout(std430, binding = 1) buffer GridColor {
    uint color[];
};

// data from previous run with 1/scalePrev of size unless INITIAL_RUN is
    set
layout(std430, binding = 2) buffer GridPrevData {
    uint[2] prevData[];
};

uint[2] first_layer(dvec2 x, int iterations, dvec2 u0, double beta, mat2 V,
     double theta)
{
    dvec2 u = u0;
    uvec2 s = uvec2(0,0);
    uvec2 spiketr = uvec2(0,0);
    for (int i = 0; i < iterations; ++i) {
        u = beta * u + W * x + b + V * s - theta * beta * s;
        s = uvec2(step(theta,u));
        spiketr = (spiketr << 1) + s;
    }
    return uint[2](spiketr.x, spiketr.y);
}

uint uint2col(uint x)
{
    float n = 10.;
    float a = ((1. / n) * log(1 + (exp(n) - 1) * x / pow(2., float(
    iterations))));
    // a should be in [0,1] by definition. But just to make sure:
    return min(255, uint(floor(255*a)));
}

void main()
{
    dvec2 nuv = gl_GlobalInvocationID.xy/iResolution.xy;
    uint index = uint(gl_GlobalInvocationID.x + gl_GlobalInvocationID.y *
    iResolution.x);
    dvec2 uv = scale*nuv+offset;

    uint[2] res;
```

```
    #ifdef INITIAL_RUN
    res = first_layer(uv, iterations, u0, beta, V, theta);
    #else
    // points of corners
    ivec2 LDCorner = ivec2(gl_GlobalInvocationID.x / scalePrev,
   gl_GlobalInvocationID.y / scalePrev);
    uint LDCIndex = uint(LDCorner.x + LDCorner.y * (iResolution.x /
   scalePrev));
    ivec2 RUCorner = ivec2(LDCorner.x + 1, LDCorner.y + 1);
    uint RUCIndex = uint(RUCorner.x + RUCorner.y * (iResolution.x /
   scalePrev));

    bool test = (RUCorner.x >= iResolution.x / scalePrev || RUCorner.y >=
   iResolution.y / scalePrev) || (prevData[LDCIndex] != prevData[RUCIndex])
   ;

    res = test ? first_layer(uv, iterations, u0, beta, V, theta) : prevData
   [LDCIndex];
    // if (RUCorner.x >= iResolution.x / scalePrev || RUCorner.y >=
   iResolution.y / scalePrev) {
    //     // out of bounds, compute new data
    //     res = first_layer(uv, iterations, u0, beta, V, theta);
    // } else if (prevData[LDCIndex] == prevData[RUCIndex]) {
    //     // due to convexity and since the borders are on the axis
    //     res = prevData[LDCIndex];
    // } else {
    //     res = first_layer(uv, iterations, u0, beta, V, theta);
    // }
    #endif

    data[index] = res;
    color[index] = uint2col(res[0]) | (uint2col(res[1]) << 8) | uint(0
   xFF000000);
}
```

Code 22: shaders/uintEfficient.old.glsl

```
struct SNNConfig {
    int iterations;
    vec2 u0;
    float beta;
    mat2 W;
    vec2 b;
    mat2 V;
    float theta;
    vec2 offset;
    float scale;
};

uint[4] first_layer(vec2 x, SNNConfig conf)
{
    vec2 u = conf.u0;
    uvec2 s = uvec2(0,0);
    uvec2 spiketrLow = uvec2(0,0);
    uvec2 spiketrUp = uvec2(0,0);
    for (int i = 0; i < conf.iterations; ++i) {
        u = conf.beta * u + conf.W * x + conf.b + conf.V * vec2(s) - conf.
   theta * conf.beta * vec2(s);
```

```
        s = uvec2(step(conf.theta,u));
        spiketrUp = (spiketrUp << 1) + (spiketrLow >> 31);
        spiketrLow = (spiketrLow << 1) + s;
    }
    return uint[4](spiketrLow.x, spiketrUp.x, spiketrLow.y, spiketrUp.y);
}

vec2 posToUV(vec2 pos, SNNConfig conf)
{
    vec2 nuv = pos/iResolution.xy;
    return conf.scale*nuv+conf.offset;
}

float uint2colScal(uint high, uint low, int iterations)
{
    float n = 10.;
    float x = float(high) * pow(2., 32.) + float(low);
    float a = ((1. / n) * log(1. + (exp(n) - 1.) * x / pow(2., float(
   iterations))));
    return a;
}

float uint2colGap(uint high, uint low, int iterations)
{
    int n = 0;
    int nMax = 0;
    for (int i = 0; i < iterations; ++i) {
        if ((low & 1u) == 1u) {
            n++;
            nMax = max(nMax, n);
        } else {
            n = 0;
        }
        low = (low >> 1) + (high << 31);
        high = (high >> 1);
    }
    return (float(nMax) / float(iterations));
}

vec4 uint2col(uint[4] res, int iterations)
{
    return vec4(uint2colGap(res[1], res[0], iterations),
                uint2colGap(res[3], res[2], iterations),
                0.0, 1.0);
}

vec4 uint2colBorder(vec2 fragCoord, uint[4] res, SNNConfig conf)
{
    vec2 aBitRightPos = posToUV(fragCoord.xy+vec2(1,0), conf);
    vec2 aBitUpPos    = posToUV(fragCoord.xy+vec2(0,1), conf);
    uint[4] aBitRight = first_layer(aBitRightPos, conf);
    uint[4] aBitUp = first_layer(aBitUpPos, conf);
    if (res != aBitRight || res != aBitUp) {
        return vec4(1.0); // white
    } else {
        return uint2col(res, conf.iterations);
    }
}
```

```

void mainImage(out vec4 fragColor, in vec2 fragCoord)
{
    int iterations = 20;
    vec2 u0 = vec2(0.0, 0.0);
    float beta = 0.5;
    vec2 b = vec2(0.0, 0.0);
    mat2 W = mat2(1.0, 0.0, 0.0, 1.0);
    mat2 V = mat2(1.0, 0.0, 0.0, 1.0);
    float theta = 1.;
    vec2 offset = vec2(0.0, 0.0);
    float scale = 2.0;
    SNNConfig conf = SNNConfig(iterations, u0, beta, W, b, V, theta, offset
    , scale);

    vec2 uv = posToUV(fragCoord.xy, conf);
    uint index = uint(fragCoord.x + fragCoord.y * iResolution.x);

    uint[4] res = first_layer(uv, conf);

    fragColor = uint2colBorder(fragCoord, res, conf);
}
```

Code 23: shaders/uintShaderToy.glsl

```
#version 460
uniform int prevBatchSize;

layout (local_size_x = 1, local_size_y = 1, local_size_z = 1) in;

layout(std430, binding = 0) buffer Data {
    uint[4] data[];
};

layout(std430, binding = 1) buffer ResNum {
    uint resNum[];
};

uint chunkSize = 4;

void main()
{
    uint batchSize = 2*prevBatchSize;
    uint batchIndex = uint(gl_GlobalInvocationID.x) * batchSize;

    // we merge the batch of uniques at batchIndex with the one at
    batchIndex + prevBatchSize
    uint readPointer = batchIndex + prevBatchSize;
    while (true) {
        uint writePointer = batchIndex;
        while (true) {
            // check if value already exists in lower batch
            if (data[writePointer] == data[readPointer]) {
                break;
            }
            writePointer += 1;

            // we have reached the end of the uniques of the lower batch
```

```
            // (either because we have reached a marker (two equal values) or because we have reached the end of the batch)
            if ((data[writePointer-1] == data[writePointer]) || (writePointer - batchIndex >= prevBatchSize)) {
                data[writePointer] = data[readPointer];
                // mark the end of the next unique values
                data[writePointer+1] = data[readPointer];
                break;
            }
        }

        readPointer += 1;
        // we have reached the end of the uniques of the upper batch
        // (either because we have reached a marker (two equal values) or because we have reached the end of the batch)
        if ((data[readPointer-1] == data[readPointer]) || readPointer == batchSize + batchIndex) {
            break;
        }
    }

    uint pointer = batchIndex;
    while (true) {
        if ((data[pointer] == data[pointer+1])) {
            resNum[0] = pointer - batchIndex + 1;
            break;
        }
        if (pointer - batchIndex >= prevBatchSize) {
            resNum[0] = pointer - batchIndex;
            break;
        }

        pointer += 1;
    }
}
```

Code 24: shaders/uniques.glsl

```
#version 460

uniform dvec2 iResolution;
uniform int iterations;
uniform dvec2 u0;
uniform double beta;
// W and b can't be used with this optimization
// uniform dvec2 b;
dvec2 b = dvec2(0.0, 0.0);
// uniform mat2 W;
mat2 W = mat2(1.0, 0.0, 0.0, 1.0);
uniform mat2 V;
uniform double theta;
uniform dvec2 offset;
uniform double scale;
uniform int scalePrev;
uniform bool initialRun;
uniform bool imagep;

layout (local_size_x = 8, local_size_y = 8, local_size_z = 1) in;
```

```

layout(std430, binding = 0) buffer GridData {
    uint[4] data[];
};
layout(std430, binding = 1) buffer GridColor {
    uint color[];
};

// data from previous run with 1/scalePrev of size unless INITIAL_RUN is
    set
layout(std430, binding = 2) buffer GridPrevData {
    uint[4] prevData[];
};

uint[4] first_layer(dvec2 x, int iterations, dvec2 u0, double beta, mat2 V,
     double theta)
{
    dvec2 u = u0;
    uvec2 s = uvec2(0,0);
    uvec2 spiketrLow = uvec2(0,0);
    uvec2 spiketrUp = uvec2(0,0);
    for (int i = 0; i < iterations; ++i) {
        u = beta * (u - theta * s) + W * x + b + V * s;
        s = uvec2(step(theta,u));
        spiketrUp = (spiketrUp << 1) + (spiketrLow >> 31);
        spiketrLow = (spiketrLow << 1) + s;
    }
    return uint[4](spiketrLow.x, spiketrUp.x, spiketrLow.y, spiketrUp.y);
}

uint uint2colScal(uint high, uint low)
{
    float n = 10.;
    float x = float(high) * pow(2., 32.) + float(low);
    double a = ((1. / n) * log(1 + (exp(n) - 1) * x / pow(2., float(
    iterations))));
    // a should be in [0,1] by definition. But just to make sure:
    return min(255, uint(floor(255*a)));
}

uint uint2col(uint[4] res)
{
    return uint2colScal(res[1], res[0]) | (uint2colScal(res[3], res[2]) <<
    8) | uint(0xFF000000);
}

uint uint2colScalSimple(uint high, uint low)
{
    float x = float(high) * pow(2., 32.-float(iterations)) + float(low)/
    pow(2., iterations);
    return uint(255.*x);
}

uint uint2colSimple(uint[4] res)
{
    return uint2colScalSimple(res[1], res[0]) | (uint2colScalSimple(res[3],
     res[2]) << 8) | uint(0xFF000000);
}
```

```

void main()
{
    dvec2 nuv = gl_GlobalInvocationID.xy/iResolution.xy;
    uint index = uint(gl_GlobalInvocationID.x + gl_GlobalInvocationID.y * iResolution.x);
    dvec2 uv = scale*nuv+offset;

    uint[4] res;
    ivec2 LDCorner = ivec2(gl_GlobalInvocationID.x / scalePrev, gl_GlobalInvocationID.y / scalePrev);
    uint LDCIndex = uint(LDCorner.x + LDCorner.y * (iResolution.x / scalePrev));
    ivec2 RUCorner = ivec2(LDCorner.x + 1, LDCorner.y + 1);
    uint RUCIndex = uint(RUCorner.x + RUCorner.y * (iResolution.x / scalePrev));

    bool test = initialRun
        || (RUCorner.x >= iResolution.x / scalePrev || RUCorner.y >= iResolution.y / scalePrev)
        || (prevData[LDCIndex] != prevData[RUCIndex]);
    if (test) {
        res = first_layer(uv, iterations, u0, beta, V, theta);
    } else {
        res = prevData[LDCIndex];
    }

    data[index] = res;
    if (imagep) {
        color[index] = uint2col(res);
    }
}
```

Code 25: shaders/uintEfficient.glsl

```
#include <iostream>
#include <pybind11/pybind11.h>
#include <pybind11/stl.h>
#include <string>
#include <unordered_set>

namespace py = pybind11;

// Function that counts unique byte sequences (chunks) of length 'chunk_size'
// in a binary string provided as a py::bytes object.
size_t count_unique_chunks(py::bytes data, size_t chunk_size) {
    // Convert the Python bytes object to a std::string.
    std::string s = data;
    size_t data_size = s.size();

    if (chunk_size <= 0 || static_cast<size_t>(chunk_size) > data_size) {
        throw std::invalid_argument("Invalid chunk size: " +
                                    std::to_string(chunk_size));
    }

    if (data_size % chunk_size != 0) {
        throw std::invalid_argument(
```

```
            "Data size must be a multiple of chunk size");
    }

    std::unordered_set<std::string> unique;
    for (size_t i = 0; i < data_size; i += chunk_size) {
        unique.insert(s.substr(i, chunk_size));
    }

    return unique.size();
}

size_t _get_index_of(size_t size, int chunk_size,
                     std::pair<size_t, size_t> point) {
    return chunk_size * (point.first * size + point.second);
}

// We regard data as a grid of width and height size, where each node has a
// value of with bytes chunk_size.
// This function should always return the same value as count_unique_chunks.
// This function should be much more efficient than count_unique_chunks,
// since it uses the fact that regions are cuboids.
namespace count_regions {
void _count_subregion(std::unordered_set<std::string> &unique,
                      const std::string &s, std::pair<size_t, size_t> corner,
                      size_t subSize, size_t size, size_t chunk_size) {
    size_t startIndex = _get_index_of(size, chunk_size, corner);
    if (subSize <= 2) {
        for (size_t i = 0; i < subSize; ++i) {
            for (size_t j = 0; j < subSize; ++j) {
                size_t index = _get_index_of(
                    size, chunk_size, {corner.first + i, corner.second + j});
                unique.insert(s.substr(index, chunk_size));
            }
        }
        // Base case: add the single chunk at the corner to the set.
        unique.insert(s.substr(startIndex, chunk_size));
        return;
    } else {
        size_t endIndex = _get_index_of(
            size, chunk_size,
            {corner.first + subSize - 1, corner.second + subSize - 1});
        if (s.substr(startIndex, chunk_size) ==
            s.substr(endIndex, chunk_size)) {
            unique.insert(s.substr(startIndex, chunk_size));
        } else {
            // Recursive case: divide the region into four quadrants.
            size_t halfSize = subSize / 2;
            _count_subregion(unique, s, {corner.first, corner.second}, halfSize,
                             size, chunk_size);
            _count_subregion(unique, s,
                             {corner.first, corner.second + halfSize}, halfSize,
                             size, chunk_size);
            _count_subregion(unique, s,
```

```
                              {corner.first + halfSize, corner.second},
    halfSize,
                              size, chunk_size);
            _count_subregion(
                unique, s, {corner.first + halfSize, corner.second +
    halfSize},
                halfSize, size, chunk_size);
        }
    }
}

bool _is_power_of_two(int x) { return x > 0 && (x & (x - 1)) == 0; }

size_t count_regions(py::bytes data, size_t size, size_t chunk_size) {
    std::string s = data;
    size_t data_size = s.size();

    if (chunk_size <= 0) {
        throw std::invalid_argument("Invalid chunk size: " +
                                    std::to_string(chunk_size));
    }

    if (size <= 0) {
        throw std::invalid_argument("Invalid size: " +
                                    std::to_string(chunk_size));
    }

    if (data_size != size * size * chunk_size) {
        throw std::invalid_argument(
            "Data byte count must be equal to size * size * chunk_size");
    }

    if (!_is_power_of_two(size)) {
        throw std::invalid_argument("Size must be a power of two");
    }

    std::unordered_set<std::string> unique;
    _count_subregion(unique, s, {0, 0}, size, size, chunk_size);
    return unique.size();
}
} // namespace count_regions

// search last element with same value as start until end in s
// values are have size chunk_size
size_t search_steps(std::string &s, size_t chunk_size, size_t start, size_t
     end,
                            size_t stepSize) {
    size_t d = 0;
    size_t n = 0;
    while (true) {
        size_t nextD = d + (1 << n);
        size_t nextIndex = start + nextD * stepSize;
        if ((nextIndex < end) &&
            (std::memcmp(s.data() + nextIndex, s.data() + start, chunk_size
    ) ==
             0)) {
            d = nextD;
            n++;
```

```
        } else {
            // if n == 0 we found the last node with same value as
            // (i,j)
            if (n == 0) {
                break;
            } else {
                n--;
            }
        }
    }
    return d;
}

std::vector<py::bytes> unique_regions(py::bytes data, size_t size,
                                      size_t chunk_size) {
    std::string s = data;
    size_t data_size = s.size();

    if (chunk_size <= 0) {
        throw std::invalid_argument("Invalid chunk size: " +
                                    std::to_string(chunk_size));
    }

    if (size <= 0) {
        throw std::invalid_argument("Invalid size: " +
                                    std::to_string(chunk_size));
    }

    if (data_size != size * size * chunk_size) {
        throw std::invalid_argument(
            "Data byte count must be equal to size * size * chunk_size");
    }

    // valueMap captures if the value of a certain element in s
    // has already been added to values.
    // If that is the case, valueMap has the index of the value in values.
    std::vector<int> valueMap(s.size() / chunk_size, -1);
    std::vector<py::bytes> values{};

    for (size_t i = 0; i < size; ++i) {
        for (size_t j = 0; j < size; ++j) {
            size_t indexVM = i * size + j;
            // check if value has already been added to values
            if (valueMap[indexVM] == -1) {
                // find the width dx and height dy of the region with the
                // same value
                size_t dx =
                    1 + search_steps(s, chunk_size,
                                     _get_index_of(size, chunk_size, {i, j
    }),
                                     _get_index_of(size, chunk_size, {size,
     j}),
                                     size * chunk_size);
                size_t dy =
                    1 + search_steps(s, chunk_size,
                                     _get_index_of(size, chunk_size, {i, j
    }),
                                     _get_index_of(size, chunk_size, {i,
```

```
    size}),
                                          chunk_size);
                values.push_back(s.substr(
                    _get_index_of(size, chunk_size, {i, j}), chunk_size));
                int valueIndex = values.size();
                for (size_t k = 0; k < dx; ++k) {
                    for (size_t l = 0; l < dy; ++l) {
                        valueMap[(i + k) * size + (j + l)] = valueIndex;
                    }
                }
            }
        }
    }

    return values;
}

std::vector<py::bytes> llcorner_values(py::bytes data, size_t size,
                                       size_t chunk_size) {
    std::string s = data;
    size_t data_size = s.size();

    if (chunk_size <= 0) {
        throw std::invalid_argument("Invalid chunk size: " +
                                    std::to_string(chunk_size));
    }

    if (size <= 0) {
        throw std::invalid_argument("Invalid size: " +
                                    std::to_string(chunk_size));
    }

    if (data_size != size * size * chunk_size) {
        throw std::invalid_argument(
            "Data byte count must be equal to size * size * chunk_size");
    }

    std::vector<py::bytes> values{};

    for (size_t i = 0; i < size; ++i) {
        for (size_t j = 0; j < size; ++j) {
            // Check if the value at the current position is different
            // from the value to the left and below it.
            bool leftIsDiff =
                i == 0 ||
                (std::memcmp(s.data() + _get_index_of(size, chunk_size, {i,
     j}),
                             s.data() +
                                 _get_index_of(size, chunk_size, {i - 1, j
    }),
                             chunk_size) != 0);
            bool belowIsDiff =
                j == 0 ||
                (std::memcmp(s.data() + _get_index_of(size, chunk_size, {i,
     j}),
                             s.data() +
                                 _get_index_of(size, chunk_size, {i, j -
    1}),
```

```
                              chunk_size) != 0);
            // If it is different indeed, the current position is a ll
    corner of
            // a region.
            if (leftIsDiff && belowIsDiff) {
                values.push_back(s.substr(
                    _get_index_of(size, chunk_size, {i, j}), chunk_size));
            }
        }
    }

    return values;
}

int llcorner_uniques(py::bytes data, size_t size, size_t chunk_size) {
    std::string s = data;
    size_t data_size = s.size();

    if (chunk_size <= 0) {
        throw std::invalid_argument("Invalid chunk size: " +
                                    std::to_string(chunk_size));
    }

    if (size <= 0) {
        throw std::invalid_argument("Invalid size: " +
                                    std::to_string(chunk_size));
    }

    if (data_size != size * size * chunk_size) {
        throw std::invalid_argument(
            "Data byte count must be equal to size * size * chunk_size");
    }

    int valuesN = 0;

    for (size_t i = 0; i < size; ++i) {
        for (size_t j = 0; j < size; ++j) {
            // Check if the value at the current position is different
            // from the value to the left and below it.
            bool leftIsDiff =
                i == 0 ||
                (std::memcmp(s.data() + _get_index_of(size, chunk_size, {i,
     j}),
                              s.data() +
                                  _get_index_of(size, chunk_size, {i - 1, j
    }),
                              chunk_size) != 0);
            bool belowIsDiff =
                j == 0 ||
                (std::memcmp(s.data() + _get_index_of(size, chunk_size, {i,
     j}),
                              s.data() +
                                  _get_index_of(size, chunk_size, {i, j -
    1}),
                              chunk_size) != 0);
            // If it is different indeed, the current position is a ll
    corner of
            // a region.
```

```
            if (leftIsDiff && belowIsDiff) {
                valuesN++;
            }
        }
    }

    return valuesN;
}

PYBIND11_MODULE(unique_bytes, m) {
    m.doc() = "Module for counting unique fixed-length byte sequences from a "
              "binary string";
    m.def("count_unique_chunks", &count_unique_chunks, py::arg("data"),
          py::arg("chunk_size"),
          "Count unique chunks in the given bytes object with the specified "
          "chunk_size");
    m.def("count_regions", &count_regions::count_regions, py::arg("data"),
          py::arg("size"), py::arg("chunk_size"),
          "Count unique regions in the given bytes object with the "
          "specified size and chunk_size");
    m.def("unique_regions", &unique_regions, py::arg("data"), py::arg("size"),
          py::arg("chunk_size"),
          "Get unique regions in the given bytes object with the "
          "specified size and chunk_size");
    m.def("llcorner_values", &llcorner_values, py::arg("data"), py::arg("size"),
          py::arg("chunk_size"),
          "Get the values of the lower-left corners of unique regions in the "
          "given bytes object with the specified size and chunk_size"
          "Only works for cuboids regions.");
    m.def("llcorner_uniques", &llcorner_uniques, py::arg("data"),
          py::arg("size"), py::arg("chunk_size"),
          "Count the number of unique lower-left corners in the given bytes "
          "object with the specified size and chunk_size"
          "Only works for cuboids regions.");
}
```

Code 26: src/unique_bytes.cpp

7.3 dt-lif-snn-visualizer

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

```
import { baseScale } from "./Globals";
import { throttle } from "./Helpers";
import { Mat2D, Vec2D } from "./Types";

const PROXY_TAG = Symbol("isProxy");
const PROXY_FN_TAG = Symbol("prxxyFn");

const makeReactiveObj = <T extends object>(obj: T, onChange: () => void) => {
  const proxy = new Proxy(obj, {
```

```
    set(target, prop, value) {
      if (prop === PROXY_TAG || prop === PROXY_FN_TAG) {
        (target as any)[prop as any] = value;
        return true;
      }
      if (
        typeof value === "object" &&
        value !== null &&
        (!(value as any)[PROXY_TAG] || (value as any)[PROXY_FN_TAG] !== onChange)
      ) {
        value = makeReactiveObj(value, onChange);
      }
      (target as any)[prop as any] = value;
      onChange();
      return true;
    },
    deleteProperty(target, prop) {
      delete (target as any)[prop as any];
      onChange();
      return true;
    },
  });

  // Wrap initial nested objects
  for (const key in obj) {
    const value = (obj as any)[key];
    if (
      typeof value === "object" &&
      value !== null &&
      (!(value as any)[PROXY_TAG] || (value as any)[PROXY_FN_TAG] !== onChange)
    ) {
      (obj as any)[key] = makeReactiveObj(value, onChange);
    }
  }

  (proxy as any)[PROXY_TAG] = true;
  (proxy as any)[PROXY_FN_TAG] = onChange;
  return proxy;
};

export const backgroundColorAlgs = [
  "countSpikes",
  "spikeTrain",
  "spikeTrainSimple",
  "u",
  "p",
] as const;

export const borderTypes = [
  "newBorder",
  "differentCountPrev",
  "differentCount",
  "other",
];

export const colorWithAlgs = [
```

```
  "spikeTrain",
  "spikeTrainPrev",
  "spikeCount",
  "spikeCountPrev",
];

export type Conf = {
  iterations: number;
  u0: Vec2D;
  i0: Vec2D;
  alpha: number;
  beta: number;
  b: Vec2D;
  W: Mat2D;
  V: Mat2D;
  theta: number;
  offset: Vec2D;
  scale: number;
  showBorders: boolean;
  colorWithAlgSpikes: [number, number];
  colorWithAlg: number; // corresponds to index in `colorWithAlg`
  showTooltip: boolean;
  showBoundary: boolean;
  tooltipPorU: "u" | "p";
  backgroundColorAlg: number; // corresponds to index in `backgroundColorAlgs`
  borderTypeColors: [number, number, number, number];
  borderTypeOrder: [number, number, number, number]; // coresponds to indices in `borderTypes`
};

const saveConf = throttle(() => {
  const params = new URLSearchParams();
  for (const k in conf) {
    params.set(k, JSON.stringify((conf as any)[k]));
  }
  window.history.replaceState(
    {},
    "",
    `${window.location.pathname}?${params.toString()}`,
  );
}, 500);

const loadConf = () => {
  const params = new URLSearchParams(window.location.search);

  for (const key in conf) {
    if (params.has(key)) {
      try {
        (conf as any)[key] = JSON.parse(params.get(key)!);
      } catch {
        // fallback: treat as plain string if not valid JSON
        (conf as any)[key] = params.get(key);
      }
    }
  }
};

```

```
export const conf = makeReactiveObj<Conf>(
  {
    iterations: 10,
    u0: [0, 0],
    i0: [0, 0],
    alpha: 0,
    beta: 1,
    b: [0, 0],
    W: [1, 0, 0, 1],
    V: [0, 0, 0, 0],
    theta: 1,
    offset: [0, 0],
    scale: baseScale,
    showBorders: true,
    colorWithAlgSpikes: [-1, -1],
    colorWithAlg: 0,
    showTooltip: true,
    showBoundary: false,
    tooltipPorU: "u",
    backgroundColorAlg: 1,
    borderTypeColors: [0xffffffff, 0x0000ffff, 0xaaaaaaff, 0x080808ff],
    borderTypeOrder: [0, 1, 2, 3],
  },
  saveConf,
);

window.addEventListener("load", loadConf);
```

Code 27: src/Conf.ts

```
export const baseScale: number = 3;

export const ParamSliderVals = {
  a: 0,
  b: 0,
  c: 0,
};
export const frontConf = {
  autoShowLeft: false,
  autoShowRight: false,
};
```

Code 28: src/Globals.ts

```
import { Conf } from "./Conf";
import { Point2D, Vec2D } from "./Types";

export function posToUV(pos: Point2D, conf: Conf): Point2D {
  const minSize = Math.min(window.innerWidth, window.innerHeight);
  const uv: Point2D = [
    conf.scale * (pos[0] / minSize - 0.5) + conf.offset[0],
    conf.scale * (pos[1] / minSize - 0.5) + conf.offset[1],
  ];
  return uv;
}

export const clientCordConv = (
  clientX: number,
```

```
  clientY: number,
  conf: Conf,
) => {
  return posToUV([clientX, window.innerHeight - clientY - 1], conf);
};

export function firstLayer(
  x: Point2D,
  conf: Conf,
): [[number[], number[]], Vec2D, string[]] {
  let i: [number, number] = [0, 0];
  let p: [number, number] = conf.u0.slice() as Vec2D;
  let u: [number, number] = p.slice() as Vec2D;
  let uhistory: [number[], number[]] = [[u[0]], [u[1]]];
  let s: [number, number] = [0, 0];
  let spiketr: [string, string] = ["", ""];

  const alpha = conf.alpha;
  const beta = conf.beta;
  const b = conf.b;
  const theta = conf.theta;
  const W = conf.W;
  const V = conf.V;
  const xWb = [W[0] * x[0] + W[2] * x[1], W[1] * x[0] + W[3] * x[1]];
  const useU = conf.tooltipPorU == "u"; // fix value during loop
  for (let l = 0; l < conf.iterations; l++) {
    const prevS = s.slice() as Vec2D;
    for (let j = 0; j < 2; j++) {
      i[j] = alpha * i[j] + xWb[j] + (V[j] * prevS[0] + V[j + 2] * prevS
[1]);
      p[j] = beta * u[j] + i[j] + b[j];
      s[j] = p[j] >= theta ? 1 : 0;
      u[j] = p[j] - theta * s[j];

      spiketr[j] = spiketr[j] + (s[j] === 1 ? "1" : "0");
    }
    if (useU) {
      uhistory[0].push(u[0]);
      uhistory[1].push(u[1]);
    } else {
      uhistory[0].push(p[0]);
      uhistory[1].push(p[1]);
    }
  }

  return [uhistory, u, spiketr];
}

export function countChar(str: string, char: string): number {
  let count = 0;
  for (let i = 0; i < str.length; i++) {
    if (str[i] === char) {
      count++;
    }
  }
  return count;
}
```

```
export function range1ToN(n: number): number[] {
  return Array.from({ length: n }, (_, i) => i + 1);
}

export function throttle<T extends (...args: any[]) => void>(
  fn: T,
  limit: number,
) {
  let inThrottle: boolean = false;
  let calledDuringThrottle: boolean = false;
  return function (this: ThisParameterType<T>, ...args: Parameters<T>) {
    if (!inThrottle) {
      const run: () => void = () => {
        fn.apply(this, args);
        inThrottle = true;
        setTimeout(() => {
          if (calledDuringThrottle) {
            calledDuringThrottle = false;
            run();
          } else inThrottle = false;
        }, limit);
      };
      run();
    } else {
      calledDuringThrottle = true;
    }
  };
}
```

Code 29: src/Helpers.ts

```
export let rerender: { fn: null | (() => void) } = { fn: null };
let rerenderRequested = false;
export function requestRerender() {
  if (rerender.fn && !rerenderRequested) {
    rerenderRequested = true;
    window.requestAnimationFrame(() => {
      rerender.fn && rerender.fn();
      rerenderRequested = false;
    });
  }
}
```

Code 30: src/Rerender.ts

```
// TODO: switch to three.js types
export type Vec2D = [number, number];

export type Point2D = Vec2D;

export type Mat2D = [number, number, number, number];
```

Code 31: src/Types.ts

```
declare module "*.glsl" {
  const value: string;
  export default value;
}
```

Code 32: src/custom.d.ts

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

Code 33: vite-env.d.ts

```
import { defineConfig } from "vite";
import react from "@vitejs/plugin-react";
import checker from "vite-plugin-checker";
import glsl from "vite-plugin-glsl";

export default defineConfig({
  plugins: [
    glsl(),
    react(),
    checker({
      typescript: true,
    }),
  ],
});
```

Code 34: vite.config.ts

```
import React, { useState } from "react";
import { Checkbox } from "@mui/material";
import { ParameterInput } from "./ParameterInput";
export function CheckboxInput({
  paramName,
  defaultValue,
  onChange,
}: {
  paramName: string;
  defaultValue: boolean;
  onChange: (checked: boolean) => void;
}) {
  const [checked, setChecked] = useState(defaultValue);

  const handlechange = (event: any) => {
    setChecked(event.target.checked);
    onChange(event.target.checked);
  };

  return (
    <ParameterInput paramName={paramName}>
      <Checkbox checked={checked} onChange={handlechange} />
    </ParameterInput>
  );
}
```

Code 35: src/CheckboxInput.tsx

```
import React, { useState } from "react";
import { Box, TextField, Typography } from "@mui/material";
import { requestRerender } from "./Rerender";

const uintToRGBA = (value: number) => {
  const r = (value >> 24) & 0xff;
  const g = (value >> 16) & 0xff;
  const b = (value >> 8) & 0xff;
  const a = value & 0xff;
  return { r, g, b, a };
};

const RGBAtoUint = (r: number, g: number, b: number, a: number) => {
  return (
    ((r & 0xff) << 24) | ((g & 0xff) << 16) | ((b & 0xff) << 8) | (a & 0xff
  )
  );
};

export function ColorInput({
  defaultValue,
  onChange,
}: {
  defaultValue: number;
  onChange: (value: number) => void;
}) {
  const [value, setValue] = useState<{
    r: number | null;
    g: number | null;
    b: number | null;
    a: number | null;
  }>(uintToRGBA(defaultValue));
  const handleChange = (cp: "r" | "g" | "b") => (event: any) => {
    const str = event.target.value;
    if (str === "") {
      setValue((v) => ({ ...v, [cp]: null }));
      return;
    }
    const newValue = Number(str);
    if (!isNaN(newValue)) {
      setValue((v) => {
        const updated = { ...v, [cp]: Math.max(0, Math.min(255, newValue))
  };
        onChange(
          RGBAtoUint(updated.r ?? 0, updated.g ?? 0, updated.b ?? 0, 255),
        );
        requestRerender();
        return updated;
      });
    } else {
      setValue((v) => ({ ...v, [cp]: null }));
      console.error("Invalid input:", str);
    }
  };

  return (
    <Box sx={{ display: "flex", gap: 1 }}>
```

```
      <Typography sx={{ fontWeight: "bold" }}>#</Typography>
      <TextField
        size="small"
        sx={{ width: 60 }}
        value={value.r ?? ""}
        onChange={handleChange("r")}
      />
      <TextField
        size="small"
        sx={{ width: 60 }}
        value={value.g ?? ""}
        onChange={handleChange("g")}
      />
      <TextField
        size="small"
        sx={{ width: 60 }}
        value={value.b ?? ""}
        onChange={handleChange("b")}
      />
    </Box>
  );
}
```

Code 36: src/ColorInput.tsx

```
import React from "react";
import { Grid } from "@mui/material";
import { Mat2D } from "./Types";
import { ParameterInput } from "./ParameterInput";
import { NumberInput } from "./NumberInput";

export function Mat2DInput({
  paramName,
  defaultValue,
  setValue,
}: {
  paramName: string;
  defaultValue: Mat2D;
  setValue: (index: number, value: number) => void;
}) {
  return (
    <ParameterInput paramName={paramName}>
      <Grid
        container
        sx={{ px: 2 }}
        rowSpacing={1}
        columnSpacing={{ xs: 1, sm: 2, md: 3 }}
      >
        <Grid size={6}>
          <NumberInput
            defaultValue={defaultValue[0]}
            setValue={(v) => setValue(0, v)}
          />
        </Grid>
        <Grid size={6}>
          <NumberInput
            defaultValue={defaultValue[2]}
            setValue={(v) => setValue(2, v)}
```

```
          />
        </Grid>
        <Grid size={6}>
          <NumberInput
            defaultValue={defaultValue[1]}
            setValue={(v) => setValue(1, v)}
          />
        </Grid>
        <Grid size={6}>
          <NumberInput
            defaultValue={defaultValue[3]}
            setValue={(v) => setValue(3, v)}
          />
        </Grid>
      </Grid>
    </ParameterInput>
  );
}
```

Code 37: src/Mat2DInput.tsx

```
import React from "react";
import {
  Box,
  Drawer,
  Grid,
  IconButton,
  Stack,
  Typography,
} from "@mui/material";
import MenuIcon from "@mui/icons-material/Menu";
import { ParamSlider } from "./ParamSlider";
import { frontConf, ParamSliderVals } from "./Globals";
import { backgroundColorAlgs, borderTypes, colorWithAlgs, conf } from "./
    Conf";
import { NumberInputStandAlone } from "./NumberInputStandAlone";
import { Mat2DInput } from "./Mat2DInput";
import { Vec2DInput } from "./Vec2DInput";
import { requestRerender } from "./Rerender";
import { CheckboxInput } from "./CheckboxInput";
import { SelectInput } from "./SelectInput";
import { ColorInput } from "./ColorInput";

export function Menu() {
  const [open, setOpen] = React.useState(false);
  const drawerWidth = 400;

  const handleDrawerFlip = () => {
    setOpen((b) => !b);
  };

  return (
    <Box>
      <IconButton
        color="inherit"
        onClick={handleDrawerFlip}
        edge="start"
        sx={[
```

```
          {
            position: "absolute",
            top: 10,
            mr: 2,
            right: open ? drawerWidth : 0,
            transition: "right 0.2s",
            zIndex: 1201, // above Drawer zIndex (default is 1200)
          },
        ]}
      >
        <MenuIcon />
      </IconButton>
      <Drawer
        sx={{
          width: drawerWidth,
          flexShrink: 0,
          "& .MuiDrawer-paper": {
            width: drawerWidth,
            boxSizing: "border-box",
          },
        }}
        variant="persistent"
        anchor="right"
        open={open}
      >
        <Typography variant="h5" sx={{ p: 2 }}>
          Parameters
        </Typography>
        <ParamSlider
          paramName="a"
          defaultValue={ParamSliderVals.a}
          min={0}
          max={1}
          step={0.01}
          setValue={(x) => (ParamSliderVals.a = x)}
        />
        <ParamSlider
          paramName="b"
          defaultValue={ParamSliderVals.b}
          min={0}
          max={1}
          step={0.01}
          setValue={(x) => (ParamSliderVals.b = x)}
        />
        <ParamSlider
          paramName="c"
          defaultValue={ParamSliderVals.c}
          min={0}
          max={1}
          step={0.01}
          setValue={(x) => (ParamSliderVals.c = x)}
        />
        <NumberInputStandAlone
          paramName="T"
          defaultValue={conf.iterations}
          setValue={(x) => (conf.iterations = Math.floor(x))}
        />
        <NumberInputStandAlone
```

```
          paramName=" "
          defaultValue={conf.alpha}
          setValue={(x) => (conf.alpha = x)}
        />
        <NumberInputStandAlone
          paramName=" "
          defaultValue={conf.beta}
          setValue={(x) => (conf.beta = x)}
        />
        <NumberInputStandAlone
          paramName=" "
          defaultValue={conf.theta}
          setValue={(x) => (conf.theta = x)}
        />
        <Mat2DInput
          paramName="W"
          defaultValue={conf.W}
          setValue={(i, v) => (conf.W[i] = v)}
        />
        <Vec2DInput
          paramName="b"
          defaultValue={conf.b}
          setValue={(i, v) => (conf.b[i] = v)}
        />
        <Vec2DInput
          paramName="u(0)"
          defaultValue={conf.u0}
          setValue={(i, v) => (conf.u0[i] = v)}
        />
        <Mat2DInput
          paramName="V"
          defaultValue={conf.V}
          setValue={(i, v) => (conf.V[i] = v)}
        />
        <SelectInput
          paramName="Background color algorithm"
          defaultValue={backgroundColorAlgs[conf.backgroundColorAlg]}
          values={backgroundColorAlgs.slice()}
          onChange={(v) => {
            conf.backgroundColorAlg = backgroundColorAlgs.indexOf(v as any)
;
            requestRerender();
          }}
        />
        <CheckboxInput
          paramName="Show borders"
          defaultValue={conf.showBorders}
          onChange={(c) => {
            conf.showBorders = c;
            requestRerender();
          }}
        />
        <Stack sx={{ mb: 1, p: 1 }}>
          <Typography variant="h6" sx={{ px: 2 }}>
            Border config
          </Typography>
          <Grid container spacing={2}>
            {[...conf.borderTypeColors.keys()].map((i) => {
```

```
              return (
                <>
                  <Grid key={`${i} 2`} size={5}>
                    <Typography sx={{ px: 2 }}>{borderTypes[i]}: </Typography>
                  </Grid>
                  <Grid key={`${i} 3`} size={7}>
                    <ColorInput
                      defaultValue={conf.borderTypeColors[i]}
                      onChange={(v) => (conf.borderTypeColors[i] = v)}
                    />
                  </Grid>
                </>
              );
            })}
          </Grid>
        </Stack>
        <CheckboxInput
          paramName="Show tooltip"
          defaultValue={conf.showTooltip}
          onChange={(c) => {
            conf.showTooltip = c;
          }}
        />
        <CheckboxInput
          paramName="Show boundary"
          defaultValue={conf.showBoundary}
          onChange={(c) => {
            conf.showBoundary = c;
            requestRerender();
          }}
        />
        <CheckboxInput
          paramName="Use post-spike potential for tooltip"
          defaultValue={conf.tooltipPorU === "u"}
          onChange={(c) => {
            conf.tooltipPorU = c ? "u" : "p";
            requestRerender();
          }}
        />
        <CheckboxInput
          paramName="Show same left"
          defaultValue={frontConf.autoShowLeft}
          onChange={(c) => {
            if (!c && frontConf.autoShowLeft) {
              conf.colorWithAlgSpikes[0] = -1;
              requestRerender();
            }
            frontConf.autoShowLeft = c;
          }}
        />
        <CheckboxInput
          paramName="Show same right"
          defaultValue={frontConf.autoShowRight}
          onChange={(c) => {
            if (!c && frontConf.autoShowRight) {
              conf.colorWithAlgSpikes[1] = -1;
              requestRerender();
```

```
          }
          frontConf.autoShowRight = c;
        }}
      />
      <SelectInput
        paramName="Color same by"
        defaultValue={colorWithAlgs[conf.colorWithAlg]}
        values={colorWithAlgs.slice()}
        onChange={(v) => {
          conf.colorWithAlg = colorWithAlgs.indexOf(v as any);
          requestRerender();
        }}
      />
    </Drawer>
  </Box>
  );
}
```

Code 38: src/Menu.tsx

```
import React from "react";
import { TextField } from "@mui/material";
import { ParamSliderVals } from "./Globals";
import { requestRerender } from "./Rerender";

export function NumberInput({
  defaultValue,
  setValue,
}: {
  defaultValue: number;
  step?: number;
  setValue: (value: number) => void;
}) {
  const [valueStr, setValueStr] = React.useState<string>(
    defaultValue.toString(),
  );
  const [intervalId, setIntervalId] = React.useState<number | null>(null);
  const handleInputChange = (event: any) => {
    const str = event.target.value;
    intervalId && clearInterval(intervalId);
    setValueStr(str);
    if (str === "") {
      return;
    }
    const newValue = Number(str);
    if (!isNaN(newValue)) {
      setValue(newValue);
    } else {
      try {
        const f = eval(`(x,a,b,c,sin,cos,exp) => ${str}`);
        let currentValue: number = NaN;
        setIntervalId(
          setInterval(() => {
            try {
              const evaluatedValue = f(
                Date.now() / 1000,
                ParamSliderVals.a,
                ParamSliderVals.b,
```

```
                ParamSliderVals.c,
                Math.sin,
                Math.cos,
                Math.exp,
              );
              if (isNaN(evaluatedValue)) {
                throw new Error("Evaluated value is NaN");
              }
              if (currentValue !== evaluatedValue) {
                setValue(evaluatedValue);
                currentValue = evaluatedValue;
                requestRerender();
              }
            } catch (e) {
              intervalId && clearInterval(intervalId);
              console.error("Error evaluating expression:", e);
            }
          }, 1000 / 20),
        );
      } catch (e) {
        console.error("Invalid input:", str);
      }
    }
    requestRerender();
  };

  return (
    <TextField
      value={valueStr}
      onChange={handleInputChange}
      variant="standard"
    />
  );
}
```

Code 39: src/NumberInput.tsx

```
import React from "react";
import { NumberInput } from "./NumberInput";
import { ParameterInput } from "./ParameterInput";
export function NumberInputStandAlone({
  paramName,
  defaultValue,
  setValue,
}: {
  paramName: string;
  defaultValue: number;
  setValue: (value: number) => void;
}) {
  return (
    <ParameterInput paramName={paramName}>
      <NumberInput defaultValue={defaultValue} setValue={setValue} />
    </ParameterInput>
  );
}
```

Code 40: src/NumberInputStandAlone.tsx

```
import React, { useCallback, useState } from "react";
import Slider from "@mui/material/Slider";
import { Typography } from "@mui/material";
import { ParameterInput } from "./ParameterInput";
import { requestRerender } from "./Rerender";

export function ParamSlider({
  paramName,
  defaultValue,
  min,
  max,
  step = 0.01,
  setValue,
  transformValue = (v: number) => v,
}: {
  paramName: string;
  defaultValue: number;
  min: number;
  max: number;
  step?: number;
  setValue: (value: number) => void;
  transformValue?: (v: number) => number;
}) {
  const [value, setValueIntern] = useState(defaultValue);
  const handleChange = useCallback(
    (_: Event, newValue: number) => {
      const transformedValue = transformValue(newValue);
      setValue(transformedValue);
      setValueIntern(transformedValue);
      requestRerender();
    },
    [setValue, setValueIntern, transformValue],
  );
  return (
    <ParameterInput paramName={paramName}>
      <Slider
        size="small"
        sx={{
          px: 1,
          py: 2,
        }}
        min={min}
        max={max}
        step={step}
        defaultValue={defaultValue}
        onChange={handleChange}
      />
      <Typography variant="h6" sx={{ flexGrow: 1, px: 2 }}>
        {value.toFixed(2)}
      </Typography>
    </ParameterInput>
  );
}
```

Code 41: src/ParamSlider.tsx

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

```

export function ParameterInput({
  paramName,
  children,
}: {
  paramName: string;
  children: React.ReactNode;
}) {
  return (
    <Stack
      spacing={2}
      direction="row"
      sx={{ alignItems: "center", mb: 1, p: 1 }}
    >
      <Typography variant="h6" sx={{ px: 2 }}>
        {paramName}
      </Typography>
      {children}
    </Stack>
  );
}
```

Code 42: src/ParameterInput.tsx

```
import React, { useState } from "react";
import { MenuItem, Select } from "@mui/material";
import { ParameterInput } from "./ParameterInput";
export function SelectInput({
  paramName,
  values,
  defaultValue,
  onChange,
}: {
  paramName: string;
  values: string[];
  defaultValue: string;
  onChange: (value: string) => void;
}) {
  const [value, setValue] = useState(defaultValue);

  const handleChange = (event: any) => {
    setValue(event.target.value);
    onChange(event.target.value);
  };

  return (
    <ParameterInput paramName={paramName}>
      <Select value={value} onChange={handleChange}>
        {values.map((v) => (
          <MenuItem key={v} value={v}>
            {v}
          </MenuItem>
        ))}
      </Select>
    </ParameterInput>
  );
}
```

Code 43: src/SelectInput.tsx

```
import { Point2D, Vec2D } from "./Types";
import { clientCordConv, countChar, firstLayer, range1ToN } from "./Helpers
  ";
import { frontConf } from "./Globals";
import { conf } from "./Conf";
import { requestRerender } from "./Rerender";
import { Box, createTheme, ThemeProvider } from "@mui/material";
import { useEffect, useState } from "react";
import {
  LineChart,
  MarkElement,
  MarkElementProps,
} from "@mui/x-charts/LineChart";
import { ChartsReferenceLine } from "@mui/x-charts/ChartsReferenceLine";

const darkTheme = createTheme({
  palette: { mode: "dark" },
});

const CustomMark = (props: MarkElementProps) => (
  <MarkElement {...props} classes={{ root: "customMarkElement" }} />
);

const uLineChart = (uhistory: [number[], number[]]) => {
  return (
    <LineChart
      xAxis={[
        {
          data: range1ToN(uhistory[0].length),
          domainLimit: "strict",
          tickMaxStep: 2,
        },
      ]}
      yAxis={[{ tickMaxStep: 1 }]}
      series={[
        {
          data: uhistory[0],
          curve: "linear",
          disableHighlight: true,
        },
        {
          data: uhistory[1],
          curve: "linear",
          disableHighlight: true,
        },
      ]}
      slots={{ mark: CustomMark }}
      sx={{
        "& .customMarkElement": {
          scale: "0.5", // Shrinks the dot to 50% of its original size
        },
      }}
      height={70}
      skipAnimation={true}
      disableAxisListener={true}
```

```
      disableLineItemHighlight={true}
      margin={{ top: 5, right: 5, bottom: 0, left: -20 }}
    >
      <ChartsReferenceLine
        y={0}
        lineStyle={{ stroke: "gray", strokeWidth: 1 }}
      />
      <ChartsReferenceLine
        y={1}
        lineStyle={{ stroke: "gray", strokeWidth: 1 }}
      />
    </LineChart>
  );
};

export const Tooltip = () => {
  const [display, setDisplay] = useState(true);
  const [pos, setPos] = useState<[number, number] | null>(null);
  const [res, setRes] = useState<
    [[number[], number[]], Vec2D, string[]] | null
  >(null);

  useEffect(() => {
    window.addEventListener("mouseout", () => setDisplay(false));
    window.addEventListener(
      "mousemove",
      (event) => {
        if ((event.target as any).tagName.toLowerCase() === "canvas") {
          setPos([event.clientX, event.clientY]);
          setDisplay(true);
          const uv: Point2D = clientCordConv(
            event.clientX,
            event.clientY,
            conf,
          );
          window.requestAnimationFrame(() => {
            const [uhistory, u, st] = firstLayer(uv, conf);
            setRes([uhistory, u, st]);

            if (frontConf.autoShowLeft) {
              conf.colorWithAlgSpikes[0] = Number("0b" + st[0]);
            }
            if (frontConf.autoShowRight) {
              conf.colorWithAlgSpikes[1] = Number("0b" + st[1]);
            }
            if (frontConf.autoShowLeft || frontConf.autoShowRight) {
              requestRerender();
            }
          });
        } else {
        }
      },
      false,
    );
  }, []);

  if (!display || !pos || !res || !conf.showTooltip) {
    return <></>;
```

```
  }

  const [uhistory, u, st] = res;

  const uv: Point2D = clientCordConv(pos[0], pos[1], conf);
  return (
    <Box
      sx={{
        position: "absolute",
        background: "rgba(0,0,0,0.7)",
        color: "white",
        padding: "5px 10px",
        pointerEvents: "none",
        borderRadius: "5px",
        left: pos[0] + 10 + "px",
        top: pos[1] + 10 + "px",
      }}
    >
      Pos: ({uv[0].toFixed(3)},{uv[1].toFixed(3)})
      <br />
      Potential: {u[0].toFixed(3)}:{u[1].toFixed(3)}
      <ThemeProvider theme={darkTheme}>{uLineChart(uhistory)}</
    ThemeProvider>
      Spikes: {st[0]}:{st[1]}
      <br />
      Spike count: {countChar(st[0], "1")}:{countChar(st[1], "1")}
    </Box>
  );
};
```

Code 44: src/Tooltip.tsx

```
import React from "react";
import { Grid } from "@mui/material";
import { ParameterInput } from "./ParameterInput";
import { NumberInput } from "./NumberInput";
import { Vec2D } from "./Types";

export function Vec2DInput({
  paramName,
  defaultValue,
  setValue,
}: {
  paramName: string;
  defaultValue: Vec2D;
  setValue: (index: number, value: number) => void;
}) {
  return (
    <ParameterInput paramName={paramName}>
      <Grid
        container
        sx={{ px: 2 }}
        rowSpacing={1}
        columnSpacing={{ xs: 1, sm: 2, md: 3 }}
      >
        <Grid size={12}>
          <NumberInput
            defaultValue={defaultValue[0]}
```

```
            setValue={(v) => setValue(0, v)}
          />
        </Grid>
        <Grid size={12}>
          <NumberInput
            defaultValue={defaultValue[1]}
            setValue={(v) => setValue(1, v)}
          />
        </Grid>
      </Grid>
    </ParameterInput>
  );
}
```

Code 45: src/Vec2DInput.tsx

```
import * as THREE from "three";
import fragmentShader from "./shaders/fragment.glsl";
import { baseScale } from "./Globals";
import { conf } from "./Conf";
import { requestRerender, rerender } from "./Rerender";
import { Point2D } from "./Types";
import { clientCordConv } from "./Helpers";
import ReactDOM from "react-dom/client";
import React from "react";
import { Menu } from "./Menu";
import { Tooltip } from "./Tooltip";

const init = async () => {
  const scene = new THREE.Scene();
  const camera = new THREE.Camera();
  const renderer = new THREE.WebGLRenderer();
  document.body.appendChild(renderer.domElement);
  const canvas = renderer.domElement;
  canvas.addEventListener("mousedown", (e) => e.preventDefault());
  canvas.addEventListener("mousemove", (e) => e.preventDefault());
  canvas.addEventListener("mouseup", (e) => e.preventDefault());
  canvas.addEventListener("touchstart", (e) => e.preventDefault());
  canvas.addEventListener("touchmove", (e) => e.preventDefault());
  canvas.addEventListener("touchend", (e) => e.preventDefault());

  // Fullscreen plane geometry
  const geometry = new THREE.PlaneGeometry(2, 2);

  // Simple fragment shader
  const material = new THREE.ShaderMaterial({
    fragmentShader: fragmentShader,
    uniforms: {
      iResolution: { value: [window.innerWidth, window.innerHeight] },
      conf: { value: conf },
    },
  });

  const mesh = new THREE.Mesh(geometry, material);
  scene.add(mesh);

  // const animate = (time) => {
  //   // material.uniforms.uTime.value = time * 0.001;
```

```
// 	requestAnimationFrame(animate);
// };

rerender.fn = () => {
  renderer.setSize(window.innerWidth, window.innerHeight);
  renderer.render(scene, camera);
};

window.addEventListener("resize", () => {
  requestRerender();
});
rerender.fn();

// allow moving the plane
var anchorPoint: null | Point2D = null;
var anchorPointUV: null | Point2D = null;
window.addEventListener(
  "mousedown",
  (event) => {
    if ((event.target as any).tagName.toLowerCase() === "canvas") {
      anchorPoint = [event.clientX, event.clientY];
      anchorPointUV = clientCordConv(event.clientX, event.clientY, conf);
    }
  },
  false,
);
window.addEventListener("mousemove", (event) => {
  if (anchorPoint !== null && anchorPointUV !== null) {
    const eventPos = clientCordConv(event.clientX, event.clientY, conf);

    conf.offset = [
      conf.offset[0] + anchorPointUV[0] - eventPos[0],
      conf.offset[1] + anchorPointUV[1] - eventPos[1],
    ];
    requestRerender();
  }
});
window.addEventListener("mouseup", () => (anchorPoint = null), false);

// allow zooming in/out of the plane
let currentScale = 0;
window.addEventListener("wheel", (event) => {
  if ((event.target as any).tagName.toLowerCase() === "canvas") {
    event.preventDefault(); // Prevent default scroll

    const delta = Math.sign(event.deltaY);
    currentScale += delta === 1 ? 1 : -1;
    conf.scale = baseScale * Math.pow(2, currentScale / 20);
    requestRerender();
  }
});

// animate();
ReactDOM.createRoot(document.getElementById("root")!).render(
  <React.StrictMode>
    <Tooltip />
    <Menu />
  </React.StrictMode>,
```

```
  );
};

window.addEventListener("load", () => {
  init();
});
```

Code 46: src/main.tsx

```
struct SNNConfig {
    int iterations;
    vec2 u0;
    vec2 i0;
    float alpha;
    float beta;
    mat2 W;
    vec2 b;
    mat2 V;
    float theta;
    vec2 offset;
    float scale;
    bool showBorders;
    bool autoShowPrev;
    bool showBoundary;
    ivec2 colorWithAlgSpikes;
    uint colorWithAlg;
    uint backgroundColorAlg;
    uint[4] borderTypeColors;
    uint[4] borderTypeOrder;
};

uniform vec2 iResolution;
uniform SNNConfig conf;

struct ST
{
    uint l0;
    uint h0;
    uint l1;
    uint h1;
};

ST shiftRightST(ST st, int n)
{
    ST stRest = st;
    if (n <= 0) {
        return stRest;
    } else if (n <= 32) {
        stRest.l0 = (stRest.l0 >> n) | (stRest.h0 << (32 -n));
        stRest.h0 = (stRest.h0 >> n);
        stRest.l1 = (stRest.l1 >> n) | (stRest.h1 << (32 -n));
        stRest.h1 = (stRest.h1 >> n);
    } else if (n <= 64) {
        stRest.l0 = stRest.h0 >> (n - 32);
        stRest.h0 = 0u;
        stRest.l1 = stRest.h1 >> (n - 32);
        stRest.h1 = 0u;
    } else {
```

```
        stRest = ST(0u, 0u, 0u, 0u);
    }
    return stRest;
}

vec4 unpackColor(uint c) {
    return vec4(
        float((c >> 24) & 0xFFu),  // R
        float((c >> 16) & 0xFFu),  // G
        float((c >> 8)  & 0xFFu),  // B
        float(c & 0xFFu)           // A
    ) / 255.0; // normalize to [0,1]
}

struct LayerRes
{
  ST spikeTrain;
  vec2 uRes;
};


LayerRes first_layer(vec2 x, SNNConfig conf)
{
    vec2 i = conf.i0;
    vec2 p = vec2(0.0);
    uvec2 s = uvec2(0,0);
    vec2 u = conf.u0;
    uvec2 spiketrLow = uvec2(0,0);
    uvec2 spiketrHigh = uvec2(0,0);
        for (int l = 0; l < conf.iterations; ++l) {
            i = conf.alpha * i + conf.W * x + conf.V * vec2(s);
            p = conf.beta * u + i + conf.b;
            s = uvec2(step(conf.theta,p));
            u = p-conf.theta*vec2(s);
            spiketrHigh = (spiketrHigh << 1) + (spiketrLow >> 31);
            spiketrLow = (spiketrLow << 1) + s;
        }
    return LayerRes(ST(spiketrLow.x, spiketrHigh.x, spiketrLow.y,
   spiketrHigh.y), u);
}

vec2 posToUV(vec2 pos, SNNConfig conf)
{
    float resMin = min(iResolution.x, iResolution.y);
    vec2 nuv = pos/resMin- vec2(0.5);
    return conf.scale*nuv+conf.offset;
}

uint bitCount(uint x) {
    uint count = 0u;
    for (int i = 0; i < 32; ++i) {
        count += (x >> uint(i)) & uint(1);
    }
    return count;
}

uvec2 bitCountST(ST st) {
    return uvec2(bitCount(st.l0)+ bitCount(st.h0), bitCount(st.l1)+
```

```
    bitCount(st.h1));
}

float uint2colScalSimple(uint high, uint low, int iterations)
{
    float x = float(high) * pow(2., 32.) + float(low);
    return x / pow(2., float(iterations));
}

float uint2colScal(uint high, uint low, int iterations)
{
    float n = 10.;
    float a = ((1. / n) * log(1. + (exp(n) - 1.) * uint2colScalSimple(high,
    low, iterations)));
    return a;
}

vec4 uint2colSimple(ST res, int iterations)
{
    return vec4(uint2colScalSimple(res.h0, res.l0, iterations),
                uint2colScalSimple(res.h1, res.l1, iterations),
                0.0, 1.0);
}

vec4 uint2col(ST res, int iterations)
{
    return vec4(uint2colScal(res.h0, res.l0, iterations),
                uint2colScal(res.h1, res.l1, iterations),
                0.0, 1.0);
}

bool[2] newestSpikes(ST res) {
    return bool[2]((res.l0 & 1u) == 1u, (res.l1 & 1u) == 1u);
}

bool vec2smallerEQ(vec2 a, vec2 b) {
    return (a.x <= b.x) && (a.y <= b.y);
}

bool bool2smallerEQ(bool[2] a, bool[2] b) {
    return (uint(a[0]) <= uint(b[0])) && (uint(a[1]) <= uint(b[1]));
}

bool stSmallerEQ(ST a, ST b) {
    return ((a.h0 < b.h0) || ((a.h0 == b.h0) && (a.l0 <= b.l0))) &&
        ((a.h1 < b.h1) || ((a.h1 == b.h1) && (a.l1 <= b.l1)));
}

// majority of change is in [0,2]
float Rto01(float x) {
    return atan((x-1.)*2.)*(1./3.15) + 0.5; // atan((x - 1)*2)/pi + 0.5
}

// majority of change is in [0,1]
float Rto01Norm(float x) {
    return atan((x-.5)*3.)*(1./3.15) + 0.5; // atan((x - 0.5)*3)/pi + 0.5
}

```

```
vec4 colU(LayerRes res, float theta, float blue) {
        return vec4(Rto01(res.uRes.x / theta) , Rto01(res.uRes.y / theta),
    blue, 1.0);
}

vec4 colSpikeCount(LayerRes res, int iterations, float blue) {
        return vec4(Rto01Norm(float(bitCount(res.spikeTrain.l0))/float(
    iterations)),
                    Rto01Norm(float(bitCount(res.spikeTrain.l1))/float(
    iterations)),
                    blue, 1.0);
}

vec4 colUNorm(LayerRes res, float theta, float blue) {
        return vec4(Rto01Norm(res.uRes.x / theta) ,
                    Rto01Norm(res.uRes.y / theta) ,
                    blue,
                    1.0);
}

vec4 colPNorm(LayerRes res, float theta, float blue) {
        return vec4(Rto01Norm((res.uRes.x + float(res.spikeTrain.l0 & 1u))
    / theta) ,
                    Rto01Norm((res.uRes.y + float(res.spikeTrain.l1 & 1u))
    / theta) ,
                    blue,
                    1.0);
}

uint firstDifferentBit(ST x, ST y) {
    if (x == y) {
        return ~0u;
    }
    for (uint i = 0u; i<32u; ++i) {
        uint mask = 1u << i;
        if (((x.l0 & mask) != (y.l0 & mask)) ||
            ((x.l1 & mask) != (y.l1 & mask))) {
            return i;
        }
    }
    for (uint i = 0u; i<32u; ++i) {
        uint mask = 1u << i;
        if (((x.h0 & mask) != (y.h0 & mask)) ||
            ((x.h1 & mask) != (y.h1 & mask))) {
            return i+32u;
        }
    }
    return ~0u; // IMPOSSIBLE
}

uint lastDifferentBit(ST x, ST y) {
    if (x == y) {
        return ~0u;
    }
    for (int i = 31; i>=0; --i) {
        uint mask = 1u << uint(i);
        if (((x.h0 & mask) != (y.h0 & mask)) ||
            ((x.h1 & mask) != (y.h1 & mask))) {
```

```
                return uint(i)+32u;
            }
        }
    for (int i = 31; i>=0; --i) {
        uint mask = 1u << uint(i);
        if (((x.l0 & mask) != (y.l0 & mask)) ||
            ((x.l1 & mask) != (y.l1 & mask))) {
            return uint(i);
        }
    }
    return ~0u; // IMPOSSIBLE
}


float geomSeries(float r, int n)
{
    if (n < 0) {
        return 0.0;
    } else if (r == 1.0) {
        return float(n+1);
    } else {
        return (1.0 - pow(r, float(n+1))) / (1.0 - r);
    }
}

// demonominator of g
float doubleGeomSeries(float beta, float alpha, int n)
{
    float sum = 0.0;
    if (conf.beta == 0.0) {
        sum += geomSeries(conf.alpha, n-1);
    } else {
        for (int k = 1; k <= n; ++k) {
            sum += pow(beta, float(n-k)) * geomSeries(alpha, k-1);
        }
    }
    return sum;
}

void mainImage(out vec4 fragColor, in vec2 fragCoord)
{
    vec2 uv = posToUV(fragCoord.xy, conf);
    uint index = uint(fragCoord.x + fragCoord.y * iResolution.x);

    LayerRes res = first_layer(uv, conf);
    ST st = res.spikeTrain;
    bool isBorder = false;

    if (conf.showBoundary) {
        vec2 maxV = vec2(max(0.0, conf.V[1][0]), max(0.0, conf.V[0][1]));
        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]);
        float inf = 1.0 / 0.0;

        vec2 lowerBound = vec2(inf);
        vec2 inputOffset = vec2(0.0);
        for (int t = 1; t <= conf.iterations; ++t) {
            inputOffset = conf.beta * inputOffset + pow(conf.alpha, float(t
```

```
)) * conf.i0 + conf.b + (geomSeries(conf.alpha, t-2))*maxV;
            lowerBound = min(lowerBound,
                                       - (inputOffset + pow(conf.beta, float(t))*
conf.u0-vec2(conf.theta))
                                       / doubleGeomSeries(conf.beta, conf.alpha, t))
;
        }

        vec2 upperBound = vec2(-inf);
        inputOffset = vec2(0);
        for (int t = 1; t <= conf.iterations; ++t) {
            inputOffset = conf.beta * inputOffset + pow(conf.alpha, float(t
)) * conf.i0 + conf.b + (geomSeries(conf.alpha, t-2))*minV*vec2(1.0);
            upperBound = max(upperBound,
                                       - (inputOffset + pow(conf.beta, float(t))*
conf.u0-conf.theta*geomSeries(conf.beta, t-1))
                                       / doubleGeomSeries(conf.beta, conf.alpha, t))
;
        }


        float eps = 1./min(iResolution.x, iResolution.y)*conf.scale;
        if (any(lessThan(abs(uv - lowerBound), vec2(2.*eps))) || any(
lessThan(abs(uv - upperBound), vec2(2.*eps)))) {
            fragColor = vec4(0.0, 0.0, 1.0, 1.0);
            return;
        }
    }

    if (conf.backgroundColorAlg == 0u) {
        fragColor = colSpikeCount(res, conf.iterations, 0.0);
    } else if (conf.backgroundColorAlg == 1u) {
        fragColor = uint2col(st, conf.iterations);
    } else if (conf.backgroundColorAlg == 2u) {
        fragColor = uint2colSimple(st, conf.iterations);
    } else if (conf.backgroundColorAlg == 3u) {
        fragColor = colUNorm(res, conf.theta, 0.0);
    } else if (conf.backgroundColorAlg == 4u) {
        fragColor = colPNorm(res, conf.theta, 0.0);
    }

    if (conf.showBorders) {
        // check values to right side and to upper side
        vec2 aBitRightPos = posToUV(fragCoord.xy+vec2(1,0), conf);
        vec2 aBitUpPos    = posToUV(fragCoord.xy+vec2(0,1), conf);
        LayerRes aBitRightRes = first_layer(aBitRightPos, conf);
        LayerRes aBitUpRes =    first_layer(aBitUpPos, conf);
        ST aBitRightSt = aBitRightRes.spikeTrain;
        ST aBitUpSt =    aBitUpRes.spikeTrain;


        ST st = res.spikeTrain;
        if (st != aBitRightSt || st != aBitUpSt) {
            isBorder = true;
            for (int i = 0; i < 4; ++i) {
                if (conf.borderTypeOrder[i] == 0u) {
                    uint lastDiffBit = min(lastDifferentBit(st, aBitRightSt
), lastDifferentBit(st, aBitUpSt));
```

```
                if (lastDiffBit == 0u) {
                    fragColor = unpackColor(conf.borderTypeColors[0]);
                    break;
                }
            } else if (conf.borderTypeOrder[i] == 1u) {
                uvec2 resCountPrev = bitCountST(shiftRightST(st, 1));
                uvec2 aBitRightCountPrev = bitCountST(shiftRightST(aBitRightSt,1));
                uvec2 aBitUpCountPrev = bitCountST(shiftRightST(aBitUpSt,1));
                if (resCountPrev != aBitRightCountPrev || resCountPrev != aBitUpCountPrev) {
                    fragColor = unpackColor(conf.borderTypeColors[1]);
                    break;
                }
            } else if (conf.borderTypeOrder[i] == 2u) {
                uvec2 resCount = bitCountST(st);
                uvec2 aBitRightCount = bitCountST(aBitRightSt);
                uvec2 aBitUpCount = bitCountST(aBitUpSt);

                if (resCount != aBitRightCount || resCount != aBitUpCount) {
                    fragColor = unpackColor(conf.borderTypeColors[2]);
                    break;
                }
            } else if (conf.borderTypeOrder[i] == 3u) {
                // default
                fragColor = unpackColor(conf.borderTypeColors[3]);
                break;
            }
        }
    }
}

if (!conf.showBorders || !isBorder) {
    uint shiftN = conf.autoShowPrev ? 1u : 0u;

    if (conf.colorWithAlg == 0u) {
        if ((st.h0==0u && (int(st.l0)) == conf.colorWithAlgSpikes[0]) ||
            (st.h1==0u && (int(st.l1)) == conf.colorWithAlgSpikes[1])) {
            fragColor.z = 1.;
        }
    } else if (conf.colorWithAlg == 1u) {
        if ((st.h0==0u && ((st.l0 & 0x80000000u) != 1u) && int(st.l0 >> 1) == (conf.colorWithAlgSpikes[0] >> 1)) ||
            (st.h1==0u && ((st.l1 & 0x80000000u) != 1u) && int(st.l1 >> 1) == (conf.colorWithAlgSpikes[1] >> 1))) {
            fragColor.z = 1.;
        }
    } else if (conf.colorWithAlg == 2u) {
        if ((conf.colorWithAlgSpikes[0] != -1 &&
             (bitCount(st.l0) + bitCount(st.h0)) == bitCount(uint(conf.colorWithAlgSpikes[0]))) ||
            (conf.colorWithAlgSpikes[1] != -1 &&
             (bitCount(st.l1) + bitCount(st.h1)) == bitCount(uint(conf.colorWithAlgSpikes[1])))) {
```

```
                fragColor.z = 1.;
            }
        } else if (conf.colorWithAlg == 3u) {
            if ((conf.colorWithAlgSpikes[0] != -1 &&
                 (bitCount(st.l0 >> 1) + bitCount(st.h0)) == bitCount(uint(
    conf.colorWithAlgSpikes[0]) >> 1)) ||
                (conf.colorWithAlgSpikes[1] != -1 &&
                 (bitCount(st.l1 >> 1) + bitCount(st.h1)) == bitCount(uint(
    conf.colorWithAlgSpikes[1]) >> 1))) {
                fragColor.z = 1.;
            }
        }
    }
}

void main()
{
    vec2 fragCoord = gl_FragCoord.xy;
    vec4 fragColor;
    mainImage(fragColor, fragCoord);
    gl_FragColor = fragColor;
}
```

Code 47: src/shaders/fragment.glsl