RWTH AACHEN UNIVERSITY

BACHELOR THESIS

Dynamic memory traces for sequence learning in spiking networks

Author:
Simon MICHAU

1st Examiner:
Prof. Dr. Abigail MORRISON
2nd Examiner:
Prof. Dr. Michael SCHAUB
Advisor:
Barna ZAJZON

Bachelor thesis in Computer Science
submitted to the

Computation in Neural Circuits Group
Department of Computer Science
RWTH Aachen University

Aachen, October 14, 2022

Acknowledgements

Writing this thesis has been profoundly interesting and challenging. Over the course of work I had the opportunity to learn much about scientific work and had a glimpse at neuroscience at the FZ Jülich. For this I want to thank Prof. Dr. Abigail Morrison and in particular my advisor Barna Zajzon. Without him this work would not have been possible, as he helped me to overcome many of its obstacles and also with understanding the context and content of my thesis more deeply, while also being very motivating and supportive.
Finally, I also want to thank my friends and family who have supported me during this time and volunteered to proofread this thesis.

Contents

Acknowledgements iii

1 Introduction 1
- 1.1 Methods . . . 3

2 Background 5
- 2.1 Terminology for Biological Neural Networks . . . 5
- 2.2 Synaptic Plasticity . . . 6
  - 2.2.1 Short-Term Plasticity (STP) . . . 7
  - 2.2.2 Spike-Timing-Dependent Plasticity (STDP) . . . 7

3 Original Work 9
- 3.1 Outline . . . 9
- 3.2 Basic Network Structure . . . 10
- 3.3 Running the Network . . . 12
- 3.4 Input Generation . . . 12
- 3.5 Dynamic Network Behaviour . . . 13
  - 3.5.1 STP . . . 15
  - 3.5.2 STDP . . . 16
    - Variance Tracking . . . 17
- 3.6 Summary of Complications . . . 18

4 Implementation 19
- 4.1 Overview . . . 19
- 4.2 WTA Circuit Network . . . 20
  - 4.2.1 Grid . . . 20
- 4.3 Input Generation . . . 22
- 4.4 Recording and Measurement . . . 26
- 4.5 Neuron and Synapse Models . . . 27
  - 4.5.1 Neurons . . . 28
  - 4.5.2 Synapses . . . 30

4.6 Summary of Complications 30

5 Results & Comparison **31**
5.1 Toy Models 31
5.1.1 Baseline Test 31
5.1.2 Scaled WTA Test 34
5.1.3 STP Test 34
5.2 Full Scale Network 37
5.2.1 Formation of Assemblies 37
5.2.2 Necessity of Variance Tracking 37
5.2.3 Necessity of STP 40
5.2.4 Performance Benchmarking 40
5.3 Further Experiments 41

6 Conclusion **43**
6.1 Conclusion 43
6.2 Future Work 44

A Glossary of Notations **45**

Bibliography **47**

List of Figures

2.1 Basic Synaptic Connection 6
2.2 STP Example (Morrison, Diesmann, and Gerstner, 2008) . . . 7
2.3 Pre- and Postsynaptic Spikes 8

3.1 Visualization of a random 10×5 WTA circuit grid with $k_{min} = 2, k_{max} = 10$ 10
3.2 WTA circuit with lateral inhibition 11
3.3 Pattern Input Generation 13
3.4 STDP window 17

4.1 Conceptual Drawing of Implementation. Definition of terms: **SG** denotes 'Spikegenerator', **PG** denotes 'Poissongenerator' and **>** denotes 'Parrot neuron'. 20
4.2 `WTACircuit` Class Diagram 21
4.3 `Network` Class Diagram 22
4.4 Single Input Assembly 22
4.5 Input rate comparison with pattern in red and noise in white . 23
4.6 `InputGenerator` Class Diagram 25
4.7 `Recorder` Class Diagram 26

5.1 Toy Model: Test of 1×2 WTA grid with $k = 1$ and one input channel for 1 s. Input and output spikes are fixed at 60 Hz and 50 Hz. STP is disabled. 32
5.2 Toy Model: Test of 1×2 WTA grid with $k = 1$ and one input channel for 1 s. Input and output spikes are fixed at 2 Hz and 100 Hz. STP is disabled. 33
5.3 Toy Model: Test of 1×2 WTA grid with $k = 2$ and one input channel for 1 s. Input and output spikes are fixed at 20 Hz and 50 Hz. STP is disabled. 35
5.4 Toy Model: Test of 1×2 WTA grid with $k = 2$ and one input channel for 1 s. Input and output spikes are fixed at 20 Hz and 50 Hz. STP is enabled. 36

5.5 Full-scale Network: NEST 3 s test after 100 s training 38
5.6 Full-scale Network: Legacy 3 s test after 100 s training. Only 100 neurons plotted to reduce clutter in network spikes plot. . 38
5.7 Full-scale Network: Legacy 3 s test after 100 s simulation with and without variance tracking 39
5.8 Full-scale Network: NEST 3 s test after 100 s training without variance tracking 39
5.9 Full-scale Network: NEST 3 s test after 100 s training without STP 40
5.10 Pattern duration set to 100 ms. Grid size 10×5 with $k \in [2; 10]$ and 100 input channels. Input rate of 5 Hz with additional 2 Hz during pattern presentation. 2 s test after 100 s training. . 41
5.11 Pattern duration set to 900 ms. Grid size 10×5 with $k \in [2; 10]$ and 100 input channels. Input rate of 5 Hz with additional 2 Hz during pattern presentation. 5 s test after 100 s training. . 42

Chapter 1

Introduction

For a long time, the brain has been the subject of scientific research and has inspired other fields and technologies, including the domain of computer science. In the last decade, the technologies derived from it have grown popular and become a part of everyday life in the form of e.g. artificial neural networks and machine learning. And deeper insights into the brain's functioning have enabled medicine to treat patients more successfully. But despite all this progress, most of its exact inner workings including learning and memory are still largely unknown, although there is a consensus that the plasticity of synapses is the substrate that enables it.
For several decades now, the possibilities of gaining insights into this study object through the emerging computational neuroscience have been growing. Computational neuroscience can test, enhance and develop new models that describe biological reality, as well as help neuroscience by providing means of cerebral simulation.
One example of this is the study on sequence learning currently conducted by the department of Computational and Systems Neuroscience (INM-6) at the FZ Jülich, which also offered a small part of this work for a group of bachelor theses on different computational models of sequential learning. This thesis is one of them and is aimed at replicating the learning model from (Klampfl and Maass, 2013).

In their work, the authors of (Klampfl and Maass, 2013) modeled so-called pyramidal cells (PCs) - a type of neuron that appears for example in the prefrontal cortex and is assumed to play an important role in cognition (Elston, 2003) - with added *lateral inhibition* by creating a modified version of liquid computing model. A liquid computing model (also called liquid state machine) implements a spiking neural network by using recurrent connections

between nodes (neurons) that transmit temporal signals (spikes) that activate the nodes in a certain way. The name "liquid state machine" comes from the fact that each node has a continuous state instead of a discrete one, and can pass it on to neighboring nodes, similar to waves (Maass and Markram, 2004). Although one might assume so due to the biological context, it has nothing to do with the plasticity (or liquidity) of connections between nodes. The phenomenon of lateral inhibition which was also mentioned stems from neuroscience and causes a neuron to prevent its neighboring neurons from firing when it is itself active. In biological reality, this is usually realized by inhibitory synapses connected to the neighboring neurons. These PCs in combination with lateral inhibition tend to form so-called Winner-Take-All (WTA) cortical microcircuits.
The main goal of the said paper was to demonstrate how memory, learning, and even certain computational abilities can emerge from WTA cell assemblies with spike-timing-dependent plasticity (STDP; explained in 2.2.2 and 3.5.2) when presented with sequential input patterns. More specifically they postulated three constraints for this behavior (Klampfl and Maass, 2013):

1. PCs and inhibitory neurons tend to be organized into specific network motifs
2. synapses between PCs are subject to STDP
3. neural responses are highly variable („trial-to-trial variability “)

By replicating the model from this paper its correctness is supposed to be verified while also providing an Open Source version of it that is accessible for further research and modification. In addition, any hidden assumptions made in the description of the original should be uncovered.
To do so, after explaining the methods of this work (Section 1.1), this thesis will first briefly discuss the biological backgrounds needed to help understand its content from the perspective of a Computer Scientist in Chapter 2. Following up on this in Chapter 3 the original model of (Klampfl and Maass, 2013) will be discussed in more detail and the exact model definitions will be laid out and put to inspection. This leads over to the summary of the implementation produced for this thesis (Chapter 4), building on the original implementation and pointing out justified alterations and different technical approaches. Finally, the results of both implementations will be compared in Chapter 5, followed by a conclusion and an outlook to future work (Chapter 6).

1.1 Methods

As already mentioned, this replication study should be viewed as part of a group of studies conducted as part of bachelor theses at the INM-6 (Institute for Medicine and Neuroscience, Computation in Neural Circuits group). All these studies have in common that they are implemented using NEST (Spreizer et al., 2022) as their simulation framework to be able to compare the studied models in a more simulator-agnostic way.
Kindly the authors of (Klampfl and Maass, 2013) agreed to share the Python 2.7 implementation code of their work, so when the implemented model deviates from the one in the paper, the implemented model is considered here, since it was used to produce the advertised results.
To study and understand the original model more closely, the first phase of this thesis was dedicated to replicating the results of the paper using their original code. The learnings of this phase are presented in Chapter 3. Along with replicating the results during this phase, it was also decided to add several means of recording and visualizing certain parameters like membrane potentials, spike activity, and synaptic weights, in order to be able to compare the results with those of the later replication of the model to verify its correctness.
After getting familiar with the original implementation, the replication phase started. For the model replication NEST 3.3 (Spreizer et al., 2022) was initially chosen but later replaced with the most recent development version at the time. NEST (**NE**ural **S**imulation **T**ool) is a simulator for spiking neural networks developed by the NEST initiative and is built to simulate various forms of learning and plasticity. While NEST itself is a standalone application operating on a C++ kernel, it is also possible to run it in Python using PyNEST. This program can translate Python commands for NEST. Due to the superior simplicity and readability of Python compared to C++, the PyNEST option was selected for this thesis.
As further elaborated later, entirely new neuron and synapse models are needed, which are not provided by NEST natively. To create these models for NEST, the modeling language NESTML (Linssen et al., 2022) was chosen. NESTML creates neuron and synapse models by translating instructions formulated in the NESTML modeling languages syntax to C++ code that integrates with the main NEST kernel. The language was created to make the

process of writing custom neuron and synapse models easier and most importantly more accessible since the construction of such models by hand requires a vast knowledge of the NEST kernel in addition to proficiency in C++ programming. Without NESTML this would further restrict users from working with NEST.
During the work on implementing the synapse and neuron models from (Klampfl and Maass, 2013), it eventually became clear that NESTML was not capable of fulfilling the requirements of the model. For this reason, NESTML could not further be used to create the models. The incomplete generated model implementations of NESTML were however used as the platform to build a highly customized C++ model by hand (described in Section 4.5).
For the Python implementation of the network NumPy and matplotlib were used to respectively help with data handling and plotting of readout data.

Chapter 2

Background

2.1 Terminology for Biological Neural Networks

To understand the concepts at work in this thesis, it is necessary to first familiarize with the biology-inspired terminology that is used to describe the roles and functions of components of the computational models that will be introduced later.
First and foremost there are the terms describing the basic components of one neuron communicating with a different neuron as illustrated in Figure 2.1. In principle, such biological neural networks only consist of two types of components, that can have highly variable properties though: synapses which establish data connections, and neurons, which serve as nodes receiving and sending signals over synapses according to a predefined rule set. The neuron on the upstream end of a particular synapse is referred to as the *presynaptic neuron*, while the direct downstream synapse is the *postsynaptic neuron*. Neurons can generally have several in- and outgoing synapses, a synapse on the other hand typically only connects two neurons in a strictly directional manner. Neurons can also be either *excitatory*, meaning that on spiking they can only excite their postsynaptic cells by increasing their membrane potential, or *inhibitory*, meaning that all their spikes inhibit the postsynaptic neuron's membrane potential from rising. This mutual exclusivity of excitation and inhibition is known also as Dale's principle in neuroscience.

If a neuron receives several spikes within a sufficiently small temporal window (dependent on the neuron's characteristics), the potentials triggered by them will be summed together. Depending on whether the incoming spikes triggering these potentials are excitatory or inhibitory, the resulting potentials are called excitatory or inhibitory post-synaptic potentials (EPSP/IPSP).

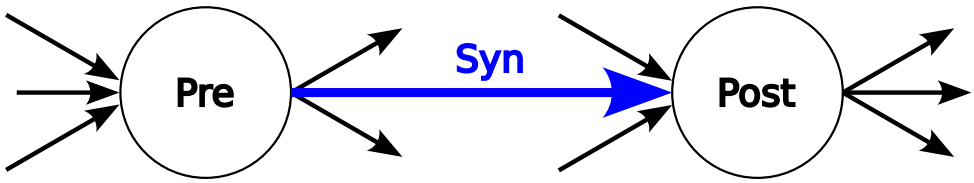

FIGURE 2.1: Basic Synaptic Connection

If the membrane potential subsequently reaches a certain threshold (called *action potential*), it will rapidly decrease its membrane potential by unloading it in the form of a spike. Over time the membrane potential deteriorates on itself however, so when the same number of excitatory spikes arrives at the neuron over a longer period of time, or when they are mixed with inhibitory spikes, they might not be able to trigger the action potential. The underlying dynamics of the handling of incoming spikes are referred to as *synaptic integration*, which is why neuron models that are based on this mechanic to determine when to trigger a neuron to fire are called integrate-and-fire (IAF or I&F) models.
The descriptions above are not accurate to the actual proceedings in brain cells and neglect many important biological structural components like dendrites, somata, and axons, to include them in the description of the synapse or neuron. This abstraction is made among other reasons to simplify these models to make them computationally less expensive.

2.2 Synaptic Plasticity

With the basic structural backgrounds of spiking neural networks out of the way, a synaptic characteristic of critical importance to biological neural networks and the main research object of this thesis is still remaining. The talk is about *synaptic plasticity*, which is generally assumed to be the substrate for learning and memory in cortical circuits (Morrison, Diesmann, and Gerstner, 2008).
Plasticity is a property of synapses that makes them form, strengthen, and degrade - for the most part - based on the spiking behavior of the neurons they connect. Due to plasticity being a massively complex mechanic of biological neural networks, that is composed of many different drivers for synaptic weight change, there is no easy way to describe it, which is why over the years, researchers came up with various rules and mathematical models to describe it. In the following two different kinds of plasticity will be briefly introduced.

2.2.1 Short-Term Plasticity (STP)

As the name already suggests, short-term plasticity operates on rather short timescales in the order of 100s of milliseconds. It can facilitate (STF) or depress (STD) synaptic connections based on the recent spiking history of a presynaptic neuron. A qualitative illustration of STP can be seen in Figure 2.2 (Morrison, Diesmann, and Gerstner, 2008), where t^f denotes an example spike at time t and $x(t)$ is the corresponding postsynaptic membrane potential resulting from the spike.
Generally, when a neuron receives a spike, its membrane potential also spikes up and then slowly decreases again as visualized by $x(t)$. Without STP, spikes with very short interspike intervals (ISIs) would continuously increase this way. But based on whether the synapse is depressing or facilitating this behavior is changed. To be more precise, STP in depressing synapses causes the increase in membrane potential to stagnate when many spikes come in with short ISIs by amplifying the decrease after an incoming spike for a short window of time. STP on facilitating synapses on the other hand causes the postsynaptic potential to decrease slower the smaller the ISIs of the incoming spikes are.

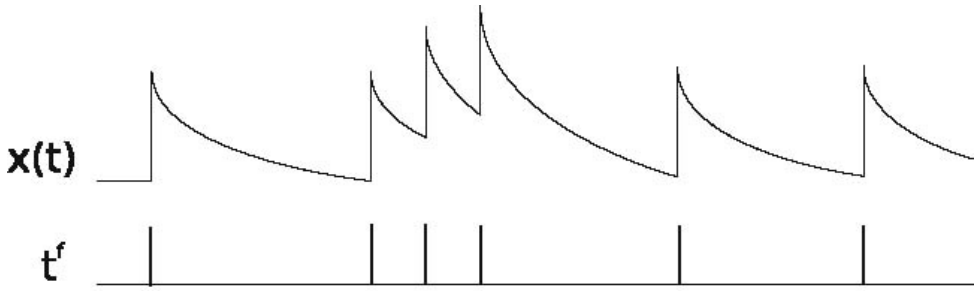

FIGURE 2.2: STP Example (Morrison, Diesmann, and Gerstner, 2008)

2.2.2 Spike-Timing-Dependent Plasticity (STDP)

Spike-timing-dependent plasticity is a form of structural plasticity that, unlike STP, allows long-term consolidation of learned behavior by adjusting the efficacy of the synapse based on the learned correlation of firing behavior. This can be better described by Hebb's postulate, an early rule for synaptic plasticity that applies well to STDP:

> *When an axon of cell A is near enough to excite a cell B and repeatedly or persistently takes part in firing it, some growth process or metabolic*

> *change takes place in one or both cells such that A's efficiency, as one of the cells firing B, is increased.*

Thus, when a postsynaptic neuron receives a spiking input from a presynaptic source shortly before firing as illustrated in Figure 2.3, the source is more likely to have a causal effect on the postsynaptic neuron, which justifies the facilitation of the connecting synapse. On the other side, if the spiking behavior of the postsynaptic neuron and the source are independent, the connection is degraded. This is a form of correlation-based unsupervised learning, but due to the popularity of Hebb's postulate, it is also referred to as Hebbian learning (Morrison, Diesmann, and Gerstner, 2008).

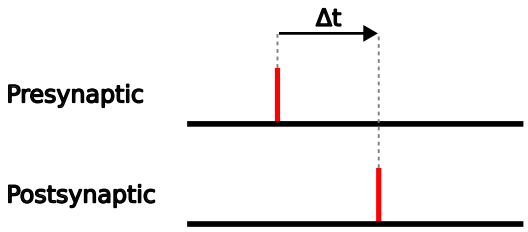

FIGURE 2.3: Pre- and Postsynaptic Spikes

Chapter 3

Original Work

3.1 Outline

The original work of (Klampfl and Maass, 2013) that is replicated in this paper is based on the *liquid computing model*. This computing paradigm is loosely inspired by biological brains and offers a way of implementing Spiking Neural Networks (SNNs). It consists of a collection of nodes (that resemble neurons) that can send and receive temporal signals (also called spikes) to and from other nodes via dedicated connections (which resemble synapses). These connections - just like the nodes - can have various other properties, such as delay or weight that influence how spikes are transmitted over the network of nodes. Additionally, neurons also possess a spatial coordinate that is necessary for having distance-dependent connection probabilities just like in the brain.

The conventional liquid computing model abstracts a fair amount of detail however, when compared to real cortical circuits. This includes a major structural trait inherent to the brain, that is assumed to enable long-term memory by forming stereotypical assemblies of neurons: synaptic plasticity. In the original paper, two forms of plasticity are used: short-term plasticity (STP) and spike-time dependent plasticity (STDP).

This chapter will first describe the initial structural setup of the network from the original implementation in Section 3.2 and then lay out how the generation and distribution of outside input is handled in Section 4.3. Finally, the dynamic properties that give the model its actual capabilities are discussed in Section 3.5.

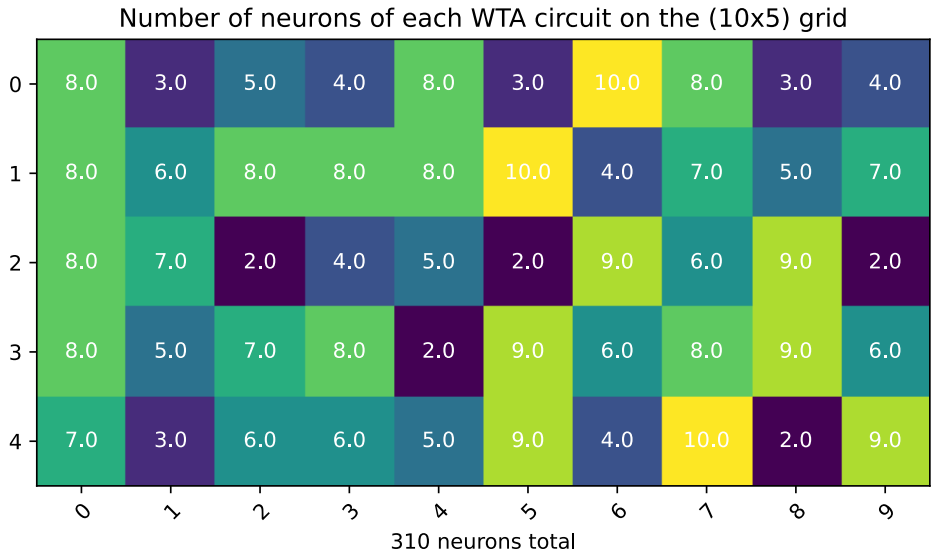

FIGURE 3.1: Visualization of a random 10×5 WTA circuit grid with $k_{min} = 2, k_{max} = 10$

3.2 Basic Network Structure

The basic setup of the liquid computing model employs a 2D grid of so-called Winner-Take-All (WTA) circuits, usually set to a size of 10×5 as in the example grid in Figure 3.1. The 2D grid is utilized to assign spatial coordinates to each WTA, for the purpose of constructing inter-circuit connections with distance-dependent probability. The reason for this is that it has been found that in the brain, the probability that two neurons are synaptically connected decreases as their spatial distance increases. To describe this structural property mathematically, there exist multiple different rules including the following chosen for the original implementation:

$$p(d) = \lambda e^{-\lambda d} \tag{3.1}$$

where d is the euclidean distance between two WTA circuits and λ is a generic setting parameter for adapting the connectivity rule to work well with the given grid size. In the original implementation $\lambda = 0.088$.
The WTA circuits themselves are each composed of neurons $z_1, \ldots, z_K$, where the integer K is uniformly drawn at random from a range $[k_{\min}, k_{\max}]$. All neurons $z_1, \ldots, z_K$ in the same WTA are not laterally connected and are subject to a mechanic called lateral inhibition. This is illustrated in Figure 3.2.

Lateral inhibition causes that when one neuron in the WTA circuit fires, all other neurons in the same circuit experience inhibition, i.e. are discouraged from also firing. The biologically most accurate solution for implementing it would be to introduce inhibitory neurons that receive input from the entire WTA and output inhibitory spikes to the same circuit (as seen in Fig. 3.2). In (Klampfl and Maass, 2013) this is however said to be realized in a different way, by adjusting the firing rate of each neuron k according to the summed firing rate of the WTA like this:

$$r_k(t) = R_{\max} \cdot \frac{e^{u_k(t)}}{\sum_{j=1}^{K} e^{u_j(t)}} \tag{3.2}$$

$R_{\max}$ is a fixed value that defines the maximum firing rate and is usually set to 100 Hz. $u_k(t)$ on the other hand stands for the current membrane potential of neuron z_k at time t and will be explained in more detail in Section 3.5. This realization of lateral inhibition bypasses the use of dedicated inhibitory neurons for lateral inhibition but is also responsible for normalizing the firing rate of the combined WTA circuit to $R_{\max}$, even when no external inputs are present. In principle, this is an unwanted side-effect, but according to (Klampfl and Maass, 2013), this is insignificant to the results of their study. Replacing this mode of lateral inhibition might be interesting for future work with the model from their paper.

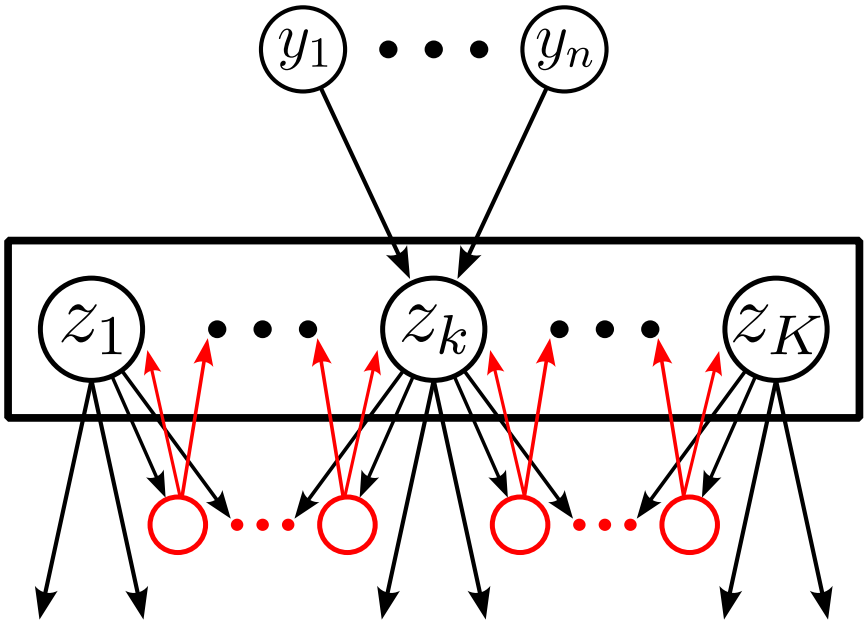

FIGURE 3.2: WTA circuit with lateral inhibition

3.3 Running the Network

The original model offers two different functions of running the network and producing readouts: `test()` and `simulate()`. The core difference is that during testing the network is not trained, i.e. the usual dynamic behavior resulting from STDP is disabled so that weights are not updated while running. STP is always enabled, while the use of variance tracking (described in Section 3.5.2) is optional but seems to be a requirement for learning with STDP, based on trials conducted with the original code. These findings are further discussed in Chapter 5.
Apart from the enabling of STDP, `test()` and `simulate()` are similar and share the same basic functionality. The network is run by progressing it step by step, where one step usually represents a time span of 1 ms (depending on the value of `dt`), after which the network states are updated according to the rules specified in Section 3.5.

3.4 Input Generation

To train the network properly for different tasks, the generation of input is key. In its simplest form, this is implemented as the presentation of recurrent pattern inputs, as illustrated in Figure 3.3. More complex sequences can also be generated based on this, but this will be briefly discussed in Chapter 4 as it is not crucial for the tests performed in this thesis. The patterns here (represented by the grey boxes) consist of a number of input spike trains, where each input source gets its own Poisson spike train generated. Poisson generation is essentially realized by randomly sampling a list of values from an exponential distribution

$$f(x; \frac{1}{\beta}) = \frac{1}{\beta} \exp\left(-\frac{x}{\beta}\right), \tag{3.3}$$

where β corresponds to the firing rate in Hz of each spike train in that pattern. This list is then cumulatively summed, i.e. the list of values $v_1, \ldots, v_k$ is updated as follows: $v_n\prime = \sum_{i=0}^{n-1} v_i$ to obtain an ascending list $v_1\prime, \ldots, v_k\prime$ of spike times with interspike intervals (ISIs) drawn from Equation 3.3. This pattern generation has to be done once for every pattern before the network is run with pattern inputs.
These pattern inputs are however not the only kind of input that the network neurons receive. Unless disabled, they also receive random noise of

frequency `rNoise` [Hz], which is constantly overlaid. The resulting two different types of total input can therefore be distinguished into noise and pattern phases of duration `tNoise` and `tPattern` respectively, where `tNoise` is uniformly drawn at random from the pre-specified interval `tNoiseRange`.
As already explained in Section 3.3, the simulation is progressed stepwise. The same goes for the presentation of input. During each simulation step a function `generate(t)` is run on the input generator object, where `t` is the current time step of the simulation, displayed conceptually as the red dotted line in Fig. 3.3. For most decisions, the input generator works with the current time relative to the beginning of the last pattern: `tp=t-last_pattern_start`. A typical cycle for the simple example would for example be: present current pattern while `tp`$\leq$`tPattern`. As soon as `time_to_draw=0`, the `last_pattern_start` is set to `t` and the pattern gets presented again for `tPattern`.

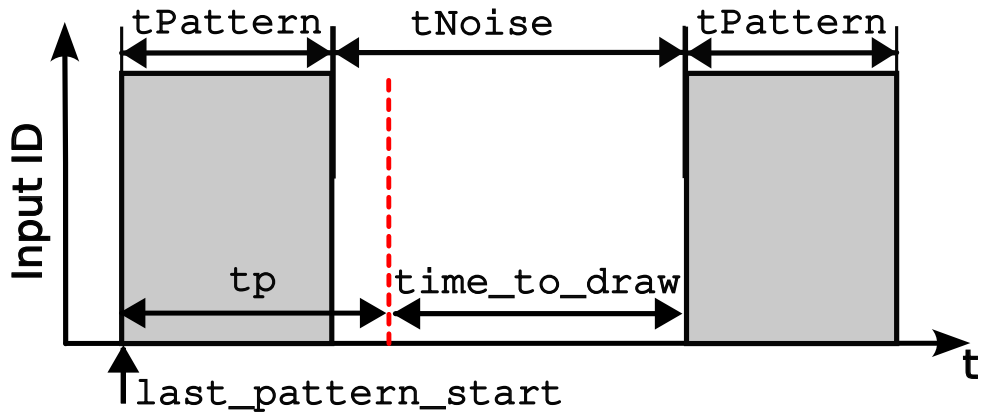

FIGURE 3.3: Pattern Input Generation

3.5 Dynamic Network Behaviour

Now that the foundations of the network have been explained, it is time to dive into the actual dynamics which implement the models of plasticity and with them the ability of sequential learning. The cornerstone for neuron dynamics is the rule for determining the current membrane potential $u_k(t)$ of neuron z_k (excluding the effect of lateral inhibition, which is added in 3.2):

$$u_k(t) = \sum_{i=1}^{n} w_{ki} y_i(t) + w_{k0}. \tag{3.4}$$

Here w_{ki} is the synaptic weight of an input synapse to neuron z_k from presynaptic neuron y_i, and w_{k0} is a bias parameter of the k-th neuron. This

parameter has, however, no effect on the membrane potential in the implementation since it is initially set to zero and never updated. It is therefore ignored for the rest of this thesis. Finally, $y_i(t)$ fittingly denotes the current value for the excitatory postsynaptic potential (EPSP) of presynaptic neuron y_i. This potential is modelled as an α-shaped kernel with rise time constant $\tau_{\text{rise}} = 2\,\text{ms}$ and decay time constant $\tau_{\text{decay}} = 20\,\text{ms}$ as follows:

$$y_i(t) = \sum_{t_p} \exp\left(-\frac{t - t_p}{\tau_{\text{decay}}}\right) - exp\left(-\frac{t - t_p}{\tau_{\text{rise}}}\right) \tag{3.5}$$

$$\left(= \underbrace{\sum_{t_p} \exp\left(-\frac{t - t_p}{\tau_{\text{decay}}}\right)}_{\approx \texttt{epsp}} - \underbrace{\sum_{t_p} exp\left(-\frac{t - t_p}{\tau_{\text{rise}}}\right)}_{\approx \texttt{epsp2}} \right)$$

where t_p is the set of presynaptic spike times at time t. This EPSP is also implemented somewhat differently from the mathematical description in Equation 3.5, which is why it was decided here to add the conversed equation for $y_i(t)$ in braces. As pointed out there, the parts of the conversed equation are realized in the code by the two variables `epsp` and `epsp2`, standing for the current EPSP values when solely considering the EPSP resulting from decay and rise respectively. While they serve the same purpose as their representations in the equation they work differently. Essentially, each of these ESPS variables decrease over time by

$$\texttt{epsp}_t = \texttt{esps}_{t-1}\left(1 - \frac{\texttt{dt}}{\texttt{tau}}\right) \tag{3.6}$$

with `dt`=0.001 the simulation time step in seconds and `tau` the time constant of the `epsp` (either τ_{rise} or τ_{decay}). With the decrease in EPSP for the rise and decay phases explained this still leaves the EPSP-increasing behavior undescribed. Although this is not directly mentioned in (Klampfl and Maass, 2013), this is in fact done by the STP mechanic described in the paper, which is why this will be further discussed in Section 3.5.1. All that will be said about this here is that in the case of an incoming spike, the EPSP for both `epsp` and `epsp2` will be updated by $\Delta\texttt{epsp} = u \cdot R$ in addition to the EPSP decreasing behavior with $u, R \in [0, 1]$ being dynamic variables that are defined in Equation 3.7.

3.5.1 STP

STP is a form of plasticity that changes the efficacy of a synapse on the scale of a few hundred to thousands of milliseconds (Stevens and Wang, 1995). It can be further categorized into Short-Term Facilitation (STF) and Short-Term Depression (STD) and generally causes the strength of an incoming spike to be determined also by the previous recently incoming spikes, instead of simply the synaptic weight.
As is clear from Equations 3.4 and 3.2, there are two options in the current model to manipulate the dynamics of firing rates: using synaptic weights or EPSPs. Because STP only temporarily changes the synaptic efficacy, it makes more sense to implement it by manipulating EPSPs, which is what happens in (Klampfl and Maass, 2013). They determine the amplitude A_k of the k-th input spike as follows:

$$\begin{aligned} A_k &= w_k \cdot \overbrace{u_k \cdot R_k}^{=\Delta\text{epsp}} \\ u_k &= U + u_{k-1}(1-U)\exp(-\Delta_{k-1}/F) \\ R_k &= 1 + (R_{k-1} - u_{k-1}R_{k-1} - 1)\exp(-\Delta_{k-1}/D) \end{aligned} \tag{3.7}$$

It is important to define the notion of "amplitude" more clearly here. Amplitude in this context refers to a combined metric of synaptic weight and EPSP associated with each given spike. The EPSP that is calculated by $u_k \cdot R_k$ was already introduced when describing the EPSP in Equation 3.5 and is responsible for the EPSP increase on spike input.
The interspike intervals of the entire spike train up to the k-th spike are given by $\Delta_1, \Delta_2, \ldots, \Delta_{k-1}$, while D and F are STD- and STF-related time parameters. U, D and F all have in common that they are set randomly for each synapse from Gaussian distributions $\mathcal{N}(\mu, \sigma^2)$ with $\sigma = 0.5\mu$ and $\mu_U = 0.5$, $\mu_D = 0.11$, $\mu_F = 0.005$. In the implementation the additional constraints $U \geq 0$, $D, F \geq \texttt{dt}$ are added to ensure that the amplitude does not take on unintended values.
Klampfl and Maass stated they adopted this short-term plasticity model from (Markram, Wang, and Tsodyks, 1998). However, on comparison, it is noticeable a detail was altered when doing so. Instead of the definition of R_k from Equation 3.7, the following definition would conform with their cited model:

$$R_k = 1 + (R_{k_1} - \mathbf{u_k} R_{k-1} - 1)\exp(-\Delta_{k-1}/D)$$

with u_k instead of u_{k-1}. It is unclear if this detail has any significant effect and if it was an intentional decision that was simply not explained by Klampfl and Maass, but it is worth mentioning and should be considered when using their model for future work.
Another mistake of the computational kind was also found in the original implementation. Usually, only recurrent connections should be subject to STP, but due to an error in the calculation of R_k, STP was partially applied to input connections as well.

3.5.2 STDP

STDP, unlike STP, is a long-term plasticity that is based on temporal differences between pre- and post-spikes. More precisely, this means that whenever a postsynaptic neuron z emits a spike at time t_{post} and receives an incoming spike from presynaptic neuron y at time t_{pre}, then the weight of the connecting synapse is increased if $t_{pre} < t_{post}$ and decreased if $t_{pre} \geq t_{post}$. There are many different rules that model STDP, but Klampfl and Maass stated they wanted to implement it using the following weight update rule:

$$\Delta w_{ki} = y_i(t) \cdot c \cdot e^{-w_{ki}} - 1 = \frac{c \cdot y_i(t) - e^{w_{ki}}}{e^{w_{ki}}}, \tag{3.8}$$

where w_{ki} denotes the weight of the synapse input to neuron z_k from presynaptic neuron y_i. This weight update should be performed every time neuron z_k fires. Furthermore, $y_i(t)$ denotes the current value of the EPSP at presynaptic neuron y_i as formulated in Equation 3.5 and c is a scaling parameter used to keep the synaptic weights in a positive range. This parameter is comprised of the learning rate $\eta^* = 0.05$ and an offset of 5 like this: $c = 0.05 \cdot \exp(5) = 7.42$.
Obviously, this STDP model does not make use of t_{pre} and t_{post} directly, which is somewhat unusual. It rather exploits the fact that the EPSP $y_i(t)$ of the presynaptic neuron should be high if it has recently spiked, and low if it has not spiked in the recent past. To better visualize STDP a common way is to plot an STDP window. STDP windows can be created practically by setting a fixed spike time for the post-spike of a neuron and confronting it with varying pre-spike times. For each pre-/post-spike combination, the resulting weight change is plotted. For the STDP rule from Equation 5 of (Klampfl and Maass, 2013) this produces the STDP window in Figure 3.4. As illustrated there, the weight is reduced by a constant of -1 every time a pre-spike arrives earlier than the post-spike is emitted and is only ever increased if the

pre-spike comes in later, but only for a brief period of time.

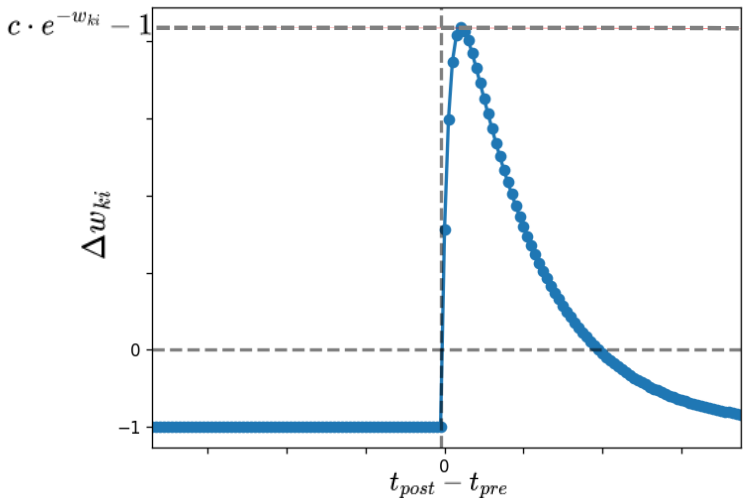

FIGURE 3.4: STDP window

It is also claimed in the original paper that the bias parameter from Equation 3.4 is updated by the following rule:

$$\Delta w_{k0} = \begin{cases} c \cdot e^{-w_{k0}} - 1 & \text{if neuron } z_k \text{ fires} \\ -1 & \text{else} \end{cases}$$

However, this could not be found in the implementation code, which is why it will not be regarded for the replication. Apart from this, the actual implementation of Δw_{ki} also seems to deviate slightly from Equation 3.8:

$$\Delta w_{ki} = \eta_{ki} \cdot \frac{y_i(t) - e^{w_{ki}}}{\max\{e^{w_{ki}}, \eta_{ki}\}} \tag{3.9}$$

The reason for this is that Klampfl and Maass decided to also adopt adaptive learning rates for synapses from (Nessler et al., 2013). This is handled by the synapse-specific learning rate η_{ki}, which replaces the static parameter c.

Variance Tracking

The integration of adaptive learning rates is done by a heuristic called variance tracking, described in (Nessler et al., 2013). Its purpose is to facilitate the spontaneous reorganization of the learned models (encoded in the form of synaptic weights) in the event that the input distribution $p^*(\mathbf{y})$ changes.

The input distribution $p^*(\mathbf{y})$ would for example change when a new different spike pattern would be presented to the network for learning.
This variance tracking is defined in Equation 71 of (Nessler et al., 2013):

$$\eta_{ki}^{new} = \frac{\mathrm{E}[w_{ki}^2] - \mathrm{E}[w_{ki}]^2}{e^{-\mathrm{E}[w_{ki}]} + 1} = \frac{\mathrm{Var}(w_{ki})}{e^{-\mathrm{E}[w_{ki}]} + 1}, \tag{3.10}$$

which is used to calculate $\eta_{ki} = \eta^* \cdot \eta_{ki}^{new}$. In the following let $S_{ki} := \mathrm{E}[w_{ki}]$ and $Q_{ki} := \mathrm{E}[w_{ki}^2]$, as these variables are used in the original implementation. The way their values are determined there is by summing over the presynaptic spike times t_p of their associated synapse up to the current point in time:

$$\begin{aligned} S_{ki}^{(t)} &= \sum_{t_p}^{t} \eta_{ki}^{(t_p)} \cdot (w_{ki}^{(t_p)} - S_{ki}^{(t_p)}) \\ Q_{ki}^{(t)} &= \sum_{t_p}^{t} \eta_{ki}^{(t_p)} \cdot (w_{ki}^{2\,(t_p)} - Q_{ki}^{(t_p)}) \\ \text{with } S^{(0)} &= 0 \text{ and } Q^{(0)} = 1 \end{aligned} \tag{3.11}$$

3.6 Summary of Complications

While working on this thesis, several major obstacles were encountered. Those specific to the paper and the associated code base are briefly summarized in this section for better outlining of difficult problems to avoid future issues.

- The rates described in the paper differ from the implementation. This is further discussed in Section 4.3.
- Despite what the paper states about neurons being individually connected with probability $p(d)$, in the code only entire WTAs are connected to each other with all-to-all connections between their respective sets of neurons.
- As discussed in Section 3.5.2, the actual implementation of STDP differed clearly from the description in the paper.
- There was also an implementation error in the STP, where STP was partially applied to input synapses, which is not intended (mentioned in Section 3.5.1)
- The used STP rule in both paper and code is adopted incorrectly from the cited source in the paper (see Section 3.5.1)

Chapter 4

Implementation

4.1 Overview

Most of the implementation apart from the custom neuron and synapse models is written in Python 3.9. This offers the opportunity to create an object-oriented implementation, loosely inspired by the original work of (Klampfl and Maass, 2013). The fundamental object classes this project uses are:

- `WTACircuit`: Serves as the basic component to organize sets of neurons that are supposed to form the same WTA circuit. (Section 4.2)
- `Network`: The centerpiece of this project, implementing the grid composition of `WTACircuit` objects as well as the inter-network neuron connections. (Section 4.2)
- `InputGenerator`: Responsible for the setup and tuning of pattern and noise input to the network during simulation. (Section 4.3)
- `Recorder`: Manages the simulation of the associated `InputGenerator` and `Network` as well as the recording of Network properties like spiking input, network spike behavior, membrane potential, EPSP, and weight. Also responsible for plotting the recorded data. (Section 4.4)

These object classes can be seen in Figure 4.1 where their relation to each other, structure, and function is conceptually illustrated. It is important to point out, however, that the Python part of this project is only responsible for the organization, creation, and manipulation of NEST components like neurons, synapses, spike generators, and measurement devices. Most of the heavy lifting in the background required to operate and simulate these is outsourced to NEST. The source code for this thesis can be found in (Michau and Zajzon, 2022).

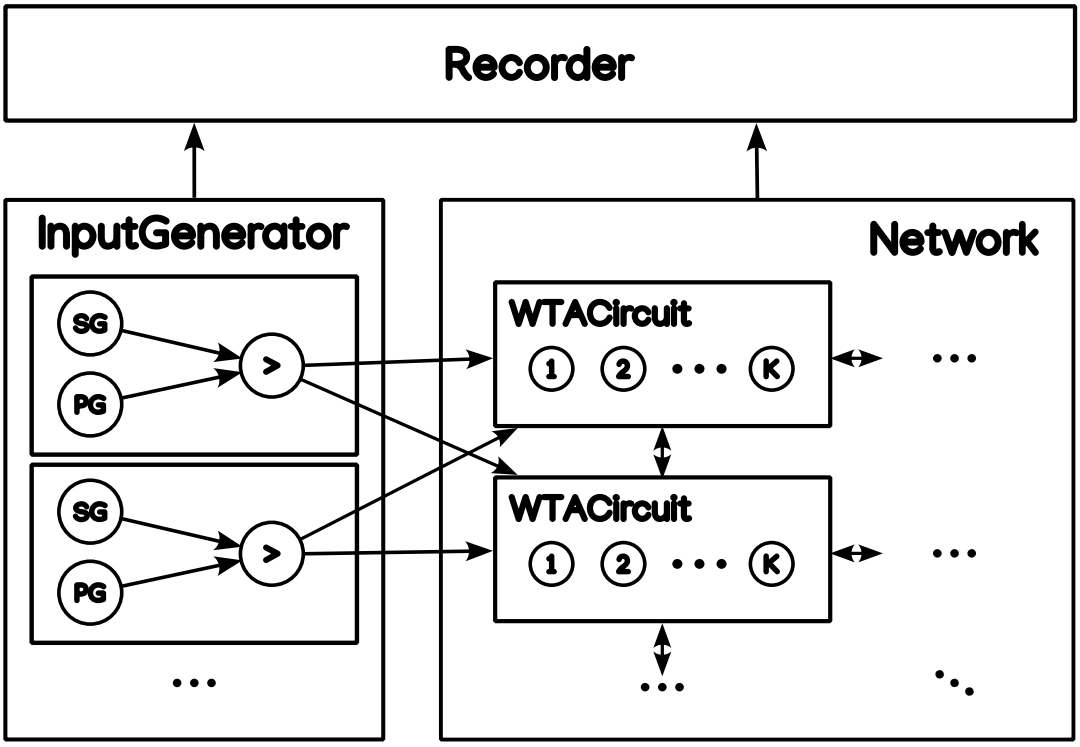

FIGURE 4.1: Conceptual Drawing of Implementation. Definition of terms: **SG** denotes 'Spikegenerator', **PG** denotes 'Poissongenerator' and **>** denotes 'Parrot neuron'.

4.2 WTA Circuit Network

4.2.1 Grid

To model the distance-dependent connection probability described in Equation 3.1 it is necessary to first assign each neuron in the network a spatial coordinate to determine the distance between different WTA circuits. While NEST provides the possibility to construct spatially structured spiking neural networks, together with a range of supporting functionality that is needed to replicate the model from (Klampfl and Maass, 2013), some details stand in the way of their use. First and foremost, the NEST spatially structured networks are designed to be biologically plausible, i.e. work in three dimensions, as do all their functionalities like e.g. functions to determine distances. Meanwhile, the (Klampfl and Maass, 2013) model is rather implemented using a 2D grid where the coordinates are assigned on a per WTA basis, leading to multiple neurons having the same coordinates. As this would be difficult to implement in NEST, a simpler approach was pursued instead.
Instead of using NEST's spatially structured network functionality, a custom `WTACircuit` object (Figure 4.2) was introduced, featuring a 2D-coordinate `pos`

indicating the position on the grid and a NodeCollection of size k:

WTACircuit
nc : **NodeCollection** pos : **tuple** k : **int**
__init__(nc: **NodeCollection**, pos: **tuple**) form_WTA() → **NoneType** get_pos() → **tuple** get_x() → **int** get_y() → **int** get_node_collection() → **NodeCollection** get_size() → **int**

FIGURE 4.2: `WTACircuit` Class Diagram

As can be seen from Figure 4.2, the `WTACircuit` object is initialized using a position tuple and a NodeCollection. This NodeCollection type is how NEST handles neurons of any kind that have been previously created. The initialization function `__init__()` also instantiates `k` with the size of the NodeCollection `nc` using `get_size()` and most notably connects every neuron in `nc` to every other neuron via an InstantaneousRateConnection. These connections do not model any synapses and are in fact a makeshift solution that had to be manually implemented to the neuron model in C++ because they are necessary for the abstract implementation of lateral inhibition without dedicated inhibitory neurons as stated by (Klampfl and Maass, 2013), which is not provided by NEST or NESTML. These InstantaneousRateConnections were originally developed in NEST for use in continuous rate models, where they would communicate rates. For this thesis, they were modified to enable the communication of membrane potentials within the WTA circuit. This is crucial for the calculation of the firing rate (Equation 3.2) since each node has to calculate it on its own and is therefore dependent on the firing rates of its fellow neurons in the WTA.

The `Network` object in turn uses the `WTACircuits` to construct a grid of shape `n` × `m` with them, where each `WTACircuit` has its size `K` uniformly drawn at random from the interval [`k_min`, `k_max`]. After the creation of the WTA grid using `create_grid()`, all neurons in the network regardless of WTA association are connected to each other with probability derived from the distance-dependent probability rule from Equation 3.1. Additionally, the formation

Network
grid_shape: **tuple** n: **int** m: **int** k_min: **int** k_max: **int** n_inputs: **int** lam: **float** circuits: **list** save_figures: **bool** show_figures: **bool**
__init__(**kwds) create_grid() → **list** form_connections() → **NoneType** get_circuit_grid() → **numpy.ndarray** get_node_collections(slice_min: **int**, slice_max: **int**) → **NodeCollection** get_pos_by_id() → **tuple** visualize_circuits() → **NoneType**

FIGURE 4.3: `Network` Class Diagram

of *autapses* (self-recurrent connections from the neuron to itself) are disabled. Initially, the synaptic weights of these connections are set to 1, but they can also be randomized just as in (Klampfl and Maass, 2013) by assigning the weights to $-\log(x)$, where $x \in [0;1]$ is a random number.

4.3 Input Generation

On the conceptual NEST level, each `InputGenerator` consists of `n` input assemblies as in Figure 4.4. NESTs *inhomogeneous_poisson_generator* is responsible for the creation of Poisson noise of specified rates for specified time intervals. The need for doing so is discussed later. Because a Poisson generator in NEST produces unique output for each neuron it is connected to - which is to be avoided to stay accurate to the original

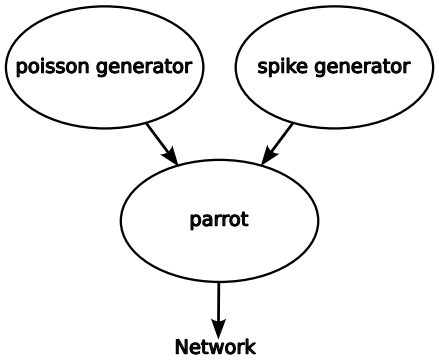

FIGURE 4.4: Single Input Assembly

implementation - an additional *parrot_neuron* is interposed between the network and the Poisson generator. A parrot neuron in NEST just repeats the spikes it receives to all its connected targets, which effectively results in each network neuron receiving exactly the same noise input from the *inhomogeneous_poisson_generator*.
The *spike_generator* handles the presentation of the pattern input as described in Chapter 4.3. Technically there is no need to connect it to a parrot, like the Poisson generator, but connecting it to the same parrot has the benefit that the input now enters the network over the same synapse. If the generator was connected to the network directly using its own synapse with plasticity, it is possible that the network might favor the pattern input over the noise by adjusting the synaptic weights, which would nullify the presence of noise in the first place. Also, this approach is more accurate to the original implementation.

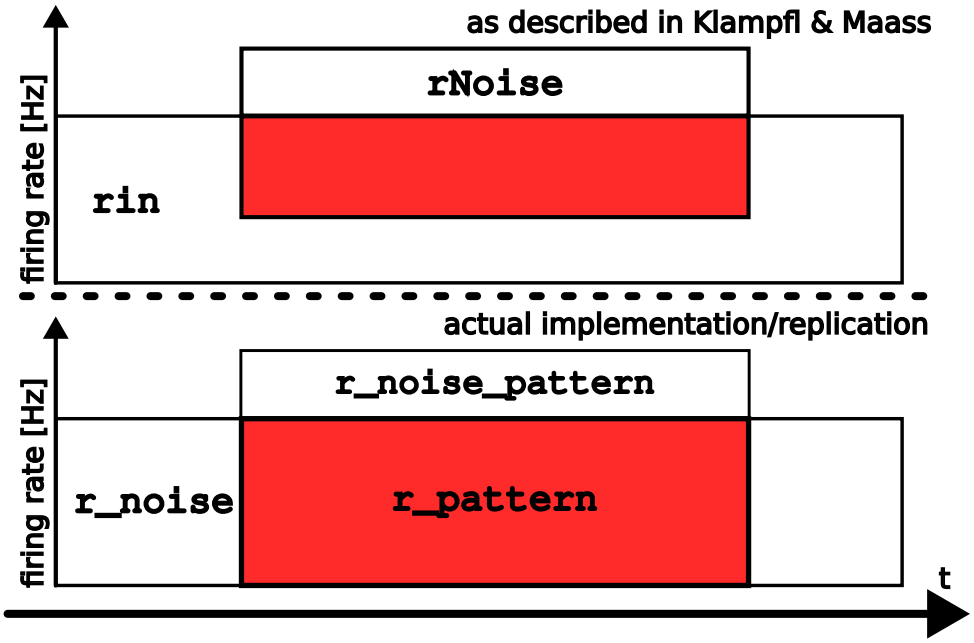

FIGURE 4.5: Input rate comparison with pattern in red and noise in white

As mentioned earlier, different noise rates are needed for different time intervals. The reason for this is that unlike the distinction in "noise" and "pattern" phases might suggest, noise and pattern input are not mutually exclusive. In fact, pattern phases are usually superimposed with additional noise, requiring different noise rates during pattern presentation. There is however a difference between the reality of both the original and replicated implementation and the description from the paper as is illustrated in Figure 4.5. According to the description of Figure 4A in (Klampfl and Maass, 2013), a pattern of

3 Hz is embedded into an input spike train with constant rate `rin`=5 Hz with additional `rNoise`=2 Hz of noise laid over pattern presentations (Figure 4.5 (top)). In the implementation of that same paper it was found however, that no embedding took place and the patterns had firing rate 5 Hz, just like `rin`. The problem with the varying noise rates can be elegantly solved, since NEST provides a spike generator that offers a simple solution to this: the *inhomogeneous_poisson_generator*. It generates Poisson noise of constant rate just like the regular *poisson_generator*, but in addition, allows for different noise rates at different intervals. This is used to specify two different noise rates: `r_noise`, which is equivalent to `rin` during the noise phase, and `r_noise_pattern`, which is equivalent to the summed rate of `rNoise` and the noise remainder of `rin` during pattern presentation. This results in the same rate behavior and noise rate as in (Klampfl and Maass, 2013), while also taking advantage of NEST and simplifying the input generation.

With the groundwork of the `InputGenerator` described, the different possibilities for input generation and general procedure can be explained now with help from Figure 4.6. The general functioning of this is equivalent to the one described in Chapter 4.3. To tie in with the paragraph above, the different firing rates `r_noise`, `r_noise_pattern`, `r_input` can be freely set, as can be the number `n` of input channel assemblies from Figure 4.4. It is however not required to play noise during simulation since it is determined by the value of `use_noise`. To make the generator produce one or more input patterns or sequences of input patterns it has to be given the total number `n_patterns` of different patterns it should create together with a duration for each of them in `t_pattern`. These are created once by `create_patterns()` and then stored for later use in `pattern_list`. The different sequences of patterns to present are contained `pattern_sequences`, a list that specifies all possible compound sequences by the indices of patterns. A possible assignment for this could be [[0, 1], [2]], meaning that possible pattern combinations to present are pattern 0 directly followed by pattern 1 and pattern 2 alone. The order in which to present these sequences is determined by the sequence switching probability `p_switch`, and `pattern_mode`, which can be set to either of two modes `'random_independent'` or `'random_iterate'`. On `'random_independent'` it selects the next sequence to present independently of previously played ones and on `'random_iterate'` the next sequence from `pattern_sequences` is presented with probability `p_switch`, or the other way around, the current sequence is repeated with probability 1−`p_switch`.

The `InputGenerator` also stores information about the beginning of noise and pattern presentation phases that are used in the `Recorder` to order the spiking neuron output by mean activation time to make assemblies of neurons better recognizable. The data needed for this is stored in `phase_times`, `pattern_trace` and `next_pattern_length`.

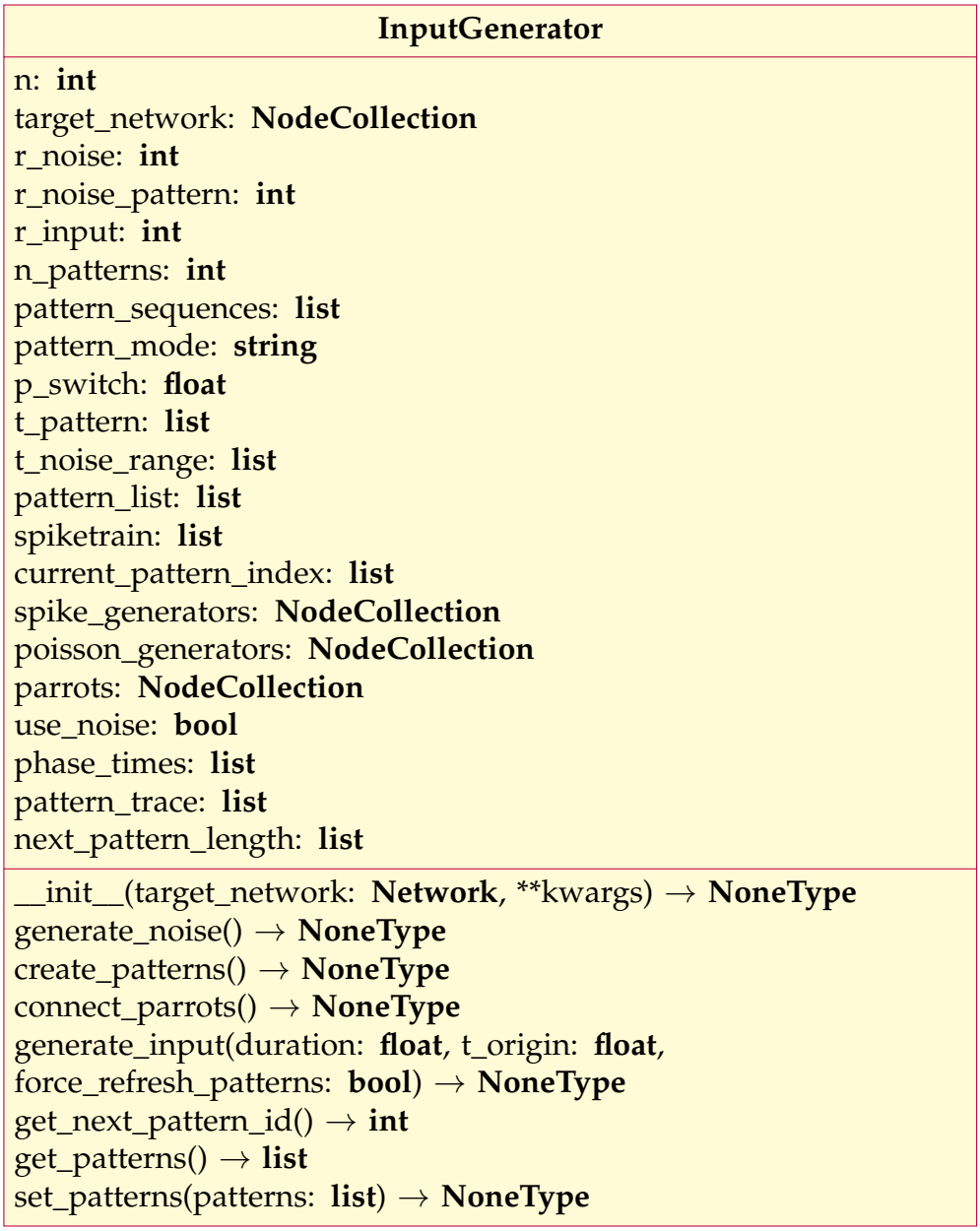

FIGURE 4.6: `InputGenerator` Class Diagram

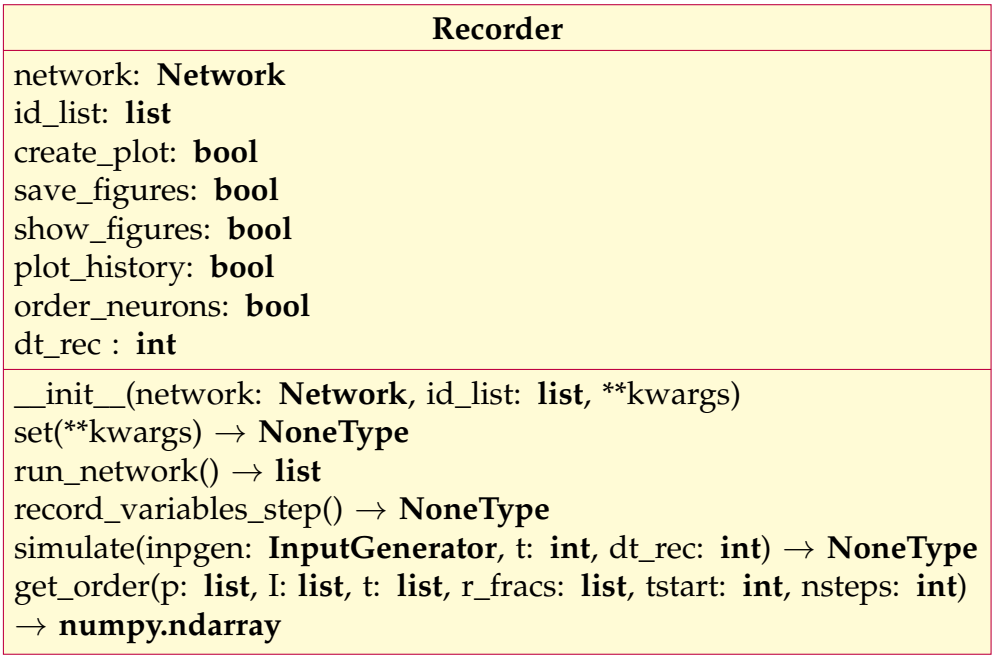

FIGURE 4.7: `Recorder` Class Diagram

4.4 Recording and Measurement

The last object class on the Python level is the `Recorder`. While the other classes are mainly used for constructive work of the network, the `Recorders` main function is to manage the simulation of that same network. For this it takes a `Network` object on construction with `__init__()`, along with an optional `id_list` that contains the global IDs of the neurons from the `Network` that should be measured and plotted later as well as a set of self explanatory Boolean parameters (see Figure 4.7) representing options for plotting outputs. Although these should be specified on initialization of a new `Recorder` object, all parameters can be changed later using the `set()` function. Since the `Recorder` handles simulation, the `run_network()` function is its main function. It takes an `InputGenerator` (although this is optional) and simulates a network for a given period of time `t_sim` with STDP/learning enabled or disabled, determined by the truth value of `train`. Because it is not always necessary or wanted to measure and plot the entire network, there are three different options besides recording the entire network:

1. "Watch List": A list of neuron IDs `id_list` to record from can be given to `run_network()`. This is a good option for recording the same neurons over several network simulation sessions.

2. "NodeCollection": A NodeCollection `node_collection`, subset of the `Network` is specified and recorded.

3. "Random *k*": *k* neurons are randomly selected from the `Network` to record from. This is useful for creating a sample to record from in the very first simulation and then reusing the same sample in the "Watch List" mode.

The general procedure of the `run_network()` function starts by setting up recording devices, i.e. NEST *multimeter* and *spikerecorder* for the network and another separate *spikerecorder* for the `InputGenerator`. The EPSPs and weights from the network are recorded by a separate `record_variable_step()` function by reading out network variables that were specifically made recordable to Python for this purpose in the C++ neuron model. Because reading out these variables does not work parallel to running the simulation, the simulation must be run in time steps of size `dt_rec`. Setting this recording time window smaller will increase the resolution of the recording but comes with the trade-off of immensely deteriorated simulation efficiency and increased runtime.
After the setup of the recording devices, the actual simulation is run using the `simulate()` function that pre-generates the patterns of the `InputGenerator` for the duration of the simulation and then commands the NEST kernel to run the simulation for increments of `dt_rec` and recording between increments until the NEST simulation has run for `t_sim`. When the simulation and data collection are finished, the results are plotted. Optionally with neurons in the *spikerecorder* readout ordered by their mean activation time during pattern presentation. This makes formed assemblies of neurons with related spiking behavior on input better visible.

4.5 Neuron and Synapse Models

As already mentioned in the Introduction, the models of the neuron and the synapse had to be implemented in C++ because NESTML did not provide the needed functionality. Before elaborating on the implementation, the concept of NEST synapse and neuron models needs to be explained. While in the original implementation all the neuron and synapse properties are handled in large arrays for the entirety of the network, in NEST each synapse and neuron is its own entity with a set of parameters. These entities are defined by models formulated in C++. Converting the original code into an object-oriented version proved to be very challenging and made it difficult

to directly compare both implementations and detect errors. This will be covered in more detail in Chapter 6.

4.5.1 Neurons

The custom neuron model follows a strict procedure to realize the same behavior as its non-object-oriented ancestor. The foundations of this procedure stem from NESTML (Linssen et al., 2022), although it was eventually abandoned as a modeling language. A neuron in NEST is generally updated on every simulation step, by the `update()` function, which first calls to `evolve_epsps()` to update the current membrane potentials as described in Equations 3.4 and 3.5. Preceding that, input in the form of both `SpikeEvents` and `InstantaneousRateConnectionEvents` (described later) is managed by individual `handle()` functions. More precisely, on an incoming event of either kind, the properties of the spike are stored for later processing (e.g. by the `update()` function).

Implementing the membrane potential in NEST required a fair amount of creativity, given that Equation 3.4, which defines it requires both the membrane potentials from all other neurons in the same WTA (although NEST neuron models have no concept of such things as WTAs) and the weights from all incoming synapses, which is not conventionally available from the postsynaptic neuron since it is a property of the synapse. These two problems made it entirely impossible to implement the model in NESTML, making the implementation even more complex and time-consuming, since the model now had to be manually written in C++ without the help of a modeling language, which was not intended when planning this thesis.
As already alluded to before, the problem of communicating the membrane potential (needed for calculating the summed firing rate of the WTA), was solved by introducing a new connection type already existing in NEST called `rate_connection_instantaneous`. Because it was originally designed for connecting rate model neurons it had to be modified to send and receive membrane potentials instead of firing rates. These connections are created *all-to-all* (excluding self loops) between the neurons in the same WTA by the `form_WTA` function on the Python level.
The second problem about the synapse weights was solved by creating a new internal vector `localWeights_Wk` that was initialized with length 1000 and filled with 0 on construction. Setting the length fixed to 1000 is not a

perfect solution because this also means that for now, each neuron can only have as much as 1000 presynaptic neurons and if it has less there may be a considerable overhead slowing down the simulation and wasting space. This can likely be fixed by using vectors of dynamic sizes and initializing them on network creation before the simulation starts. But since this was not a priority and did not cause problems for now, this remains to be implemented at a later point in time.

Following the calculation of membrane potential, `evolve_epsps()` also updates the decay-time EPSP and rise-time EPSP from Equation 3.5 using the update rule found in the original code (Equation 3.6). After the current membrane has been determined this way, the firing rate is set as defined in Equation 3.2, with two additions. The first is that the term to calculate a neuron's fraction of firing rate in the WTA

$$\texttt{rate\_fraction}_k(t) = \frac{\exp(u_k(t))}{\sum_{j=1}^{K} \exp(u_j(t))}$$

is stored in a variable for use in the ordering function to later sort the neurons by mean activation time for better identifiability of output assemblies. The second is that the firing rate of the neuron is set to 0 if the above rate fraction is larger than 1. This should be impossible, but it has been found that due to an issue with the value initialization in NEST this can occur and tamper with the correct results. Setting the rate to 0 temporarily is not technically conform with the precise model definition, but this sacrifice is justified considering it replaces a more destructive inconsistency and could not be found to alter the simulation results in any meaningful way in comparison with the original as will be shown in Chapter 5.
To make the neuron fire with the calculated firing rate a firing probability

$$p_{spike}(\texttt{rate}) = \texttt{resolution} \cdot \frac{\texttt{rate}}{1000}$$

is calculated from it and with that same probability, a spike is then scheduled to be emitted in the next simulation step by `set_spiketime`. The `resolution` here refers to the simulation time step size (set to 1 ms). Additionally, the weights are also updated by the STDP rule (Equation 3.9, including variance tracking from Equations 3.10, 3.11) when a spike is emitted, since STDP weight update should only occur on a firing postsynaptic neuron (the weight update is turned off when learning is disabled though, e.g. during testing of

a trained network). The reason the STDP weight update is performed in the neuron rather than the synapse is related to the previously described problem of calculating the membrane potential. Because the weights updated by STDP are needed for this equation and it uses the `localWeights_Wk` instead of the synapse weights, the STDP has to use them as well.
In the end of the `update()` procedure, the outgoing membrane potential communication is handled by sending an `InstantaneousRateConnectionEvent` (which is transmitted over the `rate_connection_instantaneous` connection) with the current membrane potential of the neuron.

4.5.2 Synapses

Since most aspects of the complete model, including a lot of the synapse logic, are already implemented in the NEST neuron model, the functionality covered by the synapse is decidedly smaller. An individual synapse in NEST is also accessed less frequently than a neuron because while the neuron gets updated every simulation step, the synapse is only updated on activity, i.e. when the presynaptic neuron fires. The only key characteristic from the original model it implements is the STP, which is calculated as defined by Equation 3.7.

4.6 Summary of Complications

Similar to Section 3.6, this section briefly summarizes critical problems with NEST that hindered the construction of the replication.

- communication of voltages within definable NodeCollections (for WTAs) is not supported by NEST. This required handcrafting a new connection type and embedding it into the neuron and synapse model (see Section 4.5.1).
- fetching the identity of the presynaptic neuron on spike arrival at the postsynaptic neuron is not natively supported by NEST
- managing synaptic weights from their postsynaptic neuron as required for the implementation of the model is also not natively supported by NEST

Chapter 5

Results & Comparison

To verify the correctness of the new implementation, it will be compared to the original implementation from (Klampfl and Maass, 2013), starting with toy models to prove equal behavior on the smallest possible level and proper functioning of individual properties like plasticity and lateral inhibition (Section 5.1). After that, the tested network's size will be increased until the parameters of the full-scale model described in Figure 4 of (Klampfl and Maass, 2013) are reached (Section 5.2).

5.1 Toy Models

5.1.1 Baseline Test

To start off, a minimal network of grid size 1×2, in which each WTA contains just one neuron will be tested for a singular input source over 1000 ms. The WTA size is set to 1 to bypass the effect of lateral inhibition. STP is also disabled, leaving only STDP as a dynamic network property to test. The purpose of this test is to prove the equal behavior of both implementations, which is why the embedded stochastic firing behavior of both implementations is turned off and replaced with predefined spike times, both for neuron in- and output, which were generated randomly and set to be identical for both programs externally. For the input firing rate and neuron firing rate 50 Hz and 60 Hz were chosen respectively.
The results of this test can be seen in Figure 5.1, where the pre-generated in- and output spikes are visualized along with the weights of the incoming and recurrent synaptic connections within the network and the resulting membrane potential ("Voltage trace").
From these results, we can conclude that since the plotted traces for both

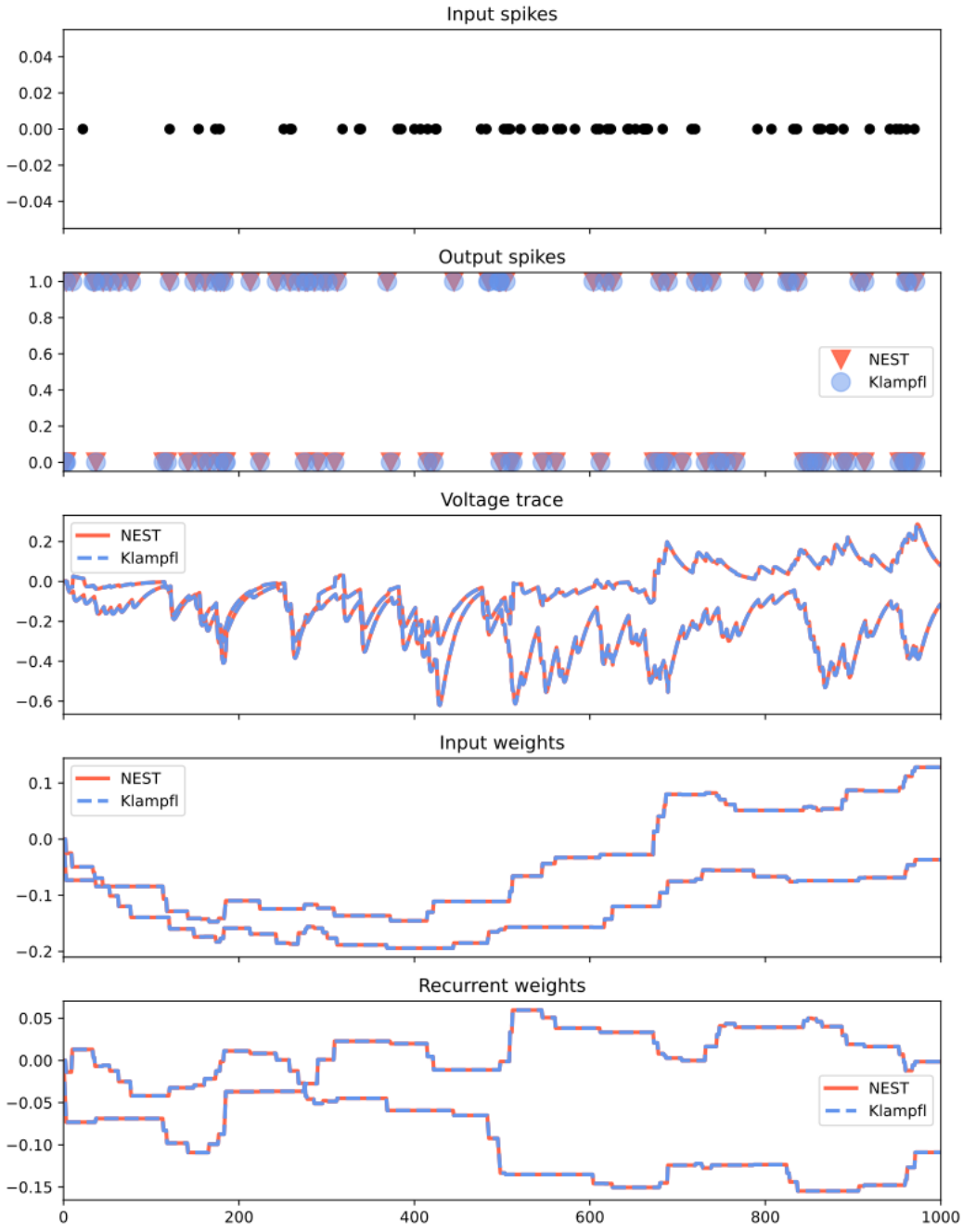

FIGURE 5.1: Toy Model: Test of 1×2 WTA grid with $k = 1$ and one input channel for 1 s. Input and output spikes are fixed at 60 Hz and 50 Hz. STP is disabled.

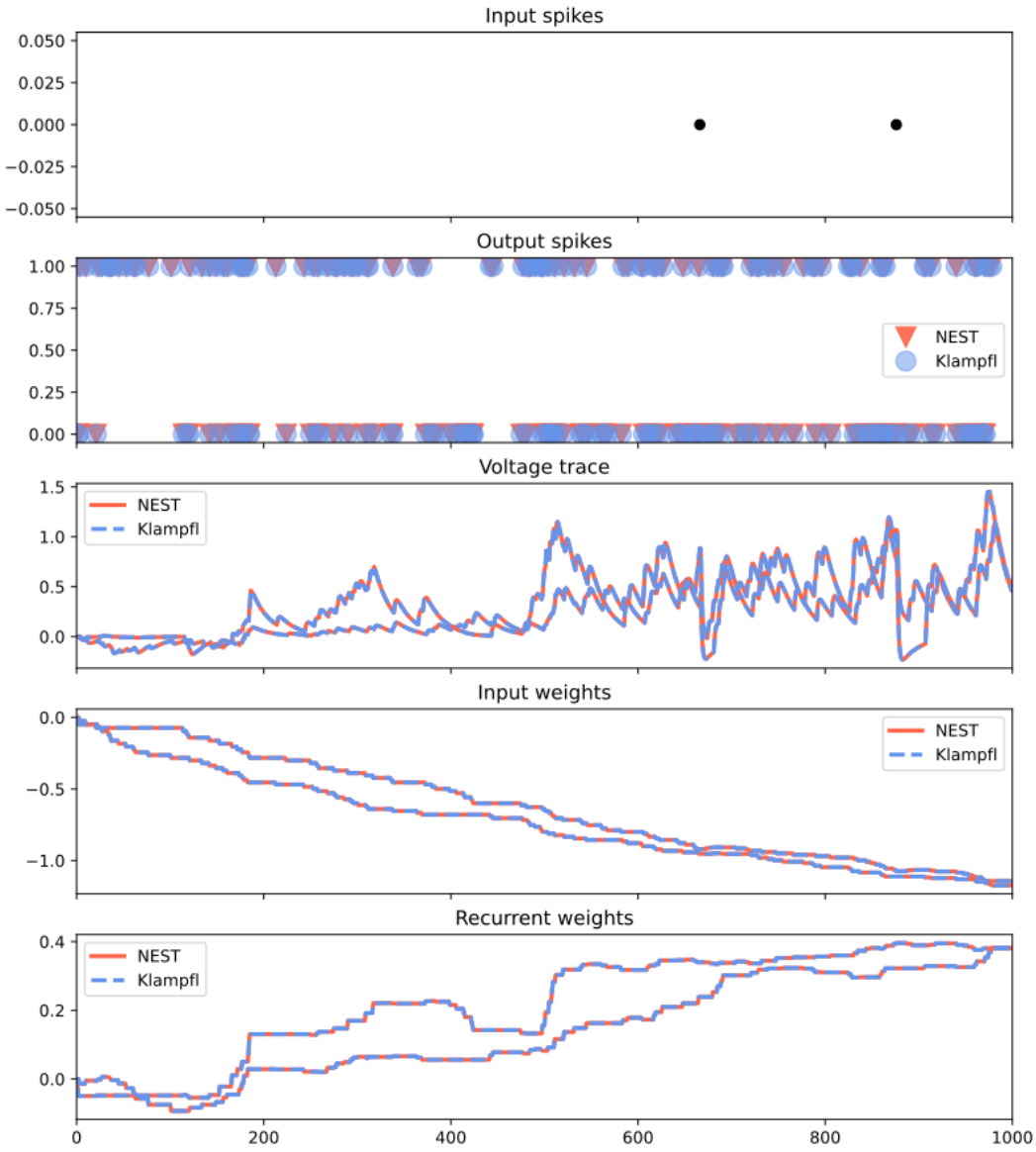

FIGURE 5.2: Toy Model: Test of 1×2 WTA grid with $k = 1$ and one input channel for 1 s. Input and output spikes are fixed at 2 Hz and 100 Hz. STP is disabled.

programs do not deviate from one another, the behavior of STDP and membrane potential in this context without lateral inhibition and STP is identical for original and replication. And since the membrane potential is computed as the sum of the EPSP scaled by the respective synaptic weights, this implies also that the EPSPs are identical. But more can be deduced from the same test setup by changing the firing rates to 2 Hz for input and 100 Hz for recurrent output spiking (Figure 5.2). With neither STP nor lateral inhibition at work, weights can switch signs depending on the rate of the external input and output firing rates of the neurons. This also shows that Dale's principle (described in Chapter 2) is not preserved when using STDP and STP disabled, since both inhibition and excitation of neurons by the same entities are possible, which is proven by the voltage traces of Figure 5.1 and 5.2 developing toward both positive and negative values. That being said, Dale's principle is not enforced in this model in any way, i.e. the weights are not bound to a specific interval during learning.

5.1.2 Scaled WTA Test

The second test (Figure 5.3) serves as an intermediate test for later comparison. It increases the WTA size from 1 in the last test to 2 and also adds a plot for EPSPs. This setup allowed testing the `rate_connection_instantaneous` along with the rate normalization in general, more specifically if the values of the `rate_fractions` were computed correctly. Most parameters are taken from 5.1.1, except for the fixed firing rates, which were set to 50 Hz and 20 Hz for recurrent and input firing rate respectively. Original and replication still produce identical results.

5.1.3 STP Test

For the next test, all parameters stay the same as in 5.1.2, except for STP which is now enabled, producing the plot in Figure 5.4 and allowing a comparison of network behavior with and without STP in combination with Figure 5.3. Notice that the addition of STP makes the recurrent weights diverge to the negative range and stabilizes the membrane potential to the negative range, whereas the voltage trace without STP contained positive valued outliers. However, weight and voltage adjustments are just the secondary effects of the EPSP being directly affected by STP, vastly reducing its maximum excitability.

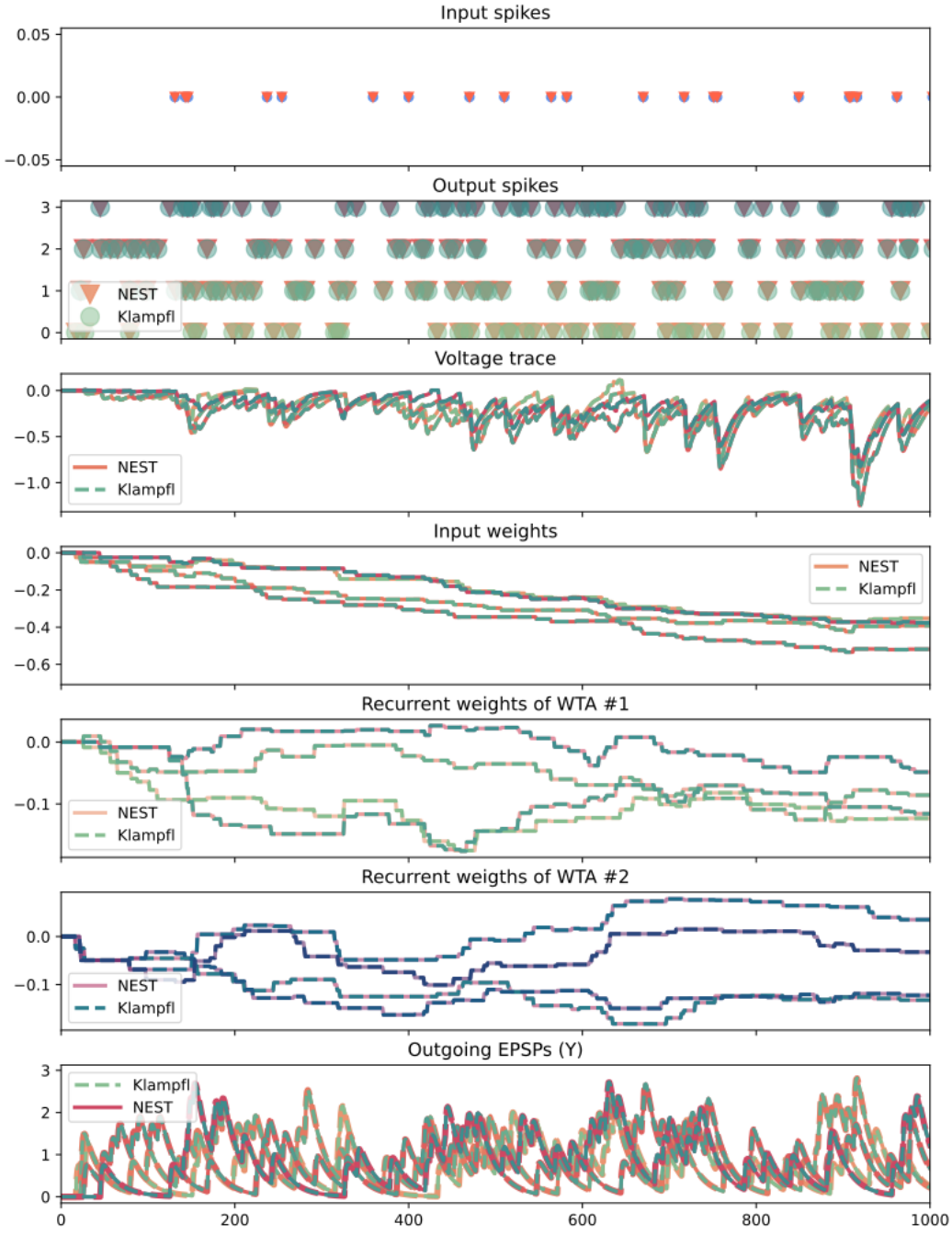

FIGURE 5.3: Toy Model: Test of 1×2 WTA grid with $k = 2$ and one input channel for 1 s. Input and output spikes are fixed at 20 Hz and 50 Hz. STP is disabled.

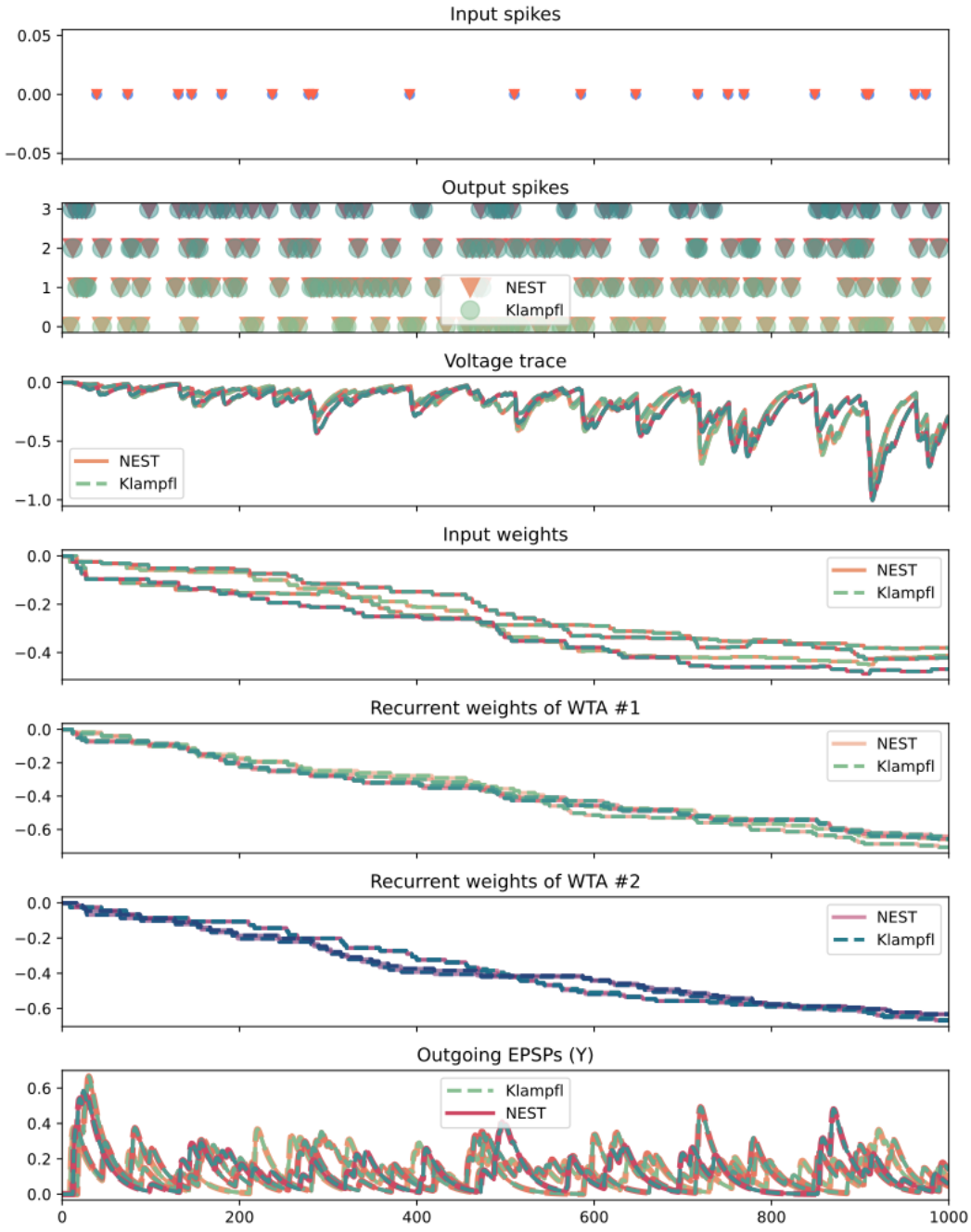

FIGURE 5.4: Toy Model: Test of 1×2 WTA grid with $k = 2$ and one input channel for 1 s. Input and output spikes are fixed at 20 Hz and 50 Hz. STP is enabled.

5.2 Full Scale Network

5.2.1 Formation of Assemblies

With the accuracy of the implementation proven for all major components of the model, it can now be compared if the full-scale models still produce the same results. For testing larger models, fixing random numbers between implementations becomes prohibitive, meaning that these tests' results are not expected to be identical and should rather show the same qualitative behavior. For the following tests, networks with the following properties will be compared in Figure 5.5 and 5.6:

Grid size	10×5
WTA size range	[2; 10]
#Input channels	100
Training time	100 s
Testing time	3 s
#Patterns	1
Noise duration range	[300 ms; 500 ms]
Pattern duration	300 ms

5.2.2 Necessity of Variance Tracking

During testing of the original implementation, it was also found that variance tracking was in fact crucial to the learning capabilities of the model as can be seen in Figure 5.7. This figure shows the results of two 3 s testing runs of the network each following a 100 s training phase. During the training phase a network of grid size 10×5 and $k \in [2; 10]$ with 100 input channels was fed with a 5 Hz of either pattern input (shown in red) or noise input (black), with 2 Hz of noise additionally laid over each pattern phase. As can clearly be seen, if variance tracking is enabled, assemblies of neurons with similar spike characteristics emerge as a consequence of pattern input, while disabling variance tracking prevents the formation of such assemblies. From this follows that variance tracking as described in (Nessler et al., 2013) is a requirement for learning in (Klampfl and Maass, 2013), while the STDP rule from Equation 3.9 alone does not appear to be able to make such learning possible.
This phenomenon can also be replicated in the NEST implementation as demonstrated in Figure 5.8.

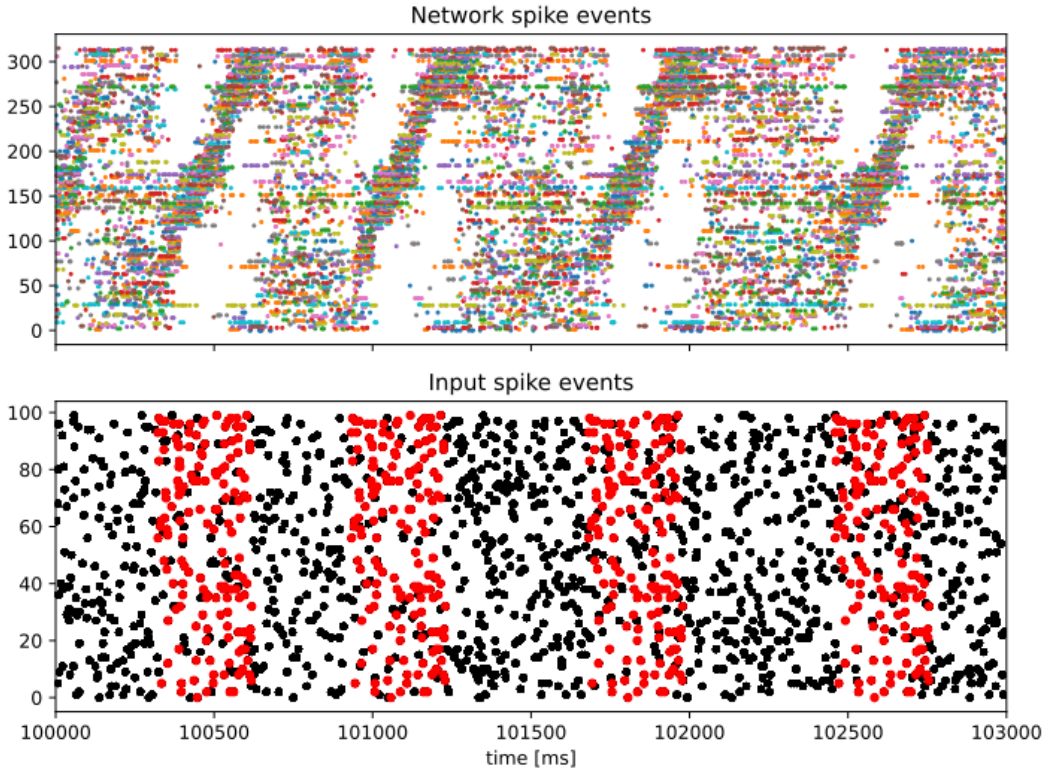

FIGURE 5.5: Full-scale Network: NEST 3 s test after 100 s training

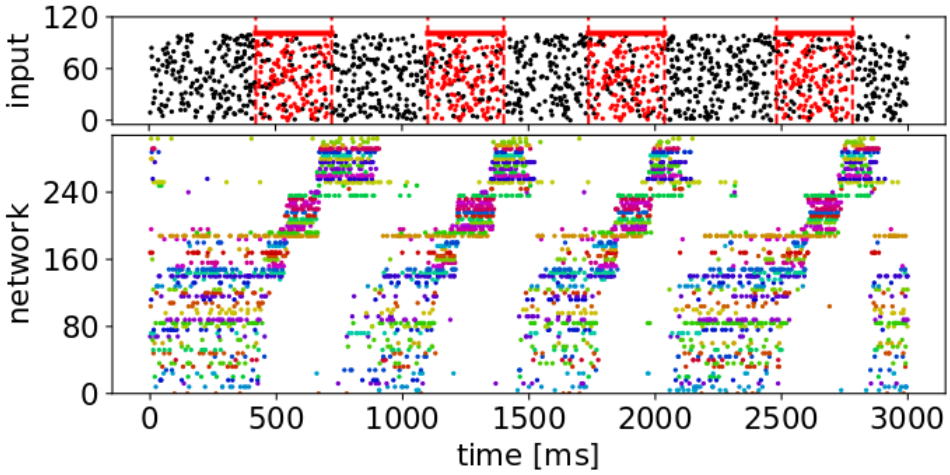

FIGURE 5.6: Full-scale Network: Legacy 3 s test after 100 s training. Only 100 neurons plotted to reduce clutter in network spikes plot.

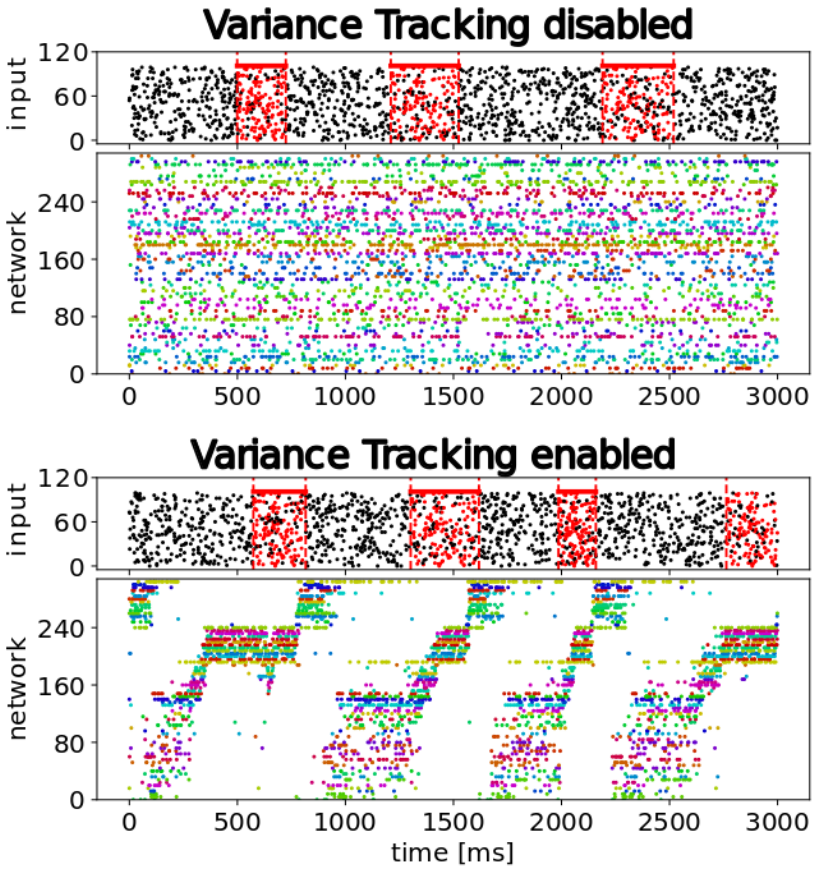

FIGURE 5.7: Full-scale Network: Legacy 3 s test after 100 s simulation with and without variance tracking

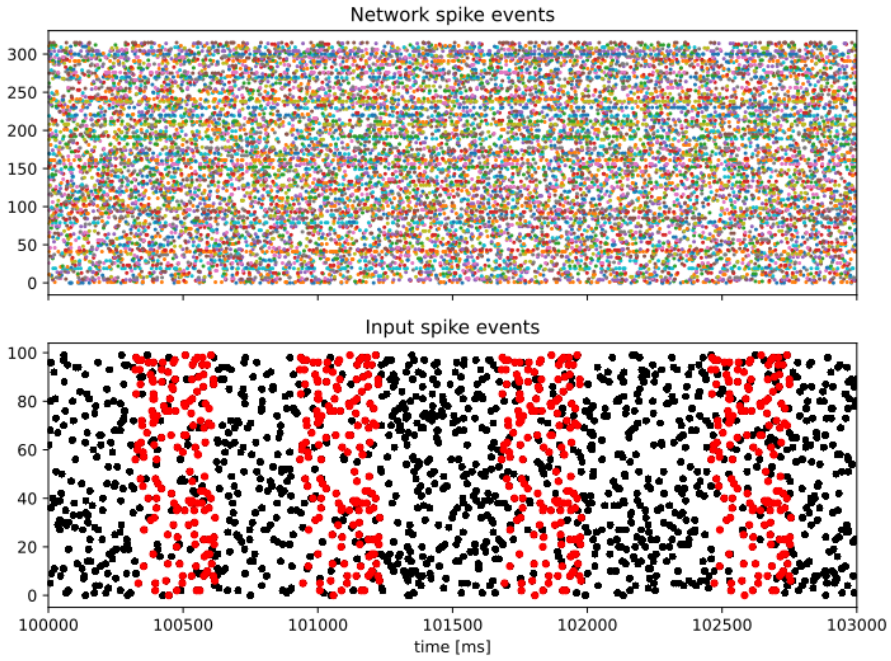

FIGURE 5.8: Full-scale Network: NEST 3 s test after 100 s training without variance tracking

5.2.3 Necessity of STP

For the following test, STP will be turned off for testing its effect on long-term network spike behavior. In (Klampfl and Maass, 2013) Figure 6A a similar test is conducted with a slightly different testing setup in the "without depressing synapses" panel, which means nothing else than "without STP". As can be seen when comparing Figure 5.9 to the Figure in (Klampfl and Maass, 2013), the behavior without STP is comparable in both cases.

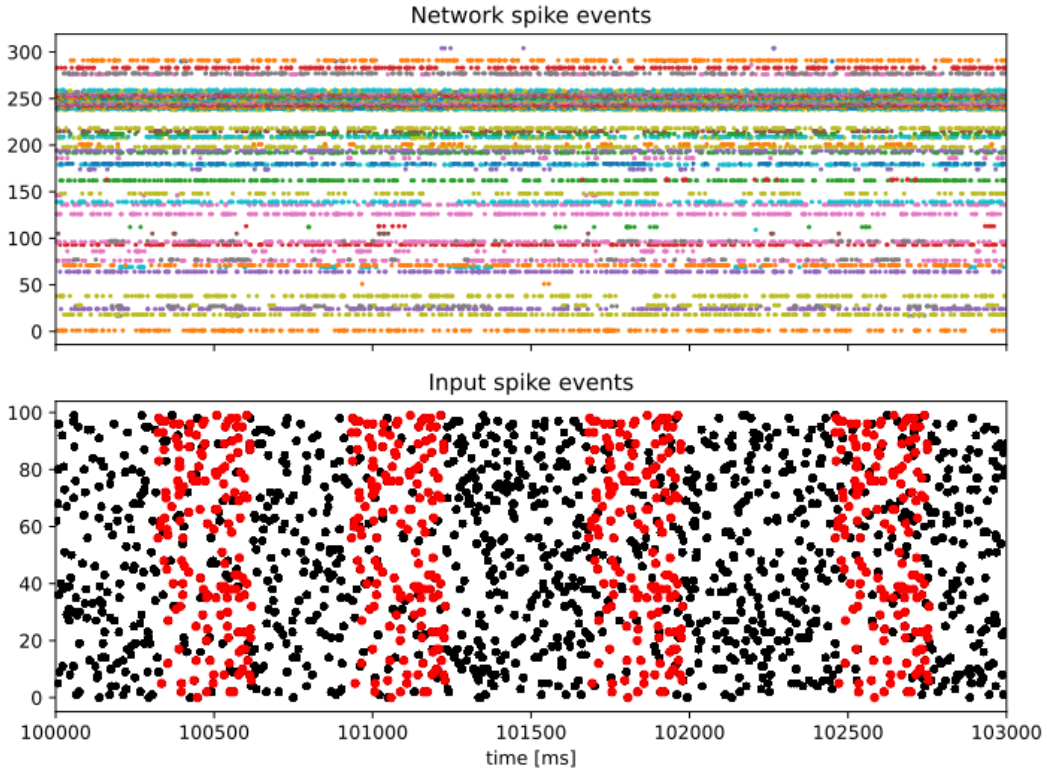

FIGURE 5.9: Full-scale Network: NEST 3 s test after 100 s training without STP

5.2.4 Performance Benchmarking

An important asset of the NEST implementation that has not yet been mentioned is its vastly superior performance. This is an effect of the more efficient simulation kernel of NEST, whereas the version of Klampfl and Maass simulates stepwise and uses matrix multiplication for calculation. In the table below the real-time factors of simulations for both implementations are benchmarked for the full-scale model task from Section 5.2. The hardware used for benchmarking is a Lenovo YOGA 730-15IWL Notebook with 16GB of memory, an Intel Core i7-8565U CPU with 8 cores, and running Ubuntu

22.04 LTS. The original implementation will use all cores while the NEST implementation here only uses 6 cores.

	Real-time factor	Simulation Time
NEST	0.7362	73.63 s
Legacy	41.83	69 min 43 s = 4183 s

5.3 Further Experiments

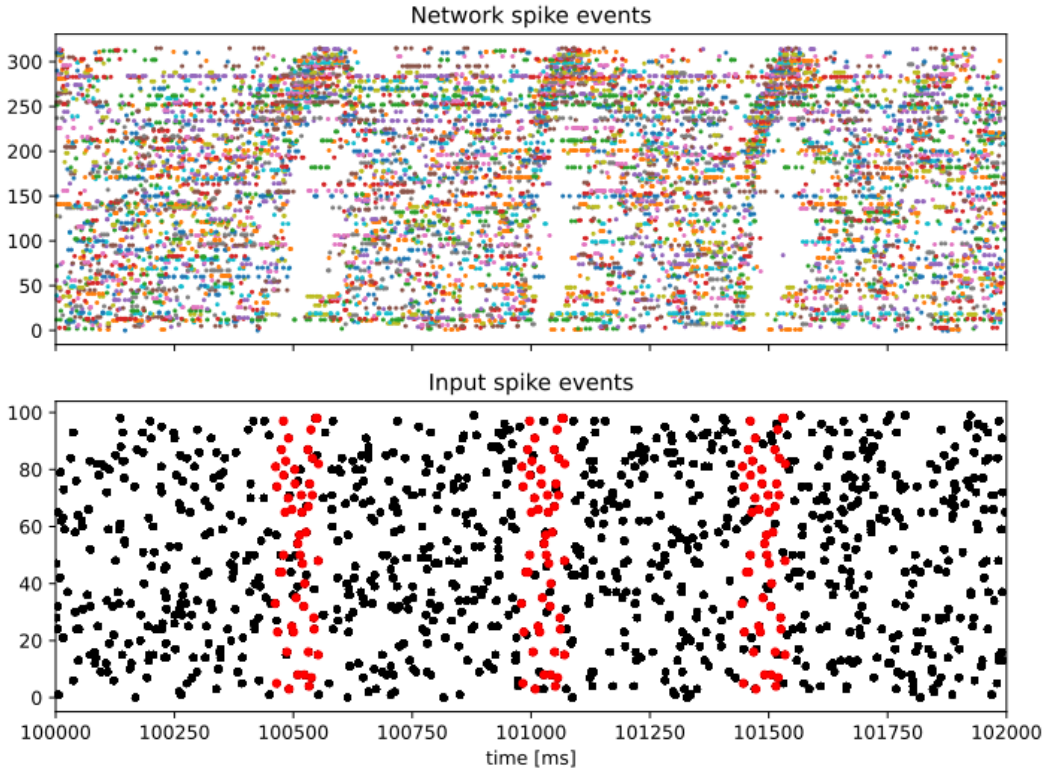

FIGURE 5.10: Pattern duration set to 100 ms. Grid size 10×5 with $k \in [2; 10]$ and 100 input channels. Input rate of 5 Hz with additional 2 Hz during pattern presentation. 2 s test after 100 s training.

To test a few of the boundaries of the replicated model, the pattern duration was varied to examine the assembly formation under these conditions. Figure 5.10 shows the results of a pattern duration of 100 ms and Figure 5.11 shows the result of a pattern duration of 900 ms. The first visible difference between short and long duration is the size of the formed assembly. Where the long pattern activates almost the entire network over time, the shorter pattern activates a much smaller assembly. This also comes with different

levels of assembly distinction. The short pattern causes a clear phase of inactivity for a part of the network, while the longer pattern already activates the whole network, preventing parts of it to become inactive and therefore impeding learning.

In both cases conclusions are deducible: Too long patterns cannot be learned by the network because they overstimulate it, preventing assemblies of reduced activity during pattern presentation to form. Too short patterns on the other hand can only excite small assemblies, to the point where they will not have a noticeable effect at all.

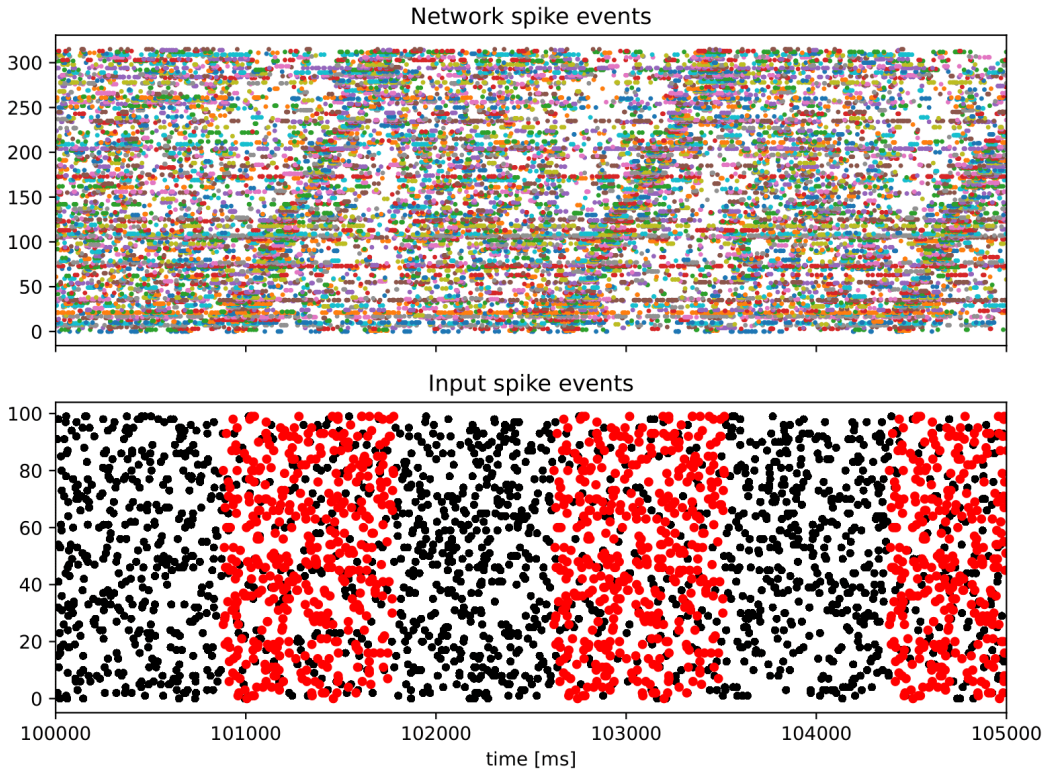

FIGURE 5.11: Pattern duration set to 900 ms. Grid size 10×5 with $k \in [2; 10]$ and 100 input channels. Input rate of 5 Hz with additional 2 Hz during pattern presentation. 5 s test after 100 s training.

Chapter 6

Conclusion

6.1 Conclusion

While this thesis ran into many unforeseeable obstacles, most of the goals set for it were eventually achieved. This allows to draw several conclusions:

The original model has limited implementability when using an object-oriented approach and dedicated neural simulator such as NEST because many equations describing the model behavior require information to be globally available that is not usually disclosed to all entities described by the model. The most prominent example of this being the availability of synaptic weights at the postsynaptic neuron, as described in 4.5.1. When using NEST, it would be more favorable to select a different model for replication that can be integrated better, especially concerning the calculation of membrane potential. Also, it became clear that writing more complicated NEST neuron and synapse models by hand is very prone to bugs and errors because the code gets cluttered very fast and may provoke rare and hard-to-find bugs in NEST because NEST can not possibly provide universal support for every implementation approach chosen by developers.

Despite the difficulty replicating the model in NEST, the process eventually was successful and rewarding. Most remarkable perhaps is the superiority in simulation speed, benchmarked at over a 55-fold speedup compared to the original, making it faster than real-time for many tests. This speedup allows a more flexible use of the model for testing. Also, the replication successfully uncovered many details and hidden assumptions about (Klampfl and Maass, 2013) that can now be considered for future work with related models to avoid similar problems.

6.2 Future Work

As the complexity of this work and its implementation increased continuously throughout its processing, there are several possible directions for future work based on this work. For one, there were a few smaller details about the original paper pointed out, like the incorrect adoption of the STP equation from (Markram, Wang, and Tsodyks, 1998). It should be ensured that the change introduced by (Klampfl and Maass, 2013) does not affect the correctness or characteristics of the STP.
Another point of interest would be the abstract implementation of lateral inhibition. It is clear that the normalization of the firing rate in each WTA is not biologically plausible behavior, although (Klampfl and Maass, 2013) pointed out that it did not affect their results in a meaningful way. Nevertheless, if this model should be used for further research and especially for tests that Klampfl and Maass did not design this model for, a more biologically accurate implementation of lateral inhibition should be deployed. A suitable approach for this could be created using the model from (Häusler and Maass, 2017), where lateral inhibition is described in a more detailed way and using two different types of inhibition, implemented by two different populations of inhibitory neurons, one for rate normalization and the other for blocking STDP.
Due to the limited time of the thesis, there also remain more tests from (Klampfl and Maass, 2013) itself, where more advanced input is presented to the network and even nonlinear computing tasks like XOR are solved.

Appendix A

Glossary of Notations

Mathematical	Meaning
w_{ki}	Weight of synaptic connection from neuron $z_i \rightarrow z_k$
u_k	dynamic facilitation variable of neuron k
R_k	dynamic depression variable of neuron k
U	STP parameter
D	Depression time constant
F	Facilitation time constant
τ_{rise}	Rise time constant of EPSP
τ_{decay}	Decay time constant of EPSP
$r_k(t)$	Firing rate of neuron k at time t
$u_k(t)$	Membrane potential of neuron k at time t
$y_i(t)$	EPSP of neuron i at time t
$R_{\max}$	Maximum firing rate of a WTA circuit
Δ_{k-1}	Interspike interval between spikes k and $k-1$
η_{ki}	Adaptive learning rate of synapse from neuron $z_i \rightarrow z_k$
A_k	Amplitude of the k-th input spike
c	Scaling constant for controlling the strength of STDP
w_{k0}	Bias/excitability parameter of neuron k
K	Number of neurons in a WTA circuit
z_k	k-th neuron in a WTA circuit
λ	Parameter of exponential distance distribution

TABLE A.1: Glossary of mathematical notations

Mathematical	Original	Replication	Value
$r_k(t)$	-	r	Equation 3.2
$R_{\max}$	rmax	R_max	100 Hz
$u_k(t)$	u	V_m	Equation 3.4
$y_i(t)$	Y	yt_epsp_traces	Equation 3.5
τ_{decay}	tau	tau_decay / tau_m	20 ms
τ_{rise}	tau2	tau_rise / tau_syn	2 ms
u_k	udyn	u	Equation 3.7
R_k	rdyn	x	Equation 3.7
U	U	U	$\mathcal{N}(\mu = 0.5, \sigma^2 = (\frac{1}{2}\mu)^2)$s
D	D	tau_d	$\mathcal{N}(\mu = 0.11, \sigma^2 = (\frac{1}{2}\mu)^2)$s
F	F	tau_f	$\mathcal{N}(\mu = 0.005, \sigma^2 = (\frac{1}{2}\mu)^2)$s
w_{ki}	W	w	Equation 3.8/3.9
λ	lam	lam	0.088

TABLE A.2: Translation of terms

PC	**P**yramidal **C**ell
WTA	**W**inner **T**ake **A**ll
STP	**S**hort-**T**erm **P**lasticity
STD	**S**hort-**T**erm **D**epression
STF	**S**hort-**T**erm **F**acilitation
STDP	**S**pike **T**iming **D**ependent **P**lasticity
EPSP	**E**xcitatory **P**ost-**S**ynaptic **P**otential
IPSP	**I**nhibitory **P**ost-**S**ynaptic **P**otential
SNN	**S**piking **N**eural **N**etwork
ISI	**I**nterspike **I**nterval
IAF	**I**ntegrate-**A**nd-**F**ire

TABLE A.3: List of Abbreviations

Bibliography

Elston, Guy N. (Nov. 2003). "Cortex, Cognition and the Cell: New Insights into the Pyramidal Neuron and Prefrontal Function". In: *Cerebral Cortex* 13.11, pp. 1124–1138. ISSN: 1047-3211. DOI: 10.1093/cercor/bhg093. eprint: https://academic.oup.com/cercor/article-pdf/13/11/1124/766840/bhg093.pdf. URL: https://doi.org/10.1093/cercor/bhg093.

Häusler, S. and W. Maass (2017). "Inhibitory networks orchestrate the self-organization of computational function in cortical microcircuit motifs through STDP". In: DOI: 10.1101/228759.

Klampfl, S. and W. Maass (2013). "Emergence of dynamic memory traces in cortical microcircuit models through STDP". In: *Journal of Neuroscience* 33.28, 11515–11529. DOI: 10.1523/jneurosci.5044-12.2013.

Linssen, Charl A.P. et al. (2022). *NESTML 5.0.0*. DOI: 10.5281/zenodo.5784175. URL: https://doi.org/10.5281/zenodo.5784175.

Maass, Wolfgang and Henry Markram (2004). "On the computational power of circuits of spiking neurons". In: *Journal of Computer and System Sciences* 69.4, pp. 593–616. ISSN: 0022-0000. DOI: https://doi.org/10.1016/j.jcss.2004.04.001. URL: https://www.sciencedirect.com/science/article/pii/S0022000004000406.

Markram, H., Y. Wang, and M. Tsodyks (Apr. 1998). "Differential signaling via the same axon of neocortical pyramidal neurons". In: *Proceedings of the National Academy of Sciences of the United States of America* 95.9, 5323–5328. DOI: 10.1073/pnas.95.9.5323. URL: https://doi.org/10.1073/pnas.95.9.5323.

Michau, Simon and Barna Zajzon (2022). *Dynamic memory traces for sequence learning in spiking networks*. https://github.com/simonmichau/dynamic-memory-traces-for-sequence-learning-in-spiking-networks.

Morrison, Abigail, Markus Diesmann, and Wulfram Gerstner (2008). "Phenomenological models of synaptic plasticity based on spike timing". In: *Biological Cybernetics* 98.6, pp. 459–478. DOI: 10.1007/s00422-008-0233-1. URL: https://doi.org/10.1007/s00422-008-0233-1.

Nessler, Bernhard et al. (Apr. 2013). "Bayesian Computation Emerges in Generic Cortical Microcircuits through Spike-Timing-Dependent Plasticity". In: *PLOS Computational Biology* 9.4, pp. 1–30. DOI: 10.1371/journal.pcbi.1003037. URL: https://doi.org/10.1371/journal.pcbi.1003037.

Spreizer, Sebastian et al. (Mar. 2022). *NEST 3.3*. Version 3.2. DOI: 10.5281/zenodo.6368024. URL: https://doi.org/10.5281/zenodo.6368024.

Stevens, C. F. and Y. Wang (Apr. 1995). "Facilitation and depression at single central synapses". In: *Neuron* 14.4, 795–802. DOI: 10.1016/0896-6273(95)90223-6. URL: https://doi.org/10.1016/0896-6273(95)90223-6.