Michael Hsieh, BSc.

Simulation and analysis of autonomous learning processes in spiking neural networks

MASTER'S THESIS

to achieve the university degree of

Diplom-Ingenieur

Master's programme: Software Development and Business Management

submitted to

Graz University of Technology

Supervisor

o.Univ.-Prof. Dr. Wolfgang Maass
Institute for Theoretical Computer Science (IGI)

Graz, December 2015

Affidavit

I declare that I have authored this thesis independently, that I have not used other than the declared sources/resources, and that I have explicitly indicated all material which has been quoted either literally or by content from the sources used. The text document uploaded to TUGRAZonline is identical to the present master's thesis dissertation.

Eidesstattliche Erklärung

Ich erkläre an Eides statt, dass ich die vorliegende Arbeit selbstständig verfasst, andere als die angegebenen Quellen/Hilfsmittel nicht benutzt, und die den benutzten Quellen wörtlich und inhaltlich entnommenen Stellen als solche kenntlich gemacht habe. Das in TUGRAZonline hochgeladene Textdokument ist mit der vorliegenden Masterarbeit identisch.

Date/Datum

Signature/Unterschrift

Acknowledgements

I want to start by thanking Dr. Wolfgang Maass, who made sure I was provided with all the support and infrastructure I required during my work. His effort and timely reaction whenever I needed his help were essential for the completion of this thesis.

Still, writing this thesis took a long time. However, it would have taken even more time had DI David Kappel not supervised me with as much patience and commitment as he did. For this, he will always have my gratitude.

Naturally I also need to mention my family and friends, whose constant worry over my progress gave me the pressure I need to function.

Among them, I would like to point out Susanne Robert, my girlfriend, who showed admirable understanding for my negligence during the last weeks of writing, and Mario Fraiß, whose graphical expertise helped me immensely during the creation of the figures.

Finally, special thanks shall be extended to Rebekka Aigner, my best friend, who made me work harder by threatening to graduate before me.

Michael Hsieh
Graz, Austria, December 2015

Abstract

Computer-based simulations allow researchers to visualize, test and eventually improve their theories. However, developing and effectively making use of those simulations are challenges that should not be underestimated.

This master's thesis uses the simulator *NEST* [13] to showcase the steps necessary to turn models of autonomous learning processes in biological, spiking neural networks into software that runs with adequate performance and produces easily analyzable results. Technical and architectural questions and problems that likely arise during software development are shown, and possible solutions are presented. Furthermore, by understanding the inner workings of simulators, theoretical models can be optimized for use in simulations. How this might work is also examined in this thesis.

Since biological neural networks can end up being quite large, the option to run the simulations in a parallel environment in order to increase performance is also discussed.

Finally, examples are presented to demonstrate how computer simulations can help identify weaknesses and strengths of learning models.

Keywords: Spiking neural networks, reward-based learning, NEST, neural network simulators, autonomous learning

Kurzfassung

Computergestützte Simulationen erlauben ForscherInnen, ihre Theorien zu visualisieren, zu testen und letztendlich zu verbessern. Das Entwickeln und effektive Nutzen dieser Simulationen sind allerdings Herausforderungen, die nicht zu unterschätzen sind.

Diese Masterarbeit zeigt anhand des Simulators *NEST* [13], welche Schritte notwendig sind, um Modelle von autonomen Lernprozessen in biologischen, spikenden neuronalen Netzen als Software umzusetzen, die performant läuft und deren Ergebnisse gut analysierbar sind. Besonderer Wert wird auf die Behandlung von technischen und architekturellen Fragen und Problemen gelegt, die bei einer solchen Softwareentwicklung wahrscheinlich auftreten werden. Für diese werden mögliche Lösungen präsentiert. Ebenfalls wird erläutert, wie durch das Verständnis der Eigenheiten eines Simulators theoretische Modelle für Simulationen optimiert werden können.

Nachdem biologische neuronale Netzwerke eine beträchtliche Größe erreichen können, wurde die Möglichkeit betrachtet, Simulationen in einer parallelisierten Umgebung berechnen zu lassen.

Abschließend demonstriert die Arbeit anhand von Beispielen, wie Computersimulationen Schwächen und Stärken der Lernmodelle aufzeigen können.

Schlagwörter: Spikende neuronale Netze, Reward-basiertes Lernen, NEST, Simulator für neuronale Netze, Autonomes Lernen

Contents

1 Introduction **1**
1.1 Motivation . . . 1
1.2 Introduction to spiking neural networks . . . 2
1.3 STDP and reward-based learning . . . 6

2 NEST, the Neural Simulation Tool **9**
2.1 Simulation of neurons . . . 11
2.1.1 Neuron state variables . . . 11
2.1.2 Neuron update mechanics . . . 12
2.2 Simulation of synapses . . . 13
2.2.1 Synapse state variables . . . 13
2.2.2 Synapse update mechanics . . . 14

3 Implementation of learning processes in NEST **20**
3.1 Translating mathematical models into code . . . 20
3.2 Reward transmitters . . . 23
3.3 Data logging and debugging . . . 25

4 Parallel computation in NEST **29**
4.1 Architecture and performance of parallel simulations . . . 29
4.2 Random numbers . . . 31
4.3 Development of simulations for parallel environments . . . 33

5 Examples **35**
5.1 Step Poisson Generator . . . 35
5.2 Bias Neuron . . . 37
5.3 Digit Classification . . . 40
5.3.1 Part 1: Learning without rewards . . . 41
5.3.2 Part 2: Reward-based learning . . . 43
5.4 Learning a sine wave spike rate pattern . . . 45
5.5 Learning to navigate a double-T-maze decision task . . . 49

6 Conclusion **52**

Bibliography **54**

List of Figures

1.1 Drawing of a biological neuron [27] 2
1.2 Membrane potential of a leaky integrate-and-fire neuron in NEST 3
1.3 Excitatory postsynaptic potential 4
1.4 Feed-forward neural network including a reward transmitter. 7

2.1 Overview of components in NEST 10
2.2 Workflow of neuron updates in NEST 13
2.3 Schematics of synaptic weight updates 19

4.1 Node distribution in NEST 30

5.1 Step Poisson Generator spike trains 37
5.2 Bias neuron usage 40
5.3 Digit classification without learning 43
5.4 Synaptic weights after learning with rewards 44
5.5 Comparison of reward decay rates 45
5.6 Input neurons with sine wave pattern spike rates 47
5.7 Spike rate of a stochastic neuron 47
5.8 Successful simulations of sine wave learning 48
5.9 Double-T-maze decision task [25] 49
5.10 Successful learning of a double-T-maze decision task 51
5.11 Spike trains during successful double-T-maze learning 51

List of Listings

2.1 Example of state variable access in NEST 12
2.2 Common synapse property in NEST 14
2.3 Shortened version of a synapse update routine in NEST, part 1 17
2.4 Shortened version of a synapse update routine in NEST, part 2 18
3.1 Performance gain by skipping unnecessary calculations 22
3.2 Reward transmitter update 23
3.3 Retrieval of rewards from reward transmitters 24
3.4 Virtual functions of a reward transmitter base class 24
3.5 Registering a reward transmitter in the synapse 25
3.6 Registration of a recordable variable 26
3.7 Logging a recordable variable 26
3.8 Manual data logging 27
3.9 Activation of data logging for a subset of synapses 28
4.1 Seeding the NEST random number generators 32
4.2 Randomizing synaptic weights in *NEST* 32
4.3 Enabling MPI if it is available on the system 33
4.4 MPI data consolidation 34
5.1 Step Poisson Generator state variables 36
5.2 Step Poisson Generator spike generation 36
5.3 Step Poisson Generator usage 37
5.4 Bias neuron threshold adaption 39
5.5 Bias neuron usage 40
5.6 PSP approximator update 42

Chapter 1

Introduction

The question of how humans think and make decisions has fascinated people for centuries. Unfortunately, despite rapid scientific progress, the human brain and its astonishing capabilities still remain largely a mystery.

Inspired by the complex tasks the brain can accomplish, artificial neural networks were developed and researched to try and build on the success of their biological counterparts. Even with simplified models, impressive results have been achieved in many applications, like face recognition [37], sampling of seemingly hand-written texts [16] and even games [42], to name just a few. This all is even more fascinating considering the fact that artificial neural networks are composed merely of interconnected neurons, which, despite occasionally having quite complex mechanics, are in essence simple signal processors (see for example the McCulloch-Pitts-Neuron in [34]).

For the research of biological neural networks like the human brain however, spiking neural networks seem to be more appropriate [15, 39]. Contrary to traditional artificial neural networks, which typically propagate continuous numerical values between neurons, neurons in spiking neural networks only send signals, called spikes, when the neuron's firing condition is met (see section 1.2). They are supposed to imitate the biological reality, but this has the consequence that they can include complex biochemical processes and mechanisms that are often not trivial to implement and might require vast amounts of computing power to simulate.

For this reason, much effort nowadays is put into overcoming these challenges, as further research into biological neural networks is one of the keys to understanding the human brain.

1.1 Motivation

Over the last decades, tremendous technological advancements coupled with ever increasing globalization have drastically improved the way research can be conducted. Not only can re-

searchers all over the world communicate and exchange knowledge rapidly, but interdisciplinary work has become easier than ever.

Thanks to this, complex fields of study like brain research can now hope to make progress much faster than previously possible. Research into neural networks requires contributions not only from the fields of biology and mathematics, but also from computer sciences. With increasing processing power and the training of professional software developers, brain researchers finally have the means to properly test their theories and simulate their models.

It is therefore the goal of this thesis to analyze and discuss how theoretical models of biological neural network dynamics can be comfortably tested and improved using software tools, and what kind of challenges one might face during the implementation of these models.

To this end, examples were created based on the models described in [24] and [25].

1.2 Introduction to spiking neural networks

Biological neurons, as mimicked in spiking neural networks, are not simple signal processors like their counterparts in artificial neural networks.

In reality, they typically consist of a cell body (also called soma), dendrites and an axon, as is illustrated in figure 1.1. Basically, dendrites are input channels with which electrical pulses called spikes are received. Those spikes originate from the axon of other neurons and they are generated when a neuron's membrane potential reaches its spike threshold due to stimuli from prior incoming spikes. After a spike is sent, many neurons enter a refractory period, which prevents them from firing again for a short amount of time, illustrated in figure 1.2. This whole procedure is explained in detail in [12] and [31].

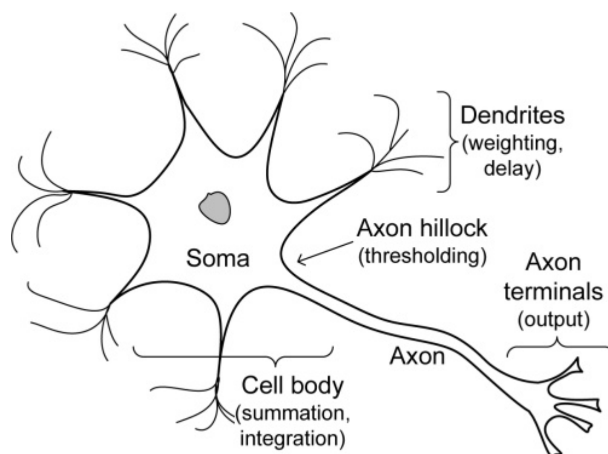

Figure 1.1: Drawing of a biological neuron [27]. It shows the main components of the neuron and summarizes their function. Dendrites are generally understood as the input channels of a neuron. The cell body accumulates this input, and when applicable sends a signal of its own through the axon.

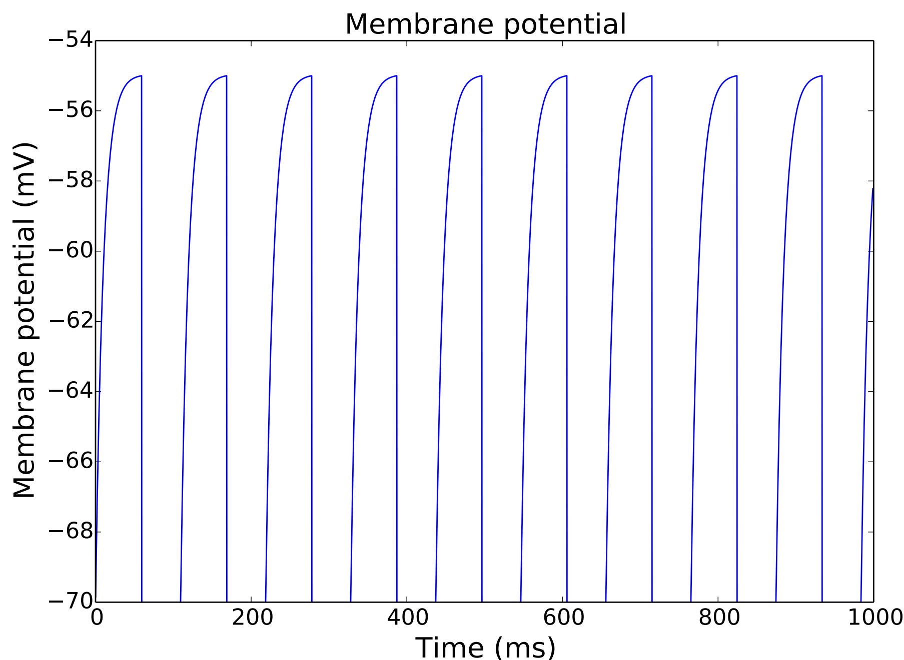

Figure 1.2: Membrane potential of a leaky integrate-and-fire neuron [12] in NEST. The neuron is supplied with a constant current forcing the membrane potential to increase whenever possible. A (very large) refractory period of 50.0 ms is added for demonstration purposes. As soon as the membrane potential reaches the spike threshold at −55 mV, a spike is generated and the potential resets to the neuron's resting potential at −70 mV.

Axon and dendrites are usually (though exceptions exist) connected via specialized organelles, which are called synapses [48]. From the point of view of a synapse, the sending neuron - i.e. the origin of the axon - is called the presynaptic neuron, and the receiving neuron - i.e. the origin of the dendrite - is called the postsynaptic neuron.

The transfer itself - as is obvious in biologically relevant settings - does not happen instantaneously. The time the spike takes to travel through the connection is called synaptic delay.

It is important to note that spikes are binary occurrences - the neuron either spikes or it does not. The strength of a spike itself does not matter for the synapse, its impact on the target neuron is solely manipulated by the synapse itself [12]. The spikes that the synapses relay to the target neuron do vary in strength though, based on biochemical factors. While these factors are complex, they can be simply thought of as synaptic weights. Larger synaptic weights cause stronger effects in the target neurons. Changes in the weights are called synaptic plasticity and form the basic of autonomous learning processes [2, 20], as discussed further in section 1.3.

When the spike finally reaches a target neuron, it causes a postsynaptic potential, which changes the neuron's membrane potential. Whether the membrane potential is increased or decreased depends on the aforementioned synaptic weight. If the membrane potential is increased by the spike, the postsynaptic potential is called excitatory (EPSP). Otherwise, it is called inhibitory (IPSP) [12]. In models and simulations the latter can often be simplified by interpreting

inhibitory synapses as synapses with negative weights.

As explained in [12] and [18], the PSP can be modelled as a mathematical function, for example with a double exponential shape. Intuitively, it can be said that the influence of a neuron on spikes of its target neurons is likely large shortly after it has caused an EPSP in them by sending a spike itself. A small time delay should be considered, because PSPs take a little while to have an effect on the target's membrane potential. After that, their effect declines over time. How fast those changes occur are defined by the PSP's facilitation and depression rates. An illustration of the effect of a PSP can be seen in figure 1.3a.

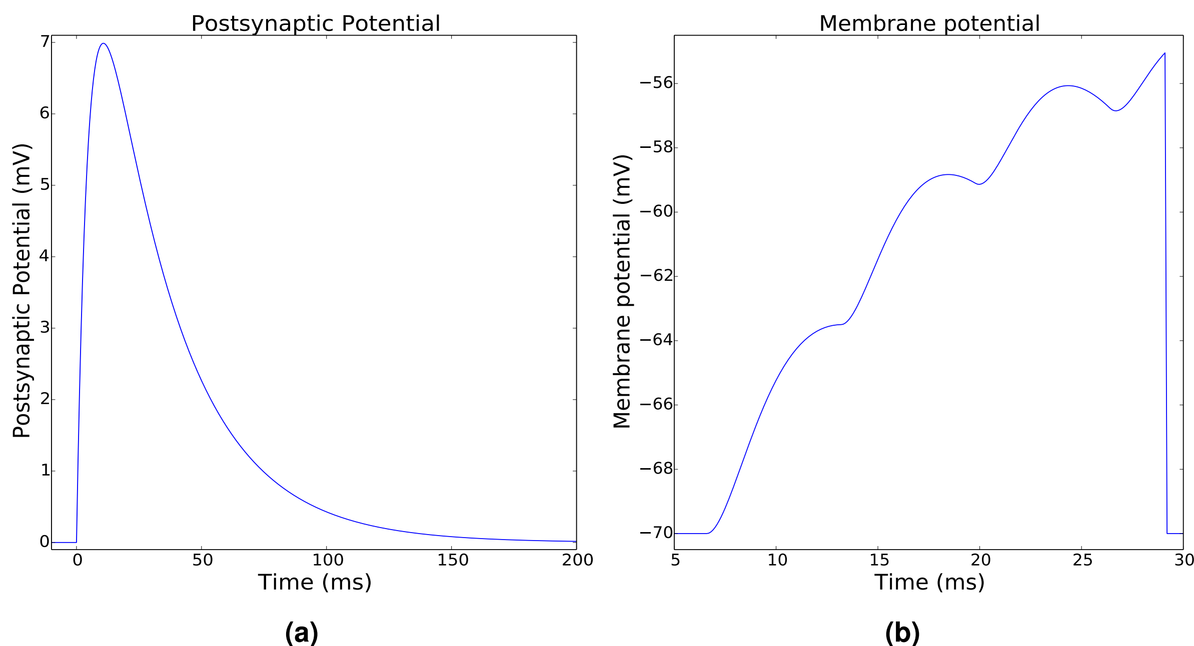

Figure 1.3: **(a)** Example of an excitatory postsynaptic potential. Assuming the spike arrived at 0 ms, the EPSP raises neuron's membrane potential by up to approximately 7 mV after a few milliseconds, before it quickly decays. **(b)** Multiple EPSPs accumulate to eventually cause a spike when the membrane potential reaches −55 mV. Immediately after the spike, the membrane potential decays to its resting potential at −70 mV.

Through accumulation - PSPs are graded - of many such potentials originating from multiple spikes fired by one or many presynaptic neurons, the membrane potential of the target neuron eventually exceeds the spike threshold, causing the neuron to spike, as illustrated in figure 1.3b. This is in contrast to figure 1.2, where the neuron's membrane potential is forced to continuously increase by a constant current. Further discussion on postsynaptic potentials can be found in the aforementioned works [12] and [18].

It is important to note that not all biological neurons have a fixed threshold potential at which they spike. Research has shown that some neurons show stochastic spiking behaviour [22, 12]. A neuron exhibiting stochastic spiking behaviour increases the probability of a spike based on its membrane potential, with a probability of 1 at its stochastic equivalent of a spike threshold.

A sample model for a neuron with stochastic spiking behaviour is presented in [25] and summarized by

$$u_k(t) = b_k + \sum_{j \neq k} \hat{z}_j(t)\, w_{kj}, \tag{1.1}$$

$$\sigma(u_k) = \frac{1}{1 + e^{-u_k}}, \quad \text{and} \tag{1.2}$$

$$p(z_k \text{ fires at } t) = f_k(t) = \sigma(u_k)\, \Theta. \tag{1.3}$$

The membrane potential u_k for a neuron z_k, as given in equation 1.1, is defined by b_k, which is the bias of the neuron, and $\hat{z}_j$, which is the PSP approximator of a neuron z_j that is connected to z_k with weight w_{kj}. The neuron x_j can be either a designated input neuron or another hidden neuron in a recurrent network.

The PSP approximator is, intuitively speaking, the synaptic guess of the presynaptic neuron's current influence on the postsynaptic neuron. Therefore, it ideally is a function with a similar shape as the neuron's actual PSP, but not necessarily with the same values. The exact values of $\hat{z}_j$ do not matter as long as they are approximately in a ratio the same as the PSP values. This is because its purpose is merely to scale the learning process based on the neuron's current influence. Note that the postsynaptic neuron's actual PSP likely looks vastly different than $\hat{z}_j$, because the former is probably influenced by multiple synapses (and their action potentials) at once, while the latter only considers the influence of the presynaptic neuron. Furthermore, in the case of inhibitory synapses ($w < 0$) the PSP approximator stays positive and turns to an approximator for the weight's absolute value, so the two parameters multiplied result in negative weight update overall. An example implementation of such an approximator is shown in listing 5.6.

Equation 1.2 defines $\sigma(\cdot)$ an exponential function that rises together with increasing u_k.

In equation 1.3, $\sigma(\cdot)$ is then multiplied with Θ, which is 1 if the neuron is not in its refractory period, and 0 if it is refractory.

This results in a higher overall spike probability if the membrane potential is high, and vice versa, as long as the neuron is not refractory. If it is, the spike probability is 0.

Lastly, structural plasticity has been observed in the human brain, which means that not only do the weights of synapses change over time, but synapses can appear or even disappear completely [6, 19, 50, 52]. This makes sense, since spontaneous and even random formations of synapses can help to discover new important connections [6], and truncating unnecessary synapses saves energy (they need to excert force in order to maintain their form and function, as discussed in [26]). In most cases, this process can easily be modelled, again by changing the synaptic weights, with a weight of 0 indicating a non-existent synapse.

To summarize, in spiking neural networks neurons can receive either constant signals or

spike events and will in return distribute spikes events themselves, in contrast to traditional artificial neural networks in which neurons send constant signals with varying strength. These spikes are transferred through synapses, which differ from each other by their weights. By manipulating these weights, the influence of one neuron on every other one can be controlled, effectively training the network to perform desired tasks.

1.3 STDP and reward-based learning

Synaptic plasticity - the modification of a synapse's strength and even existence - plays a critical role in the behaviour of the network. While other factors exist as well, synaptic weights are very obvious starting points to train networks. Thus, all learning processes discussed in this thesis utilize synaptic plasticity.

Aside from *which part* of the network is to be trained (i.e. the synapses), it is also important to decide on *when* changes ought to occur. Brain research strongly suggests that Spike-Timing Dependant Plasticity (STDP) is the answer to that [1, 5, 32, 33]. In learning strategies using STDP neural networks are trained by looking for causal relations between spikes. This approach resembles early theories on learning associations formalized by Hebbian learning [36]. The basic idea is to strengthen synapses in which presynaptic spikes occur shortly before postsynaptic spikes (thereby implying a causal relation between the two neurons in the sense of *post hoc, ergo propter hoc*[1]) and letting other synapses slowly decay and even vanish, which is consistent with biological observations [6, 26].

This mechanism is capable of building a kind of memory inside the neural network, enabling it to recall firing sequences of strongly connected neurons. However, there is no guarantee of *what* is learned by the network.

To change that, the learning process is extended with a reward parameter, resulting in reward-modulated STDP [11, 29, 30]. Like in "normal" STDP, synapses between causally related neurons gain strength, but in reward-modulated STDP this only happens - or at least it happens more prominently - if the outcome of the firing sequence results in a reward. This is also biologically sound, as in the human brain such rewards are generated by the release of neurotransmitters like dopamine [3].

Rewards often do not arrive instantaneously. In many cases the task is repeated for multiple episodes, and rewards are only distributed at the end of each episode. The synapse therefore needs to remember its recent changes, so it can work with them depending on the reward. This is usually accomplished by an eligibility trace that accumulates all recent changes the synapse performs [21].

[1] Latin: "After, therefore because of."

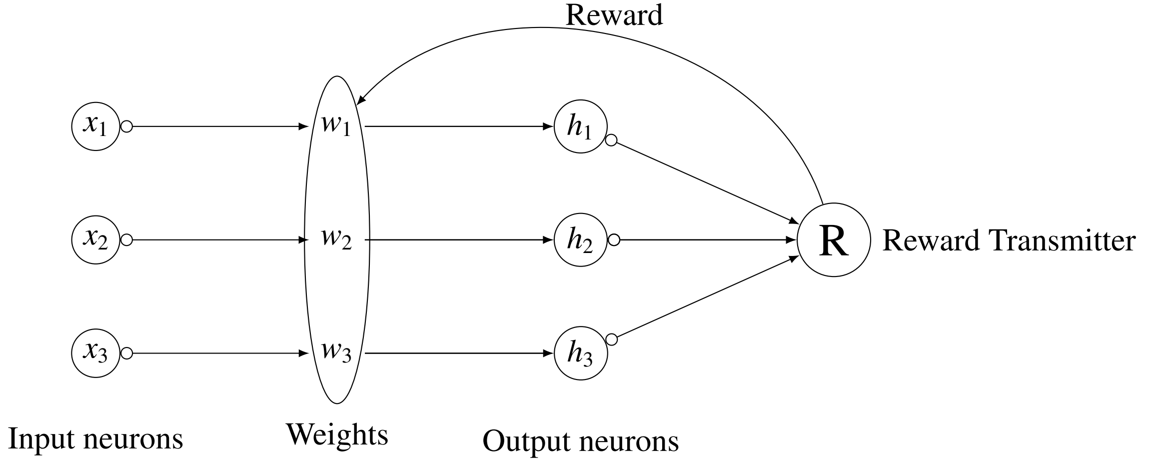

Figure 1.4: Feed-forward neural network including a reward transmitter. Three input neurons are connected to one output neuron each, which are in turn connected to the reward transmitter. The reward transmitter will send rewards (i.e. neurotransmitters) to the synapses depending on the performance of the output neuron, manipulating their synaptic weight changes.

An example of an eligibility trace is produced in [25], where the eligibility trace is concluded as

$$e_{ki}(t) = e_{ki}(t - \Delta t)e^{-\Delta t/t_{\mathrm{ep}}} + t_{\mathrm{ep}}\, W_{ki}\, \hat{z}_i(t)\, (z_k(t) - f_k(t)), \tag{1.4}$$

with $e_{ki}(t)$ being the eligibility trace between the input neuron i and the output neuron k at time step t. Δt is the time since the last update and t_{ep} denotes the episode length between two rewards (and thus the primary reason why the eligibility trace is necessary at all). W_{ki} is the synaptic weight and $\hat{z}_i$ denotes the PSP approximator (see section 1.2). z_k is 1.0 if the postsynaptic neuron has spiked at time step t (else 0), and $f_k(t)$ represents the spike probability of the postsynaptic neuron at time step t, as described in equation 1.3.

This means the eligibility trace defines the synapse trajectory based on a mechanism similar to STDP. The trace will generally increase if a presynaptic EPSP is followed by a postsynaptic spike, and decrease if the postsynaptic spike occured before the presynaptic EPSP [25].

This eligibility trace can be implemented in the synaptic weight update (again as presented in [25]) to arrive at the novel reward-modulated STDP learning rule shown in equation 1.5, based on Bayesian inference [24] and inspired by research on Bayesian Reinforcement Learning (see [4, 49, 51]).

While there are many variations of these learning rules, the basic idea is always similar to

$$\Delta\theta_{ki} = \eta\, (\Omega_i + \psi(r_k, e_{ki})) + \phi. \tag{1.5}$$

Thus, the synaptic parameter change $\Delta\theta_{ki}$ for presynaptic neuron i and postsynaptic neuron k is given by

- η, the learning rate,
- Ω_i, an attractor prior for the synapse,
- r_k, the reward held by neuron k,
- e_{ki}, the eligibility trace described in equation 1.4,
- $\psi(r_k, e_{ki})$, a function calculating an appropriate learning trajectory based on e_{ki} and r_k (a trivial example would be $\psi(r_k, e_{ki}) := r_k\ e_{ki}$), and
- ϕ, a random walk process. This is in accordance to the stochastic processes present in structural plasticity [6, 17, 26].

In other words, the eligibility trace e_{ki} (i.e. the STDP-like trajectory of the synapse) and the current reward r_k translate to a learning trajectory ψ that seems to be advantageous for the network performance. This trajectory is combined with the prior Ω_i to calculate the primary update gradient, which in turn is scaled with the learning rate η and then complemented with a random walk process ϕ to add some noise. The final result will, in theory, force the synaptic weight into a value appropriate for the learning goal.

It is to mention that the reward r_k does not have to be exactly the reward sent by the reward transmitter. The source paper [25] of the algorithm discusses the possibility of introducing reward averages and average expected rewards in order to create a kind of "reward memory". The reward can also be binary (as in section 5.3.2) or an arbitrary value (as in section 5.4), its exact type and form depends on the task at hand.

The experiments in sections 5.4 and 5.5 show how reward-modulated STDP can be used to train neurons for various tasks.

Although all learning algorithms discussed in this thesis follow the ideas outlined above, small changes in the update mechanism can cause strong changes in the network's performance that only become apparent using network simulations.

The following chapters will therefore analyze how models can be efficiently simulated using neural network frameworks and what steps can be taken in order to optimize both memory consumption and runtime performance, all for the sake of providing researchers with a convenient and feasible way to test their theories.

Chapter 2

NEST, the Neural Simulation Tool

Accompanying ongoing research into neural networks, various network simulators have been developed over the years [7]. Even though general software tools like *Matlab* [43, 44] are sufficient for the simulation and analysis of artificial neural networks [14], this might not hold true for biological networks. The need to model complex neuron and synapse mechanics while still retaining acceptable run times justifies the usage of custom-built software dedicated for these purposes alone.

This thesis focuses on *NEST* [13], as it is both easy to use and highly modular, making it possible to showcase a multitude of model implementations using a single framework. While its kernel is written in *C++*, *NEST* includes *PyNEST* [9, 45], an interpreter that allows users to write their simulations using Python, which is more convenient for many researchers.

As *NEST* is written in *C++*, its internal architecture follows an object-oriented approach, mirroring real-life entities to data object [45].

This means the simulated network consists of a variable number of *Neurons* objects which are connected to each other using *Synapse* objects, both inspired by their real-life biological counterparts.

Since researchers must be able to manipulate and analyze the networks, additional tools are often available in the simulators. In *NEST*, these include *Recorders*, *Generators* and *Transmitters*.

Recorders track the state of the network and can be used to generate statistics and graphs at the end of the simulation. *Generators* feed the network with external input that is supposed to be separate from the network state, for example visual or auditory stimuli originating from the environment. Since *Synapses* are only designed to connect *Nodes* to each other, *Transmitters* are present with the purpose of delivering signals from *Nodes* to *Synapses*.

NEST's architecture classifies those tools as *Nodes*, of which *Neurons* are also a subset of.

Customizations can be integrated into *NEST* by writing an extension module. Tutorials

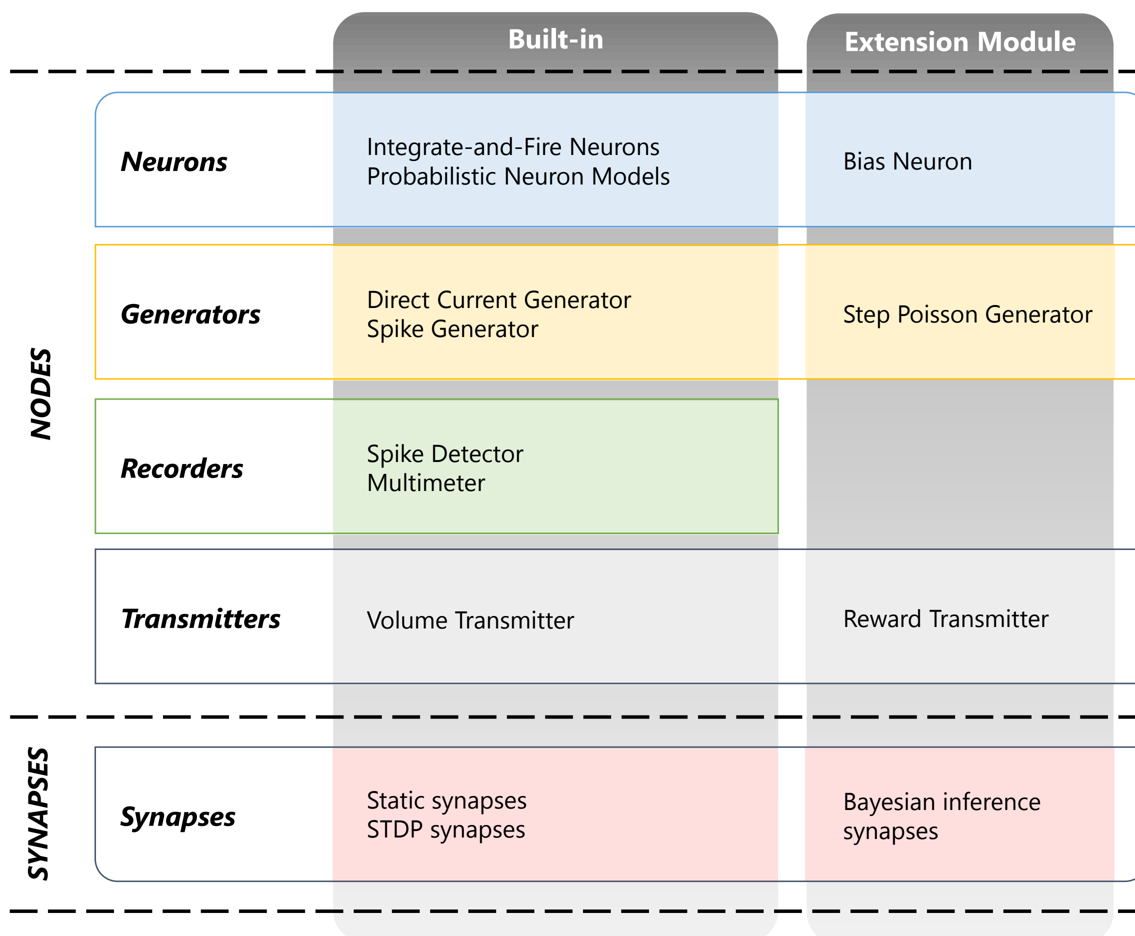

Figure 2.1: Overview of components in NEST. Illustrated are some examples of built-in features in NEST. Also shown are nodes and synapses that are discussed in this thesis and integrated into NEST via an extension module.

and examples on how to write such extension modules are provided by the *NEST* documentation [45], but basically they are just a group of nodes and synapses that can be statically or dynamically linked into *NEST*. The latter approach is more advisable, because the extension module can be compiled independently without having to recompile *NEST* as well, which, given its size, does take some time.

Figure 2.1 gives an overview of *NEST* features after some models were integrated using an extension module.

The following sections aim to provide a summary about the software implementations of neurons and synapses, based on [45]. Even though the sections and the examples within them are based on *NEST*, the information provided by them is generally applicable to any simulator for biological neural networks.

2.1 Simulation of neurons

Like other simulation frameworks, *NEST* ships with a multitude of neurons, but it is often necessary to implement new ones for various reasons.

The most obvious motivation for a custom neuron is of course the simulator's lack of a neuron designed for the desired behaviour. But custom neurons can offer far more flexibility than this, for example they allow the developer to retrieve debugging data exactly where and when it is needed.

Generally, neurons are defined by three properties that must be decided upon for every neuron to be written.

Firstly, the type of input the neuron can receive. Possible options are spike events, external current, or even data logging requests. Since they all are very different types of signals, they must be handled separately. In the case of *NEST*, the signals are sent to different event handlers, though this might vary from simulator to simulator.

Secondly, the neuron's spike variables define which properties the neuron has and how and when they can be accessed or modified.

Finally, the neuron's update dynamics specify how the neuron reacts to input and what output it produces.

2.1.1 Neuron state variables

The neuron state variables can be categorized into three categories.

The first category contains those state variables that describe the nature and properties of the neuron, and they do not change over the course of the simulation. They are, for example, time constants, facilitation/depression rates, or refraction periods.

The second category consists of variables that naturally change over the course of the simulation, like the membrane potential. Some variables that might be considered part of the first category might also end up here if the neuron mechanics demand it. For example, the spike threshold might be a static property of some neuron types, but could change in others.

Lastly, the third category holds variables used for implementation details, like spike histories or data logging buffers.

Since neurons - at least in object-oriented architectures - are stored as their own class, the configuration of the state variables as member variables is trivial in most simulation frameworks.

In *NEST*, the various categories are stored - cleanly separated from each other - in *structs*, while special accessor methods can be configured to either allow or forbid changes to the vari-

ables by the environment. These functions also serve as validators for the values.

```
void CustomNeuron::Parameters_::get(DictionaryDatum &d) const {
  // Return the resting potential
  def<double>(d, names::V_reset, V_reset_);
}

double CustomNeuron::Parameters_::set (const DictionaryDatum& d) {
  // Store the state variable
  updateValue<double>(d, names::V_reset, V_reset_);

  // Validation against the threshold potential
  if (V_reset_ >= Theta_)
  {
    throw BadProperty ("...");
  }
}
```

Listing 2.1: Example of state variable access in NEST. The `get()` function retrieves values from the neuron, as demonstrated with the resting potential. The `set()` function is responsible for storing data and validating it. In this case, the resting potential is rejected if it is higher than the threshold potential.

2.1.2 Neuron update mechanics

While changes to the neuron state happen continuously in real life, this is of course not the case in computer simulations. As described in section 3.1, simulation updates happen due to a mixture of regular grid updates and irregular events.

In *NEST*, this process is usually handled in a simplified way, because neurons are per definition always synchronized with the global state updates. They therefore are not required to respond immediately to events. Input events like spikes are merely buffered into one of the state variables, and only during the global state updates do state changes happen. Thus, the development usually consists exclusively of changes to the `calibrate()` method used for the state initialization, and the `update()` method, which is called regularly by the global state update mechanism (as described in algorithm 2).

Inside an `update()` routine, aside from cleaning up time-restricted buffers like ring buffers, the first thing that happens is usually the accumulation of all action potentials that occurred since the last update. If they increase the membrane potential enough to warrant spikes, and if the neuron is not in its refractory period, the appropriate amount of spikes is sent to its target neurons. Because there is always a synaptic delay, those spikes will only arrive at their destinations a short time later, which means the target neurons can wait until the next update cycle to handle them.

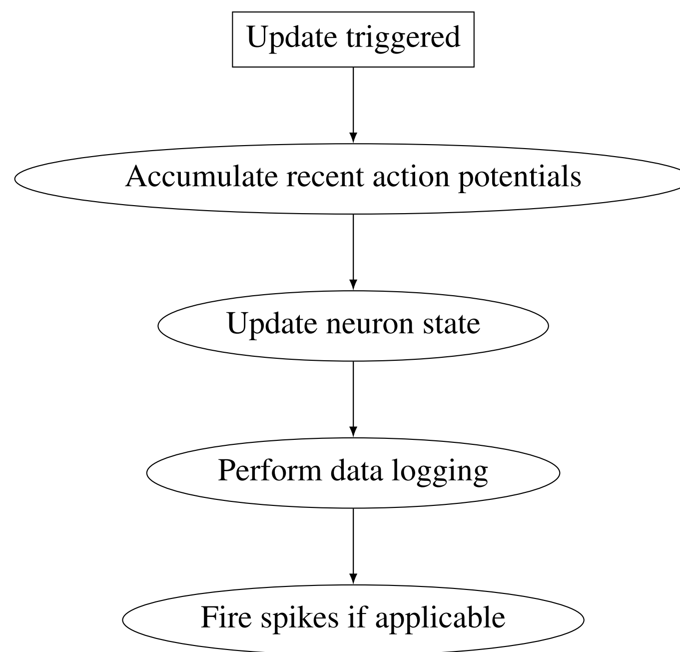

Figure 2.2: Workflow of neuron updates in NEST

Because STDP requires the synapses to know when postsynaptic spikes have occurred, and because synapses are usually far more numerous than neurons, it is only natural that neurons are tasked with memorizing the spikes they have produced. In *NEST*, neurons with this capability are called *Archiving Neurons*, and their usage is required in all STDP learning processes.

2.2 Simulation of synapses

As stated in section 1.3, synapses are a critical part of any biological neural network simulation.

Neural networks usually consist of a vast number of synapses, especially if the neurons are connected recurrently to each other. Additionally the synapse's update mechanism performs most of the STDP calculation and can therefore be quite complex.

This means that not only could there be many thousands of synapses to be stored in the computer memory, their update procedure and event handling could also be potentially slow to calculate.

Therefore, careful development and optimization of synapse code is integral to writing successful simulations.

2.2.1 Synapse state variables

Since synapses are plentiful, it is apparent that parameters applying globally to all synapses of a model should be stored only once. This ensures synchronization between synapse groups and can also increase performance by reducing the memory load.

NEST's architecture prepares for these global parameters, also called homogeneous parameters, by adding *Common Properties* to the synapse's architeture. The *Common Property* object is provided to all relevant functions as an argument and holds all global parameters as member variables. Most importantly, it is allocated only once per virtual process, so the amount of members every individual synapse has to store is reduced.

```
class CustomCommonProperties {
  friend class CustomSynapse;
  private:
    // Homogeneous parameters
    double_t learning_rate_;
};

class CustomSynapse {
  void send(..., const CustomCommonProperties &cp)
  {
    dw = cp.learning_rate_ * learning_gradient;
  }
}
```

Listing 2.2: Common synapse property in NEST. The common property stores a global learning rate for all synapses of this model. The object is provided as an argument to the *send()* method, where presynaptic spikes are handled.

Aside from homogeneous parameters though, synapse state variables are stored and accessed in the same way as their counterparts in neurons (discussed in section 2.1.1).

2.2.2 Synapse update mechanics

While neurons usually work with relatively straightforward update mechanisms, usually by performing state changes only in regular intervals, synapses are a bit more complex.

Synapses need to handle their updates based on the different external events on which STDP relies upon.

The first and most obvious event is the arrival of a presynaptic spike in the synapse. Since neurons expect their incoming spikes to have arrived by the global state update immediately following the synaptic delay, synapses must handle presynaptic spikes instantly. Furthermore, because the strength of the spike must be decided at the moment the spike is sent to its target neuron, all changes to the synaptic weight must also happen as soon as a presynaptic spike is received (because it is then immediately sent on its way to the postsynaptic neuron).

NEST calls the `send()` method for every time a presynaptic spike occurs, but there is no event trigger for postsynaptic spikes. This means that many neurons in *NEST* have to be *Archiving Nodes*, because only neurons of this type are capable of storing spikes they have sent, giv-

ing their input synapses a way of determining the times of their recent postsynaptic spikes. The timing of postsynaptic spikes is naturally important for STDP learning methods, so memorizing them is mandatory even though it is costly in terms of memory and runtime.

In algorithm 1, incremental updates are performed for every "time segment" from the last presynaptic spike until the current presynaptic spike. A "time segment" here refers to a timespan limited by two spike events that have no third spike event happening between them. For every time segment between a presynaptic spike and a postsynaptic spike, or between two postsynaptic spikes, two updates are performed. One for every time step (see section 3.1 for details on time steps) but the one that actually included a postsynaptic spike, and then one more update for the time step in which the postsynaptic spike has occurred. This behaviour might not be necessary for some learning models, but provides a great deal of control to the developer. After iterating through all postsynaptic spikes, a final update must be performed for the time segment between the last postsynaptic spike and the current presynaptic spike.

Algorithm 1 Example of a STDP update without rewards

1: $\Delta \leftarrow$ Resolution unit
2: $t_{iterator} \leftarrow$ Time of last presynaptic spike
3: $t_{current} \leftarrow$ Time of current presynaptic spike
4: $T_{postsyn} \leftarrow$ Postsynaptic spikes times between $t_{iterator}$ and $t_{current}$
5: **for** every element $t_{postsyn}$ in $T_{postsyn}$ **do**
6:     **if** $t_{iterator} < t_{postsyn} - \Delta$ **then**
7:         Call $U(t_{iterator}, t_{postsyn} - \Delta, false)$
8:     **end if**
9:     Call $U(t_{postsyn} - \Delta, t_{postsyn}, true)$
10:     $t_{iterator} \leftarrow t_{postsyn}$
11: **end for**
12: **if** $t_{iterator} < t_{current}$ **then**
13:     Call $U(t_{iterator}, t_{current}, false)$
14: **end if**

Updates are performed by a function $U(start, stop, postsyn)$, which updates the synaptic weight for the time segment between $start$ and $stop$. The argument $postsyn$ declares whether postsynaptic spikes have occurred during that segment.

Algorithm 1 shows a sample procedure outlining the aforementioned update routine. After the procedure has finished, every single time step between two presynaptic spikes has been handled in one of the executions of $U(start, stop, postsyn)$.

Additionally, rewards can also happen in irregular intervals and need to be evaluated during the synaptic weight updates. This all makes the synapse update routine quite nested and complex.

One way to structure all the events is to base all weight updates on presynaptic spikes, because the weights are irrelevant for postsynaptic spike events and reward events. Only when a presynaptic spike arrives, must the synaptic weight be updated.

Also, reward events do not require an immediate response from the synapse, so they can be buffered until needed by the `send()` method. By then iterating step-by-step through the events while cleaning up the buffers after every access, the weight update between the last and current presynaptic spikes can be calculated.

Listings 2.3 and 2.4 outline how such a routine could be realized in *NEST*. Note that this is a very granular algorithm, enabling the developer to perform calculations on every single time step of the *NEST* resolution by trading off performance. Such a level of control is not necessary for many learning processes. Figure 2.3 illustrates this routine in graphical timeline and shows which functions are called at which events.

```
void send(...) {

  // Start the iterator with the last presynaptic spike
  double_t t_last_postsyn = t_last_presyn;

  // Fetch postsynaptic spikes
  target->get_history();

  while (start != finish) {
    double_t t_postsyn = start->t_;

    // "Negative" updates until t_postsyn spike - resolution_
    if (t_last_postsyn < t_postsyn - resolution_) {
      updateSynapseState(t_last_postsyn, t_postsyn - resolution_,
         false);
    }

    // "Positive" update to t_postsyn
    updateSynapseState(t_postsyn - resolution_, t_postsyn, true);

    Update t_last_postsyn for the next loop iteration
    t_last_postsyn = t_postsyn;
    ++start;
  }

  // Update until current presynaptic spike
  if (t_last_postsyn < t_current_presyn) {
    updateSynapseState(t_last_postsyn, t_current_presyn, false);
  }
}
```

Listing 2.3: Shortened version of a synapse update routine in NEST, part 1. This is basically the implementation of algorithm 1. `updateSynapseState()` receives the start time, the stop time, and a boolean flag indicating the presence of postsynaptic spikes as arguments.

```
void updateSynapseState(double_t t_from, double_t t_to, bool
   postsyn) {

  // Start the iterator with the beginning of the segment
  double_t t_last_reward = t_from;

  // Iterate through rewards
  while (!reward_history_.empty()) {
    RewardHistoryEntry reward = reward_history_.front();

    // reward.first holds the reward time
    // reward.second holds the reward value
    if (reward.first <= t_to) {
      for (double_t time = t_last_reward + resolution_; time <
         reward.first; time += resolution_) {
        // Update synapse variables for time steps without reward
        updateSynapseVariables(time, postsyn, 0.0);
      }

      // Update synapse variables for the time step of the reward
      updateSynapseVariables(reward.first, postsyn, reward.second);

      // Update t_last_reward for the next iteration loop
      t_last_reward = reward.first;
    }
    else {
      // Remaining rewards are not relevant for this segment
      break;
    }
  }

  // Update until end of segment
  for (double_t time = t_last_reward + resolution_; time <= t_to;
     time += resolution_) {
    updateSynapseVariables(time, postsyn, 0.0);
  }
}
```

Listing 2.4: Shortened version of a synapse update routine in NEST, part 2. Following listing 2.3, every update segment is again sliced into multiple segments based on reward times. `updateSynapseVariables` performs the actual weight update for a single time step based on the presence of a postsynaptic spike and the reward.

Figure 2.3: Schematics of synaptic weight updates, as implemented in listings 2.3 and 2.4. The illustration shows various events between two presynaptic spikes and how they affect the update function arguments. A *postsyn* of 1.0 means that a postsynaptic spike has occurred in this time step. *postsyn* is 0.0 otherwise. *r* denotes the reward at the given time step. The example interprets inactivity of the reward transmitter as a reward of 0.0.

Chapter 3

Implementation of learning processes in NEST

3.1 Translating mathematical models into code

Models for autonomous learning processes tend to be described using mathematical formulas, usually differential equations, that assume continuous updates coupled with instantaneous calculations and need not (and should not) worry about optimization and runtime.

The first step to implementing such a model is therefore always to map the formulas into functions that can be written as code within the simulator framework. To do this, a thorough understanding of the framework's time management strategy is imperative.

Since simulation frameworks operate on computers, and often even clusters of computers, the computational time varies from the simulated biological time.

The calculation time for a biological second of network changes might be shorter or longer than an actual second, depending on the circumstances and hardware. Spike transfers that should be completed within the synaptic delay might take longer, simply because the source neuron and target neuron are stored on different host servers.

It is therefore necessary to implement some kind of synchronization strategy. The most trivial of those strategies is also the one most commonly used, and forms the basis for *NEST*'s time management as well.

An arbitrarily chosen amount of time is defined as the basic resolution for a given simulation. Using this resolution, usually called Δ, the simulation is forced into a time grid with regular time steps. Global state updates will then occur in regular intervals of Δ and ensure that the neurons remain synchronized.

This means the whole simulation is based on a loop which updates the simulation state in

regular intervals, as shown in algorithm 2.

Algorithm 2 Simulator main loop used for network state updates

1: define resolution: $\Delta \leftarrow 0.1$
2: $T \leftarrow 0$
3: **while** $T < T_{stop}$ **do**
4: update network state to time T (i.e. trigger neuron updates)
5: increment network time with resolution: $T \leftarrow T + \Delta$
6: **end while**

Learning rules executed within the network updates can and must take into consideration that a time span of Δ has passed since the last update. This can be done by scaling the updates with the elapsed time, as demonstrated in equation 5.6 and [25]. Also, the resolution must be chosen carefully, because on the one hand a smaller resolution increases the simulation time, but on the other hand a larger resolution will cause the network to react slowly to changes.

Using this resolution, continuous processes can be broken down into update steps of interval Δ. As shown in section 2.2.2, the various events influencing the learning process can be nested so that in the end, an update function is called for every single time step. The broken down continuous processes are to be calculated here. Inside the function, the developer has access to the reward transmitter (or the current reward, depending on the implementation), the synaptic state variables, and the postsynaptic neuron. The latter can then be queried for whether a postsynaptic spike has occurred at the current time step or not.

Given that every single synapse runs the update function for every single time step, it is obvious that optimizing it is crucial.

The best way to optimize the function is to remove unnecessary calculations, for example:

- If a decaying property is already tiny and has no reason to rise, calculating more decay is wasteful. Instead, the property can be just set to 0.0 or ignored altogether, as demonstrated in listing 3.1.

- The generation of random numbers is computationally expensive. If a stochastic process can be calculated analytically for longer time periods - for example in the case of Wiener processes [41] - then this should be done.

- Constants like scaling factors of double exponential function (which are sometimes used for the calculation of PSPs, see listing 5.6) can be calculated once beforehand and stored, if possible in the synapse's common properties.

Finally, all configuration parameters of the model should be exposed as configurable settings to the simulation script so that parameter searches do not require recompilation of the extension

module, and base classes can be defined to reduce boilerplate code, as briefly demonstrated in section 3.2.

```
void CustomSynapse::updateSynapseVariables(...) {
  // Only decay PSP approximator if it is active, i.e. > 0
  if (isPSPActive()) {
    updatePresynapticSpikePotential(time, cp);
  }

  // Only decay eligibility trace if the PSP approximator
  // is active and the trace itself is > 0
  if (isEligibilityTraceActive()) {
    updateEligibilityTrace(time, postsynaptic_spike, cp);
  }
}
```

Listing 3.1: Performance gain by skipping unnecessary calculations. At the time of `updateSynapseVariables()`, the PSP approximator can only decay. So if it is already inactive, the calculation of further decay can be omitted. The same is true for the eligibility trace.

3.2 Reward transmitters

In reward-based learning simulations, external, output-dependent signals must be sent to not only the network nodes, but also the synapses. Since synapses can not be used to connect nodes to other synapses, *NEST* provides a way to register *Transmitters* (see figure 2.1) in synapses. The synapses can then access the transmitters to access (reward) data, and *NEST* even provides a built-in event trigger for when rewards are released.

Using these transmitters, reward-modulated plasticity can be implemented in *NEST*.

If the output neurons show activity at the appropriate time, a reward is sent using NEST's built-in `trigger_update_weight()`-method. This method is originally designed to handle dopamine rewards, but can be used for other reward types as well.

A reward transmitter can thus be implemented as a *NEST* node, which means its `update()` method is called in regular intervals. In here, the reward transmitter should calculate the current reward, and, if necessary, notify its synapses.

```
void RewardTransmitter::update(...) {
  // Rewards are often not sent at every update
  if (should_send_reward)
  {
    // Calculate the current spike rate
    double_t sum = 0;
    for (const double_t &value : spike_memory_) {
      sum += value;
    }
    double_t current_spike_rate =
      1000.0 * sum / (resolution_in_ms * spike_memory_.size());

    // Calculate current reward
    performRewardUpdate(current_spike_rate);

    // Notify synapses about current reward
    network()->trigger_update_weight();
    }
}
```

Listing 3.2: Reward transmitter update. In this example, the accumulated spike rate of all neurons connected to the reward transmitter is calculated from a spike memory that stores all spikes for a certain duration. `performRewardUpdate()` is then called to calculate the reward, before the synapses are notified. Here, the rewards are not sent to the synapses directly but need to be queried from the synapses themselves, as demonstrated in listing 3.3.

Since `trigger_update_weight()` is quite rigid, only allowing the passing of an array of floating values as reward, the reward transmitter may be fitted with accessor methods that can be called directly by the synapses. This of course causes a decrease in performance because the neurons have to query their data, but sometimes this drawback is necessary.

Listing 3.2 demonstrates how such a reward update could be implemented. When the weight update is triggered by `trigger_update_weight()`, the synapses use accessor methods like in listing 3.3 to retrieve the rewards.

```
CustomSynapse::trigger_update_weight(...) {
  // Retrieve reward from reward transmitter
  double_t reward = reward_transmitter_->getReward();

  // Perform reward handling, for example just buffer it for the
     next update.
  reward_history_.push(RewardHistoryEntry(time, reward));
}
```

Listing 3.3: Retrieval of rewards from reward transmitters.

Additionally, by turning the reward transmitter into a base class, other variants of transmitters can be easily written without having to copy the boilerplate code present in all *NEST* nodes. All they need to do is override the update calculation method, as demonstrated in listing 3.4.

```
// Virtual functions of the RewardTransmitter base class
class RewardTransmitterBase : public Archiving_Node {
  virtual double_t getReward() const
  {
    return B_.current_reward_;
  }
  virtual double_t getAverageReward() const
  {
    return B_.current_average_reward_;
  }

  virtual void performRewardUpdate(double_t spike_rate) = 0;
}
```

Listing 3.4: Virtual functions of a reward transmitter base class. `performRewardUpdate()` transforms the class into an abstract base class, meaning that RewardTransmitterBase can not be instantiated itself anymore. Child classes must be developed that implement at least `performRewardUpdate()`.

To make a synapse model eligible to receive rewards from the reward transmitter, it must register as a possible reward receiver. This is done by providing the synapse with the global

ID of the appropriate reward transmitter in the simulation script. The synapse will retrieve the transmitter from NEST and store it, if possible even in its common properties. This allows it to call methods of the transmitter, if necessary. An example implementation is shown in listing 3.5.

```
void CommonSynapseProperties::set_status(...) {
  long_t rtgid;
  if (updateValue<long_t>(d, "reward_transmitter", rtgid)) {
    reward_transmitter_ =
      dynamic_cast<RewardTransmitterBase*>(network.get_node(rtgid))
         ;
  }
}

nest::Node* CommonSynapseProperties::get_node()
{
  if (reward_transmitter_ == 0)
    return CommonSynapseProperties::get_node();
  else
    return reward_transmitter_;
}
```

Listing 3.5: Registering a reward transmitter in the synapse. In this demonstration, the reward transmitter is stored in the common properties of the synapse. `dynamic_cast` will set the member to 0 if it fails (i.e. an invalid ID was provided). This can either be ignored or handled as an error.

3.3 Data logging and debugging

Data logging can be accomplished in two ways. Either the neuron stores its historical data in buffers that can be accessed using the API (i.e. *PyNEST*), or a *NEST* recorder periodically retrieves the current data from its assigned neuron objects.

NEST by default implements a data recorder called *Multimeter* that can be used for such tasks. The neuron exposes variables that should be tracked to the recorder, which then periodically polls their values and stores them. Since *Multimeters* are nodes themselves, they can be easily connected to only a subset of neurons, greatly decreasing their memory consumption.

Listings 3.6 and 3.7 (both based on [45]) showcase how a neuron model can be configured to work with *Multimeters* in order to log data.

While this is a convenient way to store data, it is not as flexible as sometimes necessary. Due to the *Multimeter*'s regular updates, it is for example not possible to only record data when the researcher knows it is relevant.

```
// Read out the current membrane threshold
double_t get_Theta_() const { return threshold_potential_; }

// Register V_th as a valid variable to record
void RecordablesMap<CustomModule::CustomNeuron>::create() {
  // The first parameter denotes the out-facing name of the
     variable
  // The second parameter declares the corresponding accessor
     method
  insert_(names::V_th, &CustomModule::CustomNeuron::get_Theta_);
}
```

Listing 3.6: Registration of a recordable variable. First, an accessor method is defined that can return the desired variable. This method is then registered in a dictionary used by the multimeter.

```
neuron = nest.Create('CustomNeuron')

// 'record_from' specifies the data that should be logged
rec = nest.Create('multimeter', params={'record_from': ['V_th']})

nest.Connect(recorder, neuron)
nest.Simulate(...)
events = nest.GetStatus(rec,'events')
```

Listing 3.7: Logging a recordable variable. A multimeter is created and configured for the threshold potential. After the simulation, the data is available.

By having the neuron store its historical data itself, data logging can be heavily customized.

For example, it is straightforward to log only sparse data by discarding data irrelevant to the simulation. The addition of a boolean flag could then allow the researcher to toggle which neurons should log their data at all. Alternatively, an interval can be used to specify when data should be logged.

Another drawback of *Multimeters* is their inability to log data from synapses.

Listing 3.8 demonstrates a trivial approach of memorizing data, simply by storing it into `std::vector` objects. This way, the data can be retrieved from the simulation code by using the common *NEST* command `GetStatus()`, and it has the same layout as data retrieved from *NEST*'s built-in recorders.

Storing debugging data from synapses is sometimes necessary, but depending on the network very costly, given the sheer amount of synapses that might be present. Therefore, it is advisable to activate debugging only for a subset of synapses. Fortunately, in *NEST* models can be duplicated and then independently modified. Listing 3.9 shows how two populations of

synapses are created that have the same behaviour except for logging.

```
class CustomSynapse : public nest::Connection
{
  private:
  // Recorder buffers
  std::vector<double_t> recorder_times_;
  std::vector<double_t> eligibility_trace_values_;
  std::vector<double_t> psp_values_;
}

void CustomSynapse::get_status(DictionaryDatum & d) const
{
  (*d)["recorder_times"] = recorder_times_;
  (*d)["eligibility_trace_values"] = eligibility_trace_values_;
  (*d)["psp_values"] = psp_values_;
}

void CustomSynapse::updateSynapseVariables(...)
{
  // Perform updates according to model
  ...

  // Perform logging
  if (is_logging_active())
  {
    recorder_times_.push_back(time);
    eligibility_trace_values_.push_back(eligibility_trace_);
    psp_values_.push_back(psp_);
  }
}
```

Listing 3.8: Manual data logging of the eligibility trace and the PSP approximator of a synapse. Since both variables are recorded in the same interval, a single buffer to hold the recorder times is sufficient. Before the data is logged, a check is run to ensure that the data should be recorded. Reducing the amount of logged data reduces the memory load considerably.

```
# Synapse configuration with logging deactivated
synapse_configuration = {"weight":0.5, "recorder_interval":0.0}

# Copy the synapse configuration
logging_synapse_configuration = synapse_configuration.copy()

# Activate logging, log data once every 10 ms
logging_synapse_configuration["recorder_interval"] = 10.0

# Create new NEST model with logging activated
nest.CopyModel("custom_synapse", "logging_synapse",
   logging_synapse_configuration)

# Connect neurons without logging
nest.Connect(population1, population2, syn_spec="custom_synapse")

# Connect neurons with logging
nest.Connect(population1, population2, syn_spec="logging_synapse")
```

Listing 3.9: Activation of data logging for a subset of synapses

Chapter 4

Parallel computation in NEST

4.1 Architecture and performance of parallel simulations

Simulations of neural networks require large amounts of computational power. Performing the calculations in parallel is one way to tackle this challenge [8, 35].

The following paragraphs summarize how parallel computation works in *NEST*, based on the documentation in [38] and [46].

Aside from support for multi-threading, *NEST* implements parallelization by distributing the calculation over *Virtual Processes*. Each virtual process can, from the researcher's point of view, be considered equivalent regardless of the host on which the virtual process is run, making the simulation development easier and far less prone to errors.

Performance gains are achieved by having virtual processes hold only a part of the whole network and perform only the calculations relevant to their share of the network, as is illustrated in figure 4.1. Nodes are generally distributed evenly among all VPs, though it is possible to force nodes onto specific ones. The reasoning behind this is that neurons that are causally related interact more frequently and should be placed on the same process to reduce communication overhead. This technique can be helpful for simulations of for example cortical columns, where information about stronger connections might be known beforehand.

Nodes are substituted on foreign VPs by so-called proxies. They store no data and only act as placeholder. Synapses are always stored on the VP of the target neuron. Generators cause no significant overhead (compared to the rest of a decently sized network) and are therefore cloned onto every VP. Recorders also exist on every VP, but every recorder tracks only the data generated on its own process, thereby reducing communication overhead.

Although *NEST* handles most of this procedure automatically, the network update becomes more complicated (and thus slower), because the network state, network time and events must be synchronized over all virtual processes.

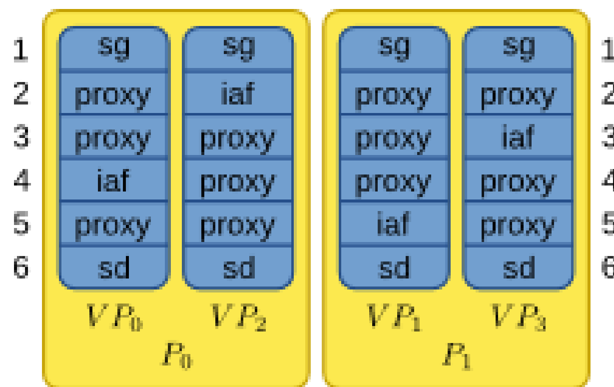

Figure 4.1: Node distribution in NEST, taken from [46]. The illustration considers a network with 6 nodes on 4 VPs. The VPs are in turn hosted on 2 hosts P_0 and P_1. Generators (sg) and recorders (sd) are cloned onto all VPs and are independent, neurons (iaf) exist only on a single VP and are substituted by proxies on the other VPs.

Algorithm 3 Parallelized computation loop in *NEST*, taken from [38]

1: $T \leftarrow 0$
2: **while** $T < T_{stop}$ **do**
3: **parallel on all** $vp \in N_{VP}$ **do**
4: deliver all events due
5: call U(S_T) for all nodes
6: **end parallel**
7: exchange events between VPs
8: increment network time: $T \leftarrow T + \Delta$
9: **end while**

This means that adding more threads and processes does not automatically increase performance.

Complex networks in which individual calculations, like weight updates, require the majority of the calculation time, benefit the most from parallel computations. In contrast, the overhead caused by the additional synchronization requirements quickly offset potential speed gains in simple networks where spike transfers and communication require more time than the calculations themselves.

Aside from the network and the hardware in use, the effectiveness of the parallelization approach also depends on the properties of the parallel environment.

- *Number of hosts*: A small number of hosts means that they might individually have too much pressure on them. But every host added to the simulation increases the communication overhead significantly due the distribution of processes onto separate servers.

- *Number of processes per host*: Multiple processes enable the simulation to make use of the multiple CPU cores available on current hardware and thereby significantly improve performance. However, if the number of processes exceed the host's capabilities (i.e. the CPU executes them serially and not in parallel) then only communication overhead is added without gaining any performance in return.
- *Number of threads per process*: Most modern CPUs are capable of high performance multithreading. This adds considerably less communication overhead than processes do, but burdening the CPU with too many threads will also cause a drastic reduction of performance.
- *Neuron distribution*: As mentioned, clever distribution of neurons based on their connections may improve performance by allowing causally connected neurons to communicate faster.

Since all those factors influence each other and many are hardware-specific, there are no general rules on when and how to use a parallel environment.

Generally speaking though parallel setups, even if only multi-threading is used, will likely increase performance in most cases. *NEST* has shown that even in benchmark scenarios with simple networks, supra-linear speed-up can be achieved until up to 8 virtual processes. Complex networks have displayed supra-linear scaling with up to 80 virtual processes [38].

4.2 Random numbers

Even if a neural network does not implement stochastic features, random numbers are often required in order to either initialize the network to an independent state or to generate random training and test samples.

The *NEST* documentation explains in [47] how random numbers should be handled. This information is summarized in this section.

Ideally, the whole simulation would only access a single random number generator. As this would require far too much locking and synchronization effort, the virtual processes of simulators often each use a generator of their own. However, since the random number generators are likely initialized with different seeds on different virtual processes, special considerations must be taken into account for them in parallel environments.

The two requirements for random numbers are that they must be synchronized across multiple virtual processes, and simulations using random numbers must be reproducible (this is also true for non-parallel simulations). If every simulation run were to be calculated with different seeds, debugging and analysis would become almost impossible to do. Fortunately, *NEST* allows the researcher to seed its random number generators manually, as shown in listing 4.1.

```
random_seed = 20000 # Seed for NEST
random.seed(10000) # Seed python's RNG
numpy.random.seed(30000) # Seed numpy's RNG
N_vp = nest.GetKernelStatus(["total_num_virtual_procs"])[0]
pyrngs = [numpy.random.RandomState(s) for s in range(random_seed,
   random_seed+N_vp)]
nest.SetKernelStatus({"grng_seed" : random_seed+N_vp})
nest.SetKernelStatus({"rng_seeds" : range(random_seed+N_vp+1,
   random_seed+2*N_vp+1)})
```

Listing 4.1: Seeding the NEST random number generators, based on [47]

By setting explicit seeds, results can be reproduced if necessary. Additionally, all virtual processes executing this script will use the same seeds. This synchronization is important because, while virtual processes will naturally end up with the same results in deterministic computations, the almost guaranteed presence of random numbers will produce contradicting results across the processes if the various simulator instances do not use the same random seed.

Aside from that, `pyrngs` in listing 4.1 contains a distinct random number generator for every virtual process. They are used for cases in which random numbers are generated that must explicitly be independent from each other on different virtual processes.

For example, in listing 4.2, all synaptic weights are randomized at the beginning of the simulation. The code showcases both the access to the synapses (in *NEST*, synapses are only stored on the virtual process of the target neuron) and the usage of the per-process random seeds.

```
node_info = nest.GetStatus(neurons)
local_nodes = [(ni['global_id'], ni['vp']) for ni in node_info if
   ni['local']]
for gid, vp in local_nodes:
  synapses_for_gid = [synapse for synapse in structural_synapses if
      synapse[1] == gid]
  for synapse in synapses_for_gid:
    nest.SetStatus([synapse], {"weight":pyrngs[vp].uniform(
       min_initial_weight, max_initial_weight)})
```

Listing 4.2: Randomizing synaptic weights in *NEST*, based on [47]

Thus, as long as all random number generators used in the simulation are managed as aforementioned, parallel simulations can be run correctly and with reproducible results.

4.3 Development of simulations for parallel environments

While simulators generally take care of most kernel-side requirements for parallel computing, it is likely that user-written simulation code has to be adapted manually.

Since the parallel environment is likely managed by the *Massage Passing Interface* standard [10], the required changes are demonstrated with it.

At the beginning of the simulation, MPI must be included. It might be worth it to consider that some users of the experiment have no access to MPI-enabled features.

```
mpi_enabled = True
try:
  from mpi4py import MPI
except ImportError:
  mpi_enabled = False
```

Listing 4.3: Enabling MPI if it is available on the system

The simple code block shown in listing 4.3 sets a boolean flag depending on whether MPI is available on the system. By checking the flag before running code specific to MPI, unnecessary errors are avoided.

Secondly, random numbers must be handled correctly as described in section 4.2.

Finally, plotting and printing must be handled manually, for example using *MPI4PY*, in order to consolidate results. In order to avoid redundant data on multiple hosts, every virtual process in *NEST* has its own data recorders and only tracks changes occurring on itself, as described in section 4.1. Simply displaying data as one would do in single-process applications would therefore only yield multiple incomplete fragments.

To consolidate the data, the results from the different processes must be gathered and displayed only on a single process. *MPI4PY* conveniently offers a functionality called `Gather()` for this particular purpose.

Listing 4.4 demonstrates how the event arrays stored inside the entity buffers can be gathered by MPI and merged into a single array which can then be displayed normally.

This procedure is slightly different for node recorders and synapses, because synapse data is stored in tuples, while recorder data is stored in arrays. This is an implementation detail of *NEST* though and might be subject to change at any given time.

Note the use of `mpi_enabled`, ensuring that the code will also run as expected when multiple processes are not supported.

```
# Status retrieval of local nodes
local_hidden_events = nest.GetStatus(spike_detector, "events")[0]

# Only connections targeting neurons on the process return
   meaningful data
local_connectionStati = nest.GetStatus(nest.GetConnections(),["
   source","target","weight"])

# Consolidate results
if mpi_enabled == True:
  # MPI Gathering
  comm = MPI.COMM_WORLD
  gathered_hidden_events = comm.gather(local_hidden_events, root=0)
  gathered_connectionStati = comm.gather(local_connectionStati,
     root=0)
else:
  gathered_hidden_events = [local_hidden_events]
  gathered_connectionStati = [local_connectionStati]

# Initialize data structures for consolidated results
global_hidden_events = {"times":[], "senders":[]}
global_connectionStati = ()

if nest.Rank() == 0:
  # Populate global data structures

  for hidden_event in gathered_hidden_events:
    global_hidden_events["times"] = np.append(global_hidden_events[
       "times"], hidden_event["times"])
    global_hidden_events["senders"] = np.append(
       global_hidden_events["senders"],hidden_event["senders"])

  for connectionStatus in gathered_connectionStati:
    global_connectionStati += connectionStatus

  # Proceed with plotting and printing based on global_* arrays.
  ...
```

Listing 4.4: MPI data consolidation

Chapter 5

Examples

5.1 Step Poisson Generator

Generators are nodes that create input for the network, either in the form of currents or spike trains.

NEST ships with a number of generators that suffice for most cases, but as is the case with neurons it is sometimes necessary to develop a custom generator.

For example, *NEST* includes a *Step Current Generator* and a *Poisson Generator*. The *Step Current Generator* can be configured to send specific constant currents at different simulation times, and the *Poisson Generator* sends spike trains into the network based on a Poisson distribution with an expected value of λ. Inconveniently, λ can only be set once at the beginning of the simulation, and remains the same during the run of the simulation. There is no way (as of yet) to generate Poisson spike trains that change their λ at specific time intervals.

Therefore, a *Step Poisson Generator* is shown that works like the *Poisson Generator*, except that it can use different λ at different simulation times. The Generator uses a probability mass function similar to the standard Poisson distribution, only with an added time parameter, as described in equation 5.1, where $k = 0, 1, 2, ...$ denotes possible numbers of spikes.

$$f(k; \lambda; t) = Pr(X = k \mid t) = \frac{{\lambda_t}^k \cdot e^{-\lambda}}{k!} \tag{5.1}$$

The first step of implementing the generator is to specify the necessary state variables.

Just like the *Step Current Generator* stores current values, *Step Poisson Generator* stores λ values, as can be seen in listing 5.1.

Then, as shown in listing 5.2, at every update, the current λ is updated and an appropriate number of spikes is generated. The example implementation simply borrows the Poisson device also used in the implementation of the built-in *Poisson Generator* and reconfigures it with the

```
class StepPoissonGenerator : public Node {
  struct Buffers_ {
    size_t idx_; // index of current lambda
    double_t lambda_; // current lambda
  };
  struct Parameters_ {
    // A list of times when lambda should change
    std::vector<double_t> lambda_times_;
    // A list of values to which lambda should change
    std::vector<double_t> lambda_values_;
  };
};
```

Listing 5.1: Step Poisson Generator state variables, based on *NEST*'s Step Current Generator.

```
void StepPoissonGenerator::update (...) {
  for (long_t lag = from; lag < to; ++lag) {
    poisson_dev_.set_lambda(resolution_in_ms_ * rate_ * 1e-3);
    if (device_.is_active() && rate_ > 0) {
      DSSpikeEvent se;
      network ()->send (*this, se, lag);
    }
  }
}

void StepPoissonGenerator::event_hook (DSSpikeEvent& e) {
  librandom::RngPtr rng = net_->get_rng(get_thread());
  long_t n_spikes = V_.poisson_dev_.ldev (rng);

  if (n_spikes > 0) { // Do not send events with multiplicity 0
    e.set_multiplicity(n_spikes);
    e.get_receiver().handle (e);
  }
}
```

Listing 5.2: Step Poisson Generator spike generation, based on *NEST*'s Poisson Generator.

current λ to calculate the number of spikes.

The final *Step Poisson Generator* can be easily configured with λ change times and values, demonstrated in listing 5.3. While this is a simple example, it clearly shows that being able to write their own extension modules, even trivial ones, can quickly raise the comfort researchers have testing their theories.

```
step_poisson_generator = nest.Create("step_poisson_generator", 10)
times = [0.01, 1000.0, 2000.0, 3000.0, 4000.0]
lambda = [10.0, 50.0, 100.0, 50.0, 10.0]
nest.SetStatus(step_poisson_generator, {"lambda_times":times, "
    lambda_values":lambda})
```

Listing 5.3: Step Poisson Generator usage. The result can be seen in figure 5.1.

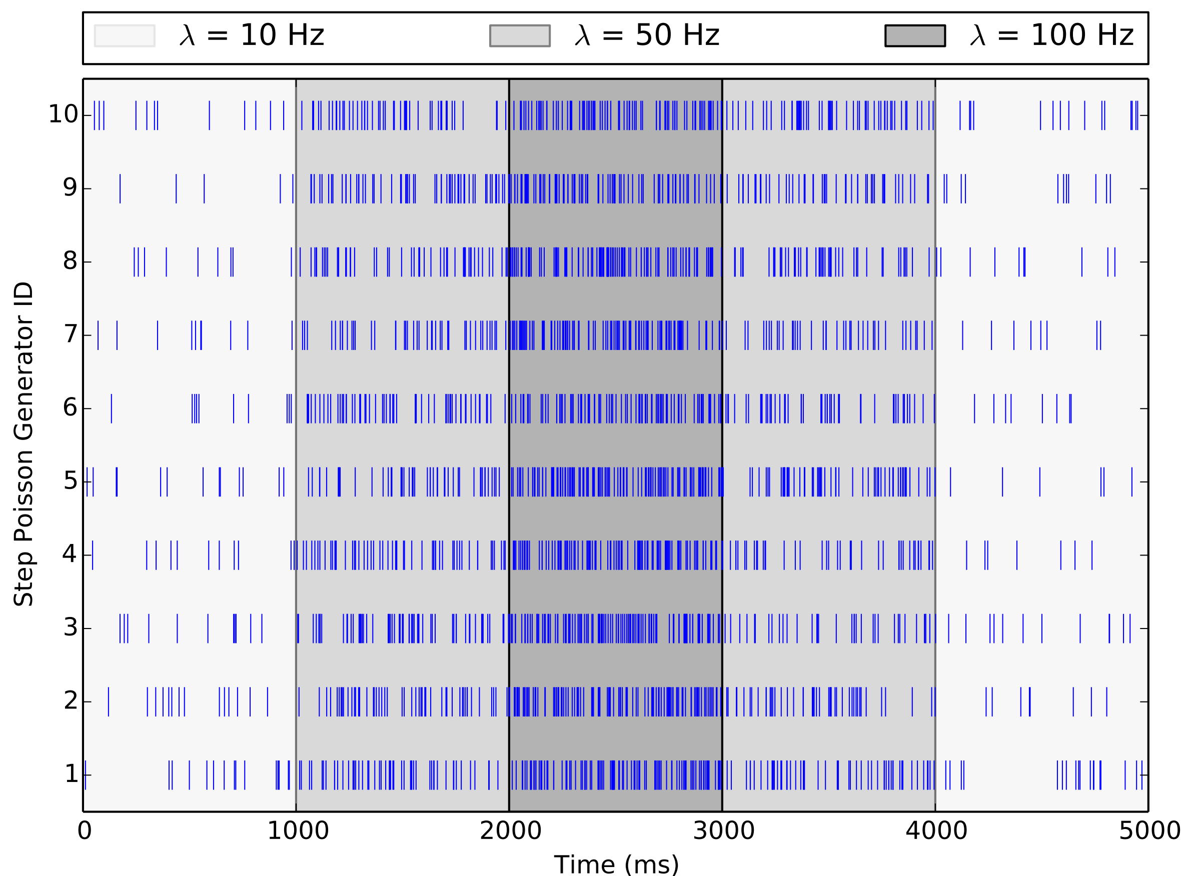

Figure 5.1: Step Poisson Generator spike trains based on the simulation code shown in listing 5.3. Each spike train is produced by an independent Step Poisson Generator. The generators are initially configured with an estimated mean spike rate λ of 10 Hz. During the simulation, the estimated rates are changed once for every simulated second, first to 50 Hz, then to 100 Hz, then back to 50 Hz and finally again to 10 Hz.

5.2 Bias Neuron

A problem often encountered when working with spiking neural networks is the lack of a stable spiking behaviour.

For example, should the environmental stimulus be too weak to cause spiking activity in the

network (a leaky integrate-and-fire neuron requires a constant current of at least 376 mV for the membrane potential to reach the common spike threshold of −55 mV if it does not receive spikes), it simply does nothing.

On the other hand, if the environmental stimulus is too strong, the neurons might start to spike uncontrollably at their maximum spike rate, suffering the simulated equivalent of a seizure. This can occur quite easily in recurrently connected networks where feedback loops can happen.

It would thus be convenient to have a mechanism that automatically dampens or heightens the network's sensitivity, ideally at the neuron level. The goal is to reduce the spike likelihood of neurons that have too much spike activity, and do the opposite for neurons that do not spike often enough.

Various solutions can be used to accomplish this task. An additional stimulus could be sent into the neuron that changes its strength based on the neuron's spike rate. Alternatively, the additional stimulus stays the same and a synapse is trained to change its weight appropriately. The stimulus can be either a current, or it could even be a spiking neuron. Given its convenience and predictability, the former option is probably the better choice if the network model does not explicitly demand the latter one.

Alternatively, the neuron itself could adapt its state variables to achieve the desired effect, completely circumventing external stimuli. Manipulating the spike threshold is a trivial example for that.

In *NEST*, both strategies can be easily accomplished. Faking an external current is built-in in the standard neurons, and the manipulation of the spike threshold simply requires a change to the associated state variable.

In this example, the threshold approach is chosen and implemented in the *Bias Neuron*. This neuron works the same as the standard *Integrate-and-fire neuron* (and in fact uses the neuron code as its template), but it will change its spike threshold depending on its spike rate. The desired minimum and maximum spike rates, the update speed and the neuron's memory length can all be configured from the simulation script.

Equation 5.2 summarizes the desired behaviour, where T is the threshold potential, μ is the bias update rate, V_r is the membrane resting potential, z the current spike rate and z_{max} and z_{min} are the highest and lowest acceptable spike rates respectively.

$$\Delta T = \begin{cases} \mu, & \text{if } z > z_{max} \\ -\mu, & \text{if } z < z_{min} \text{ and } T - \mu > V_r \\ 0, & \text{otherwise} \end{cases} \tag{5.2}$$

The neuron holds a ring buffer consisting of one entry for every n last update steps, where n is the memory length. If the average sum of the values in the ring buffer exceed the maximum

spike rate, the threshold is increased, thereby making it harder for the neuron to spike. If the average sum is lower than the minimum spike rate, the opposite happens.

Obviously, the bias neuron's update size must be very small, lest it interferes with learning. The learning update size must be significantly larger than the bias update size, otherwise the learning algorithm will interpret performance changes due to the bias as results of the learning process.

Furthermore, it is very important to choose a sufficiently long memory length. The reason being, that spike rates are defined as spikes per *second*. If the memory length is shorter than that, then the spike rate will be extrapolated linearly from the spikes within the memory length. This is fine as long as the neuron spikes fairly regularly. However, if the neuron only spiked a lot for a fraction of a second, and the memory length is not much longer than that, a far too high spike rate is calculated for that period, potentially destabilizing the learning process. This must be kept in mind especially for the first second of simulation, where nothing more than extrapolated data is available.

```
void BiasNeuron::update(...) {

  // Calculate the current spike rate
  double_t current_rate =
    1000.0 * outgoing_spikes_.sum () / (std::min(time_in_ms,
       bias_memory_length_));


  if (current_rate > bias_spike_rate_maximum_) {
    // Current spike rate is too high, increase threshold
    Theta_ += bias_update_size_;
  }
  else if (current_rate < bias_spike_rate_minimum_) {
    // Current spike rate is too low, decrease threshold
    // Make sure that the threshold is still higher than the
       resting potential
    Theta_ = std::max (V_reset_ + 0.0001, Theta_ -
       bias_update_size_);
  }
}
```

Listing 5.4: The threshold potential is increased and decreased in steps of `bias_update_size`, depending on the current spike rate. The code makes sure that the threshold potential can not become smaller than the resting potential.

Listing 5.5 showcases the usage of the *Bias Neuron* by connecting it to a step current generator. The generator changes the bias input every second, prompting a change of the threshold potential in the *Bias Neuron*. In this simulation, the neuron is tasked with maintaining a spike

rate of 10 Hz. The effort of the neuron can be seen in figure 5.2.

```
bias_neuron = nest.Create("bias_neuron")
bias_neuron_configuration = {"bias_spike_rate_minimum":10.0, "
   bias_spike_rate_maximum":10.0, "bias_update_size":0.001, "
   bias_memory_length":100.0}
nest.SetStatus(bias_neuron, bias_neuron_configuration)
nest.Connect(generator, bias_neuron)
```

Listing 5.5: Bias neuron usage

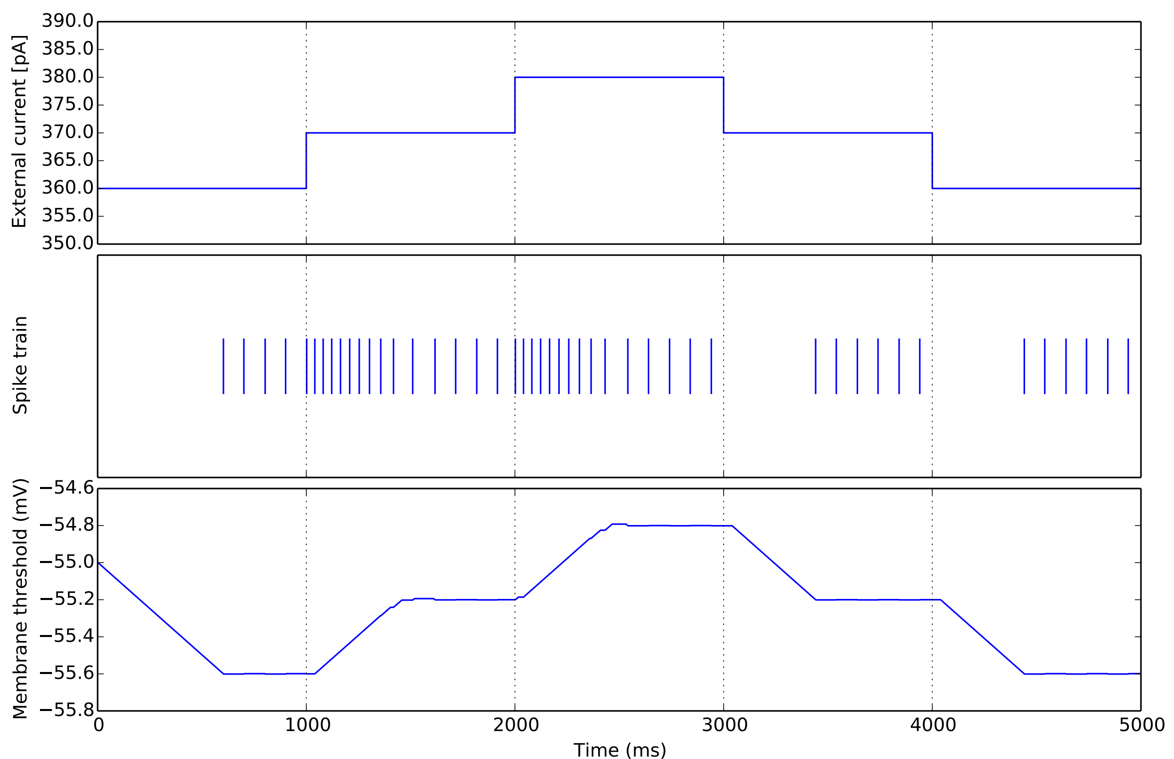

Figure 5.2: Bias neuron usage. The first plot shows the neuron input, with step-wise changes every second. The neuron's spikes are visible in the second plot. Notice how the neuron returns to a spike rate of 10 Hz shortly after every change in the input current. The third plot visualizes the changes in the membrane threshold. The threshold does not change as long as the desired spike rate is maintained, but will promptly adjust if this is not the case.

5.3 Digit Classification

This example is based on [23]. The goal is to train a network for the classification of *MNIST* [28] images depicting different handwritten digits.

The experiment uses a straightforward approach of mapping the images to network input.

Every pixel of a image is mapped to a single input neuron. This neuron bases its spiking activity on the gray-scale intensity of its image. Specifically, in this example spikes are generated based on Poisson distributions with the pixel intensity as base for their expected values.

The pixel intensities are normalized so that the input neurons spike in a predictable way regardless of the digits that are used. The values are changed in a way that ensures that estimated values for any given input neuron are limited between a predefined interval.

Given a set $\boldsymbol{Q}$ of values representing the pixel intensities of a *MNIST* image, the corresponding set $\boldsymbol{\lambda}$ of expected values for the input neuron Poisson distributions is generated by applying the common linear scaling method shown in equation 5.3 to every $\boldsymbol{Q}_i$ in $\boldsymbol{Q}$ to calculate the corresponding λ_i in $\boldsymbol{\lambda}$. X_{min} and X_{max} denote the desired minimal and maximal spike rates respectively.

$$\lambda_i = X_{min} + \left((\boldsymbol{Q}_i - \boldsymbol{Q}_{min}) \cdot \frac{X_{max} - X_{min}}{\boldsymbol{Q}_{max} - \boldsymbol{Q}_{min}} \right) \tag{5.3}$$

The input is provided by *Step Poisson Generators* described section 5.1, while the output neurons are *Bias Neurons* from section 5.2.

In the particular case of *NEST*, STDP synapses typically cannot connect directly to generators, so the generators first have to send their spike trains to so-called *parrot neurons* that in turn can mirror their input to the actual hidden neurons with STDP synapses featuring a learning process based on in [23].

The network is thus a feed-forward network using generators as mappings for *MNIST* digits, which use parrot neurons to indirectly send spikes to the output neurons. Thus, any single output neuron is connected to every input neuron.

5.3.1 Part 1: Learning without rewards

In the first part of the experiment, the network receives no rewards whatsoever.

The synaptic update of the learning process is described in equations 5.4 and 5.5 (taken from [23]), where w_{ki} is the synaptic weight between the presynaptic neuron i and the postsynaptic neuron k, μ is the learning rate and z_k is a classifier denoting whether the neuron k has spiked or not ($z_k = 0$ if neuron has not spiked, $z_k = 1$ if neuron has spiked).

$$\Delta w_{ki} = \mu \, z_k \, (\hat{z}_i - \exp(w_{ki})) + \sqrt{2T\mu} \, \nu \tag{5.4}$$

$$\nu \sim \text{NORMAL}(0,1) \tag{5.5}$$

The term « $\sqrt{2T\mu}\, \nu$» is a simple random walk process that adds some noise based on the temperature T to the synapse, as described in section 1.3.

The PSP approximator $\hat{z}_i$ is described in section 1.2.

One way to calculate the PSP approximator $\hat{z}_i$ is to approximate it using a double exponential function. Of course, this is only possible if the PSP of neuron behaves similarly to the chosen function. Listing 5.6 demonstrates a sample implementation. In the `send()` function of the synapse, two exponential functions are incremented by 1 for every presynaptic spike, mimicking an action potential. Assuming that a facilitation rate and a depression rate are predefined the code then combines the two exponential functions into a single $\hat{z}_i$.

```
void updatePSPApproximator(double_t dt_last_update) {

  // Function shape: u/(u-v) * (e^(-dt/u) - e^(-dt/v))

  psp_facilitation_ *=
    exp(-1.0 * dt_last_update / psp_fac_rate_);
  psp_depression_ *=
    exp(-1.0 * dt_last_update / psp_dep_rate_);

  psp_approximator_ =
    (psp_fac_rate_/(psp_fac_rate_ - psp_dep_rate_)) *
    (psp_facilitation_ - psp_depression_);
}
```

Listing 5.6: PSP approximator update. It generates a shape similar to figure 1.3a.

From the update rule it becomes apparent that the synaptic weight does not change ($\Delta w_{ki} = 0$) if the postsynaptic neuron has not spiked ($z_k = 0$). The simulation can therefore omit the calculation of the weight update in those cases. The update rule in equation 5.4 can therefore be transformed to equation 5.6, which is only run when a postsynaptic spike has occurred ($z_k = 1$) in the time step.

$$\Delta w_{ki} = \mu \; t_k \; (\hat{z}_i - \exp(w_{ki})) + \sqrt{2T\mu t_k} \; \nu \tag{5.6}$$

Since the calculation is omitted between postsynaptic spikes, every time the weight update is calculated the time that has passed since the last update must be taken into account. Thus, the factor t_k, denoting the time since the last update, is added into both the actual learning part and the random walk process of the update rule in equation 5.6.

After the synaptic weights have learned for a while, every neuron will have "decided" for a digit and spike only if that particular digit is shown to the network.

This learning process produces decent results, as shown in figure 5.3, but the network can hardly be called a reliable classifier if every neuron is allowed to randomly "specify" a digit it will identify. This happens because, at the beginning, each neuron just fires at random. Then if,

for a short time depending on the learning rate, its spike behaviour happens to coincide with the appropriate behaviour for a certain digit, the STDP mechanism will cause the neuron to slowly fixate on the digit. In a sense, the digits serve as attractors to the neurons, but which neurons happen to move into which digit's attractor basin is completely random.

Figure 5.3: Digit classification without learning, visualizing the synaptic weights after showing the network 1000 images for 50 ms each. Every sub-figure shows the incoming synapses for a single output neuron, in a layout consistent with the *MNIST* input. As can be easily seen, most neurons specify one of the four possible digits (2, 3, 4, or 5).

Aside from having no control over the digit each neuron learns, this also means that the network must consist of a relatively large number of neurons to increase the probability that every digit is learned by at least one neuron.

Consequently, in the second part of the experiment, the learning process is reinforced with a reward mechanism.

5.3.2 Part 2: Reward-based learning

This example uses a very basic form of reward-based learning. At regular time intervals, the synapses receive their rewards based on their target neuron's performance. If a positive reward is received, the last few weight changes are kept. If no reward is received, the last few weight changes are discarded, effectively resetting the synapse to an earlier state. Equation 5.7 shows the handling of the weight w for every time step ρ in which a reward r is received.

$$w_\rho = \begin{cases} w_\rho, & \text{if } r_\rho = 1 \\ w_{\rho-1}, & \text{if } r_\rho = 0 \end{cases} \tag{5.7}$$

A subclass of *Reward Transmitter* (as discussed in section 3.2) has to be developed that can be configured with a target digit. Every neuron is connected to a single reward transmitter, and it spikes while its transmitter's target digit is shown to the network, it receives a reward. The same happens if the spiking activity stops while the digit is not shown.

To tune the reward transmitter a bit stricter, it is configured to only send rewards when the performance actually improves, not when it stays the same. For example, no reward is sent if the neuron spikes with a constant rate while its digit is shown. It has to increase its spike rate to receive a reward. This effectively means that synapses can only permanently change their weights if the change resulted in a stronger recognition of the target digit. This behaviour is shown in equation 5.8, where x_t is the neuron's spike rate at time t and y_t is a classifier denoting whether the reward transmitter's digit was shown to the network at time t ($y_t = \infty$ if digit was shown, $y_t = 0$ if another digit was shown).

$$r_t = \begin{cases} 1, & \text{if } |y_t - x_t| < |y_{t-1} - x_{t-1}| \\ 0, & \text{otherwise} \end{cases} \tag{5.8}$$

It is important to note that this is only feasible because there is no punishment mechanism included in the learning process - if no reward is sent, the network simply resets. If this was not the case, a reward transmitter like the one described could potentially hurt the network performance by having the synapse weights move away from reasonably good values.

Figure 5.4 shows that neurons were indeed forced to recognize predefined digits.

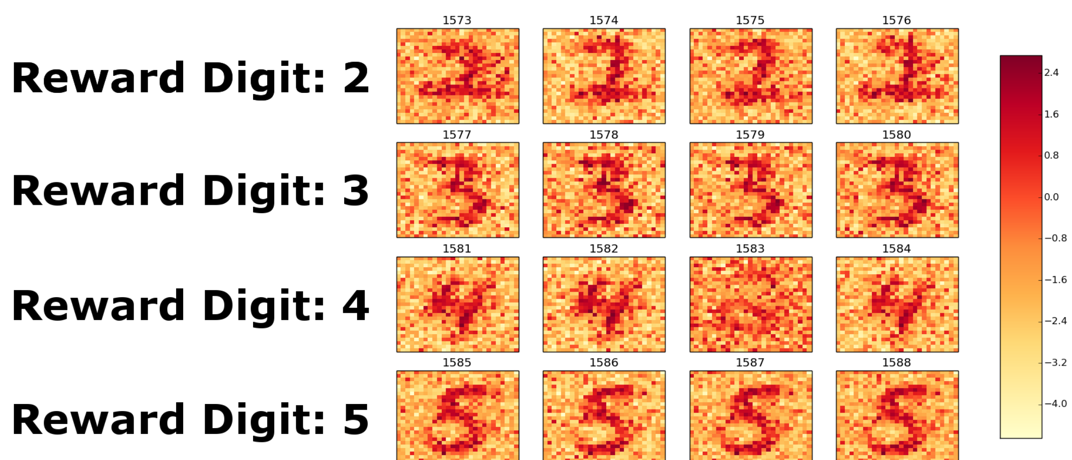

Figure 5.4: Synaptic weights after learning with rewards. Each row represents a group of neurons with a different target digit. The neurons have not perfectly learned their desired digits yet, but it is apparent that they will if given enough learning time.

5.4 Learning a sine wave spike rate pattern

When a new learning algorithm is tested, it seems obvious to start with a simple experiment. One apparently trivial task is to train a neuron to imitate another neuron's spike rate.

Such a network consists of only a few input neurons, an output neuron and a reward transmitter.

The input neurons can be either generators or neurons. Again, the *Step Poisson Generator* comes in handy. The input neurons are connected to the output neuron, but not to each other. One of these neurons is designated as target neuron, and its spike rate is the one that should be imitated by the output neuron.

The output neuron is connected to the input neurons via autonomously learning synapses. In this case, the usage of biased neurons would skew the results too much, so a stochastic neuron model (as described in section 1.2) is used for the output neurons.

Finally, the reward transmitter is configured to offer a high reward if the output neuron spikes at a frequency similar to the target neuron. One possible solution for this is to exponentially decay the reward the farther the actual value is away from the target, as shown in equation 5.9. In this equation, κ can be used to specify the rate of the decay. r, y and x are the reward, the target value and the actual value, respectively.

$$r = \exp^{-\kappa\ (y-x)^2} \tag{5.9}$$

Figure 5.5 demonstrates the effect of different decay rates.

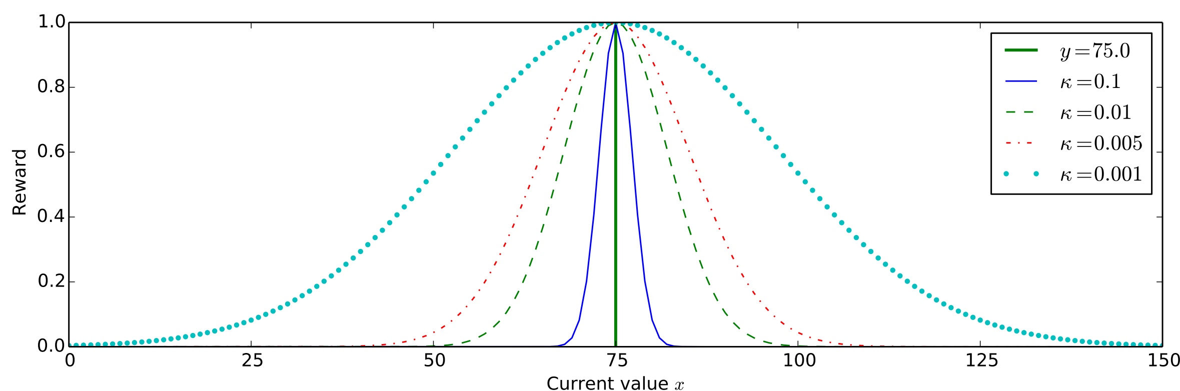

Figure 5.5: Comparison of reward decay rates. The target value y is set to 75. Bigger values for the decay rate κ cause the reward transmitter to become stricter, while small κ values will result in decent rewards even when the current value x is very different from y.

Algorithm 4 summarizes the synaptic learning rule used in this simulation, which is a simplified version of the synapse model presented in [25] (see section 1.3).

Algorithm 4 Synaptic model of reward-based learning [25]

At each time step t, $\forall k, i$:

1: update eligibility trace: $e_{ki}(t) = e_{ki}(t - \Delta t)e^{-\Delta t/t_{\mathrm{ep}}} + t_{\mathrm{ep}}\, W_{ki}\, \hat{z}_i(t)\, (z_k(t) - f_k(t))$

2: update synaptic parameters: $\theta_{ki}(t) = \theta_{ki}(t - \Delta t) + \Delta\theta_{ki}(t)$

with $\Delta\theta_{ki}(t) = \eta \left(\frac{1}{\sigma^2} (\mu - \theta_{ki}(t - \Delta t)) + \frac{r(t)}{\bar{r}(t)}\, e_{ki}(t) \right) + \sqrt{2T\eta}\, \nu$

and $\nu \sim$ NORMAL(0,1)

3: update synaptic weight: $W_{ki} = \exp(\theta_{ki} - \theta_0)$

4: update average expected reward: $\bar{r}(t) = (1 - \tau_r)\, r(t - \Delta t) + \tau\, r(t)$

In the algorithm, k represents the output neuron, while i denotes one of the input neurons. The other terms are variations of the terms used in equation 1.5 in section 1.3. The prior term (Ω) demands that the synapse exerts some effort in order to change its weight and significant deviations from the attractor must be warranted by high rewards. μ and σ are parameters that configure this prior. The reward in this case is dependent on a reward average $\bar{r}(t)$, which in turn decays with an integration time constant of τ. The result of this is multiplied with the eligibility trace $e_{ki}(t)$ discussed in section 1.3 in order to calculate a trajectory for the weight update (ψ in section 1.3). This, combined with the prior and the random walk term (the same one as in 5.4), results in the overall change of the weight.

For this experiment, two input neurons are used that spike with rates that increase and decrease in a sine wave pattern. The wave pattern of one neuron is phase shifted relative to the other neuron. The goal is to have the output neuron decide for one of the input neurons and imitate its spike rate pattern while learning to ignore the other input neuron. Figure 5.6 illustrates the spike rates of the two input neurons.

Running the simulation with various parameter configurations keeps producing results like the one in figure 5.7. It does look promising, but apparently the experiment is not as simple as it appears to be and it quickly turns out that balancing the parameters is quite tricky.

If the temperature is set too high, the neuron's spike rate is not reliable enough to learn anything meaningful. If it is set too low, there is little incentive for the neuron to differ its spike rate from its natural attractor. Waiting for that to happen by chance will likely significantly increase the required simulation time, making it more tedious to experiment with the parameters. Similar statements can be made for the other parameters. As long as the neuron receives a reward, even

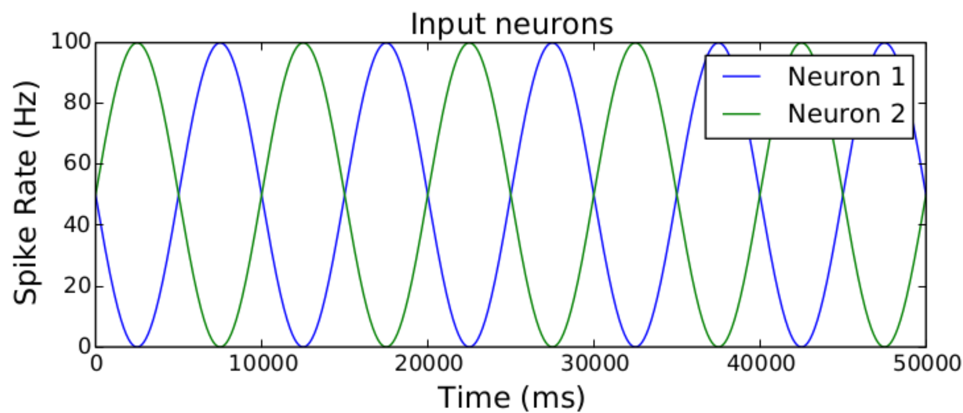

Figure 5.6: Input neurons with sine wave pattern spike rates. The spike rates follow a sine wave pattern with a mean of 50 Hz, an amplitude of 50 Hz and a frequency of 0.1. The phase shift of neuron 1 is π radians, the phase shift of neuron 2 is 0 radians.

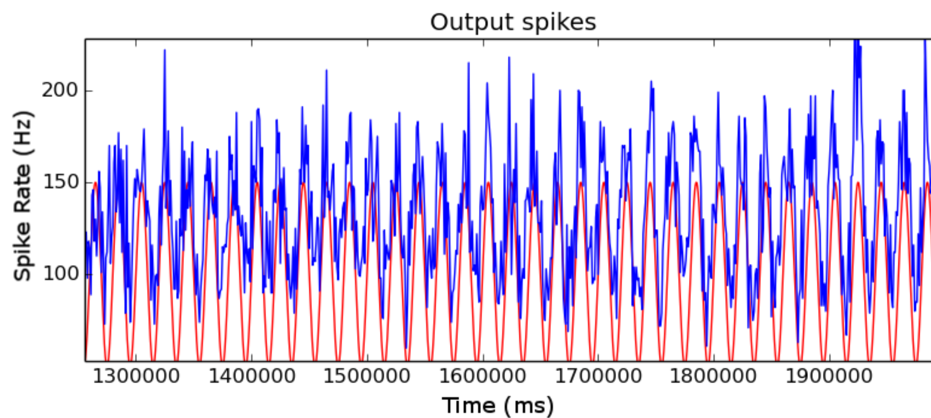

Figure 5.7: Spike rate of a stochastic neuron trained to spike following a sine wave pattern. In red, the desired output spike rates. In blue, the actual spike rates.

if it is a small one, the synapse weight continues to increase. This is unfortunately also true if the synapse weight has reached the global optimum. This is for example the reason why the spike rate in figure 5.7 keeps overshooting the desired spike rates and cannot seem to reach low spike rates. In the whole time during which the output neuron correctly decreases its spike rate, the reward increases the synapse's weight, causing the spike rate to rise unwittingly.

It can therefore be shown that the learning process, though theoretically sound, does have some difficulties with tasks like the one that was tested. Fortunately, from these simulation results alone a number of possible ways to improve the model have already become apparent.

Implementing some kind of stabilization mechanism could prevent the weights to change too much while rewards are received. The mechanism could in theory cause the synapse to just offset its prior with its weight changes as soon as a sufficiently high reward is consistently reached.

There is no "punishment" when the reward decreases. While higher rewards will actively increase the synaptic weights, lower rewards will merely reduce the effect of the eligibility trace, relying solely on the prior to reduce the weight. This could help to mitigate the kind of overfitting observed in figure 5.7.

Utilizing some kind of simulated annealing to gradually decrease the strength of the noise could speed up the learning process without breaking it. With the model presented, the noise must be very small to not interfere with learning. Regrettably, this means that it can take a long while for the synapse weight to randomly find a value good enough to warrant a reward. As soon as a reward is received, the learning process takes over, but until then, random walk and the prior are the only things changing the synaptic weight. Annealing would allow the synapse to start with a high degree of noise and then quickly reduce it as soon as a basin of reward has been found. Even better, mechanism could increase the noise at any time during the simulation if the synaptic weight somehow leaves the reward space again. However, it is not clear how the addition of such a mechanism could be justified in the theoretical model.

For all of these suggestions, the underlying theoretical model needs to be re-evaluated. But this is a good thing, since the simulation thereby might have helped to improve the model.

Even without those changes though, correctly configuring the various parameters of the model can cause a significant increase in performance, as can be seen in figure 5.8. The parameter search can be very tedious and time-consuming though, motivating further improvements to the model.

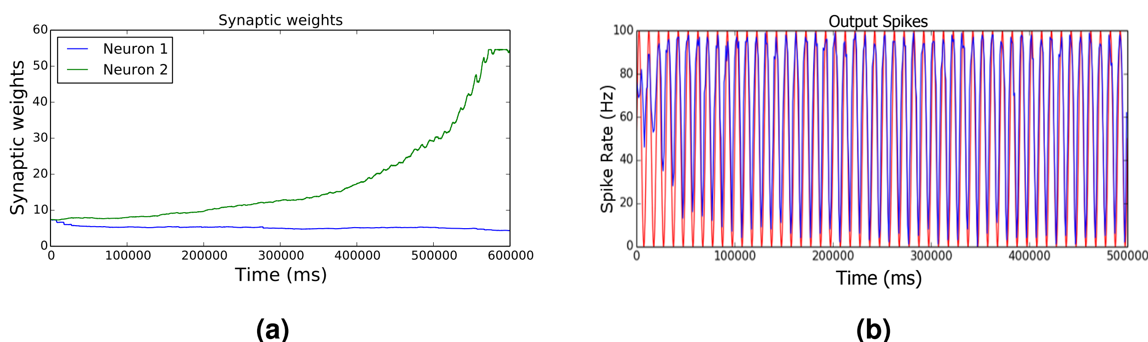

Figure 5.8: Successful simulations of sine wave learning. **(a)** Weight changes in a successful simulation run. The synaptic weight between neuron 1 and the output neuron diminishes while the weight from neuron 2 keeps increasing until it reaches the appropriate value. **(b)** Another successful simulation run, this time showing the spike rate of the output neuron. The output neuron's spike rate aligns perfectly with the target sine wave pattern.

5.5 Learning to navigate a double-T-maze decision task

Inspired by [40], this experiment implements a simple T-maze decision task from [25], as illustrated in figure 5.9. There are two junctions, each modelled by an input neuron (s_1 and s_2). The goal is for the agent to navigate the maze and retrieve the highest reward possible. The agent's decision relies on an action neuron a_1 which is connected to s_1 and s_2 with synaptic weights w_1 and w_2, respectively. If it is active, the agent goes right. If it is inactive, the agent chooses the left way.

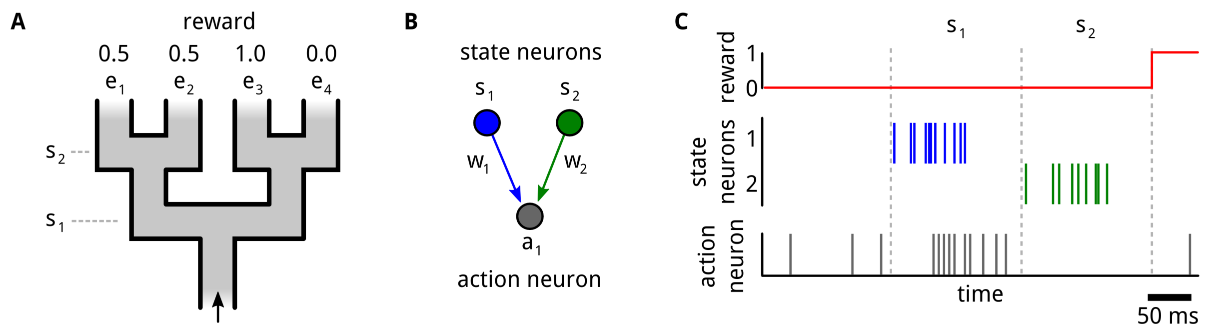

Figure 5.9: Double-T-maze decision task, taken from [25]. **(A)** Illustration of the structure of the maze. **(B)** The network consists of only 2 input neurons (s_1 and s_2), the action neuron (a_1), an a reward transmitter (not illustrated). The connections from s_1 and s_2 to a_1 are denoted by w_1 and w_2. At the first junction, s_1 is active. At the second junction, s_2 is active. The highest reward is behind exit e_3. **(C)** A successful episode, in which a_1 spikes together s_1 but not with s_2.

If the network were to only utilize STDP without any rewards, the agent would merely memorize the effects of decisions it makes inside the maze. Simply put, if it happened to fixate on going the wrong way, it would certainly recall that there is not going to be a reward at the end. That however would not stop it from making the wrong decision every time.

Reward-modulated STDP causes the agent to only memorize the action that led to a reward, thereby ensuring that it always picks the right way after a few training runs.

Since rewards in this example can by definition only be distributed at the end of each episode, the eligibility trace is far more important here than in examples 5.3.2 and 5.4, where rewards could theoretically be sent instantly.

The eligibility trace is calculated as shown in equation 1.4, and the other synaptic parameters discussed in section 1.3 are defined similarly to algorithm 4. However, ψ is implemented in a more complex way, resulting in the following update procedure:

Algorithm 5 shows that the learning rules for the maze task are almost identical to those in algorithm 4, only with the addition of a reward gradient ψ that acts as a memory for previous learning trajectories (based on the rewards and eligibility traces of previous episodes).

Algorithm 5 Synaptic model of reward-based learning with a reward gradient [25]

At each time step t, $\forall k, i$:

1: update average expected reward: $\bar{r}(t) = (1 - \tau_r)\, r(t - \Delta t) + \tau\, r(t)$

2: update eligibility trace: $e_{ki}(t) = e_{ki}(t - \Delta t)e^{-\Delta t/t_{\text{ep}}} + t_{\text{ep}}\, W_{ki}\, \hat{z}_i(t)\, (z_k(t) - f_k(t))$

3: update reward gradient: $a_{ki}(t) = \psi = \left(1 - \frac{r(t)}{\bar{r}(t)}\right) a_{ki}(t - \Delta t) + \frac{r(t)}{\bar{r}(t)}\, e_{ki}(t)$

4: update synaptic parameters: $\theta_{ki}(t) = \theta_{ki}(t - \Delta t) + \Delta\theta_{ki}(t)$

with $\Delta\theta_{ki}(t) = \eta\left(\frac{1}{\sigma^2}\,(\mu - \theta_{ki}(t - \Delta t)) + a_{ki}(t)\right) + \sqrt{2T\eta}\, \nu$

and $\nu \sim$ NORMAL(0,1)

5: update synaptic weight: $W_{ki} = \exp(\theta_{ki} - \theta_0)$

Figure 5.10 shows how the reward-modulated STDP learning rule manages to learn the task at hand, strengthening w_1 while weakening w_2.

Figure 5.11 illustrates the spike rates of the three neurons at the beginning and at the end of a simulation run. As can be clearly seen, the agent is successfully trained using the learning rule, suggesting that the model works as intended.

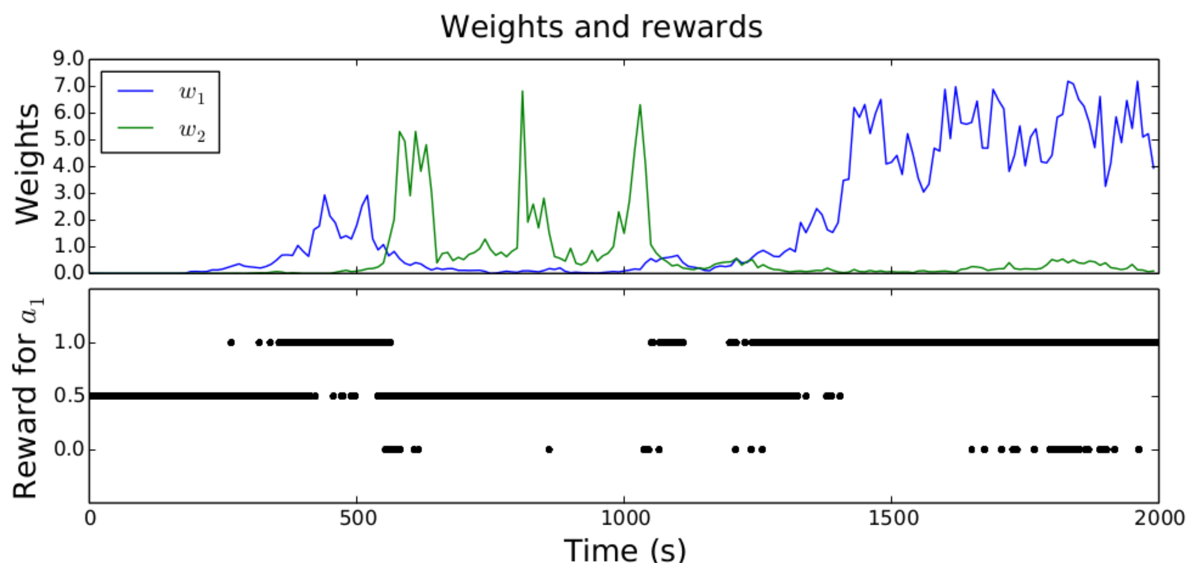

Figure 5.10: Successful learning of a double-T-maze decision task. **Top:** The synaptic weight w_1 between s_1 and the action neuron a_1 gradually increases, while w_2 between s_2 and a_1 declines, which is the optimal solution for this task. **Bottom:** Reward sent by the reward transmitter. At first the network mostly receives rewards of 0.5, because the neuron spikes randomly and there are more decisions that lead to this reward. As soon as the weights are learned appropriately, the performance drastically increases.

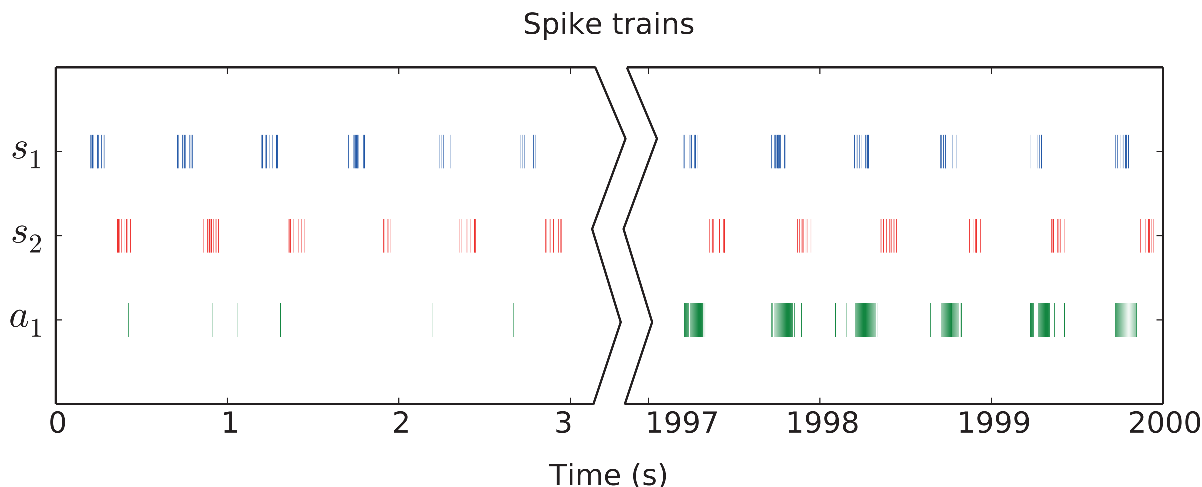

Figure 5.11: Spike trains during successful double-T-maze learning. The plot shows how every episode starts with a pause, after which s_1 becomes active, followed by s_2. After that, a new episode starts. The simulation is run for 2000 seconds. At the beginning of the simulation, a_1 has very low activity and spikes randomly, if at all. In the last 3 seconds of the simulation, a_1 has been successfully trained to activate together with s_1 and cease spiking as soon as s_2 becomes active.

Chapter 6

Conclusion

Even though researchers nowadays are provided with potent tools in the form of both powerful hardware and well-designed software, their proper usage still poses a decent challenge.

The complex nature of biological spiking neural networks force simulation software to provide a multitude of features, starting from customizable neurons to support for parallel computation, which inevitably increase the difficulty to work with them.

The time management alone can cause a variety of bugs and confusion, due to the need to mimic continuous real time while only having discrete update steps to work with. The fact that models are often explained using time-dependent differential equations that need to be mapped to the simulator's grid-based time does not help either. When combined with parallel environments, this challenge becomes even more difficult, because multiple processes or even hosts need to be synchronized. It is apparent that a thorough understanding of the simulator's time management is required to reliably develop simulations for it. Fortunately, if the simulator is cleverly designed, it can manage most of the synchronization effort itself, even in parallel settings. The researcher is still left with occasional pitfalls and some boilerplate code, but as long as the underlying principles are understood, they can be avoided.

Additionally, having decent understanding of the simulation framework from a software developer's point of view does not only help prevent mistakes, but can greatly increase the quality of the simulation. For example, when the researcher does not need to rely on the simulator's built-in data loggers which will hardly ever produce the exact data in the exact granularity needed, vast flexibility and precision is possible. By recording only necessary data from the right sources both the runtime and the memory performance can be increased, let alone the researcher's comfort when analyzing the results.

Knowing the details on how the simulator handles updates and events can also proof invaluable. Mathematical models often need to define their formulas using continuous changes coupled with control statements like Kronecker Deltas (i.e. $\Delta x = ... \cdot \delta(y, z)$) and the Heav-

iside step function (i.e. $\Theta(x) = 1$ for $x \geq 0$ and 0 otherwise [25]), but simulators can raise events exactly when they are needed. This can be used to eliminate unnecessary calculations in many steps or even fast-forward expensive operations like random number generations. By defining a solid update strategy based on the mathematical model, but adapted to the simulator's capabilities, valuable calculation time can be saved.

As the case studies have shown, even theoretically sound models may exhibit weaknesses that are only obvious through simulations. But visualizations don't only show failings, but can also inspire ideas. For example, the necessity to adjust learning rates, volatility of noise or even the learning rule itself might become apparent. Even the addition of mechanisms like simulated annealing or a way to stabilize weights after they have been trained could spring to mind.

The ability to experiment with parameters and try out different combinations of models comfortably and in decent time is crucial to gain a better understanding of the models and improve them, or maybe even verify their feasibility as in the case of Bayesian reward-modulated learning [25].

Therefore, while working with simulators can at times be tedious, spending resources to thoroughly understand their inner workings and developing the simulations with as much care as the underlying models might be well worth the effort.

Bibliography

[1] L. F. Abbott and S. B. Nelson. “Synaptic plasticity: taming the beast.” In: *Nature neuroscience* 3 Suppl (Nov. 2000), pages 1178–1183. doi:10.1038/81453.

[2] M. Baudry, R.F. Thompson, and J.L. Davis. *Synaptic Plasticity: Molecular, Cellular, and Functional Aspects*. A Bradford book. MIT Press, 1993.

[3] K. C. Berridge and T. E. Robinson. “What is the role of dopamine in reward: hedonic impact, reward learning, or incentive salience?” In: *Brain research. Brain research reviews* 28.3 (Dec. 1998), pages 309–369.

[4] M. Botvinick and M. Toussaint. “Planning as inference”. In: *Trends in Cognitive Sciences* 16.10 (2012), pages 485–488.

[5] Natalia Caporale and Yang Dan. “Spike Timing–Dependent Plasticity: A Hebbian Learning Rule”. In: *Annual Review of Neuroscience* 31.1 (Feb. 14, 2008), pages 25–46. doi:10.1146/annurev.neuro.31.060407.125639.

[6] D. B. Chklovskii, B. W. Mel, and K. Svoboda. “Cortical rewiring and information storage”. In: *Nature* 431.7010 (Oct. 14, 2004), pages 782–788. doi:10.1038/nature03012.

[7] Computational Cognitive Neuroscience Laboratory. *Comparison of Neural Network Simulators*. Nov. 30, 2015. https://grey.colorado.edu/emergent/index.php/Comparison_of_Neural_Network_Simulators.

[8] Mikael Djurfeldt et al. *Massively parallel simulation of brain-scale neuronal network models*. Technical report TRITA-NA-P0513. Stockholm: KTH, School of Computer Science and Communication Stockholm, 2005.

[9] Jochen Martin M. Eppler et al. “PyNEST: A Convenient Interface to the NEST Simulator.” In: *Frontiers in neuroinformatics* 2 (2008). doi:10.3389/neuro.11.012.2008.

[10] Message P Forum. *MPI: A Message-Passing Interface Standard*. Technical report. Knoxville, TN, USA, 1994.

[11] Nicolas Frémaux, Henning Sprekeler, and Wulfram Gerstner. "Functional Requirements for Reward-Modulated Spike-Timing-Dependent Plasticity". In: *The Journal of Neuroscience* 30.40 (Oct. 6, 2010), pages 13326–13337. doi:10.1523/jneurosci.6249-09.2010.

[12] Wulfram Gerstner and Werner M. Kistler. *Spiking Neuron Models: Single Neurons, Populations, Plasticity*. 1st edition. Cambridge University Press, Aug. 26, 2002.

[13] Marc-Oliver Gewaltig and Markus Diesmann. "NEST (NEural Simulation Tool)". In: *Scholarpedia* 2.4 (2007), page 1430.

[14] J. Ghorbanian, M. Ahmadi, and R. Soltani. "Design predictive tool and optimization of journal bearing using neural network model and multi-objective genetic algorithm". In: *Scientia Iranica* 18.5 (2011), pages 1095–1105. doi:10.1016/j.scient.2011.08.007.

[15] Samanwoy Ghosh-Dastidar and Hojjat Adeli. "Spiking Neural Networks". In: *International Journal of Neural Systems* 19.04 (2009). PMID: 19731402, pages 295–308. doi:10.1142/S0129065709002002. eprint: `http://www.worldscientific.com/doi/pdf/10.1142/S0129065709002002`.

[16] W.L. Goh, D.P. Mital, and H.A. Babri. "An artificial neural network approach to handwriting recognition". In: *Knowledge-Based Intelligent Electronic Systems, 1997. KES '97. Proceedings., 1997 First International Conference on*. Volume 1. May 1997, 132–136 vol.1. doi:10.1109/KES.1997.616872.

[17] Stefan Habenschuss. *Synaptic and structural plasticity as MCMC sampling*. Technical report. 2013.

[18] A. Herrmann and W. Gerstner. "Noise and the PSTH Response to Current Transients: I. General Theory and Application to the Integrate-and-Fire Neuron". English. In: *Journal of Computational Neuroscience* 11.2 (2001), pages 135–151. doi:10.1023/A:1012841516004.

[19] Anthony J. Holtmaat et al. "Transient and persistent dendritic spines in the neocortex in vivo." In: *Neuron* 45.2 (Jan. 20, 2005), pages 279–291. doi:10.1016/j.neuron.2005.01.003.

[20] Anthony Holtmaat and Karel Svoboda. "Experience-dependent structural synaptic plasticity in the mammalian brain". In: *Nature Reviews Neuroscience* 10.9 (Sept. 1, 2009), pages 647–658. doi:10.1038/nrn2699.

[21] Eugene M Izhikevich. "Solving the distal reward problem through linkage of STDP and dopamine signaling". In: *Cerebral cortex* 17.10 (2007), pages 2443–2452.

[22] G. Kallianpur and R. Wolpert. “Weak Convergence of Stochastic Neuronal Models”. In: *Proceedings of a Workshop on Stochastic Methods in Biology*. Nagoya, Japan: Springer-Verlag New York, Inc., 1987, pages 116–145.

[23] David Kappel. *Binary Classification with Spiking Neurons*. Technical report. 2014.

[24] David Kappel, Stefan Habenschuss, Robert Legenstein, and Wolfgang Maass. “Network Plasticity as Bayesian Inference”. In: *PLoS Comput Biol* 11.11 (Nov. 2015), e1004485. doi:10.1371/journal.pcbi.1004485.

[25] David Kappel et al. “Reward-based Network Plasticity as Bayesian Inference”. In preparation, 2015.

[26] Haruo Kasai et al. “Structural dynamics of dendritic spines in memory and cognition”. In: *Trends in Neurosciences* 33.3 (Mar. 5, 2010), pages 121–129. doi:10.1016/j.tins.2010.01.001.

[27] Konstantin Kravtsov, Mable P. Fok, David Rosenbluth, and Paul R. Prucnal. “Ultrafast all-optical implementation of a leaky integrate-and-fire neuron”. In: *Opt. Express* 19.3 (Jan. 2011), pages 2133–2147. doi:10.1364/OE.19.002133.

[28] Yann Lecun and Corinna Cortes. *The MNIST database of handwritten digits*. Nov. 22, 2015. http://yann.lecun.com/exdb/mnist/.

[29] Robert Legenstein, Steven M. Chase, Andrew B. Schwartz, and Wolfgang Maass. “A Reward-Modulated Hebbian Learning Rule Can Explain Experimentally Observed Network Reorganization in a Brain Control Task”. In: *The Journal of Neuroscience* 30.25 (June 23, 2010), pages 8400–8410. doi:10.1523/jneurosci.4284-09.2010.

[30] Robert Legenstein, Dejan Pecevski, and Wolfgang Maass. “A Learning Theory for Reward-Modulated Spike-Timing-Dependent Plasticity with Application to Biofeedback”. In: *PLoS Comput Biol* 4.10 (Oct. 10, 2008), e1000180+. doi:10.1371/journal.pcbi.1000180.

[31] I.B. Levitan and L.K. Kaczmarek. *The Neuron: Cell and Molecular Biology*. Oxford University Press, 2015.

[32] Henry Markram, Wulfram Gerstner, and Per Jesper Sjöström. “A history of spike-timing-dependent plasticity”. In: *Frontiers in synaptic neuroscience* 3 (2011).

[33] Henry Markram, Wulfram Gerstner, and Per Jesper Sjöström. “Spike-timing-dependent plasticity: a comprehensive overview”. In: *Frontiers in Synaptic Neuroscience* 4.2 (2012). doi:10.3389/fnsyn.2012.00002.

[34] Warren McCulloch and Walter Pitts. “A logical calculus of the ideas immanent in nervous activity”. In: *Bulletin of Mathematical Biology* 52.1 (Jan. 1, 1990), pages 99–115. doi:10.1007/bf02459570.

[35] M. Migliore et al. “Parallel network simulations with NEURON”. In: *J. Comput. Neurosci* 21 (2006), pages 119–129.

[36] R.G.M Morris. “D.O. Hebb: The Organization of Behavior, Wiley: New York; 1949”. In: *Brain Research Bulletin* 50.5–6 (1999), pages 437–. doi:10.1016/S0361-9230(99)00182-3.

[37] S.A. Nazeer, N. Omar, and M. Khalid. “Face Recognition System using Artificial Neural Networks Approach”. In: *Signal Processing, Communications and Networking, 2007. ICSCN ’07. International Conference on*. Feb. 2007, pages 420–425. doi:10.1109/ICSCN.2007.350774.

[38] Hans E. Plesser et al. “Efficient Parallel Simulation of Large-Scale Neuronal Networks on Clusters of Multiprocessor Computers”. In: *Euro-Par 2007 Parallel Processing*. Edited by Anne-Marie Kermarrec, Luc Bougé, and Thierry Priol. Volume 4641. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007. Chapter 71, pages 672–681. doi:10.1007/978-3-540-74466-5_71.

[39] Filip Ponulak and Andrzej Kasinski. “Introduction to spiking neural networks: Information processing, learning and applications.” In: *Acta neurobiologiae experimentalis* 71.4 (2011), pages 409–433.

[40] A. Solway and M.M. Botvinick. “Goal-Directed Decision Making as Probabilistic Inference: A Computational Framework and Potential Neural Correlates”. In: *Psychological Review* 119.1 (2012), pages 120–154.

[41] Henry Stark and John W. Woods. *Probability and Random Processes with Applications to Signal Processing (3rd Edition)*. 3rd edition. Prentice Hall, Aug. 3, 2001.

[42] Gerald Tesauro. “Programming backgammon using self-teaching neural nets”. In: *Artificial Intelligence* 134.1–2 (2002), pages 181–199. doi:10.1016/S0004-3702(01)00110-2.

[43] The MathWorks, Inc. *MATLAB*. Nov. 30, 2015. http://mathworks.com/products/matlab.

[44] The MathWorks, Inc. *Neural Network Toolbox. Create, train, and simulate neural networks.* Nov. 30, 2015. http://mathworks.com/products/neural-network/.

[45] The NEST Initiative. *NEST Developer Space*. Nov. 22, 2015. http://nest.github.io/nest-simulator/.

[46] The NEST Initiative. *Parallel Computing*. Dec. 1, 2015. http://www.nest-simulator.org/parallel_computing/.

[47] The NEST Initiative. *Random numbers*. Dec. 1, 2015. http://www.nest-simulator.org/random_numbers/.

[48] R.F. Thompson. *The brain, a neuroscience primer.* New York: W.H. Freeman and Company, 1993.

[49] Marc Toussaint and Amos Storkey. "Probabilistic inference for solving discrete and continuous state Markov Decision Processes". In: *International Conference on Machine Learning* 23 (2006), pages 945–952.

[50] Joshua T. Trachtenberg et al. "Long-term in vivo imaging of experience-dependent synaptic plasticity in adult cortex." In: *Nature* 420.6917 (Dec. 19, 2002), pages 788–794. doi:10.1038/nature01273.

[51] Nikos Vlassis, Mohammad Ghavamzadeh, Shie Mannor, and Pascal Poupart. "Bayesian Reinforcement Learning". English. In: *Reinforcement Learning*. Edited by Marco Wiering and Martijn van Otterlo. Volume 12. Adaptation, Learning, and Optimization. Springer Berlin Heidelberg, 2012, pages 359–386. doi:10.1007/978-3-642-27645-3_11.

[52] Yi Zuo, Aerie Lin, Paul Chang, and Wen-Biao B. Gan. "Development of long-term dendritic spine stability in diverse regions of cerebral cortex." In: *Neuron* 46.2 (Apr. 21, 2005), pages 181–189. doi:10.1016/j.neuron.2005.04.001.