Master's Thesis

Analysis of Probabilistic Inference in Recurrent Networks of Spiking Neurons

by **Sabine Schneider**

Study Program: Computer Science

Institut für Grundlagen der Informationsverarbeitung (IGI)
Institute for Theoretical Computer Science
Faculty of Computer Science
Graz University of Technology
A-8010 Graz, Austria

Advisor and Assessor:
O.Univ.-Prof. Dipl.-Ing. Dr.rer.nat. Wolfgang Maass
Dipl.-Phys. Johannes Bill

Graz, December 2011

Deutsche Fassung:
Beschluss der Curricula-Kommission für Bachelor-, Master- und Diplomstudien vom 10.11.2008
Genehmigung des Senates am 1.12.2008

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 entnommene Stellen als solche kenntlich gemacht habe.

Graz, am …………………………… ………………………………………………..
(Unterschrift)

Englische Fassung:

STATUTORY DECLARATION

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 marked all material which has been quoted either literally or by content from the used sources.

…………………………… ………………………………………………..
date (signature)

Zusammenfassung

Die Entwicklung neuronaler Netzwerkmodelle wird aus verschiedenen Gründen vorangetrieben, zum einen um dem Ziel etwas näherzukommen, das echte biologische Gehirn und die darin ablaufenden Prozesse zu verstehen, und andererseits auch, um Probleme und Aufgaben der künstlichen Intelligenz zu lösen. Während der letzten Jahre und Jahrzehnte wurden nennenswerte Erfolge in beiden Bereichen erzielt, sowohl im Verstehen der Vorgänge im Gehirn, als auch in der Anwendung auf verschiedene Aufgaben. Spikende neuronale Netze, wie sie in dieser Arbeit verwendet werden, gehören der sogenannten dritten Generation von künstlichen neuronalen Netzen an, die durch den zusätzlichen Aspekt der Zeit in Form von Spikes charakterisiert sind. Kürzlich wurde eine neue Lernmethode - SEM - für spikende Netzwerke entwickelt. Weiters wurde ein implizites generatives Wahrscheinlichkeitsmodell entdeckt, das es erlaubt, probabilistische Inferenz, basierend auf wohldefinierten statistischen Mechanismen, in spikenden Netzen durzuführen, und dadurch Samples von Netzwerken spikender Neuronen zu ziehen.

Das erste Ziel dieser Arbeit ist es, die neu entdeckten Mechanismen in einer möglichst flexiblen und modularen Art zu implementieren. Ein effizientes objektorientiertes C++ Framework wird erstellt, das diverse Schnittstellen und Zugangspunkte auf verschiedenen Ebenen der Abstraktion auch zu anderen Programmiersprachen bietet.

In einem zweiten Schritt wird das mathematisch exakte Modell durch die Verwendung biologisch realistischerer neuronaler Dynamiken, etwa synaptische Verzögerungen, leicht abgeändert. Derartige Modifikationen werden systematisch auf (annähernde) Exaktheit analysiert, d.h. es wird untersucht, bis zu welchem Realitätsniveau das Modell noch verhältnismäßig verlässliche Resultate liefert. Dazu werden der systematische Fehler, der durch die Abänderung der Parameter entsteht, und die begrenzte Samplingzeit, die einen unvermeidbaren stochastischen Fehler zur Folge hat, verglichen. Daher kann es vorkommen, dass in zeitlich begrenzten Situationen, die zusätzlichen Kosten des nicht exakten Samplens vergleichsweise gering, und daher nicht relevant für das Verhalten sind.

Zusätzlich wird das Modell des Netzwerks auf ein stark vereinfachtes Modell des menschlichen visuellen Systems angewandt, um Mustererkennung auf einer symbolischen Ebene nachbilden zu können. In diesem Zusammenhang wird der Sampler in ein mehrschichtiges Netzwerk spikender Neuronen, die ein generatives Modell bilden, integriert. Dieses Modell wird mittels maximum Likelihood Methoden auf speziellen Mustern trainiert, und anschließend auf ähnlichen aber teilweise unvollständigen Bildern getestet, die es durch Informationsaustausch vervollständigen kann, nachdem die synaptischen Verbindungen während des autonomen Lernvorganges angepasst und gelernt wurden. Die Ergebnisse dieser Simulation werden mit den Resultaten biologischer Untersuchungen und Experimente des visuellen Systems verglichen, sodass gezeigt wird, dass die Methode des neuronalen Sampelns erfolgreich zur Simulation einiger biologisch begründeter Aspekte angewandt werden kann.

Stichwörter: Sampling; Spikende neuronale Netzwerke; Probabilistische Inferenz; Eigenständiges Lernen; Visuelles System

Abstract

The investigation of neural network models has two different major goals: to help people to infer the processes going on in real biological brains and to solve artificial intelligence problems. Over the last decades and years the evolution of these network structures lead to improvements in both - understanding the causes and effects of brain processes, and performing well in various problems of different fields. The networks studied in this thesis, i.e., spiking neural networks, belong to the so-called third generation of neural network models: spiking neural networks, being characterized by the addition of the aspect of time as spikes. Recently a novel method for training spiking neural networks has been developed, called SEM. Furthermore, an implicit generative model (i.e., a probability model) has been found, allowing to perform probabilistic inference on the basis of well-defined statistical mechanisms, and in this way to draw samples from networks of spiking neurons.

The first goal of this work is to implement this newly discovered mechanism in a highly flexible and modular fashion. An efficient object oriented C++ framework is developed providing different interfaces on various levels of abstraction to more high-level programming languages.

In a second step the mathematically exact neural network model is slightly modified using biologically more realistic neuronal dynamics like synaptic delay. Such modifications are systematically analyzed for their (approximate) correctness: It is explored up to which level of "realism" the model still provides reliable results. This is done by comparing the systematic error of modifying the parameters with the finite sampling time (and the so caused stochastic error) present anyways. Thus, in limited-time situations the cost of non-exact sampling might remain relatively low and might thus be not relevant for a behaving organism.

Additionally the neural network model is applied to a simplified model of vision, more precisely to pattern or image recognition and completion. In this context, the neural dynamics sampler is applied to a multilayered network of spiking neurons, representing a generative model. This network is trained on certain image patterns using maximum likelihood learning and then tested on similar but incomplete images which it is able to complete, using feedback information, after having learned and adapted the appropriate synapses during the autonomous learning period. The outcomes of this simulation are compared to findings of biological experiments in the visual cortical system, showing that neural sampling can successfully be applied to simulate some of the aspects found in biological experiments.

Keywords: Sampling; Spiking neural network; Probabilistic inference; Autonomous learning; Visual system

Acknowledgements

First of all I'd like to thank the institute for theoretical computer science (IGI) from Graz University of Technology - especially the faculty head O. Univ.-Prof. Dr. Wolfgang Maass - for offering me the possibility to join the ongoing work and research at the institute. I found it very interesting to see how work in the institute goes on and to join the seminar meetings with talks and presentations about recent research problems and findings of the institute-internal work.

The main ideas and concepts which form the basis of this master thesis where developed by Dipl.-Phys. Dr. Lars Büsing, Dipl.-Phys. Johannes Bill, Dipl.-Ing. Bernhard Nessler and O. Univ.-Prof. Dr. Wolfgang Maass. During the elaboration of my work I was particularly supported by Dipl.-Phys. Johannes Bill, who came up with many ideas and helped me to develop solution strategies and assisted me in solving any problems. I would like to thank all the people who supported me during this work, also the ones who I do not mention here by name.

This thesis was developed under the financial support of the EU projects FP7-243914 *Brain-i-Nets* (*Novel Brain-Inspired Learning Paradigms for Large-Scale Neuronal Networks*) and FP7-231267 *ORGANIC* (*Self-Organized Recurrent Neural Learning for Language Processing*).

Contents

1 Introduction

"Science has shown you that 'you', your joys and your sorrows, your memories and your ambitions, your sense of identity and free will, are in fact no more than the behavior of a vast assembly of nerve cells and their associated molecules; as Lewis Carroll's Alice might have phrased it: 'You're nothing but a pack of neurons."'

Francis Crick, 1994

1.1 Motivation and aim of this work

Since decades people try to "understand" the processes going on in their brains during ordinary actions. Even though it is commonly agreed nowadays that brains are built up as networks of billions of interconnected neurons, and even though the single sub-processes are quiet well explored today, it still remains a mystery how exactly a human being comes to some decision or even intuition. The unsolved problem thus is to understand *how* our brains really work.

One attempt to this understanding are artificial neural networks, making very strong simplifications, with the goal to understand - and possibly reproduce - tiny parts of these huge processes. So-called first generation artificial neural networks concentrate on the aspect of having (possibly connected) neurons, but they only allow for digital, i.e., binary, output signals, which seems quiet unrealistic. So the second generation networks implement some additional feature known from biology - namely weighted synapses - and thus enable analog output signals which seems already more plausible. But only the third generation of artificial neural networks is now capable of including the aspect of time, by using spikes instead of averaged firing rates.

It has recently been managed to apply well-defined and commonly known machine learning strategies to the field of more biologically realistic - spiking - neural networks, to apply probabilistic inference on recurrent networks of spiking neurons and to sample from them. This is a big step towards the comprehension of "thinking" - even though the final goal is still far away.

At this point it gets now really interesting to see what happens when further parameters are modified and a neural network model is driven to get more realistic in some sense. The work of this thesis is based on this background idea, namely to systematically analyze some possible settings - even though some of them do not exactly match the requirements of the theoretical proofs.

Additionally, this idea of a neural network that samples from probability distributions should now be analyzed in more detail by applying it to some practical example, in this case some very strongly simplified symbolic realization of the human cortical visual system as a multilayer network of sampling neurons, where the synaptic weights are learned using well-known machine learning techniques, and the effects of the resulting model are tested on a simplified model for visual processing.

These two fields are also the main expected outcomes of the work presented in the following, namely the analysis of different model settings and the application of the model to some practical problem, where the outcomes of biological experiments in the visual cortex should be simulated.

1.2 Overview

This thesis is subdivided into two major parts, the first one covering concepts and background information which can be seen as the basis of the work for this thesis, explaining and summarizing the fundamental ideas of the areas and topics underlying this thesis. In this first part, the following aspects are covered:

2 The visual system of the human brain: This section explains the biological and neuroscientific fundaments, starting with some brief explanation of the cortical structures in the human brain, with focus on the human cortical visual system, and finally summarizing some ideas on hierarchical structures in the brain, especially the visual areas.

3 Artificial neural networks: In this section the concepts of artificial or computational neuroscience are covered, from the general idea of an artificial neural network to the historical evolution of such networks, starting with some very early feed-forward models, going on with recurrent networks, and finally illustrating the idea of stochastic systems.

4 Statistical Background: As the title already indicates, in this section the mathematical and statistical theories and concepts, required to understand the neural network model used afterwards, are explained. First of all the general idea of sampling is mentioned, followed by the definition of MCMC methods, and finally also going into the field of bayesian inference and related models.

5 Spiking neural network models for probabilistic computation: This section contains the definition of some concepts developed at the institute for theoretical computer science of the Graz University of Technology, namely first of all the idea of SEM - spike-based expectation maximization - and after that also the novel theory about neural dynamics sampling, i.e., performing probabilistic inference in spiking neural networks by drawing samples.

After this first part of summaries and explanations of already well-known concepts, the second part is dedicated to new contributions, developed during the workout of this thesis. Again, a brief overview over the illustrated topics is given in the following:

6 A highly configurable software framework for neural sampling: In this section, the C++ implementation of the neural dynamics sampler, developed in order to have a fast, but still very flexible system required for the further analysis of different models, is presented on a high level, including some rough class structure, the different interfaces and also some examples how the implementation can be used.

7 Neural sampling with more realistic neuron models: This section contains the analysis of different model settings and the effects on the neural dynamics sampler, investigating the trade-off between theoretical exactness and biological reality. Additionally, experimental results are given, showing some dos and don'ts for selected model settings (e.g., in terms of shapes of postsynaptic potentials or synaptic delays).

8 Application of neural sampling to a simplified model of vision: This section is based on biological findings which suggest to view the human cortical visual system as a hierarchical structure having both, bottom-up but also top-down connections. The implementation of the neural dynamics sampler is applied to a hierarchical multilayer neural network model, representing on a symbolic level some strongly simplified model of the visual hierarchy. This multilayer model is then trained, and tested on (incomplete) visual input, in order to verify whether such an artificial model is able to do some simplified version of image completion - like it is done in a much more complex version in real human brains. In the end the behavior of the symbolic model is compared with the outcomes of some of the above mentioned biological experiments.

Part I
Recapitulation of underlying concepts

2 The visual system of the human brain

This section briefly introduces the main terms that can be seen as basic background knowledge for the scenarios described later on. In order to get deeper information about these topics one should refer to existing literature on neuroscience or on introductions to the human cortical systems, like [7], [59], [72] or many others.

The following sections will then deal in more detail with the visual system of human brains and it will be explained briefly how images and visual information are processed inside the brain.

2.1 Generic cortical structures

In nearly all living beings or animals, the brain is the center of the nervous system, which makes it a highly organized and complex system [74]. Even though the processes going on in single cells of the brain are investigated for decades, and are thus by now understood quiet well, the synergy of these cells and how an individual cell influences the entire system remains a mystery [32].

When looking at the overall structure of a brain, one talks about six main regions (cerebral cortex, cerebellum, temporal lobe, occipital lobe, parietal lobe and frontal lobe), each with complex internal structures, that make up the brain [5]. Some of these areas, like the cortex, which will be discussed in more detail later on, consist of convoluted layers of cells, due to the limited space, while other areas are clustered.

Human brains are composed of networks of about 100 billions connected units called neurons of various types [32], where the exact number depends, among others, on age and sex of an individual [63]. In order to get an idea about how such a neurons is structured, one can take a look at figure 2.1.

A neuron is a cell, having a cell body - also called soma - containing the nucleus. The soma is covered by the membrane. The membrane potential, representing the difference between the potential inside and outside the membrane of a neuron, will later on play a quiet important role in the simulations of artificial networks of neurons [19], [1].

This cell body of a neuron is continued in tree-structured dendritic arbors, and it has one long outgoing protoplasmic fibre, the axon, which carries trains of action potentials, and branches at the end. These transmitting ends of the branches are the synapses. Through them the electrical impulses are sent to the other neurons, either axoaxonic (from axon to axon), axosomatic (from axon to soma) or axodendritic (from axon to dendrite).

Usually each neuron is connected to other cells via thousands of synapses. In this context one refers to long range connectivity in case if a neuron's axon is longer than usual (sometimes a number of centimeters).

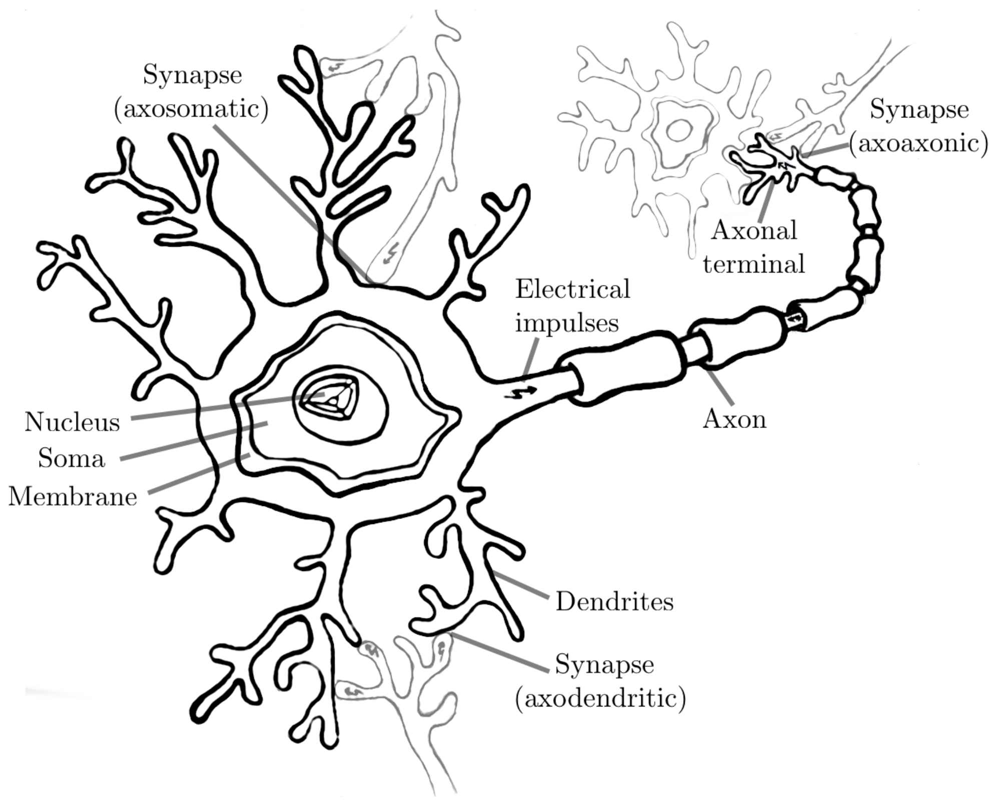

Figure 2.1: Structure of a typical neuron. The cell body or soma, which contains the nucleus, is covered by the membrane. The tree-like extensions are called dendrites. The axon - the long outgoing fibre - connects the neuron to other ones through synapses.

Each neuron can release chemical neurotransmitters [24]. In order to get a better idea of how the transmission of the electrical pulses through the synapses goes on, one can look at figure 2.2. It shows a strongly simplified overview of a synapse. The neurotransmitter molecules from the axon terminal are moved by diffusion to the receptors of the dendrite (or the other axon, or soma), which can shortly be described as the transmission of electrical impulses or signals between neurons.

The process of spiking is a complex biophysical process, where the electric pulse is only elicited through the axon if the potential inside the cell soma is sufficiently high, ofter modeled as a voltage threshold. After a spike, the neuron can only spike again after some refractory period has passed, which is the time the neuron requires to “recover” from firing [32].

Not all neurons are connected to all other ones, and there are usually more connections between neurons that are located nearby each other. Thus the neurons inside one brain areas are supposed to be responsible for “same” tasks, like vision or motor control, even though these areas interfere each other and are not separated strictly.

Even though one neuron is connected directly “only” to a few thousands of other neurons, a rise in its action potential above the threshold may cause effects in brain areas that are not directly connected to this neuron, by influencing the action potentials of directly connected units, which again affect their neighboring units.

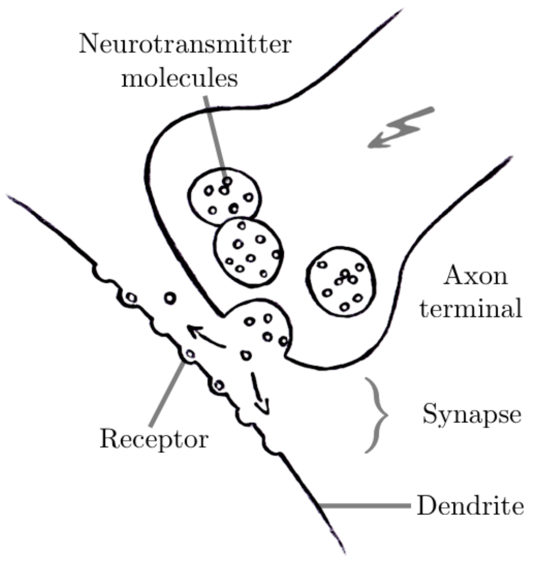

Figure 2.2: Synapse: Transmission of electrical impulses: Neurotransmitters are diffused from the axon terminal of one neuron to the receptors of another neuron (in this case an axodendritic synapse is shown).

2.2 The visual pathway

A large system in the human brain, known as the visual cortex, is responsible for the processing of visual information [40]. A very rough overview over the visual cortex is shown in figure 2.3, where some areas of the visual system are highlighted. These marked areas are among the ones which are explored and understood in much detail, and they will briefly be covered in the following explanations of the human visual pathway.

The visual pathway [62] starts at the retina of the eye, where some sensors get signals containing the visual information, i.e., the image one sees. The cells of the retina immediately respond to visual input, and transmit information about this input through optic nerve fibres to the LGN, the lateral geniculate nucleus, which is located in the thalamus - the mid area - of the human brain. A LGN is located in both, the left and the right thalamus of the brain, where both LGNs only get half of the visual information.

Among the areas in the human brain which are best studied there is the primary visual cortex V1 [69], which it is strongly connected to the LGN. Each brain hemisphere's V1 gets information directly from the corresponding LGN. The exact connectivity of the visual areas V1, V2, V3, V4 and V5/MT depends on the species of the living being, where the following refers to humans.

The primary visual area is connected to two main pathways to which information flows: the dorsal, and the ventral stream [51]. In figure 2.3 it can be seen that both streams connect V1 to V2, but in both cases different information is processed: The dorsal stream is associated with information about the “where” and “how”, like motion, and it is further connected to the visual area MT. On the other side the ventral stream is associated with information about the “what”, like object recognition and memory. It further sends information to V4 and finally the IT.

Another quiet well investigated brain area is the secondary visual cortex V2. V2, together with V1, is primarily supposed to be responsible for edge and corner detection, as well as for basic color and motion information [40], and the neurons inside this cortex are usually tuned to respond to particular orientations of bars.

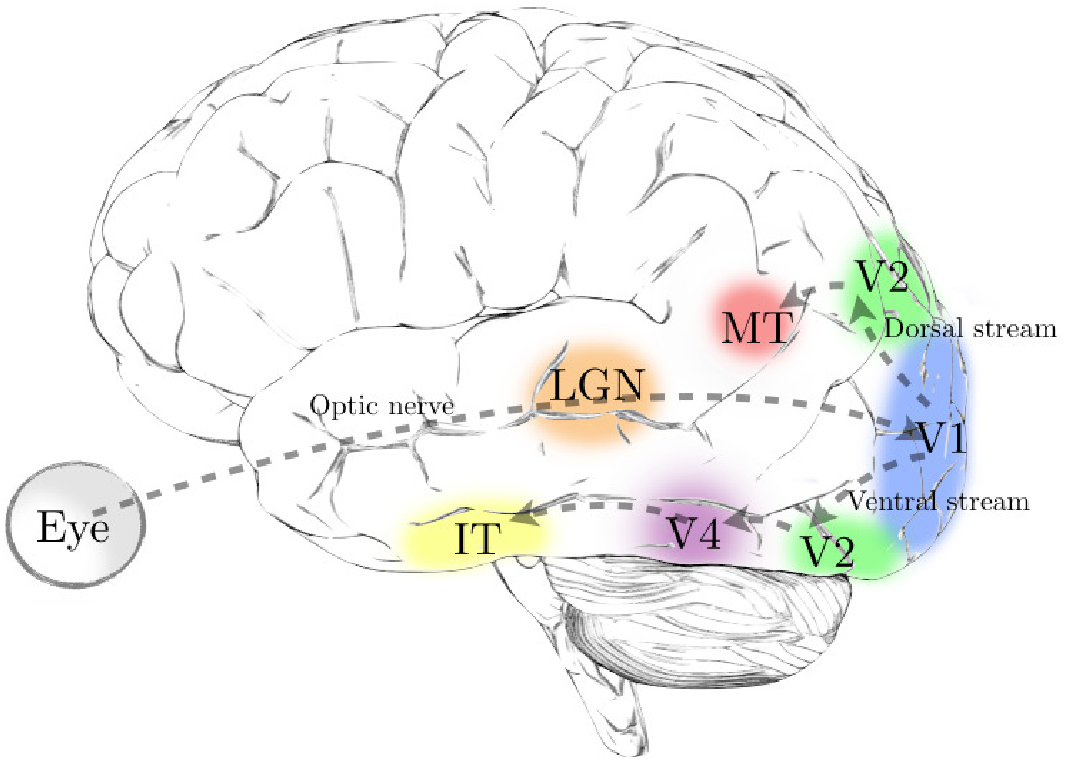

Figure 2.3: Overview over the human cortical visual system: Rough structure of the visual pathway, where information is passed from the eye through the LGN to the primary visual area, from which it is then passed to higher visual areas.

2.3 Hierarchical architecture

According to [37], the organization of the mammalian visual system can be explained in a simplified way by feedforward information passing: Visual signals and information, coming directly from the retina is sent through the optic nerve to the lacerate geniculate body. From here, the signals are sent in a feedforward fashion to the different visual cortical layers.

Other scientists are convinced that information is exchanged in between the visual areas not only in a feedforward, but also in a feedback fashion. These theories are evidenced in biological experiments by Lamme et al [39],[41],[42],[79], by Niell et al [57], and on an experiment of Tai Sing Lee et al [43],[44],[45]. This last one will be shortly summarized now to briefly explain how the bottom-up - top-down information passing process in the visual system is thought to be organized.

Lee et al wanted to observe the hierarchy of the visual cortical areas in a monkey's brain in this experiment. The basic idea was to observe the neural reaction when showing various images, where the monkeys should react on different kinds of square contours.

Figure 2.4 shows the major images used for this experiment: The monkey was shown the image of the four black disks (on the left) and the four partial black disks (on the right) alternately.

This process should create the illusion of a bright square in front of the four black disks when moving to the partial disks.

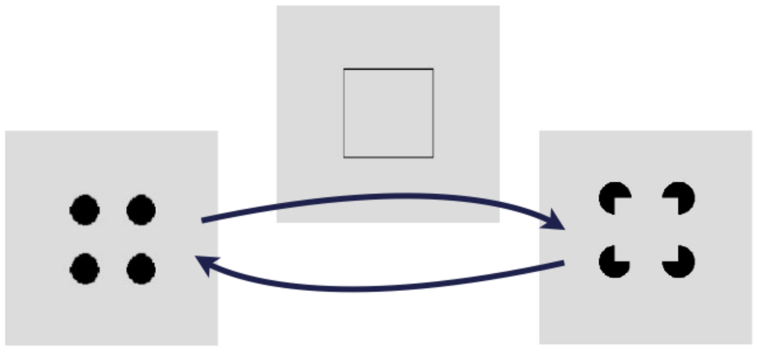

Figure 2.4: Biological experiment to observe the hierarchy of a monkey's visual cortex: Changing images of four black disks (left) and four partial black disks (right) shown, in order to create the illusion of a square (center image) in front of the four black disks. The response when showing the illusory contour image was compared to the response to the square with the line contour. Figure adapted from Lee [43].

Therefore, the neurons in the monkey's brain tuned to the corresponding orientation of bars were expected to react in a similar way to the real line contour of the square (the image in the middle of figure 2.4), and to the illusory square, assuming that some kind of contour "completion" is done in the case of the illusory contour. In order to be able to explore this assumption that neurons in the visual areas react on both of these squares - the real and the illusory square - the scientists measured the activity of neurons in the cortical visual areas V1 and V2 of the monkeys using electrodes.

Neural response was measured in terms of the spiking rate. In this experiment a group of neurons in both visual areas was chosen, which responded to the real line contour and had its receptive field on the part of the line contour where in the illusory contour image nothing is visible, i.e., on the top center line of the "square". The average firing rate of these neurons was computed and tracked over time, beginning with the stimulus onset, which is the moment when the image is shown to the monkey.

Again according to [44], neurons in the first visual area V1 are tuned to edge detection, but there was no evidence for edge completion in this cortical area. On the other hand edge completion does take place in the second - higher - visual area V2.

Having this background knowledge one will expect both cortical areas to respond with a relatively high firing rate on the contrast contour, which shows a real black square on a light background. As a second consequence of this known functions of the different cortical visual areas it could be expected that the illusory contour is only recognized in V2.

The tracked average firing rates of the neurons in both layers are visible in figure 2.5. Here it becomes visible that this second assumption is not true: There is a response also to the illusory contour in the first cortical visual area, but it is delayed when comparing it to the response to the real contrast contour in the same visual area V1.

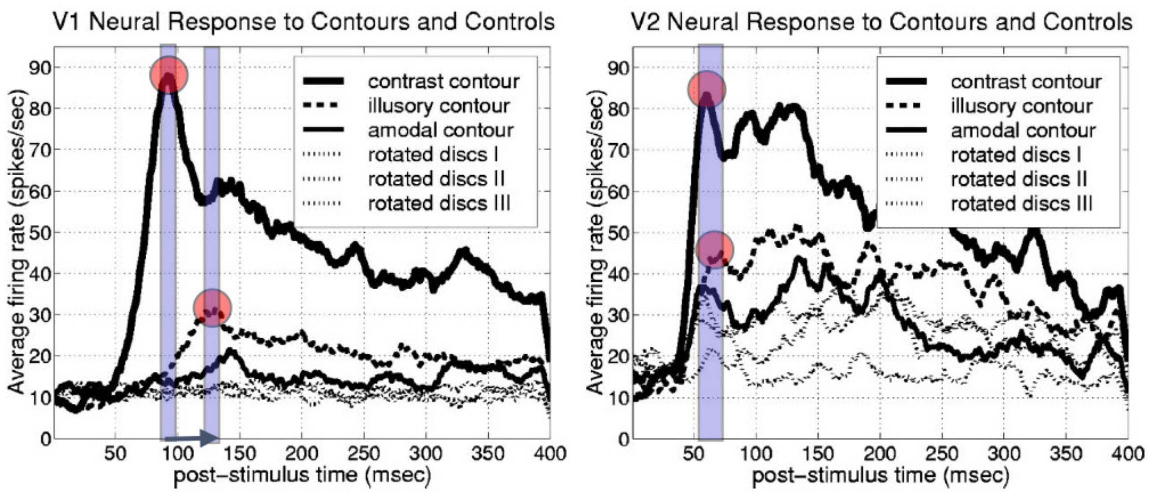

Figure 2.5: Population averaged responses of V1 (left) and V2 (right) neurons to the various square contours shown to the monkey: Delayed response to the illusory contour in V1, but simultaneous responses to both contours in V2. Figure adapted from Lee [43].

When comparing this response to the firing rates in the second visual area V2, one observes that the time between the stimulus onset and the response onset is about the same for both, the real square and the illusory one, as expected. According to Lee et al. this means that the contrast and the illusory contour are both recognized in cortical visual area V2, with roughly the same delay.

According to Lee et al this delayed response in V1 can be explained by the fact that illusory contour completion is done in V2, but there is some top-down (not only bottom-up) information passing between the two cortical layers. In this way, delayed feedback is returned to the first visual layer, where the delay can partly be explained by the synaptic delay when passing information between different areas in the brain.

Note that the reason why the response time to the real square (i.e., the contrast contour) is lower in V2 than in V1 is not covered in the mentioned paper by Lee et al and will thus not be investigated further. The following work will concentrate on the difference between the response time to real and illusory contours in the two visual areas.

As mentioned before, such biological experimental findings suggest that there are feedforward-feedback loops in the visual cortical areas and that information is not only passed in a bottom-up fashion through the layers, but there are also top-down connections that enable the transmission of information in both directions, and not only from the lower layer to the top.

The idea and outcome of the paper by Lee and Mumford represents the basis of this work, where - similar as done by Shi et al [71] using importance sampling and by others - such a multilayer behavior of neurons will be modeled, and then applied to a strongly simplified model of the visual system.

3 Artificial Neural Networks

3.1 Idea and principle

Since decades people try to understand and reconstruct the processes that are going on in our brains - while we are learning new things, reconstructing them, and even while we are sleeping. Already in the 19th century the idea came up, that human thoughts and behavior was a result of interacting neurons in the brain [4],[38].

Some years later, a computational model called threshold logic was invented by McCullouch and Pitts (see section 3.2). This first model for neural networks was based on mathematics and algorithms. It was the basis for future development in this field. So, a very simple unsupervised learning strategy, known as Hebbian learning, variants of which are still used, was invented [31].

The aim to construct an artificial agent that acts like a human was already formulated very early, and still represent the major goal in artificial intelligence. In 1950 Alan Turing proposed the legendary Turing test to determine whether a machine is able to behave (and think) like a human being does [76].

But when we talk about artificial neural networks [67],[3] we do not intend to simulate entire humans, but rather to simulate or artificially reconstruct small subsets of the processes going on in a human brain: The goals when dealing with artificial neural networks are to better understand biological networks, or to solve problems from the field of artificial intelligence. Neuroscientists work on algorithms which describe learning and recognition processes that can be observed in experiments with animals like monkeys or rats [11], [12], [22].

These simulations are realized through the implementation of networks of (partially) connected neuron-like units [32],[67], deciding stochastically or deterministically (depending on the type of artificial neural network) which state to get into. Depending on the type of units used, such a state might either be a binary value - indicating whether a unit is active or inactive - or some value in a certain range (e.g., sigmoidal activation function). In this way, the units reflect the current network situation, e.g., they react on the input given to the network. An artificial neuron can therefore be described as a programming construct, miming some of the properties of biological neurons [35].

Artificial neural networks consist of neuron-like units, each of which at any point in time can take on a state, belonging to a specific predefined set of states, depending on the type of unit. As we know from human beings and animals, the processes in a brain are usually not deterministic, and therefore these units or neurons in many kinds of artificial neural networks, have some stochasticity in their activity.

As shown in figure 3.1, in between these neuron-like units, there might be some connections, which represent the synapses in real brains, through which the single neurons are able to "communicate" their current states to the other units in the network. These connections can be (positive or negative) weighted (indicating excitatory and inhibitory neurons). In certain networks the connections are only existing in feedforward fashion, while other networks have additional feedback connections.

The different areas in a brain, like for example the visual areas described in section 2.2, are represented in artificial neural networks by layers. A network - depending on its type and purpose - might consist of an input layer, an output layer and one or more hidden layers in between.

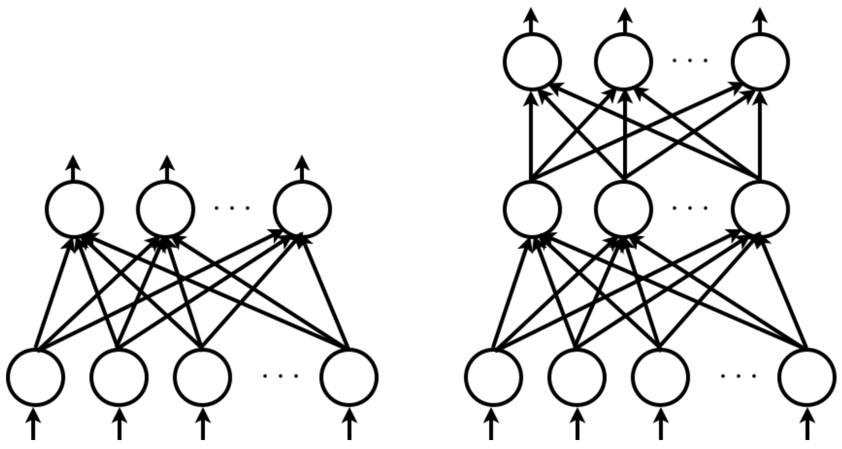
Figure 3.1: Left: Simple perceptron, with only one layer, the output layer (the input layer is generally not counted); Right: two-layer-perceptron, having in addition to input and output layer a hidden layer in between. In both sketched cases there are connections between, but not within the layers.

Each layer consists of a number of units or neurons, and the various layers are connected through synapses. Usually there are also some connections between the units within each layer.

An early pattern recognition algorithm from 1958, called perceptron, was already based on a network consisting of two layers, which should learn [66]. From this time on, the evolution of new algorithms and functionalities went on much faster. In the 70s a first multilayer-network was designed, including some training algorithm [26], and some years after that the backpropagation algorithm was rediscovered and applied to the field of artificial neural networks.

Today there exist very different approaches to artificially simulate these processes going on inside and in between the neurons of a brain, as it is described, e.g., in the following sections 3.2, 3.3 and 3.4, all based on the same general idea of how an artificial neural network looks like [32]. Some very simple models of perceptrons are the ones shown in figure 3.1.

3.2 Forward Perceptron models: McCulloch-Pitts neurons

In 1943 McCulloch and Pitts defined the first and simplest model of neural structures, called McCulloch-Pitts neurons [47]. Using this model they wanted to investigate whether all functions which are defined as Turing-computable can really be computed by a brain.

McCulloch-Pitts neurons are binary units [32], getting a certain number of excitatory input signals $x_1, \ldots, x_n$, but allowing also for some inhibitory input signals $y_1, \ldots, y_m$. The binary output of the neuron is then computed deterministically, depending on the (possibly weighted) input signals and some threshold value which is defined for each unit, as it can be seen in figure 3.2.

If the neuron gets some inhibitory input signals, and at least one of these signals has value 1, then the output of the neuron is 0, independent on the threshold value. This means that one single inhibitory input signal is able to disable the unit.

Otherwise, i.e., if all inhibitory inputs are zero, the excitatory input signals are summed up and compared to the threshold value: If the sum is greater or equal to the threshold, then the unit switches to 1, otherwise it gets value 0.

The inventors proved that when assembling such computational units in a synchronous way, the so created network is able to perform any ordinary digital computer computation, even though it might be slow or inconvenient [32]. Such a directed graph of McCulloch-Pitts gates is also called McCulloch-Pitts network. The gates themselves are still used today in electrotechnics as so-called threshold gates.

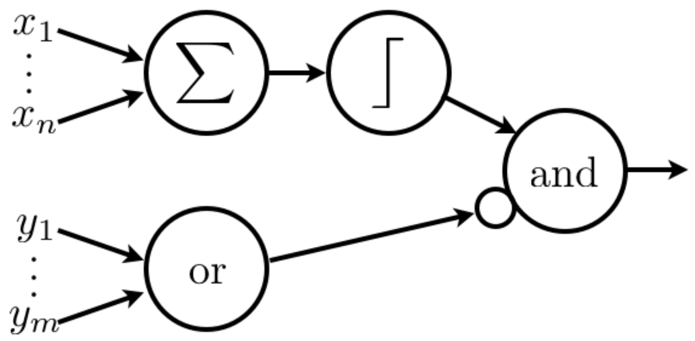

Figure 3.2: A McCulloch-Pitt neuron: Inputs are processed deterministically to compute a binary output. A single inhibitory input forces the output to be 0. Otherwise the sum of the excitatory inputs $x_1, \dots, x_n$ is compared to a threshold value, and only if this threshold is exceeded, the output is 1.

3.3 Adding recurrence: Hopfield network

A Hopfield network is a special kind of artificial neural network, published in 1982 by John Hopfield [32],[36]. The binary neuron-like units in Hopfield networks can be either active (+1) or inactive (0). They are arranged in one single layer - being the input and the output layer at once - and each of them is connected with every other unit (except itself) through a synapse, as it is shown in figure 3.3.

The states of the neurons in such a network are updated one at a time, in any order. The update step is deterministic: If the sum of inputs from other neurons $z_l, l \neq k$, weighted by the connection weights W_{kl}, and the external input b_k is positive, then this current neuron z_k gets active, otherwise its value becomes 0:

$$z_k^{(t+1)} = \begin{cases} 1 & \text{if } \boldsymbol{z}^{(t)\top} \boldsymbol{W}_k + b_k > 0, \\ 0 & \text{otherwise.} \end{cases} \tag{1}$$

where $\boldsymbol{z}^{(t)}$ denotes the state vector at time t, and $\boldsymbol{W}_k$ is a vector containing the connection weights from all units to the currently updated z_k.

The energy E of the network can be computed as the sum of the products of all connected neurons' values $z_k \cdot z_l$, weighted by the connection strengths W_{kl} and the sum of all neurons' values z_k weighted by their external inputs b_k:

$$E = -\,0.5 \cdot \boldsymbol{z}^\top \boldsymbol{W} \boldsymbol{z} - \boldsymbol{z}^\top \boldsymbol{b} \tag{2}$$

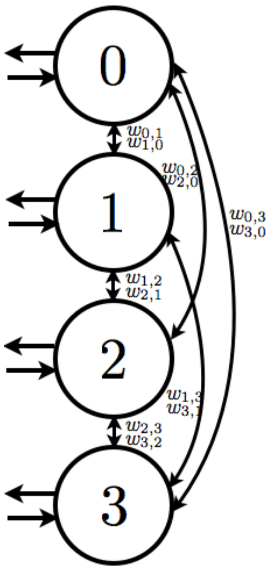

Figure 3.3: Structure of a Hopfield network

A Hopfield network has a special synaptic connection pattern: The neurons have no self-interaction, meaning that $W_{kk} = 0$. Belonging to the class of feedback networks, the synaptic connections are recurrent, i.e., feedforward *and* feedback. Additionally in Hopfield networks the connections weights are symmetric, i.e., $W_{kl} = W_{lk}$ for all neurons k, l.

3.4 Towards stochastic computing: Boltzmann machines

A Boltzmann machine is a stochastic system, describing a special kind of neural network. [33],[34], invented by Geoffrey Hinton and Terry Sejnowski. According to Hinton, the Boltzmann machine can be described as a Hopfield network (Sections 3.3 and 4.2.3) using additional stochastic update rules (Monte Carlo methods, as described in section 4.2.2) for the simulation of non-zero temperatures [36].

The name of Boltzmann machines comes from the structure of its underlying distribution function, a Boltzmann distribution:

$$p(\boldsymbol{z}) = \frac{e^{\frac{1}{2}\cdot\boldsymbol{z}^\top \boldsymbol{W}\boldsymbol{z}+\boldsymbol{z}^\top \boldsymbol{b}}}{\sum_{\boldsymbol{z}'} e^{\frac{1}{2}\cdot\boldsymbol{z}'^\top \boldsymbol{W}\boldsymbol{z}'+\boldsymbol{z}'^\top \boldsymbol{b}}} \tag{3}$$

$$= \frac{e^{-E(\boldsymbol{z})}}{\sum_{\boldsymbol{z}'} e^{-E(\boldsymbol{z}')}} \tag{4}$$

$$= \frac{e^{-E(\boldsymbol{z})}}{Z} \tag{5}$$

where $\boldsymbol{z}$ represents a state, described by the variables $z_1, \ldots, z_K$ taking a certain value each, and $p(\boldsymbol{z})$ denotes the probability of state $\boldsymbol{z}$ to occur.. The symmetric weight matrix $\boldsymbol{W}$, that contains the weight values for the connections between any two neurons in the network, in this case has a zero-diagonal, as the bias values b_k of all neurons are specified separately, for a better understanding. E denotes the energy of a state vector $\boldsymbol{z}$, and Z is a normalizing factor, ensuring that the distribution values sum up to 1.

In a Boltzmann machine, the membrane potential of a neuron can be computed easily as the sum of this neuron's bias value b_k and the weighted sum of the activity of all other neurons in this layer $\sum_{z_l \in \boldsymbol{z} \setminus z_k} W_{kl} \cdot z_l$:

$$u_k = b_k + \boldsymbol{z}^\top \boldsymbol{W}_k \tag{6}$$

In this context, each neuron has a probability of being active which is equal to the logistic function:

$$p(z_k = 1 | z_l, l \neq k) = \frac{1}{1 + e^{-u_k}} \tag{7}$$

Hinton himself applied the Boltzmann machine to a variety of problems, especially in the field of experiments with images, image recognition and reconstruction, like optical character recognition and many others [68].

Its popularity is highly related to the simple learning algorithm by the inventors [2]. Even though it is very slow in complex networks consisting of a large number of layers of feature detectors, learning is fast and efficient in so called restricted Boltzmann machines, which have only one single layer of feature detectors: They consist of a single layer of visible units, and a single layer of hidden units, having no connections within the layers, but only from hidden units to visible ones and vice-versa. Therefore, complex networks are often decomposed into small Boltzmann machines - each one with one single layer of feature detectors. These smaller networks are then stacked, such that the feature activations of lower Boltzmann machines are the training data for the next higher network.

The probability distributions from which Boltzmann machines sample are defined through the quadratic weight matrix of the network, $\boldsymbol{W}$, which defines the synaptic weights from each neuron to each other neuron inside the network, and through the bias vector $\boldsymbol{b}$, which contains one entry for each neuron, describing its intrinsic excitability.

Through so-called clamping, i.e., setting some neurons to specific values and not updating them any more, it is possible to compute conditional probabilities using Boltzmann machines, and thus to use Boltzmann machines to apply Bayesian inference: The conditions are clamped, and the remaining units are updated as usually.

Even though the application of Bayesian inference would theoretically be possible, in practice it is unfeasible, as the partition function, more exactly the normalization constant, rises with the state space. Therefore in general for applying inference, some mechanisms to approximate inference are used. To obtain the posterior, samples can be drawn from this kind of network, for example by using Markov Chain Monte Carlo methods like Gibbs sampling (see section 4.2.3), which are able to take advantage from the stochastic dynamics of a Boltzmann machine. This is the basis for Boltzmann machine learning algorithms like the wake-sleep algorithm or contrastive divergence.

Nevertheless, Boltzmann machines are still far away from a network of stochastically spiking neurons: The use symmetric connection weights, have point neurons, rectangular PSPs and a very restricted dynamic due to Gibbs sampling. This last point will tried to be overcome with the Neural Sampling technique described later on in section 5.2.

4 Statistical Background

4.1 Sampling

Sampling is commonly known as the process to make spot checks (i.e., to draw samples), in order to obtain representative facts describing the composition of some set, which is checked [67], [12]. Correspondingly, in statistics it describes the process of drawing samples out of some probability distribution.

For most problems and cases in practice, to make exact inference about a model would require a huge effort or it would even be infeasible to do so [12]. Therefore in these cases sampling techniques can be applied in order to approximate the real distribution [3]. Different methods which are based on such numerical sampling, i.e., approximate inference methods, are called Monte Carlo methods, described in section 4.2.

An application of sampling is the following problem [12]: Suppose someone needs to find the expected value of a specified function $f(\boldsymbol{z})$, with respect to some given distribution $p(\boldsymbol{z})$. Depending on the shapes of the two functions, computing the correct solution analytically (in case of continuous values by integrating over the product, in case of discrete values by summing them up) will be infeasible. Therefore, some approximation to the real solution has to be found. This is where sampling comes in.

The first step in this case is to draw a certain number of independent samples $\boldsymbol{z} = \{z^{(1)}, \ldots, z^{(K)}\}$ from the distribution $p(\boldsymbol{z})$, such that this distribution can be approximated by its samples. Therefore, the expectation of the real function can be computed as the average over the values $f(z^{(k)})$ for $k = 1, \ldots, K$, i.e., as the average sampled function values [22], [12].

Usually, when sampling, one may however face various problems. A common problem in sampling is that successive samples are not independent from each other, and thus not all consecutive samples can be used, increasing the number of required sample steps to approximate the distribution. Another problem relates to the probability distribution itself: If there are regions with very low probabilities, these regions are hardly covered by the sampling process, if too few samples are drawn. Additionally direct sampling is difficult, as the absolute value of the prior is not known, whereas no samples can be drawn from it.

To overcome these problems, various sampling techniques have been developed, like importance sampling to be able to better account for properties of the distribution, or rejection sampling, where samples are drawn from another distribution $q(\boldsymbol{z})$, which, multiplied with some constant M, satisfies $Mq(\boldsymbol{z}) \geq p(\boldsymbol{z})$ for all possible values of $\boldsymbol{z}$, and the samples that do not match the original distribution are rejected [3], [12].

The next section introduces the role of sampling in the field of neural networks, and describes the way how sampling and samples can be understood in this context, while the following sections describe different sampling techniques and introduce some statistical basics required later on.

4.1.1 Sampling from neural networks

In an artificial neural network (see section 3.1), units which are physical entities are associated with random variables. The states of such entities are represented by realizations or states of these random variables, and thus it is possible to speak about a probability distribution in this context [25].

The current state of some units, i.e., a snapshot (or sample) of the network state and activity, can be obtained through sampling [22], [11]. In each sampling step, the network is advanced by some time, e.g., by 1 millisecond, and the new states of the units are observed. Of course, this process requires some mixing time (or "burn-in") for the network to adjust to a new situation, e.g., changing input.

There exist hypotheses that it is possible to draw samples from neural networks [15]: It is assumed that such a network samples from its underlying probability distribution, after having passed the mixing time, i.e., when the network has established after some initial time. If the sampling states of the neuron-like units are tracked over time, then the samples approximate the distribution from which the network samples. Thus it is a usual approach to sample for some time to approximate the distribution of the network in reaction to some input signal [61].

The always limited sampling time in experiments is the reason why the sampled distribution in practice does "only" approximate, and not match perfectly the correct distribution. Nevertheless, the difference between the sampled distribution $q(\boldsymbol{z})$ and the real distribution $p(\boldsymbol{z})$ will decrease over time, which can be seen when looking at some pseudo metric computing the distance, e.g., the Kullback-Leibler divergence [12], which can be computed as $D(p(\boldsymbol{z})||q(\boldsymbol{z})) = KL(p(\boldsymbol{z}), q(\boldsymbol{z})) = \sum_{k=1}^{K} \left(q(z^{(k)}) \cdot \log \frac{q(z^{(k)})}{p(z^{(k)})} \right)$ and goes to zero as the sampling time increases.

In this context one has to pay attention: Due to the limited sampling time it might be possible - or in larger networks even most probable - that some samples will never be drawn, which would lead $p(z^{(k)}) = 0$ for some values of k. These cases have to be treated in a special way not leading to divisions by zero or to the logarithm of zero [9]. This is also the reason why in this case it is more appropriate to compute the divergence from the true to the sampled distribution, and not vice-versa, as the Kullback-Leibler divergence is not a symmetric measure, and its value has to be non-negative, which cannot be guaranteed very easily if the distance from the sampled distribution is computed - due to the possible zero-values.

4.2 Random walks: Markov Chain Monte Carlo methods

4.2.1 Markov chains

A Markov chain or Markov process [13],[8],[12], [3] describes a structured set of numbered random variables $S_0, \ldots, S_N$. They are characterized by the property that it is possible to forecast the future development of the chain having only some restricted knowledge of the past, in the same way as the whole past development would be known. For a first order Markov chain this means that the future of the whole system $S_{t+1}, \ldots, S_N$ only depends on the current state S_t, but not on past states or developments $S_0, \ldots, S_{t-1}$.

$$p(S_{t+1} = s|S_t = s_t) = p(S_{t+1} = s|S_t = s_t, S_{t-1} = s_{t-1}, \ldots, S_0 = s_0) \tag{8}$$

Such chains can be *continuous* or *discrete* in time: While Markov chains might be both, continuous or discrete, the term Markov process describes only continuous-time Markov chains, having a continuous index. The following explanations only refer to discrete time Markov chains with0 a finite state space.

Another property that Markov chains can have, is to be *time-homogeneous*: These kinds of chains have transition probabilities which are independent of the time step. This means, that the probability p of going from state y to state x is the same at any time step t.

$$p(S_{t+1} = x|S_t = y) = p(S_t = x|S_{t-1} = y),\ \forall t \tag{9}$$

Additionally, a Markov chain can have a *stationary distribution* (or invariant measure). Under some conditions there exists a *unique* stationary distribution, to which the distribution of the Markov chain converges. A time-homogeneous Markov chain has a stationary distribution $\boldsymbol{\pi}$, if and only if $\boldsymbol{\pi}$ is a probability distribution and $\pi_{s'} = \sum_s \pi_s \cdot p(S_{t+1} = s'|S_t = s)$, i.e., the value of the distribution for any state s' can be computed as the sum of all incoming path probabilities p, weighted by the distribution value π_s of the state s this path path comes from.

A state is said to be *recurrent* or *persistent*, if the probability that this state occurs infinitely often is exactly one, i.e., if it is sure that the initial state returns again in finitely many steps. In any other case, the state is called *transient* or *irreducible*, as shown in figure 4.1. A Markov chain is recurrent (or transient) if and only if each state of it is recurrent (or transient).

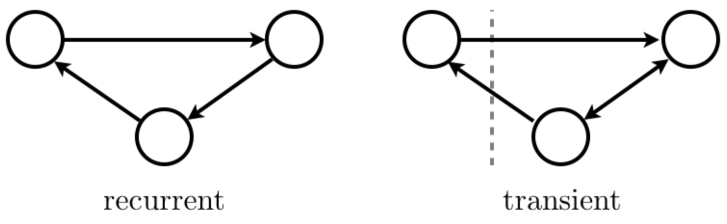

Figure 4.1: Examples of a recurrent (left) and transient Markov chain (right).

A Markov chain is called *irreducible* if each state in the chain is accessible from each other state in finitely many steps, as shown in figure 4.2. As Markov chains can be represented by graphs, irreducibility means that from any state there exists a path to any other state, not necessarily in one step.

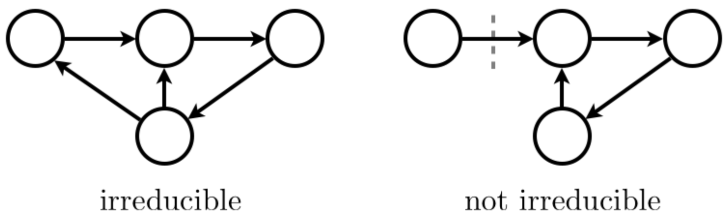

Figure 4.2: Examples of an irreducible (left) Markov chain and a not irreducible one (right).

For Markov chains, if it is possible to return to some state s only in some multiple of k steps (with $k > 1$ and k being the greatest common denominator) then the Markov chain is *periodic*, shown in figure 4.3. Therefore, one can look at all time steps t at which some state is reachable (with a probability greater than zero), and if these numbers have a greatest common divisor greater than one, then there is some periodicity in the system. Otherwise it is *aperiodic*, meaning that there are some irregularities in the patterns of returning to the different states.

$$k = \gcd\{t : p(S_t = s|S_0 = s) > 0\} \tag{10}$$

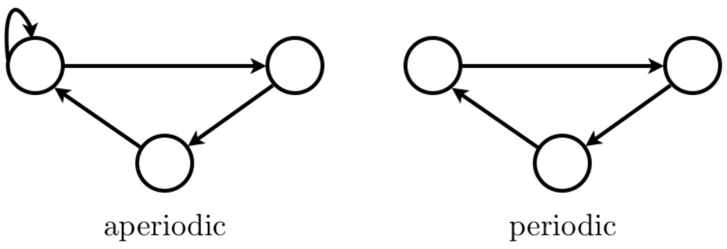

Figure 4.3: Examples of aperiodic (left) and periodic (right) Markov chains.

A Markov chain is called *ergodic*, if it has a finite set of states, it is irreducible and aperiodic. An example is shown in figure 4.4.

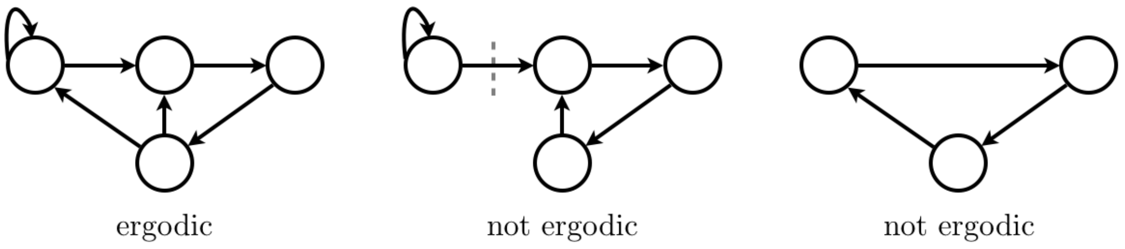

Figure 4.4: Examples of Markov chains which are ergodic (left) or not (center and right).

In order to apply Markov Chain Monte Carlo sampling (see the following sections), a Markov chain has to fulfill some requirements: It has to be irreducible and aperiodic, i.e., ergodic. In this case of an ergodic Markov chain, its underlying distribution will converge to the unique stationary distribution of this chain (the target distribution) [3].

4.2.2 Monte Carlo method

The idea behind Monte Carlo methods is to draw an independent, identically distributed (iid) set of samples out of a large and high-dimensional state space, in order to approximate the density of the target distribution [3], i.e., to explore the major features and properties of a distribution on a high-dimensional state space by performing some random walk on this space [11].

In other words, Monte Carlo methods are based on the idea that the properties of a distribution can be approximated the realizations (samples) of this distribution. In the context of probabilistic inference this means, that any posterior distribution $p(z|y)$ can be approximated by taking random samples of realizations of z from $p(z|y)$. In this way mean, variance, mode and quantiles of the realizations estimate mean, variance, mode and quantiles of the posterior [73].

The advantage is that it is possible to generate samples from a posterior in several ways, even without exact knowledge of the real posterior. *Rejection sampling*, for example, generates independent proposals for z, which are then accepted according to a certain rate, such that the retained samples are proportional to the desired posterior. When using *importance sampling* on the other hand, independent samples (from a user-chosen distribution on z) are weighted in such a way that the properties of the posterior can be estimated. In this way, relevant integrals are evaluated numerically.

Monte Carlo methods are randomized algorithms which with a restricted probability might lead to a wrong solution. In return they are usually much more efficient than deterministic algorithms. When repeating the algorithm using independent random initial states or bits, the error probability can be reduced [73].

4.2.3 Markov Chain Monte Carlo

Markov Chain Monte Carlo (MCMC) methods [73],[3],[12] describe a class of algorithms to draw samples out of probability distributions on the basis of an irreducible, aperiodic Markov chain, which is constructed such that its stationary distribution is the desired one. A sample for the desired distribution is drawn by evolving the chain for some time and by then using its current state as sample.

In general, MCMC algorithms are applied to obtain a sequence of random samples from probability distributions for which direct sampling is difficult (or even impossible), and these methods are used to approximate these distributions.

In many cases where direct sampling is not possible it is feasiblee to construct a Markov chain featuring the desired properties. Usually the most tricky problem is to determine the number of necessary steps until converging to the stationary distribution (with acceptable precision). In other words, the algorithm should be built such that independent system states can be generated effectively. It has to be considered that not only the chain structure itself, but also the initial state, influences the convergence time, and should therefore be chosen appropriately.

The most common examples for MCMC algorithms are the Metropolis (Hastings) algorithm, and Gibbs sampling, explained in the following.

Metropolis algorithm

The original Metropolis algorithm [49] is a local method which generates system states according to a Boltzmann distribution, by generating random moves and accepting or rejecting them with certain probabilities. It is easy to implement, but it has the problem that the generated states of the system are relatively highly auto-correlated. The following explanations always refer to a one-dimensional case (but they can also be generalized to more dimensions).

A new state s_{t+1} is selected by randomly searching some state within a certain distance δ of the previous (i.e., the current) state s_t. With the introduction of the distance, there is a trade-off between acceptance rates and auto-correlation: Small distances lead to high acceptance rate, but on the other hand the auto-correlation in these cases gets higher. On the other hand, when using high distances, the rejection rate increases.

After this new state is chosen, the energy difference ΔE is computed as the difference between the energy of the new state $E(s_{t+1})$ and the previous energy $E(s_t)$:

$$\Delta E = E(s_{t+1}) - E(s_t) \tag{11}$$

The new configuration s_{t+1} is accepted with probability p, which depends on this energy difference ΔE:

$$p(s_{t+1} \text{ is accepted}) = \min\left(1; e^{-\Delta E}\right) \tag{12}$$

This means, if the energy difference is negative, the new state is accepted with probability 1, otherwise with the previously mentioned probability $e^{-\Delta E}$.

Metropolis-Hastings algorithm

The Metropolis-Hastings algorithm [17],[12],[3] is a generalization of the Metropolis algorithm. It allows to draw samples from *arbitrary* probability distributions. After generating a Markov chain, a proposal density depending on the current state is used to generate a new proposed sample. As in the less general Metropolis algorithm, the new proposal state is accepted as next value with some probability depending on the probability of the proposed and the previous state, and on the proposal density, which can, e.g., be represented by some Gaussian, centered at the state whose density is required.

Gibbs sampling

Gibbs sampling [27],[12],[3] is a special case of the Metropolis-Hastings algorithm, which is usually applied if the conditional distributions are possible to be calculated, even though the joint distributions are complicated.

Gibbs sampling is used to approximate the joint distribution of a random vector for which the conditional distribution of each random variable is known [50]. For Gibbs sampling, a Markov chain has to be ergodic (i.e., irreducible and aperiodic) [12].

In each iteration step, only one of the K variables z_k is chosen and according to its distribution, conditioned on current values of all other variables z_l, a new value for this random variable is generated. The values of all other variables remain unchanged in this step.

One problem to be considered when using Gibbs sampling is that, as in each step only one of the variables is updated, successive samples have huge dependencies (at most one value may change). Thus, if independent samples are required, many of the successive samples have to be rejected, which was the main reason for the development of further variations, which update entire sets of variables at a time.

A typical application for Gibbs sampling is sampling from Boltzmann machines (see section 3.4). In this case, the probability of the selected variable is updated according to the current membrane potential u_k (described in formula 6 in section 3.4) of the neuron z_k.

In a Boltzmann machine, the updated probability value p is calculated from the logistic function, which is a sigmoidal function of the membrane potential value $\sigma(u_k)$ which has been updated, as previously shown in formula 7 of section 3.4.

The resulting sequence of sampled vectors is a Markov chain. It can also be proven, that the stationary distribution of this Markov chain is exactly the target distribution of the random vector of formula 22 of section 3.4 which was looked for [12].

4.3 Bayesian inference

4.3.1 Terms and definitions

Bayesian inference is an important statistical concept to deal with uncertainty problems, allowing to estimate statistical parameters, to draw logical conclusions, and to predict probabilities [12],[18],[22],[11],[67].

The so-called prior probability $p(z)$ summarizes the statistical information about the a-priori known evidence of the problem, while the posterior distribution $p(z|y)$ is used to express the predictions about the uncertainty, given some additional information y [10].

Mathematically, Bayes' theorem provides the foundation for computing the probability of an event z to occur given some evidence on y, and it can be expressed as:

$$p(z|y) = \frac{p(z) \cdot p(y|z)}{p(y)} \tag{13}$$

where the posterior probability to be estimated is the first term $p(z|y)$, while $p(z)$ is the prior probability - the probability that event z would occur at all, without taking into account the evidence, which can be interpreted as some initial belief. The factor $p(y|z)$, called likelihood, measures the impact of the evidenced event y on event z. The probability of the evidence, $p(y)$, represents some normalizing factor which ensures that all posterior probability values again sum up to 1.

The term joint probability describes the probability of a set of sub-events to happen "together". An event $\boldsymbol{z}$ can be defined as the combination of one or more observations. In a mathematical formulation this means that having a set of observations $\boldsymbol{z} = \{z^{(0)}, \dots, z^{(K)}\}$, which are assumed to be i.i.d., and where each single observation occurs according to some probability value $p(z^{(k)}|y)$, and the single observations are conditionally independent, given some evidence on another event y, the joint probability of the set $\boldsymbol{z}$ given the evidence can be expressed as:

$$p(\boldsymbol{z}|y) = p(z^{(0)}, \dots, z^{(K)}|y) \tag{14}$$

$$= \prod_{k=0}^{K} p(z^{(k)}|y) \tag{15}$$

4.3.2 Mixture models

A mixture model [22],[11],[12],[21],[48],[28] is a generative probability model, which is able to represent sub-populations inside a pool of data, including also the distribution of the sub-populations [58].

A *generative model* [12],[70] defines, how data is generated by means of auxiliary data. As the joint probability $p(y, z)$ can then be formulated, a generative model provides a mechanism to generate data: By applying Bayes' theorem (as described previously) one can infer the latent variables from this information. Mixture models, presented now, are one type of generative model, which means that inference in mixture models is performed in the same way.

A mixture model divides the entire population in N subpopulations, each of which has an associated weight value $0 \leq w_n \leq 1$, summing up to 1. It further consists of M random variables, representing the observations, i.e., $X_1, \ldots, X_M$, having certain values x_i.

Each of these N sub-populations or categories is described by a probability density function $f_n(x_i)$, which describes the likelihood that observation x_i occurs in subpopulation n. In other words, a sample x_i is drawn out of such a category n, and it occurs with the p.d.f. $f_n(x_i)$.

The mixture model itself is defined by another mixture density function, being the weighted sum over all sub-populations' pdfs:

$$f(x_i) = \sum_{n=1}^{N} w_n \cdot f_n(x_i) \tag{16}$$

4.3.3 Winner-Take-All

When performing inference in a network, the result is such that the current state has been "produced" by one of the output categories, leading to some kind of competition in the output layer. Winner-Take-All is a computational principle about units that compete among each other [29],[78],[60],[14]. Thus, according to the common idea of WTA, only one of these output units - the winning one - is allowed to be active, while all other competing units are suppressed at the same time.

Mathematically, the joint distribution of some observations in the input y and some activity in the WTA units z, namely $p(z, y)$, is defined as a mixture of multinomials, and selecting one winning unit z, given some input y is equal to drawing a sample from some posterior distribution $p(z|y)$ [14].

The major field of application of the WTA strategy as discussed here are recurrent neural networks, where WTA is a computationally efficient and powerful mechanism for competitive learning [46], even though the same strategy is also applied to very different fields, like the plurality voting system where the candidate with the most votes is selected, i.e., wins the competition.

In a neural network sense, the competing units are neurons located in the same layer. Among these neurons there is very strong mutual inhibition, such that at most one of them gets active at a time. Usually such neurons are located in the output layer, causing only the output neuron with the strongest input to be active. An additional requirement to the full inhibitory connectivity in the WTA layer is complete connectivity between all input units and all WTA units.

WTA strategies are often applied to computationally model various processes like making decisions, focussing the attention or drawing conclusions [64],[16].

4.3.4 Expectation Maximization

Expectation maximization is a particular, very popular way to apply maximum likelihood estimation to the problem of parameter estimation for a mixture model (not being limited to this kind of model) with a predefined number of components [20],[67],[21],[12],[22],[3].

The aim of EM is to maximize some likelihood function $p(y|\theta)$ of observed variables y with respect to θ, given the joint distribution $p(y, z|\theta)$, where y are the observed and z are the latent variables.

From the algorithmic point of view, EM is an iterative technique that can be divided into two major distinct steps - expectation and maximization - that are executed repetitively. After having chosen some initial settings for the parameters θ, the loop over the so called E- and M-steps can begin:

In the E-step the expectation is computed, i.e., the current probability $p(z|y, \theta)$ is computed.

The M-step tries to maximize the likelihood with respect to the parameters θ. For this purpose one defines a quality function for the parameters has to be defined, as the weighted sum over all the log-probabilities of possible “new” joint distributions. Very small values of the joint distribution would in this case lead to stronger negative terms in the sum, and would therefore decrease the quality function value:

$$Q(\theta_{new}|\theta) = \sum_z p(z|y, \theta) \cdot \ln p(y, z|\theta_{new}) \qquad (17)$$

The real M-step is then to maximize this quality value by choosing the best parameter setting:

$$\theta_{new} = \arg\max_{\theta'} Q(\theta_{new}|\theta') \qquad (18)$$

As long as the algorithm has not converged, this goes on, by updating the parameter setting $\theta = \theta_{new}$ and going back to the E-step.

For various applications, the original form of expectation maximization is not applicable, like in the case of spiking neural networks. For such cases, variations of the EM algorithm for stochastic online learning have been developed, which link the M-step to spike-timing dependent plasticity (measured biologically) [56]. For more information, refer also to section 5.1 on spike-based EM [55],[30].

5 Spiking neural network models for probabilistic computation

5.1 SEM

5.1.1 The basic SEM model

By applying spike-time dependent plasticity rules in a spiking artificial network of WTA units, it is possible to approximate stochastically an online version of expectation maximization [56],[55],[30]. The way this works is summarized in the following.

Consider an artificial neural network model, that is advanced in discrete time steps [56], as sketched in figure 5.1. The input units $y_1, \dots, y_N$, as well as the units in the output layer $z_1, \dots, z_K$ are binary. The weights of the synaptic connections between the two layers are contained in $\boldsymbol{W}$, where W_{ki} weights the connection from input y_i to the lateral unit z_k. Additionally, the lateral layer units have an associated parameter vector $\boldsymbol{W_0}$, containing one bias value W_{k0} for each unit z_k.

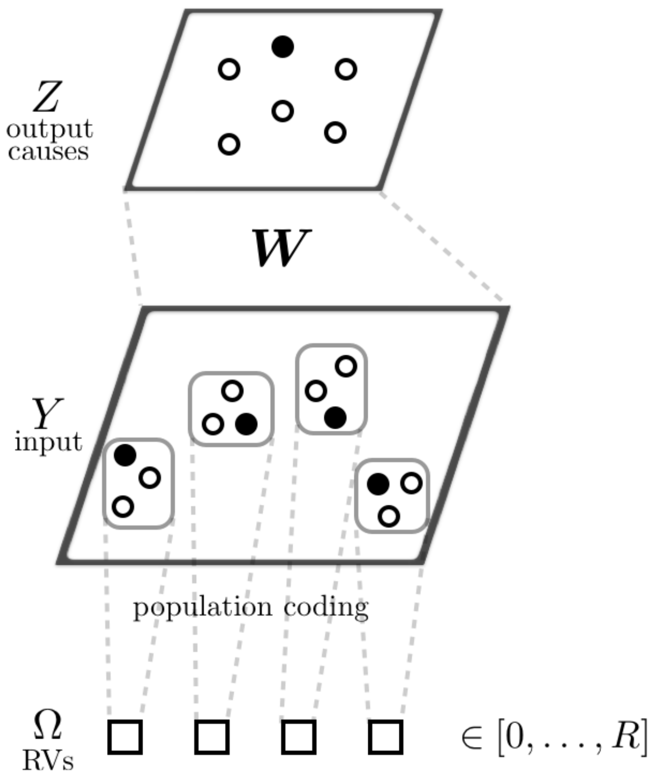

Figure 5.1: Structure of the SEM model: The input units $\boldsymbol{y}$ encode multinomial random variables $\boldsymbol{\omega}$, and project to (hidden) cause units $\boldsymbol{z}$ in a higher layer, which represent the outputs.

The input units again get external input from a set of discrete random variables $\omega_1, \dots, \omega_M$, where each variable at each point in time gets a value in the range $[0; R]$. This means, as all possible inputs have to be covered, that the number of units in the input layer Y has to be equal to $N = M \cdot R$. The population coding of $\boldsymbol{\omega}$ in $\boldsymbol{y}$ is achieved with:

$$\text{if } \omega_m = r \in [0, \dots, R] \tag{19}$$

$$\text{then } y_{R\cdot(m-1)+r} = 1 \tag{20}$$

Hence, there are exactly M active input values in the input layer Y:

$$\sum_{i=1}^{N} y_i = M \tag{21}$$

According to this, a group G_m is defined as the set of variables $y_{R\cdot(m-1)+1}, \dots, y_{R\cdot(m-1)+R}$ in the input layer Y, that encode one concrete input ω_m. Therefore, group G_m is said to define the population coding for the discrete variable ω_m.

As the output layer Z is defined as a WTA layer, exactly one unit z_k is active at each point in time, meaning $\sum_{k=1}^{K} z_k = 1$. This active unit is also called the hidden "cause" for the current input $\boldsymbol{\omega}$. The active unit can be interpreted as a sample, drawn from some distribution, which is proportional to the exponential of the membrane potential u_k of unit z_k:

$$p(z_k = 1|\boldsymbol{y}, \boldsymbol{W}) = \frac{1}{\sum_{l=1}^{K} e^{u_l}} \cdot e^{u_k} \tag{22}$$

where the membrane potential is computed as

$$u_k = \sum_{n=1}^{N} W_{ki} \cdot y_i + W_{k0} \tag{23}$$

$$= \boldsymbol{y}^\top \boldsymbol{W_k} + W_{k0} \tag{24}$$

Learning in such a network can be formulated as the problem of finding a weight vector $\boldsymbol{W}$, which causes $p(\boldsymbol{y}|\boldsymbol{W})$ to approximate the distribution of the $\boldsymbol{y}$ units. Maximizing this is done via EM, but it can be shown (still, see [56]) that applying (not exact) STDP to the same problem implements stochastic online EM learning, where the maximization step is now the application of STDP itself, and the expectation is the firing response of the $\boldsymbol{z}$ neurons.

5.2 Neural Sampling

5.2.1 Theory and idea of neural sampling

In recent publications (see [15],[14],[53]) Büsing et al. are describing a novel idea of a biologically plausible network model consisting of synaptically connected spiking neurons which are able to sample from a large class of different probability distributions. The properties of a distribution of which a neural network should sample has to be reflected by the network dynamics: A network in which such probabilistic inference is possible has to fulfill the so-called neural computability condition, meaning that the membrane potential u_k of each neuron z_k, at each point in time t, equals the log-odds of the probability that the neuron is active:

$$u_k(t) = \log \frac{p(z_k = 1|\boldsymbol{z}_{\backslash k})}{p(z_k = 0|\boldsymbol{z}_{\backslash k})} \tag{25}$$

where $z_k = 1$ indicates that neuron z_k is active, while $z_k = 0$ means it is inactive, and $\boldsymbol{z_{\backslash k}}$ is the set of all neurons in the Z layer, except from z_k.

The Boltzmann machine (section 3.4) is a rather popular set of such probability distributions, implemented by recurrent neural networks, which fulfills the neural computability condition. Commonly, Gibbs sampling (section 4.2.3) is used as soon as the underlying probability model of such a network structure, namely a Boltzmann machine, is searched.

But the usage of Gibbs sampling in combination with a spiking neuron model is not easy to think of. Instead, the idea for probabilistic inference in this case is also based on MCMC sampling (section 4.2.3): In the previously mentioned paper, a spiking neuron model is presented, which is proven to fulfill the properties of a Boltzmann machine and to have the desired underlying stationary distribution, from which it is then also possible to draw samples. The definition of this neuron model is briefly summarized in the following.

Obviously, when using spiking neurons, one has to introduce the definition of a spike probability. Each neuron can emit a spike at any point of time stochastically, depending on its own refractory state ζ_k and the refractory function $g(\zeta_k)$, and on the firing rate function $f_k(u_k)$. The spiking probability can be interpreted as the transition probability $T^{(k)}$ for going from the current refractory state ζ_k to the state where a spike is emitted, namely $\zeta_k = \tau_{ref}$:

$$T^{(k)}_{\tau_{ref},\zeta_k} = f_k(u_k) \cdot g(\zeta_k) \tag{26}$$

This means, that each neuron spikes stochastically, according to the known probability $T^{(k)}_{\tau_{ref},\zeta_k}$, intending that the refractory state of the current neuron is updated stochastically, either to the spiking state τ_{ref} or it is reduced by 1 (until it becomes zero):

$$\zeta_k^{(\text{new})} = \begin{cases} \tau_{ref} & \text{if } \mathit{random} \geq T^k_{\tau_{ref},\zeta_k}, \\ \max\{0; \zeta_k - 1\} & \text{otherwise.} \end{cases} \tag{27}$$

The membrane potential u_k of a neuron can be computed as the sum of some internal value (i.e., the bias value b_k of this neuron z_k), the synaptic input from inside this Boltzmann machine, computed as sum of the other neurons' values $\boldsymbol{z_{\backslash k}}$, weighted according to their synaptic connections to z_k (which is summarized in the vector $\boldsymbol{W}_k$ contained in the weight matrix $\boldsymbol{W}$), and a third factor, representing the external input to this neuron. This external input could for example be a sum of synaptically weighted values of neurons not belonging to this Boltzmann machine. As mentioned above, this kind of membrane potential fulfills the neural computability condition:

$$u_k = b_k + \boldsymbol{z}^\top \boldsymbol{W}_k + \text{external input}_k \tag{28}$$

The firing rate function $f_k(u_k)$ can be computed as the solution of an implicit function, depending again on the refractory mechanism $g_k(\zeta_k)$ and on the membrane potential u_k:

$$f_k(u_k) \cdot \frac{\sum_{i=1}^{\tau_{ref}} \prod_{j=i+1}^{\tau_{ref}} (1 - f_k(u_k) \cdot g_k(\zeta_k = j))}{\prod_{j=1}^{\tau_{ref}} (1 - f_k(u_k) \cdot g_k(\zeta_k = j))} = e^{u_k} \tag{29}$$

5.2.2 More realistic settings

The application of some very specific network settings is clearly described in [14], and it is also proved analytically to be a sampler which draws its samples from the correct analytical distribution. But when further dealing with these proofs and mathematical concepts of this neuron model, a new question might arise.

In this context one might think of a biologically more realistic situation: In biological networks there is always some delay which might be significant especially when transferring signals between neurons in different brain areas. Additionally, the postsynaptic potentials might have some rising and falling time, which is not considered in the previously described theoretically proven setting, and some neurons might also be able to fire before their refractory period is over, i.e., based on some relative refractory mechanism, and not on the absolute one considered in the theoretical model.

What would happen with the sampler and the sampled distributions in combination with settings which are not theoretically proven? How does the distributions of samples change when using more realistic settings in a biological sense? How can such changes be explained? Is worth thinking of such biologically realistic - but analytically not correct - settings, and where are the borders that should not be exceeded in order to keep the sampled distributions still acceptably similar to the correct ones?

In order to be able to better analyze these possibilities and network models, and in order to answer these questions, a new implementation of the neural sampling application was required, which is described briefly in the following section. The analysis of the various settings is then shown in section 7.

5.3 Extension of the SEM model used in this work

In order to apply neural sampling to a practical problem, a specific general model is build, based on the idea of SEMs. Nevertheless, when dealing with sampling, due to the stochasticity it can never be guaranteed that always exactly one unit is active, and that at the same time never more than one unit is active. Additionally the notion of time has to be taken into account when dealing with neural sampling.

Therefore, for the applications of the neural sampler described later on in this work, some extensions to the basic SEM model are introduced, in order to be able to apply neural sampling to SEM-like units of a very specific structure.

5.3.1 Introduction of an additional unit to encode the background hypothesis

Consider a very simple model, consisting of N input units $\boldsymbol{y}$ and K binary output units $\boldsymbol{z}$ as shown in figure 5.2. It is relatively simple to guarantee that not more than one of the output units is active at a time, by introducing (infinitely high) inhibitory weights among them: If one unit z_k is active, it will inhibit all the other ones.

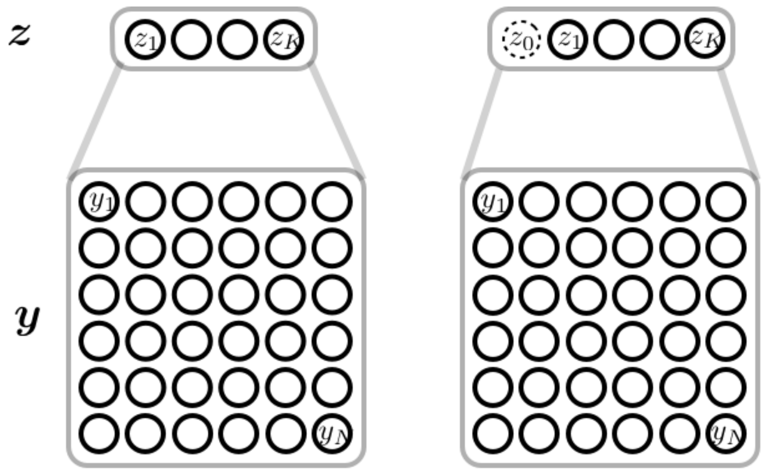

Figure 5.2: Adding a background hypothesis to the SEM unit: The output units $\boldsymbol{z}$ all have the same receptive field on the entire set of input units $\boldsymbol{y}$. The model on the right hand side has been extended by a hypothetical unit z_0, which adds a background hypothesis if none of the other cause neurons is active, in order to fulfill the requirements of SEM. The random variable z_0 is not represented by a neuron.

For plastically modeled, stochastically spiking units it cannot be excluded however that at some points in time none of the outputs is active. Therefore, an additional (non plastic) unit z_0 is introduced. In the following it will be shown that the resulting model represents a Boltzmann machine, which can be used for neural sampling, according to the definition.

First of all, the posterior of this simple model has to be found:

$$p(\boldsymbol{z}|\boldsymbol{y}) = \frac{p(\boldsymbol{z}) \cdot p(\boldsymbol{y}|\boldsymbol{z})}{p(\boldsymbol{y})} \tag{30}$$

$$= \frac{p(\boldsymbol{z}) \cdot p(\boldsymbol{y}|\boldsymbol{z})}{\sum_{\boldsymbol{z}'} p(\boldsymbol{z}') \cdot p(\boldsymbol{y}|\boldsymbol{z}')} \tag{31}$$

Thus, the likelihood of the input units given the outputs should be computed. In this context it should be noted that the input units $\boldsymbol{y}$ are conditionally independent, given the output neurons $\boldsymbol{z}$, which is the reason why the likelihood of the entire input vector can be split up into the likelihoods of the single units:

$$p(\boldsymbol{y}|\boldsymbol{z}) = \prod_{i=1}^{N} p(y_i|\boldsymbol{z}) \tag{32}$$

As mentioned, in SEMs the binary output units $\boldsymbol{z}$ are defined such that exactly one of them is active at a time. The same holds for the extended model (including the non-plastic unit z_0 encoding the background hypothesis). When taking this into consideration, one gets:

$$p(y_i = 1|\boldsymbol{z}) = \prod_{k=0}^{K} p(y_i = 1|z_k = 1)^{z_k} \tag{33}$$

$$= e^{\log \prod_{k=0}^{K} p(y_i=1|z_k=1)^{z_k}} \tag{34}$$

$$= e^{\sum_{k=0}^{K} (z_k \cdot \log p(y_i=1|z_k=1))} \tag{35}$$

Generally, in SEMs: $W_{ki} = \log p(y_i = 1|z_k = 1)$. But for the new extended model, this has to be further explored, introducing new weights V_{ki} for the connections between input and output units:

$$p(y_i = 1|\boldsymbol{z}) = \dots \tag{36}$$

$$= e^{\sum_{k=1}^{K} z_k \cdot \log p(y_i=1|z_k=1) + \left(1 - \sum_{k=1}^{K} z_k\right) \cdot \log p(y_i=1|z_0=1)} \tag{37}$$

$$= e^{\log p(y_i=1|z_0=1)} \cdot e^{\sum_{k=1}^{K} \left(z_k \cdot \log \frac{(y_i=1|z_k=1)}{(y_i=1|z_0=1)}\right)} \tag{38}$$

$$= e^{\log p(y_i=1|z_0=1)} \cdot e^{\sum_{k=1}^{K} (z_k \cdot V_{ki})} \tag{39}$$

Knowing that in this case the entire set of inputs $\boldsymbol{y}$ encodes a single multinomial variable x (of the SEM model), the likelihood can now be written as:

$$p(\boldsymbol{y}|\boldsymbol{z}) = \prod_{i=1}^{N} e^{y_i \cdot \left(\log p(y_i=1|z_0=1) + \sum_{k=1}^{K} \left(z_k . V_{ki}\right)\right)} \tag{40}$$

$$= e^{\sum_{i=1}^{N} (y_i \cdot \log p(y_i=1|z_0=1))} \cdot e^{\sum_{i=1}^{N} \left(y_i \cdot \sum_{k=1}^{K} (z_k \cdot V_{ki})\right)} \tag{41}$$

$$= e^{\sum_{i=1}^{N} (y_i \cdot \log p(y_i=1|z_0=1))} \cdot e^{\boldsymbol{z}^{\top} \cdot \boldsymbol{V} \cdot \boldsymbol{y}} \tag{42}$$

Having now found a way to express the likelihood, the same has to be done for the prior of the output units $\boldsymbol{z}$. Here it has to be ensured that the requirement of a SEM holds, i.e., that exactly one of the outputs is active. This is done by setting the probability of all other combinations of output unit values to 0. Then it is again possible to take advantage from this property when factorizing the prior. Additionally a bias vector $\boldsymbol{b}$ is introduced.

$$p(\boldsymbol{z}) = \delta\left(\sum_{k=0}^{K} z_k, 1\right) \cdot \prod_{k=0}^{K} p(z_k = 1)^{z_k} \tag{43}$$

$$= \delta\left(\sum_{k=0}^{K} z_k, 1\right) \cdot e^{\sum_{k=0}^{K}(z_k \cdot \log p(z_k=1))} \tag{44}$$

$$= \delta\left(\sum_{k=0}^{K} z_k, 1\right) \cdot e^{\sum_{k=1}^{K}(z_k \cdot \log p(z_k=1)) + \left(1-\sum_{k=1}^{K} z_k\right)\cdot \log p(z_0=1)} \tag{45}$$

$$= \delta\left(\sum_{k=0}^{K} z_k, 1\right) \cdot e^{\log p(z_0=1)} \cdot e^{\sum_{k=1}^{K}\left(z_k \cdot \log \frac{p(z_k=1)}{p(z_0=1)}\right)} \tag{46}$$

$$= \delta\left(\sum_{k=0}^{K} z_k, 1\right) \cdot e^{\log p(z_0=1)} \cdot e^{\sum_{k=1}^{K}(z_k \cdot b_k)} \tag{47}$$

Now the so obtained prior is rewritten in terms of a Boltzmann machine:

The first term in the prior ensures that exactly one of the $\boldsymbol{z}$ units is active. The same effect can also be obtained by multiplying a matrix $\boldsymbol{W}$ on both sides with the output units, i.e., $\boldsymbol{z}^\top \boldsymbol{W} \boldsymbol{z}$, if the matrix has a special form, namely a 0 diagonal, and (infinitely) inhibitory entries else. It does not influence the result when an additional constant factor $\frac{1}{2}$ is multiplied with this term. One should note that if the δ is substituted with this new term, this term does not guarantee any more that exactly one state is active, as the z_0 unit is not included into this product. But it has to be mentioned that due to the last term in line 45, this is ensured nevertheless.

The second term of the prior does not depend on the output vector, and can thus be seen as some constant normalizing factor $\frac{1}{Z} = e^{\log p(z_0=1)}$.

In this way one gets a Boltzmann machine prior:

$$p(\boldsymbol{z}) = \frac{1}{Z} \cdot e^{\frac{1}{2}\cdot \boldsymbol{z}^\top \boldsymbol{W} \boldsymbol{z} + \boldsymbol{z}^\top \boldsymbol{b}} \tag{48}$$

Now the prior and the likelihood can be taken to rewrite the posterior:

$$p(\boldsymbol{z}|\boldsymbol{y}) = \frac{p(\boldsymbol{z}) \cdot p(\boldsymbol{y}|\boldsymbol{z})}{\sum_{\boldsymbol{z}'} p(\boldsymbol{z}') \cdot p(\boldsymbol{y}|\boldsymbol{z}')} \tag{49}$$

$$= \frac{\frac{1}{Z} \cdot e^{\frac{1}{2}\cdot \boldsymbol{z}^\top \boldsymbol{W} \boldsymbol{z} + \boldsymbol{z}^\top \boldsymbol{b}} \cdot e^{\sum_{i=1}^{N}(y_i \cdot \log p(y_i=1|z_0=1))} \cdot e^{\boldsymbol{z}^\top \boldsymbol{V} \boldsymbol{y}}}{\sum_{\boldsymbol{z}'} p(\boldsymbol{z}') \cdot p(\boldsymbol{y}|\boldsymbol{z}')} \tag{50}$$

All the terms that do not depend on $\boldsymbol{z}$ can be written in terms of a constant, which results in the posterior of a Boltzmann machine:

$$p(\boldsymbol{z}|\boldsymbol{y}) = \frac{1}{Z'} \cdot e^{\frac{1}{2}\cdot \boldsymbol{z}^\top \boldsymbol{W} \boldsymbol{z} + \boldsymbol{z}^\top (\boldsymbol{b} + \boldsymbol{V} \boldsymbol{y})} \tag{51}$$

5.3.2 A model consisting of multiple extended SEMs

As now the idea of how to overcome the situation where no output unit inside a SEM would spike has been explained, in the following it will be explained how multiples of such extended SEMs can be used in parallel, as it is shown in the simple sketch of figure 5.3.

In this further extension, multiple such models, which have been shown to form a Boltzmann machine, are put next to each other, by adding lateral connections between the output units of different extended SEMs. Some additional hidden units $\boldsymbol{x}$ are introduced, and connected to the output units $\boldsymbol{z}$.

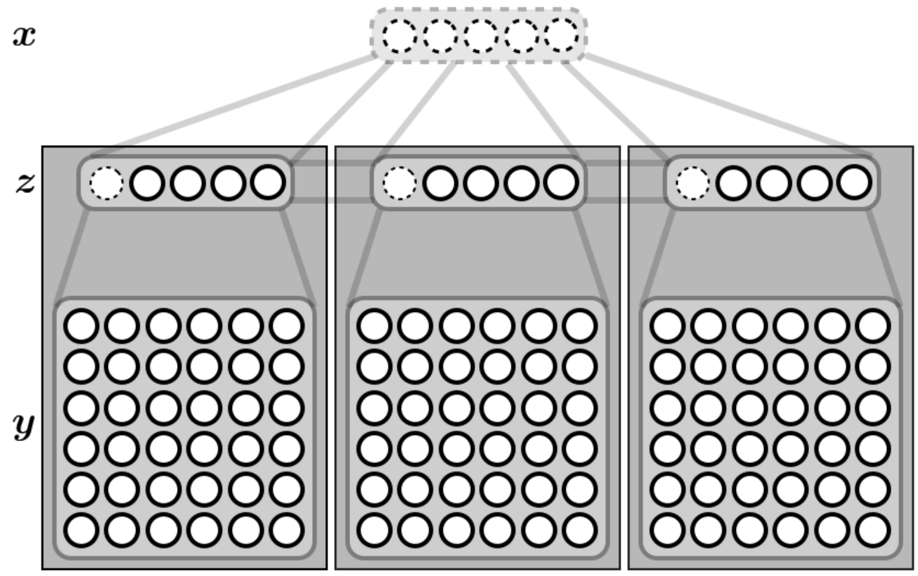

Figure 5.3: Multiple of the previously explained models are used in parallel, introducing lateral layer connections between the output units $\boldsymbol{z}$ and additional hidden units $\boldsymbol{x}$ connected to the outputs.

The internal structure of the single models is not influenced by this further extension, and the new introduced connections are guaranteed to be symmetric. Thus, this step is allowed due to the definition of a Boltzmann machine, and results in a new, even bigger Boltzmann machine.

5.3.3 Introducing the concept of time in this model

The last concept which is missing in this model such that neural sampling can be applied to it is related to time. Therefore, a spiking input model is defined, which introduces a temporal aspect. In the following again the simple model of figure 5.2 in order to keep a better overview. In the previous section it has been explained how this simple model can be extended to a larger Boltzmann machine, which can also be done afterwards with this model in the same way.

Assume that any output neuron z_k might have "caused" one or more spikes during the previous τ_y milliseconds, which is the PSP time constant of the input units $\boldsymbol{y}$, as it is shown in figure 5.4.

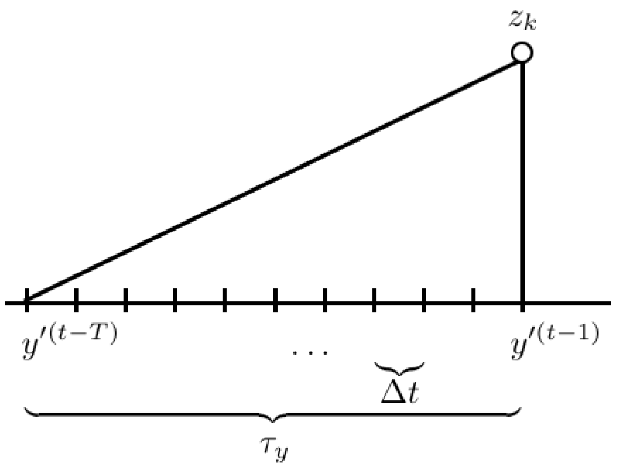

Figure 5.4: Extension of the generative model to spiking input: A cause or output unit z_k aims to explain the recent spikes in the input which occurred in the time interval τ_y. The duration of the time interval matches the PSP time constant of the input units $\boldsymbol{y}$.

The time period τ_y is subdivided into T bins each of length Δt:

$$\tau_y = T \cdot \Delta t \tag{52}$$

This means that if the number of bins R goes to infinity, the bin size Δt will contemporaneously go towards zero, such that it results in a constant PSP time constant τ_y.

As mentioned, an output might have caused several spikes in the input units, as it is shown in figure 5.5. $\boldsymbol{y}'^{(t-S)}$ denotes the input units $\boldsymbol{y}$ at exactly S time steps (or bins) before $z_k = 1$, and the state of the input vector $\boldsymbol{y}^{(t)}$ describes the amount of spikes occurred during the previous τ_y milliseconds:

$$\boldsymbol{y}^{(t)} := \sum_{S=1}^{T} \boldsymbol{y}'^{(t-S)} \tag{53}$$

When now coming back to the model and its probability distributions, an additional missing value has to be taken into account, as it is sketched in figure 5.6.

An active output unit z_k might either have "caused" exactly one spike at a past moment in time $t - S$ with a probability of $\alpha = p(1 \text{ spike in } \boldsymbol{y} | z_k = 1)$, or with probability $1 - \alpha$ it has "caused" some missing value, which stands for the event that no spike has occurred.

Thus the likelihood has now combined with this new probability α, resulting in:

$$p(y_i = 1 | z_k = 1) \mapsto p(y_i^{(t)} = 1 | z_k = 1) \cdot \alpha \tag{54}$$

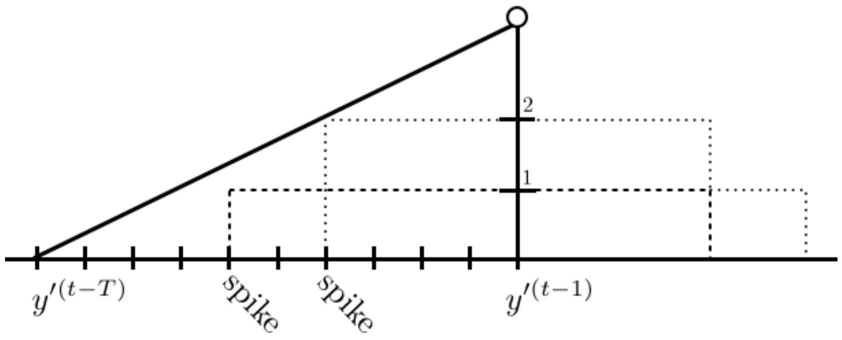

Figure 5.5: Inference in the model with spiking input: Each spike of y' defines the onset of a PSP of time τ_y, which means that the PSP then lasts for the time period τ_y. The number of spikes is counted and expressed as $y_i^{(t)}$.

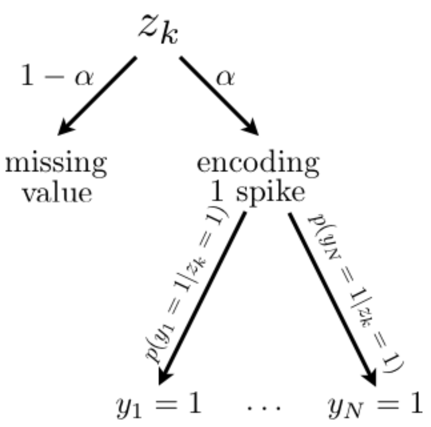

Figure 5.6: Sketch of the likelihood of input unit spikes, given the activity of an output unit z_k. In each time bin the input is only encoded with probability α, otherwise it is missing. This approach results in a sparse input coding and limits the firing rate of the input.

However, this change in the likelihood does not influence the weight matrix:

$$V_{ki} = \log \frac{p(y_i^{(t)} = 1|z_k = 1) \cdot \alpha}{p(y_i^{(t)} = 1|z_0 = 1) \cdot \alpha} \tag{55}$$

$$= \log \frac{p(y_i^{(t)} = 1|z_k = 1)}{p(y_i^{(t)} = 1|z_0 = 1)} \tag{56}$$

Nevertheless, the definition of the likelihood itself has to be modified in order to adapt to the new time-dependent situation of the model:

$$\begin{aligned}
p(\boldsymbol{y}^{(t)}|\boldsymbol{z}^{(t)}) &= \prod_{S=1}^{T}\prod_{i=1}^{N} p(y_i'^{(t-S)}|\boldsymbol{z}^{(t)}) && (57)\\
&= e^{\sum_{i=1}^{N}\left(y_i'^{(t)}\cdot\sum_{S=1}^{T}\left(\log p(y_i'^{(t-S)}=1|z_0=1)\right)\right)}\cdot e^{\sum_{i=1}^{N}\left(\left(\sum_{S=1}^{T} y_i'^{(t-S)}\right)\cdot\left(\sum_{k=1}^{K}(z_k\cdot V_{ki})\right)\right)} && (58)\\
&= e^{\sum_{i=1}^{N}\left(y_i'^{(t)}\cdot\sum_{S=1}^{T}\left(\log p(y_i'^{(t-S)}=1|z_0=1)\right)\right)}\cdot e^{\sum_{i=1}^{N}\left(y_i'^{(t)}\cdot\left(\sum_{k=1}^{K}(z_k\cdot V_{ki})\right)\right)} && (59)\\
&= e^{\sum_{i=1}^{N}\left(y_i'^{(t)}\cdot\sum_{S=1}^{T}\left(\log p(y_i'^{(t-S)}=1|z_0=1)\right)\right)}\cdot e^{\boldsymbol{z}^\top\cdot\boldsymbol{V}\boldsymbol{y}} && (60)
\end{aligned}$$

Similar as before, the first term of this product cancels out in the computation of the posterior, as it does not depend on the outputs, and can therefore be interpreted as some normalization value.

Therefore this modification of the model, introducing the concept of time and spikes, does not affect the definition of the posterior, such that the model still is a Boltzmann machine.

Therefore, also this extended model can be used in parallel and with additional hidden units, as described in section 5.3.2, which will be done for the application of the sampler in section 8.

At this point the concept of a firing rate can be introduced. The firing rate measures the number of spikes per time period. Thus, the so-called group firing rate denotes the number of spikes in a group G_j of units, which is encoded in the probability α, occurred per time Δt:

$$\begin{aligned}
\nu_j &= \frac{\text{Number of spikes in group } G_j}{\Delta t} && (61)\\
&= \frac{\alpha}{\Delta t} && (62)
\end{aligned}$$

Based on this, the number of bins can also be defined as:

$$\begin{aligned}
T &= \frac{\tau_y}{\Delta t} && (63)\\
&= \frac{\tau_y\cdot\nu_j}{\alpha} && (64)
\end{aligned}$$

Additionally to the group firing rate, during learning, also the firing rate of the background hypothesis will be of importance, which is defined as the number of spikes “caused” by the background hypothesis unit z_0 per time:

$$\begin{aligned}
\nu_{0i} &= \frac{p(y_i=1|z_0=1)\cdot\alpha}{\Delta t} && (65)\\
&= p(y_i=1|z_0=1)\cdot\nu_j && (66)
\end{aligned}$$

In general one should note that the weight matrix $\boldsymbol{V}$ describes the relation between the likelihood of the inputs given the output neurons z_k for $k\in[1,\dots,K]$ and given the background hypothesis encoded in z_0. This is the same as looking at the firing rates, meaning that

$$\begin{aligned}
V_{ki} &= \log\frac{p(y_i^{(t)}=1|z_k=1)}{p(y_i^{(t)}=1|z_0=1)} && (67)\\
&= \log\frac{\nu_{ki}}{\nu_{0i}} && (68)
\end{aligned}$$

5.3.4 Learning in this expanded model

In the following, the traditional learning rule of SEMs [56][54][55] will be adapted to the new extended SEM structure, such that it can then be used to learn the model when applying neural sampling to it in section 8.

The traditional learning rule for SEMs is defined in [56][54][55] as:

$$\Delta W_{ki} = \eta \cdot z_k \cdot \left(y_i \cdot e^{-W_{ki}} - 1\right) \tag{69}$$

In a first extension of SEMs, an additional non-plastic neuron z_0 encoding the background hypothesis and ensuring that always an output unit is active, has been added to the model. The only effect this has on the learning rule of the weights between inputs and outputs is that now also the log-likelihood of the inputs given the new added neuron has to be considered. This gives:

$$\Delta V_{ki} = \eta \cdot z_k \cdot \left(y_i \cdot e^{-V_{ki} - \log p(y_i=1|z_0=1)} - 1\right) \tag{70}$$

In order to apply neural sampling to a generative model, this has to be defined in terms of time which has been done in the previous extension of the SEM model. When now also taking into account this aspect of time in the learning rule, one obtains:

$$\begin{aligned}
\Delta V_{ki} &= \eta \cdot z_k \cdot \left(\sum_{S=1}^{T} \left(y_i'^{(t-s)} \cdot e^{-V_{ki} - log(p(y_i=1|z_0=1)\cdot\alpha)} - 1\right)\right) && (71)\\
&= \eta \cdot z_k \cdot \left(y_i^{(t)} \cdot e^{-V_{ki} - log(p(y_i=1|z_0=1)\cdot\alpha)} - T\right) && (72)\\
&= \eta \cdot z_k \cdot \left(y_i^{(t)} \cdot e^{-V_{ki} - \log p(y_i=1|z_0=1)} \cdot \frac{1}{\alpha} - \frac{\tau_y \cdot \nu_j}{\alpha}\right) && (73)\\
&= \frac{\eta}{\alpha} \cdot z_k \cdot \left(y_i^{(t)} \cdot e^{-V_{ki} - \log \frac{\nu_{0i}}{\nu_j}} - \tau_y \cdot \nu_j\right) && (74)\\
&= \eta' \cdot z_k \cdot \left(y_i^{(t)} \cdot e^{-V_{ki} - \log \frac{\nu_{0i}}{\nu_j}} - \tau_y \cdot \nu_j\right) && (75)
\end{aligned}$$

This is the learning rule which is then used in section 8, when neural sampling is applied to a simplified model of vision. More details of how exactly the learning rule is then used are given in section 8.2.

Part II
New contributions

6 A highly configurable software framework for neural sampling

6.1 Structure of the implementation

In the following the implementation of the neural dynamics sampler, realized in C++ with additional interfaces for accessing it through Matlab and Python, is described briefly, including the specification of the available functionalities.

A simplified class diagram of the neural dynamics sampler implementation and its interfaces is shown in figure 6.1. The following sections will briefly cover the structure, which is described in more detail in the technical manual of the sampler.

For performance issues, the core of the sampling network simulator is written in C++, using the GSL (Gnu Scientific Library, at `http://www.gnu.org/software/gsl`) and the Eigen library (`http://eigen.tuxfamily.org`). It is written in an object- and component oriented fashion, such that it becomes modular and thus it is easy to exchange, add or modify single functions or objects.

Summarizing, the structure of such a sampling network can be described in the following way: Each sampling network consists of a number of neurons, of whom each is a separate object. Each neuron can thus have a proper firing rate function, characteristic function and refractory mechanism. The absolute refractory mechanism and some relative one are already implemented, but additional ones can simply be added, following the extension guidelines in the technical manual. Each neuron can also have its proper PSP shape specified. Several different types are already pre-implemented, and also here additional ones can be written and used in an easy way.

The whole network is used by the sampler. Each network consists of one or more neurons, and each network has some type of membrane potential associated. Currently only the membrane potential of a Boltzmann machine is implemented. The membrane potential specifies the types of synapses and synaptic weights, and the learning rules.

Note that if someone is interested in the details of how this implementation of neural sampling can be used, modified or extended, one might refer to the technical manual in the appendix.

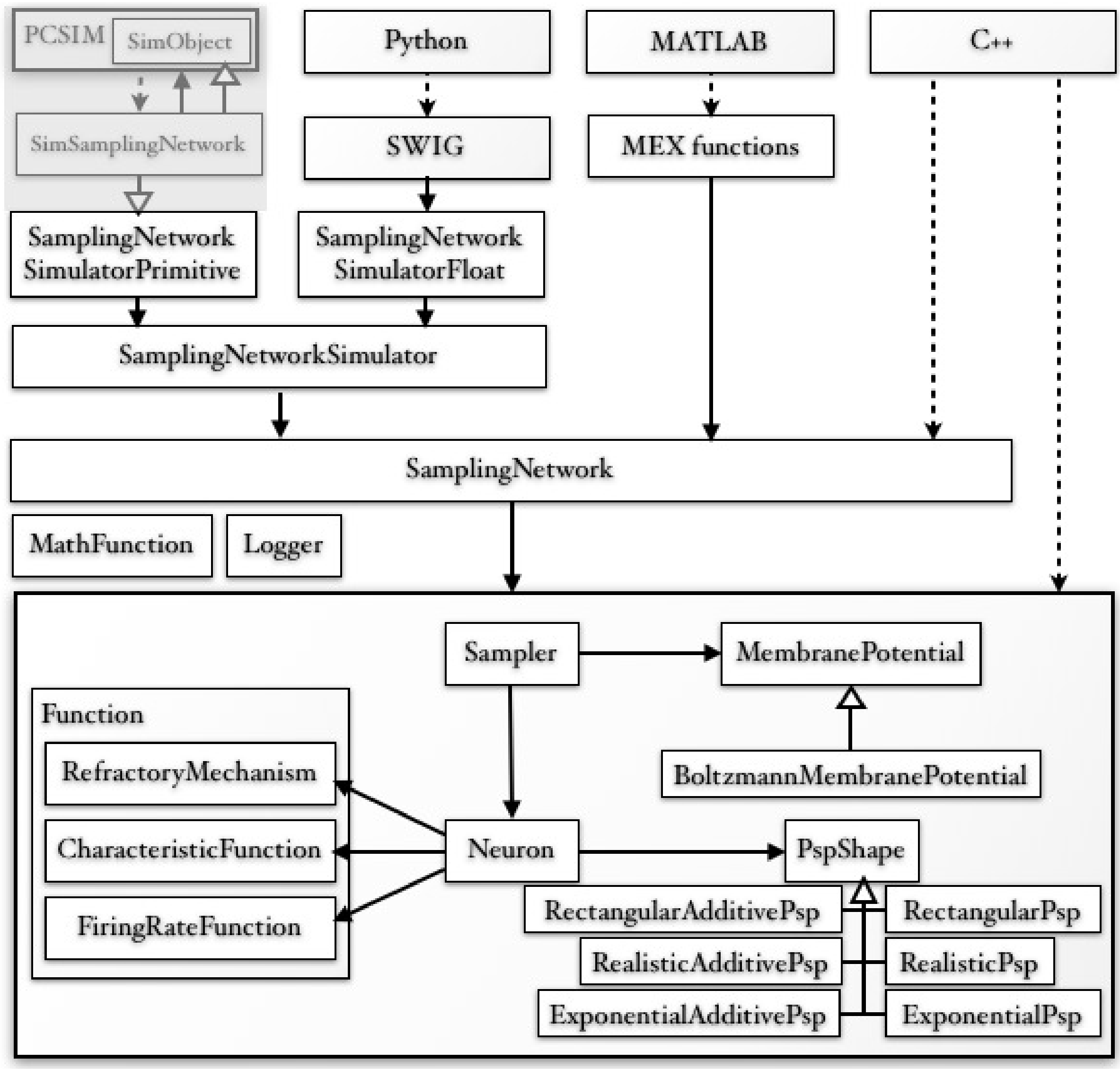

Figure 6.1: Rough class structure and interfaces to access the neural dynamics sampler

6.2 Interfaces

As it is also shown in the class diagram in figure 6.1, there are various possibilities to use the neural dynamics sampler implementation.

When using C++, a developer might use the classes of the neural dynamics sampler, such as `Sampler`, `Neuron`, or `PspShape` directly, having probably the best overview over the processes going on inside the implementation.

If someone wants to avoid dealing with the memory management by his own, i.e., creating and destroying each object that is needed, then one might consider using the `SamplingNetwork` class - or even the `SamplingNetworkSimulator`, which handles additional objects for the user. These classes works as some kind of black box portal to the neural network simulator, such that users can simply call the available methods, which take care of objects and (more or less) complex structures.

For Matlab users, the access to the neural dynamics sampler is provided through the use of MEX functions. When using them, one will not have to care about memory management in detail, but only about the main object, i.e., the neural dynamics sampler itself. Due to the fact that complex C++ objects cannot be ported to Matlab directly, they are kept in C++ internally, and only the address where this main object is located in memory is passed to Matlab. Therefore, this address has to be stored, until the sampling network object is deleted - by calling the destructor explicitly.

Also Python users are able to use the neural dynamics sampler. This interface is realized through the use of SWIG (Simple Wrapper and Interface Generator, at `http://www.swig.org`). The C++ black box class `SamplingNetworkSimulatorFloat` (using no internal other objects as parameters) is wrapped to a Python object, which can be created, used and destroyed through Python. The parameters which in C++ require the library Eigen, are wrapped to Scipy Python data types, such that vector and matrix parameters can be efficiently handled also in Python.

6.3 Evaluation

In order to be able to evaluate the implementation, it was directly compared to the previously existing Python implementation.

For all of the following comparisons and experiments, the same settings were used (if not indicated differently, namely randomly connected Boltzmann machines of 100 neurons, whose probabilities were estimated from 10^5 samples with $\boldsymbol{W} \sim \mathcal{N}(0; 0.15), \boldsymbol{b} \sim \mathcal{N}(-2; 0.25)$, no synaptic delay, a precision of $\delta u = 10^{-4}$ and 10^5 milliseconds of mixing time.

In a first experiment, the implemented sampler was compared to standard Gibbs sampling (in the same network), such that it could be shown that the neural sampler samples from the same distribution (which is the correct one).

The firing pattern of 40 neurons in the same network over 10^4 milliseconds is shown in figure 6.2. The firing rate in this case is at about 7.2Hz. The plot shows the spikes in black, and the refractory period of the neurons in gray. As the absolute refractory mechanism is used, neurons are not able to fire during the refractory period.

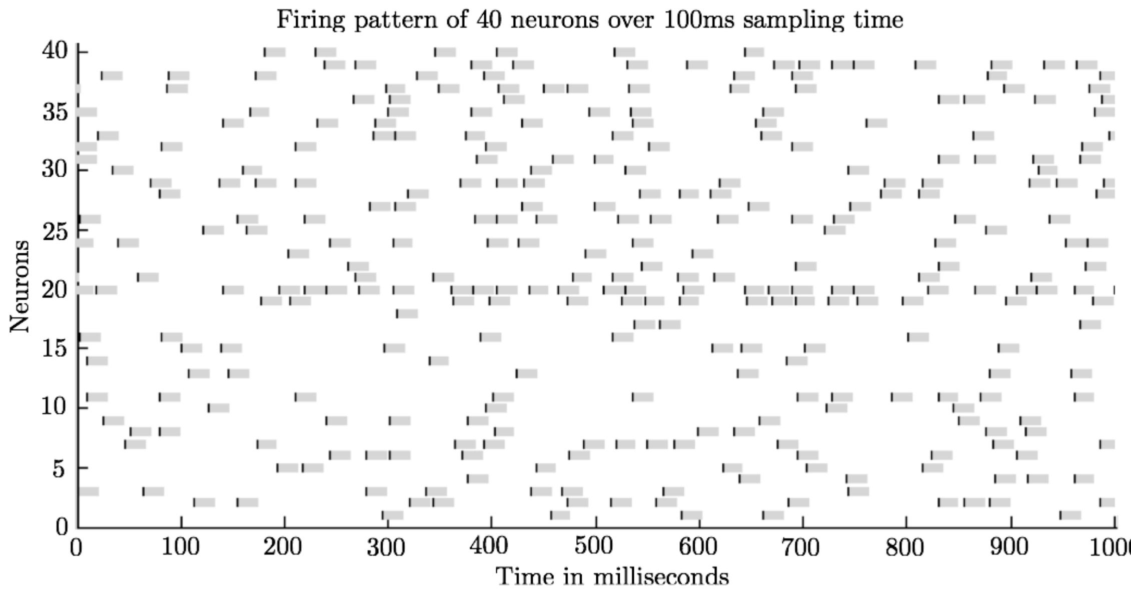

Figure 6.2: Firing rates of 40 neurons with a firing rate of 7.2 Hertz

The execution time was then additionally compared to the previously existing Python implementation: With the same settings the C++ implementation (using the Matlab interface) was about 3 times faster than the Python implementation of the arbitrary refractory mechanism: For initializing and resampling the previously described neuron model for 10^5 milliseconds the existing Python implementation required 480 seconds, while the new C++ implementation through Matlab finished in 150 seconds.

Note that when using the C++ implementation directly (i.e., without interfaces), the execution gets a bit faster, as no conversion is required.

6.4 Functionalities of the neural sampling implementation

In this new implementation, different kinds of settings can be used: The underlying network model can be exchanged by modifying the membrane potential. The refractory mechanism can be selected, as well as the type of PSP shape. And some global synaptic delay can additionally be set for the entire sampling network.

6.4.1 Membrane potential

The underlying neuron model of the sampling network can be modified by setting or changing the membrane potential. The only kind of membrane potential implemented currently is the one of a Boltzmann machine. As described already in section 5.2, a Boltzmann machine fulfills the neural computability condition, and is therefore a valid setting. Due to the object-oriented way of implementation it is quiet easy to add other types which - of course - should also fulfill this necessary condition.

6.4.2 Refractory mechanism

A neurons' state is interpreted as "active" for a certain period - the refractory period - after a spike has been emitted. This means, that after the emission of a spike the neuron's refractory value does not return to its original "inactive" state immediately and the activity of a neuron is measured as $z_k = (\zeta_k > 0)$. In this interpretation, the ζ_k value of a neuron rises to $\zeta_k = \tau_{ref}$ (i.e., the highest possible refractory value) at the moment of a spike of neuron k and it decreases linearly after that until it reaches $\zeta_k = 0$.

The refractory value is therefore some way to measure the time since the last spike. There are two major different classes of refractory mechanisms, whose state is depending on the current refractory value of a neuron: absolute, and relative mechanisms.

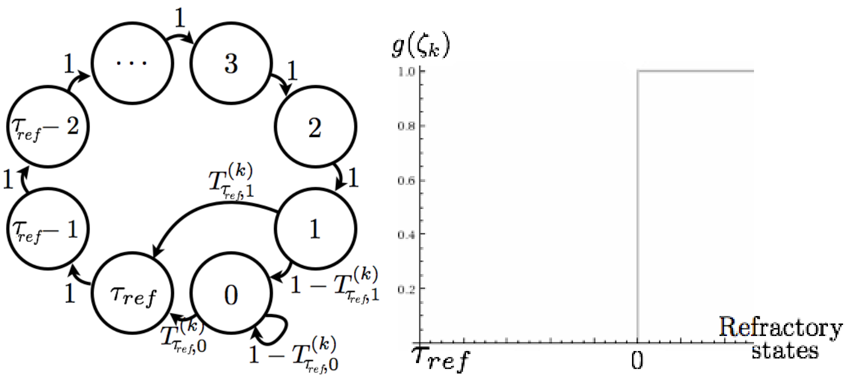

Figure 6.3: Refractory states ζ_k of a neuron z_k, being in the interval $[\tau_{ref}; 0]$, and the corresponding transition probabilities $T^{(k)}$ with an absolute refractory mechanism (left) and shape of an absolute refractory mechanism of decreasing refractory values (right): A neuron is only able to re-emit a new spike if has returned to its "inactive" state $\zeta_k = 0$, or if it would return to the state in the next step at $\zeta_k = 1$.

From these different kinds, only for the absolute refractory mechanism the above properties have been proven to be fulfilled. An absolute refractory mechanism allows a neuron only to emit a new spike if the refractory period (of the previous spike) is over, i.e., the only possible transactions to go to a new spike are at $\zeta_k = 0$ and $\zeta_k = 1$. The sequence of refractory states of one neuron having an absolute refractory mechanism is shown in figure 6.3: The nodes correspond to the different refractory states ζ_k, while the transitions (and their probabilities) are represented by arcs. The right plot in this figure shows the shape of such a refractory mechanism, where a spike is only allowed if the neuron is inactive.

On the other hand, there exists also the concept of relative refractory mechanisms, which might have various shapes, always going from $g(\zeta_k = 0) = 1$ to $g(\zeta_k = \tau_{ref}) = 0$, e.g., a sigmoidal function (see figure 6.4). Such relative refractory mechanisms allow a neuron also to emit a new spike - with a certain probability - if the refractory period of the previous spike is not yet over, i.e., if the neuron is still in its "active" state. A sampler using this relative refractory mechanism has not been proven to sample from the "correct" probability distribution, but it has further been shown in experiments that it is then sampling from very different distributions.

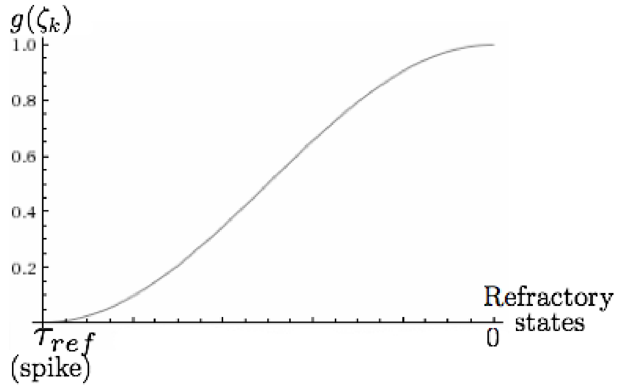

Figure 6.4: Some relative refractory mechanism, where transitions from any state to the spiking state τ_{ref} are possible at any time with some probability $T^{(k)}$, but still the probability of immediately subsequent spikes is low, even though it starts to recover immediately (and not stepwise at the end of the refractory period).

6.4.3 PSP shapes

For a neuron, the concept of a PSP shape can be introduced: Postsynaptic potentials are changes in the membrane potential of a neuron. As soon as a neuron emits a spike, this postsynaptic potential of this neuron rises to a certain rate, and it then can stay active for a certain period, depending also on the refractory period. In this period the potential can stay constant, or it might decay slowly, until it returns to 0 or a very low value after the refractory period is over.

There two major different concepts of PSP shapes: They can either be renewal or additive. The difference between these two is very clear in combination with a relative refractory mechanism, namely in the situation of the emission of a new spike when the neuron is still active, i.e., when the refractory period is not yet over.

As the term already indicates, when using additive PSP shapes, the potential (i.e., the PSP value) of a new spike is added to the current one. On the other hand side, when renewal PSPs are used, the previous spike becomes irrelevant, and the neuron's potential is set to the one of the new spike.

Additionally, there exist different shapes a postsynaptic potential could have. The current implementation of the neural sampler includes three different ones: rectangular, exponential and alpha-shaped ones.

Rectangular PSPs shown in figures 6.5 and 6.6 are the only ones proved to be correct - at least in combination with the absolute refractory mechanism. As it can be seen in these plots, the potential of a neuron rises to some value (in this case to 1) if a spike is emitted, and it stays at this value until it returns to 0, which is the "inactive" state, after the refractory period is over. Therefore, this kind of PSP shape is biologically not very realistic, due to these high (infinite) rising and falling rates.

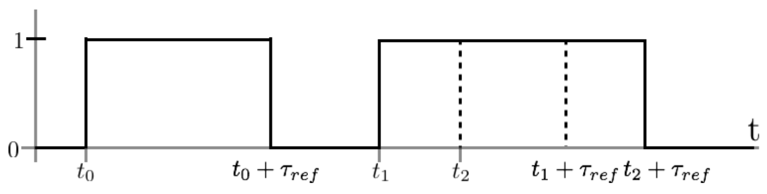

Figure 6.5: Rectangular renewing PSP shapes: In case of a relative refractory mechanism, if the neuron re-emits a spike during its refractory period, this new spike revokes the previous rise in the postsynaptic potential, and the new spike is "counted" instead.

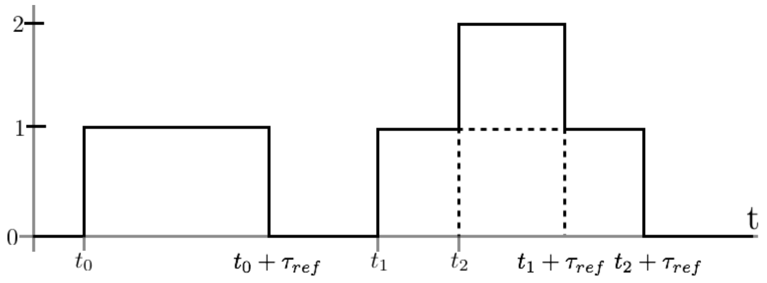

Figure 6.6: Rectangular additive PSP shapes: All subsequent rises in the potential are summed up, i.e., the previous spike is not revoked, but the new increase in the potential is added to it.

When talking about rectangular PSPs with an absolute refractory mechanism there is no difference between renewal and additive PSPs, as in any case a new spike can occur only after the refractory period is over, and in this situation the potential caused by the previous spike has returned to 0 anyways.

Another possibility of such PSP shape types are exponential PSP shapes, shown in figures 6.7 and 6.8. These shapes have again an infinitely high rising rate - which renders them biologically unrealistic - but they then have an exponential falling rate. There are thus two major settings for such shapes - namely the height and the falling rate. Depending on these parameters, such exponential PSP shapes can approximate the true distribution at different precisions.

In various experiments it has been found that the settings of these parameters have a rather strong influence on the difference between the sampled and the correct distributions. Therefore it has also been observed which parameters would be the "best" ones for various network settings in terms of a minimal Kullback-Leibler divergence to the correct distribution.

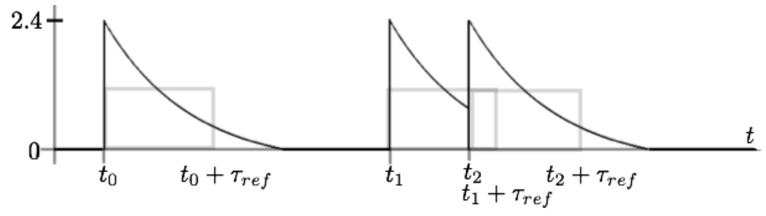

Figure 6.7: Exponential renewing PSP shapes

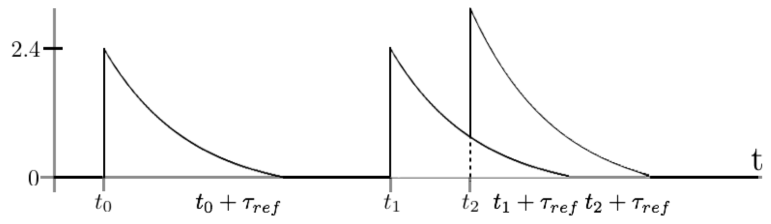

Figure 6.8: Exponential additive PSP shapes

It has been discovered that the optimal parameters for such PSP shapes, leading to the best approximation of the true distribution, lead to an integral under this PSP be only slightly lower than the integral under a rectangular PSP, while the optimal height should be between 2 and 3. This can also be seen in figure 6.7, where the shape (and integral) of a rectangular PSP is sketched in gray. The same is also true for additive exponential PSPs, with the only difference that in this case the integral should be again slightly lower. Note that these experimentally gained theories are based on tests in various network models of two to eight neurons with different synaptic weights and connections, all leading to the same (or very similar) results.

A third class of PSP shapes are alpha-shaped PSPs. The are supposed to be a rather biologically realistic kind of postsynaptic potentials, shown in figures 6.9 and 6.10. Their precision is based on three different parameters, namely the height, the rising rate and the falling rate of a PSP shape.

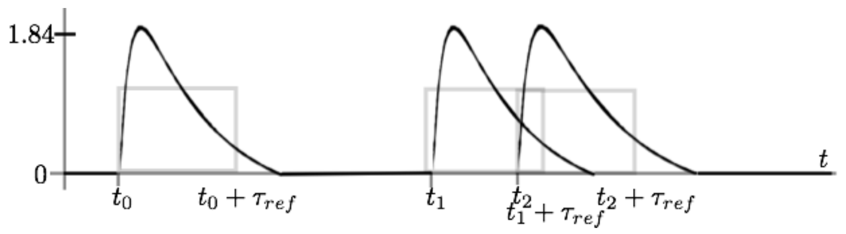

Figure 6.9: Alpha-shaped, renewing PSPs: Difference between two different exponential functions

Again, the settings of these PSP shapes influence the difference between the sampled and the correct probability distributions. And also in this case, various parameters were observed, searching for the "optimal" settings, in terms of the minimal Kullback-Leibler divergence.

In the same way as for exponential PSP shapes, also here the various settings were tested for

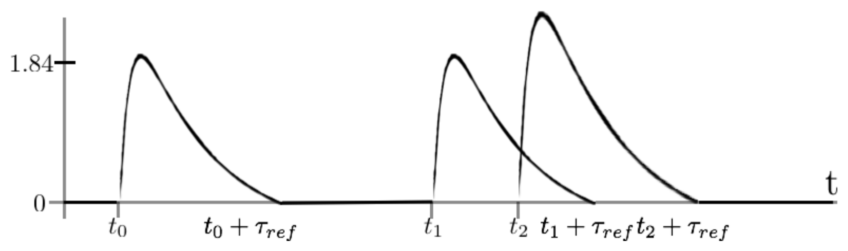

Figure 6.10: Additive alpha-shaped PSPs

different sizes of neuron models, and different strengths of synaptic weights and connections. In this case it was observed, that the integral under such an optimal PSP shape is nearly equal to the integral under a rectangular PSP shape, which can be seen in figure 6.9, where the shape of a rectangular PSP is sketched again in gray. Also here, when using additive PSPs the integral under the optimal PSP shape is slightly lower than the one under rectangular PSPs (and under the optimal renewal PSP shape).

All of these now described kinds of PSP shapes are already included in the current version of the neural sampling implementation, and additional ones could easily be added to it.

6.4.4 Global synaptic delays

It is also possible to set some arbitrary global delay to all synapses in the network model. In this way, the realistic situation can better be simulated, as in biology information cannot be passed through synapses without any (very low) delay.

The concept of such a synaptic delay is shown in figure 6.11. In some situations this delay does not influence the correctness of the information. Such a situation is shown in green in the figure. But in other situations - shown in red - the delayed state of the neuron can be interpreted as "wrong" information that is passed to the other neurons.

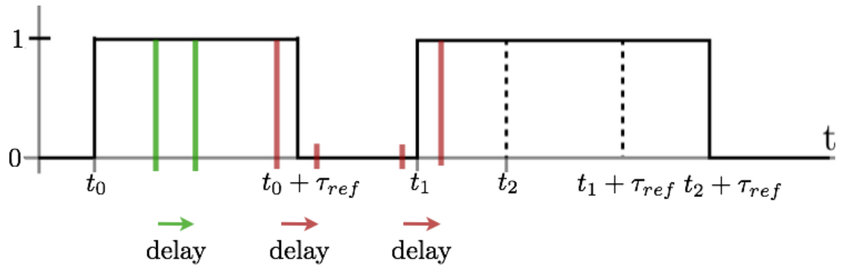

Figure 6.11: Idea of synaptic delay: Other neurons get informed only about delayed (previous) states of a neuron: At the green point in time there is no difference, at the red points these information is not correct any more.

Therefore, such a global synaptic delay can have influences on the probability distribution from which the samples are drawn, but they render a network model much more realistic in biological terms.

6.5 Example of use

In order to get a better idea about how this implementation of the neural sampler works and can be used, the process of sampling the network and tracking the firing rates of all neurons in this time period is shown in a very short and simple example, written in python.

Before the sampling process can start, some minimalistic model of the network itself has to be set up. This is shown in the following, where some parameters have already been specified, like the precision and the global delay of the whole network:

```
net = sampler.SamplingNetworkSimulatorFloat()

dt = 1.0
du = 0.001 # precision: milliseconds
u_min = -15
u_max = +15
delay = 0 # global synaptic delay
net.createSampler(dt,du,u_min,u_max,delay)
```

Note that there are some default parameters for almost every setting, in case if someone is not sure about a possible choice of parameters, but in any way most of them should be chosen specifically to be able to really control the model and keep the overview.

After having set up the network itself, the neurons have to be added to it, specifying the number of neurons to be added at once, the length of the refractory period, as well as the refractory mechanism itself, some possible external input to the different neurons, and the shape (and parameters) of the PSPs:

```
num_neurons = 8
tau_ref = 20.0 # refractory steps (period: relative to dt)
ref_mechanism = 1 # absolute refractory mechanism
ext_input = 0 # external input to the neurons (vector or 0)
psp_type = 0 # rectangular psp shapes
# no additional psp parameters required
net.createNeurons(num_neurons,tau_ref,ref_mechanism,ext_input,psp_type,0,0)
```

Of course there could also be added more neurons in a sequence, having different parameters, but for equal neurons it might be convenient to use the predefined method which adds a certain number of same neurons to the model automatically.

The last important setting is the specification of the weight matrix, and the membrane potential. The following code snipped shows how a Boltzmann machine is set up, for which the weight matrix and the vector of bias values are known explicitely:

```
#Creating Boltzmann machine membrane potential
W = scipy.zeros((num_neurons,num_neurons), dtype=float) #weight matrix
b = scipy.ones((num_neurons,1), dtype=float) * -1.0 #bias values
self.net.createBoltzmannMembranePotential(W,b)
```

The last step before the sampling network can really be used is to set up the sampler. This command finalizes some internal variables, and checks whether the model settings are compatible among themselves, e.g., whether the weight matrix contains exactly one weight vector of the correct length for each of the neurons in the network. This is done, by calling the following command:

```
self.net.prepareSampler(0)
```

After this, the model is ready to be used for sampling. It is possible to track various internal variables that might be of interest. In the following snippet, the network is advanced for 1000 milliseconds, and the firing patterns of all neurons in the network are tracked, and returned afterwards:

```
net.storeFiringRates(True)
net.advance(1000)
firing_rates = net.getFiringRates()
```

For more detailed information on all of these methods and on the available parameter settings, as well as on the other functions and methods that are provided by the neural sampling implementation, please refer to the technical manual.

7 Neural sampling with more realistic neuron models

7.1 Limits of theory

When thinking of a more realistic neuron model with varying, non-rectangular, hill- or alpha-shaped PSPs, and relative refractory mechanisms, in combination with synaptic delays and limited sampling time, one reaches the limits of theory and proofs [15]. In biology, PSP shapes are assumed to be affected by some rising and falling time which is not possible at infinite speed as is it would be required for rectangular PSPs.

It could thus be stated that the rectangular PSP shapes, which have been proven to sample from the correct distribution in [15], are not very plausible in a biological sense. It has rather been found these shapes of postsynaptic potentials are alpha- or even hill-shaped, depending among others on the exact position where the potential is measured [77].

It is also known that the transmission of signals or potentials through synapses might take some time. These delays vary from a few (one to two) milliseconds up to higher values when signals have to be transferred between different areas of the brain [65].

Some of these aspects are covered in the following sections, dealing with such biologically more realistic but not analytically exact sampling neuron models. The parameters and factors considered are various PSP shapes and refractory mechanisms, as well as global synaptic delays and realistic situations where only limited samples can be acquired.

In section 7.2 the influences of both, PSP shapes and also refractory mechanisms, are observed in an network model of eight neurons, also including combinations of these settings. These parameters and the various combinations are observed and analyzed there.

Section 7.3 includes the analysis of the influence of global synaptic delays on the sampled distributions in a network. These influences have been observed in various networks of different sizes and with different connections (inhibitory and excitatory), while this section shows the effects on a tiny two-neuron network in order to make the consequences of delays better visible.

Finally, section 7.5 contains the observation of an even more realistic network model, combining the different biologically realistic factors in addition to a realistic - limited - time for sampling. This experiment should analyze whether in some situations it might be reasonable to use non-realistic sampler settings, when the sampling time is in any case limited.

Note that the following diagrams and results all cover quiet small network models with low numbers of neurons, in order to keep a clearer overview over the evolution of the situation. An additional problem faced with very large networks is the correspondingly huge amount of different samples: With a finite sampling time so-called zero-states cannot be avoided, i.e., samples that are theoretically possible with a very low probability, but in practice do never occur. This leads to additional problems in computing metrics and measurement values, like the KL-divergence (see section 4.1 for more information about the problematic consequence of zero-states). Additionally, with limited computational resources like memory and time the number of neurons gets problematic for the analytical computations quiet soon, as for them the exponentially increasing amount of samples have all to be considered.

In order to overcome these problems, smaller networks have been chosen, which are models consisting of two neurons where the underlying probability distribution is quiet obvious and thus completely visible and available for comparisons, as well as larger models consisting of eight neurons, still comprehensible, but already more complex and more general.

7.2 PSP shapes and refractory mechanisms

In this section the influence of PSP shapes and refractory mechanisms on the resulting distribution of samples is observed. In order to analyze these influences, a small network model - representing a Boltzmann machine - was set up, consisting of eight symmetrically connected neuron-like units, with a Gaussian-distributed randomly generated weight matrix $\boldsymbol{W} \sim \mathcal{N}(0; 0.2)$ having a zero-diagonal (thus no self-interaction) and a random bias vector $\boldsymbol{b} \sim \mathcal{N}(-1; 0.5)$. The sampling time in all these cases was at 10^7 milliseconds, i.e., 10^4 seconds.

All used PSP shapes are optimized in terms of minimal KL divergence to the analytically correct distribution as described in section 6.4.3. When observing renewal PSP shapes, it was discovered that in any of these experiments they caused the distributions to be more exact than with additive PSPs, but nevertheless in the following only samplers with additive PSPs are shown, as the renewal ones are supposed to be not very biologically realistic.

In these first observations, no synaptic delay is used. Instead this is then done in section 7.3. The refractory period of these neurons is at $\tau_{ref} = 20$ milliseconds. For all of these experiments the firing rates were about 20 to 30 Hz, depending of course on the refractory mechanisms, the PSP shapes and their parameters, and therefore varying slightly throughout the different experiments.

For a first analysis, the refractory mechanism was chosen to be the absolute one, and the different PSP shapes were compared to the analytically computed distribution of samples. In order to be able to better visualize the $2^8 = 256$ samples in figure 7.1, the samples were re-ordered according to the height of their relative frequency $q(\boldsymbol{z})$ using the analytical distribution.

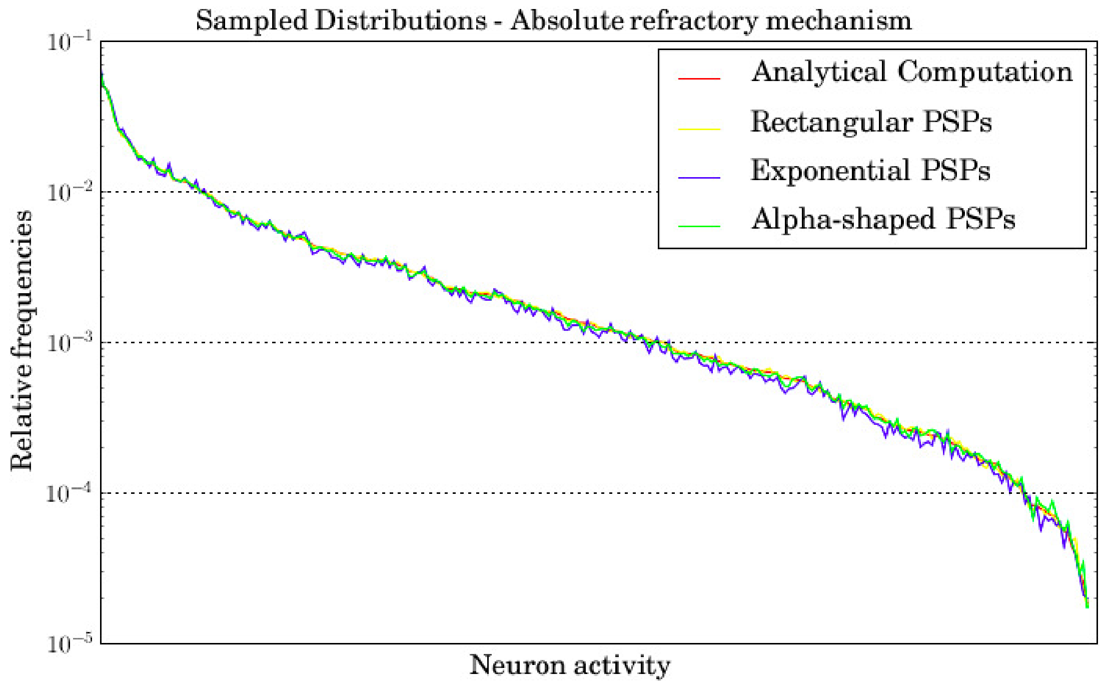

Figure 7.1: Sampled distributions of 8 neurons using various (additive) PSP shapes and the absolute refractory mechanism: Samples sorted according to the height in the correct distribution: Rectangular PSPs lead to correct sampling, exponential PSPs cause quiet incorrect sampling.

In this plot, the X-axis represents these samples - reordered - while on the Y-axis the relative frequencies of the various samples are plotted, in a log-scale. The plot compares the correct and analytical computation to various samplers.

The sampled distributions of the rectangular PSP shapes (using the absolute refractory mechanism) are equal to the analytically computed distribution. The small differences that can still be observed between these two distributions are due to the still limited sampling time.

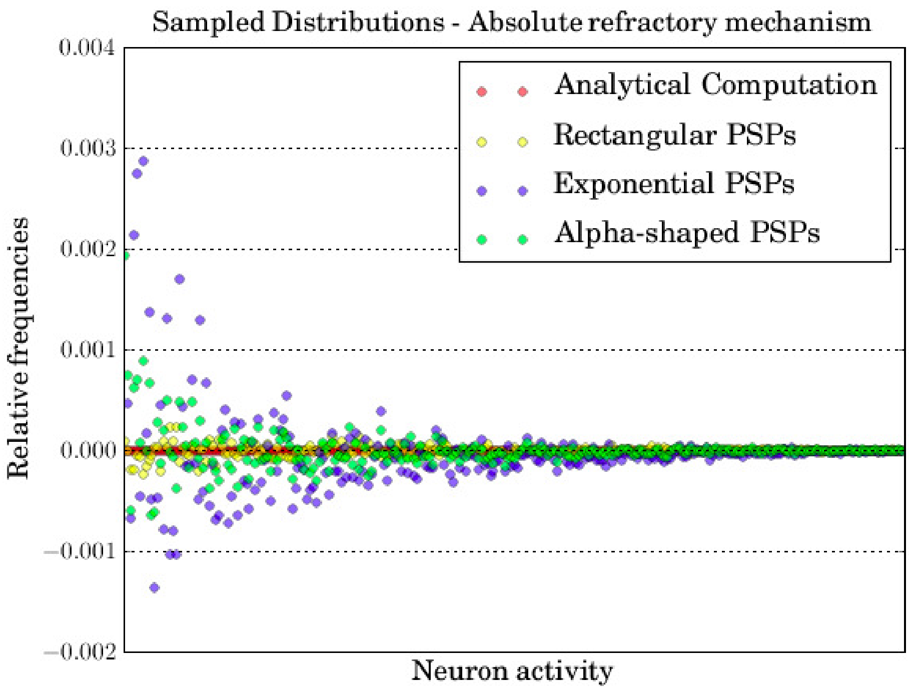

Figure 7.2: Relative frequencies in relation to the analytical distribution, brought to the baseline: Big difference between analytical distribution and sampler with exponential PSPs.

In order to be able to better observe the differences in these sampled distributions, they were analyzed in relation to the analytical distribution: In the second plot in figure 7.1 the analytical distribution is brought down to the X-axis, and all other distributions are shown in this relation. In this plot one can see very well that the exponential PSPs perform worst, while the rectangular ones are perfect, with limited precision.

In order to make the setting a bit more realistic, the absolute refractory mechanism was replaced by a relative one, bringing the neuron model a step further away from the proved theory, and nearer to biologically plausible settings.

Again, various PSP shapes - namely rectangular, exponential and alpha-shapes - were observed by sampling over some time period, and the resulting distributions or relative frequencies of samples were plotted in figure 7.3.

Again the samples on the X-axis were sorted according to the relative frequencies in the analytical distribution, and the height of the relative frequencies was plotted on a log-scale to better visualize the differences.

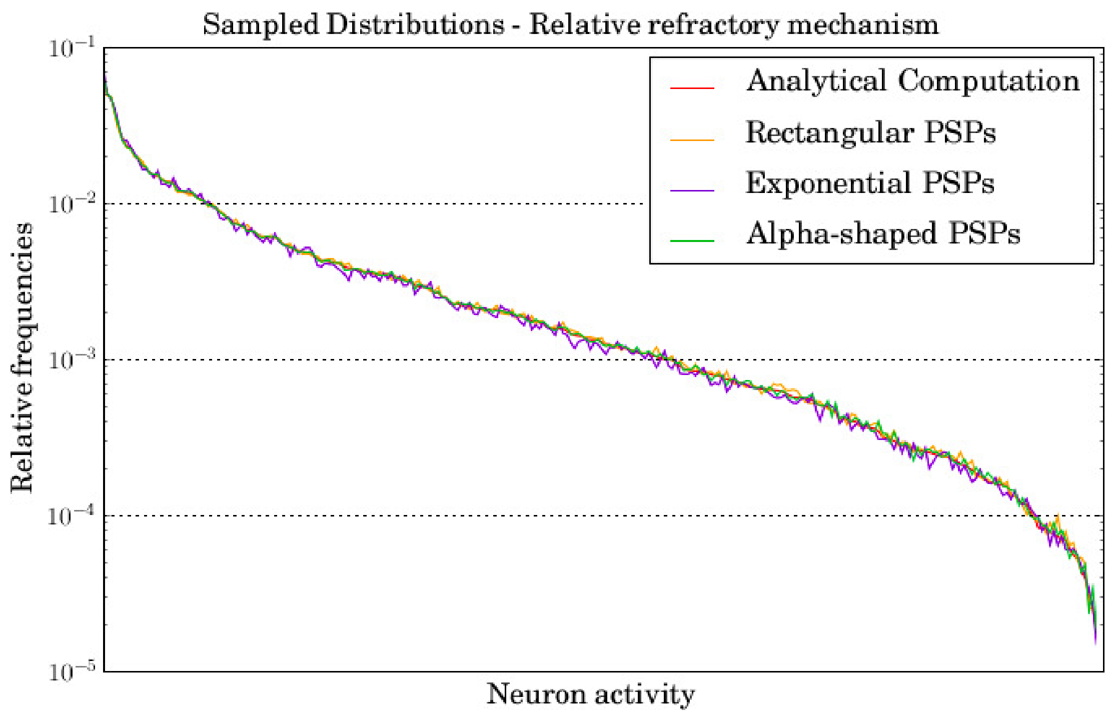

Figure 7.3: Sampled distributions: The samples sorted according to their height in the correct distribution, relative frequencies on a log-scale: No correct sampling with rectangular PSPs, alpha-shaped PSPs working quiet good.

When looking at the digram, one can observe that the exponential PSPs again are the worst in approximating the correct distribution, even though now the similarity to the analytical distribution is higher.

The details can again be better observed when looking at figure 7.4, where the various distributions are plotted relatively to the analytically computed and correct distribution, which has been moved to the X-axis, representing now the baseline.

The rectangular PSPs are not leading to correct sampling any more, while for the other two PSP shapes - exponential and alpha-shaped ones - the relative refractory mechanism leads to an improvement in approximating the correct, analytically computed distribution.

The differences between the various distributions are already visible in these figure, but they can be better analyzed when looking at the Kullback-Leibler divergences to the analytical distribution. The KL divergences are plotted in the diagram in figure 7.5 on the Y-axis.

All used PSP shapes are additive ones. The KL divergences of the first two bars (yellow and orange) are computed using rectangular PSPs, the second two bars (blue and violet) had exponential PSP shapes, and the last two bars (green) were with alpha-shaped PSPs. The first bar of each pair used the absolute refractory mechanism, and the second bar a relative one.

As mentioned before, the distance between the exact sampler (yellow) and the analytical distribution can be explained by the sampling time which is always limited in practice.

As another quiet obvious finding one can see that rectangular PSPs do not work correctly any more, when using a relative refractory mechanism.

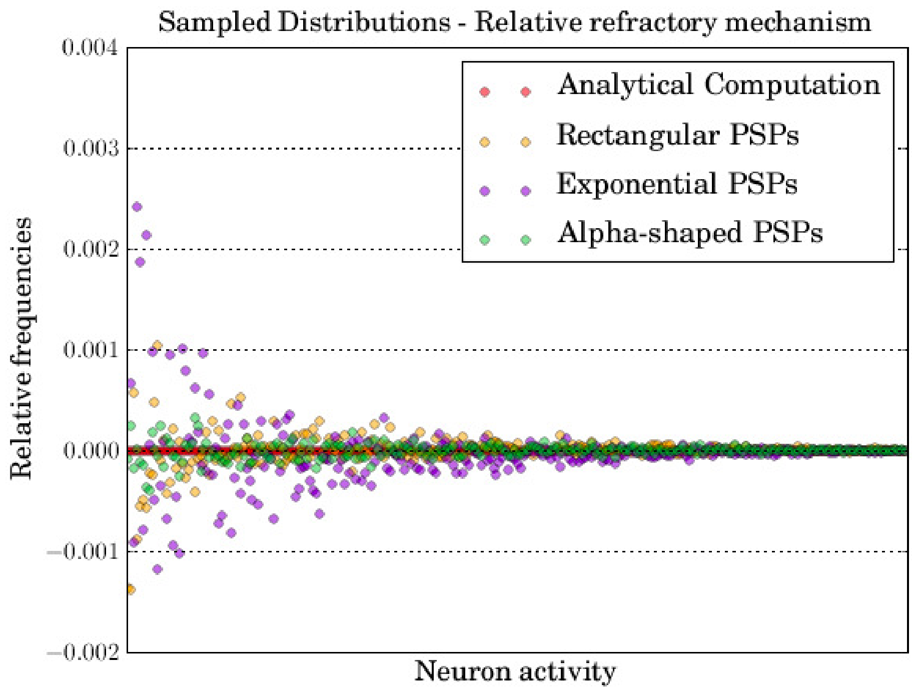

Figure 7.4: Relative frequencies in relation to analytical distribution, drawn on X-axis: Alpha-shaped PSPs working quiet good. In general: smaller difference of non-rectangular PSPs when using relative refractory mechanisms

On the other hand, one can observe also immediately that both non-rectangular, i.e., exponential and alpha-shaped, PSPs are approximating the correct distribution better when using a relative, and not the absolute refractory mechanism.

In general, renewal PSPs lead to even lower KL divergences, but again they are not included in this figure due to their limited relevance in a biological sense.

A rather more interesting observation is the performance of alpha-shaped PSPs using the relative refractory mechanism: Already with this experiment one could imagine that in certain situations it might be worth thinking of having more realistic network models and thus taking this (relatively small) loss.

When observing different kinds of networks with alpha shaped PSPs and the relative refractory mechanism, one notices an interesting evolution: While in networks with quiet strong synaptic connections (like the one mentioned previously) renewal PSPs perform much better, this changes completely when switching to sparsely and/or weakly connected networks, where the additive PSPs result in lower Kullback-Leibler divergences.

One should note, that all these findings can be observed in a similar matter when using other network settings, with the only difference that the errors vary with the network settings (and with the exception mentioned in the previous paragraph): When having, for example, a extremely small network of only two - strongly connected - neurons, this effects and differences observed also in this network model are strengthened even more.

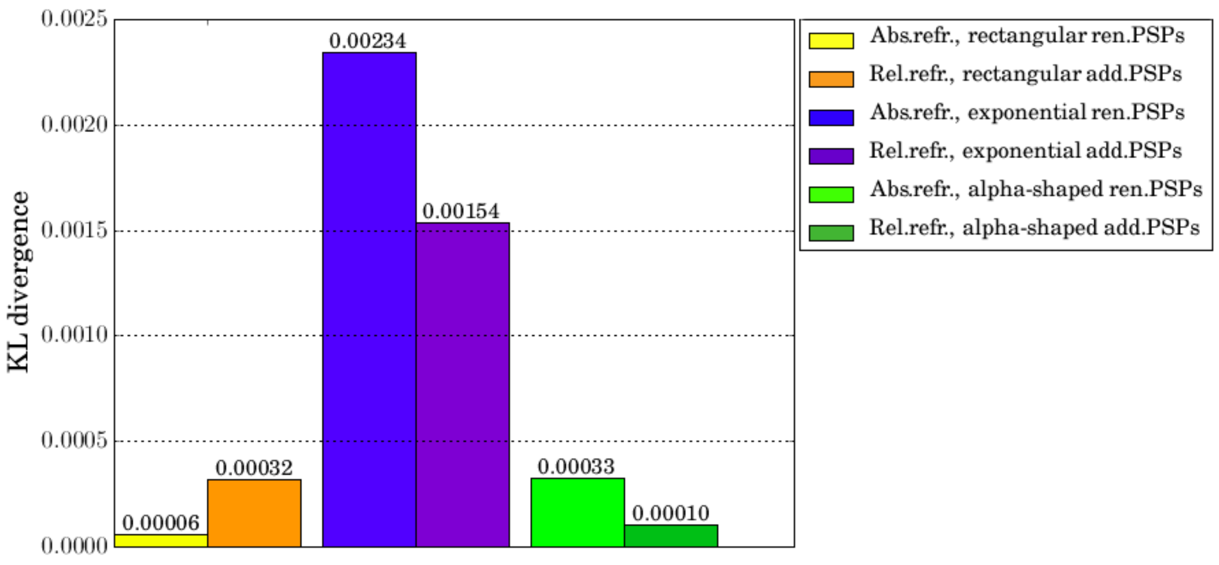

Figure 7.5: Kullback-Leibler divergences of various sampled distributions to correct analytical distribution: Rectangular PSPs sample correctly only in combination with the absolute refractory mechanism. Relative refractory mechanisms are better for not-rectangular PSP shapes. Especially good performance of alpha-shaped PSPs in combination with relative refractory mechanisms (Renewal PSPs would lead to even better sampling than these additive ones do).

7.3 Delays

In this section the influence of (increasing) global synaptic delays on the distribution or relative frequencies of various samples is analyzed. For this experiment, two different networks were chosen, one being the same as in the experiments about PSP shapes and refractory mechanisms, consisting of eight sparsely connected neurons, and a second one which is really simple, such that it has obvious symmetries (in the correct, analytical distribution), and it is therefore understandable how the distributions should be, which is the reason why one can simply understand the problems caused by delays.

In order to observe the influence of delays in general, the larger network was used, and linearly increasing global synaptic delay was set, starting from the exact sampler (absolute refractory mechanism and rectangular PSP shapes) with no delay, until a delay of 30 milliseconds. For each of these settings for a sampling time of 10^6 milliseconds, the Kullback-Leibler divergence to the correct analytical distribution was computed, as it is shown in figure 7.6.

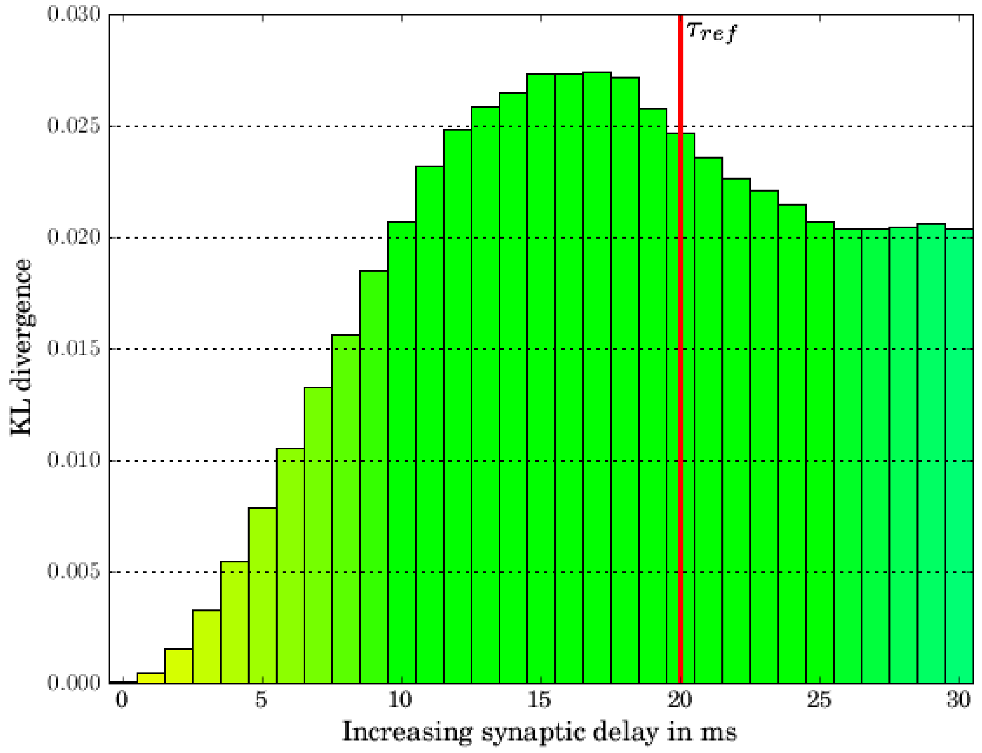

Figure 7.6: Kullback-Leibler divergence of a two-neuron (exact) sampler with increasing global synaptic delay in a strongly connected two-neuron network to the analytical distribution: The KL values increase in comparison to the exact sampler (with no delay) until a certain threshold, reached when the refractory time is over.

This figure shows the values of the Kullback-Leibler divergence on the Y-axis and the increasing synaptic delay on the X-axis, where the end of the refractory period is marked by a vertical red line.

It is visible in this plot, that the distance between the exact sampler (with limited sampling time) and the correct distribution is quiet small, and that the divergence values rise with increasing synaptic delays, as it would be expected. Additionally it is visible that after the refractory period of $\tau_{ref} = 20$ milliseconds is over, the Kullback-Leibler values remain nearly constant.

These findings can also be observed using different networks. As in the above case it is not possible to show the evolution of the relative frequencies of all samples, as the number of samples grows exponentially with the number of neurons in the network, and thus the analysis of the evolution of all samples becomes infeasible very soon, another setting was used to observe the detailed effects of delays. Additionally, the correlation between the neurons in a network is much easier visible in a tiny network, where all influencing factors inside the network can be observed at any point in time.

Therefore, for additional observations, another network, consisting of only two strongly connected neurons was used, which makes all interactions and internal effects visible. Therefore, analyzing the influences of increasing delays is much easier in this case.

The observed network is again a Boltzmann machine, this time consisting of only two connected neuron-like units, having $W_{kl} = w$ for $k \neq l$, and $b_k = -\frac{w}{2}$, where w is a free parameter, for which various values were tested. It was found that a high coupling leads to more visible and observable errors in the distributions, and therefore a value of $w = +2$ was chosen for the following plots that are shown. In this case additionally the marginal probabilities in the network are at 50% for both neurons, such that the correlations are visible easily.

In order to observe the influence of delays in an isolated network model, the used sampler is the exact one, i.e., it uses rectangular PSP shapes in combination with the absolute refractory mechanism, where it does not matter whether renewal or additive PSPs are used. Therefore, in the following experiments only the delay (and - of course - the limited sampling time) influences the correctness of the samplers.

In order to analyze how the sampled distributions change when the global synaptic delay in a network model increases, the relative frequencies of all samples created by samplers with linearly increasing delay are plotted in figure 7.7, in comparison to the analytical distribution.

This diagram shows the different samples on the X-axis, where the first digit stands for the first neuron in the network, and the second digit stands for the second neuron. A 0 means, that this neuron is inactive, while 1 indicates that this neuron is currently active. The Y-axis shows the amount of the relative frequency of each of these samples.

The analytical distribution (in red) reflects the symmetries in the network: The marginal probabilities of both neurons are equal (as explained previously both have a marginal probability of 50%), and it is also clearly visible that in this network there is a high correlation between the two neurons: The probabilities of both neurons having the same state (either both being active, i.e., 11, or both being inactive 00) are much higher than the probability of one neuron being active, and the other one being inactive.

The same properties of the analytically computed distributions are also mirrored by the exact sampler (in yellow), having no delay: Again it is visible that this sampler samples correctly (with some small variances that can be explained by the limited sampling time), and the sampled distributions are therefore the same.

As it was expected, with increasing global synaptic delay (in yellow, green an cyan) - linearly increasing from one to 30 milliseconds - these sampled distributions (and even the marginal distributions of the two neurons) change drastically.

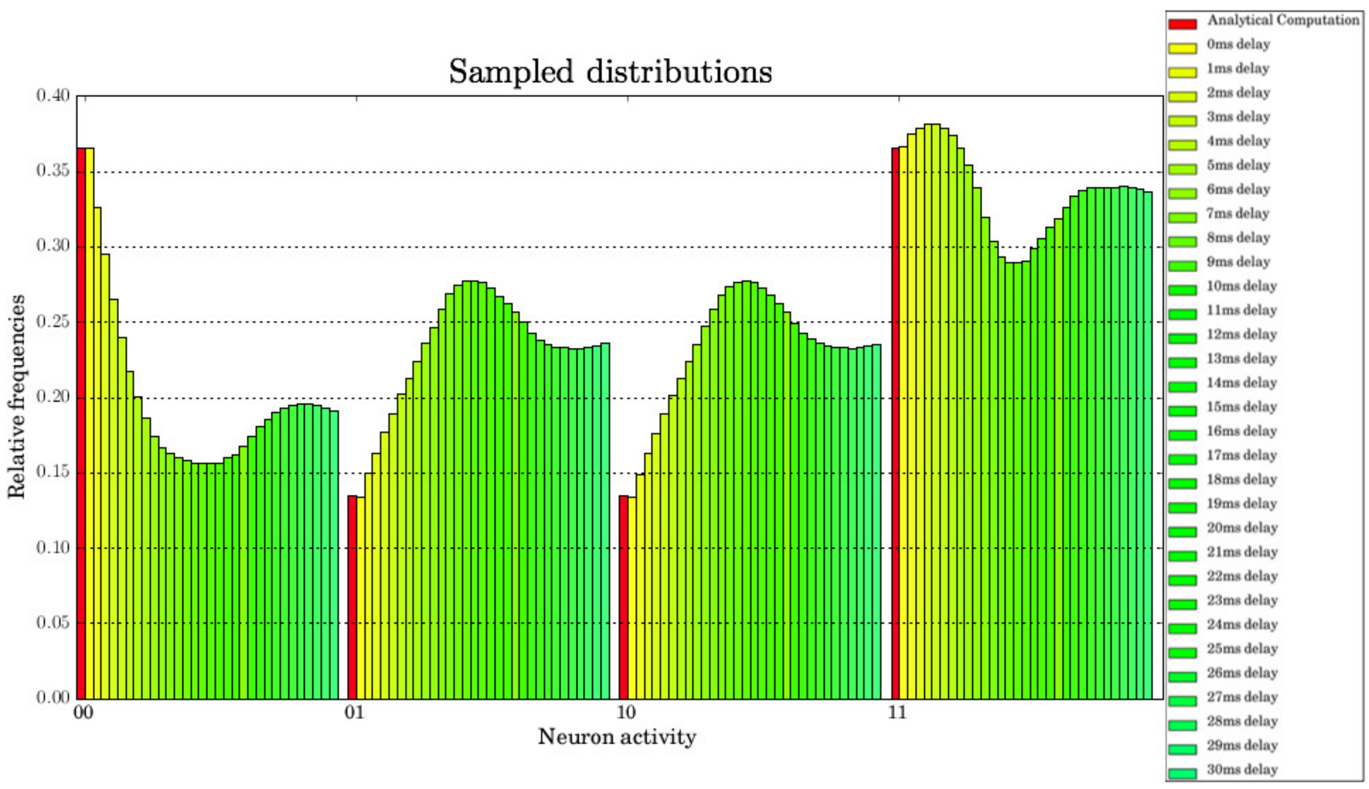

Figure 7.7: Sampled distributions of an (exact) sampler with increasing global synaptic delay in a strongly connected two-neuron network, in comparison to the analytical distribution: Sampled distributions and marginals change with increasing delay, until a delay of the refractory period.

On the other hand, when observing the samplers with 20 milliseconds or more of delay, one can observe that the distributions do not change much any more, and they seem to have converged - to a new distribution. It might be important to mention, that also in this case the refractory period was set to be $\tau_{ref} = 20$ milliseconds.

This was the reason why a new assumption came up: This convergence to a new - different - distribution could be caused by some kind of de-correlation of the (two) neurons in the network. If the delay is too high - and even higher than the refractory period - then the information that is passed to the other units in the network is transmitted only at a moment where this information is already out-dated, as any spike-caused rise in the PSP shape vanishes after the end of the refractory period.

So as to explore this assumption, the linear correlation between the two neurons in the exact sampler with increasing synaptic delay was looked at. A plot showing this linear correlation can be seen in figure 7.8.

The X-axis shows the increasing delay in milliseconds, on a $\log_2$ scale, where the end of the refractory period is indicated by a thicker light gray vertical line at R. The Y-axis indicates the linear (Pearson) correlation between the two neurons, computed as:

$$\rho_{z_0,z_1} = \frac{\mathrm{cov}(z_0, z_1)}{\sigma_{z_0} \cdot \sigma_{z_1}} \tag{76}$$

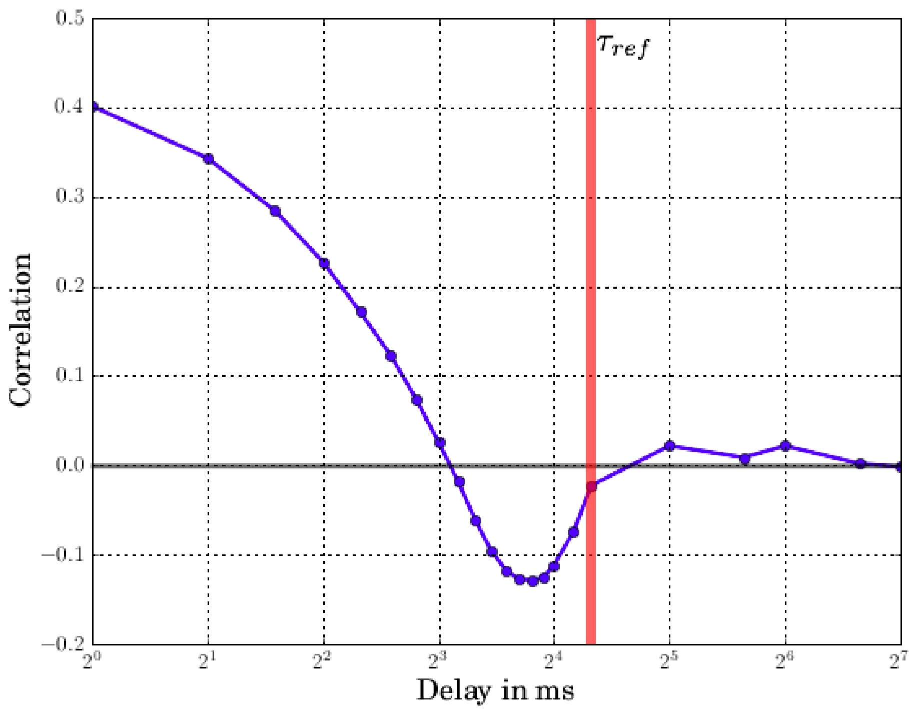

Figure 7.8: Linear correlation between the neurons in a strongly connected two-neuron network of samplers with increasing global synaptic delays (log-scale): Linear de-correlation with a delay larger or equal to the refractory time τ_{ref}, shown in red in the plot.

When observing this diagram, one can see immediately, that the linear correlation between the two neurons, starting at some positive value (as the weights $\boldsymbol{W}$ are strongly positive), decreases until it comes to a inverse correlation (with a minimum at about 13 to 16 milliseconds).

But the relevant information for the above stated assumption comes after this minimum: When observing the linear correlation after the refractory period is over, it is visible that the correlation converges approximately to 0. Thus, it indicates a linear de-correlation, approving the previous assumption.

Again, note that the same findings are also true for different network models, with the difference that when using negative weights the correlation line is mirrored on the X-axis of the above plot. But the linear de-correlation of neurons in such a network model with increasing delays growing to the size of the refractory time is true for all observed neural networks.

An additional interesting but also mysterious effect can be observed in all executed test runs: While in the beginning - until about two thirds of the refractory time has passed - the Kullback-Leibler divergence, and thus the difference to the correct distribution increases monotonically, it seems that afterwards there is a slight short-temporal decrease in the performance.

In all three of the previously described plots one can see that in the last third of the refractory time the performance of the sampler is even lower than when the refractory period is over. The reason for this phenomenon could not be found until now but it might be that the knowledge about a neuron's state, which is nearly as old as the refractory period, contains even less information about the real current state than some "random" information, that can be obtained after the refractory period is already over. Especially when looking at the correlation between the units this feature becomes remarkable: The correlation becomes negative in this phase, and de-correlation turns into a positive correlation, before the correlation approaches zero.

On the other hand, note also that a low delay of one or two milliseconds, which seems quiet plausible in a biological sense, does not influence the resulting probability distributions that much and might therefore be acceptable in some situations where biological realism is important.

7.4 Synaptic connections which are not symmetric

Already when the concept of a Boltzmann machine was invented, it was clear that in real biological networks the synaptic connections are usually not symmetric [34]. As in this work, Boltzmann machines are used for neural sampling, the same restrictions of having only symmetric synapses apply also here. The consequences that result from not using symmetric connections are shortly covered in the following.

The idea is to use still the same network model as in the previous analyses, consisting of eight neurons with renewal rectangular PSPs with a refractory period of $\tau_{ref} = 20$ milliseconds, using an absolute refractory mechanism and no synaptic delay. The bias vector is still the same, i.e., Gaussian distributed according to $\boldsymbol{b} \sim \mathcal{N}(-1; 0.5)$. As a basis, the same (still symmetric) matrix of Gaussian distributed values $\boldsymbol{W} \sim \mathcal{N}(0; 0.2)$ is used.

The principle according to which the weight matrix $\boldsymbol{W}$ is then modified, is sketched in figure 7.9. The upper triangular matrix is multiplied with a factor $(1 + \epsilon)$, while $(1 - \epsilon)$ is multiplied with the lower triangular. Thus, $\epsilon = 0$ indicates the symmetric matrix, while higher values of ϵ make the matrix non symmetric. Note that - as the weight values can be both, positive and negative - this does not force one of the triangles to increase and the other one to decrease.

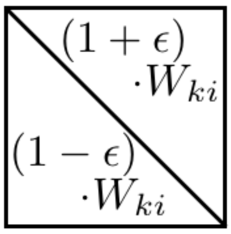

Figure 7.9: Analyzing non-symmetric synapses: The figure shows the structure of the weight matrix $\boldsymbol{W}$, which is modified, resulting in synaptic connections which are not symmetric any more.

In order to obtain some comparable values, one can look at the Kullback-Leibler divergence from the sampled to the correct and analytically computed distribution also in this case, which is shown in figure 7.10. The X-axis denotes increasing values of ϵ, i.e., increasing changes in the symmetry, while the Y-axis shows the hight of the KL divergence.

As it was expected, this diagram shows that increasing values of ϵ cause the Kullback-Leibler divergence to rise. This means that the farther away one gets from the theoretical requirement of having symmetric synaptic connections, the higher the systematic error of the sampling network gets.

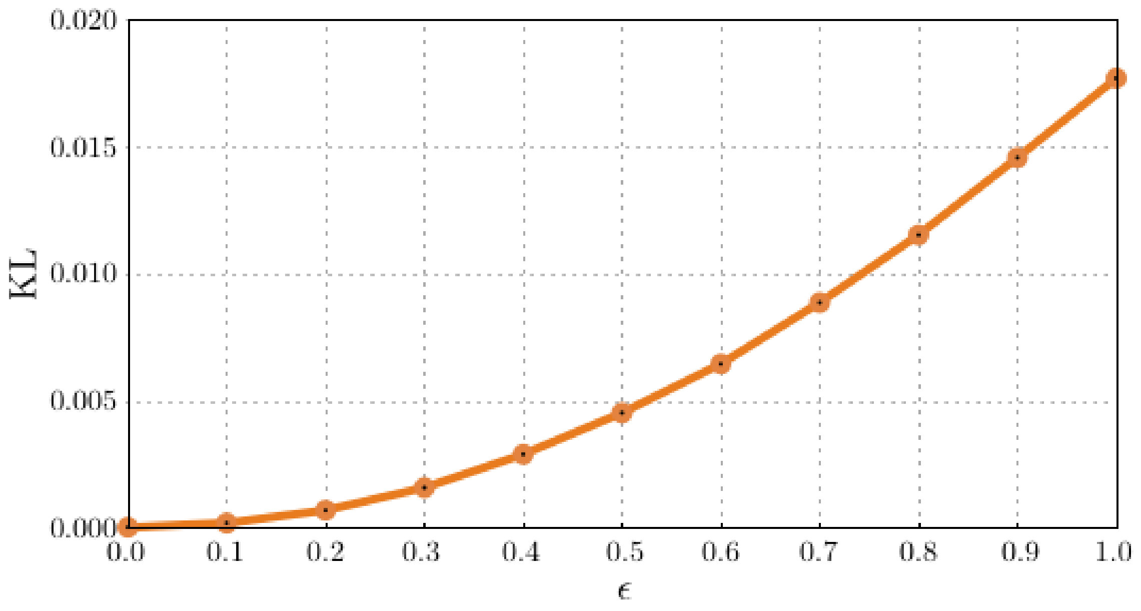

Figure 7.10: The Kullback-Leibler divergence to the correct (and theoretically computed) distribution: The X-axis shows the increasing values of ϵ, for $W_{ki} \mapsto W_{ki} \cdot (1 \pm \epsilon)$, while the Y-axis indicates the KL-divergence to the correct distribution. The further the network model gets away from the theoretical requirement of having symmetric connections, the more the KL divergence between the sampled and the true distributions increases.

7.5 More realistic situations

When talking about a realistic situation like how it could be in a biological experiment, then one always has only a very limited number of samples - due to the limited sampling time. For this following observations a new assumption was defined:

Assuming that the sampling time in a network is limited, meaning that in any case there is only a limited precision possible (in approximating the true distribution) - is there a “large” additional systematic error of the sampler when using biologically realistic settings instead of an exact, correct-proven sampler in this case?

This is exactly what is observed in this section: Therefore, a limited sampling time of only ten seconds was assumed. The settings taken into account were a relative, sine-wave refractory mechanism, in combination with alpha-shaped additive postsynaptic potentials and an additional global synaptic delay of two milliseconds, which is quiet biologically realistic according to [6] or [52] (about 1.5 milliseconds).

Again, the refractory period was set to $\tau_{ref} = 20$ milliseconds. The settings for the postsynaptic potentials were the optimal ones in terms of minimal Kullback-Leibler divergence (as these parameters are not known from biology).

The settings of the network models used are the same as in section 7.2: A Boltzmann machine consisting of eight neurons, having symmetric synaptic connections characterized by $\boldsymbol{W} \sim \mathcal{N}(0; 0.2)$ and $\boldsymbol{b} \sim \mathcal{N}(-1; 0.5)$, i.e., initialized randomly according to some Gaussian distributions.

For the observation of the influences of realistic but not exact network models over such a limited sampling time of only 10 seconds, 500 executions of this limited-time sampling with each network model were executed.

In figure 7.11 the biologically realistic and the exact samplers are directly compared, using the Kullback-Leibler divergences to the correct and analytically computed distribution of the samples.

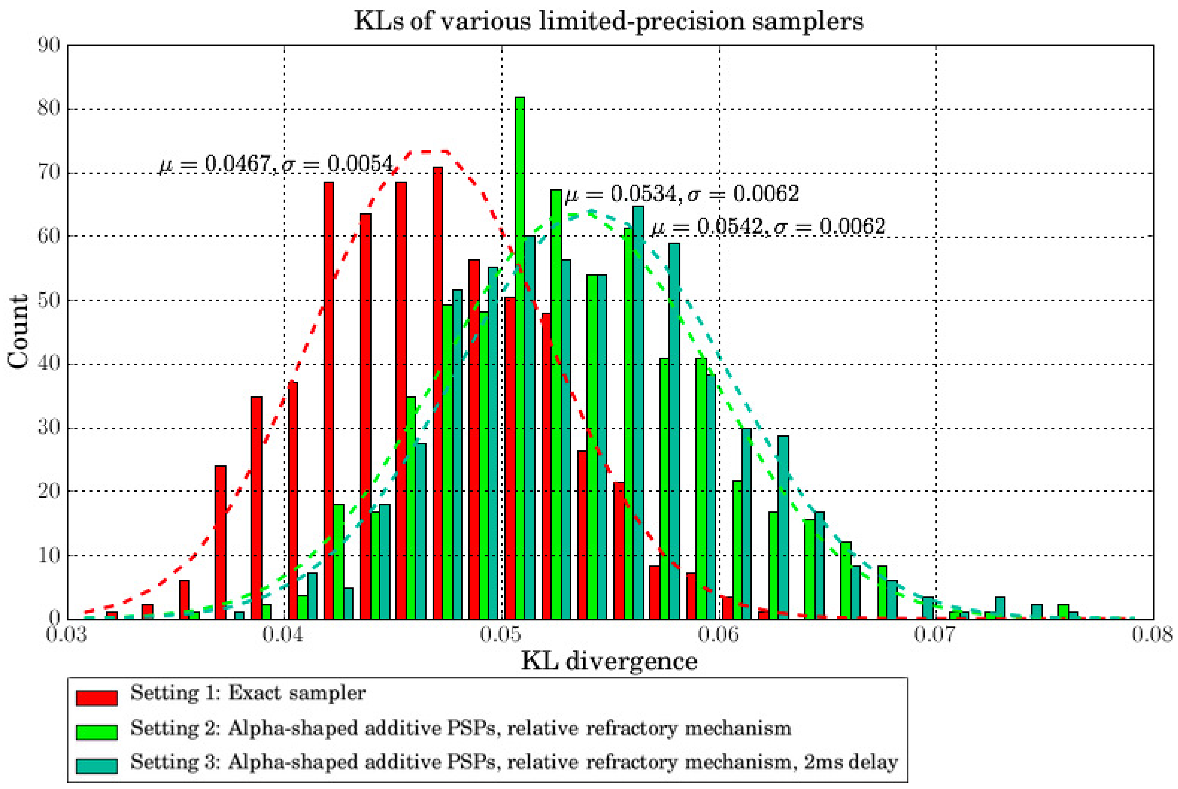

Figure 7.11: Kullback-Leibler divergence distribution of different samplers in combination with realistic settings, i.e., only 10 seconds of sampling time, leading to situations with only limited precision. Amount of KL heights in 500 trials using the same network. Comparison of exact sampler to biologically more realistic samplers with alpha-shaped additive PSPs and a relative refractory mechanism: Mean KL divergence of exact sampler only slightly lower than of realistic samplers, having the quotient of the difference between the means and sigma in the magnitude of 1: Systematic and stochastic errors are similar.

In this figure, on the X-axis the heights of the Kullback-Leibler divergences are plotted, while the Y-axis indicates the absolute frequencies of these KL values (of the 500 executions per network model setting). The Kullback-Leibler values are binned in 25 equally distributed bins in the range of $[0.03; 0.08]$.

There are three different network settings plotted in this figure, sampling always with the same limited precision:

- Setting 1: The first setting (shown in red) represents the exact sampler, were all neurons have rectangular PSP shapes, and an absolute refractory mechanism is used.

- Setting 2: This setting (plotted in green) is much more realistic in a biological sense: It uses alpha-shaped additive PSPs for all neurons, and a relative refractory mechanism (a sigmoidal function).

- Setting3: The last setting (in green-cyan) is the same, realistic one as in setting 2, with the only difference, that this time an additional delay of 2 milliseconds was added to the network model.

Additional to the binned frequencies of the different Kullback-Leibler divergence values, in the same figure the distributions of all KL values are indicated with dashed lines (in the same colors respectively), using the mean values of the Kullback-Leibler divergences and their standard deviations.

Now, as sampling time is even more limited, the Kullback-Leibler divergence of the exact sampler to the correct distribution is not equal to 0 any more, and already when comparing the KL distributions (or the dashed lines of them) of the different samplers one can observe that these differences are not very large.

This can further be observed when looking at the mean and standard deviation values. The biologically realistic samplers have higher mean KL divergences, but nevertheless the variance is higher than the one of the exact sampler.

Mathematically, the quotient of the absolute difference between the means μ_1, μ_2 of two distributions and a standard deviation σ of any of them can be computed, being

$$\frac{|\mu_1 - \mu_2|}{\sigma} \approx \frac{|0.053 - 0.047|}{0.005} = \frac{0.006}{0.005} \approx 1 \tag{77}$$

As this metric is in the magnitude of 1, this means that in this case the systematic error (defined as the absolute difference between the means of the KL divergence distributions) and the stochastic errors (which can be measured by any of the standard deviations) are quiet similar, and nearly the same.

As the stochastic error decreases with increasing sampling time, but the systematic error does not (or at least not so much), one can infer the following conclusions:

- When more samples are available, i.e., when sampling time increases, the stochastic error will decrease. Thus the systematic error of using biologically realistic network settings will be very crucial, whereas this decision will probably influence sampling too much. With large sampling times, the error of using realistic settings instead of exact ones is comparatively very high:
  Many samples - high influence of systematic error: Realistic settings should not be used.

- If even fewer samples are available, which means that sampling time would be even shorter, then the stochastic error increases, forcing the systematic error of biologically realistic networks to become comparatively small. Therefore, in such situations the use of realistic settings does not influence the resulting distributions very much, and it could therefore be a good decision to use them:
  Few samples - low influence of systematic error: Realistic settings could be used.

This means that in some situations - especially when being confronted to experiments with limited samples - it might be worth thinking of using realistic and not exact network settings, as they might cause only a small additional systematic error relative to the stochastic one which cannot be avoided in these cases.

Note again that these findings have been reproduced using different network models (with other sizes and synaptic connections), leading to the same conclusions. Also note that when using renewal PSPs - instead of the additive ones used above - the Kullback-Leibler divergence distributions get nearly congruent to the ones of the exact sampler.

7.6 Conclusion

From the above experiments, one can deduce the following major implications:

- Rectangular PSPs sample correctly only with the absolute refractory mechanism.
- With non-rectangular PSPs relative refractory mechanisms are better than the absolute.
- Delays cause de-correlation of the neurons, when they are equal to the refractory time.
- With limited sampling time, realistic settings cause only small additional errors.

8 Application of neural sampling to a simplified model of vision

This section describes how the previously described implementation of the neural dynamics sampler can be applied to a concrete problem: It is used to model a strongly simplified model of the human visual system on a symbolic layer. The modeled system is learned through maximum likelihood techniques and should then be able to recognize and complete very simple input patterns.

In the end the model should represent on a strongly simplified, symbolic layer the processes going on in the human brain during edge, contour or image completion, as described in the experiments of Lee et al in section 2.3.

8.1 Probabilistic model

There are obviously many fields of application for this neural sampler. In the setup described now, the neural sampler is used to simulate some realization on a symbolic layer of a strongly simplified model of the human visual system.

The outcomes of the biological experiments by Tai Sing Lee et al (see section 2.3) are the basis for the following neural network model on which the neural sampling implementation is applied. The generative model consists of multiple layers of partially, bottom-up - top-down connected neuron-like units, which - after learning - should be able to recognize and complete certain images and image segments.

The underlying idea of this sampler application is based on the architecture of the cortical areas in the visual system, described as a feedforward - feedback connected system in the biologically evidenced papers. The symbolic realization consists of three distinct layers of neurons, as it is shown in the sketched architecture in figure 8.1, and as it was already described shortly in section 5.3.

The layers represent - on a symbolic level - different visual cortical areas, namely some representation of the input (Y), an early sensory stage (Z) and higher areas (X).

In a real brain, the LGN gets directly some input signals from the eyes' retina through the optic nerve fibers. In the symbolic representation, the input is modeled by direct picture input that comes from outside the model to the input layer Y. There is exactly one neuron in the input layer for each pixel in the external input image. The input image is split up in disjoint receptive fields, and the neurons representing the pixels of one such subpart of the image are called one group G_j. The neurons $\boldsymbol{y}$ in this layer are not connected, but their firing rates directly reflect the intensities of the input image, summing up to the group firing rate ν_j of each group G_j (see section 5 for the explanation), such that the preferred null-causes are respected.

The external input image is divided into subparts, which represent the receptive fields of the units in the lateral layer Z. Each such quadratic subpart shows either a pre-defined pattern, or some noise, as described in section 8.3, and as shown in the sketched architecture. Internally, each quadrate is represented by its pixels, or more exactly, by a vector of values, defining the intensities of the image pixels.

From this input layer Y there is a direct feedforward connection to the lateral layer Z. This lateral layer should symbolically stand for the first (and perhaps partially also the second) visual area. It gets the input signals from the lower layer through synaptic connections: The lateral

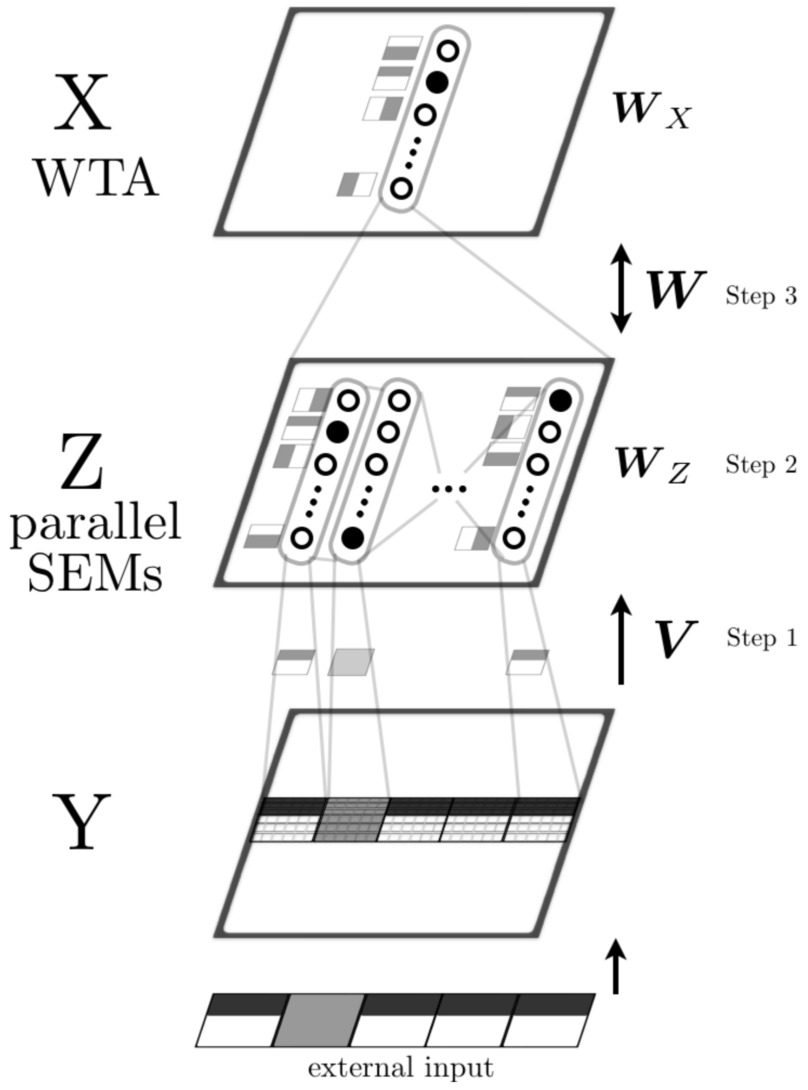

Figure 8.1: Application of the neural sampling implementation to a multilayer neuron model.

layer is subdivided in different parallel SEM-units [56],[55],[30] (see section 5.1), each consisting of a number of neurons.

The neurons inside such a SEM-unit are strongly (theoretically infinitely) inhibitory among each other, meaning that only at most one of them can be active at a time, causing all others to stay inactive for its refractory period. If none of the neurons inside a unit is active, this is interpreted as being the background hypothesis. In this model this can be thought of as if another unit (which is not plastically modeled) inside this SEM is active. In the following this situation will be referred to as null cause (i.e., there is no modeled cause for this situation), or background hypothesis. Learning in such SEM-units is done according to a modification of [56], describing how such spiking neurons are able to detect the “hidden causes” of some input.

Each unit has its distinct receptive field on the lower layer, meaning that each SEM-unit only gets input from one non-overlapping subpart of the represented image in the input layer. Therefore, each SEM-unit should recognize the image of such a subpart of the input.

The connections between the two lower layers Y and Z are all feedforward connections, encoded in the matrix $\boldsymbol{V}$. These synaptic connections ensure that each neuron in the lateral layer only gets input from neurons that are located in its receptive fields. In this way, each SEM-unit is able to recognize the image or pattern in its receptive field.

The two higher layers Z and X are modeled as being a single structured Boltzmann machine, having symmetric synaptic connections, included in the weight matrix $\boldsymbol{W}$ and the vector $\boldsymbol{b}$, where some weights - where no synaptic connections exist - are never learned.

SEM-units with neighboring receptive fields are connected among each other, while units whose receptive fields have a higher distance are not connected. This enables the lateral layer to do already some kind of line or pattern completion, by excitatory connections between these SEM-units: The information has to be passed from one neighbor to the next one. After learning, this is encoded in the Boltzmann weights $\boldsymbol{W}_Z$ and $\boldsymbol{b}$.

As in this lateral layer Z all single input parts can already be recognized, and simple completion of patterns is already possible through the synaptic connections between neighboring SEM units, the top layer X has the duty to recognize the image as a whole, by getting information through weighted synaptic connections of all neurons in the lateral layer.

The top layer X itself consists of a single WTA-like unit, meaning that again inside this unit the synaptic connections are strong (in theory infinitely) inhibitory. In this case, one neuron has to be active in any case. In order to guarantee this, again the case if no (modeled) neuron is active is interpreted as having another neuron which is active at this time. All neurons of this top layer are connected to all neurons in the lateral layer, independently of the receptive fields, meaning that this unit is able to recognize the complete image as a whole. Again, the connections inside the WTA-like unit, but also between the top and the lateral layer are managed inside the Boltzmann weight matrix $\boldsymbol{W}$.

It should be noted - as these synaptic connections between the top two layers are symmetric weights - that the information is not only passed from the lateral layer Z upwards to the top layer X, but there occurs also top-down information transmission. Therefore, the lateral layer gets information about the understanding of the whole image by the top layer, and reacts according to that again by completing patterns or whole images.

A further expansion of this model is shown in figure 8.2. While the input layer and the lateral layer remain the same in this new version of the neural sampler application, the top layer has been modified. Instead of the WTA-like unit with inhibitory connections between the single neurons, it contains now a WTA structure (when including the non-modeled neuron which is active in case of the background hypothesis - if no other neuron is active), where each WTA-"unit" is represented by a population of L neurons.

In this case, the neurons inside such a population stimulate each other, which is encoded as excitatory (positive) weights inside the weight matrix $\boldsymbol{W}^+$. Neurons of different populations can still be seen as being different WTA-units, and thus they inhibit each other strongly. Therefore the single WTA-units of the first version have now been replaced by populations of neurons.

The connections between the second and the top layer are still the same: As the receptive field of the whole WTA-like system of the top layer is on the entire lateral layer, each of the top layer units is connected to each neuron of the lateral layer.

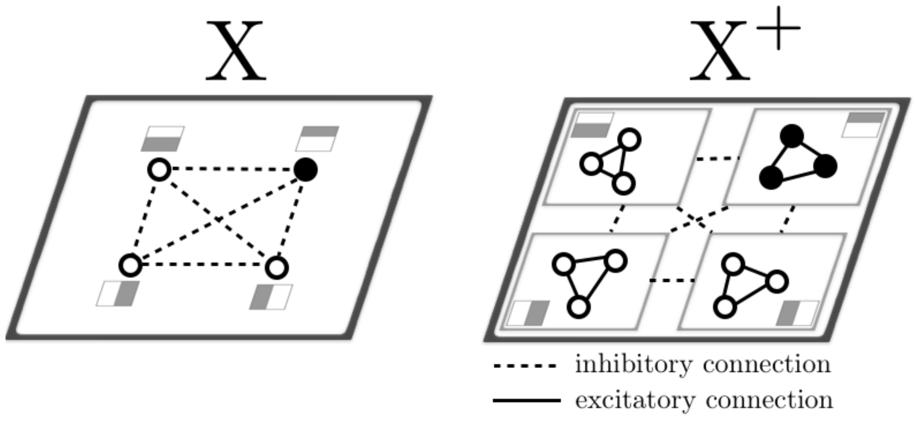

Figure 8.2: Another version of the application of the neural sampling implementation to a multilayer neuron model. This time, the third (top) layer is further expanded to consist of a WTA-like unit of populations of neurons. The expansion of this top layer is shown in here.

8.2 Learning in the multilayer network model

In the following, the learning process of the model is introduced and briefly described. In this case of the three-layered generative model representing on a symbolic level the cortical visual system in real brains, learning is basically done in two distinct steps [75],[23]:

In a first step the feedforward weights between the input layer Y and the lateral layer Z have to be learned, which is done by some sort of Hebbian learning.

In the next step the symmetric feedforward - feedback weights of the synaptic connections between the middle layer Z and the top layer X, as well as within both of the higher layers (Z and X), are learned. As these two layers are realized and implemented as a Boltzmann machine, learning is done by standard Boltzmann machine learning.

Both of these two methods to learn synaptic weights are described in the following.

8.2.1 Step 1: Feedforward Hebbian learning

As described previously, the weight matrix $\boldsymbol{V}$ contains all synaptic weights between the input layer Y and the lateral layer Z, i.e., between the neurons $\boldsymbol{y}$ of the input layer Y and the regarding neurons $\boldsymbol{z}$ in the SEM-units.

The idea for this learning step is to apply machine learning rules, more exactly expectation maximization. In this model as already described briefly in section 5.3.4 this is done by minimizing the difference between the actually observed numbers of spikes in the input layer Y and the expected one, given the activity of certain neurons $\boldsymbol{z}$ in the lateral layer Z. As the input to the model is not modified internally, this approximation has to be realized by adapting (or learning) the expectation. Therefore, the idea behind EM in this case is to base the learning step on this difference to be minimized:

$$\Delta V_{ki} \propto (\text{observed \# spikes of } y_i|z_k = 1) - (\text{expected \# spikes of } y_i|z_k = 1) \tag{78}$$

where ΔV_{ki} is the update of the weight matrix entry V_{ki}, indicating the synaptic weight from neuron y_i to the lateral layer neuron z_k in the weight matrix $\boldsymbol{V}$.

Mathematically, this is formulated as:

$$\Delta V_{ki} = \tilde{\eta}_{ki} \cdot \eta_V \cdot z_k \cdot \left(y_i \cdot \mathrm{e}^{-V_{ki} - \log \frac{\nu_{0i}}{\nu_j}} - \tau_y \cdot \nu_j \right) \tag{79}$$

where again ΔV_{ki} indicates the change in the weight of the connection between y_i and z_k in the matrix $\boldsymbol{V}$, and ν_{0i} denotes the background hypothesis firing rate. $\tilde{\boldsymbol{\eta}} \in \{0,1\}$ is used to ensure that no weights are learned for non-existing synaptic connections.

In practice, this means that the $\boldsymbol{V}$ matrix contains the weights of the synaptic connections between these two layers Y and Z, where of course there are only non-zero entries in the matrix $\boldsymbol{V}$ if a lower-layer neuron y_i is lying in the receptive field of the regarding neuron z_k in the lateral layer.

The expected numbers of spikes is computed as the group firing rate ν_j of group G_j, times the PSP time constant τ_y of the neurons, while the observed amount of spikes is calculated by weighting the effective PSP shape values of y_i in the lower layer (i.e., additive rectangular PSPs, each of hight 1) with the current weight matrix $\boldsymbol{V}$, divided by the number of neurons inside a receptive field. Note that this last factor is only due to the specific structure of the model used here, having a fixed relation between the background hypothesis and the group firing rate:

$$e^{-log \frac{\nu_{0i}}{\nu_j}} = \frac{\nu_j}{\nu_{0i}} = |R_k| \tag{80}$$

Therefore, after learning has finished, this weights should enable the lateral layer Z to "understand" the input of Y.

One could also check the idea behind this learning rule: After the learning process, the weights should not change any more, meaning that the expected change is $E(\Delta V_{ki}) = 0$. When then applying this expectation to the learning rule, one gets:

$$0 = z_k \cdot \left(y_i \cdot \mathrm{e}^{-V_{ki} - \log \frac{\nu_{0i}}{\nu_j}} - \tau_y \cdot \nu_j \right) \tag{81}$$

If $z_k = 0$, then this equation is true in any case, therefore one can go to the other case, where $z_k = 1$, and proceed for this situation:

$$\begin{align} 0 &= y_i \cdot \mathrm{e}^{-V_{ki} - \log \frac{\nu_{0i}}{\nu_j}} - \tau_y \cdot \nu_j \tag{82} \\ \mathrm{e}^{-V_{ki} - \log \frac{\nu_{0i}}{\nu_j}} &= \frac{\tau_y \cdot \nu_j}{y_i} \tag{83} \\ V_{ki} &= -\log \frac{\tau_y \cdot \nu_j}{y_i} - \log \frac{\nu_{0i}}{\nu_j} \tag{84} \\ &= \log \frac{\frac{y_i}{\tau_y}}{\nu_{0i}} \tag{85} \\ &= \log \frac{\nu_{ki}}{\nu_{0i}} \tag{86} \end{align}$$

which should be the case after learning.

8.2.2 Step 2: Boltzmann machine learning

Boltzmann machine learning is based on updating both - the weight matrix, in this case $\boldsymbol{W}$, and the vector $\boldsymbol{b}$, which directly influences the single neurons' membrane potentials [2],[33],[34],[68].

In the case of the neural sampler application to the multilayer neuron model the Boltzmann machine includes the lateral layer Z and the top layer X, meaning that all weights within these layers, as well as the ones between the layers are all integrated in the Boltzmann weights.

The idea of learning in a Boltzmann machine is a well-known process in machine learning. In contrast to other learning rules like the previously described one, here the update process is not based on EM, but on maximizing the log-likelihood of the observations.

In practice, this is equivalent to minimizing the difference, in this case between the expected value of the data, and the expected value of the model itself, i.e., to modify the weight matrix in such a way that the model approximates the data it should be able to represent [2].

The update or learning step is based on trying to approximate the current posterior with the prior (to be learned), by trying to minimize the difference between prior $\boldsymbol{z_{prior}}$ and posterior samples $\boldsymbol{z_{posterior}}$. This is in the same way done for the weight matrix $\boldsymbol{W}$ and the bias vector $\boldsymbol{b}$, by using a learning rate η_W, η_b for both of them for the update:

$$\Delta W_{kl} = \tilde{\eta}_{kl} \cdot \eta_W \cdot (z_{posterior_k} \cdot z_{posterior_l} - z_{prior_k} \cdot z_{prior_l}) \tag{87}$$

$$\Delta b_k = \eta_b \cdot (z_{posterior_k} - z_{prior_k}) \tag{88}$$

where ΔW_{kl} stands for the update of weight matrix entry W_{kl}, being the connection between the neurons z_k and z_l, while Δb_k indicates the change in the bias value $\boldsymbol{b}$ of neuron z_k.

The weights $\boldsymbol{W}$ inside this Boltzmann machine should also be learned only for existing connections between neurons z_k, z_l, which is realized through the mask $\tilde{\boldsymbol{\eta}}$. Such non-existing connections are for example located between neurons of different SEM units which do not have neighboring receptive fields, but also synapses which would enable self-interaction of neurons (which here is encoded in the bias values).

Other synaptic connections that should not be learned are the connections within the parallel SEM units, and also within the WTA-like unit in the top layer: These connections have to be very strong - in theory infinitely - inhibitory, such that at most one of the neurons in such a unit at a time is active.

For the previously introduced formulas for learning, two different samples of neurons of the Z layer are required, namely posterior $\boldsymbol{z_{posterior}}$ and prior samples $\boldsymbol{z_{prior}}$. Even though in biology it is not completely clear how such prior samples are generated (perhaps during some kind of sleep phase), in this simplified model they are generated by the same model, not getting any external input, such that the membrane potential u_k of neuron z_{prior_k} can be computed from its bias value $\boldsymbol{b}_k$, and the values of the other neurons $\boldsymbol{z_{prior}}$ in this model, weighted by the corresponding synapses to neuron z_k encoded in $\boldsymbol{W_k}$:

$$u_k = b_k + \boldsymbol{z_{prior}}^\top \boldsymbol{W_k} \tag{89}$$

8.3 Experimental setup

In this section the setup used in the experiments with the sampler are explained in more detail, always referring to the same architecture as already shown in figure 8.1. The following more detailed descriptions start with the external input, and thus with the lowest level, going up to the top layer.

The external input is based on some picture. This picture is defined as a rectangular, gray-scaled image. It is divided horizontally in five individual, quadratic subparts. Each such subpart is a 6×6 pixel gray-scaled pattern. Four different patterns have been defined, shown in figure 8.3, dividing the picture horizontally and vertically in two halfs, where always one half is nearly black with 90% intensity, while the other half is nearly white with an intensity of 10%.

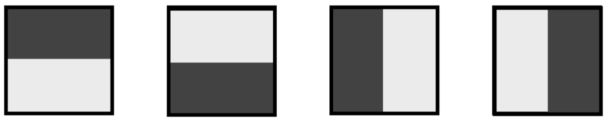

Figure 8.3: Four distinct patterns for the input image, to be learned by the sampler

These patterns are combined, five in a row, to build up the complete input image, as it is shown in figure 8.4, which shows the structure of the input picture, and various possible combinations of subparts, to build up the entire input image.

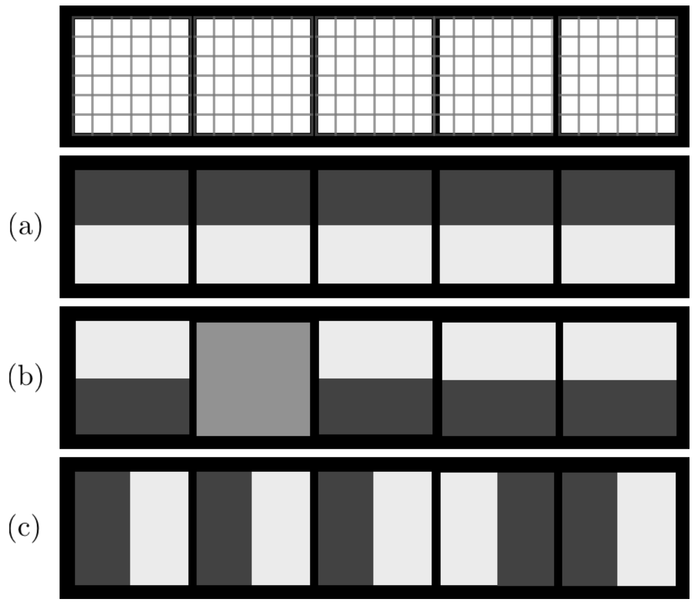

Figure 8.4: Various possible combinations as input picture. The uppermost picture shows the structure of an input picture, being subdivided in different receptive fields
(a) image consisting of five same patterns, which should be learned
(b) image with a “hole”, which corresponds to the “average” of all the learned patterns
(c) image with one “wrong” pattern inside

The model will have to learn, that a usual input image consists of five same patterns in a row, as the second row in figure 8.4 shows. It would of course in the same way be possible to learn a certain sequence of images, but in order to keep a clearer overview over the learned images it is more convenient to use a "simple" and clearly visible input combination to be learned.

In addition to the different patterns, some subparts of the input image might also be completely gray, as in the third row of the figure. Such "holes" are the average between all possible learned input patterns. After learning, the model should be able to (internally) complete such an image, i.e., the system should recognize the learned image as a whole.

Another possibility is to have - with a certain probability - input images that are not consisting of same patterns, but that some patterns are exchanged, like shown in the last one of the sample pictures. It is not completely clear how the model should react on such an input: It should - in any case - recognize all the matching patterns. As the inputs are always learned as complete images consisting of five same patterns, this last input example could represent some ambiguity or some overlap. Therefore, as this input might represent some partially covered image, the system could recognize this ambiguous situation, knowing that this input has been learned "differently".

For learning, the model is presented images consisting of five equal patterns, covering each pattern. The sequences and types of the input images have been defined manually, in order to keep the set of images for learning relatively small, nevertheless covering all types of images.

For testing the learned model afterwards, various combinations have been tried, starting from the complete image with five times the same pattern in it to check mostly the connections $\boldsymbol{V}$ between the input layer Y and the lateral layer Z. For testing the $\boldsymbol{W}$ matrix, various other combinations have been tried, like single holes on different positions to check whether and how the completion of such images works, and also single "wrong" patterns, in order to check whether the model would exchange such patterns.

As mentioned earlier, this external input is directly mapped to the corresponding neurons in the input layer Y, where for each pixel of the external input there is one neuron y_i. The firing rate of the neurons y_i is determined by considering the intensity of the corresponding pixel of the image, and the group firing rate of the group G_j it belongs to, i.e., the receptive field.

In each of the 5 groups G_j there are 36 neurons y_i in this model, as there is the same number of neurons in one subpart or receptive field of the image, and one group corresponds to one receptive field. Thus, in total layer Y consists of 180 unconnected neurons. Each neuron y_i has a PSP time constant of $\tau_y = 20.0$ milliseconds, and rectangular, additive shapes of postsynaptic potentials, each of height W_{ki}.

The null-cause of each of the neurons has been set to about $\nu_0 \approx 20$ Hz and as there are 36 neurons in each group G_j, the group firing rate is set to around $\nu_j \approx 750$ Hz.

The input layer Y is connected via the weight matrix $\boldsymbol{V}$ to the lateral layer Z. The Z layer consists of 5 SEM-like units, each having its own disjoint receptive field on the lower layer Y. This constraint is encoded in the synaptic weights $\boldsymbol{V}$, where all entries from neurons y_i to neurons z_k that do not lie in the same receptive field are set to 0.

The weight matrix $\boldsymbol{V}$ is learned through Hebbian learning, as described in section 8.2, using a learning rate of $\eta_V = 0.00005$ for 500 learning cycles of 400 milliseconds each.

Each single SEM unit in Z consists of 4 neurons, which should be able to recognize the different patterns after learning. The neuron for the null-cause is not explicitly included in the model, but it is defined, that if none of the 4 effectively existing neurons in one SEM is active, this is interpreted as being the null-cause, i.e., some “invisible” background hypothesis is active at this time.

In total there are 5 units of 4 neurons, thus 20 neurons z_k in the lateral layer, and each one has a refractory period of 20.0 milliseconds, and the global synaptic delay is set to 0. The single neurons have rectangular, renewing PSPs, and the absolute refractory mechanism is used.

The units inside the Z layer are connected among each other, having the weights contained in the weight matrix $\boldsymbol{W}$. Additionally their membrane potentials are influenced by the bias values $\boldsymbol{b}$. These bias values are set to $b_k = -2$ for all neurons z_k and are not learned.

The top layer X, which is connect both, in bottom-up and in top-down fashion to the lateral layer Z, consists of a single winner-take-all-like unit. Again, also this, as the SEM units in the lateral layer, consists of 4 neurons, which make up the whole top layer X. As the structure is very similar to the lateral layer, this top layer has in the internal architecture been embedded into the lateral layer, i.e., into the same sampling network.

Also the neurons x_k in the top layer X, as the ones in the lateral layer, have a refractory period of 20.0 milliseconds, and are characterized by rectangular, non-additive PSPs, with the absolute refractory mechanism, and a constant bias value of $b_k = -2$ for all neurons x_k.

As both higher layers Z and X are embedded in the same neural dynamics sampler, they also share one weight matrix $\boldsymbol{W}$, and the associated learning rates and mask matrix (see figure B.2). Inside the lateral Z layer, the neurons within each SEM unit are connected, and their connections have strong inhibitory weights, which should in theory be infinite. Additionally, self-interaction of neurons is avoided by setting the diagonal of the weight matrix to 0. The same is also true for the top layer X: The neurons cannot have any self-interaction, and the connections between the neurons are strong inhibitory, i.e., negative weight with high absolute values.

For the neurons in the Z layer some neighborhood has been defined: Neurons of SEM-units whose receptive fields are direct neighbors (by this definition, only the ones with common edges, but not with diagonal connections) may influence each other. The weights of these neurons are initially set to 0.

The other Z neurons, i.e., the ones whose receptive fields are further away from each other, do not influence each other directly, but only step-by-step, passing their values from one “neighbor” to the next one. These entries of $\boldsymbol{W}$ are set to 0.

For learning the weights inside the lateral layer without having a top layer defined (i.e., step 2 of figure 8.1), a learning rate of $\eta_W = 0.00001$ for 5000 learning cycles has been used, each of 400 milliseconds.

Finally, there are the weight matrix entries for the synaptic connections between lateral layer Z neurons and higher layer X neurons. Here there is an all-to-all connection possible, i.e., each neuron in the lateral layer is (possibly) connected to each neuron in the top layer. The initial values for these weights are again set to 0.

To learn the connections between the two higher layers and the lateral layer connections (i.e., step 3 of figure 8.1) again the same learning rate and time as for step 2 has been used.

Note that for learning the weight matrix $\boldsymbol{W}^+$ of the top layer containing populations of neurons in each WTA-unit, the same kind of learning can be used as in step 3. The reason is that still, all top layer neurons are connected to the lateral layer neurons - which are the synapses to be learned - while still all connections among top layer neurons are fixed, either to strong negative values (for neurons of different populations or WTA units), to zero (for self-interaction) or to some positive values (for neurons within one population, i.e., one WTA unit). Therefore the weight mask matrix is the same, except that there are some more neurons in the third layer, and thus the matrix gets bigger (in both dimensions).

8.4 Simulation results

In this section, the performance of the multilayered neural network model after learning the different synapses are presented, showing the learned weights, and their effects on the network's behavior. From this it should be possible to gain an overview over which of the previously mentioned input patterns can be learned or recognized using which kind of architecture and network complexity.

For each of the following network models, various tests have been made. As it might be difficult to understand each model at a a first sight, and to get an idea about the effects of each setting, some heuristics for each tested model are shown: The input (shown as the firing rate of the input layer neurons $\boldsymbol{y}$) is shown in comparison to what the network "thinks" about the input, i.e., a reconstruction of the firing rate, computed as the average of the firing rate (either ν_{ki} or ν_0) over the sampling time:

$$y_i' = \frac{1}{T} \cdot \sum_{t=1}^{T} e^{\ln \nu_0 + \sum_k z_k \cdot V_{ki}} \tag{90}$$

$$= \frac{1}{T} \cdot \sum_{t=1}^{T} \left(e^{\ln \nu_0} \cdot e^{\sum_k z_k \cdot V_{ki}} \right) \tag{91}$$

$$= \frac{1}{T} \cdot \nu_0 \cdot \sum_{t=1}^{T} e^{\boldsymbol{z}^\top \boldsymbol{V}_i} \tag{92}$$

8.4.1 Step 1: Feed-forward connections from input to lateral layer

The first kind of connections to be covered are the synapses connecting the input layer Y with the lateral layer Z, where the connections have always been restricted by the receptive fields, which has been shown as step 1 in figure 8.1.

The result of learning only such connections can be seen in figure B.3, which shows the weight matrix $\boldsymbol{V}$, containing all connections between neurons $\boldsymbol{y}$ of the input layer Y, and neurons $\boldsymbol{z}$ of the lateral layer Z.

Since the network is such that it can represent the input exactly, the exact values which should be reached after learning can also be computed analytically, knowing that the weights expected from theory would be

$$V_{ki} = \log \frac{\nu_{ki}}{\nu_0} \tag{93}$$

Additionally, there is some information available about the input images, i.e., about the patterns the network should learn, namely the size, and the distribution of "black" and "white" pixels, as well as their intensities:

$$Px = 36 \tag{94}$$

$$Px_{\text{black}} = 18 \tag{95}$$

$$Px_{\text{white}} = 18 \tag{96}$$

$$\text{black} = 0.9 \tag{97}$$

$$\text{white} = 0.1 \tag{98}$$

From the network settings, the null-cause is known, which makes it possible to compute the group firing rate of a group of neurons representing one SEM-unit in the lateral layer:

$$\nu_0 = 20.8 \text{ Hz} \tag{99}$$

$$\nu_j = Px \cdot \nu_0 = 747.2 \text{ Hz} \tag{100}$$

Knowing group firing rates, and having some additional information about the input patterns, one can compute the firing rates of the neurons $\boldsymbol{y}$ corresponding to (nearly) black, and white pixels in the input image:

$$\nu_{k,\text{black}} = \frac{\text{black} \cdot \nu_j}{Px_{\text{black}}} = 37.4 \text{ Hz} \tag{101}$$

$$\nu_{k,\text{white}} = \frac{\text{white} \cdot \nu_j}{Px_{\text{white}}} = 4.2 \text{ Hz} \tag{102}$$

When now applying the formula describing the weights of the synaptic connections to these values, it is possible to compute the "optimal" weight values, which are approximated during the learning phase:

$$V_{ki,\text{black}} = \log \frac{\nu_{k,\text{black}}}{\nu_0} \approx 0.6 \tag{103}$$

$$V_{ki,\text{white}} = \log \frac{\nu_{k,\text{white}}}{\nu_0} \approx -1.6 \tag{104}$$

After the learning process, the values of the synaptic connections approximate these values. For this first test, the initial weight matrix $\boldsymbol{W}$ for the connections among higher layer neurons $\boldsymbol{z}$ is used in its unlearned form.

It is then possible to make some tests, using these connections, and to observe what a network model with such settings would be able to recognize. This is done by using again the same input pictures as shown in figure 8.4, in this case especially (a).

As there are no connections between different input image subparts, or between the receptive fields, the results of these tests are quiet obvious. Having only adapted the feedforward weights $\boldsymbol{V}$ it is possible to reconstruct entire images as shown in figure 8.5, i.e., to recognize each single subpart or pattern of the input, and therefore to recognize input (a) of figure 8.4. It is also possible to "correct" some small errors in the input image, like missing or swapped pixels.

On the other hand, it is not possible to do any kind of completion of the entire input image if some subparts are missing (as shown in figure 8.6), swapped (as in figure 8.7) or ambiguous, as there is no connection at all between different subparts, and thus no information can be exchanged among them.

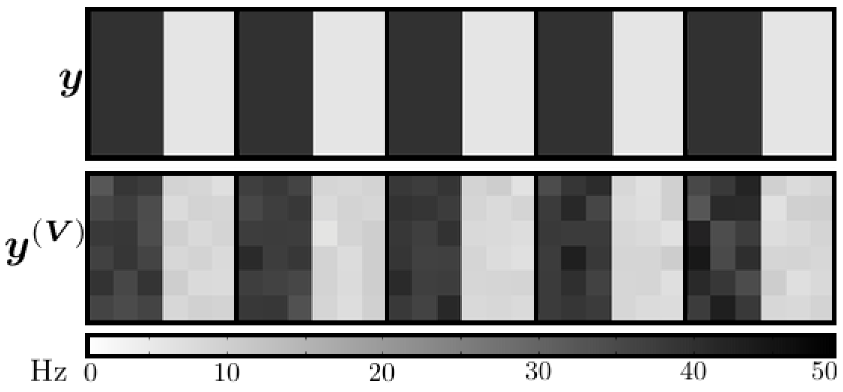

Figure 8.5: Internal reconstruction of the learned input image. The upper picture shows the firing rates of the input neurons $\boldsymbol{y}$. The picture below is a representation of what the system internally thinks it sees. In this case, the input is recognized.

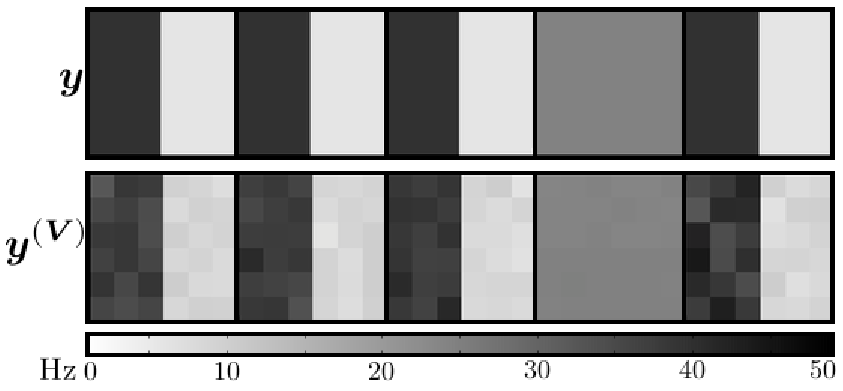

Figure 8.6: Incomplete input image and the internal representation of what the system intends to see (below)

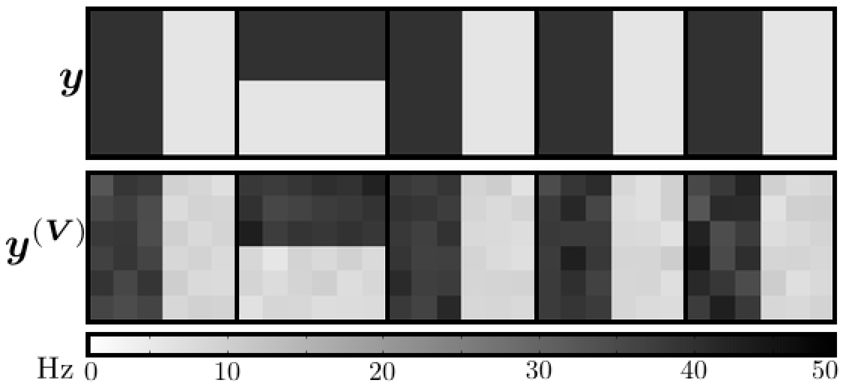

Figure 8.7: Firing rates of the input neurons when representing an image with one non-matching subpart, and the internal reconstruction of the input inside the model.

8.4.2 Step 2: Lateral layer connections

In a second step, learning was extended to the lateral layer. In this next step, also the weight matrix $\boldsymbol{W}$ defining the synapses between neurons of the same, lateral layer Z, should be learned. In this case, as described in the previous section, learning was restricted to this lateral layer. Thus, still only the input layer and the lateral layer are used, but no connections to the top layer are defined or learned, as it is shown in step 2 of figure 8.1.

For testing the learned weights, again different input patterns were used as input to a network model, defined by the feedforward weights $\boldsymbol{V}$ learned before, and additionally by the lateral layer connections contained in the currently learned weight matrix $\boldsymbol{W}$.

This time, it is already clear that the completion of the single patterns is done, as this problem is already solved using only the feedforward weights $\boldsymbol{V}$, which is also used in this test.

In addition to the previously tested model, this network has now also some additional communication among neurons that are not responsible for the same receptive field. Thus, there is some information exchange about the pattern content of the "neighboring" input image parts.

Therefore, if some receptive field does not give much information about the contained pattern, like it is shown in figure 8.8, the information about the patterns next to it is taken into account. This means, that if there are some ambiguities or "holes" in the image, i.e., some parts of the image cannot be recognized as any of the learned patterns, these parts are internally "completed" or filled, by looking at the nearby image parts, and inside the model the input is recognized as one of the learned patterns, i.e., the model "thinks" of a complete input. But still this "completion" is only very lightly visible.

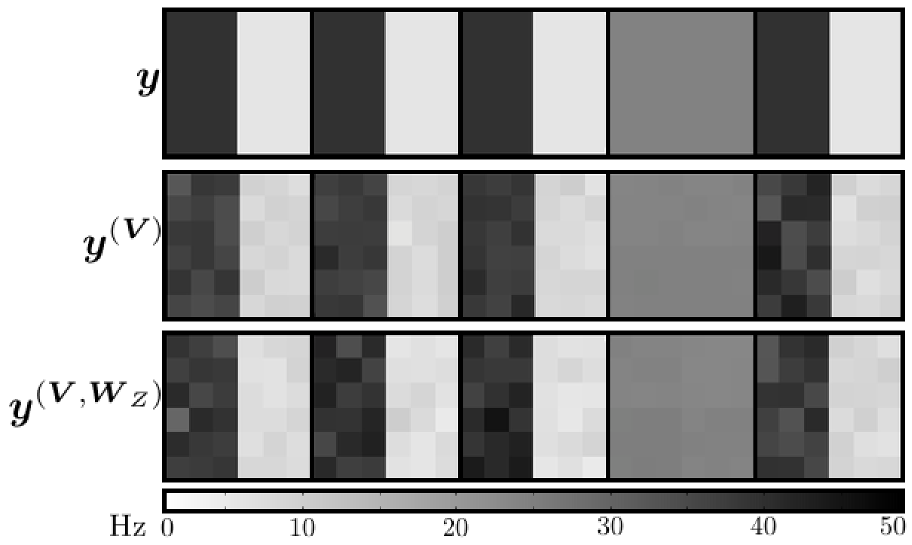

Figure 8.8: Firing rates of the input neurons when seeing an incomplete input image (with one ambiguous subpart), and the internal interpretation of the system of this input, knowing only the feedforward weights (middle), and then also the lateral connections (bottom).

Figure 8.9 shows the average firing rates over 1000 trials, each of one second of length, of the four lateral layer units (the background hypothesis unit which is not modeled is not shown here). When looking at the means of the firing rates one can see that the firing rate of the unit which would recognize the pattern given as input to the rest of the image (but not to this subpart) is slightly higher than the firing rates of the other lateral layer units. But still the errors is relatively high, leading to the conclusion that this observation is still not very reliable.

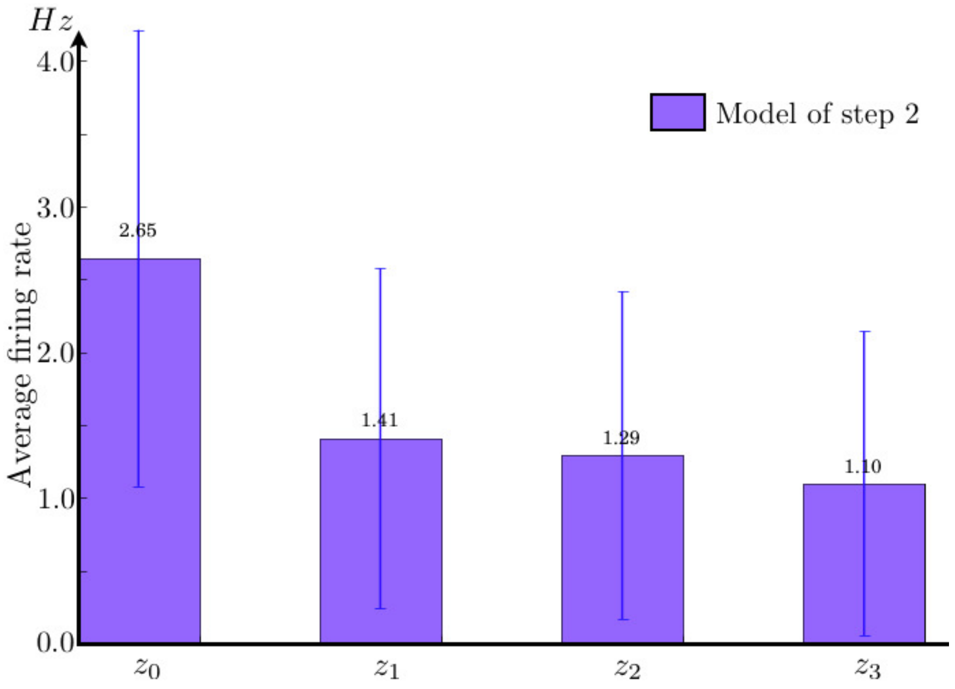

Figure 8.9: Average firing rates over 1000 trials, each of 1 second, of the lateral layer units having their receptive fields on the "gray" input image part which is not set, i.e., which represents the average over all learned input patterns.

In other words, this means that image or pattern completion is "nearly" possible, using only these two layers, and some restricted connections between neighboring receptive fields. To be sure about some pattern (i.e., to get a clearer reconstructed image with higher contrast), more information of neighbors would be required. In order to achieve this, the neighborhood could be expanded, by increasing the input image size, or by expanding the definition of the "neighborhood" to include more than the direct neighbors.

Another situation can be defined as having an image where one sub-part does not match the other ones. In this case the system does not recognize this input as one of the learned images (as a whole), but it recognizes the different subparts of the image, as in the previous step 1.

The reason why the "wrong" pattern is not detected can be found when looking more precisely at the learned values in the connection weight matrices $\boldsymbol{V}$ and $\boldsymbol{W}$: The weights among units inside the lateral layer are relatively low. Therefore the connections between input layer and lateral layer neurons (i.e., respectively the direct input from the receptive field to some neurons) overweights the information about neighboring units.

This phenomenon could possibly be overcome again by extending the definition of neighborhood and using much larger pictures (having more patterns, and a larger neighborhood) than in this experiment.

8.4.3 Step 3: Adding a third layer

The previously explained step already allows for some communication among lateral layer neurons $\boldsymbol{z}$, whose receptive fields are next to each other. In this third step, this communication and (indirect) information passing about the content and the structure of the entire image as a whole is further strengthened.

This additional information is encoded as some "global" knowledge or interpretation of the input image (as a whole), i.e., the third layer which is added in this step tries to decide to which (learned) category the input image belongs. Based on the (learned) knowledge about how an input image should look like, it tells the system, which of these known and learned structures the input image belongs to.

In this context, the receptive field of these top layer neurons $\boldsymbol{x}$ lies on the entire lateral layer, meaning that each of the top layer neurons $\boldsymbol{x}$ is synaptically connected to each neuron $\boldsymbol{z}$ of the lateral layer. Note that at each step learning starts from scratch, i.e., in this case also the feedforward and the lateral weights are learned again, but this time including also the top-layer weights.

After learning, each single input pattern activates the corresponding neurons in the input layer. The neurons of one input pattern together stimulate some lateral layer neuron of this receptive field, causing it to become active, while the other lateral layer neurons of this unit get inhibited and thus repressed. The active lateral layer neuron additionally stimulates the neurons responsible for the same input pattern of the lateral layer units having a neighboring receptive field, and it additionally stimulates the top layer neuron which recognizes this pattern.

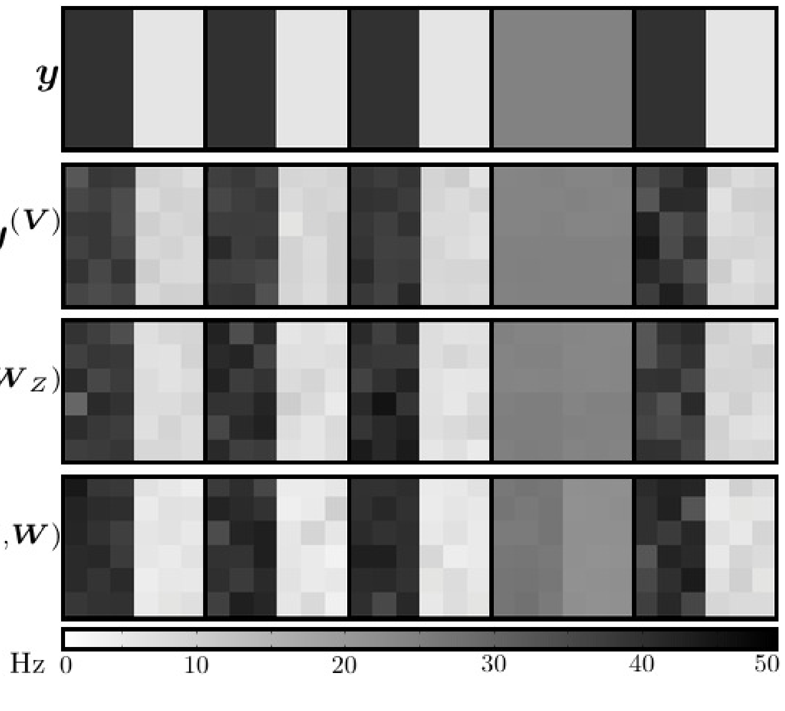

Figure 8.10: Firing rates of an incomplete or ambiguous input image (top), and internal reconstruction of the systems intention with increasing number of learned synapses below. In the third step the incomplete part is "filled" with the pattern which matches the rest of the image.

Therefore, as the synapses between the two higher layers are both - bottom-up and top-down - information of the top layer flows back to the lateral layer, and therefore some additional information about the content of the entire image is given to all of the lateral layer neurons directly by the top layer, and additionally in an indirect manner by passing the information from “neighbor” to “neighbor”.

The learned images are again recognized by the model, as well as images having some small errors. When looking at the images to be completed, one can see that the missing subpart is recognized much better now, when using also top-level information, as shown in figure 8.10: The top-down information of the third layer further strengthens the interpretation of the missing image pattern as the one which “matches” the hole. This is what the model should do, as in this simple case the system has been trained to “know” that an input consists of five same patterns, not having an incomplete part among them.

The influence of the top layer gets even better visible when looking at figure 8.11. It shows again the average over 1000 trials of tracking the firing rates of the lateral layer units which get “gray” input. In the plot, the previous setup of step 2, having only lateral layer connections, is plotted in violet, and it is compared to the current model, where also top-down information is available to these units, shown in blue. It is shown that the firing rate of the unit which “recognizes” the missing image part (denoted as z_0) increases with the introduction of the top layer.

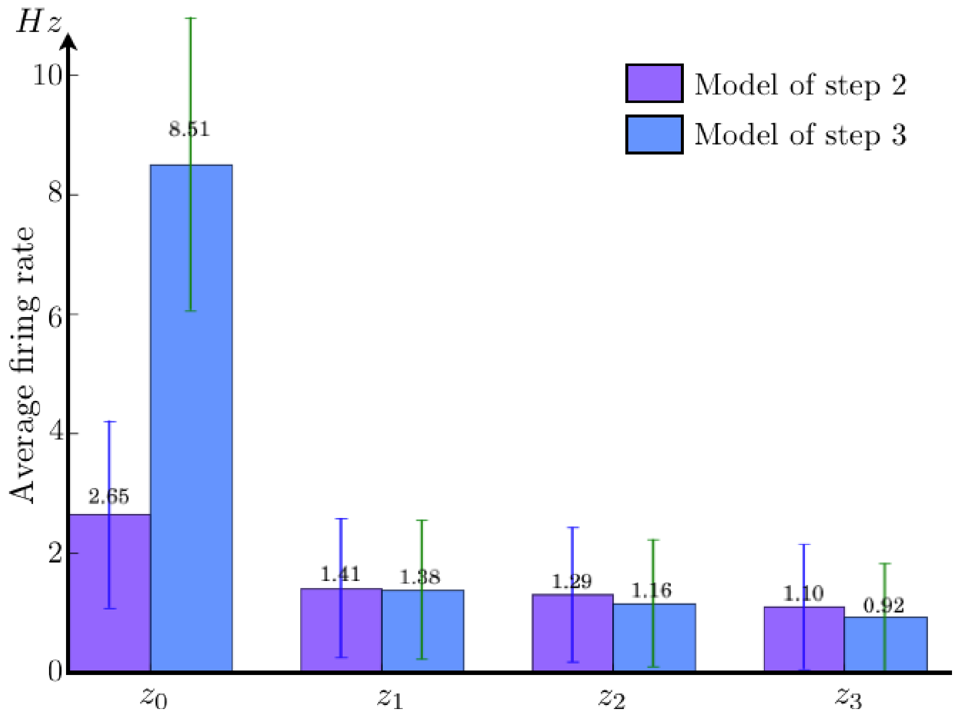

Figure 8.11: The average firing rate of the lateral layer units don’t getting any direct information about the “input” pattern by their receptive fields. The left bars show the firing rates of step 2, when using only lateral layer connections, and the right bars indicate the current rates, i.e., having the additional top layer information. The increase in the firing rate of the unit which recognizes the “missing” pattern is visible very well.

The effect of applying this model to the image with one non-matching or “wrong” subpart are presented in figure 8.12. Still the model is not able to detect the “wrong” pattern.

Again note that it is not really clear what the system should do and whether it shout internally exchange the not-matching pattern. But as the model has learned that a “correct” input image consists of five equal parts it might be possible that it could detect the ambiguity or “problem” in such an image.

Due to the still low firing rates of the units when not getting direct input (but only lateral and top-down information), and also because the system still is not able to detect the “problem” in the image with the non-matching pattern, the model is further expanded, such that it should then be able to do even better or stronger image completion and recognition.

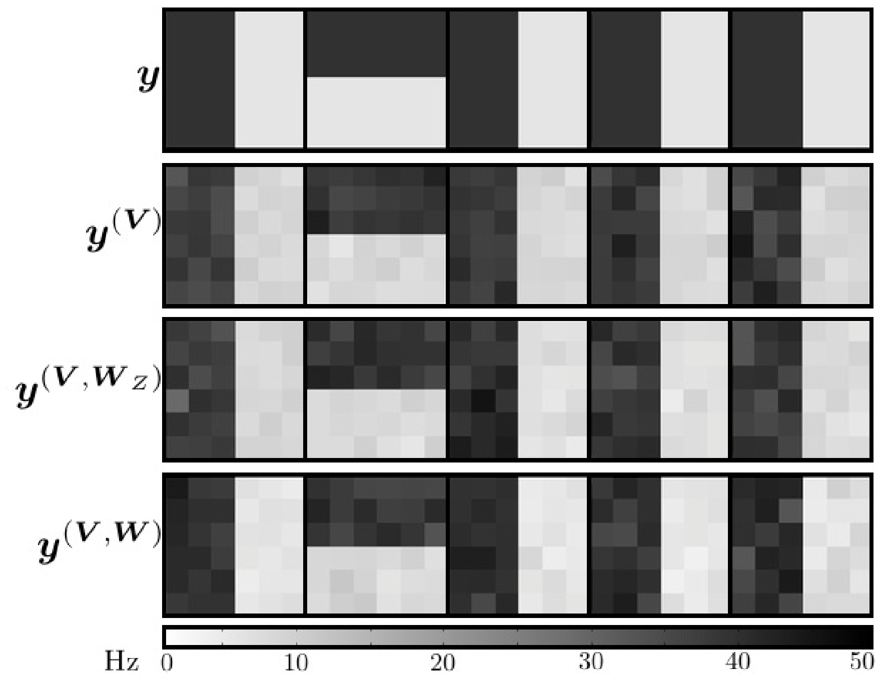

Figure 8.12: Input firing rates and the systems internal behavior (thought) when getting this input. In step 3 the ambiguous situation is slightly visible (a slight shadow of the “matching” pattern in the picture on the bottom).

8.4.4 Step 4: Expanding the third layer

In this last step, the model's top layer is further expanded. The single WTA-like units are now replaced by small populations, each one consisting of $L = 5$ units. As explained before, the single units inside one such population stimulate each other, while the different populations inhibit all other populations, resulting again in some WTA-like structure.

This means that in the weight matrix the right upper and the left bottom parts are now learned, while the right bottom part remains unchanged: This is the part which has been initially set to guarantee that the model behaves in the desired WTA-like way:

After learning this model, again all lateral layer neurons $\boldsymbol{z}$ which are responsible for some specific input pattern stimulate the same top layer neurons $\boldsymbol{x}$ (and get feedback from them). Among the neighboring SEM-units, the neurons responsible for recognizing the same input pattern stimulate each other, while all others are inhibited, as the model learns that all input parts should contain the same pattern. In the top layer X, neurons inside one population are set to stimulate each other while the other top layer neurons are inhibited. The single weights have about the same strength as the connections in the previous test, when using the single WTA-like neurons in the top layer.

This implies, that the influence from the top layer - and thus the opinion about the entire image as a whole is now much stronger, and the input that the lateral layer neurons $\boldsymbol{z}$ get from the top layer Z has a higher influence on their behavior and firing patterns than before, meaning that now the top layer's prior of the top-down information has been changed by the model itself.

The effects of this phenomenon can be seen when looking at figure 8.13: The images with the missing subparts are now internally completed entirely, i.e., the influence of the other patterns has grown again, and the system internally thinks that it is presented the complete (learned) picture.

Even though the difference to the previous model is already visible in when comparing what the model "thinks" about the input picture, again the firing rates of the units which do not get direct external input are compared in figure 8.14. Now the model of step 3 (shown in blue) is compared to the current one with the expanded top layer, plotted in green. The increase is clearly visible, as now the firing rate is about as high as the one of the units with direct external input.

But the influence of such a powerful top layer is visible even better when now looking again at the test images of category (c), where one part of the image has a different pattern than the rest, as shown in figure 8.15. Now the network clearly detects that one of the patterns does not match the remaining image, and in the reconstruction it is visible that the model partially completes the "correct" image. This means that in this setting the model is able to recognize internally the ambiguity of this input image, which is shown by the internal firing pattern.

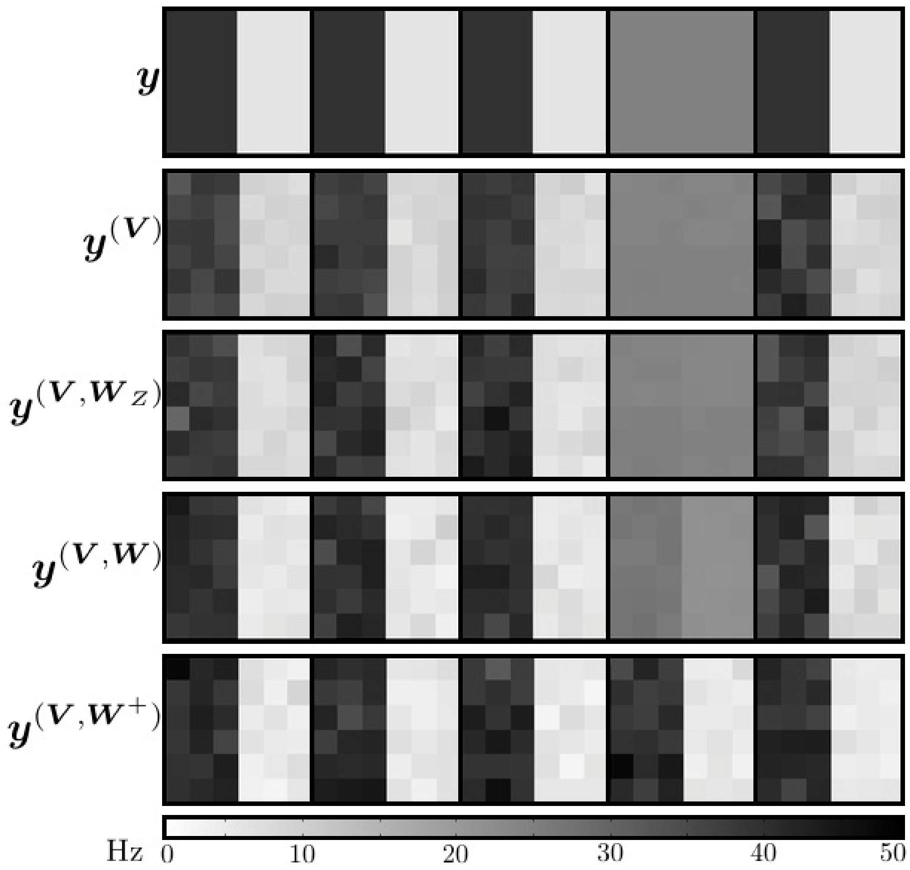

Figure 8.13: The firing rates of the input neurons (top), and what the system internally thinks it is shown, in the order of the increasing number of synapses in the model: With lateral connections of step 2 a “shadow” of the completed part is visible, which is strengthened when adding the third layer in step 3. In this fourth step the system internally recognizes now the image as the learned one, and therefore it is able to internally complete such an image.

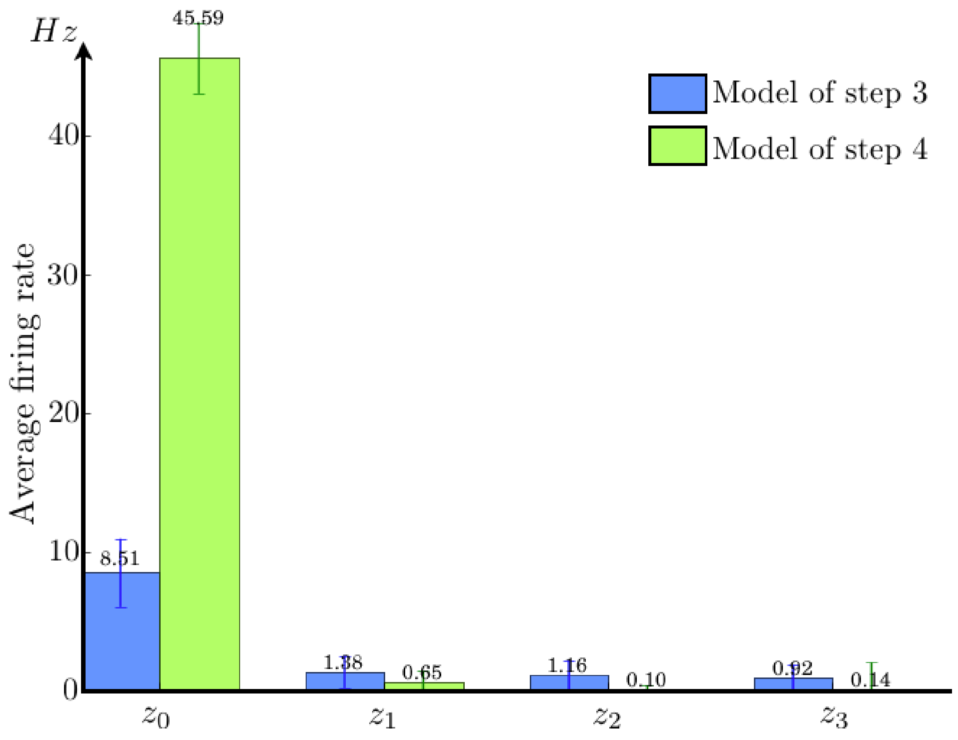

Figure 8.14: Average firing rates of the units without direct external input, comparing the model of step 3 (in blue) to the current model with the expanded top layer containing now populations of five units each (in green): The increase in the firing rate of the unit which responds to the "missing" pattern is clearly visible.

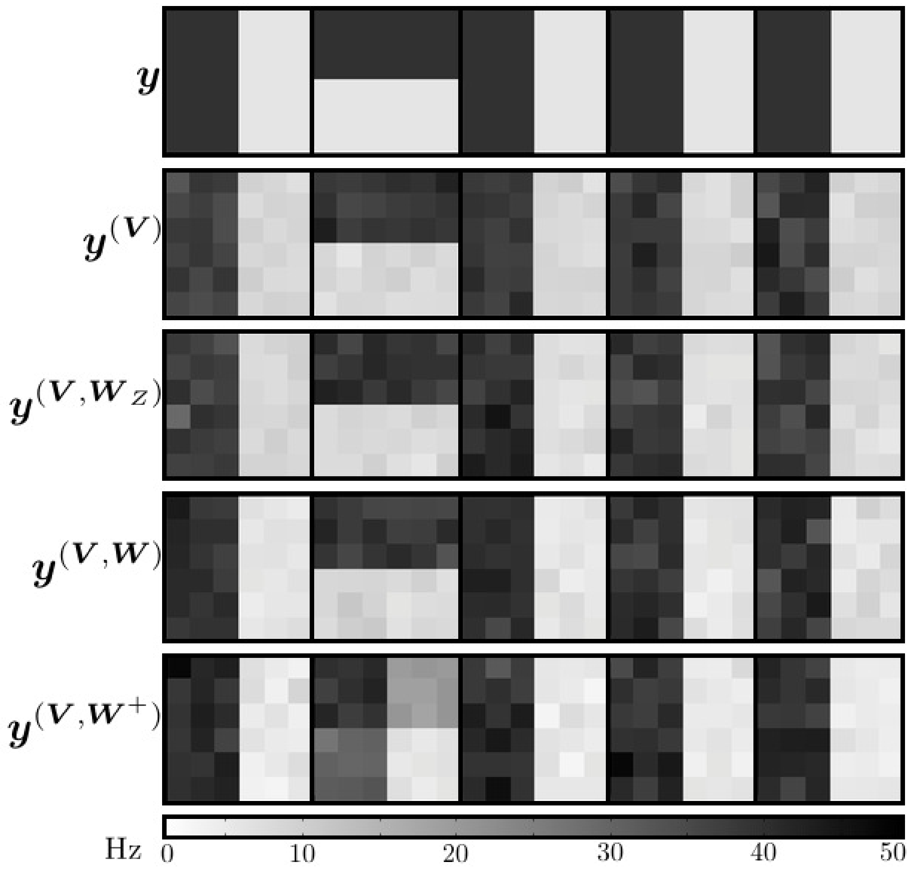

Figure 8.15: The firing rates of the input neurons when being confronted with an image where one of the parts does not match the other ones. Even when using the top layer of step 3 only a very slight shadow of the matching pattern is visible in the "ambiguous" part, while in step 4, when now having an expanded top layer, the system internally is able to detect the ambiguity, and it realizes that the given part of the image is not consistent with what it has learned to be a "correct" input image.

8.5 A symbolic version of the experiment by Lee

Lee et al [45], [43], [44] developed a hierarchical model of vision based on biological findings from experiments testing input and illusory contour completion in the visual system of animals (see section 2.3).

In this section, the new sampling architecture presented in the previous sections is applied to a symbolic and strongly simplified version of this model by Lee et al, and some of the results from the biological experiment are shown to be well covered by this simple model.

8.5.1 Experimental setup

In the biological experiment by Lee and Mumford, summarized in section 2.3, the processes going on in the brain when “completing” an illusory contour image were observed, as shown in figure 8.16 (the original image of the biological experiment was shown in figure 2.4).

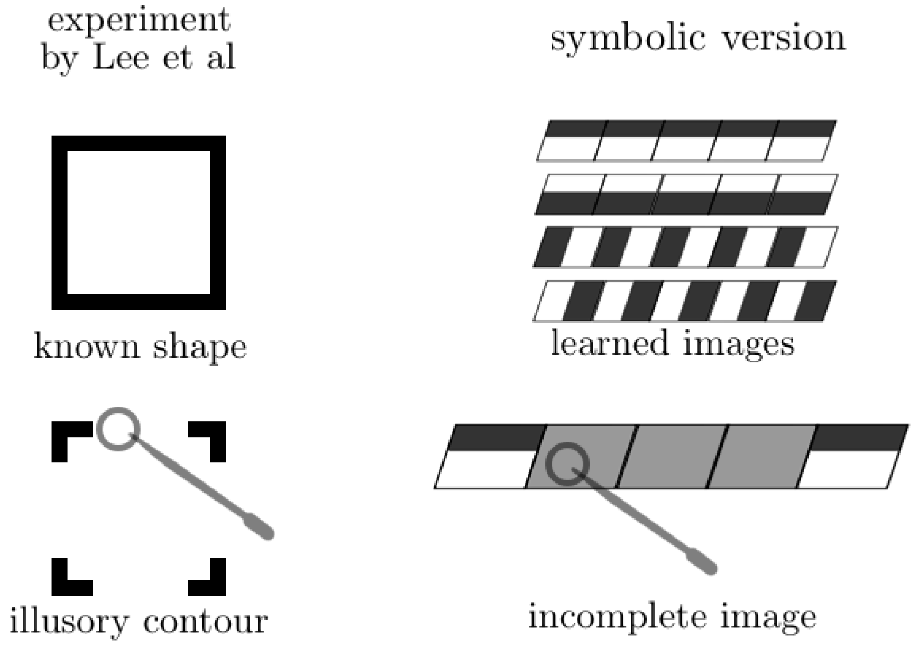

Figure 8.16: Symbolic version of the input images: Illusory contours experiment on the left (figure adapted from Lee et al [45]), where a neuron has its receptive field on that part of an illusory contour image where there is no visible contour, and the input images for the symbolic version examined here on the right hand side. The upper part of the figure shows the “complete” images. Below some image parts are ambiguous, which in the symbolic version is realized by averaging over all pattern inputs. In both cases neurons are observed, which have their receptive fields on an “incomplete” or “missing” image part.

The idea of “hierarchical” contour completion derived from biological experiments is simulated in a symbolic and strongly simplified version by means of the network model presented in the previous section. Here this model will further be expanded, leading to a biologically slightly more realistic setup, having alpha-shaped additive PSPs, synaptic delay and relative refractory mechanisms (see section 8.5.2). The results of the experiment are presented and compared with the outcomes of the biological experiments by Lee et al.

The focus of the experiment is to observe the spike response of neurons during different stages of the experimental protocol: Before the onset of the stimulus, they get completely ambiguous input, consisting of the average over all learned input images. After stimulus onset, i.e. after 500 milliseconds, an “incomplete” image, where some subparts still are ambiguous, is given as input to the network, as it is shown in figure 8.17.

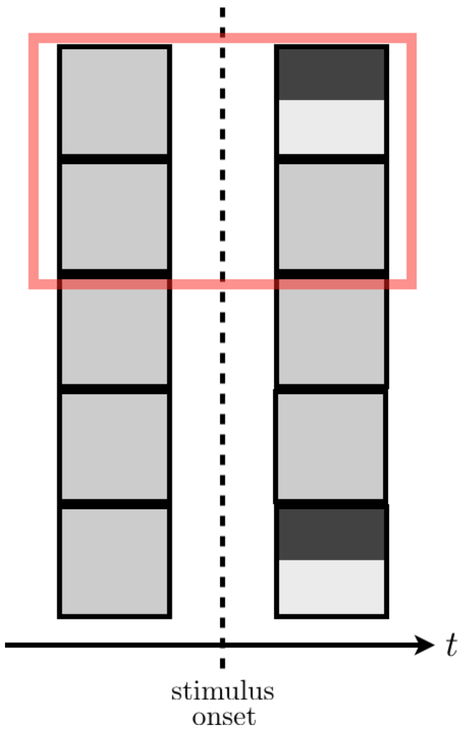

Figure 8.17: Experimental setup: Before stimulus onset, Poisson spike trains with a uniform firing rate are resented to the network (i.e., images representing the average over all learned images). And after the stimulus onset the network gets Poisson input which encodes an incomplete image: Only two of five subparts of the image show a distinct and learned pattern while the others still show the average (gray) picture. These gray images display the mean of all learned patterns and, hence, do not prefer any interpretation. The subparts of the image highlighted with the red box will be looked at in more detail when presenting the results.

This is realized, using generally a network structure which is very similar to the one of the previous experiments, and which is sketched in figure 8.18: The input units are connected to some of the lateral layer neurons, building up extended SEMs (i.e., SEMs [56] with an additional background hypothesis, as it was described previously in section 5.3), where the connections depend on the receptive fields. All lateral layer neurons are additionally connected to the entire top layer, which consists of “populations” of units.

For the experiment itself, the model which was before trained on the different complete input images, was first given ambiguous (gray) input for 500 milliseconds. After the onset of the stimulus (which consisted in giving an incomplete image as shown in figure 8.17 as input), the network response was observed for another 500 milliseconds.

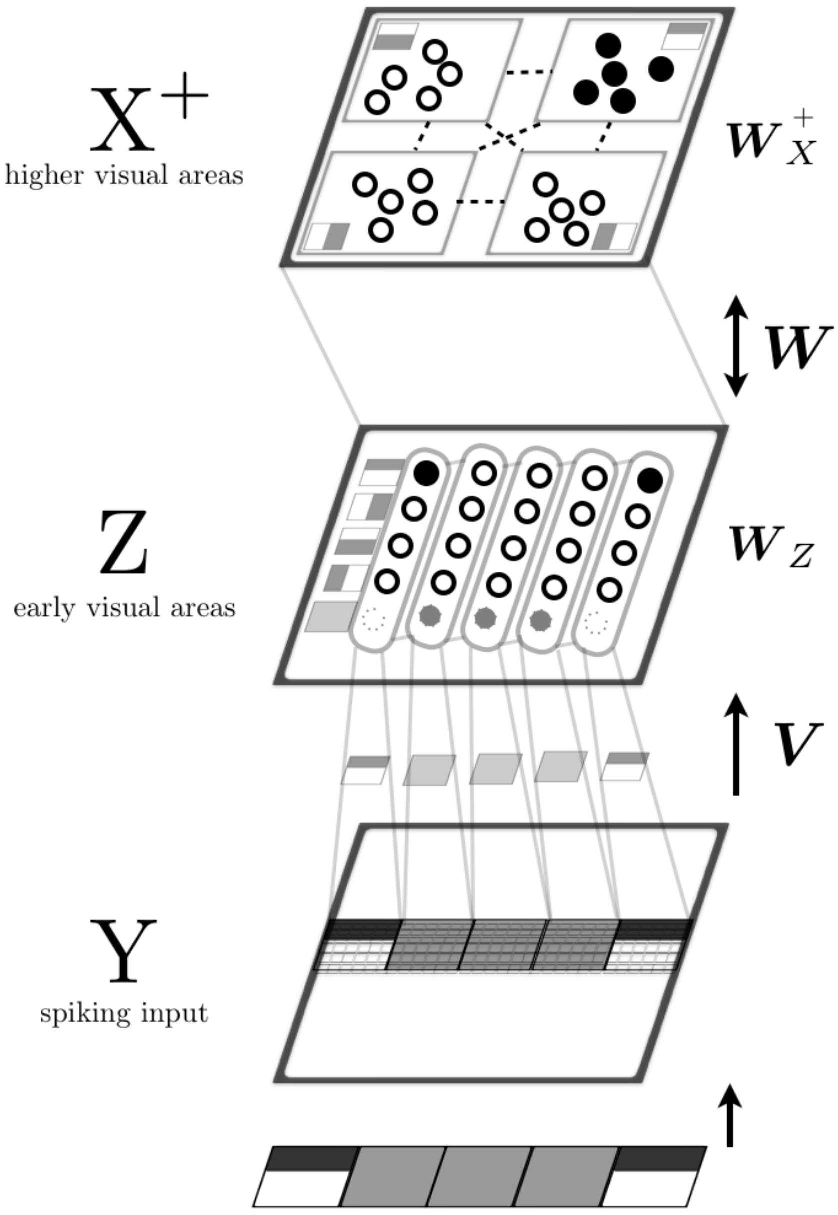

Figure 8.18: Experimental setup along with the input after stimulus onset (while before the onset the model gets entire "gray" input images). The connections between the units are encoded in the feedforward weights $\boldsymbol{V}$ and the feedforward-feedback connections $\boldsymbol{W}$. Five distinct receptive fields, and extended SEMs (being SEMs [56] with additional background hypotheses; see section 5.3) in the lateral layer are used. Each extended SEM has one unit "specialized" for each pattern. In the top layer, populations of five units each are used, where again one population is specialized to one distinct pattern (this time a global view of the image). The synaptic weight matrices are taken from the learning experiments in section 8.2.

The idea behind this experiment is the following: Inputs representing unambiguous patterns in image subparts are the most “realiable” information to the network, which also reflects in the lateral layer neurons. For image subparts which get ambiguous external input, still some information can be inferred from the remaining (distinct) image parts: Neighboring extended SEMs, and to top layer units provide top-down information. The influence of the neighboring extended SEMs is quiet low, but still there might be some visible effect. The information passed to the top layer is not entirely unambiguous (as only two of five structures have distinct external input in their receptive fields). After some time, the top layer units should “adapt” their firing rates to this situation, then causing also the lateral layer units with ambiguous (gray) input to change their firing patterns. In this way, the image is internally interpreted as being one of the learned ones, meaning that some kind of pattern completion is performed inside the network through top-down information and prior knowledge of how a distinct, unambiguous (learned) image looks like.

8.5.2 Details to the network model

As shown in figure 8.18, for this experiment the input picture was subdivided again in five subparts, each being a quadratic image of 6×6 pixels, which results in a total number of $N = 5 \cdot (6 \cdot 6) = 180$ pixels in the input picture. The same number N of input layer neurons $\boldsymbol{y}$ are required, to represent one pixel each.

The connections $\boldsymbol{V}$ between the input layer units $\boldsymbol{y}$ and the lateral layer units $\boldsymbol{z}$ are such that each group of lateral layer units $\boldsymbol{z}$ (i.e., each extended SEM) has its receptive field on one distinct subgroup of input pixels $\boldsymbol{y}$. The values of the synaptic weights are the ones learned in the previous section.

The lateral layer Z consists of five extended SEMs, each containing four units, which represent the four distinct patterns which are learned The lateral layer neurons within one group inhibit each other. Directly neighboring extended SEMs (i.e., the ones having their receptive fields next to each other) are connected with plastic synapses. These weights within the lateral layer, as well as all weights between the lateral and the top layer have been learned in the previous section.

The top layer X is connected to the entire lateral layer Z. It is subdivided into four disjunct populations, each consisting of five units. The units inside such a population excite each other, while among the different populations there is strong inhibition.

The difference to the previous experiment lies in the types of neurons used. Instead of using the exact sampler, now the PSPs are alpha-shaped and additive, having a rise time of 3 milliseconds, and a falling time of about 18 milliseconds using $psp(t) = 5 \cdot (0.7^t - 0.5^t)$. The refractory mechanism used for this experiment was relative, as shown in figure 6.4. An additional synaptic delay of 3 milliseconds was set globally, to all the synapses encoded in the matrix $\boldsymbol{W}$.

8.5.3 Results

The input to the network is shown in figure 8.19, showing the spike trains of the input units of the first two image parts (the remaining three ones are quiet similar to the respective ones of these). The firing behavior is Poisson-distributed with equal firing rates in all input neurons before the stimulus onset. After stimulus onset it remains like this in the second (also in the third and fourth) input part, which still get ambiguous input, while in the first (and fifth) picture part the firing behavior changes strongly, resulting in a much more structured pattern after this first 500 milliseconds: It is shown that the firing rates of the units getting a "dark" pixel input rises, while the units getting their input from a "bright" pixel fire much less often.

Figure 8.20 shows the spike trains of the units of the lateral layer Z, and of the top layer X^+ in response to this time. The upper part of the plot shows the states of the lateral layer neurons, where the different extended SEMs are subdivided with lines. In each such structure, the top one of the four plotted neurons has "learned" the pattern which is given as input, and is highlighted in green. The uppermost unit is the one with the receptive field on the first image part, the one on the bottom gets the last image part as input. On the left the input patterns after stimulus onset are shown (the pattern in the first and fifth image part, and an average gray pattern in all other parts).

In the lower part of the plot the top layer neurons are shown, consisting of four populations (one specialized for each "complete" input picture) of five units each. The input patterns shown on the left are the ones for which the specific population has specialized, again with the one which is expected to react to the given stimulus highlighted in green.

Before stimulus onset the network randomly jumps between different interpretations of the ambiguous input. This behavior is caused by the MCMC dynamics, dominated in this situation by the prior knowledge about complete and unambiguous images.

After stimulus onset, the lateral layer neurons of the first and last input part "react" almost instantaneously to the new stimulus. The short delay in the reaction in some trials is caused by the influence of the top-down information which can currently interpret the (previous) ambiguous input as some other prior known image.

After a delay of about 50 to 70 milliseconds, the top layer units change their firing pattern to respond to the input mediated by the $\boldsymbol{z}$ neurons. And even later also the lateral layer units of the image parts which get ambiguous external input are provided the top-down information and show their "reactions".

It is also well visible that the firing rates of the lateral layer neurons of the first and last image parts are much higher than the ones of the other parts which only get feedback information from their neighboring units and from the top-down information.

For a decent analysis this experimental protocol was repeated 100 times. Figures 8.21 and 8.22 show the average firing rates of the neurons associated with the given input, i.e., the neurons marked in green in figure 8.20.

In figure 8.21 the average firing rates are color-coded, showing the average firing rates of the neurons $z_1, z_5, \ldots$ and $x_1, \ldots, x_5$ (population average in case of the top-layer neurons), which are associated with the presented input pattern, i.e., the ones which were highlighted in green before in figure 8.20.

Before the stimulus onset (indicated with the dashed line), the firing rates of all shown units are quiet low. After that, the firing rates of the associated units of the first and last input part rise very fast. Then the average firing rate of the corresponding top layer units increases - but slower and with an additional delay of about 50 to 70 milliseconds. They provide additional top-down information to all lateral layer units. After that, the firing rates of the associated lateral layer units of the second, third and fourth input picture parts (i.e., the ones which only get ambiguous input from their receptive fields) rise.

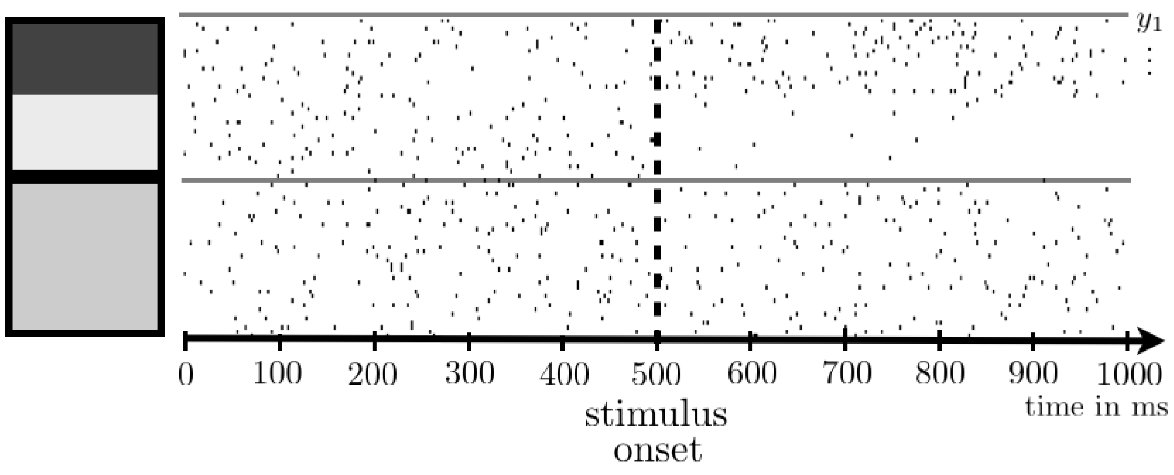

Figure 8.19: Input units of the first two image parts: Spike trains of the neurons for one second. Initially, Poisson spike trains with a uniform rate are presented, while after stimulus onset (at 500 ms, indicated with the dashed line) the first (upper) image subpart encodes the pattern shown on the left, while the second (lower) part still encodes the uniform and ambiguous (gray) input.

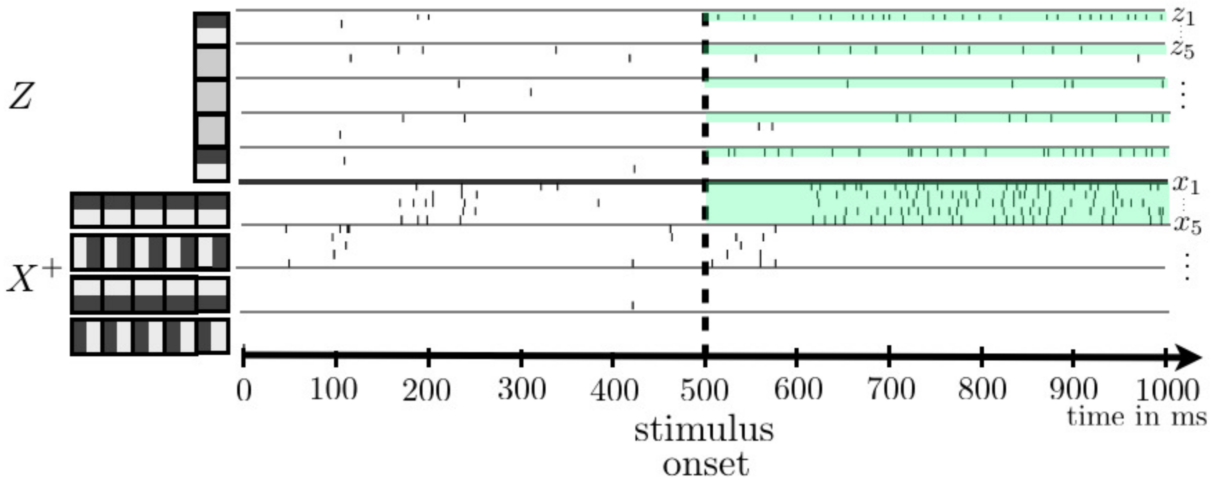

Figure 8.20: Spike response of the lateral and top layer neurons. The upper part of the plot shows the lateral layer. The four extended SEMs are divided by lines. In each SEM, the unit associated with the given input pattern ($z_1, z_5, \dots$) is indicated in green. Below, the five units of each population of top layer neurons are shown. Again, the units which are expected to "react" on the given input $x_1, \dots, x_5$ are highlighted in green. Before stimulus onset the network jumps between the different possible interpretations of the ambiguous input. After the onset, z_1 and z_{17} respond almost instantaneously to the new given input. After a delay of about 70 milliseconds also the top layer units $x_1, \dots, x_5$ adapt to the mediated input. Then, also the remaining lateral layer units ($z_5, \dots$) show their response to the top-down information they get, but with a lower firing rate than the units z_1, z_{17} which get directly the unambiguous external input.

The influence of the lateral layer connections is visible when comparing the second and fourth units of the lateral layer to the one in the middle: The firing rates of the outer two units is slightly higher, as they get additional excitation by their direct neighbors (which are the ones getting direct unambiguous input).

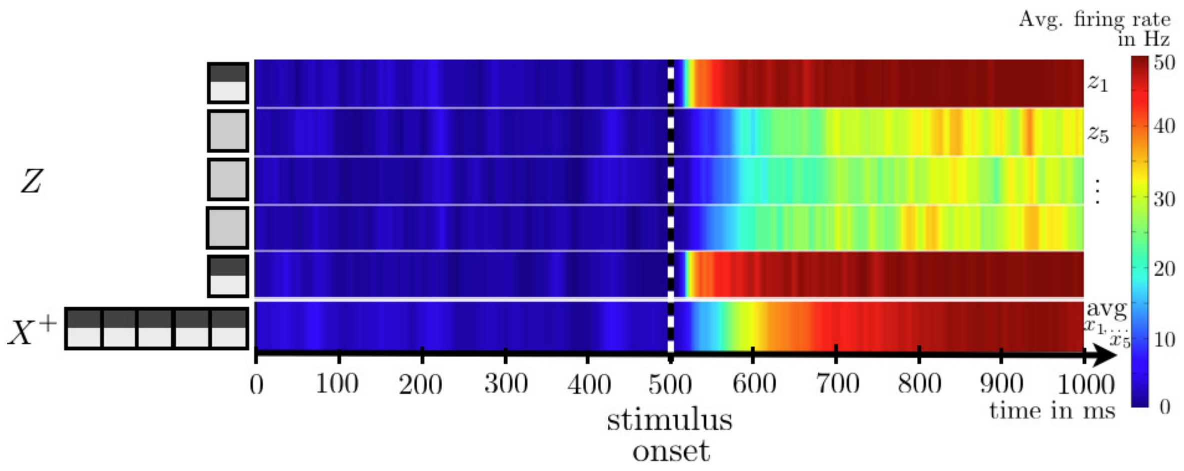

Figure 8.21: The average firing rates over 100 trials of the five lateral layer neurons $z_1, z_5, \ldots,$ which are associated with the presented input pattern. The bottom line shows the average firing rate in the population of top layer units which is expected to recognize the presented input pattern, i.e., the average firing rate of the units $x_1, \ldots, x_5$. The lateral layer neurons with unambiguous input (i.e., a distinct pattern) in their receptive fields respond first and have the highest average firing rates. The top layer neurons follow with a delay of about 50 to 70 milliseconds. Only after the top layer units have modified their firing patterns and provide top-down information to the lateral layer units with ambiguous direct external input, also these units adapt to this situation.

To better visualize the firing response after stimulus onset, data from figure 8.21 is replotted in figure 8.22 for three neurons which are representative for the network dynamics. The red line shows the average firing rate of the lateral layer neuron z_1 associated with the stimulus pattern of the first extended SEM, i.e., having a receptive field with unambiguous pattern input. The green line is the associated neuron z_5 of the second extended SEM, which gets ambiguous gray input, while the yellow line (shown for completeness) is the average firing rate of the five top-layer neurons $x_1, \ldots, x_5$ (population average).

One can see that before stimulus onset, i.e. when the entire input to the model is ambiguous (gray), the firing rate is relatively low, as expected. In this period, the fluctuations of the average firing rate in the top layer reflect also on the lateral layer neurons, which is caused by the underlying MCMC dynamics, in this case again dominated by the prior knowledge of how complete input images should look like. The dashed line indicates the stimulus onset at 500 milliseconds.

Immediately after the onset, the firing rate of the unit z_1 (which gets unambiguous distinct external information) rises rapidly. Soon after this, with a delay of about 60 milliseconds, also the average firing rate of the corresponding top layer population (i.e., neurons $x_1, \ldots, x_5$) increases.

The average firing rate of the lateral layer unit z_5 starts to rise only when the average firing rate of the “neighboring” unit z_1, i.e., the one with direct external input, has reached about half its height (indicated with the red dotted line). After that, z_5 gets unambiguous feedback information from its neighbor and its firing rate rises slightly, but then further increases as also the top layer units' average firing rate reaches a higher point.

This indicates that the units without direct information about the input (like z_5) get some - but only few - information by other units in the same lateral layer (e.g., by z_1), and then they get delayed top-down information (e.g., $x_1, \ldots, x_5$) which strongly influences their firing behavior.

The red and the green dotted lines indicated the point in time when the different lateral layer units reach half of their maximum firing rates. When now comparing this shift or delay in the onset time with the outcomes of the experiments of Lee et al in figure 2.5, one can see some similarities: In both cases, the lower layer units react soon after the stimulus onset to unambiguous (line) input, but top-down information is required in order to perform pattern or contour completion. The need for this feedback information causes - again in both experiments - a delay in the response to the “incomplete” or ambiguous input, compared with input consisting of “complete” and unambiguous lines or patterns. Additionally, in both plots it is clearly visible that the response is higher with unambiguous and complete input.

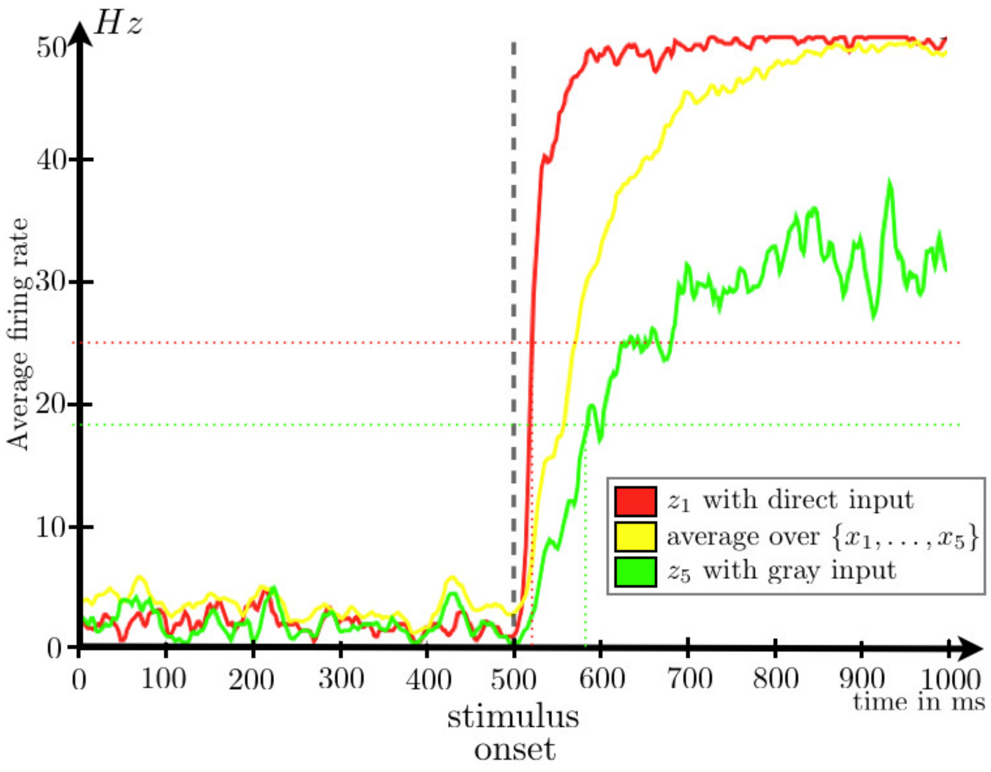

Figure 8.22: Average firing rates of representative lateral and top layer units: Lateral layer unit z_1 gets directly unambiguous external input, and is associated with this input pattern (Its average firing rate is indicated by the red line). Another lateral layer unit z_5, associate with the same pattern, gets ambiguous external input, but is associated with the same pattern (The green line shows its average firing rate). For completeness, the average firing rate of the top layer neurons $x_1, \ldots, x_5$ are shown (indicated by the yellow line).
After stimulus onset it is visible that first the average firing rate of the lateral layer unit z_1 (with the unambiguous external input) rises, and after this the average firing rate of the top layer units $x_1, \ldots, x_5$ follow. Only after an additional delay, the lateral layer unit z_5 (with ambiguous external input) gets the indirect top-down and lateral information and its average firing rates increases.
The solid dotted lines indicate the delay, which is measured at half of the maximum height where the firing rate of the corresponding neuron saturates. When comparing the dotted vertical red and green lines a delay of about 60 milliseconds is visible in the reaction time.
When comparing this figure with figure 2.5, it can be observed that some properties are similar to the findings by Lee et al, where a similar situation (namely a neuron's reaction to a square contour and to an illusory square) is measured biologically: In both cases the unambiguous direct information about a line or a pattern causes a immediate (or soon) response, while there is some delay in the response when top-down information is required. Additionally, in both cases the response to an unambiguous image is higher than to ambiguous input.

8.6 Conclusion

Now it has been shown on which probabilistic model the multilayer neural network model is based, how it is set up and learned, and also the results of learning and testing it (i.e., performing probabilistic inference in this model) have been demonstrated. Additionally the neural network model has been applied to a simplified model of vision, simulating on a symbolic and simplified level some findings of a biological experiment by Lee et al.

If the network model only defines the feedforward weights between the input layer and the lateral layer, then the network can recognize single subparts of the input image. In such a model, no image completion is possible since this requires an information exchange between the different subparts of the image.

When adding lateral layer connections, some information can flow among the units responsible for different receptive fields, and so some communication is possible. Still, the influence of these other lateral layer neurons turned out to be quiet weak in the studied network model and input statistics (using only direct neighborhood, small images consisting of few subparts, and choosing relatively low firing rates).

A significant change (or improvement) in the posterior distribution, i.e., the response of lateral neurons, was observed when adding a third layer on the top, which introduces additional prior knowledge: This top layer encodes a prior knowledge on how a complete picture *should* look like. This top layer is connected to all lateral layer neurons, using top-down bottom-up connections, and it results in better image completion and partially and very weak detection of errors in an image. The influence of this prior knowledge turned out to depend on the size of the populations which represent the complete images.

This multilayer network model was applied to a simplified version of the visual system, as proposed by Lee et al [45]:

According to Lee the early visual areas are only responsible for edge detection, but not for completion of missing input fields. In the symbolic setup this feedforward processing corresponds to the weights $\boldsymbol{V}$ between the input layer (a symbolic representation of the visual input), and the lateral layer.

According to Lee, edge completion is done in higher visual areas - symbolically represented by the $\boldsymbol{W}$ connections in the described recurrently connected network model - which supply prior knowledge and exchange information on their current state for, e.g., image completion..

In an extension of the experiment, the theoretically ideal neuron model was modified in order to account for biologically more realistic dynamics, such as additive alpha-shaped PSPs, a relative refractory mechanism and synaptic delays. This more realistic model was applied to a symbolic version of the experiment by Lee et al.

In this experiment it has been shown that neural sampling can successfully be applied to a strongly simplified model of the experiment by Lee et al, simulating some aspects of the behavior of real neurons in a symbolic and artificial way: It is possible to create a model based on neural sampling, where the influence of top-down information causes delayed response of particular neurons getting ambiguous information. In this way, a model is able to reproduce and “understand” (partially) ambiguous inputs through feedforward-feedback information passing, having learned some prior knowledge.

9 Discussion

It has recently been discovered how Bayesian inference can be performed in a network of spiking neurons, by interpreting the neural network as a probabilistic model, where a sample is represented by the neurons' spike response at a certain point in time. In this work, various aspects and properties of such a neural sampling network have been analyzed and tested on an implementation.

In this work an implementation of the neural dynamics sampler has been presented and its usage has been explained. Additionally various parameters and settings have been tested and systematically analyzed. In the last part it was demonstrated how the sampler can be applied to a model of vision, by constructing a multilayer neuron model representing (on a symbolic level) the visual pathway of the mammalian cortex.

The efficient and user-friendly C++ implementation displays a modular and highly flexible framework of the theoretically grounded Network model. Furthermore, it provides various interfaces to more modern and high-level programming languages at different levels of abstraction.

This modular framework was used to investigate modifications of neuron models, such as PSP shapes, refractory mechanisms and synaptic delays, which render neuronal dynamics biologically more realistic. It was found that, for a reasonable strength of synaptic coupling, the distortions introduced by these modifications are likely to not be behaviorally relevant: The systematic error of such modifications might be not so relevant when being compared to the stochastic error caused by the limited sampling time, which cannot be avoided in practice.

The next aim was to implement the abstract vision framework proposed by Lee et al in a symbolic version on the neuron model. Therefore, the sampler is combined with a generative model, which allows to give spiking input to the network, such that it is able to sample from the resulting posterior distribution. This model was trained for maximum likelihood on the input statistics of complete images.

To do so, the influence of the lateral synaptic connections and of top-down information was systematically analyzed. When looking at the findings of these simulations, the delayed response of those neurons which have their receptive fields on ambiguous input parts can be understood inside the dynamics of such a recurrent network.

Future work in this field could consist in the analysis of even more realistic properties (e.g., the influence of missing symmetries in the weight matrix could be observed in more detail), and also in applying the concept to more complex models (deeper hierarchies and different restrictions in the connectivity) which should then be able to perform also more "difficult" tasks, like pattern completion in less obvious pictures.

The results of this work indicate that the dynamics in recent spiking neural network models for probabilistic inference and sampling are in accordance with data from biological recordings. I hope that such findings help to bring neuroscience one additional tiny step forward towards the goal of exploring and finally understanding the brain, which still represents a big mystery for scientists.

> "What is a scientist after all?
> It's a curious man looking through a keyhole, the keyhole of nature,
> trying to know what's going on."
>
> Jacques Cousteau

Part III
Appendix

A Appendix: Technical manual: Neural Dynamics Sampler

This technical manual contains a complete description of the implemented features of the neural dynamics sampler, including also detailed guidelines about how to build and use the sampler. Additionally, some instructions regarding possible extensions are given, as well as some instructions how to build, use or extend it.

A.1 Compiling and Linking

A.1.1 Building the sampler (C++ files)

For the neural network to run, a C++ compiler has to be installed (e.g., `g++`).

Additionally, the library "Eigen" is needed, possibly in version 3, which can be found at `http://eigen.tuxfamily.org` and has to be installed on the machine. For the compilation, it has to be included: `-I/PATH/lib/eigen`.
This library is needed in the neural network model for performant matrix and vector operations and computations.

Another library is required (for the solution of the implicit equation system for $f_k(u_k)$), namely the "GSL - GNU Scientific Library", available at `http://www.gnu.org/software/gsl/`. It should however be already available on most machines. The library should be installed in the local standard include directory, and can thus be used as: `-Igsl -lgslcblas`.

During compilation the compiler optimization should be switched on to a higher level (e.g., using `-O3`), such that the execution of the sampler is more performant.

Compilation and linking of the C++ files can therefore be done by using

```
g++ -I"lib/eigen/" -lgsl -lgslcblas -O3 -g3 *cpp -o network
```

The compiled program is then called: `./network`

A.1.2 Building the sampler and the Matlab interface

If also the additional *mex* files for the Matlab communication should be installed, it might be useful to do compiling and linking separately: For the compilation, the standard C++ compiler can be used, including also the location of the Matlab libraries. These paths have to be adjusted.

```
MATLAB_EXTERN=/usr/local/matlab74/extern/include
MATLAB_SIMULINK=/usr/local/matlab74/simulink/include
g++ -I"../lib/eigen/" -I$MATLAB_EXTERN -I$MATLAB_SIMULINK -DMATLAB_MEX_FILE
    -fPIC -c *cpp
```

Each *mex* function has to be linked separately. Therefore, for each of the pre compiled *mex* files `mexFile.o`, the linker has to be called, using Matlab's *mex* executable and specifying that the object files are pre compiled C++ files. All the C++ classes of the sampling network have to be listed as well. Again, the path to the Matlab directory has to be adjusted before:

```
MATLAB_MEX=/usr/local/matlab74/bin/mex
$MATLAB_MEX -cxx mexFile.o BoltzmannMembranepotential.o ExponentialPsp.o
    ExponentialPspLimitedHeight.o Function.o Logger.o MathFunction.o
    Membranepotential.o Neuron.o PspShape.o RealisticPsp.o
    RealisticPspAdditive.o RectangularAdditivePsp.o RectangularPsp.o Sampler.o
    SamplingNetwork.o StaticInitialization.o -I"../lib/eigen"
    -I/usr/local/include/ -lgsl -lgslcblas -lm -lstdc++ -largeArrayDims
```

Note that a shell script is available, which does the compiling and linking for you, if you specify the paths of your system.

A.1.3 Building the sampler and the python interface

Similar as for Matlab usage, also for building the python interface it might be better to separate the compile and the build process: Compiling of the C++ source code is done in the same way as for the standard C++ compilation:

```
g++ -I"../lib/eigen/" -fPIC -c *cpp
```

In a next step, the SWIG input files have to be wrapped. In order to do so, the files contained in the `swig` folder are needed. Additionally, the installation of *SWIG* (`www.swig.org`) and its python addon is required.
First of all, the input file has to be wrapped into a C++ file:

```
swig -c++ -python -I"../lib/eigen/" -o sampler_wrap.cpp simulator.i
```

The wrapped file has now to be compiled, together with the C++ source code file. During this process, the path to the python installation is required, and has therefore to be specified:

```
PYTHON_INCLUDE=/usr/include/python2.4/
PYTHON_LIB=/usr/lib/python2.4/
PYTHON_INCLUDE_NUMPY=/usr/lib64/python2.4/site-packages/numpy/core/include/
g++ -I"../lib/eigen/" -I$PYTHON_INCLUDE -I/usr/local/include
    -I$PYTHON_INCLUDE_NUMPY -fPIC -c NetworkSimulator.cpp sampler_wrap.cpp
```

Now everything should be compiled, and therefore linking can be done:

```
g++ -shared sampler_wrap.o SamplingNetworkSimulatorFloat.o
    BoltzmannMembranepotential.o ExponentialPsp.o ExponentialPspLimitedHeight.o
    Function.o Logger.o MathFunction.o Membranepotential.o Neuron.o PspShape.o
    RealisticPsp.o RealisticPspAdditive.o RectangularAdditivePsp.o
    RectangularPsp.o Sampler.o SamplingNetwork.o SamplingNetworkSimulator.o
    StaticInitialization.o -I"../lib/eigen"
    -I/usr/local/include/ -lgsl -lgslcblas -lm -lstdc++
    -L$PYTHON_LIB -I$PYTHON_INCLUDE -o _sampler.so
```

Note that a shell script is available (for python 2.4 as well as python 2.5), which does the compilation and linking for you, if you specify the paths of your system. Be sure to use the same version of python for compilation and for running the script.

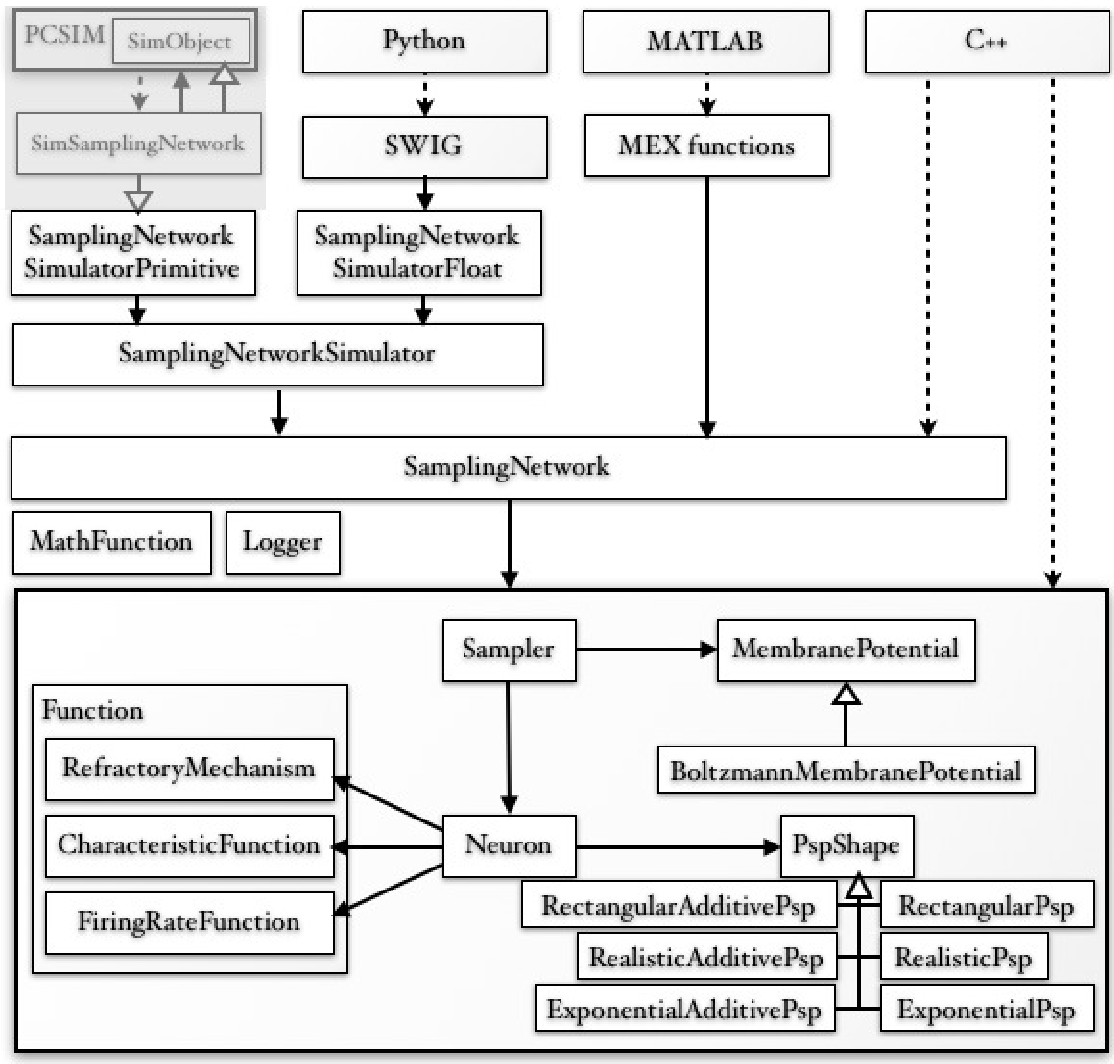

Figure A.1: Neural dynamics sampler class structure

A.2 Class structure

A brief overview over the class structure of the neural network model is shown in figure A.1.

The black arrows in this rough class diagram denote an association between the classes, while the arrows which are not filled indicate inheritance or "is-a". The dotted lines indicate some access which is not directly part of the implementation, e.g., the possibility to use the sampler from different programming languages.

The `Sampler` class is the most general class. It has an associated `MembranePotential` - e.g., `BoltzmannMembranePotential`, which is a specific implementation of such a membrane potential. It is responsible for calculating the current u_k value of each neuron. Additionally, the sampler contains also one or more `Neuron` objects, each of which can be resampled. A neuron knows its own `RefractoryMechanism` $g(\zeta_k)$, `CharacteristicFunction` $\gamma(u_k)$ and `FiringRateFunction` $f_k(u_k)$, as well as its `PspShape` - which is responsible for computing the current PSP value of

the neuron, given its refractory state ζ_k. This shape might be of a concrete specific type, like `RectangularPsp`, `RectangularAdditivePsp`, `ExponentialPsp`, `ExponentialAdditivePsp`, `RealisticPsp`, or `RealisticAdditivePsp`. The other two classes - `Logger` and `MathFunction` contain static methods (for logging and mathematical calculations) which are available to all the other classes, but cannot be associated directly to the other classes.
A Matlab interface for the sampler is already available and running, as well as access through python, while an additional access interface is available, and could for example be accessed easily through external classes (from PCSIM or others) as the used syntax is standard "primitive" C++ without any extensions or libraries.

A.2.1 Class "Sampler"

The `Sampler` is the most general class of the neural network.
The sampler contains a set of neurons (`Neuron`), with its specifications. It has also a specific kind of membrane potential (`MembranePotential`) with weight matrices valid for the whole sampler. Also the time difference value dt is stored once for the sampler, together with the specifications of the range of u (namely the minimum u_{min}, the maximum u_{max}, and the precision $u_{precision}$). In addition to these values, also the global delay is stored in the sampler: it specifies whether all refractory values (or current states) of the neurons should be passed to the other neurons with a certain delay, which is biologically plausible.
A sampler is initialized using the following functions:

- `void addNeuron(Neuron *neuron, PspShape *pspShape=new RectangularPsp())`
  This function adds a neuron to the sampler, whose specifications are already set (using the neuron's constructor). The shape of the postsynaptic potential is passed separately, if some specific shape except from a simple rectangular PSP is desired.
- `void setMembranepotential(Membranepotential *mem)`
  This function sets the membrane potential of the sampler to the given `Membranepotential` object. The weight matrix (or matrices) are already specified in this object, which is then responsible for the computation of the current u_k values.

During the sampling in the network, the following functions might be of interest:

- `bool check()`
  This function has to be called after the initialization of the sampler, before advancing the network. It internally checks whether all values have been set at all, whether matrices and vectors have the correct size, and whether the values are allowed and valid. Only after a positive `check` (which means that the call to the function returns `TRUE`), the network can be advanced.
- `bool reset()`
  This function can be called before advancing the network. Internally it calls the `check` method. If the parameter `double dt` is passed, and this value is the same as the one specified in the constructor, then this method returns the value of `check`, otherwise `FALSE` in any case.
- `VectorXd advance(unsigned steps=1, bool randomOrder=FALSE)`
  This function is a call to advance the network, which means that all neurons are resampled. In the parameters it might be specified for how many steps the network should be advanced

(if not only one step), and whether the resampling of the neurons should be done in random order.

- Various "getter" methods are also available, to get the time difference dt, a pointer to the membrane potential, and to the current PSP values, as well as methods to get information about the existence of a global delay.
- `VectorXd getStates()`
  This function returns a vector of the (relative) states ζ_k/τ_{ref} of all neurons k. The returned values are not delayed.
- `VectorXd getSpikes()`
  This function returns a vector of values, indicating whether the neuron k spiked right now, i.e., $\zeta_k = \tau_{ref}$, for all neurons k. Also these values are not delayed.
- `VectorXd getActivity()`
  This function returns a vector which contains the $z_k := (\zeta_k > 0)$ values of all neurons k. Again, the returned values are not delayed.

- `void setExternalInputOfNeuron(unsigned int k, double value)`
  This function sets the external input of the neuron with index k to the given (pre computed) value.
- `void setExternalInput(VectorXd inputs)`
  This function sets the external inputs of all neurons to the pre computed input values in the given vector. Note therefore, that the number of elements in this input vector has to be equal to the number of neurons in the network.

A.2.2 Class "Neuron"

This class represents an abstract definition of a unit of computation in a neural network.
Each neuron corresponds to one (and only one) specific sampler, and can be uniquely identified inside this sampler by its index. A neuron might have some external input, that is pre calculated (e.g., $< y, W_k >$), and has a specified τ_{ref} and refractory period $\tau_{ref} = \lfloor \frac{\tau_{ref}}{dt} \rceil$.
For each neuron, a kind of refractory mechanism $g(\zeta_k)$ (`RefractoryMechanism`) can be specified, or the user might also directly define a vector containing the $\tau_{ref} + 1$ precomputed values for $g(\zeta_k = 0) \ldots g(\zeta_k = \tau_{ref})$. A characteristic function (`CharacteristicFunction`) can be set, which - if not stated differently - is set automatically to e^{u_k}. The values for the refractory function $g(\zeta_k)$ (for each $r_k \in [0 : \tau_{ref}]$), for the characteristic function for $u_k \in [u_{min} : u_{precision} : u_{max}]$, and for the firing rate function $f(u_k)$ (`FiringRateFunction`, again for all possible u_k) are precomputed at the initialization (`init`) of the neuron for performance issues. If any of these functions has already been precomputed before with exactly the same combination of settings, it is not computed again, but the stored values are reused.
Additionally, a neuron has a corresponding `PspShape`, which represents a very abstract definition of a postsynaptic potential shape. This characterizes the behavior of the neuron's potential after a spike has occurred.
At each point in time, a neuron has specific values for ζ_k (the refractory value), and z_k which measures whether the refractory value is positive. The transmission of these values to the other neurons in the sampler might be delayed, depending on the `globalDelay` set for the whole

`Sampler`.
A neuron provides different functions. The most important ones for a user of the network might be:

- `unsigned getIndex()`
  This function returns the index of this neuron, which is unique inside the sampler.

- `void setClamped(bool one)`
  This function should be called to clamp the current neuron. If it should be clamped to be active, `one` should be `TRUE`, otherwise `FALSE`. A clamped neuron does not change its z_k value at the `resample` step.

- `void setNotClamped()`
  This function sets a clamped neuron to "normal" again, which means that it might spike (or not spike) stochastically again, and not stay unchanged.

- `int resample()`
  This function advances (or resamples) the current neuron by one step. This means, the probability of a spike at the current moment is calculated, and according to this probability, it is decided stochastically whether the neuron should spike right now, or whether the refractory value is lowered (or stays the same if the neurons is already inactive).
  The function is called internally by the sampler (at the `advance` step), but might in some specific cases also be of interest for an external user of the neural network.

A.2.3 Class "MembranePotential"

Biologically, the membrane potential describes the voltage difference between the interior and exterior of some neuron. The implemented neuron model typically uses a very abstract representation of it.
This class `MembranePotential` can be seen as some kind of interface or general structure of how a concrete implementation of a membrane potential class should look like.
The following functions have to be overwritten in specific implementations:

- `virtual double getValues(Neuron* neuron)`
  This function should calculate and return the current value of the given neuron's u_k value.

- `virtual bool checkWeights(unsigned neuronNumber)`
  This function is called at the reset: It should check whether the dimensions of the weight matrices or vectors are correct and valid.

A.2.4 Class "MembranePotential - BoltzmannMembranePotential"

This class represents a special kind of membrane potentials, namely using Boltzmann machines. In this case, the constructor requires to specify a weight matrix $\boldsymbol{W}$ of size $neuronNumber \times neuronNumber$ and a bias vector $\boldsymbol{b}$, also being of the same length as the neurons in the network. Usually the diagonal of the $\boldsymbol{W}$ matrix should only contain zeros, as the intrinsic excitability is already contained in the bias vector, and it could therefore be confusing to have an additional value for it. It is however also possible to have some additional self-interaction, and thus to have a non-zero diagonal.

The class `BoltzmannMembranePotential` implements the virtual functions defined in the superclass `MembranePotential`, and some additional methods which are required (especially for the learning process) for Boltzmann machines:

- The `getValues` function returns $u_k = b_k + externalInput + internalInput$.
  The external input of a neuron is specified at the declaration of a neuron and stored then separately for each neuron (e.g., $externalInput = \boldsymbol{y}^\top \boldsymbol{W'}_k$ with $\boldsymbol{y}$ being some external values, and $\boldsymbol{W'}_k$ indicating the weights from the external values to this neuron).
  The internal input represents the influence coming from other neurons inside this network to this neuron. It is given as $internalInput = \boldsymbol{psp}^\top \boldsymbol{W}_k$ where $\boldsymbol{psp}$ are the current postsynaptic potential values (computed using the `PspShape`), and $\boldsymbol{W}_k$ represents the kth row of the weight matrix $\boldsymbol{W}$.

- The `checkWeights` function checks whether the dimensions of the weight matrix $\boldsymbol{W}$ and of the bias vector $\boldsymbol{b}$ both match the number of neurons inside the network.

- The functions `setWeights` and `getWeights` both are related to the weight matrix $\boldsymbol{W}$.

- Using the function `addToWeights` it is possible to add some $\Delta\boldsymbol{W}$ value to each entry of the weight matrix. The given matrix has therefore to be of the same dimensions as the original $\boldsymbol{W}$ matrix and it contains the elements that should be added to this original weight matrix. This function might be helpful during the learning phase.

- Also the function `updateWeights` might be of interest during the learning phase of the network. Given the $\boldsymbol{z}_{posterior}$ vector of some posterior sample, the $\boldsymbol{z}_{prior}$ vector of a prior sample and a learning rate η (being a scalar), the $\boldsymbol{W}$ matrix is updated, using an additional mask matrix $\tilde{\boldsymbol{\eta}}_{\boldsymbol{W}} \in [0,1]$, whose elements define which entries of the weight matrix should be learned, and which should not. Note that this mask matrix should again be symmetric. The update is then done:

$$\Delta\boldsymbol{W} = \eta \cdot \left(\left(\boldsymbol{z}_{posterior} \boldsymbol{z}_{posterior}^\top \right) - \left(\boldsymbol{z}_{prior} \boldsymbol{z}_{prior}^\top \right) \right)$$
$$W_{kl} = W_{kl} + \tilde{\eta}_{\boldsymbol{W}_{kl}} \cdot \Delta W_{kl}$$

- The function `setWeightMask(Eigen::ArrayXXd)` can be used to change or set the mask for learning and updating the weight matrix, i.e., the array $\tilde{\boldsymbol{\eta}}_{\boldsymbol{W}}$. Again, this array has to be of the same size as the original weight matrix. The standard value is an array of ones with a zero-diagonal.

- The functions `setPi` and `getPi` can be used to handle the bias vector $\boldsymbol{b}$.

- Using `addToPi` it is possible to add some $\Delta\boldsymbol{b}$ values to each entry of the current $\boldsymbol{b}$ vector. Therefore the dimensions of both vectors must match. This function could be useful during learning.

- The function `updatePi` might also be interesting during the learning phase of the network. Given again the $\boldsymbol{z}_{posterior}$ vector of some posterior sample, the $\boldsymbol{z}_{prior}$ vector of a prior sample and a learning rate η_b (being a scalar), the vector $\boldsymbol{b}$ is updated, again using an additional mask vector $\tilde{\boldsymbol{\eta}}_{\boldsymbol{b}} \in [0,1]$, with entries defining for each vector element whether is should be updated (and learned) or not. The update step is then the following:

$$\Delta\boldsymbol{b} = \eta_b \cdot (\boldsymbol{z}_{posterior} - \boldsymbol{z}_{prior})$$
$$b_k = b_k + \tilde{\eta}_{b_k} \cdot \Delta b_k$$

- The function `setPiMask(Eigen::ArrayXd)` can be used to set the mask for learning the $\boldsymbol{b}$ vector, being $\tilde{\boldsymbol{\eta}}_{\boldsymbol{b}}$. The given array has to be of the same size as the original vector. The standard value is an array of ones.

A.2.5 Class "PspShape"

This class represents an interface or skeletron, the abstract idea of a neurons' PSP shape. It contains the definitions of two virtual functions that should be overwritten in any subclass:

- `virtual double getPspValue(int refractoryValue)`
  This method returns the PSP value of a neuron, given the current refractory value ζ_k. The standard implementation returns 0, regardless of the refractory state.
- `virtual double updatePspValue(int refractoryValue)`
  This method updates the PSP value of a neuron, given the current refractory value ζ_k, and returns it. The standard implementation returns 0, regardless of the refractory state.
- `virtual bool isRectangular()`
  This method is only used to check whether all of the PSP shapes are rectangular. For all implementations except `RectangularPsp` this method should return `false`.

Additionally, this interface contains the type definitions for all the different available PSP shapes

A.2.6 Class "PspShape - RectangularPsp"

The class `RectangularPsp` represents one specific kind of PSP shape, namely rectangular PSPs (see figure A.2). The implementations of the methods `getPspValue` as well as `updatePspValue` both return 1 in case if $\zeta_k > 0$, and 0 otherwise. The implementation of `isRectangular` returns `true`.

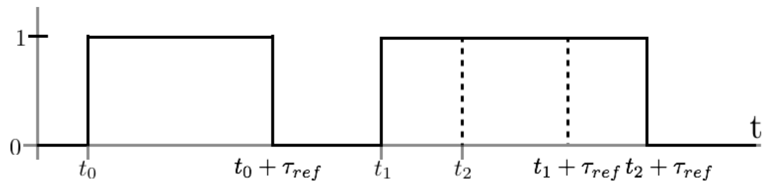

Figure A.2: PSP shape: Rectangular renewing

A.2.7 Class "PspShape - RectangularAdditivePsp"

The class `RectangularAdditivePsp` is a more specific type of the rectangular PSP shapes: While the normal rectangular PSP shape has a height of 0 or 1, this one can take also other values, as shown in figure A.3. It gets as parameters in the constructor the values for

- the *height*, which represents the value that the PSP rises in case of a spike,
- and the refractory period τ_{ref}, which is also needed for the computation.

As the name of the class already says, the shape is additive, i.e., if the neuron spikes, the PSP value raises to *height*. After the refractory period τ_{ref} is over, it would return to 0, unless a new spike occurs already before the refractory period is over. In this case the PSP value rises to $2 \cdot height$, until the first refractory period is over, then decreases again to *height* until the second refractory period is over, and returns to 0 afterwards.

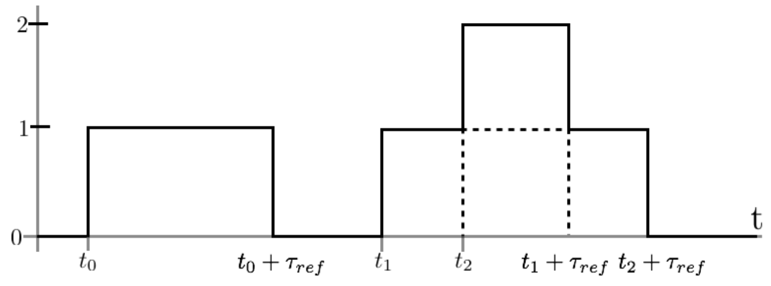

Figure A.3: PSP shape: Rectangular additive

A.2.8 Class "PspShape - ExponentialPspLimitedHeight"

The class `ExponentialPspLimitedHeight` is another subclass of `PspShape`. It describes renewal exponential PSP shapes, like shown in figure A.4.
In the constructor, different values have to be defined:

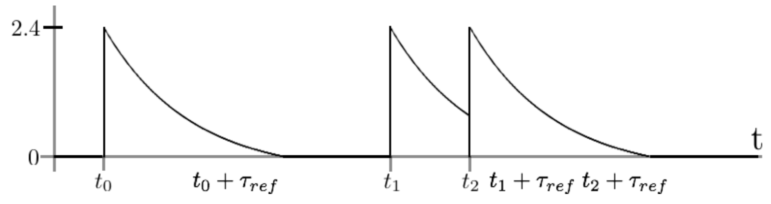

Figure A.4: PSP shape: Exponential renewing

- the *height*, which represents the value that the PSP rises in case of a spike,
- a λ value which defines the steepness of the shape,
- and the refractory period τ_{ref}, which is also needed for the computation.

The current value is given by

$$psp_{t+1} = \begin{cases} height & \text{if } \zeta_k = \tau_{ref}, \\ psp_t \cdot \lambda & \text{otherwise.} \end{cases}$$

A.2.9 Class "PspShape - ExponentialPsp"

This class `ExponentialPsp` is very similar to the previously described sort of PSP shapes, `ExponentialPspLimitedHeight`, with the difference that - as the name already tells - these PSPs have not limited heights, i.e., they are additive (see figure A.5).

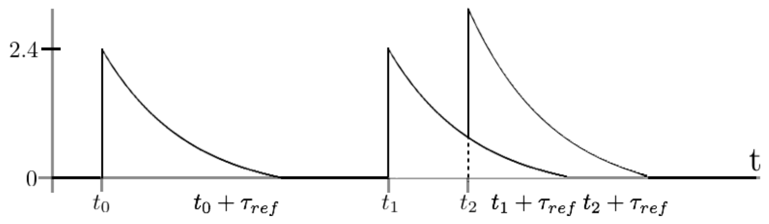

Figure A.5: PSP shape: Exponential additive

The parameters required for the constructor are the same as for the unlimited exponential PSPs. The current value corresponding to a given refractory value is then calculated as

$$psp_{t+1} = \begin{cases} psp_t \cdot \lambda + height & \text{if } \zeta_k = \tau_{ref}, \\ psp_t \cdot \lambda & \text{otherwise.} \end{cases}$$

A.2.10 Class "PspShape - RealisticPsp"

The class `RealisticPsp` represents some PSPs with alpha-shapes. Such shapes are computed as the difference between two exponential PSP shapes (see figure A.6).

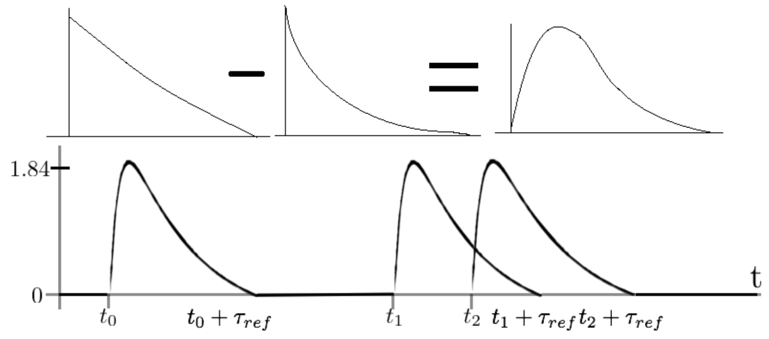

Figure A.6: PSP shape: Alpha-shaped, renewing, computed as difference between exponentials

The parameters needed in the constructor and for the computation are the following

- the *height*, which is not the entire height of a PSP spike, but the maximum height of the individual exponential shapes, whose difference is then computed,
- a $\lambda^{(1)}$ value which defines the steepness of the shape of the first exponential shape,
- a $\lambda^{(2)}$ value which defines the steepness of the shape of the second exponential shape,
- and the refractory period τ_{ref}.

Therefore, the current value corresponding to a given refractory value is calculated as:

$$val^{(1)}_{t+1} = \begin{cases} height & \text{if } \zeta_k = \tau_{ref}, \\ val^{(1)}_t \cdot \lambda^{(1)} & \text{otherwise.} \end{cases}$$

$$val^{(2)}_{t+1} = \begin{cases} height & \text{if } \zeta_k = \tau_{ref}, \\ val^{(2)}_t \cdot \lambda^{(2)} & \text{otherwise.} \end{cases}$$

$$psp_{t+1} = val^{(1)}_{t+1} - val^{(2)}_{t+1}$$

A.2.11 Class "PspShape - RealisticPspAdditive"

The class `RealisticPspAdditive` corresponds to the same kind of PSP shapes as the previously described one, but it is now additive, not renewal, as shown in figure **??**.

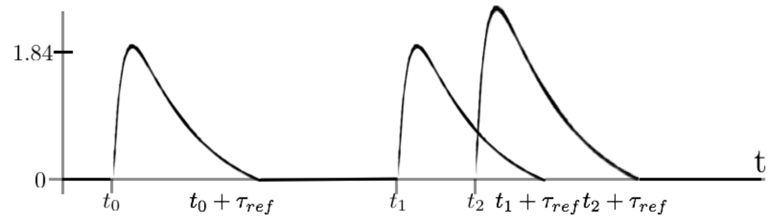

Figure A.7: PSP shape: Alpha-shaped additive

It takes the same parameters as the renewal alpha-shaped PSPs, and the current value is again computed as the difference between two exponentials:

$$val^{(1)}_{t+1} = \begin{cases} val^{(1)}_t \cdot \lambda^{(1)} + height & \text{if } \zeta_k = \tau_{ref}, \\ val^{(1)}_t \cdot \lambda^{(1)} & \text{otherwise.} \end{cases}$$

$$val^{(2)}_{t+1} = \begin{cases} val^{(2)}_t \cdot \lambda^{(2)} + height & \text{if } \zeta_k = \tau_{ref}, \\ val^{(2)}_t \cdot \lambda^{(2)} & \text{otherwise.} \end{cases}$$

$$psp_{t+1} = val^{(1)}_{t+1} - val^{(2)}_{t+1}$$

A.2.12 Class "RefractoryMechanism"

The neuron's current refractory mechanism value $g(\zeta_k)$ is needed for the computation of the spiking probability (together with the characteristic function).
The refractory mechanism is precomputed for each neuron already at initialization time, such that the values can simply be used later on, and further computations can be avoided. It is also possible, to set a neuron's refractory mechanism directly, by giving to the constructor of the neuron the set of precomputed $g(\zeta_k)$ values for all $\zeta_k = [0; \tau_{ref}]$.
The class `RefractoryMechanism` contains different definitions of refractory functions $g(\zeta_k)$ for a neuron.
`static VectorXd *computeRefractorymechanism(Type mechanism, int R)` is the function that can be called to obtain the all values $g(\zeta_k)$ according to the refractory states ζ_k of neuron k. It takes as parameter the type of the refractory mechanism *mechanism* of this neuron, as well as the refractory period τ_{ref}.
The `enum Type` should be used to identify the desired definition of a refractory mechanism:

- One example for a refractory mechanism, that is already implemented, is an absolute refractory mechanism (`absoluteRefractoryMechanism`). Here, the neuron cannot spike during the refractory period, until the refractory value reaches $\zeta_k = 1$, meaning

$$g(\zeta_k) = \begin{cases} 1 & \text{if } \zeta_k \leq 1, \\ 0 & \text{otherwise.} \end{cases}$$

- Another already implemented definition of refractory mechanism is some relative refractory mechanism representing some sigmoidal (`sineWaveMechanism`). In this case, the refractory function is computed as $g(1-\zeta_k) = g(t - t_{spike})$, which means

$$g(\zeta_k) = \frac{\tau_{ref} - k}{\tau_{ref}} - \frac{\sin\left(2\pi \cdot \frac{\tau_{ref}-k}{\tau_{ref}}\right)}{2\pi}$$

A.2.13 Class “CharacteristicFunction”

The class `CharacteristicFunction` $\gamma(u_k)$ is responsible for calculating the characteristic function of a neuron, given its current membrane potential value u_k. The characteristic function is equal to

$$\frac{p(\zeta_k > 0 \mid r_{\setminus k})}{p(\zeta_k = 0 \mid r_{\setminus k})} = e^{\log \frac{p(\zeta_k > 0 \mid \zeta_{\setminus k})}{p(\zeta_k = 0 \mid \zeta_{\setminus k})}} = \gamma(u_k)$$

Usually this is obtained as

$$\gamma(u_k) = e^{u_k}$$

This is also the standard characteristic function used for the sampler.

A.2.14 Class “FiringRateFunction”

This class is needed to get the values of the firing rate function $f_k(u_k)$, as the solution to the implicit equation system

$$f_k(u_k) \cdot \frac{\sum_{i=1}^{\tau_{ref}} \prod_{j=i+1}^{\tau_{ref}} \left(1 - f_k(u_k) \cdot g_k(\zeta_k = j)\right)}{\prod_{j=1}^{\tau_{ref}} \left(1 - f_k(u_k) \cdot g_k(\zeta_k = j)\right)} = \gamma(u_k)$$

The function `computeFiringRate` can be called to obtain the set of precomputed values of the firing rate function, given the set of precomputed values of the characteristic function e^{u_k} for the desired (and predefined) range and precision of u_k values, and the precomputed vector of values for $g(\zeta_k)$ for all $\zeta_k \in [0; \tau_{ref}]$. The function then returns a pointer to a vector filled with the values of $f_k(u_k)$ for the same range and precision of u_k values.

A.2.15 Class “MathFunction”

This class contains static method for various mathematical calculations and random number generators.
All of these methods can be used statically from any of the other classes, and the `MathFunction` class might be extended arbitrarily by adding additional methods for mathematical functions.
For a simple use of the neural network model, all of these methods might be helpful during the initialization and the evaluation of the results, bot none of them is necessary for the network model itself.

- `static VectorXd getRandomGaussNumbers(double mu, double sigma, unsigned amount)`
  This method returns a vector of double values, containing the specified amount of random Gaussian distributed numbers. The Gauss values have mean μ and covariance σ^2: $\mathcal{N}(\mu; \sigma^2)$

- `static Matrix<double, Dynamic, Dynamic, RowMajor> getRandomGaussNumberMatrix`
  `(double mu, double sigma, unsigned rows, unsigned cols)`

  This method returns a matrix of double values of size $< rows, cols >$, which contains normal distributed values around mean μ with covariance σ^2: $\mathcal{N}(\mu; \sigma^2)$. As these values are might be used for the initialization of the $\boldsymbol{W}$ matrix, this function is optimized for this: The resulting matrix is diagonally symmetric, meaning that $\boldsymbol{W} = \boldsymbol{W}^\top$, such that $W_{kl} = W_{lk}$. This should be considered, if this function might be used for any other matrix of Gaussian numbers.

- `static double kullbackLeiblerDivergence(VectorXd p, VectorXd q)`
  This method computes the Kullback Leibler divergence or relative entropy between the probability distributions p and q as being

$$KL(p\|q) = \sum_i p(i) \cdot \log_{10} \frac{p(i)}{q(i)}$$

- `static int getDecimalValue(VectorXd binary)`
  This method takes a vector of double values, containing only zeros and ones. This vector is interpreted as a binary number by concatenating the vector values, starting from the first and going on to the last one. The resulting decimal representation of the number is returned as an integer value.

- `static int getDecimalValue(VectorXd binary, VectorXd powerTwoValues)`
  This method does the same as the previously explained one, but by relying on another user input: In this case, the user specifies the vector of "power-2" values, which are then used to compute the decimal value. In this way it is possible, to "consider" only some of the values in the binary array, which might be helpful when analyzing the states of neurons in large networks, if only few neurons - with known indices - are of interest.

- `VectorXd getBinaryValue(double decimal)`
  This function is the inverse of the previously explained ones: It computes a vector of binary values for the given decimal number. Note that if you do not specify an integer number, the decimal part of the number is ignored.

- `VectorXd getBinaryValue(double decimal, VectorXd powerTwoValues)`
  This function again converts a decimal number into a vector of binary values, but this time using the given pre defined vector of values. It works in a similar fashion as the previous functions.

- `static VectorXd computePdAnalytically`
  `(MatrixXd J, VectorXd pi, unsigned numCombinations, unsigned numNeurons)`

  This method is used to compute the probability distribution of a Boltzmann machine neural network analytically. It takes a weight matrix $\boldsymbol{W}$ with zero main diagonal (if there is no self-interaction of the neurons), and a bias vector $\boldsymbol{b}$.

All *numCombinations* possible combinations of *numNeurons* neurons (or the neurons' $z_k = (\zeta_k > 0)$ values) are used for the computation:

$$p(z) = \frac{1}{Z} \cdot e^{\frac{1}{2}\boldsymbol{z}^\top \boldsymbol{W}\boldsymbol{z} + \boldsymbol{z}^\top \boldsymbol{b}} \text{ with } Z = \sum_{\boldsymbol{z}} e^{\frac{1}{2}\boldsymbol{z}^\top \boldsymbol{W}\boldsymbol{z} + \boldsymbol{z}^\top \boldsymbol{b}}$$

A.2.16 Class "Logger"

This class contains some general methods for logging all details. It may be extended arbitrarily by adding any static methods.

A.3 Setup: Using the sampler

This section gives a brief introduction and a short example of how the neural network model can be set up and used in practice. There are two different possibilities how the system can be set up and used, namely either by using the provided class `SamplingNetwork` which acts as kind of a black-box, holding all objects and caring about memory aspects internally, or by using the provided classes and objects described in the previous section directly. Both of these options are briefly discussed in the following, letting the user decide for which possibility to go.

A.3.1 Setup "manually"

Sampler initialization

For the initialization of the sampler, the following parameters should already be given:

- *dt* . . . a *double* value, indicated in milliseconds, which in combination with the τ_{ref} value (that can be specified for each neuron separately), defines the refractory period of that neuron: $R = \lfloor \frac{\tau_{ref}}{dt} \rceil$. This should be considered when choosing the *dt* value: This value is usually rather small (not above 1.0, but always positive, i.e., > 0), while the τ_{ref} value is much higher (e.g., 20), in order to obtain a R value that is positive ($R > 0$).

- *du* . . . a *double* value, which indicates the precision of the membrane potential values u_k. It defines to which precision or detail level the u_k values have to be rounded to obtain one specific value out of the set of precomputed function values. It is therefore some precision indication for the pre-computation and for the functions depending on the current membrane potential values. As an indication, this setting should be very near to 0 (but always positive), e.g., at 10^{-4}, to obtain good approximation values. Note, that if *du* becomes much smaller (e.g., from 10^{-6} onwards), the initialization of the sampler might take quiet some time, as all function values have to be pre-computed with the given precision.

- u_{min} . . . a *double* value, which indicates the minimal u_k value that has to be considered. This means that if the membrane potential value u_k goes below this value, it is clamped to this specific u_{min}. Usually $u_{min} = -15.0$ is low enough to consider all possible u_k values without clamping.

- u_{max} . . . a *double* value, representing the maximum u_k value to be pre-calculated and considered. In most cases, $u_{max} = +15.0$ is a good indication of this value, such that the arising u_k values do not have to be changed or clamped.

- *delay*... an *unsigned* value, which defines the global delay (measured in seconds) in the network, which is the number of steps of delay for all the r_k and z_k values for the other neurons in the network. This means, each neuron does not get the simultaneous information about the current state of all the other neurons, but only some delayed information. Usually this value should be rather small, as it also slows down computation. In our example we use 0 delay.

Knowing these values, the sampler can be initialized using its constructor:

```
double   dt         = 1.0;
double   du         = 0.0001,
         umin       = -15.0,
         umax       = 15.0;
unsigned delay      = 0;

Sampler *sampler = new Sampler(dt, du, umin, umax, delay);
```

Creating and adding neurons

In a next step, the neurons can be initialized and added to the sampler.
For this the τ_{ref} values (again in milliseconds) described previously are now needed, and additionally the value for the $input_{external}$, which represents the input which comes from outside the network to a certain neuron. This external input has to be precomputed, and only a *double* value has to be passed to the neuron's constructor. In the example, 0 is used as external input, which indicates that no external input from outside the network comes to the neurons.
Also the refractory mechanism has to be defined for each neuron. In the example, the standard $g(\zeta_k)$ mechanism is used.
The characteristic function can be specified as well for each neuron, but not indicating any kind, leads to use e^{u_k} automatically.
Having these values (which can also be set differently for each neuron), the neurons can be created.
In order to add them to the sampler, also the shape of the postsynaptic potentials can be defined for each neuron. In the example, no specific shape is specified, which means that rectangular PSPs are used:

```
unsigned numNeurons  = 8;
double   tauRef      = 20.0;
double   extInput    = 0;

vector<Neuron*> neurons;
unsigned i;
for (i=0; i<numNeurons; i++) {
    Neuron *neuronNew = new Neuron ...
            (tauRef, Refractorymechanism::subtractSpikeMechanism, extInput);
    neurons.push_back(neuronNew);
    sampler->addNeuron(neuronNew);
}
```

Initializing the membrane potential

Now the membrane potential has to be defined and initialized.
For its initialization, the weight matrix $\boldsymbol{W}$ and the vector $\boldsymbol{b}$ have to be defined:

- $\boldsymbol{W}\ldots$ has to be a quadratic row-major matrix of the same size as there are neurons in the network. This indicates, that from each neuron to each other neuron, a weight (possibly also 0) has to be defined. Usually the matrix has a zero diagonal, which means that there is no self-interaction of the neurons. In the example below, quiet high, Gaussian distributed values are used: $\boldsymbol{W} = \mathcal{N}(0; 1.0)$. The $\boldsymbol{W}$ matrix has to be symmetric, i.e., $W_{kl} = W_{lk}$.

- $\boldsymbol{b}$ is a vector with the same length as there are neurons in the neural network. Therefore, there is one b_k value for each neuron k. In the following example, the values in the vector are Gaussian distributed: $\boldsymbol{b} = \mathcal{N}(-2.0; 0.5)$.

Having a matrix $\boldsymbol{W}$ and a vector $\boldsymbol{b}$, the Boltzmann membrane potential can be initialized, and then set for the sampler:

```
double muB    = -2,
       sigmaB = 0.5;
double muW    = 0,
       sigmaW = 1;

// Initializing membrane potential
VectorXd b = MathFunction::getRandomGaussNumbers(muB, sigmaB, numNeurons);
Matrix<double, Dynamic, Dynamic, RowMajor> W = ...
        MathFunction::getRandomGaussNumberMatrix ...
                (muW, sigmaW, numNeurons, numNeurons);
W.diagonal() = VectorXd::Zero(numNeurons);

// Setting membrane potential
BoltzmannMembranepotential *mem = new BoltzmannMembranepotential(W, b);
sampler->setMembranepotential(mem);
```

Final setup before sampling

Before starting the sampling, some additional steps have to be considered.
After having initialized the sampler itself, and added neurons and a membrane potential, the sampler has to be checked. Only if this process could be finished successfully, the sampler can be used. During the reset process, all the settings and parameters of the sampler are checked, and sampling is only possible with existing and valid settings:

```
sampler->check();
```

After this, still the mixing time should be skipped. To do so, about 5000 milliseconds should be enough. Only after this mixing or burn-in time, the sampler gives reasonable results, as the probability distribution should be adjusted:

```
unsigned mxingTime = 5000;
sampler->advance(mixingTime);
```

Sampling

After having set all parameters and initialized the sampler, the real sampling process can begin. During this, it is also possible to write some statistics to a file, or to adjust some parameters during the sampling steps, e.g., the weight matrix could be adjusted, or some neurons could be clamped for a specific time.
The sampling itself is done by advancing the network:

```
unsigned milliseconds = 10000;

for (i=0; i<milliseconds; i++) {
    VectorXd z_i = sampler->advance(1);
}
```

Closing sampler

After everything has been done, the sampler can be closed. For this, one should consider to also free the allocated memory. In the above example, some objects have been created, that should now be deleted from memory, i.e., the sampler itself, the neurons, and the membrane potential object:

```
delete sampler;
unsigned k;
for (k=0; k<neurons.size(); k++) {
    delete neurons[k];
}
neurons.clear();
delete mem;
```

A.3.2 Setup using the provided “Blackbox”

In this case, using the class `SamplingNetwork`, the neural network can be set up easily, without creating any objects (except from the sampling network itself), and therefore without caring about what is done with the used memory storage after the sampling process has finished. The way this “blackbox” class might be used, is described by showing a brief example.

For the initialization of the sampling network itself, no parameters are needed, yet:

```
SamplingNetwork *network = new SamplingNetwork();
```

Immediately afterwards, the sampler can be created and initialized, using the provided function:

```
network->createSampler(dt, du, umin, umax, delay);
```

All of these parameters are optional:

- *dt* . . . a *double* value, indicating the internal time difference in seconds of the sampler, from which - in combination with τ_{ref} required later - the refractory period is obtained as $R = \lfloor \frac{\tau_{ref}}{dt} \rceil$. The standard value is $dt = 1.0$.
- *du* . . . a *double* value, which represents the precision for the computation of the u_k values (and therefore also for the $f_k(u_k)$ values). If not stated differently, it is set to $du = 10^{-4}$.

- $u_{min} \dots$ a *double* value, indicating the lowest u_k value that should occur (or that is considered). The standard value - if not set differently - is $u_{min} = -15.0$
- $u_{max} \dots$ a *double* value, indicating the highest u_k value that is considered, usually being $u_{max} = +15.0$.
- *delay*... a *unsigned int* value, representing the global delay (in seconds) of the network. This means, that internal states are visible to the other neurons inside the network only with a temporal delay. The standard value in this case is $delay = 0$.

In a next step, the neurons can be added to the network. To do so, either single neurons can be added separately, or it is also possible to add a set of neurons with equal specifications at a time. To do so, there are four options:

```
network->createNeuron(tauRef, gPrecomputed, input, pspType, param, fType);
network->createNeuron(tauRef, gType, input, pspType, param, fType);
network->createNeurons(amount, tauRef, gPrecomputed, input, pspType, param, fType);
network->createNeurons(amount, tauRef, gType, input, pspType, param, fType);
```

All of these methods do basically the same, namely creating neurons and adding them to the sampler. Again, most parameters are optional, meaning:

- $\tau_{ref} \dots$ a *double* value (indicated in seconds), which in combination with the *dt* value mentioned before defines the refractory period τ_{ref}. This value is mandatory, and no standard value is defined.
- $g_{precomputed} \dots$ a *vector* of *double* values, representing the precomputed values of $g(\zeta_k = i)$ for $i = [0; \tau_{ref}]$, i.e., $\tau_{ref} + 1$ different values, representing the precomputed refractory mechanism values. If this kind of refractory mechanism should be used, it is mandatory to specify these values.
- $g_{Type} \dots$ a *type* of *refractory mechanism*, indicating which kind of pre-implemented refractory mechanism should be used for the computation of the $g(\zeta_k)$ values. This parameter is optional, and its standard value is the absolute refractory mechanism.
- $f_{Type} \dots$ a *type* of *characteristic function*, specifying which kind of characteristic function $\gamma(u_k)$ should be used. The specification of this type is optional, and the standard function to be used is $\gamma(u_k) = e^{u_k}$.
- *input*... a *double* value, specifying the external input (from outside the network) to this neuron. Again, also this value is optional, meaning that it is set to $input = 0.0$ if not defined differently.
- *pspType*... a *type* of *PSP shape*, indicating which sort of PSP shape this neuron has. If no PSP type is indicated by the user, rectangular renewing PSPs are used.
- *param*... an *array* of *double* values, containing all the parameters that are required to initialize the given kind of PSP shape. This means, if rectangular PSPs should be used, no parameters are needed (this can be set to 0), or if, e.g., exponential PSPs (with or without limited height) should be used, then the required parameters are $\{\lambda, height\}$, and these parameters should be put in the array in the order they are required for constructing

the PSP shape. For realistic PSPs additionally the parameter λ_2 should be given. Note that the refractory period τ_{ref} is not required, as it can be derived from the other given parameters. If no paramters are specified by the user, it is assumed to be an empty array.

- *amount*... an *unsigned int* value, which represents a mandatory value, indicating how many neurons with the given parameter setting should be created and added to the sampler.

After having set all the neurons, the matrices for the membrane potential computation have to be set. Currently, only Boltzmann membrane potentials are available. This setting can be done in two different ways, either by specifying a certain weight matrix and vector directly, or by choosing mean and covariance for Gaussian distributed random values for the weights:

```
network->createBoltzmannMembranePotential(numNeurons, muW, sigmaW, muB, sigmaB);
network->createBoltzmannMembranePotential(W, b);
```

Basically, these two options are self-explanatory. Nevertheless, a short description of the parameters is given in the following:

- *numNeurons*... a *unsigned int* value, describing how many neurons there are in the network. Note that this setting is checked later on.
- $\mu_{\boldsymbol{W}}, \sigma^2_{\boldsymbol{W}}$... *double* values, which indicate the mean and covariance for the weight matrix $\boldsymbol{W}$. The matrix gets a zero-diagonal and it is symmetric, meaning $\boldsymbol{W} = \boldsymbol{W}^\top$. The values are Gaussian distributed: $\boldsymbol{W} = \mathcal{N}(\mu_{\boldsymbol{W}}, \sigma^2_{\boldsymbol{W}})$
- $\mu_{\boldsymbol{b}}, \sigma^2_{\boldsymbol{b}}$... *double* values, representing the mean and the covariance for the Gaussian distribution of the random values, which are used to fill the $\boldsymbol{b}$ vector: $\boldsymbol{b} = \mathcal{N}(\mu_{\boldsymbol{b}}, \sigma^2_{\boldsymbol{b}})$
- $\boldsymbol{W}$... a *row-major matrix*, containing *double* values, representing the weights of the synapse connections, i.e., some already created $\boldsymbol{W}$ matrix.
- $\boldsymbol{b}$... a *Vector* of *double* values, being some previously set bias vector $\boldsymbol{b}$.

Before starting the sampling process, the sampler has to be reset and some burn-in time should advance. All this is done, using

```
network->prepareSampler(burnInSteps);
```

Where the only parameter, namely *burnInSteps* is an *unsigned int* value, which describes, for how many steps (or seconds) the sampler should advance, until the burn-in time is over. If no value is set, the burn-in time is assumed to be 0, even though it is recommended to have some burn-in time such that the initial state does not influence the probability distribution from which to sample.
Alternatively, one could also use the `check` method and advance the burn-in time explicitly, as it is done when using the `Sampler` class directly.

Now the sampling process may start. Sampling itself is done by calling:

```
network->advanceSampler(timePeriod);
```

The parameter *timePeriod* should be an *unsigned int* value, that indicates how many steps the sampler should advance. If no value is given, all neurons in the sampler are only resampled once.

There are several kinds of information that can be stored during sampling:

- If the sampled probability distribution should be known afterwards, the sampled states are stored during the sampling process, which is switched on using:

  ```
  network->storeProbabilities(true);
  ```

- It might also be necessary to store only some specific probabilities. For example, in very large networks, storing the complete probability distribution, i.e., 2^K different states, might be as well infeasible as useless. In this case, one can decide to switch on the distribution storage process only for some neurons, passing a vector, containing the indices of these relevant neurons:

  ```
  network->storeProbabilities(relevantNeuronIndices);
  ```

- If someone is interested in the evolution of the neurons' states, it is also possible to store the (relative) refractory states of the neurons $\frac{\zeta_k}{\tau_{ref}}$ during the sampling process, for a predefined number of time steps (in milliseconds). This is switched on by a call to:

  ```
  network->storeEvolution(true, 1000);
  ```

- If the spike trains of all neurons should be stored, this is done using

  ```
  network->storeSpikeEvolution(true);
  ```

- If a user is only interested in the evolution of the activity, i.e., the $\boldsymbol{z}$ values of all neurons, then tracking of the z_k values can be switched on (again indicating the number of steps to track), using

  ```
  network->storeActivityEvolution(true, 1000);
  ```

- The firing rate (in Hertz) - averaged over all neurons - can also be stored and tracked during sampling. This can be done, by enabling the tracking before the sampling process:

  ```
  network->storeFiringRate(true);
  ```

- The firing rates (again in Hertz) of all single neurons can be tracked by enabling tracking before sampling:

  ```
  network->storeFiringRates(true);
  ```

After the sampling process (if tracking was switched on during sampling), it might be interesting to obtain some measure about how well the sampling process could simulate the real (analytically computed) probability distribution. An appropriate measure to do so, might be the Kullback-Leibler divergence:

```
network->getKLtoAnalyticalDistribution();
```

which returns a double value, indicating the Kullback-Leibler distance $KL(p\|q)$ between the sampled probability distribution $p(z)$ and the analytically calculated distribution $q(z)$.
Also for the other tracked information there are functions to get the data, namely

- `VectorXd getSamplingProbabilities()` returns the sampled probability distribution $p(\boldsymbol{z})$ for all combinations of $\boldsymbol{z}$ values.
- `VectorXd getAnalyticalProbabilities()` returns the analytically computed probability distribution. It might be interesting to compare this distribution to the sampled one.
- `MatrixXd getStateEvolution()` returns a matrix, in which each column represents the state vector $\boldsymbol{\zeta}/\tau_{ref}$ at one sampling time step.
- `MatrixXd getSpikeEvolution()` returns a matrix, in which each column stands for the spiking situation at one time step. This means, a 1 means that this neuron spiked right now, i.e., $\zeta_k = \tau_{ref}$.
- `MatrixXd getActivityEvolution()` returns again a matrix, whose columns represent the $\boldsymbol{z} := \boldsymbol{\zeta} > 0$ values of all neurons at one sampling time step.
- `double getFiringRate()` returns the firing rate in Hertz, which was iteratively computed during the sampling time.
- `VectorXd getFiringRates()` returns a vector containing the firing rates in Hertz of all neurons in the network, which were iteratively computed throughout the sampling time.

During sampling, some additional functions might be useful, among them

- clamping single neurons to be either active or inactive all the time, using `setClamped`, or unclamping them by `setNotClamped`.
- setting the external input of one single neuron (given its index), or of all neurons, for which the values should be specified in a vector, calling `setExternalInput`
- changing the weight matrix $\boldsymbol{W}$ or the bias vector $\boldsymbol{b}$, using the methods `setWeightMatrix`, `addToWeightMatrix`, `updateWeightMatrix`, or respectively `setPiVector`, `addToPiVector`, `updatePiVector`. In the update functions, a posterior sample for learning has to be provided, as well as a learning rate η. If no prior sample is given, the current $\boldsymbol{z}$ states of the network are used. The weights can be obtained by a call to `getWeightMatrix`, or `getPiVector`. To set or change the mask arrays for both, the $\boldsymbol{W}$ matrix and the $\boldsymbol{b}$ vector, the functions `setWeightMask` and `setPiMask` can be used, respectively.
- getting the current network state, using `getActivity`, `getStates` or `getSpikes`.

The sampling network can be deleted from memory very easily, calling its destructor:

```
delete(network);
```

A.4 Extension guidelines

A.4.1 Membrane potentials

The only implemented sort of membrane potential, is the one of a Boltzmann machine.
If any other kind of membrane potential should be added to the neural network model, this can be done by creating a class that extends `Membranepotential` and by overwriting all its methods.

Some additional methods might be added, that should then also be made available through the "blackbox" class `SamplingNetwork` (by adding regarding methods to it).

A.4.2 PSP shapes

Any new sort of PSP shape should be a subclass of `PspShape`, and it should overwrite its virtual methods `isRectangular` (returning `false`, as this new PSP shape is not a standard rectangular one), and `getPspValue(refractoryValue)`, on which the main focus in the implementation of such a new PSP shape should lie.

If any new specific type of PSP shape should be added, and if the "blackbox" sampling class should be used for creating the network, these additional steps have to be considered:

- In the header file of the class `PspShape`, the name of the *type* of this new PSP shape has to be added to the enumeration of the types.
- Then, in the `SamplingNetwork` class, in the method `addPspShape`, a new *case* for this PSP shape should be added, in which this new type of PSP shape is generated by calling the constructor, using the given parameters. Don't forget to add the new class you created in the `include` part of the `SamplingNetwork.h` file.
- For python usage: Also in the class `SamplingNetworkSimulator`, in the method `getPspType`, a case for the new type has to be added. Here some unique index number has to be associated with any PSP type accessible through other languages.
- For Matlab usage: In order to provide the usage of this new type also to Matlab users, a new case statement has also to be added to the MEX function `initNeurons.cpp` (which is located in the `mex` folder). The new unique index which is associated with this type has to be added as possible value of the variable `pspTypeIdx`, and the regarding new PSP shape type has to be set as `pspType` (while the additionally needed parameters have to be set as `pspParameters`, e.g., λ and $height$).

A.4.3 Refractory mechanisms

In case if another refractory mechanism function $g(\zeta_k)$, additional to the absolute and the relative mechanisms, should be added, the following hints should be considered:

- An `enum Type` entry for the new function definition should be added to the existing one in the `.h` file. This is done by adding the new mechanism's name to the inline definition of the `Type`, e.g., next to `absoluteRefractoryMechanism`, inside the curly brackets, separating the definitions by a column (`,`).
- As a next step one should switch to the `.cpp` file, more exactly to the implementation of the function `computeRefractoryMechanism`: There is a `switch` statement where the new

version of the refractory mechanism should be added as possible `case`. This should be done by adding the statement

```
case newRefractoryMechanism:
    return newRefractoryFunction(R);
    break;
```

to the marked position in the statement, replacing `newRefractoryMechanism` with the name declared before in the `enum Type`, and `newRefractoryFunction` by a desired function name.

- Again back in the `.h` file, the declaration of the new function has to be added:
  In the `private` section, a new function of the type
  `static VectorXd *newRefractoryFunction(int R)` should be added, using the same function name as before in the `.cpp` file.

- The concrete implementation is then written to the `.cpp` file, using the header
  `VectorXd *Refractorymechanism::newRefractoryFunction(int R)`.
  Inside this function, the computation of the τ_{ref} values for $g(\zeta_k)$ are precomputed. Due to performance issues, the best solution is to precompute the set (or vector) of values only once for each setting, and to store these values statically for the execution of the network model.
  In order to do so, before the implementation of the computation a statement has to be added to check, whether the values for exactly the same settings are already stored.

- Therefore, in the `.h` file one should add some vectors (in the `private` section) to store the settings and the precomputed values. The way how to do so might be the same as in the case of the already implemented version of refractory mechanism, namely by adding `static vector<VectorXd> newMechanismList` (to store the precomputed values) and `static vector<int> newMechanismListR` (to store the settings - in this case only the number of refractory states τ_{ref}).

- The definitions should be repeated again in the `.cpp` file, before the implementation of the function, e.g., `vector<VectorXd> Refractorymechanism::newMechanismList` and `vector<int> Refractorymechanism::newMechanismListR`.

- Inside the implementation of the function, after the check (whether the values have already been computed and stored), the set of τ_{ref} refractory mechanism values should be computed in the desired way, and then stored to the vectors defined in the `.h` file. The way how to do so can be looked up in the implementation of the function `sineWave` or `absoluteRefractory`.

- For python usage: In order to enable the accessibility of this new refractory mechanism also through the other languages, the new type has also to be defined in the method `getRefractoryMechanismType` in `NetworkSimulator`: A new case has to be added, which assigns a new unique positive integer number with this new type.

- For Matlab usage: In the MEX function `initNeurons.cpp` in the folder `mex`, a new case has to be added when switching through the values of `gTypeIdx`, specifying the new type as `gType`.

A.4.4 Characteristic functions

Currently, the only implemented type of characteristic function (and therefore also the standard value), is $\gamma(u_k) = e^{u_k}$. If any other characteristic function should be added, this can be done in a similar way as for the refractory mechanisms.

A.5 Sampling process

An overview over the messages and information flow that occurs during the sampling process in the implementation is shown in Figure A.8. In this flow diagram, solid arrows represent function calls, and dashed lines indicates the returned answers to these calls. Note that this is only a very rough overview, showing briefly the interaction among the different classes during sampling.

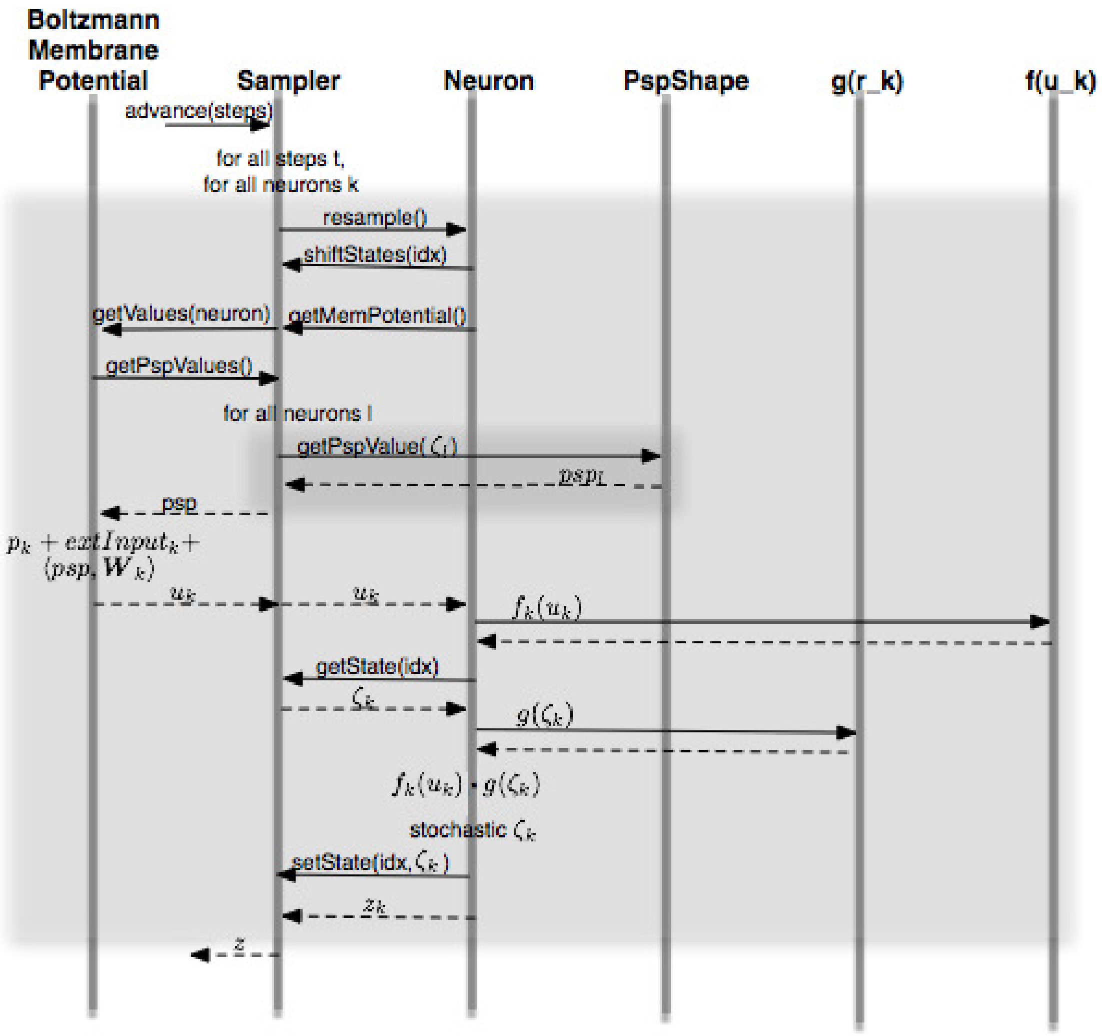

Figure A.8: Rough flow diagram of sampling process

The sampling process is started as soon as someone calls to *advance* the sampler. In this case, the sampler is told how many steps it should advance, and it loops this number of times over all neurons k (the gray box indicates this loop), *resampling* one after the other.

When a neuron is asked to resample itself, it first tells the sampler to shift its delayed states, which refreshes the delayed state list, where the new state to be resampled is then put.

Immediately afterwards, the neuron checks for the current membrane potential value u_k, requesting the sampler, which then forwards this request to the *Boltzmann membrane potential.* In order to obtain the current membrane potential value, the PSP values of all other neurons are

required. The sampler obtains these values by looping over all neurons again, and checking each time for the current PSP value (if not all neurons have rectangularly shaped PSPs, in which case the $\boldsymbol{z} = (\boldsymbol{\zeta} > 0)$ vector can be used immediately).
Then, the neuron's membrane potential value can be computed as

$$u_k = b_k + externalInput_k + \boldsymbol{psp}^\top \boldsymbol{W}_k$$

This value is returned back to the neuron which requested it.

Now the neuron contacts the *firing rate function*, to obtain the precomputed value for the firing rate corresponding to the current membrane potential value $f_k(u_k)$, which corresponds to the solution of the implicit equation system

$$f_k(u_k) \cdot \frac{\sum_{i=1}^{\tau_{ref}} \prod_{j=i+1}^{\tau_{ref}} (1 - f_k(u_k) \cdot g_k(\zeta_k = j))}{\prod_{j=1}^{\tau_{ref}} (1 - f_k(u_k) \cdot g_k(\zeta_k = j))} = e^{u_k}$$

To get the new value of the *refractory mechanism*, the current refractory state of the neuron $\zeta_k = j$ is required (which is again known by the sampler). Having this value, also the pre calculated value $g(\zeta_k = j)$ for the current refractory state and the initially selected kind of refractory mechanism can be obtained.

Having all these values, the spiking probability can be computed, which is equal to the transition probability for going from the current refractory state $\zeta_k = j$ to a spike $\zeta_k = \tau_{ref}$:

$$T^{(k)}_{\tau_{ref},j} = f_k(u_k) \cdot g(\zeta_k = j)$$

The neuron now spikes stochastically - with this known probability $T^{(k)}_{\tau_{ref},j}$. This means, that the refractory state of the current neuron k is updated stochastically, either to $\zeta_k = \tau_{ref}$, or to $\zeta_k = \max\{0, \zeta_k - 1\}$.

After this computation, the neuron's state is updated, and the new $z_k = (\zeta_k > 0)$ value is returned to the sampler.

After the loops have terminated, the sampler returns the newest updated version of $\boldsymbol{z}$ and the sampling process is finished.

A.6 Matlab interface

A Matlab interface for the neural dynamics sampler is available through the use of *mex* functions. For each of these functions that are accessible through Matlab, and that internally call C++ functions, a separate `.cpp` file is stored in the `mex` folder. The communication with these functions is done using structures (`struct`), and a pointer to the object of type `SamplingNetwork` that is created when initializing the sampler, has to be passed to the other functions which then should manipulate this object.

The currently implemented functions are shortly described in the following:

- `[samplerPointer]  = initSampler(struct)`
  This method initializes the sampler, i.e., it creates an object of type `SamplingNetwork` in C++, and returns a pointer to this object as a unsigned integer of 64 bit. It takes as a parameter a structure, which should contain the following fields:

  `.dt` ... a double value indicating the time instances for the sampler in milliseconds. If this field is missing, $dt = 1.0$ is used as a default value

  `.du` ... a double value which indicates the precision for the computation of u_k. If this property is missing, $du = 10^{-4}$ is used automatically

  `.umin` ... a double value indicating the minimum u_k value that is computed. If no value is given, it is set automatically to $umin = -15.0$

  `.umax` ... a double value indicating the maximum u_k value that is computed. In case if this field is not specified, $umax = +15.0$ is used

  `.delay` ... an integer value ≥ 0, setting the global synaptic delay (in milliseconds). This value is not mandatory: If no delay value is specified, $delay = 0$ is used

  Note that the object in memory is persistent, which means that it is not deleted automatically from memory, but it remains there until the method to clear memory is called. It is therefore important to store the address of this object, as soon as it is returned by this method, and to keep it (unmodified) in memory until the method to clear it is called.

- `initNeurons(struct)`
  This method is responsible for creating the neurons, and adding them to the sampling network. The input is again a structure having these fields:

  `.samplingNetwork` ... an unsigned (i.e., positive) integer of 64 bit, containing the address in memory where the sampling network object is located. Note that this is the output of the method which initialized the sampler

  `.amount` ... a positive integer value, which indicates how many neurons with the given parameter settings should be initialized and added to the network. The standard value, if not value has been specified, is $amount = 1$

  `.tauref` ... a double value, representing the τ_{ref} value, which together with the dt value declared when initializing the sampler, defines the refractory period $R = \lfloor\frac{\tau_{ref}}{dt}\rceil$ of this neuron(s). If this field should be missing, $\tau_{ref} = 20.0$ is used

  `.externalinput` ... a double value, containing the pre-computed external input for this neuron(s). If no value is given, $externalinput = 0.0$ is used automatically

- `.g` ... a structure, describing the refractory mechanism. If the whole structure is missing, an absolute refractory mechanism is used. Otherwise, also the following field should be specified:
- `.g.type` ... an integer value, denoting the type of the refractory mechanism:
  - 0 indicates that the refractory mechanism's values $g(\zeta_k)$ have been precomputed for all $\zeta_k = [0, \tau_{ref}]$, and that a vector containing these values is passed as an additional parameter
  - 1 indicates that the absolute refractory mechanism should be used
  - 2 stands for the relative, sine-wave shaped refractory mechanism
- `.g.values` ... a row-vector of length $\tau_{ref} + 1$, containing the pre-computed $g(\zeta_k)$ values. This field might be empty, if the type of the refractory mechanism is different from 0
- `.psp` ... a structure, containing the details about the neuron's PSP shape. If this structure is not specified, the standard value is used, namely rectangular PSP shapes. Otherwise, the following fields are required:
- `.psp.type` ... an integer value, denoting the type of PSP shape to be used:
  - 0 stands for rectangular PSP shapes
  - 1 indicates that the PSP shapes should be exponentially shaped (additive)
  - 2 denotes renewing exponential PSP shapes
- `.psp.lambda` ... a double value, which only has to be set, if the PSP shape is exponential, either additive or renewing. It represents the λ value which indicates the slope of the exponential shape
- `.psp.height` ... a double value, which is necessary for exponential PSP shapes (additive and renewing), indicating the height the exponential function rises in case of a spike

• `initBoltzmannMembranepotential(struct)`
This method creates a Boltzmann membrane potential and sets it for the sampler. The input structure is required to contain these fields:

- `.samplingNetwork` ... again the same uint64 value, indicating the location in memory of the sampling network
- `.type` ... an integer value, indicating the type of initialization parameters that are passed:
  - 0 indicates that the $\boldsymbol{W}$ matrix and the $\boldsymbol{b}$ vector are explicitly stated and contained as additional fields in the structure:
    - `.J` ... a matrix of size $numberOfNeurons \times numberOfNeurons$, containing the synaptic weights as double values, i.e., $\boldsymbol{W}$.
    - `.pi` ... a row vector of double values of the same length as the $\boldsymbol{W}$ matrix, describing the bias vector $\boldsymbol{b}$
  - 1 denotes that the weights have not been set yet, and therefore the $\boldsymbol{W}$ matrix and the $\boldsymbol{b}$ vector should be some Gaussian distributed values according to the given μ, σ^2. Therefore these fields have to be included in the structure:
    - `.muJ` ... a double value, indicating the mean of the Gaussian distributed values for the $\boldsymbol{W}$ matrix
    - `.sigmaJ` ... a double value, indicating the covariance of the Gaussian distributed values for the $\boldsymbol{W}$ matrix. Therefore, $\boldsymbol{W} \sim \mathcal{N}(\mu_{\boldsymbol{W}}, \sigma_{\boldsymbol{W}})$. Note that the resulting $\boldsymbol{W}$ matrix will automatically be symmetric, and have a 0 diagonal

`.muPi` … a double value, representing the mean of the Gaussian distributed values for the $\boldsymbol{b}$ vector

`.sigmaPi` … a double value, being the covariance of the Gaussian distributed values for the $\boldsymbol{b}$ vector. Therefore the vector is defined as $\boldsymbol{b} \sim \mathcal{N}(\mu_{\boldsymbol{b}}, \sigma_{\boldsymbol{b}})$

`.numNeurons` … a (positive) integer value, denoting the number of neurons in the network

- `prepareSampler(struct)`
  This method resets the sampler and prepares it for the sampling process by advancing the burn-in time. The input struct should contain:

  `.samplingNetwork` … the pointer to the sampler

  `.burnInSteps` … an integer value ≥ 0, indicating the burn-in time in milliseconds

- `[z] = advanceSampler(struct)`
  This method advances the sampler, i.e., it resamples all neurons for a given time period. It returns then a row vector of length *numberOfNeurons*, containing the current $\boldsymbol{z} := (\boldsymbol{\zeta} > 0)$ values of the neurons, after sampling, i.e., a vector of zeros and ones. Its input is a struct, containing:

  `.samplingNetwork` … the pointer to the sampler

  `.steps` … an integer value ≥ 0, indicating the sampling time in milliseconds

- `freeMemory(struct)`
  This methods deletes the sampling network object, and all other object that have been created using the previously mentioned *mex* functions from memory. This should be the last call, before one can delete the address of the sampling network. But it has to be kept in Matlab until this method has been called in order to avoid a memory leak. Its input is therefore a struct, containing only:

  `.samplingNetwork` … the pointer to the sampler

- `setWeights(struct)`
  This method sets the $\boldsymbol{W}$ matrix to the given one. Thus, the required input is:

  `.samplingNetwork` … the pointer to the sampler

  `.J` … a matrix of the same size as the previous $\boldsymbol{W}$ matrix was, containing the new weights, that should be set

- `updateWeights(struct)`
  This method updates the weight matrix, by adding η times the difference between the posterior and the prior samples to the current $\boldsymbol{W}$ matrix. The input is a structure, containing:

  `.samplingNetwork` … the pointer to the sampler

  `.samplePosterior` … a row vector of zeros and ones, of length *numberOfNeurons*, being some posterior sample

`.samplePrior` ... a row vector of zeros and ones, of the same length as the posterior sample, containing the prior sample. Note that in case if this prior sample is missing, the current $\boldsymbol{z}$ values of the neurons in the network are used instead. Therefore, this value is not necessarily needed

`.eta` ... a double value, indicating the learning rate η

- `addToWeights(struct)`
  This method updates the weight matrix $\boldsymbol{W}$ by adding to each of its entries W_{kl} a given value δW_{kl}. The input structure should be composed of:

  `.samplingNetwork` ... the pointer to the sampler

  `.deltaJ` ... a matrix of the same size as the current weight matrix $\boldsymbol{W}$ is, containing the double values that should be added to the weight matrix

- `[W] = getWeights(struct)`
  This method returns the current weight matrix $\boldsymbol{W}$. Its input is:

  `.samplingNetwork` ... the pointer to the sampler

- `[b] = getPi(struct)`
  This method returns the $\boldsymbol{b}$ vector, taking as input a structure, containing:

  `.samplingNetwork` ... the pointer to the sampler

- `setClamped(struct)`
  This method is used to clamp (or unclamp) a specified neuron to be either active or inactive all the time. Its input structure should consist of:

  `.samplingNetwork` ... the pointer to the sampler

  `.neuronIndex` ... an integer value, being the index of the neuron to be clamped (or unclamped). Note that the index is set in Matlab format, i.e., the indices are starting from 1

  `.clamped` ... a double value, being > 0 if the neuron should be clamped, or 0 if the neuron should not be clamped

  `.value` ... if the neuron should be clamped, a double value > 0 for clamping the neuron to be active, or 0 to clamp it to be inactive. If the neuron should not be clamped, this value is not necessary

- `setExternalInput(struct)`
  This method is responsible for setting the pre-computed external input to a specified neuron. It takes as input:

  `.samplingNetwork` ... the pointer to the sampler

  `.neuronIndex` ... an integer value, indicating the index of the neuron, whose external input should be set. Note that the index is set in Matlab format, i.e., the indices are starting from 1

  `.input` ... a double value, representing the pre computed value of the external input to the given neuron

- `setAllExternalInputs(struct)`
  This method can be used to set the external inputs of all neurons in the network instantaneously, by passing all the values at a time, specifying:

  `.samplingNetwork` ... the pointer to the sampler

  `.inputs` ... a row vector, containing $numberOfNeurons$ double values, being the pre computed values for the external inputs for all the neurons that should be set

- `[z] = getActivity(struct)`
  This method gets the current states of all the neurons in the networks, and returns values, indicating whether the neurons are currently active or inactive. Therefore, it returns a row vector of length $numberOfNeurons$, containing the current $\boldsymbol{z} := \boldsymbol{\zeta} > 0$ values of all neurons in the network, i.e., a row vector of zeros and ones. It requires, as input, a structure, containing:

  `.samplingNetwork` ... the pointer to the sampler

- `[r] = getStates(struct)`
  This method gets the current refractory states of the neurons, and returns their relative values. The values are relative to the neurons refractory periods, because these might vary from neuron to neuron. Therefore, the method returns a row vector, consisting of $numberOfNeurons$ rows, containing the current $\frac{\boldsymbol{\zeta}}{\tau_{ref}}$ values of the neurons in the network, each one being in the interval $[0, 1]$. The required input is:

  `.samplingNetwork` ... the pointer to the sampler

- `[spikes] = getSpikes(struct)`
  This method checks which neurons in the network are currently spiking. It returns a row vector of length $numberOfNeurons$, containing the current $\boldsymbol{\zeta} = \tau_{ref}$ values, i.e., a vector of zeros and ones. The method requires the input to be a structure with the field:

  `.samplingNetwork` ... the pointer to the sampler

- `storeFiringRate(struct)`
  This method should be called before the sampling process starts. It causes the network to track the firing rate during sampling, such that this firing rate can be obtained after the process. It requires, as input, the following fields in the structure:

  `.samplingNetwork` ... the pointer to the sampler

  `.store` ... a double value, where > 0 indicates that the firing rate should be tracked, and 0 means that no tracking of the firing rate is desired

- `[firingRate] = getFiringRate(struct)`
  This method can only be used in combination with the previously explained one: If the tracking of the firing rate was turned on before, this method returns the observed firing rate in Hertz as an output. Otherwise it returns 0. It again requires the address of the sampling network object as input:

  `.samplingNetwork` ... the pointer to the sampler

- `storeEvolutionOfSpikes(struct)`
  This method switches the tracking of the spike train evolution on: It causes the network to store the spikes $\boldsymbol{\zeta} = \tau_{ref}$ for all neurons over a given period of time. It requires as input:

  `.samplingNetwork` ... the pointer to the sampler

  `.store` ... a double value, being > 0 for switching on tracking, or 0 to not store the spike train evolution

  `.steps` ... a positive integer number, that indicates - if the evolution should be stored - for how long (in milliseconds) this should be done

- `[spikeEvolution] = getEvolutionOfSpikes(struct)`
  This method is again only available in combination with the previous one: If the evolution of spikes in the network was stored over some time period, then a matrix of $numberOfNeurons$ rows and *trackedTimeSteps* columns containing the regarding $\boldsymbol{\zeta} = \tau_{ref}$ values is returned by this function. The required input is:

  `.samplingNetwork` ... the pointer to the sampler

- `storeEvolutionOfDistribution(struct)`
  This method is responsible for switching on (or off) the tracking of the evolution of $\frac{\boldsymbol{\zeta}}{\tau_{ref}}$ values of all neurons in the network over a specified period of time. Its input structure has to contain the following fields:

  `.samplingNetwork` ... the pointer to the sampler

  `.store` ... a double values, where a value > 0 means that the evolution of relative refractory states should be tracked, and 0 switches tracking off

  `.steps` ... a positive integer number, indicating the time period in milliseconds, for which the evolution of relative refractory states should be stored

- `[rEvolution] = getEvolutionOfDistribution(struct)`
  This method returns the tracked evolution of relative refractory states, if tracking has been switched on before, or 0 otherwise. The returned matrix is of size $numberOfNeurons \times$ *timeSteps*, and it contains double values in the interval $[0, 1]$, indicating the $\frac{\boldsymbol{\zeta}}{\tau_{ref}}$ values of all neurons for the previously specified time period. Its input is:

  `.samplingNetwork` ... the pointer to the sampler

- `storeEvolutionOfActivity(struct)`
  This method turns the tracking of the evolution of neuron activities $\boldsymbol{z} := (\boldsymbol{\zeta} > 0)$ on or off. It requires as input:

  `.samplingNetwork` ... the pointer to the sampler

  `.store` ... a double value, being 0 if the evolution of $\boldsymbol{z}$ should not be tracked, or > 0 if it should be tracked

  `.steps` ... a positive integer number, indicating the time period for tracking the neurons' activities in milliseconds. This value is not required, if the evolution should not be stored

- `[zEvolution] = getEvolutionOfActivity(struct)`
  This method returns a matrix, containing the evolution of the neurons activities over time. It has $numberOfNeurons$ rows, and $numberOfMilliseconds$ columns, containing the regarding $\boldsymbol{z}$ values of all neurons. If tracking has not be switched on before, it returns 0. Its input structure must contain:

  `.samplingNetwork` ... the pointer to the sampler

- `storeDistribution(struct)`
  This method can be called to enable tracking of the sampled distirbution of the neuron's activity $p(\boldsymbol{z})$. Not that storing these probability distributions is only available for a limited amount of neurons in the network, as $2^{numberOfNeurons}$ different distributions are possible, and a value for each of them has to be computed (and stored). Therefore, it is also possible to specify the indices of the neurons that should be tracked. The required parameters are:

  `.samplingNetwork` ... the pointer to the sampler

  `.store` ... a double value, where 0 indicates that the sampled distribution should not be stored, and > 0 means that it should be stored

  `.neuronIdx` ... a row vector of positive integer values, that, if the sampled distribution should be stored, can indicate the indices of the neurons that should be tracked. In this case, only these neurons are considered for the distribution. If this vector is not given, automatically all neurons are tracked. Note that the indices should be in the form of Matlab indices, i.e., starting from 1 (for the first item).

- `[pZ] = getDistribution(struct)`
  This method is only available in combination with the previous one: If the distribution was stored during sampling, it can be obtained using this method. It returns a row vector, containing the relative frequencies of the sampled state vectors $\boldsymbol{z}$, either of all neurons, or of the previously indicated ones. Its input is a structure, containing:

  `.samplingNetwork` ... the pointer to the sampler

- `[qZ] = getAnalyticalDistribution(struct)`
  This method computes the analytical probability distribution of all possible network states, in terms of the activity of the neurons $\boldsymbol{z}$. It might be helpful to compare this analytically computed distribution to the sampled relative frequencies obtained by the previously described method. Note that also here the number of values to be computed is $2^{numberOfNeurons}$. Therefore the use of this method is only recommended if the number of neurons is relatively low. The method requires as input (in order to obtain the number of neurons in the network):

  `.samplingNetwork` ... the pointer to the sampler

- `[kl] = getKLtoAnalyticalDistribution(struct)`
  This method is responsible for computing the Kullback-Leibler divergence between the relative frequencies of the sampled network states, and the analytically computed probability distribution. It returns a double value, being the KL distance between the two distributions. As input it takes:

  `.samplingNetwork` ... the pointer to the sampler

A.7 Python interface

A python interface through the use of *SWIG* (Simplified Wrapper and Interface Generator, `www.swig.org`) is available. The methods that can be called from python are written in C++ and are converted into a python object at compile time. The instructions how the wrapping is done are specified in the `swig` folder, i.e., the input files for swig (`.i` files) as well as the specifications how the `Eigen` types in C++ are mapped to python's `scipy` types.

In order to be able to use the python interface, it has to be imported and initialized, such that an object is available in python on which the sampler methods can be called. For various methods `scipy` is needed as well.

```
import sampler
import scipy
net = sampler.SamplingNetworkSimulatorFloat()
```

After this initialization, all the available methods can be called on the `net` object. A short description of these methods is given in the following:

```
void createSampler(double dT=1.0,
    double uDiff=0.0001, double uMin=-15, double uMax=15,
    unsigned globalDelay=0)
```

This method calls the constructor of the sampler and initializes it with the basic settings, i.e., the *dt* value, the values specifying the precision of the membrane potential values u_k, being u_{diff}, u_{min} and u_{max}, as well as the possible global delay for the whole network.

```
void createNeuron(double tau_ref=20.0, unsigned gType=1, double input=0,
    unsigned pspType=0, double height=1, double lambda=0.95, double lambda2=0.5)
```

```
void createNeuron(const MatrixXf& gPrecomputed, double tau_ref=20.0,
    double input=0, unsigned pspType=0, double height=1, double lambda=0.95,
    double lambda2=0.5)
```

```
void createNeurons(unsigned amount=1, double tau_ref=20.0, unsigned gType=1,
    double input=0, unsigned pspType=0, double height=1, double lambda=0.95,
    double lambda2=0.5)
```

```
void createNeurons(unsigned amount, const MatrixXf& gPrecomputed,
    double tau_ref=20.0, double input=0,
    unsigned pspType=0, double height=1, double lambda=0.95, double lambda2=0.5)
```

All of these methods can be used to initialize one or more neurons, and to add them to the sampling network. For this initialization the following parameters can be set: τ_{ref} specifying the refractory period, the pre computed external input to this neuron(s), and either the type of the refractory mechanism $g(\zeta_k)$ identified by its unique id or a column vector containing the values for all possible ζ_k values. Also the PSP shape type can be specified by its unique id, and depending on the type of PSP shape the additional parameters (*height*, λ, λ_2) have to be specified or not. If more than one neuron of this kind should be created (i.e., one of the second two variants is used), also the amount of neurons has to be given.

```
void createBoltzmannMembranePotential(unsigned numNeurons,
    double muJ, double sigmaJ, double muPi, double sigmaPi)
```

```
void createBoltzmannMembranePotential(const MatrixXf& J, const MatrixXf& pi)
```

One of these two methods has to be used to create and initialize the Boltzmann membrane potential. One option is to specify the number of neurons in the network, and the values $\mu_{\boldsymbol{W}}, \sigma_{\boldsymbol{W}}$ according to which the values of the weight matrix $\boldsymbol{W}$ should be Gaussian distributed, being $\boldsymbol{W} \approx \mathcal{N}(\mu_{\boldsymbol{W}}, \sigma_{\boldsymbol{W}})$, as well as $\mu_{\boldsymbol{b}}, \sigma_{\boldsymbol{b}}$ specifying the normal distribution of the b vector, namely $\boldsymbol{b} \approx \mathcal{N}(\mu_{\boldsymbol{b}}, \sigma_{\boldsymbol{b}})$. The other variant is to specify the $\boldsymbol{W}$ matrix and the column bias vector $\boldsymbol{b}$ explicitly, i.e., passing `scipy` arrays as parameters.

```
void prepareSampler(unsigned burnInSteps=0)
```

```
virtual int check()
```

```
virtual int reset(double dt)
```

One of these methods has to be used after the initialization. They check whether all values are set and whether they are valid. Only after one of them has successfully been executed, the sampling network can be advanced. The `prepareSampler` method additionally advances the sampler for the given number of milliseconds after checking the values, i.e., it advances the burn-in time, such that the sampler can be used immediately.

```
void advance(MatrixXf *res, unsigned steps=1)
```

This method advances the sampler for the given number of steps, i.e., it resamples all neurons in the network for the given number of milliseconds. It returns the $\boldsymbol{z}$ values of all neurons in a column vector.

```
void storeProbabilities(bool store)
```

```
void storeProbabilities(const MatrixXf& neuronIdx)
```

```
void getSamplingProbabilities(MatrixXf *res)
```

```
void getAnalyticalProbabilities(MatrixXf *res)
```

```
double getKLtoAnalyticalDistribution()
```

These methods are responsible for storing the relative frequencies of the neurons' activities $\boldsymbol{z}$ during sampling. If there is a very large number of neurons in the network, it is recommended to only give some neurons' indices in a column vector when specifying to store the distributions, as this requires a considerably high amount of memory. The methods that specify to store the probability distributions have to be called before the sampling process (such that the distributions are tracked). After the sampling process, the sampled distribution can be obtained using the `get` method. The analytically computed distribution might be requested using the second `get` method. Additionally, also the Kullback-Leibler divergence between the sampled distribution and the analytically computed distribution can be obtained using the method above.

```
void storeEvolution(bool store, unsigned numSteps=0)
```

`void getStateEvolution(MatrixXf *res)`

If someone wants to track the evolution of the relative refractory values $\boldsymbol{\zeta}/\tau_{ref}$ over time, then the above two methods might be useful. Before the sampling process, tracking of the refractory states' evolution can be switched on. Already at this point, the number of steps (or milliseconds) over which the states should be tracked, has to be specified. After sampling, the evolution of all neurons over time is returned in a matrix.

`void storeSpikeEvolution(bool store, unsigned numSteps=0)`

`void getSpikeEvolution(MatrixXf *res)`

These methods are helpful, if the evolution of spikes $\boldsymbol{\zeta} = \tau_{ref}$ over time should be tracked. Before the sampling process, tracking can be switched on for a given number of milliseconds. After sampling, the resulting spike evolution is returned in a matrix for all neurons.

`void storeActivityEvolution(bool store, unsigned numSteps=0)`

`void getActivityEvolution(MatrixXf *res)`

To track the activities of all neurons over time $\boldsymbol{z} := \boldsymbol{\zeta} > 0$, these methods can be used. Again, tracking can be switched on for a specific number of milliseconds, and after the sampling process has finished the evolution can be obtained using the second of these methods.

`void storeFiringRate(bool store)`

`double getFiringRate()`

These methods can be used to store and get the firing rate - averaged over all neurons - in Hertz. As it cannot be computed analytically, also this is tracked during the sampling process, and can be obtained afterwards.

`void storeFiringRates(bool store)`

`double getFiringRates()`

These methods can be used to store and get the firing rate of each single neuron in Hertz. Again, the firing rates have to be tracked, and can be returned after some sampling time.

`void setClamped(unsigned neuronIdx, bool one)`

`void setNotClamped(unsigned neuronIdx)`

Each neuron can be clamped to be either active or inactive, or unclamped, specifying the index of the neuron.

`void setExternalInput(unsigned neuronIdx, double input)`

`void setExternalInput(const MatrixXf& input)`

The external input of each neuron can be set by specifying its index and the precomputed value for the external input. It is also possible to set the external inputs of all neurons at a time, by giving the values for all neurons in a column vector.

`void setWeightMatrix(const MatrixXf& W)`

```
void addToWeightMatrix(const MatrixXf& deltaW)
```

```
void updateWeightMatrix(const MatrixXf& samplePosterior,
    const MatrixXf& samplePrior, double eta)
```

```
void updateWeightMatrix(const MatrixXf& samplePosterior, double eta);
```

```
void setWeightMask(const MatrixXf& WMask);
```

```
void getWeightMatrix(MatrixXf *res)
```

The weight matrix $\boldsymbol{W}$ can be set explicitly, or the matrix can be updated in other ways. It is possible to pass a matrix $\Delta\boldsymbol{W}$, which is then added to the original $\boldsymbol{W}$ matrix. Another possibility is to update the weight matrix by passing a column vector containing a posterior sample, a prior sample, and the η value, such that the new matrix can be learned. If no prior sample is given, the current network state is assumed:

$$\Delta\boldsymbol{W} = \eta \cdot \left(\left(\boldsymbol{z}_{posterior}\boldsymbol{z}_{posterior}^{\top}\right) - \left(\boldsymbol{z}_{prior}\boldsymbol{z}_{prior}^{\top}\right)\right)$$

The current weight mask (that defines which elements in the weight matrix should be updated, and which shouldn't) can also be set. By default it is a matrix of ones, with a zero diagonal. The current weight matrix $\boldsymbol{W}$ can be obtained using the last of these methods.

```
void setPiVector(const MatrixXf& b)
```

```
void addToPiVector(const MatrixXf& deltaB)
```

```
void updatePiVector(const MatrixXf& samplePosterior,
    const MatrixXf& samplePrior, double eta)
```

```
void updatePiVector(const MatrixXf& samplePosterior, double eta);
```

```
void getPiVector(MatrixXf *res)
```

The $\boldsymbol{b}$ vector might also be set explicitly, but additionally it is possible to update it in other ways, i.e., either by specifying some equally shaped vector $\Delta\boldsymbol{b}$, which should then be added to the original vector, or by passing a column vector containing a posterior sample, another one being the prior sample (not needed if the current values should be taken), and the learning rate η, which is a scalar. In this case, the update is done as:

$$\Delta\boldsymbol{b} = \eta \cdot (\boldsymbol{z}_{posterior} - \boldsymbol{z}_{prior})$$

The weight mask, defining the single vector elements to be updated or not, can be set and changed. By default it is a vector of ones. The current vector $\boldsymbol{b}$ can also be requested.

```
void getActivity(MatrixXf *res)
```

This method returns a column vector containing the activity of all neurons $\boldsymbol{z} := \boldsymbol{\zeta} > 0$.

```
void getStates(MatrixXf *res)
```

This method returns the relative refractory values of all neurons in a column vector: $\frac{\boldsymbol{\zeta}}{\tau_{ref}}$

`void getSpikes(MatrixXf *res)`

This method returns a column vector specifying for each neuron whether it spiked right now, or not, i.e., $\boldsymbol{\zeta} = \tau_{ref}$.

`unsigned getNumberOfNeurons()`

This method returns the number of neurons in the network.

As soon as the sampling process is finished, the sampling network and all containing objects are deleted from memory by calling `del net`.

B Appendix: Learning the model

B.1 Weight masks

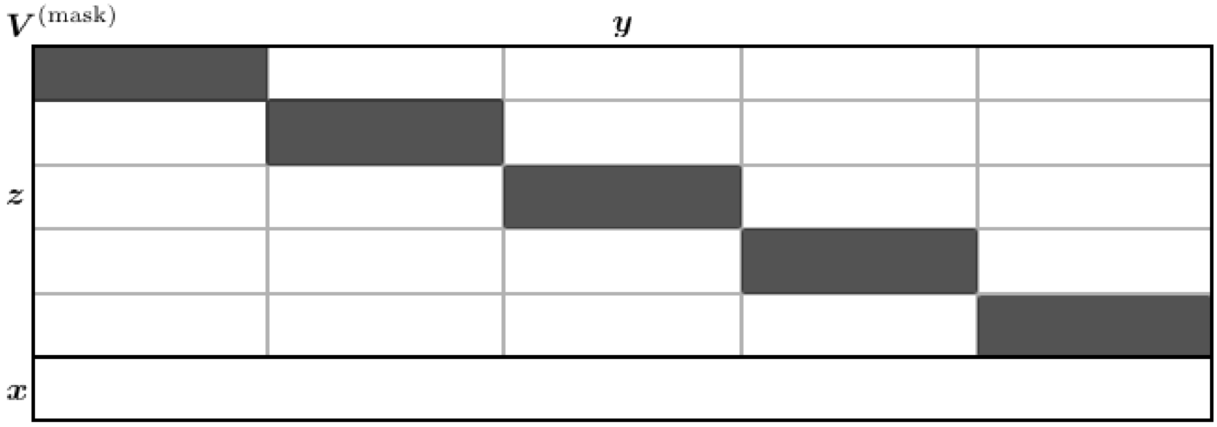

Figure B.1: Mask for weight matrix V, where clamped values, i.e., zeros in the mask matrix, are indicated by white, and values to be modified and learned are painted in dark. The only entries to be learned are the ones defining the receptive fields between layer Y and Z. Of course, the input layer neurons y_i are not connected to the neurons x_k in the top layer X.
Note that the visualized grid only separates the different groups, not the single neurons, from which there are 36 in each group G_j of neurons y_i, and 4 in each SEM-unit or group of neurons z_k (and x_k).

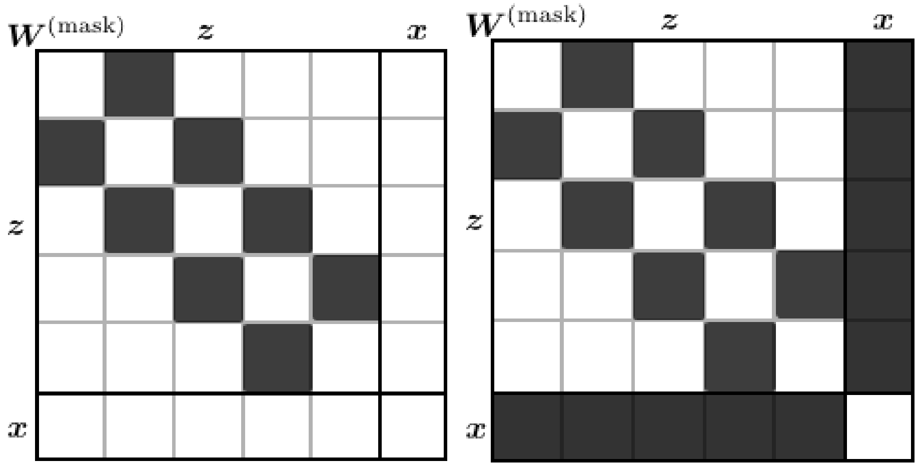

Figure B.2: Symmetric mask for learning the weight matrix $\boldsymbol{W}$ in steps 2 and 3, where white fields indicate zero-values, and ones are indicated in dark: The first picture shows the mask for learning only the synapses inside the lateral layer Z, i.e., the connections between "neighboring" SEM-units, while the connections inside the single units, and the connections between units with receptive fields that do not touch each other directly are clamped, and their entries in the mask matrix are 0.
The second picture shows all the feedforward-feedback synaptic connections that are learned, also the ones between the lateral layer Z and the top layer X, where the receptive files of the higher layer neurons $\boldsymbol{x}$ is the entire lower layer, i.e., all neurons $\boldsymbol{z}$, and thus all these connections are allowed to be adapted (or learned).
Note again that the grid does only divide the different SEM-units, or groups, while the single neurons - in this case exactly four in each row and column per group - are not visible.

B.2 Learned weights

B.2.1 Step 1: Feedforward Hebbian weights

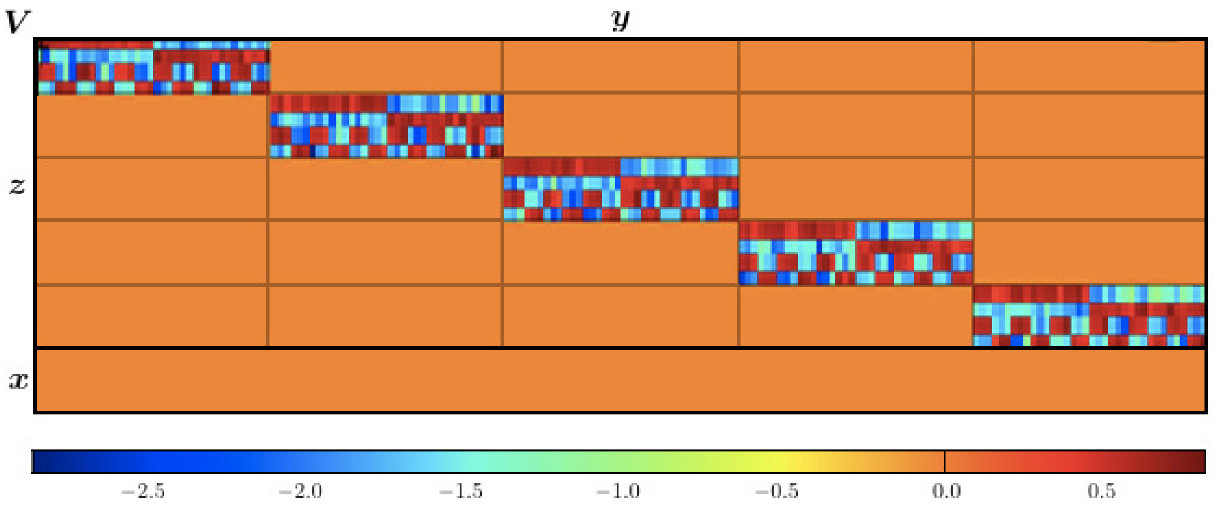

Figure B.3: Learned weight matrix $\boldsymbol{V}$, sorted: Connections between input layer neurons $\boldsymbol{y}$ and lateral layer neurons $\boldsymbol{z}$. In each SEM unit the same neuron is responsible for recognizing a certain pattern.
Each row in the matrix represents the connections of different input neurons $\boldsymbol{y}$ to one of the lateral layer neurons $\boldsymbol{z}$. Thus a row indicates some “flattened” representation of an input pattern. A column indicates the connections from a single input neuron, i.e., from one input pixel, to the different lateral layer neurons $\boldsymbol{z}$. Due to the structure of the used patterns, exactly two lateral layer neurons are inhibited by each of the input pixels, and exactly the two other ones are excited.
Therefore, for a better overview over the system the weight matrix can be sorted, row-by-row, such that in each receptive field the same SEM-neuron (i.e., the one with the same internal index) is responsible for recognizing the same input pattern.

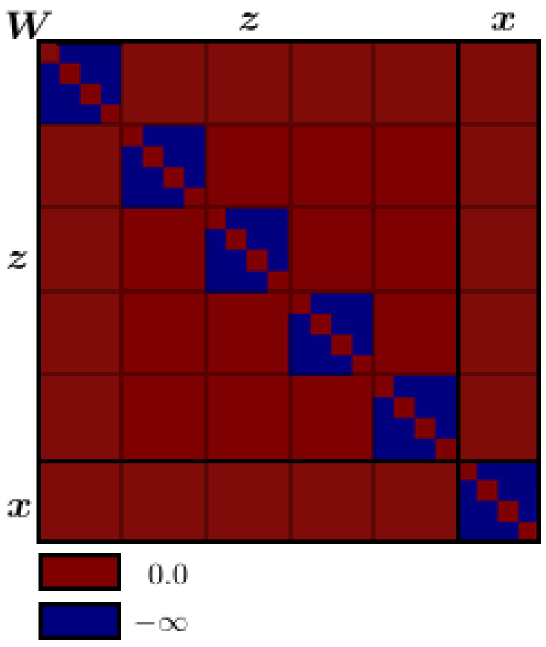

Figure B.4: Weight matrix $\boldsymbol{W}$ in its still unlearned form, as it has been initialized, namely having only (inhibitory) connections among neurons having the same receptive field.

B.2.2 Step 2: Lateral layer weights

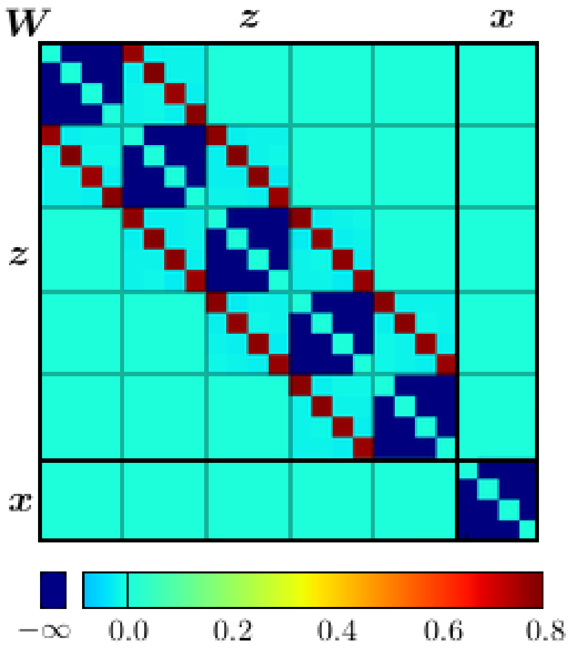

Figure B.5: Weight matrix $\boldsymbol{W}$: The resulting weights after having learned the synapses inside the lateral layer Z (again sorted). It can be seen that the units which “recognize” the same patterns stimulate each other (if they have receptive fields which are next to each other - otherwise the connection is not established), while the units responsible for different patterns inhibit each other slightly.

B.2.3 Step 3: Adding a third layer

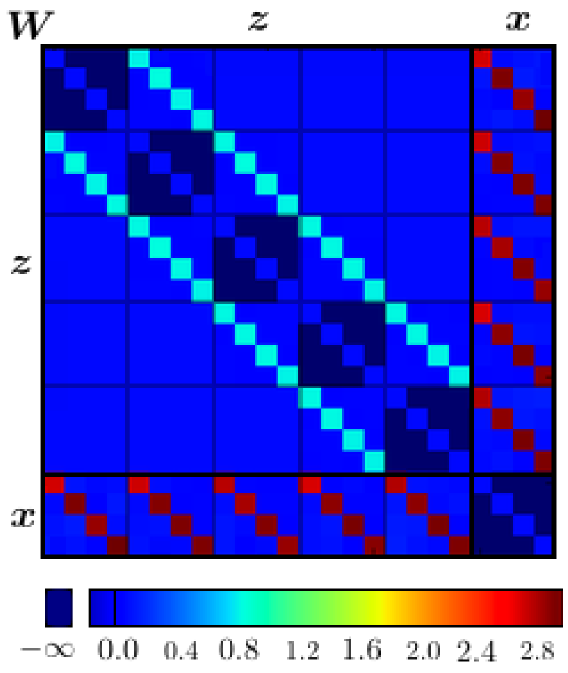

Figure B.6: The final, learned weight matrix $\boldsymbol{W}$, after having learned the connections among the lateral layer neurons $\boldsymbol{z}$, and additionally between lateral layer neurons $\boldsymbol{z}$ and top layer neurons $\boldsymbol{x}$.
It is shown that each neuron in the lateral layer, which is responsible for some specific shape of input pattern, stimulates the same top layer neurons. This means that the model correctly learned that the input should usually consist of five same patterns.

B.2.4 Step 4: Expanding the third layer

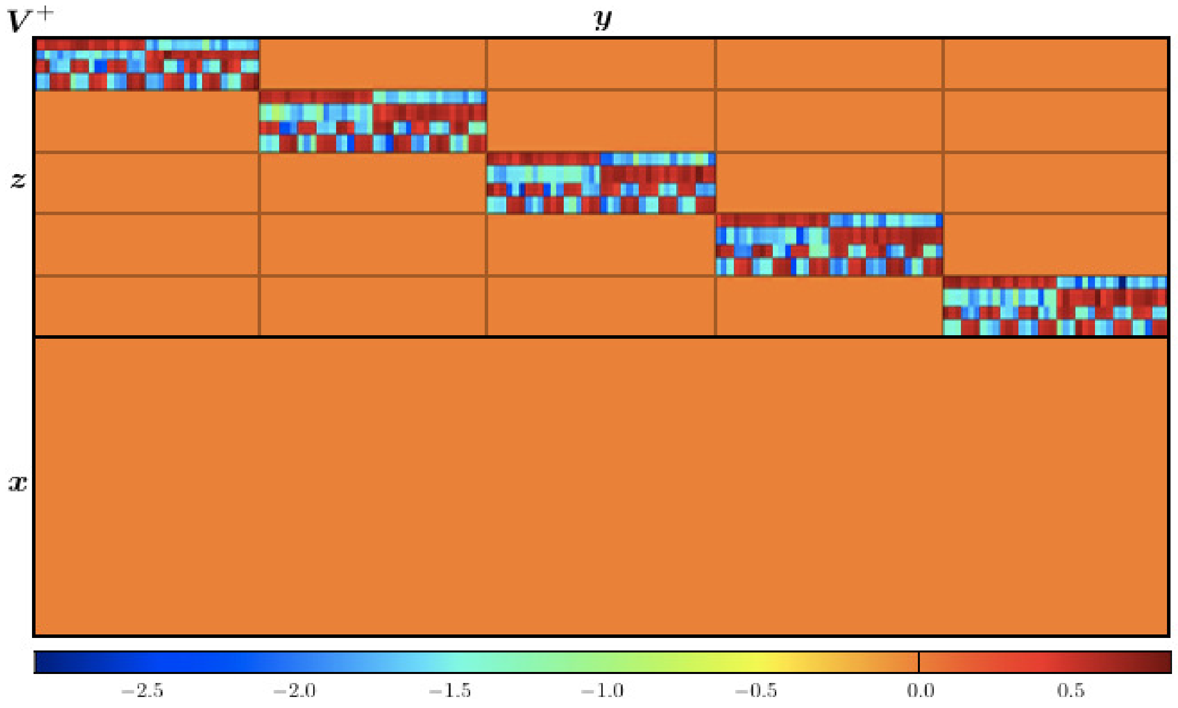

Figure B.7: The weight matrix $\boldsymbol{V}^+$, encoding the synaptic weights from the input layer Y to the lateral layer Z (and the top layer X). Now the feed-forward weight matrix gets larger according to the growing top layer X.
Also in this version, the weights have been sorted, such that the lateral layer neuron with the same "index" is responsible for the same shape of input pattern in each of the SEM-like lateral layer units. Also the height of the feedforward weights remains the same as in the previously used model. The only change in this weight matrix is the expansion of the X-layer part. As the weights going from the input layer to the top layer are nevertheless set (and clamped) to zero, this only expands the matrix with additional zero-rows on the bottom, as it can be seen in the figure.

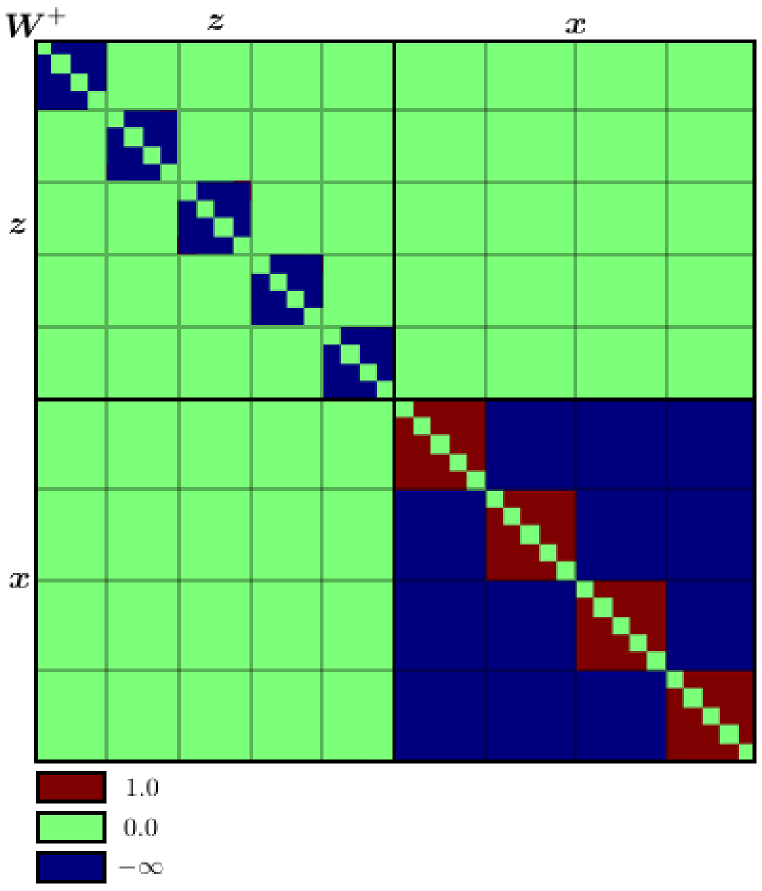

Figure B.8: The expanded weight matrix $\boldsymbol{W}^+$, encoding the connections among lateral layer neurons $\boldsymbol{z}$ (in the upper left square), the (initialized and fixed) connections among top-layer neurons $\boldsymbol{x}$ (in the square on the right bottom of the matrix), as well as the connections between lateral layer neurons $\boldsymbol{z}$ and top layer neurons $\boldsymbol{x}$.
Most of the initial weight matrix is similar or equal to the kind of weights used in the previous steps. However, the left bottom square, i.e., the connections among the top layer neurons have changed now: Instead of the inhibition among all neurons, now there is some positive excitation for neurons that belong to the same population, while all neurons of different populations still inhibit each other strongly. This should cause only one population to be active at a time, allowing for a clear "decision" about the image at any point in time.

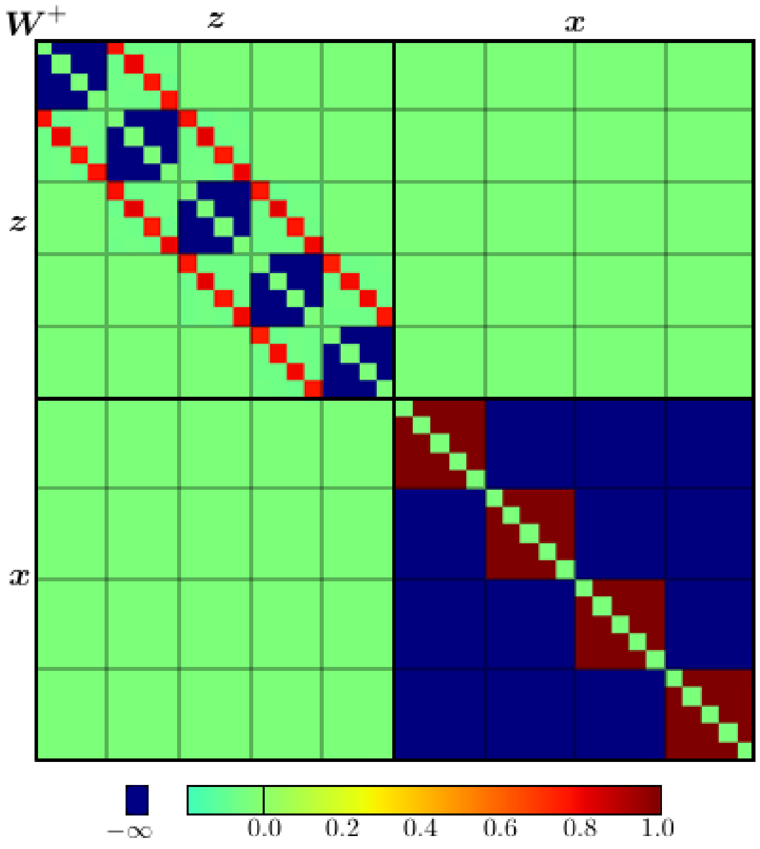

Figure B.9: The weight matrix $\boldsymbol{W}^+$, after having learned the lateral layer connections, i.e., the synapses between neurons $\boldsymbol{z}$ with receptive fields next to each other. As in the previous steps (without the top-layer populations), each lateral layer neurons specialized for one specific input pattern (encoded in $\boldsymbol{V}^+$ stimulates the neurons responsible for the same pattern of the other SEM-like units.

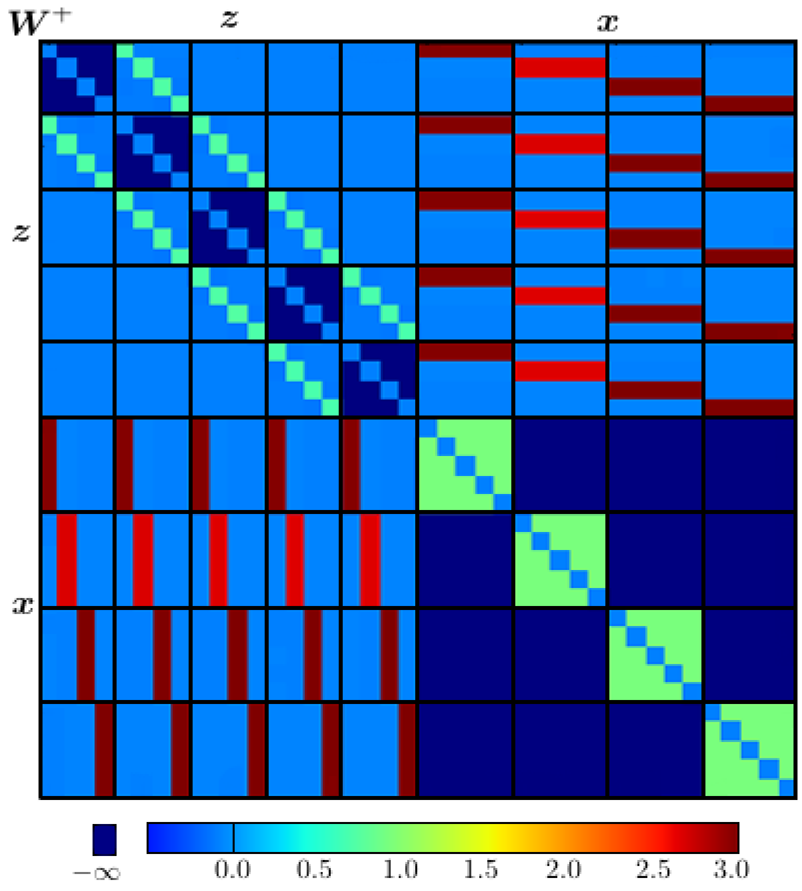

Figure B.10: Weight matrix $\boldsymbol{W}^+$, after having learned not only the lateral layer connections, but also the top-down, bottom-up connections between the neurons of the two higher layers.
It is shown that the weight of the single synapses between lateral and top layer neurons is similar to the weights when not using populations, even though now each lateral layer neuron stimulates more (in this case $L = 5$) top layer neurons, and gets feedback from them.

B.3 Simulating the experiment by Lee: PSP values

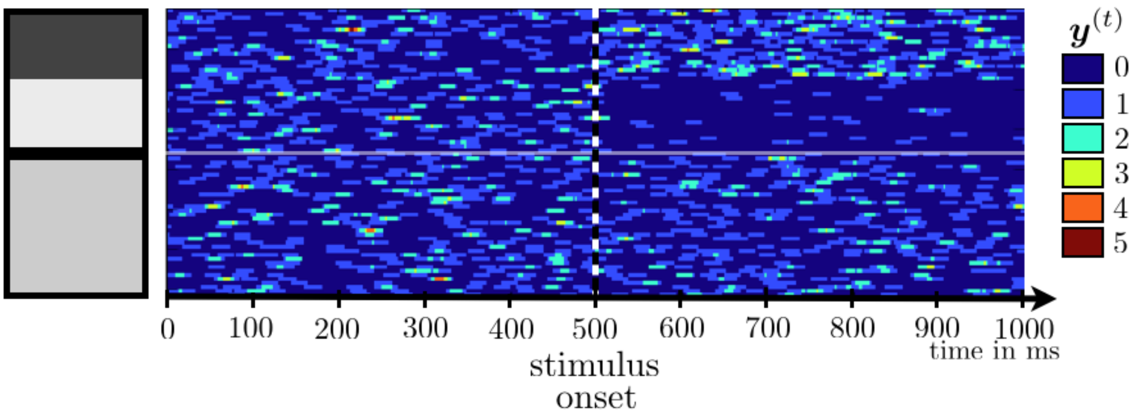

Figure B.11: Evolution of the input neurons' PSP values of the first and the second image subpart: For 500 ms the input represents the average pattern; after the stimulus onset, the first part reflects the pattern shown on the left, while the second image part still shows the average pattern.
The color indicates the "height" of this unit's state, counting the number of spikes occurred during the past $\tau_y = 20.0$ milliseconds: Dark blue indicates that no spike occurred, while yellow or red colors indicate a high activity in the previous period.

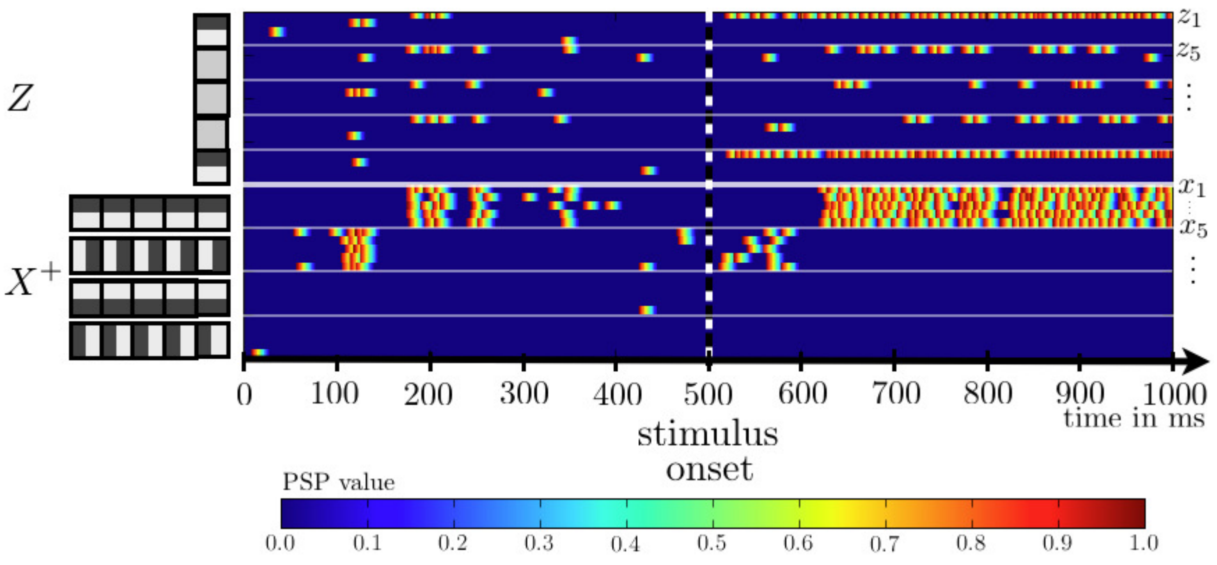

Figure B.12: PSP values of lateral and top layer units: After 500 ms the first and the last image subpart get direct external input, and their firing pattern changes accordingly after that. With some delay, the top layer units corresponding to the given input pattern get this information and switch to the same state. After an additional delay, the units without direct external input in the lateral layer reflect the same pattern with lower firing rates.
The colors indicate the current hight of the PSP values of the single neurons, where alpha-shaped PSPs with a rising time of about 3 milliseconds were used.

C Appendix: Input Patterns and Pattern Completion

C.1 Learned input images

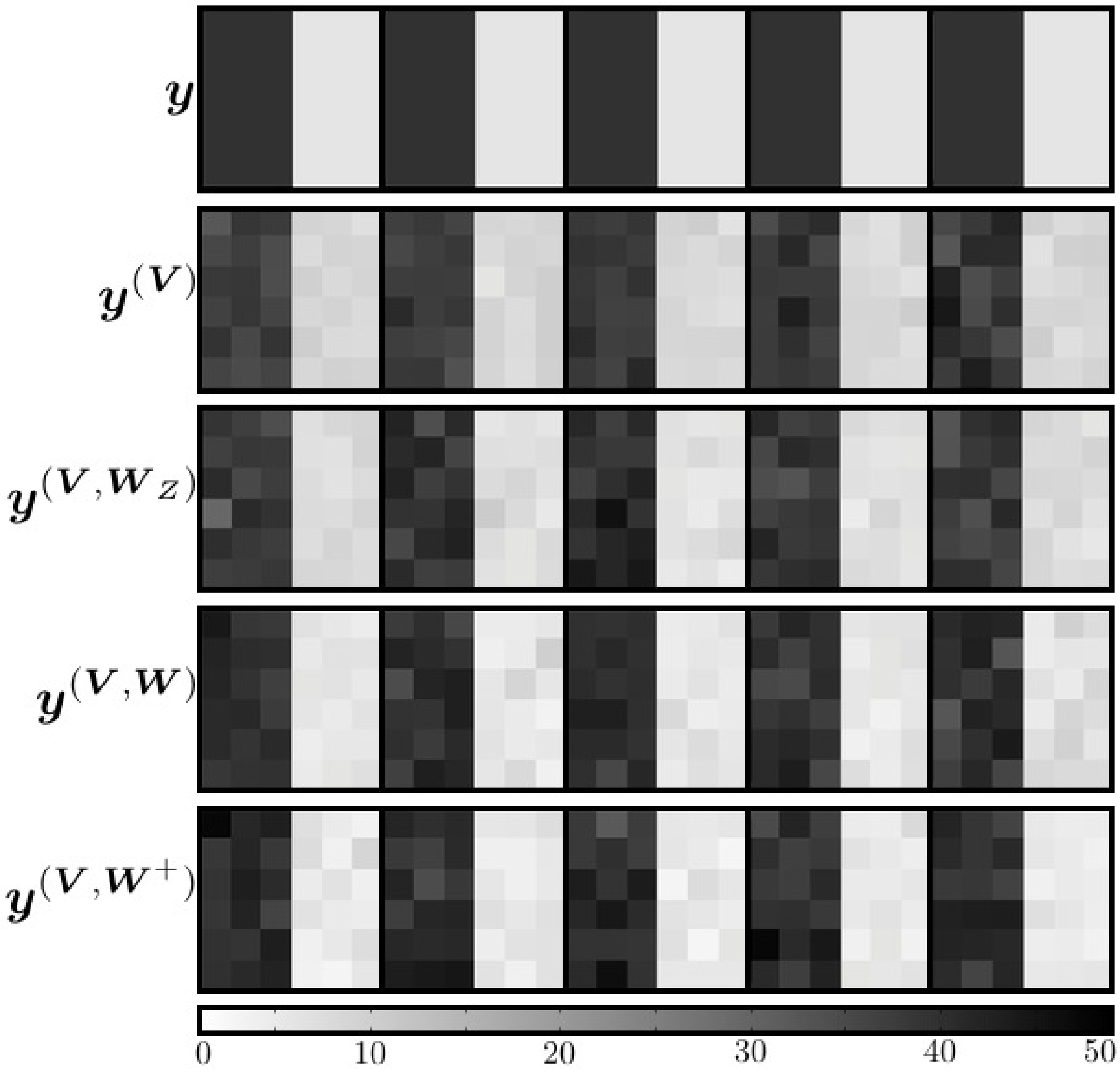

Figure C.1: Learned input image: Firing rate of the input layer neurons $\boldsymbol{y}$
- Average firing rate of the model using only feedforward weights $\boldsymbol{V}$: already recognized
- Average firing rate when also using the lateral layer connections $\boldsymbol{W}_Z$
- Average firing rate after adding the top layer X
- Average firing rate using populations in the top layer

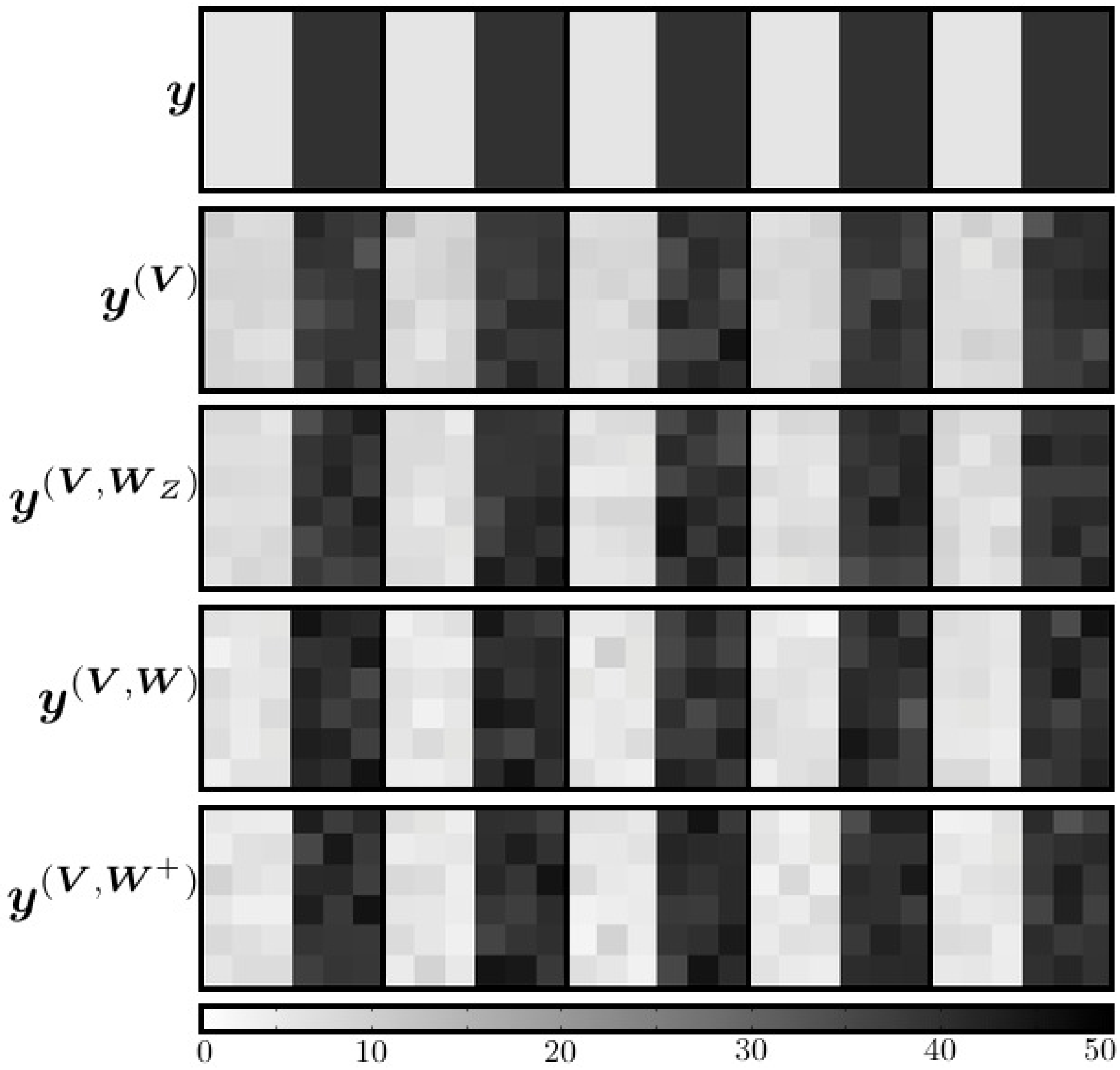

Figure C.2: Learned input image 2:
- Firing rate of the input layer neurons $\boldsymbol{y}$
- Average firing rate of the model using only feedforward weights $\boldsymbol{V}$
- Average firing rate when also using the lateral layer connections $\boldsymbol{W}_Z$
- Average firing rate after adding the top layer X
- Average firing rate using populations in the top layer

Already with the feedforward weights $\boldsymbol{V}$ the image can be recognized and “reconstructed”.

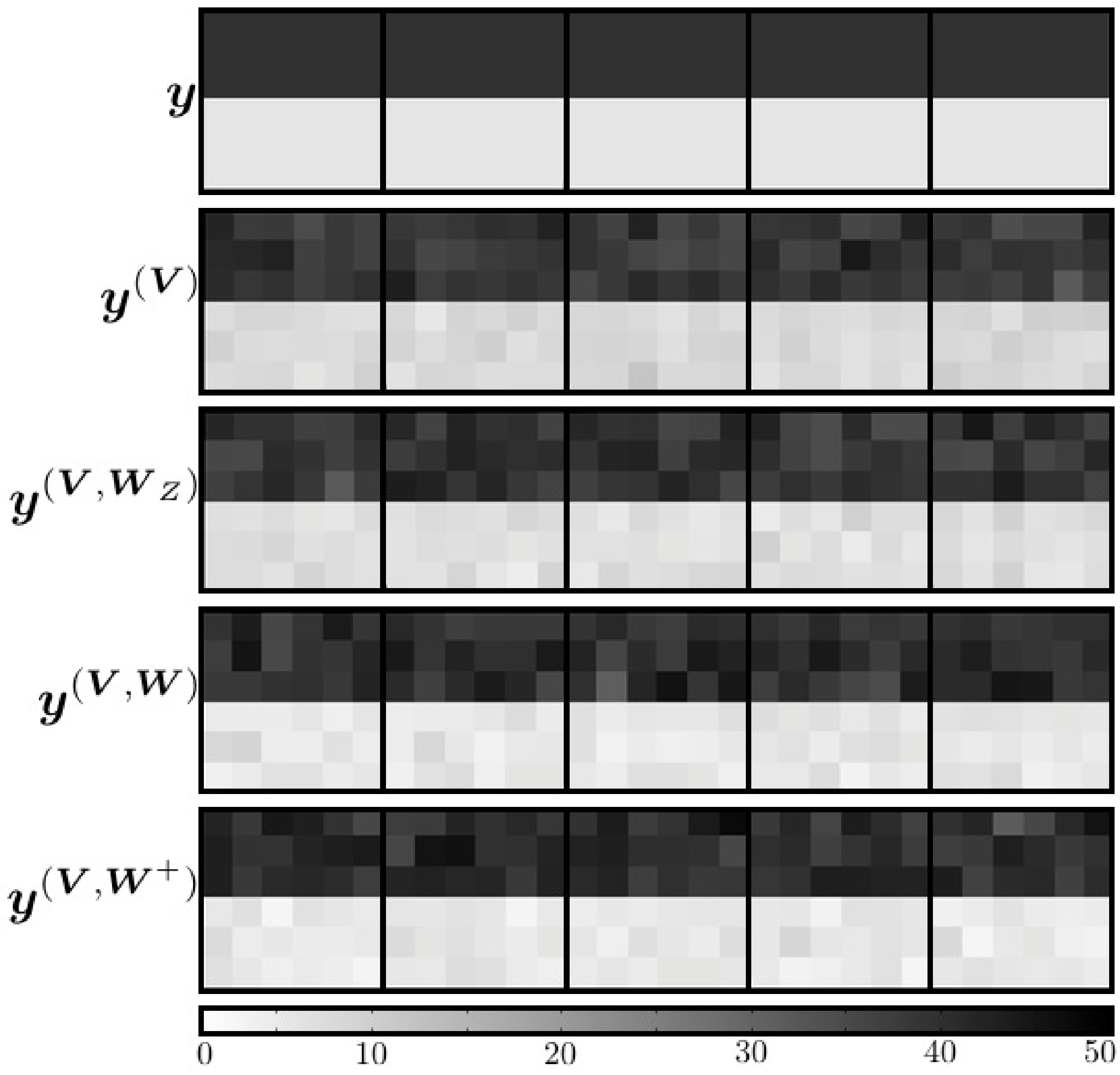

Figure C.3: Learned input image 3:
- Firing rate of the input layer neurons $\boldsymbol{y}$
- Average firing rate of the model using only feedforward weights $\boldsymbol{V}$
- Average firing rate when also using the lateral layer connections $\boldsymbol{W}_Z$
- Average firing rate after adding the top layer X
- Average firing rate using populations in the top layer

Already with the feedforward weights $\boldsymbol{V}$ the image can be recognized and “reconstructed”.

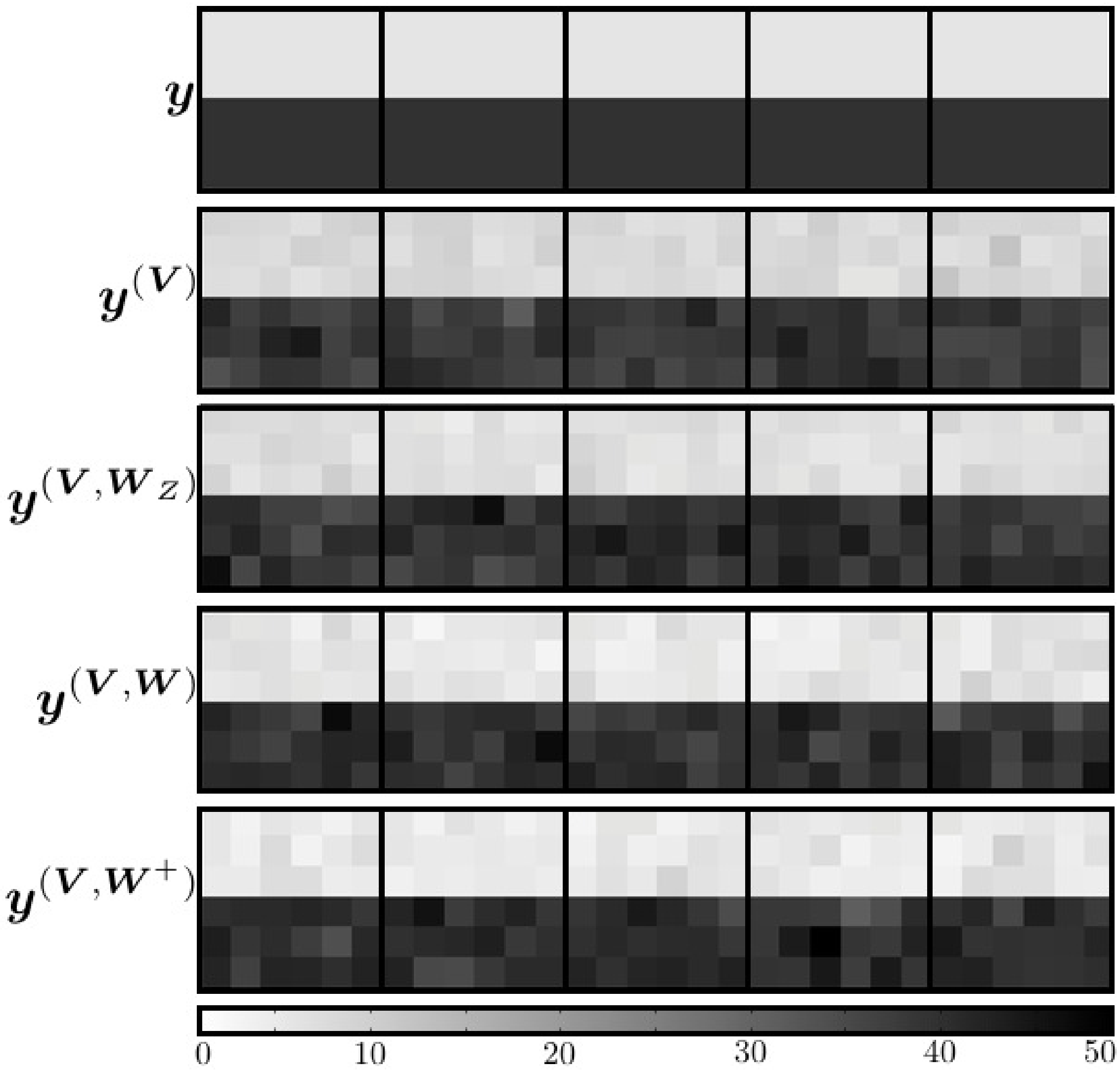

Figure C.4: Learned input image 4:
- Firing rate of the input layer neurons $\boldsymbol{y}$
- Average firing rate of the model using only feedforward weights $\boldsymbol{V}$
- Average firing rate when also using the lateral layer connections $\boldsymbol{W}_Z$
- Average firing rate after adding the top layer X
- Average firing rate using populations in the top layer

Already with the feedforward weights $\boldsymbol{V}$ the image can be recognized and “reconstructed”.

C.2 Missing part in image

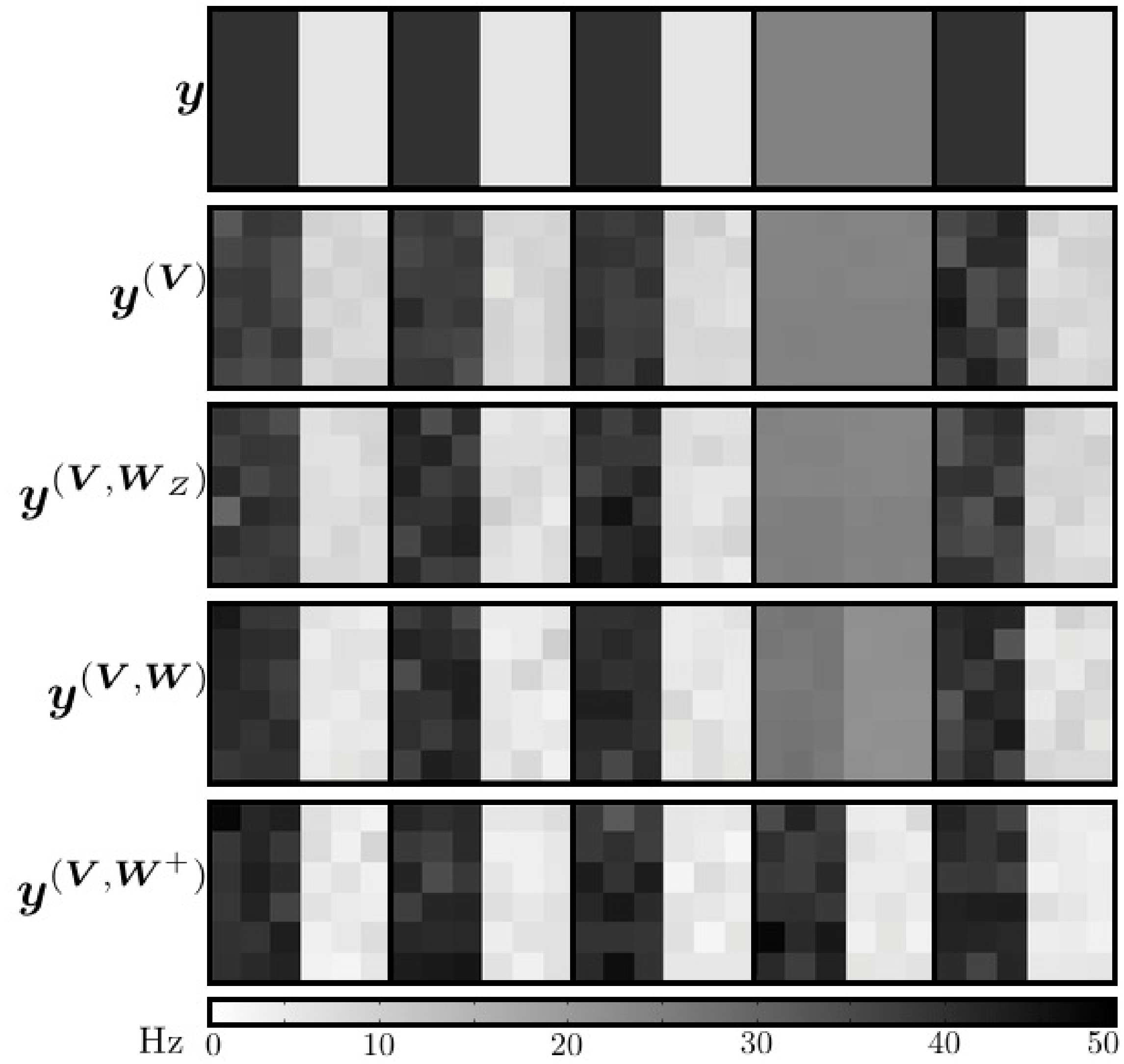

Figure C.5: Incomplete input image: Firing rate of the input layer neurons $\boldsymbol{y}$
- Average firing rate of the model using only feedforward weights $\boldsymbol{V}$
- Average firing rate when also using the lateral layer connections $\boldsymbol{W}_Z$: recognized partially
- Average firing rate after adding the top layer X: better completion
- Average firing rate using populations in the top layer: missing part is filled

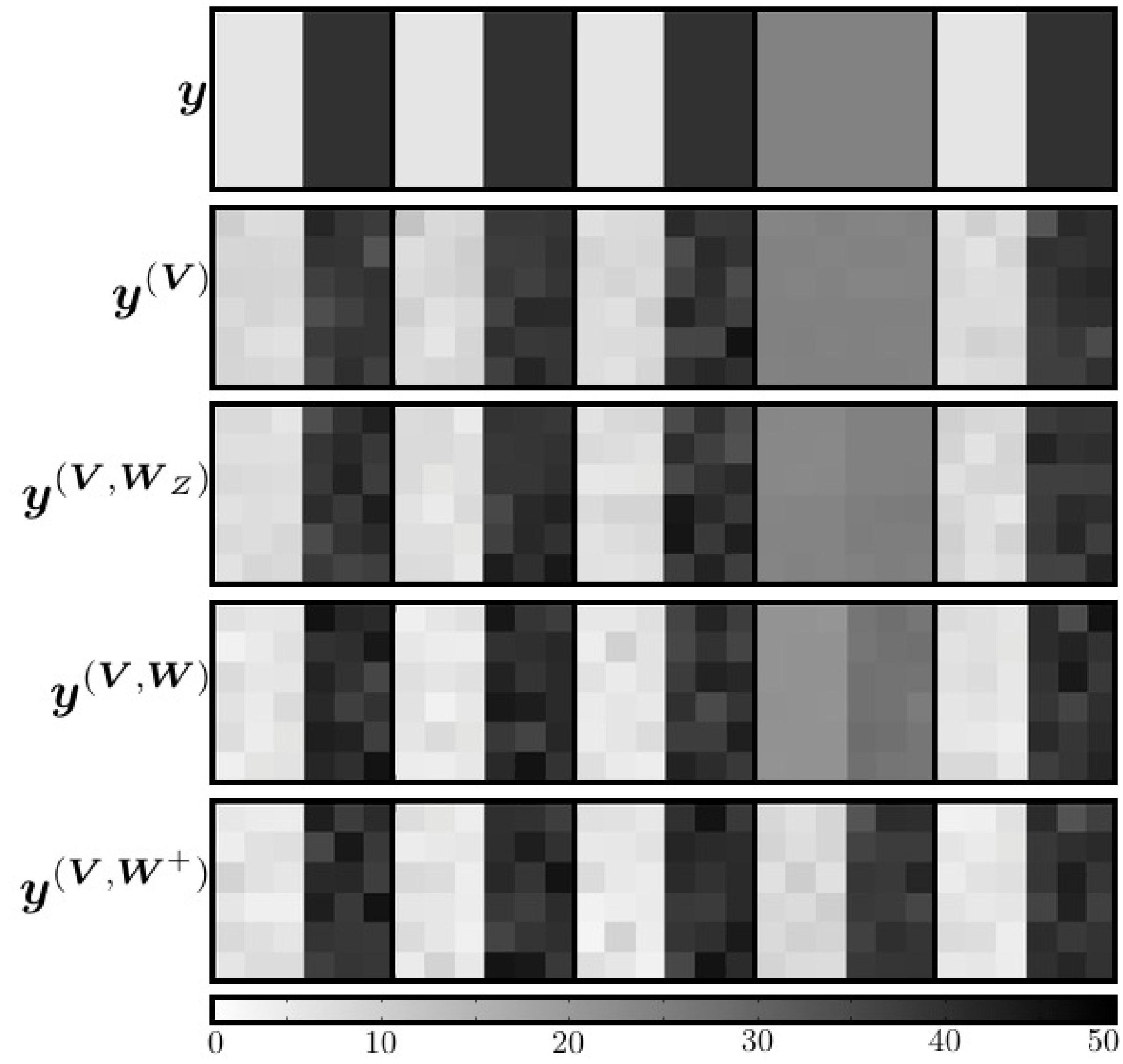

Figure C.6: Incomplete input image 2:
- Firing rate of the input layer neurons $\boldsymbol{y}$
- Average firing rate of the model using only feedforward weights $\boldsymbol{V}$
- Average firing rate when also using the lateral layer connections $\boldsymbol{W}_Z$
- Average firing rate after adding the top layer X
- Average firing rate using populations in the top layer

Using only the feedforward weights $\boldsymbol{V}$ it is not possible to recognize the missing part as there is no information exchange between different image parts. When using the lateral layer connections, the missing pattern is already recognized and completed in a light fashion. When adding the top layer, this opinion about the missing part is enforced, and the missing pattern gets a higher contrast. When also adding the top-layer populations to the model, the image is entirely completed.

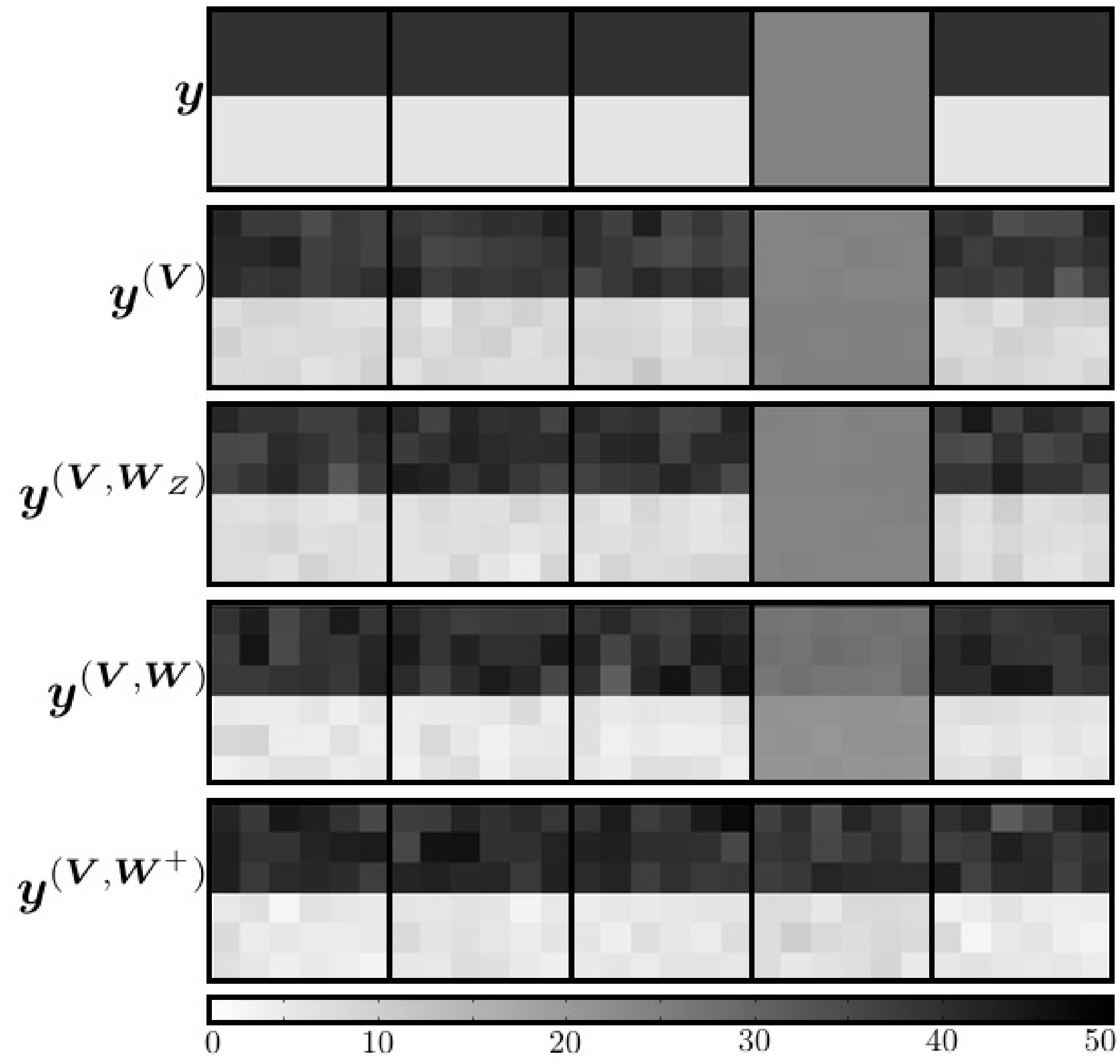

Figure C.7: Incomplete input image 3:
- Firing rate of the input layer neurons $\boldsymbol{y}$
- Average firing rate of the model using only feedforward weights $\boldsymbol{V}$
- Average firing rate when also using the lateral layer connections $\boldsymbol{W}_Z$
- Average firing rate after adding the top layer X
- Average firing rate using populations in the top layer

Using only the feedforward weights $\boldsymbol{V}$ it is not possible to recognize the missing part as there is no information exchange between different image parts. When using the lateral layer connections, the missing pattern is already recognized and completed in a light fashion. When adding the top layer, this opinion about the missing part is enforced, and the missing pattern gets a higher contrast. When also adding the top-layer populations to the model, the image is entirely completed.

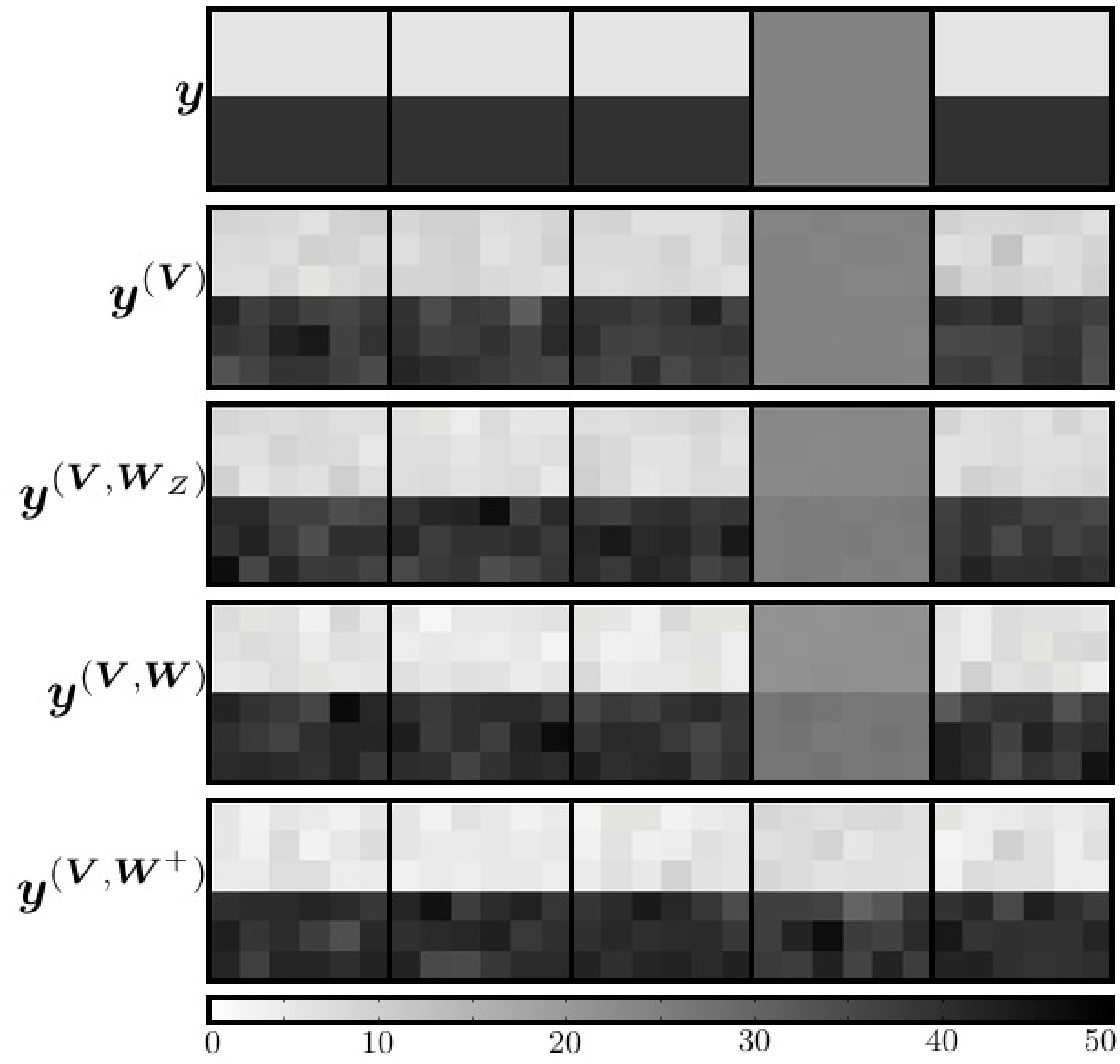

Figure C.8: Incomplete input image 4:
- Firing rate of the input layer neurons $\boldsymbol{y}$
- Average firing rate of the model using only feedforward weights $\boldsymbol{V}$
- Average firing rate when also using the lateral layer connections $\boldsymbol{W}_Z$
- Average firing rate after adding the top layer X
- Average firing rate using populations in the top layer

Using only the feedforward weights $\boldsymbol{V}$ it is not possible to recognize the missing part as there is no information exchange between different image parts. When using the lateral layer connections, the missing pattern is already recognized and completed in a light fashion. When adding the top layer, this opinion about the missing part is enforced, and the missing pattern gets a higher contrast. When also adding the top-layer populations to the model, the image is entirely completed.

C.3 Non-matching subpart of the image

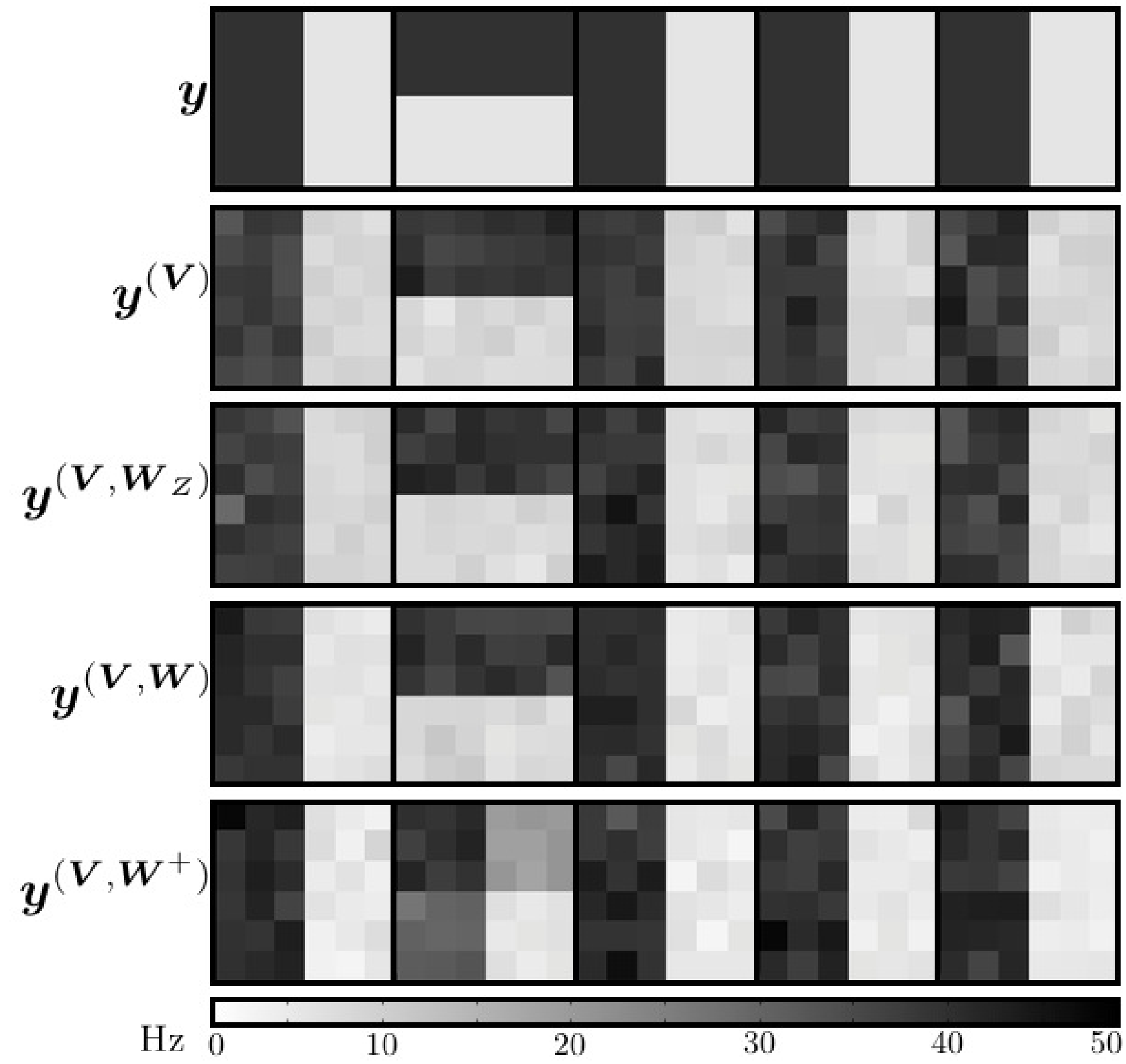

Figure C.9: Image with non-matching part: Firing rate of the input layer neurons $\boldsymbol{y}$
- Average firing rate of the model using only feedforward weights $\boldsymbol{V}$
- Average firing rate when also using the lateral layer connections $\boldsymbol{W}_Z$: weights not large enough
- Average firing rate after adding the top layer X: still only low influence of "neighbors"
- Average firing rate using populations in the top layer: non-matching part is detected

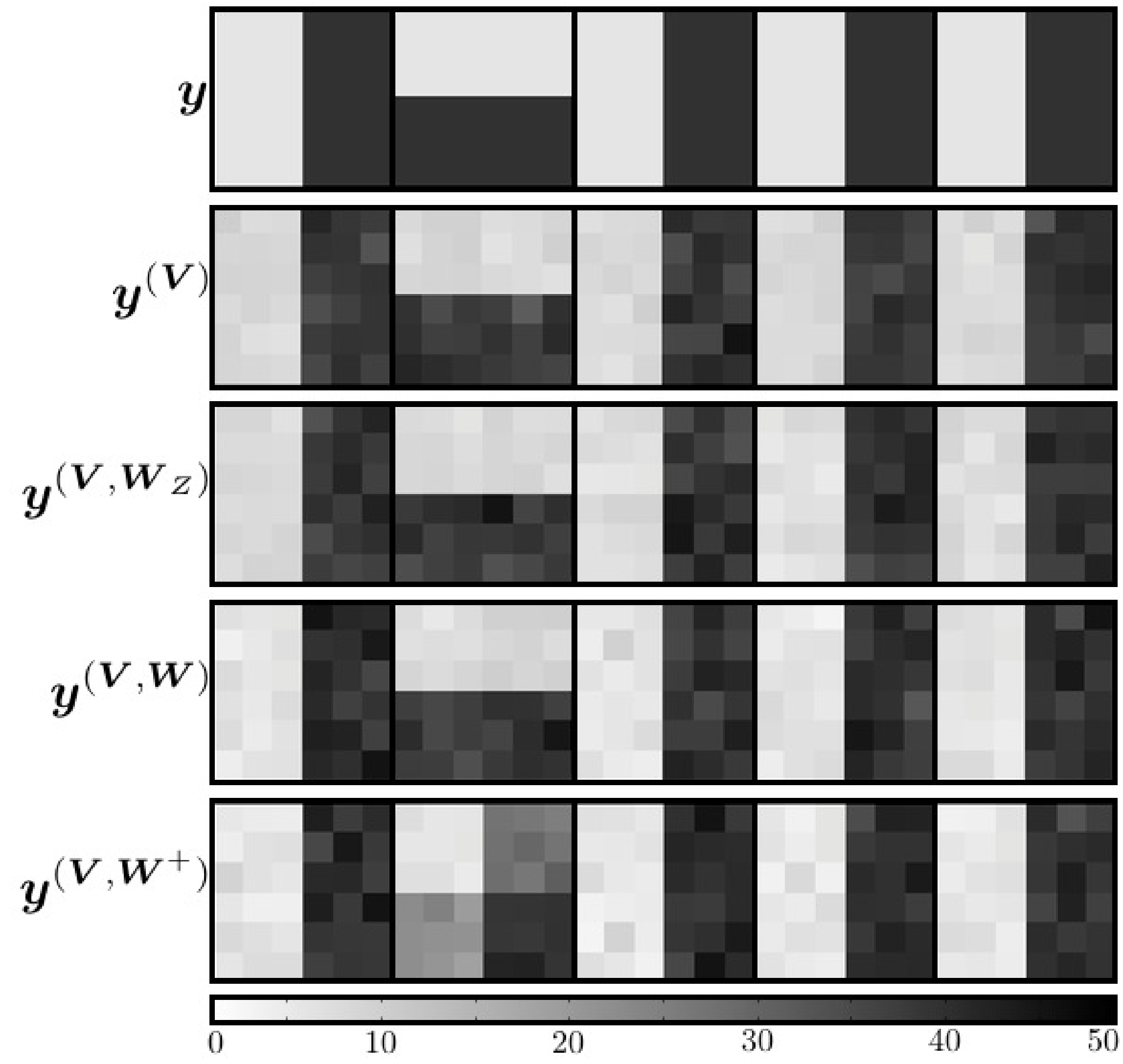

Figure C.10: Image with non-matching part 2:
- Firing rate of the input layer neurons $\boldsymbol{y}$
- Average firing rate of the model using only feedforward weights $\boldsymbol{V}$
- Average firing rate when also using the lateral layer connections $\boldsymbol{W}_Z$
- Average firing rate after adding the top layer X
- Average firing rate using populations in the top layer

When only having the information from direct neighbors, the influence of this idea is too low to affect the firing rates of the model. With the additional information of the top layer, the system's firing rates are slightly affected, but still the influence of this image as a whole is not large enough. With the expansion of the top layer to populations, the wrong or non-matching part of the image is detected, and the firing rates change towards recognizing the "correct" prior-known image.

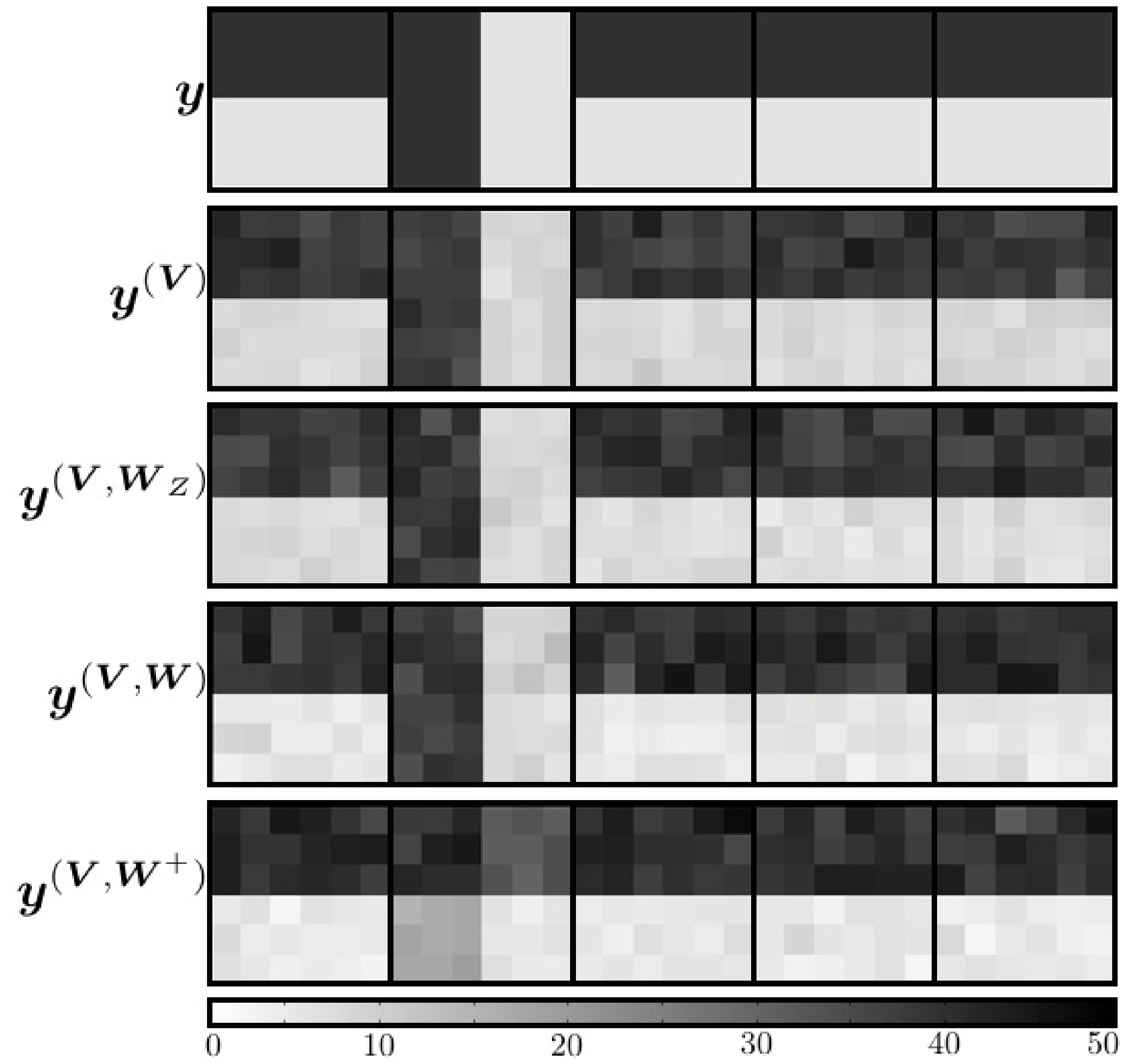

Figure C.11: Image with non-matching part 3:
- Firing rate of the input layer neurons $\boldsymbol{y}$
- Average firing rate of the model using only feedforward weights $\boldsymbol{V}$
- Average firing rate when also using the lateral layer connections $\boldsymbol{W}_Z$
- Average firing rate after adding the top layer X
- Average firing rate using populations in the top layer

When only having the information from direct neighbors, the influence of this idea is too low to affect the firing rates of the model. With the additional information of the top layer, the system's firing rates are slightly affected, but still the influence of this image as a whole is not large enough. With the expansion of the top layer to populations, the wrong or non-matching part of the image is detected, and the firing rates change towards recognizing the "correct" prior-known image.

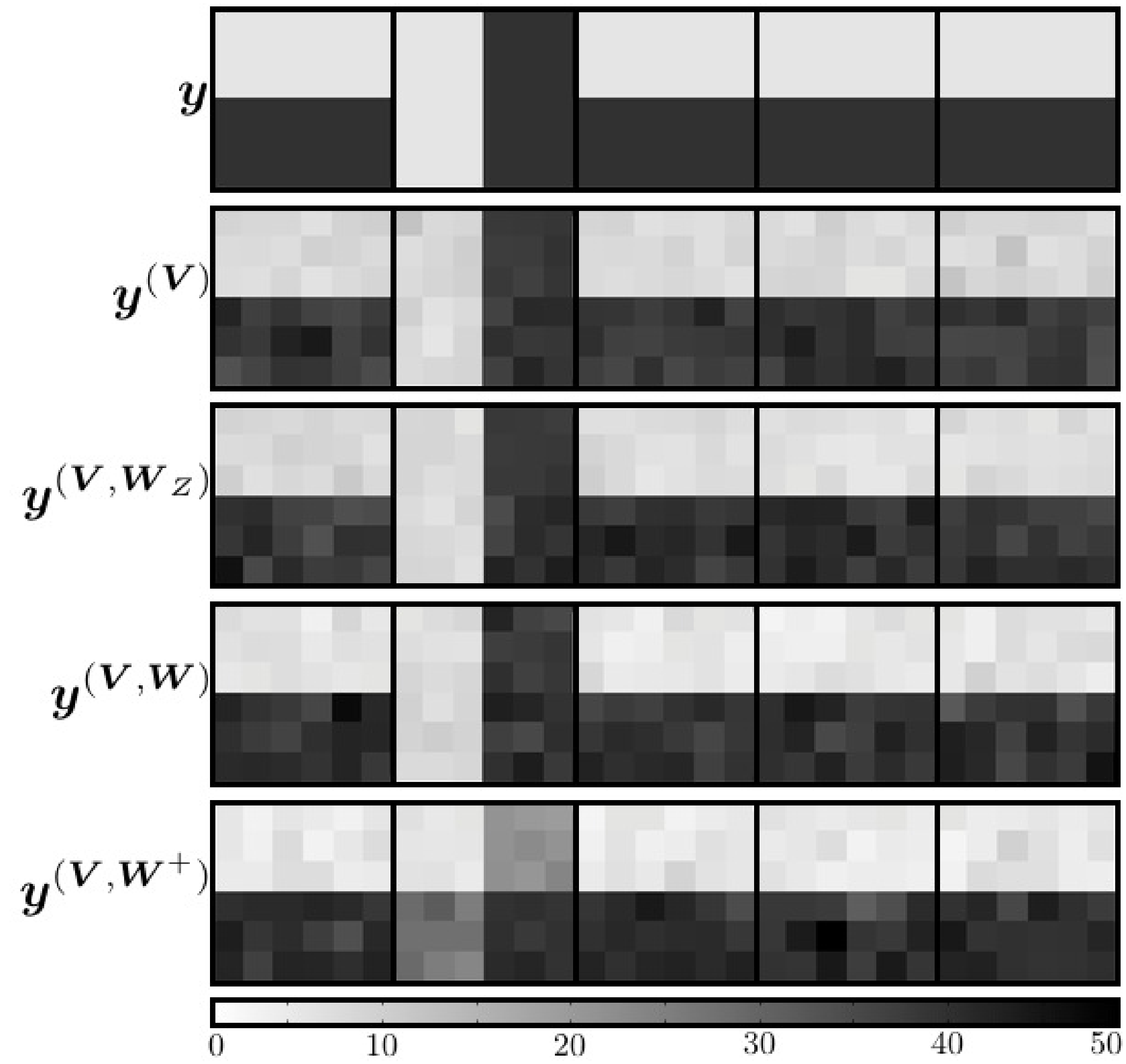

Figure C.12: Image with non-matching part 4:
- Firing rate of the input layer neurons $\boldsymbol{y}$
- Average firing rate of the model using only feedforward weights $\boldsymbol{V}$
- Average firing rate when also using the lateral layer connections $\boldsymbol{W}_Z$
- Average firing rate after adding the top layer X
- Average firing rate using populations in the top layer

When only having the information from direct neighbors, the influence of this idea is too low to affect the firing rates of the model. With the additional information of the top layer, the system's firing rates are slightly affected, but still the influence of this image as a whole is not large enough. With the expansion of the top layer to populations, the wrong or non-matching part of the image is detected, and the firing rates change towards recognizing the "correct" prior-known image.

D Appendix: Bibliography

[1] Francisco Aboitiz, Daniver Morales, and Juan Montiel. The evolutionary origin of the mammalian isocortex: Towards an integrated developmental and functional approach. *Behavioral and Brain Science*, 5:535–52, 2003.

[2] David H. Ackley, Geoffrey E. Hinton, and Terrence J. Sejnowski. A learning algorithm for boltzmann machines. *Cognitive Science*, 9(1):147–169, 1985.

[3] Christophe Andrieu, Nando de Freitas, Arnaud Doucet, and Michael I. Jordan. An introduction to mcmc for machine learning. *Machine Learning*, 50:5–34, 2003.

[4] Alexander Bain. *Mind and Body: The Theories of their Relation.* D. Appleton and Company, 1873.

[5] Guillaume Balavoine and André Adoutte. The segmented urbilateria: A testable scenario. *Integrative and Comparative Biology*, 43(1):137–47, 2003.

[6] Michel Baudry and Joell Davis, editors. *Long-Term Potentiation.* Massachusetts Institue of Technology, 1991.

[7] Mark F. Bear, Barry W. Connors, and Michael A. Paradiso. *Neuroscience: Exploring the Brain.* Lippincott Williams and Wilkins, third edition edition, 2006.

[8] Ehrhard Behrends. *Introduction to Markov Chains.* ISBN 3-528-06986-4. Vieweg, 2000.

[9] Pietro Berkes, Gergö Orbán, Máté Lengyel, and József Fiser. Spontaneous cortical activity reveals hallmarks of an optimal internal model of the environment. *Science*, 331(83), 2010.

[10] José M. Bernardo. A bayesian mathematical statistics primer. *ICOTS*, 2006.

[11] Christopher M. Bishop. *Neural Networks for Pattern Recognition.* Clarendon Press, 1995.

[12] Christopher M. Bishop. *Pattern Recognition and Machine Learning.* Springer Verlag, 2006.

[13] Pierre Brémaud. *Markov Chains.* Number ISBN 0-387-98509-3. Springer Verlag, 1999.

[14] Lars Büsing. A model for probabilistic inference in sheets of local wta circuits in the cortex. Technical report, Graz University of Technology, Institute for Theoretical Computer Science, 2010.

[15] Lars Büsing, Johannes Bill, Bernhard Nessler, and Wolfgang Maass. Neural dynamics as sampling: A model for stochastic computation in recurrent networks of spiking neurons. *PLoS Computational Biology*, 2011.

[16] Gail A. Carpenter and Stephen Grossberg. A massively parallel architecture for a self-organizing neural pattern recognition machine. *Computer Vision, Graphics and Image Processing*, 37:54–115, 1987.

[17] Siddhartha Chib and Edward Greenberg. Understanding the metropolis-hastings algorithm. *American Statistician*, 49(4):327–335, 1995.

[18] D.R. Cox and D.V. Hinkley. *Theoretical Statistics.* Chapman and Hall, 1974.

[19] Peter Dayan and Larry F. Abbott. *Theoretical Neuroscience: Computational and Mathematical Modeling of Neural Systems.* MIT Press, 2011.

[20] Arthur P. Dempser, Nan M. Laird, and Donald B. Rubin. Maximum likelihood from incomplete data via the em algorithm. *Journal of the Royal Statistical Society*, 39(1):1–38, 1977.

[21] Ivo D. Dinov. Expectation maximization and mixture modeling tutorial. *Statistics Online Computational Resource, California Digital Library*, 2008.

[22] Richard O. Duda, Peter E. Hart, and David G. Stork. *Pattern Classification.* John Wiley and Sons, Inc., second edition edition, 2001.

[23] Cheng en Guo, Song-Chun Zhu, and Ying N. Wu. Visual learning by integrating descriptive and generative methods. Department of statistics papers, Department of Statistics, UCLA, UC Los Angeles, 2000.

[24] Stanley Finger. *Origins of Neuroscience: A History of Explorations into Brain Function.* Oxford University Press, 2001.

[25] József Fiser, Pietro Berkes, Gergö Orbán, and Máté Lengyel. Statistically optimal perception and learning: From behavior to neural representations. *Trends in Cognitive Sciences*, 14(3):119–130, 2010.

[26] Kunihiko Fukushima. Cognitron: A self-organizing multilayered neural network. *Biological Cybernetics*, 20(3-4):121–136, 1975.

[27] Stuart Geman and Donald Geman. Stochastic relaxation, gibbs distribution, and the bayesian restoration of images. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 6:721–741, 1984.

[28] Zoubin Ghahramani and Michael I. Jordan. Mixture models for llearning from incomplete data. In Russel Greiner, Thomas Petsche, and Stephen J. Hanson, editors, *Computational Learning Theory and Natural Learning Systems: Making Learning Systems Practical*, pages 67–85. MIT Press, 1997.

[29] Stephen Grossberg. Contour enhancement, short-term memory, and constancies in reverberating neural networks. *Studies in Applied Mathematics*, 1973.

[30] Stefan Habenschuss. Novel methods for probabilistic inference and learning in spiking neural networks. Master's thesis, Graz University of Technology, Institute for Theoretical Computer Science, 2010.

[31] Donald Hebb. *The Organization of Behavior.* Wiley, 1949.

[32] John Hertz, Andreas Krogh, and Richard G. Palmer. *Introduction to the Theory of Neural Computation.* Westview Press, 1991.

[33] Geoffrey E. Hinton. Boltzmann machine. *Scholarpedia*, 2(5):1668, 2007.

[34] Geoffrey E. Hinton, Terrence J. Sejnowski, and David H. Ackley. Boltzmann machines: Constraint satisfaction networks that learn. Technical report, CMU-CS-84-119, 1984.

[35] John J. Hopfield. Neural networks and physical systems with emergent collective computational abilities. *Proceedings of the National Academy of Sciences of the United States of America (PNAS)*, 79:2554–2558, 1982.

[36] John J. Hopfield. Hopfield network. *Scholarpedia*, 2(5):1977, 2007.

[37] David Hubel. Eye, brain and vision.

[38] William James. *The Principles of Psychology.* H. Holt and Company, 1890.

[39] Janneke F. M. Jehee, Pieter R. Roelfsema, Gustavo Deco, Jaap M. J. Murre, and Victor A. F. Lamme. Interactions between hhigher and lower visual areas improve shape selectivity of higher level neurons - explaining crowding phenomena. *Brain Research*, 1157:167–176, 2007.

[40] Eric R. Kandel, Thomas M. Jessell, and Joshua R. Sanes. *Principles of Neural Science.* McGraw-Hill, fourth edition edition, 2000.

[41] Victor A. F. Lamme and Pieter R. Roelfsema. The distinct modes of vision offered by feedforward and recurrent processing. *Trends in Neuroscience*, 23(11):571–579, 2000.

[42] Victor A. F. Lamme, Hans Supèr, Rogier Landman, Pieter R. Roelfsema, and Henk Spekreijse. The role of primary visual cortex (v1) in visual awareness. *Vision Research*, 40:1507–1521, 2000.

[43] Tai Sing Lee. Computations in the early visual cortex. *Journal of Physiology - Paris*, 97:121–139, 2003.

[44] Tai Sing Lee and David Mumford. Hierarchical bayesian inference in the visual cortex. *Optical Society of America*, 20(7):1434–1448, 2003.

[45] Tai Sing Lee and My Nguyen. Dynamics of subjective contour formation in the early visual cortex. *Proceedings of the National Academy of Sciences of the United States of America (PNAS)*, 98(4):1907–1911, 2001.

[46] Wolfgang Maass. On the computational power of winner-take-all. *Neural Computation*, 2000.

[47] Warren McCulloch and Walter Pitts. A logical calculus of the ideas immanent in nervous activity. *Bulletin of Mathematical Biophysics*, 5:115–133, 1943.

[48] Geoffrey J. McLachlan and David Peel. *Finite Mixture Models.* Wiley, 2000.

[49] Nicholas Metropolis. The beginning of the monte carlo method. *Los Alamos Science*, pages 125–130, 1987.

[50] Frederic P. Miller, Agnes F. Vandome, and Johne McBrewster. *Gibbs Sampling.* Alphascript Publishing, 2010.

[51] A. David Milner. Separate pathways for perception and action. *Trends in Neuroscience*, 15(1):20–25, 1992.

[52] Abigail Morrison, Ad Aertsen, and Markus Diesmann. *Spike-Timing Dependent Plasticity in Balanced Random Networks.* MIT Press, 2007.

[53] Bernhard Nessler, Johannes Bill, Dejan Percevski, and Lars Büsing. Spike-based sampling with refractory states from general probability distributions. Technical report, Technical University of Graz IGI, 2010.

[54] Bernhard Nessler, Michael Pfeiffer, and Wolfgang Maass. Hebbian learning of bayes optimal decisions. *In Advances in Neural Information Processing Systems, MIT Press*, 21:1169–1176, 2009.

[55] Bernhard Nessler, Michael Pfeiffer, and Wolfgang Maass. Spike-based expectation maximization. *Cosyne*, 2010.

[56] Bernhard Nessler, Michael Pfeiffer, and Wolfgang Maass. Stdp enables spiking neurons to detect hidden causes of their inputs. *In Proceedings of NIPS 2009: Advances in Neural Information Processing Systems*, 22:1357–1365, 2010.

[57] Christopher M. Niell and Michael P. Stryker. Highly selective receptive fields in mouse visual cortex. *The Journal of Neuroscience*, 28(30):750–7536, 2008.

[58] Petteri Nurmi. Mixture models. Technical report, Helsinki Institute for Information Technology.

[59] Michael O'Shea. *The Brain: A very short Introduction.* Oxford University Press, first edition edition, 2006.

[60] Matthias Oster, Rodney Douglas, and Shih-Chii Liu. Computation with spikes in a winner-take-all network. *Neural Computation*, 21(9):2437–2465, 2009.

[61] Dejan Percevski, Lars Büsing, and Wolfgang Maass. Probabilistic inference in general graphical models through sampling in stochastic networks of spiking neurons. *PLoS Computational Biology*, 2011.

[62] Stephen Polyak. *The Vertebrate Visual System.* University of Chicago Press, 1968.

[63] Matt Ridley. *Nature via Nurture: Genes, Experience, and What Makes Us Human.* Harper Collins, 2003.

[64] Maximilian Riesenhuber and Tomaso Poggio. Hierarchical models of object recognition in cortex. *Nature Neuroscience*, 1999.

[65] Pieter R. Roelfsema, Victor A. F. Lamme, and Henk Spekreijse. The implementation of visual routines. *Vision Research*, 40:1385–1411, 2000.

[66] Frank Rosenblatt. The perceptron: A probabilistic model for information storage and organization in the brain. *Psychological Review*, 65:386–408, 1958.

[67] Stuart J. Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach.* Prentice Hall, second edition edition, 2003.

[68] Ruslan Salakhutdinov and Geoffrey E. Hinton. Deep boltzmann machine. In *Proceedings of the International Conference on Artificial Intelligence and Statistics*, volume 5, pages 448–455, 2009.

[69] Matthew Schmolesky. The primary visual cortex. In Helga Kolb, Eduardo Fernandez, and Ralph Nelson, editors, *Webvision: The Organization of the Retina and Visual system*. University of Utah Health Sciences Center, 1995-2005.

[70] Claude E. Shannon. A mathematical theory of communication. *Bell Sytem Technical Tournal*, 27:379–423,623–656, 1948.

[71] Lei Shi and Thomas L. Griffiths. Neural implementation of hierarchical bayesian inference by importance sampling. *Advances in Neural Information Processing Systems*, 22:1669–1677, 2009.

[72] Allan Siegel and Hreday N. Sapru. *Essential Neuroscience*. Lippincott Williams and Wilkins, second edition edition, 2010.

[73] David Spiegelhalter and Kenneth Rice. Bayesian statistics. *Scholarpedia*, 4(8):5230, 2009.

[74] G. F. Striedter. *Principles of Brain Evolution*. Sinauer Associates, 2005.

[75] Zhuowen Tu and Song-Chun Zhu. Image segmentation by data-driven markov chain monte carlo. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 24(5):657–673, 2002.

[76] Alan Turing. Computing machinery and intelligence. *Mind*, 59(236):433–460, 1950.

[77] Stephen R. Williams and Greg J. Stuart. Dependence of epsp efficacy on synapse location in neocortical pyramidal neurons. *Science (New York, N.Y.)*, 295(5561):1907–1910, 2002.

[78] Alan L. Yuille and Davi Geiger. *The Handbook of Brain Theory and Neural Networks*, chapter Winner-Take-All mechanisms. MIT Press, 1995.

[79] Karl Zipser, Victor A. F. Lamme, and Peter H. Schiller. Contextual modulaiton in primary visual cortex. *The Journal of Neuroscience*, 16(22):7376–7389, 1996.

D 168

E Appendix: List of Figures

F Appendix: List of Acronyms

BM	Boltzmann machine
BU	Bottom-up (information passing)
EM	Expectation maximization
ePSP	Excitatory postsynaptic potential
FB	Feedback (information passing)
FF	Feedforward (information passing)
GS	Gibbs sampling
GWTA	Generalized winner take all
HMM	Hidden Markov model
Hz	Hertz
IID	Independent and identically distributed (random variables)
iPSP	Inhibitory postsynaptic potential
IT	Inferotemporal cortex
KL	Kullback-Leibler divergence
LGN	Lateral Geniculate Nucleus: Brain area where optic nerve fibres terminate
MC	Markov Chain
MCMC	Markov Chain Monte Carlo
MEX	Matlab external interface
MM	Mixture model
MT	Middle temporal: Visual area V5
NN	Neural network
PDF	Probability density function
PSP	Postsynaptic potential
Px	Pixel; picture element
RF	Receptive field
SEM	Spike-based expectation maximization
STDP	Spike-timing dependent plasticity
SWIG	Simple wrapper and interface generator
TD	Top-down (information passing)
VA	Visual area
V1	Visual area 1: Primary visual cortex
V2	Visual area 2: Secondary visual cortex
WTA	Winner take all

G Appendix: List of Symbols

Z	Lateral/Middle layer (in the multilayer neural network model)
K	Number of neurons k in a layer
z_k	Neuron with index k (in layer Z)
$\boldsymbol{z}_{\setminus k}$	Values of all neurons except neuron k
$\boldsymbol{z}$	Combination of z_k states of all neurons k: a sample
$\boldsymbol{z}_{posterior}$	Posterior sample, i.e., combination of posterior values z_k
$\boldsymbol{z}_{prior}$	Prior sample, i.e., combination of prior values z_k
$p(\boldsymbol{z})$	Probability that a sample $\boldsymbol{z}$ occurrs: sampled probability
$q(\boldsymbol{z})$	Probability that a sample $\boldsymbol{z}$ occurrs: analytical computation
Y	Input layer (in the multilayer neural network model)
y_i	Neuron i in input layer Y
$\boldsymbol{i}$	Vector of direct external input values, one for each neuron
X	Top layer (in the multilayer neural network model)
X^+	Expanded top layer, consisting of populations of neurons instead of single units
L	Number of neurons in one population of the top layer X
$\boldsymbol{V}$	Weight matrix (between input layer Y and lateral layer Z)
$\boldsymbol{V}^+$	Feedforward weights of the extended model
$\boldsymbol{W}$	(Boltzmann machine) weight matrix
$W_{l,k}$	Synaptic weight value of the connection from neuron k to neuron l
$\boldsymbol{W}_Z$	BM weights among lateral layer neurons
$\boldsymbol{W}_X$	BM weights between lateral layer and top layer
$\boldsymbol{W}^+$	BM weight matrix of the extended model
$\boldsymbol{b}$	Bias vector for Boltzmann machine
$\eta_{\boldsymbol{V}}$	Learning rate for weight matrix $\boldsymbol{V}$
$\eta_{\boldsymbol{W}}$	Learning rate for weight matrix $\boldsymbol{W}$
$\eta_{\boldsymbol{b}}$	Learning rate for vector $\boldsymbol{b}$
$\tilde{\boldsymbol{\eta}}$	Mask matrix: $\eta_{kl} = 1$ if there exists a synapse between neurons k and l, 0 otherwise
$u_k(t)$	Membrane potential of neuron k at time t
$f_k(u_k)$	Firing rate function value of neuron k, depending on the membrane potential value
τ_{ref}	Refractory period

ζ_k	Refractory value of neuron k
$g(\zeta_k)$	Refractory mechanisms value, depending on the current refractory value ζ_k
$T^{(k)}_{j,i}$	Transition probability for going from refractory value $\zeta_k = i$ to $\zeta_k = j$ in neuron k
$T^{(k)}_{\tau_{ref},i}$	Probability of a spike of neuron k when its current refractory value is $\zeta_k = i$
G_j	Group, formed by a set of input neurons
R_k	Receptive field of neuron k on Y. Either all units of group G_j belong to R_k or none
$\|R_k\|$	Receptive field size, i.e., number of input neurons in the receptive field of neuron k
$\nu_{0,k}$	Background hypothesis firing rate, of neuron k
$\nu_{i,k}$	Expected firing rate of neuron y_i, if neuron z_k is active
ν_j	Group firing rate of group j
s	A state of a Markov Chain
S_t	The state of a Markov Chain at time t
δ	Distance between states when using the Metropolis algorithm
$E(s_t)$	Energy of state s_t when using the Metropolis algorithm
θ	Parameters
$\max(X)$	Maximal value of the random variable X
$\min(X)$	Minimum value of the random variable X
μ_X	Mean value of random variable X
σ_X	Standard deviation of random variable X
$\rho_{X,Y}$	Linear (Pearson) correlation between random variables X and Y
$\mathrm{cov}(X,Y)$	Covariance between random variables X and Y
$\gcd(X,Y)$	Greatest common denominator of X and Y
$D(q\|\|p)$	Kullback-Leibler divergence from q to p
$\log(x)$	Logarithm of the value x
$\ln(x)$	Natural logarithm (to the base e) of the value x