RUPRECHT-KARLS-UNIVERSITÄT HEIDELBERG

Oliver Julien Breitwieser

Towards a Neuromorphic Implementation of Spike-Based Expectation Maximization

Master Thesis

KIRCHHOFF-INSTITUT FÜR PHYSIK

Department of Physics and Astronomy
University of Heidelberg

Master Thesis in Physics
submitted by

Oliver Julien Breitwieser

born in Essen (Germany)

2015

Towards a Neuromorphic Implementation of Spike-Based Expectation Maximization

This Master Thesis has been carried out by Oliver Julien Breitwieser at the

Kirchhoff Institute for Physics in Heidelberg

under the supervision of

Prof. Dr. Karlheinz Meier

Towards a Neuromorphic Implementation of Spike-Based Expectation Maximization

In the *spike-based expectation maximization* (SEM) model, a population of stochastic neurons learns to detect salient features in the spike patterns emitted by a forward projecting input layer in an unsupervised manner. Such unsupervised learning models are particularly suitable for neuromorphic emulation, which is usually characterized by low power consumption and high speed-up compared to simulation on conventional computing architectures. However, the original SEM model is rather abstract and therefore not amenable for a straightforward translation to existing neurmorphic devices. This thesis presents NSEM, a mechanistic implementation of SEM that is compatible with state-of-the-art neuromorphic hardware. In NSEM, we use LIF neurons with exponential synapses, which are a de-facto standard for neuromorphic devices, as well as double-exponential STDP. In particular, NSEM is targeted for implementation on the NM-PM1 platform currently under development in the Human Brain Project. Therefore, particular emphasis is put on modifying the synaptic plasticity rules to be compatible to the characteristics of the NM-PM1. We provide a detailed discussion of the network architecture and parametrization of NSEM and demonstrate the performance of our implementation in a series of hardware-constrained software simulations. Furthermore, potential obstacles of a successful emulation on hardware are discussed and methods for compensation presented.

Eine neuromorphe Implementierung von Spike-basierter Erwartungsmaximierung

Im sogenannten Modell von *Spike-basierter Erwartungsmaximierung* (SEM) lernt eine Population stochastischer Neuronen unbeaufsichtigt typische Merkmale in Spikemustern einer auf sie projizierenden Eingangsschicht zu erkennen. Modelle unüberwachten Lernens wie dieses sind primäre Kandidaten für Emulation in neuromorpher Hardware, welche sich typischerweise durch eine niedrige Energieaufnahme sowie einen hohen Beschleunigungsfaktor im Vergleich zur Simulation auf konventioneller Rechnerarchitektur auszeichnet. Jedoch ist das ursprüngliche SEM-Modell eher abstrakt und daher nicht geeignet, direkt auf neuromorphe Hardware portiert zu werden. Diese Arbeit stellt NSEM vor, eine mechanistische Implementierung von SEM, die kompatibel zu zeitgenössischen neuromorphen Systemen ist. NSEM nutzt LIF-Neuronen mit exponentiellen Synapsen, welche einen de facto Standard für neuromorphe Systeme darstellen, sowie doppelt-exponentielles STDP. NSEM ist insbesondere für die Implementierung auf der NM-PM1 Plattform gedacht, die im Rahmen des Human Brain Project entwickelt wird. Daher wird besonderer Wert auf jene Modifikationen gelegt, die notwendig sind, um die synaptischen Plastizitätsregeln an die Charakteristiken der NM-PM1 anzupassen. Netzwerkarchitektur und Parametrisierung werden im Detail erläutert, sowie die Leistungsfähigkeit der Implementation in einer Serie Softwaresimulationen, welche Hardwarebeschränkungen unterliegen, gezeigt. Des Weiteren werden mögliche Mechanismen, die eine erfolgreiche Emulation in Hardware behindern könnten, sowie geeignete Kompensationsmethoden, diskutiert.

Contents

1 Introduction

A relatively recent development in theoretical neuroscience is to understand human reasoning in a Bayesian context [Knill and Pouget, 2004; Griffiths and Tenenbaum, 2006; Griffiths et al., 2008; Oaksford and Chater, 2007; Doya et al., 2011]. Bayesian inference is quintessentially probabilistic. Consider an observer who needs to find an explanation to a (possibly noisy and ambiguous) piece of evidence, a task which the brain is practically constantly required to perform (e.g., deducing the nature of an object that is partially hidden from view). A Bayesian observer takes into account both the nature of this evidence (essentially, a probability distribution) and his prior expectation (also a distribution) to produce a probabilistic model of the underlying cause (again, a distribution over predictions). An increasing amount of experimental evidence, from the behavioral [Körding and Wolpert, 2004] to the electrophysiological [Berkes et al., 2011] level, appears to support this hypothesis.

However, it is still a mystery how the probabilistic computations required to perform such inference tasks are implemented at the level of individual neurons and synapses. The sampling hypothesis – one possible explanation – states that the brain is not analytically computing probability distributions, but much rather operates on samples that are computed on-the-fly. An intuitive example is provided by scenarios of perceptual ambiguity, such as the popular duck-rabbit illusion shown in Figure 1.1. Our perception switches between the "duck" and "rabbit" interpretations, in accordance with the bimodal distribution that would be predicted by Bayesian reasoning, but the perceived animal is always a singular – we never experience a "duck-rabbit superposition". This suggests that, at some level, the brain activity encodes samples from inferred probability distributions rather then explicitly representing the distributions themselves.

Recently, the neural sampling theory developed in [Buesing et al., 2011] has linked activity of spiking stochastic neuron models to samples drawn from probability distributions over binary *random variables* (RVs). However, the original neural sampling model is rather abstract, with built-in neuronal stochasticity, membrane potentials that are not affected by outgoing spikes, rectangular *post-synaptic potentials* (PSPs) etc. All of these properties are neither found in biology, nor are they featured in commonly used neural simulators. Even more importantly for our purposes, they are largely incompatible with existing neuromorphic architectures.

[Petrovici et al., 2013] extend the theory to more biologically plausible and hardware-compatible neural dynamics. In particular, the LIF sampling framework employs LIF neurons with exponential synapses. The required stochasticity is provided by embedding in a noisy environment, e.g., a larger spiking network. This framework has been successfully applied to tasks such as strict Bayesian inference [Probst et al., 2015] or *maximum likelihood* (ML) learning in the form of training deep *Boltzmann machines* (BMs) [Leng, 2014].

Building upon the neural sampling theory, [Nessler et al., 2013; Bill et al., 2015] show that mutually inhibiting stochastic neurons are able to perform an online version of *expectation maximization* (EM). By employing a certain form of synaptic plasticity, neurons are able to identify hidden causes in their perceived input streams. In other words, neurons learn to encode the presence of particular spatial patterns in their receptive fields and are subsequently able to perform classification tasks, such as discerning between handwritten digits. This learning scheme – aptly named *spike-based expectation maximization* (SEM) – is completely self-organized as well as unsupervised and can therefore serve as a stepping stone towards understanding how the brain learns the Bayesian inference models that it appears to use.

Learning experiments are notoriously expensive in terms of raw simulation time when run on conventional computing architectures. They are therefore prime candidates for implementation on inherently parallel and potentially accelerated computing substrates. Neuromorphic hardware is usually designed with this exact purpose in mind. The *Neuromorphic Physical Model System 1* (NM-PM1) [Schemmel et al., 2010], under development in the *Human Brain Project* (HBP) and based on previous architectures developed in the *BrainScaleS* (BSS) and *Fast Analog Computing with Emerging Transient States* (FACETS) projects, aims for both inherent parallelism and a very high speed-up factor by implementing physical models of neurons and synaptic plasticity in analog hardware. Here, networks are no longer simulated by numerical integration of large systems of differential equations (as is done on conventional computers), but much rather *emulated* directly in analog circuitry. The achieved speed-up of $10^3 - 10^5$ compared to realtime and $10^4 - 10^6$ compared to regular large scale simulations conducted on super-computers is particularly remarkable. Furthermore, its energy efficiency compared to traditional simulations [Müller, 2014] is an additional bonus.

However, while highly configurable, neuron models and especially synaptic plasticity mechanisms are fixed at a conceptual level on the NM-PM1 system: they obey an immutable set of differential equations, with the only freedom lying in the choice of parameters. Therefore, in this thesis, we build an extended model of SEM, which we denote as *neuromorphic spike-based expectation maximization* (NSEM), which is specifically designed for compatibility with the NM-PM1 system. It uses stochastic *leaky integrate-and-fire* (LIF) neurons with exponential synapses while moderating neuronal activity via a spike-based implementation of homeostasis. We provide an in-depth description of the SEM to NSEM translation and evaluate the resulting network's performance in extensive simulations. This investigation may also serve as an exemplary study on the most important aspects of adjusting abstract theoretical models for neuromorphic hardware. Furthermore, the developed software framework is presented, along with a detailed discussion of potential future developments.

Thesis Outline

This manuscript is organized the following way: The theoretical concepts on which this thesis is based are presented in Chapter 2. Here – in order to provide a general introduction to the topic – we explain the general concepts of sampling and learning in spiking neural networks and derive some of the core equations in greater detail.

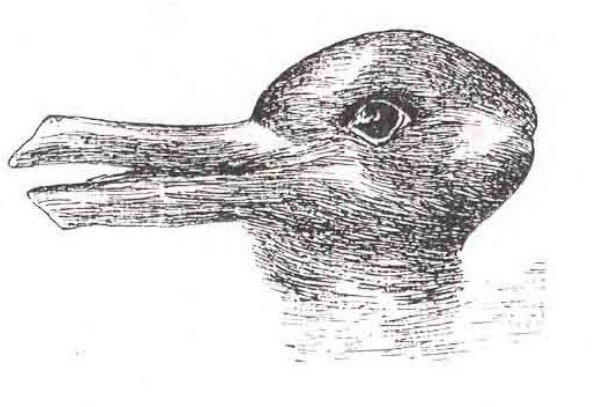

Figure 1.1: The commonly known duck rabbit illusion serves as an illustration of the sampling hypothesis: Human observers see either duck or rabbit and not a superposition of both. This suggests, that the brain is actively drawing samples from the abstract distribution of possible explanations of the perceived visual stimulus. Taken from: [McManus et al., 2010]

We then introduce the target neuromorphic hardware system in Chapter 3 and explain its key characteristics. Furthermore, we detail the unique characteristics of the *spike timing dependent plasticity* (STDP)-circuitry we have to take into consideration when aiming to emulate theoretical learning models on a neuromorphic substrate.

All work in this thesis was performed using traditional computing devices. We therefore take time to present the existing software upon which the new frameworks developed in this thesis were based in Chapter 4.

The main topic of this thesis, the implementation of SEM in a LIF sampling environment while ensuring fundamental compatibility for neuromorphic emulation, can be found in Chapter 5. Here we give a step-by-step illustration on how to adjust the original synaptic update rule as well as present a way of regulating the activity of stochastic neurons in a spike-based manner. This is a crucial component of successful learning in spike-based networks. Following this, we demonstrate the resulting network dynamics in a series of experiments. Also, we show potential challenges for successful network operations and how to circumvent them if possible.

Finally, in Chapter 6, we give a brief overview over what new software libraries and simulator models were developed during this thesis. Where applicable, we evaluate the resulting speed-up compared to previously available implementations.

2 Theoretical Background

After a short introduction to probability theory that just serves to establish nomenclature, this chapter introduces the theoretical concepts regarding sampling as well as learning that are used throughout this thesis. The theoretical foundations of Neural Sampling, *spike-based expectation maximization* (SEM) and *leaky integrate-and-fire* (LIF) sampling have been laid down in [Buesing et al., 2011; Nessler et al., 2013; Petrovici et al., 2013]. Here, we take the opportunity of presenting inference and learning in spiking neural networks as an integrated concept, by providing both an intuitive approach, as well as a formal and detailed derivation of its core equations, and by explaining connections to related formalisms from machine learning.

2.1 Introduction to Probability Theory

The following short summary of essential concepts in probability theory is largely based on [Bishop, 2006, chap. 1-2].

A *random variable* (RV) X describes the outcome of a stochastic process which is distributed with a certain probability distribution $p(x)$. In case the stochastic process can emit events x from a set of finitely many outcomes $\mathcal{X} = \{x_1, x_2, \ldots\}$, X is said to be *discrete*. For discrete RVs, the *probability mass function* $p : \mathcal{X} \to \mathbb{R}$ associates each outcome $x \in \mathcal{X}$ with the probability of it to occur. In order for p to be a proper probability distribution, two conditions have to hold:

$$\forall x \in \mathcal{X} : \; p(X = x) \geq 0 \tag{2.1}$$

$$\sum_{x \in \mathcal{X}} p(X = x) = 1 \tag{2.2}$$

If – on the other hand – $\mathcal{X} \subseteq \mathbb{R}$ is uncountable, the RV X is said to be *continuous*. The *probability density function* (PDF) $f : \mathcal{X} \to \mathbb{R}$ can then be used to determine the probability of the outcome falling into a certain interval:

$$p(a \leq X < b) = \int_a^b f(x)\,\mathrm{d}x \tag{2.3}$$

The following conditions have to hold for f:

$$\forall x \in \mathcal{X} : f(x) > 0 \tag{2.4}$$

$$\forall x \in \mathbb{R} \setminus \mathcal{X} : f(x) = 0 \tag{2.5}$$

$$\int_{\mathbb{R}} f(x)\,\mathrm{d}x = 1 \tag{2.6}$$

When dealing with several discrete RVs X, Y, Z we can define the *joint probability distribution* describing the probabilities of certain outcomes co-occuring:

$$\forall x \in \mathcal{X}, y \in \mathcal{Y}, z \in \mathcal{Z} : p(x, y, z) := p(X = x, Y = y, Z = z) \geq 0 \tag{2.7}$$

$$\sum_{x,y,z} p(x, y, z) = 1 \tag{2.8}$$

Please note that – while somewhat ambiguous – the shorthand $p(x) := p(X = x)$ was introduced to abbreviate the notation. The same goes for the notation $\sum_x \equiv \sum_{x \in \mathcal{X}}$.

The *marginal probability distribution* for a subset of RVs can be obtained by marginalizing (summing out) over all other variables:

$$p(x) = \sum_{y,z} p(x, y, z) \tag{2.9}$$

By fixing the outcomes of some RVs and renormalizing we obtain the *conditional probability distribution*, denoted by $p(x|y, z)$ (the outcome of X conditioned on the outcome of Y and Z).

$$p(x|y, z) = \frac{p(x, y, z)}{\sum_{x'} p(x', y, z)} = \frac{p(x, y, z)}{p(y, z)} \tag{2.10}$$

The joint, marginal and conditional probability distributions for sets of continuous (or mixed) RVs can be written analogously.

Equation (2.10) is often called Bayes' Theorem and written in the following way:

$$p(z|y) = \frac{p(y|z)p(z)}{p(y)} \tag{2.11}$$

In this context $p(z)$ is known as the prior-distribution, while $p(y|z)$ and $p(y)$ denote likelihood and evidence. Finally, $p(z|y)$ is the posterior distribution which can – as shown in Equation (2.11) – be obtained by combining prior knowledge and the likelihood. This process is known as *probabilistic inference*.

We can define the *expectation value* of X given p for discrete and continuous RVs:

$$\langle x \rangle_{p(x)} := \mathbb{E}_{p(x)}[x] = \sum_x x\, p(x) \tag{2.12}$$

$$\langle x \rangle_{p(x)} := \mathbb{E}_{p(x)}[x] = \int_{\mathcal{X}} x\, p(x)\ \mathrm{d}x \tag{2.13}$$

The *variance* is defined accordingly:

$$\mathrm{Var}_{p(x)}[x] = \left\langle \left(x - \langle x \rangle_{p(x)} \right)^2 \right\rangle_{p(x)} \tag{2.14}$$

$$= \left\langle x^2 \right\rangle_{p(x)} - \langle x \rangle^2_{p(x)} \tag{2.15}$$

2.2 Generative Models & Maximum Likelihood Learning

When trying to understand a stochastic process in the real world, it is often useful to model the *hypothetical* data generation process of observable data points $\mathcal{Y} = \{\mathbf{y}_1, \mathbf{y}_2, \ldots\}$ [Pearl, 1988]. The resulting *probabilistic model* – a probability distribution $p(\mathbf{y}|\boldsymbol{\theta})$ – quantifies how likely it is to generate a data sample $\mathbf{y}$ given the current parameter vector $\boldsymbol{\theta}$. For this reason these models are called *generative*: We could draw samples (see Section 2.4) from the probability distribution in order to generate "new" synthetic data – which fits the observed data depending on how well the generative model approximates reality [Bishop, 2006, chap. 8].

We assume that the data points observed were drawn from a probability distribution $p^*(\mathbf{y})$ which we do not know, but which we can approximate by the samples from our training data $\mathcal{Y}$. After choosing a suitable model – a task of scope beyond this thesis [Wit et al., 2012] – we wish to approximate $p^*(\mathbf{y})$ as closely as possible, that is to find the parameter vector $\hat{\boldsymbol{\theta}}$ minimizing the *Kullback-Leibler divergence* ($\mathrm{D_{KL}}$) between $p^*(\mathbf{y})$ and the likelihood $p(\mathbf{y}|\boldsymbol{\theta})$.

$$\hat{\boldsymbol{\theta}} = \arg\min_{\boldsymbol{\theta}} \mathrm{D_{KL}}(p^*(\mathbf{y}) || p(\mathbf{y}|\boldsymbol{\theta})) \tag{2.16}$$

In general, $\mathrm{D_{KL}}$ is a kind of difference measure between two probability distributions. In order to intuitively understand what the $\mathrm{D_{KL}}$ encodes, suppose we have a random process that can emit a set of events $\mathcal{X} = \{x_1, x_2, \ldots\}$ and we want to communicate these events to a receiver. We assume each $x \in \mathcal{X}$ is distributed with $q(x)$ when in reality it occurs with probability $p(x)$. If we now construct an ideal coding scheme for $\mathcal{X}$ based on $q(x)$, a message coding for x will ideally have length[1] $\propto \ln \frac{1}{q(x)} = -\ln q(x)$ – the more likely an event is, the shorter its encoding can be kept and vice versa. Our messages will be longer *on average* because of the following consideration: For each possible event x we can compare its probability in

[1] Please note that a message length in bits is only obtained when using the binary logarithm (base 2), because one bit is needed to encode an event occurring with a probability of $\frac{1}{2}$ (whether it happened or not). But – since all logarithms differ only by a constant factor – the $\mathrm{D_{KL}}$ can be used to quantify differences between probability distributions no matter which logarithm base is used.

both distributions. If $p(x) < q(x)$ the encoding should have been longer – we waste space by having a short encoding for not so frequent events. If $p(x) > q(x)$ then the event is more frequent, hence we could have made the encoding shorter – again our message is longer than it could have been.

The D_{KL} quantifies the *expected* overhead.

$$\mathrm{D}_{\mathrm{KL}}(p||q) = \langle(-\ln q(x)) - (-\ln p(x))\rangle_{p(x)} \tag{2.17}$$

$$= \sum_{x\in\mathcal{X}} \underbrace{p(x)}_{\substack{\text{actual}\\ \text{probability to}\\ \text{occur}}} [\underbrace{-\ln q(x)}_{\substack{\text{assumed ideal}\\ \text{encoding}\\ \text{length}}} - (\underbrace{-\ln p(x)}_{\substack{\text{actual ideal}\\ \text{encoding}\\ \text{length}}})] = \sum_{x\in\mathcal{X}} p(x) \ln\left(\frac{p(x)}{q(x)}\right) \tag{2.18}$$

This means that it is non-negative ($\mathrm{D}_{\mathrm{KL}}(p||q) \geq 0$) and 0 if and only if the two distributions are identical. However, the D_{KL} is not symmetric ($\mathrm{D}_{\mathrm{KL}}(p||q) \neq \mathrm{D}_{\mathrm{KL}}(q||p)$). For continuous distributions the sums in Equation (2.17) are replaced by integrals.

Inserting the definition of the D_{KL} into Equation (2.16) leads to:

$$\hat{\boldsymbol{\theta}} = \arg\min_{\boldsymbol{\theta}} \sum_{\mathbf{y}} p^*(\mathbf{y}) \ \ln\left(\frac{p^*(\mathbf{y})}{p(\mathbf{y}|\boldsymbol{\theta})}\right) \tag{2.19}$$

$$= \arg\min_{\boldsymbol{\theta}} \sum_{\mathbf{y}} \underbrace{p^*(\mathbf{y}) \ln(p^*(\mathbf{y}))}_{\text{independent of } \boldsymbol{\theta}} - p^*(\mathbf{y}) \ln(p(\mathbf{y}|\boldsymbol{\theta})) \tag{2.20}$$

$$= \arg\max_{\boldsymbol{\theta}} \sum_{\mathbf{y}} p^*(\mathbf{y}) \ \ln p(\mathbf{y}|\boldsymbol{\theta}) \tag{2.21}$$

$$= \arg\max_{\boldsymbol{\theta}} \ \langle \ln p(\mathbf{y}|\boldsymbol{\theta})\rangle_{p^*(\mathbf{y})} \tag{2.22}$$

$$\approx \arg\max_{\boldsymbol{\theta}} \sum_{\mathbf{y}\in\mathcal{Y}} \ln p(\mathbf{y}|\boldsymbol{\theta}) \tag{2.23}$$

In the last step we approximated the true probability distribution by its data samples. The term $\ln p(\mathbf{y}|\boldsymbol{\theta})$ is known as *log-likelihood*, whereas the obtained parameter vector $\hat{\boldsymbol{\theta}}$ is called the *maximum likelihood* (ML) estimate.

If instead of the likelihood-function we wish to employ Bayes' theorem (Equation (2.11)), that is to find the set of parameters most probable given our prior-knowledge of parameters $p(\boldsymbol{\theta})$

and the likelihood of such a parameter set generating our data $p(\mathbf{y}|\boldsymbol{\theta})$:

$$\boldsymbol{\theta}^*_{\text{MAP}} = \arg\max_{\boldsymbol{\theta}} \langle \ln p(\boldsymbol{\theta}|\mathbf{y}) \rangle_{p^*(\mathbf{y})} \tag{2.24}$$

$$\approx \arg\max_{\boldsymbol{\theta}} \sum_{\mathbf{y}\in\mathcal{Y}} \ln p(\boldsymbol{\theta}|\mathbf{y}) \tag{2.25}$$

$$= \arg\max_{\boldsymbol{\theta}} \left[\sum_{\mathbf{y}\in\mathcal{Y}} \ln \left(p(\mathbf{y}|\boldsymbol{\theta})p(\boldsymbol{\theta})\right) - \underbrace{\sum_{\mathbf{y}\in\mathcal{Y}} \ln p(\mathbf{y})}_{\text{independent of } \boldsymbol{\theta}} \right] \tag{2.26}$$

$$= \arg\max_{\boldsymbol{\theta}} \sum_{\mathbf{y}\in\mathcal{Y}} \ln \left(p(\mathbf{y}|\boldsymbol{\theta})p(\boldsymbol{\theta})\right) \tag{2.27}$$

$\boldsymbol{\theta}^*_{\text{MAP}}$ is called the *maximum a-posteriori probability* (MAP) estimate.

2.3 Boltzmann Machines

Boltzmann machines (BMs) [Ackley et al., 1985; Hinton, 2007] represent a certain form of probabilistic model over binary RVs $\mathbf{Z} = (Z_1, Z_2, Z_3, \ldots)$. It is best to visualize them as a network of interconnected units. Each unit (RV) Z_i has an intrinsic tendency to be either "active" ($Z_i = 1$) or "inactive" ($Z_i = 0$), represented by its bias b_i. Pair-wise interactions between RVs Z_i, Z_j are realized via symmetric weights W_{ij}. The higher the weight the more likely two units are to be active at the same time. If the weight W_{ij} is different from zero, the two units are said to be "connected", otherwise "unconnected". The full probability distribution for a state $\mathbf{z}$ can hence be given as

$$p(\mathbf{z}) = \frac{1}{Z} \exp\left[\sum_{i,j} \frac{1}{2} z_i \, W_{ij} \, z_j + \sum_k b_k \, z_k \right] \tag{2.28}$$

where Z is a normalization constant to ensure $p(\mathbf{z})$ is a proper distribution. Z is often also called *partition function*:

$$Z = \sum_{\mathbf{z}} \exp\left[\sum_{i,j} \frac{1}{2} z_i \, W_{ij} \, z_j + \sum_k b_k \, z_k \right] \tag{2.29}$$

By associating each state with a corresponding energy, the probability distribution can be reformulated as

$$E(\mathbf{z}) = -\sum_{i,j} \frac{1}{2} z_i \, W_{ij} \, z_j - \sum_k b_k \, z_k \tag{2.30}$$

$$p(\mathbf{z}) = \frac{1}{Z} \, e^{-E(\mathbf{z})} \tag{2.31}$$

Finding states with high probability is then analogous to minimizing the associated energy function.

Restricted Boltzmann machines

A subclass of regular BMs are *restricted Boltzmann machines* (RBMs): Here, the units are partitioned into a set of layers $\mathcal{L}_1, \ldots, \mathcal{L}_L$ each z_k belongs to exactly one layer. The connectivity is reduced in such a way that adjacent layers are fully connected with each other, while there are no connections within each layer. The energy function then becomes:

$$E(\mathbf{z}) = -\sum_{l=1}^{L-1} \sum_{\left\{i,j:\ \substack{z_i \in \mathcal{L}_l \\ z_j \in \mathcal{L}_{l+1}}\right\}} z_i W_{ij} z_j - \sum_k b_k\ z_k \tag{2.32}$$

Note that the factor of $\frac{1}{2}$ in the weight-term vanishes (compared to Equation (2.30)) because we no longer sum over all pairs of units twice.

Usually, the first layer is called the *visible layer* and the corresponding RVs are denoted with $\mathbf{v}$ whereas all other layers are called hidden with their RVs denoted by $\mathbf{h}$. For the simplest case, a two-layer RBM, the energy function can also be written as

$$E(\mathbf{v}, \mathbf{h}) = -\sum_{ij} v_i W_{ij} h_j - \sum_i a_i\ v_i - \sum_j b_j\ h_j \tag{2.33}$$

where the newly introduced a_i denote the biases of the visible units.

2.4 Sampling methods

Whenever one is dealing with probability models of practical interest, exact inference is intractable. For example, a BM with N RVs has 2^N states, all of which have to be summed over in order to calculate the partition function. It is therefore necessary to consider approximate inference methods such as sampling [Bishop, 2006, chap. 11]. Here a probability distribution is approximated by drawing samples from it without ever explicitly calculating its partition function.

2.4.1 Markov chain Monte Carlo sampling

In *Markov chain Monte Carlo* (MCMC) methods [Metropolis and Ulam, 1949], instead of starting from scratch with every new sample drawn, we keep track of where we are in the probability space in step τ by noting the current sample $\mathbf{z}^{(\tau)}$. Each new sample $\mathbf{z}^{(\tau+1)}$ is drawn from a transitional distribution $p_T(\mathbf{z}^{(\tau+1)}|\mathbf{z}^{(\tau)})$ – also referred to as the transition operator $T(\mathbf{z}^{(\tau+1)}, \mathbf{z}^{(\tau)})$. Each sample only depends just on the previous, not on the whole sequence of samples drawn beforehand. The probability space is therefore only searched *locally* instead of globally. This avoids calculating the partition function. Overall, the samples $\{\mathbf{z}^{(0)}, \mathbf{z}^{(1)}, \mathbf{z}^{(2)}, \ldots\}$ form a first-order Markov chain, which means the conditional distribution satisfies:

$$p(\mathbf{z}^{(\tau)}|\mathbf{z}^{(\tau-1)}, \ldots, \mathbf{z}^{(1)}) = p(\mathbf{z}^{(\tau)}|\mathbf{z}^{(\tau-1)}) \tag{2.34}$$

In other words: It does not matter where we already were in the probability space, it only matters where we are "now". This is also known as *Markov property*.

Metropolis-Hastings Algorithm

The way a new sample is actually produced differs from method to method. In some methods samples are directly drawn from the transition distribution (see Section 2.4.2), in others they are produced from a distribution $q(\mathbf{z}^*|\mathbf{z}^{(\tau)})$ easier to sample from and then accepted with an acceptance probability:

$$A(\mathbf{z}^*, \mathbf{z}^{(\tau)}) = \min\left(1, \frac{q(\mathbf{z}^{(\tau)}|\mathbf{z}^*)\ \tilde{p}(\mathbf{z}^*)}{q(\mathbf{z}^*|\mathbf{z}^{(\tau)})\ \tilde{p}(\mathbf{z}^{(\tau)})}\right) \tag{2.35}$$

where $\tilde{p}(\mathbf{z}) = Z \cdot p(\mathbf{z})$ is the unnormalized probability distribution to be sampled from. The acceptance criterion in Equation (2.35) is known as *Metropololis-Hastings algorithm* [Hastings, 1970].

Invariance

Another important concept is *invariance*. We do not want to alter the probability distribution p^* from which we sample by our choice of sampling process. In other words, we want that the following holds:

$$\sum_{\mathbf{z}'} p_T(\mathbf{z}|\mathbf{z}')\ p^*(\mathbf{z}') \stackrel{!}{=} p^*(\mathbf{z}) \tag{2.36}$$

The overall probability of sampling a certain state $\mathbf{z}$ may not change from one sampling step to another.

A sufficient, but not necessary condition to ensure invariance is *detailed balance*:

$$p_T(\mathbf{z}'|\mathbf{z})\ p^*(\mathbf{z}) \stackrel{!}{=} p_T(\mathbf{z}|\mathbf{z}')\ p^*(\mathbf{z}') \tag{2.37}$$

Since then

$$\sum_{\mathbf{z}'} p_T(\mathbf{z}|\mathbf{z}')\ p^*(\mathbf{z}') = \sum_{\mathbf{z}'} p_T(\mathbf{z}'|\mathbf{z})\ p^*(\mathbf{z}) \tag{2.38}$$

$$= p^*(\mathbf{z}) \sum_{\mathbf{z}'} p_T(\mathbf{z}'|\mathbf{z}) = p^*(\mathbf{z}) \tag{2.39}$$

For the Metropolis-Hastings algorithm, detailed balance is easily verifiable:

$$p_T(\mathbf{z}'|\mathbf{z})\ p^*(\mathbf{z}) = A(\mathbf{z}', \mathbf{z})\ q(\mathbf{z}'|\mathbf{z})\ p^*(\mathbf{z}) \tag{2.40}$$

$$= \min\left(q(\mathbf{z}'|\mathbf{z})\ p^*(\mathbf{z}), q(\mathbf{z}|\mathbf{z}')\ p^*(\mathbf{z}')\right) \tag{2.41}$$

$$= \min\left(\frac{q(\mathbf{z}'|\mathbf{z})\ p^*(\mathbf{z})}{q(\mathbf{z}|\mathbf{z}')\ p^*(\mathbf{z}')}, 1\right) q(\mathbf{z}|\mathbf{z}')\ p^*(\mathbf{z}') \tag{2.42}$$

$$= A(\mathbf{z}, \mathbf{z}')\ q(\mathbf{z}|\mathbf{z}')\ p^*(\mathbf{z}') \tag{2.43}$$

$$= p_T(\mathbf{z}|\mathbf{z}')\ p^*(\mathbf{z}') \tag{2.44}$$

where we used that $\frac{p^*(\mathbf{z}')}{p^*(\mathbf{z})} = \frac{\tilde{p}^*(\mathbf{z}')}{\tilde{p}^*(\mathbf{z})}$.

Ergodicity

Apart from leaving the desired distribution $p^*(\mathbf{z})$ invariant, the Markov chain also needs to be *ergodic*, which means that

$$\lim_{\tau\to\infty} p(\mathbf{z}^{(\tau)}) = p^*(\mathbf{z}) \tag{2.45}$$

irrespective of the initial choice of state $\mathbf{z}^{(0)}$.

Two necessary conditions for ergodicity are *aperiodicity* and *irreducibility*.

If there are transition probabilities such as

$$\exists\tilde{n},\tilde{r}\in\mathbb{N}:\forall n,r\in\mathbb{N}\lim_{\tau\to\infty} p(\mathbf{z}^{(n\tau+r)})\begin{cases}\neq 0 & \text{if } n=\tilde{n}, r=\tilde{r}\\ 0 & \text{otherwise}\end{cases} \tag{2.46}$$

the Markov chain is said to be *periodic* with period n.

Irreducibility means that it has to be possible to get from any state to any other state in a finite number of steps with non-vanishing probability as $\tau \longrightarrow \infty$. This means that no part of the state space can become inaccessible to us.

If the Markov chain is ergodic $p^*(\mathbf{z})$ is said to be the *equilibrium* or *stationary* distribution. It is straightforward to see that a Markov chain can have only one ergodic distribution. Also here, detailed balance is a sufficient, but not necessary condition for ergodicity. A Markov chain respecting detailed balance is said to be *reversible*. It is important to note that, among others, the sampling dynamics of neural networks discussed later (see Section 2.4.3) are not reversible while still sampling from correct stationary distributions.

2.4.2 Gibbs Sampling

Gibbs sampling [Geman and Geman, 1984] is a simple, but nevertheless widely applicable MCMC algorithm. Here the state vector $\mathbf{z} = (z_1, z_2, \ldots, z_K)^\top$ is successively updated with the conditional probability of one RV $z_k^{(\tau+1)}$ conditioned on all others:

$$p_T(\mathbf{z}^{(\tau+1)}|\mathbf{z}^{(\tau)}) = \prod_k p(z_k^{(\tau+1)}|z_1^{(\tau+1)},\ldots,z_{k-1}^{(\tau+1)},z_{k+1}^{(\tau)},\ldots,z_K^{(\tau)}) \tag{2.47}$$

$$=: \prod_k p(z_k^{(\tau+1)}|\underline{\mathbf{z}}_{k-1}^{(\tau+1)},\bar{\mathbf{z}}_{k+1}^{(\tau)}) \tag{2.48}$$

As should already be obvious from Equation (2.47), a full sample is only obtained after all RVs have been updated. Since the indexing of the RVs is essentially arbitrary, the order in which they are updated does not matter but is fixed within a sampling step.

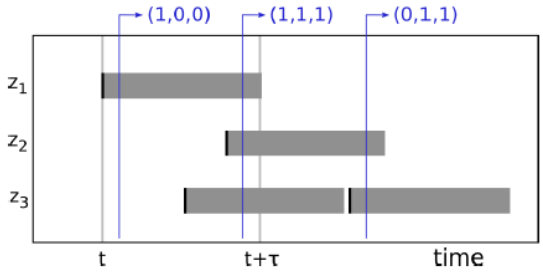

Figure 2.1: Interpretation of spiking dynamics as samples. Each neuron (in this example three) encodes a binary RV. Each spike represents a state switch of the corresponding RV from 0 to 1 for a time period τ_{on}. At any time t a sample of the underlying distribution can be drawn by checking which neurons spiked in the interval $(t - \tau_{\text{on}}, t]$. Illustration taken from: [Buesing et al., 2011]

Detailed balance is easily verifiable:

$$\begin{aligned} p_T(\mathbf{z}|\mathbf{z}')\, p^*(\mathbf{z}') &= \prod_{k=1}^{K} p^*(z_k|\underline{\mathbf{z}}_{k-1}, \bar{\mathbf{z}}'_{k+1})\, p^*(\mathbf{z}') \\ &= \left[\prod_{k=2}^{K} p^*(z_k|\underline{\mathbf{z}}_{k-1}, \bar{\mathbf{z}}'_{k+1})\right] \underbrace{p^*(z_1|\mathbf{z}')\, p^*(\mathbf{z}')}_{p^*(z_1,\mathbf{z}')} \\ &= \left[\prod_{k=\tilde{k}}^{K} p^*(z_k|\underline{\mathbf{z}}_{k-1}, \bar{\mathbf{z}}'_{k+1})\right] p^*(\underline{\mathbf{z}}_{\tilde{k}-1}, \mathbf{z}') \\ &= p^*(\mathbf{z}, \mathbf{z}') \end{aligned} \tag{2.49}$$

$$\implies \quad p_T(\mathbf{z}|\mathbf{z}')\, p^*(\mathbf{z}') \overset{2.49}{=} p^*(\mathbf{z}, \mathbf{z}') = p^*(\mathbf{z})\, p^*(\mathbf{z}') = p^*(\mathbf{z}')\, p^*(\mathbf{z}) = p^*(\mathbf{z}', \mathbf{z}) \tag{2.50}$$

$$\overset{2.49}{=} p_T(\mathbf{z}'|\mathbf{z})\, p^*(\mathbf{z}) \tag{2.51}$$

Here we used that the joint distribution of two samples factorizes into the product of marginals as well as

$$p^*(z_{k+1}|\underline{\mathbf{z}}_k, \bar{\mathbf{z}}'_{k+2})\, p^*(\underline{\mathbf{z}}_k, \mathbf{z}') = p^*(z_{k+1}|\underline{\mathbf{z}}_k, \bar{\mathbf{z}}'_{k+2})\, p^*(\underline{\mathbf{z}}_k, \bar{\mathbf{z}}'_{k+2})\, p^*(\underline{\mathbf{z}}'_{k+1}|\underline{\mathbf{z}}_k, \bar{\mathbf{z}}'_{k+2}) \tag{2.52}$$

$$= p^*(\underline{\mathbf{z}}_{k+1}, \bar{\mathbf{z}}'_{k+2})\, p^*(\underline{\mathbf{z}}'_{k+1}|\underline{\mathbf{z}}_k, \bar{\mathbf{z}}'_{k+2}) \tag{2.53}$$

$$= p^*(\underline{\mathbf{z}}_{k+1}, \mathbf{z}') \quad . \tag{2.54}$$

As can be seen from Equation (2.47), for each sample drawn, only $M \cdot K$ evaluations of conditional probability distributions are needed (assuming each of the K RVs has M states) – whereas evaluating the partition function would need up to M^K.

2.4.3 Neural Sampling

The network dynamics of interconnected spiking neurons can be linked to MCMC sampling, as shown in [Buesing et al., 2011]. Each neuron encodes for a binary RV Z_k that is said to be 1 whenever the neuron is in a refractory period after emitting a spike and 0 otherwise (see Figure 2.1).

The sufficient condition for neurons to sample from the stationary probability distribution $p^*(\mathbf{z})$ is the *neural computability condition* (NCC):

$$u_k(t) = \ln \frac{p^*(z_k = 1|\mathbf{z}_{\setminus k}(t))}{p^*(z_k = 0|\mathbf{z}_{\setminus k}(t))} \tag{2.55}$$

Each neuron needs to fulfill the condition that its membrane potential encodes the log-odds of the corresponding binary RV Z_k being active or inactive – given the state of *all* other RVs $\mathbf{z}_{\setminus k}$ – at any time step. Then the spiking activity of the network corresponds to samples from the underlying probability distribution $p^*(\mathbf{z})$.

Please note that the NCC implicitly forbids distributions with vanishing probabilities (that is, states $\tilde{\mathbf{z}}$ so that $p(\tilde{\mathbf{z}}) = 0$). Reordering Equation (2.55) while using that $p(z_k = 0|\mathbf{z}_{\setminus k}) = 1 - p(z_k = 1|\mathbf{z}_{\setminus k})$ shows that the overall probability for the neuron k to be active at any given point in time is:

$$p(z_k = 1|\mathbf{z}_{\setminus k}) = \sigma(u_k(\mathbf{z}_{\setminus k})) := \frac{1}{1 + \exp\left(-u_k(\mathbf{z}_{\setminus k})\right)} \tag{2.56}$$

Hence each neuron has a *logistic activation function.*

In order to preserve the Markov property Equation (2.34), [Buesing et al., 2011] introduce a set of internal variables $\boldsymbol{\zeta}$. Whenever a neuron fires, its corresponding ζ_k is set to the refractory period[2] $\tau_{\text{on}} \in \mathbb{N}$. Once $\zeta_k > 1$, it decays linearly. Overall, the transition probability is defined as follows:

$$p_T(\boldsymbol{\zeta}, \mathbf{z}|\boldsymbol{\zeta}', \mathbf{z}') = T(\boldsymbol{\zeta}, \mathbf{z}|\boldsymbol{\zeta}', \mathbf{z}') = \prod_k T^k(\zeta_k, z_k|\zeta_k', \mathbf{z}'_{\setminus k}) = \prod_k p(z_k|\zeta_k)\; T^k(\zeta_k|\zeta_k', \mathbf{z}'_{\setminus k}) \tag{2.57}$$

$$p(z_k|\zeta_k) = \begin{cases} 1 & \text{if } (z_k = 1 \wedge \zeta_k \geq 1) \vee (\; z_k = 0 \wedge \zeta_k = 0) \\ 0 & \text{otherwise} \end{cases} \tag{2.58}$$

$$T^k(\zeta_k, z_k|\zeta_k', \mathbf{z}'_{\setminus k}) = \begin{cases} 1 & \text{if } \zeta_k = \zeta_k' - 1 \wedge \zeta_k' \in [1, \tau_{\text{on}}] \\ \sigma(u_k(\mathbf{z}_{\setminus k}) - \ln \tau_{\text{on}}) & \text{if } \zeta_k = \tau_{\text{on}} \wedge \zeta_k' \in \{0, 1\} \\ 1 - \sigma(u_k(\mathbf{z}_{\setminus k}) - \ln \tau_{\text{on}}) & \text{if } \zeta_k = 0 \wedge \zeta_k' \in \{0, 1\} \\ 0 & \text{otherwise} \end{cases} \tag{2.59}$$

A schematic of the transition operator T^k can be seen in Figure 2.2. The introduction of the internal variables $\boldsymbol{\zeta}$ is necessary because otherwise the state $\mathbf{z}^{(\tau)}$ would depend on the last τ_{on} states $\{\mathbf{z}^{(\tau-1)}, \ldots, \mathbf{z}^{(\tau-\tau_{\text{on}})}\}$ in order to know when the absolute refractory period was over. This would lead to a Markov chain of order τ_{on}.

The term $-\ln \tau_{\text{on}}$ in the transition probability is due to the discretization of the model: The finer the time steps, the larger τ_{on} gets (relatively) and the transition probability *per time step* gets lower. We can show this by noting that the activation function Equation (2.56) denotes

[2] We are dealing with a discrete model, therefore the refractory time period is integer-valued. It can easily be made biologically plausible by specifying what biological time interval one step in the discrete model corresponds to.

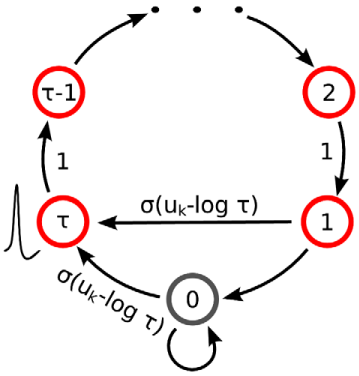

Figure 2.2: Illustration of the theoretical neuron model with absolute refractory mechanism. Shown is a schematic of the transition operator T^k for the state variable ζ_k. Whenever the neuron can spike ($\zeta_k = \{0, 1\}$) the probability to do so during one time step is $\sigma(u_k - \ln \tau_{\text{on}})$. Once spiked, ζ_k is set to τ_{on} and decays in deterministic fashion by one each time step. The state of the corresponding binary RV is then $Z_k = 1 \Leftrightarrow \zeta_k > 0$. Illustration taken from [Buesing et al., 2011].

the probability to find the neuron k in the active state at a random point in time. This is different from the spiking probability p_{spike} of the neuron in each time step. We can motivate this by deriving the activation from the transition probability: We simulate a stochastic neuron with fixed u_k for n time steps. In each time step, it has a probability to spike of p_{spike}, hence it fires an average of $n \cdot p_{\text{spike}}$, each time remaining in the active state for τ_{on} steps (T_{ON} steps in total). Every time the neuron spikes, we increase the simulation time by τ_{on} steps. Conversely, $T_{\text{OFF}} = n \cdot (1 - p_{\text{spike}})$ steps the neuron does not spike. We hence have for the activation function:

$$\frac{1}{1 + e^{-u_k}} \overset{!}{=} p(z_k = 1 | \mathbf{z}_{\setminus k}) = \frac{T_{\text{ON}}}{T_{\text{ON}} + T_{\text{OFF}}} \tag{2.60}$$

$$= \frac{n \cdot p_{\text{spike}} \tau_{\text{on}}}{n \cdot (1 - p_{\text{spike}}) + n \cdot p_{\text{spike}} \tau_{\text{on}}} \tag{2.61}$$

$$= \frac{1}{\frac{1}{\tau_{\text{on}}} \cdot (\frac{1}{p_{\text{spike}}} - 1) + 1} \tag{2.62}$$

$$\Longrightarrow p_{\text{spike}}(u_k) = \frac{1}{1 + \tau_{\text{on}}\, e^{-u_k}} = \sigma(u_k - \ln \tau_{\text{on}}) \tag{2.63}$$

The formal proof in [Buesing et al., 2011] then shows that $p(\boldsymbol{\zeta}, \mathbf{z})$ is the unique invariant distribution of operator T.

$$p(\boldsymbol{\zeta}, \mathbf{z}) = p(\boldsymbol{\zeta}|\mathbf{z}) p(\mathbf{z}) = p(\mathbf{z}) \prod_k p(\zeta_k | z_k) \tag{2.64}$$

$$p(\zeta_k | z_k) = \begin{cases} \frac{1}{\tau_{\text{on}}} & \text{if } z_k = 1 \wedge \zeta_k > 0 \\ 1 & \text{if } z_k = 0 \wedge \zeta_k = 0 \\ 0 & \text{otherwise} \end{cases} \tag{2.65}$$

The Markov chain therefore samples from the joint probability distribution $p(\boldsymbol{\zeta}, \mathbf{z})$ (Equation (2.64)), but since $\sum_{\boldsymbol{\zeta}} p(\boldsymbol{\zeta}, \mathbf{z}) = p^*(\mathbf{z})$ by construction, samples from $p^*(\mathbf{z})$ are obtained by omission of $\boldsymbol{\zeta}^{(\tau)}$. The network can therefore carry out probabilistic inference over the underlying distribution $p^*(\mathbf{z})$. For example, the state of some $z_1, \ldots, z_l$ could be fixed to either 0 or 1 by forcing the membrane potential to either a very high or very low state. Then the dynamics of the remaining neurons correspond to samples from the probability distribution

$$p(z_{l+1}, \ldots, z_K | z_1, \ldots, z_l) \tag{2.66}$$

Furthermore, it is important to note that the sampling dynamics are not reversible as was the case for MCMC chains demonstrating detailed balance (Section 2.4.1): While the transition $z_k = 0 \rightarrow z_k = 1$ is possible at any time and can be made arbitrarily likely by increasing the membrane potential u_k, once the spike has occurred, a neuron will stay in the refractory period for τ_{on}. There is no way to induce the state switch $z_k = 1 \rightarrow z_k = 0$ any sooner. Nevertheless, the MCMC chain samples from the correct distribution.

In [Buesing et al., 2011], the theoretical framework is extended even further in order to include continuous time and relative refractory periods, but these go beyond the needed scope[3] of this thesis and are not discussed further.

Neuron Dynamics

Until this point, we discussed neither the actual neuron dynamics that fulfill the NCC nor how the stationary probability distributions $p^*(\mathbf{z})$ of such dynamics look like. A straightforward example is to take the BM distribution (2.28) and insert it into Equation (2.55):

$$u_k(\mathbf{z}_{\setminus k}(t)) = \sum_{j \neq k} W_{jk} z_j(t) + b_k \tag{2.67}$$

where we made use of the fact that the weights are symmetric. We see that Equation (2.67) corresponds to a basic neuron model, where Boltzmann weights W_{ij} can identified as the strength of synaptic interaction and the bias b_k as inherent excitability of a neuron. The synaptic interactions are strictly rectangular (neuron j contributes a fixed amount W_{jk} to the membrane potential of neuron k the moment it is active; there is no rise or decay period) and the model is inherently stochastic. The probability to spike is $\sigma(u_k(\mathbf{z}_{\setminus k}) - \ln \tau_{\text{on}})$ – see Equation (2.59) – resulting in an instantaneous firing rate $r_k(t)$:

$$\begin{aligned} r_k(t) &= \lim_{\Delta t \to 0} \frac{p(\text{spike in } [t, t + \Delta t])}{\Delta t} \\ &= \begin{cases} \frac{1}{\tau_{\text{on}}} \exp\left(u_k(t)\right) & \text{if } z_k = 0 \\ 0 & \text{if } z_k = 1 \end{cases} \end{aligned} \tag{2.68}$$

BMs are a suitable model for many real-world problems, e.g., *binocular rivalry* [Alais and Blake, 2005]. Nevertheless, they are limited to second-order interactions only – corresponding to direct connections between point neurons exchanging spikes. By including more complex interactions, either via (inter-) neurons, dendrites or other precomputing elements, the class of representable joint distributions can be extended [Nessler et al., 2008]. This allows for probabilistic inference in arbitrary Bayesian networks over binary RVs [Pecevski et al., 2011].

[3] As will be discussed in Sections 2.4.4 and 3.1, the employed neuron model on the *Neuromorphic Physical Model System 1* (NM-PM1) features absolute refractory mechanisms only.

2.4.4 Neural Sampling with stochastic LIF-Neurons

The theory of Neural Sampling can be extended into the domain of deterministic neuron models, namely the *leaky integrate-and-fire* (LIF) neuron model [Petrovici et al., 2013]. When a neuron is brought into a *high-conductance state* (HCS), the free membrane potential can be described by an *Ornstein-Uhlenbeck* (OU) process, effectively adding stochasticity to the otherwise deterministic neuron model. It can then be shown by rigorous theoretical treatment that its activation function – the probability of finding the neuron in a refractory state given a certain mean membrane potential – follows a logistic function, finally linking it to the NCC (Equation (2.56)). We will go into more detail in the following.

Using stochastic LIF neurons for sampling has been successfully applied to further tasks, such as Bayesian inference [Probst, 2014; Probst et al., 2015] as well as training deep BMs [Leng, 2014].

Deterministic Neuron Models

Other than the neuron dynamics discussed in Section 2.4.3, where neuron k had an inherent probability to spike-based on its momentary membrane potential (Equation (2.68)), the *adaptive exponential integrate-and-fire* (AdEx) model [Brette and Gerstner, 2005] implemented on the NM-PM1 (see Chapter 3) is completely deterministic. The dynamics of its membrane potential u_k are described by the following differential equations:

$$C_m \frac{\mathrm{d}u_k}{\mathrm{d}t} = -g_L(u_k - E_L) + g_L \Delta_T \exp\left(\frac{u_k - V_T}{\Delta_T}\right) - w(t) + I(t) \tag{2.69}$$

$$\tau_w \frac{\mathrm{d}w}{\mathrm{d}t} = a(u_k(t) - E_L) - w \tag{2.70}$$

where I is the input current, C_m is the membrane capacitance, E_L is the leak reversal potential, g_L the leakage conductance, w the adaption current, V_T the threshold, Δ_T the slope factor, a the adaption coupling parameter and τ_w the adaption time constant. In theory, a spike is said to occur when the membrane potential diverges towards infinity. In practice, however, a spike is detected when the membrane potential reaches a certain threshold $\vartheta := V_{\text{spike}} > V_T$. The membrane potential is thus set to the reset value V_{reset} and kept there for the refractory period τ_{refrac}. Furthermore, the adaption current is augmented by an amount b: $w \to w + b$. If $a = 0,\ b = 0$ (no adaption) and in the limit $\Delta_T \to 0$ (no exponential sub-threshold dynamics), the AdEx model reduces to the *leaky integrate-and-fire* (LIF) model.

The total input current $I = I^{\text{syn}} + I^{\text{ext}}$ is comprised of two parts. On the one hand, there is external stimulus I^{ext}, e.g., a current injection, on the other we have synaptic input I^{syn} from other neurons. Depending on whether the synaptic interactions are modelled current or conductance based, we have different dynamics.

In case of a current-based approach, we have

$$I^{\text{syn}}(t) = \sum_{\text{syn } i} w_{ik} \sum_{t_i^s} \exp\left(-\frac{t - t_i^s}{\tau_{\text{syn}}}\right) \Theta\left(t - t_i^s\right) \tag{2.71}$$

where index i iterates over all afferent synapses, $\{t_i^s\}$ are the spike times arriving via synapse i, w_{ik} is the weight of the synapse (connecting neuron i to k), τ_{syn} its time constant and $\Theta(x)$ is the Heaviside step function[4].

Whereas for conductance-based synapses we have

$$I^{\text{syn}}(t) = \sum_{\text{syn } i} g_i^{\text{syn}}(t) \ (u_k(t) - E_i^{\text{rev}}) \tag{2.72}$$

$$g_i^{\text{syn}}(t) = w_{ik} \sum_{t_i^s} \exp\left(-\frac{t - t_i^s}{\tau_{\text{syn}}}\right) \Theta\left(t - t_i^s\right) \tag{2.73}$$

where E_i^{rev} is the reversal potential and g_i^{syn} the conductance of the i-th synapse. The main difference between the two approaches – besides the latter being more biologically plausible – is the total contribution of input current by a single spike is fixed in the current-based approach but highly variable in the conductance-based case, depending on the distance between membrane potential and the reversal potentials. On the NM-PM1 (see Chapter 3) all synaptic connections are conductance-based.

Noisy Environment

When a conductance based LIF neuron is subjected to a lot of synaptic input, it enters a so-called *high-conductance state* (HCS) [Destexhe et al., 2003]. Its membrane potential dynamics are then predominantly driven by the synaptic currents. This allows the reformulation of Equation (2.69) – reduced to LIF – as

$$\tau_{\text{eff}}(t)\frac{\mathrm{d}u_k}{\mathrm{d}t} = u_k^{\text{eff}}(t) - u_k \tag{2.74}$$

$$\tau_{\text{eff}}(t) = \frac{C_m}{g_{\text{tot}}(t)} \tag{2.75}$$

$$u_k^{\text{eff}}(t) = \frac{g_L E_L + \sum_{\text{syn } i} g_i^{\text{syn}}(t) \ E_i^{\text{rev}} + I^{\text{ext}}}{g_{\text{tot}}(t)} \tag{2.76}$$

$$g_{\text{tot}}(t) = g_L + \sum_{\text{syn } i} g_i^{\text{syn}}(t) \tag{2.77}$$

where τ_{eff} is the effective time constant of the membrane dynamics (replacing $\tau_m = \frac{C_m}{g_L}$), u_k^{eff} the effective leak reversal potential which the membrane potential follows with delay τ_{eff} and g_{tot} the total conductance.

In the context of sampling, the strong synaptic input is assumed to be diffuse noise in the form of random spikes from surrounding neurons, which recurrent networks are known to produce [Brunel, 2000]. It is modelled via Poisson spike trains. The higher the firing rate of the noise ($\nu_{\text{syn}} \to \infty$), the larger the *average total conductance* will become ($\langle g_{\text{tot}} \rangle \to \infty$) causing the effective time constant to vanish ($\tau_{\text{eff}} \to 0$): The actual membrane potential essentially follows the effective membrane potential instantaneously. We can thus rewrite

[4]The Heaviside function is defined as follows: $\Theta(x > 0) = 1$, $\Theta(x < 0) = 0$ and $\Theta(x = 0) = \frac{1}{2}$

Equation (2.74) as

$$u_k(t) \approx u_k^{\text{eff}}(t) = \frac{g_L E_L + \sum_{\text{syn } i} \langle g_i^{\text{syn}} \rangle \; E_i^{\text{rev}} + \sum_{\text{syn } i} \Delta g_i^{\text{syn}}(t) \; E_i^{\text{rev}} + I^{\text{ext}}}{\langle g_{\text{tot}} \rangle + \sum_{\text{syn } i} \Delta g_i^{\text{syn}}(t)} \tag{2.78}$$

$$\langle g_{\text{tot}} \rangle = g_L E_L + \sum_{\text{syn } i} \langle g_i^{\text{syn}} \rangle \tag{2.79}$$

$$\Delta g_i^{\text{syn}}(t) = \langle g_i^{\text{syn}} \rangle - g_i^{\text{syn}}(t) \tag{2.80}$$

As shown in [Bytschok, 2011], for Poisson input we have

$$\langle g_i^{\text{syn}} \rangle = w_{ik} \nu_i \tau_{\text{syn}} \tag{2.81}$$

$$\text{Var}[g_i^{\text{syn}}] = \frac{1}{2} w_{ik}^2 \nu_i \tau_{\text{syn}} \tag{2.82}$$

and therefore the relative fluctuation of the synaptic conductances start to vanish for high input rates:

$$\frac{\sqrt{\text{Var}[g_i^{\text{syn}}]}}{\langle g_i^{\text{syn}} \rangle} = \frac{1}{\sqrt{2 \nu_i \tau_{\text{syn}}}} \overset{\nu_i \to \infty}{\longrightarrow} 0 \tag{2.83}$$

This justifies a first order Taylor-expansion of Equation (2.78) in all Δg_i^{syn}:

$$u_k(t) \approx \frac{g_L E_L + \sum_{\text{syn } i} g_i^{\text{syn}}(t) \; E_i^{\text{rev}} + I^{\text{ext}}}{\langle g_{\text{tot}} \rangle} =: \frac{1}{\langle g_{\text{tot}} \rangle} J^{\text{syn}}(t) + \frac{g_L E_L + I^{\text{ext}}}{\langle g_{\text{tot}} \rangle} \tag{2.84}$$

The membrane potential is hence a simple linear transformation of the synaptic noise J^{syn}. From Equation (2.73) is follows that the synaptic noise is described by this *ordinary differential equation* (ODE):

$$\frac{\mathrm{d}J^{\text{syn}}}{\mathrm{d}t} = -\frac{J^{\text{syn}}}{\tau_{\text{syn}}} + \sum_{\text{syn } i} \sum_{t_i^s} \Delta J_i^{\text{syn}} \delta(t - t_i^s) \tag{2.85}$$

where $\Delta J_i^{\text{syn}} = w_{ik} E_i^{\text{rev}}$ and δ is the Dirac delta function[5]. [Petrovici et al., 2013] then link Equation (2.85) to an *Ornstein-Uhlenbeck* (OU) process [Uhlenbeck and Ornstein, 1930]

$$\mathrm{d}x(t) = \theta \cdot (\mu - x(t)) \, \mathrm{d}t + \Sigma \cdot \mathrm{d}W(t) \tag{2.86}$$

with $\theta, \Sigma > 0$. An OU process consists of two parts: The immediate dynamics are described by a *Wiener* process $W(t)$, corresponding to a continuous random walk with variance Σ^2. Overall though, the dynamics tend to drift toward a long time mean μ, mitigated by the drift constant θ.

[5]The Dirac delta function is defined as follows: $\delta(0) = \infty, \delta(x) = 0$ otherwise, so that $\int_{-\infty}^{\infty} \mathrm{d}x f(x) \delta(x - x_0) = f(x_0)$ for any function f.

As for example shown by [Ricciardi, 1977], the probability density function $f(x,t)$ of an OU processes satisfies the following *Fokker-Planck equation*:

$$\frac{\partial f(x,t)}{\partial t} = \theta \frac{\partial}{\partial x}\left[(x-\mu)f\right] + \frac{\Sigma^2}{2}\frac{\partial^2 f}{\partial x^2} \tag{2.87}$$

For initial condition $f(x, t=0) = \delta(x - x_0)$, the unique solution is

$$f(x,t|x_0) = \sqrt{\frac{\theta}{\pi\Sigma^2(1-e^{-2\theta t})}}\exp\left(-\frac{\theta}{\Sigma^2}\left[\frac{(x-x_0e^{-\theta t}-\mu)^2}{1-e^{-2\theta t}}\right]\right) \tag{2.88}$$

which decays over time to a stationary Gaussian distribution

$$f_s(x) = \lim_{t\to\infty} f(x,t|x_0) = \sqrt{\frac{\theta}{\pi\Sigma^2}}\exp\left(-\frac{\theta}{\Sigma^2}(x-\mu)^2\right) \quad . \tag{2.89}$$

As proven in [Gerstner and Kistler, 2002; Petrovici et al., 2013], the following relation holds for the synaptic noise J^{syn} distribution

$$\begin{aligned}\frac{\partial f(J^{\text{syn}},t)}{\partial t} = \frac{1}{\tau_{\text{syn}}}\frac{\partial}{\partial J^{\text{syn}}}\left[\left(J^{\text{syn}} - \sum_{\text{syn }i}\nu_i \Delta J_i^{\text{syn}}\tau_{\text{syn}}\right) f(J^{\text{syn}},t)\right] \\ + \frac{\sum_{\text{syn }i}\Delta J_i^{\text{syn}2}}{2}\frac{\partial^2 f(J^{\text{syn}},t)}{\partial J^{\text{syn}2}}\end{aligned} \tag{2.90}$$

Since Equation (2.90) has the same form as Equation (2.87), it follows that the synaptic noise – and by linear transformation Equation (2.84) the membrane potential as well – can be described by an OU process:

$$\mathrm{d}u_k(t) = \theta\cdot(\mu - u_k(t)) + \Sigma\cdot\mathrm{d}W(t) \tag{2.91}$$

$$\theta = \frac{1}{\tau_{\text{syn}}} \tag{2.92}$$

$$\mu = \left\langle u_k^{\text{eff}}\right\rangle = \frac{I^{\text{ext}} + g_L E_L + \sum_{\text{syn }i}\nu_i w_{ik} E_i^{\text{rev}}\tau_{\text{syn}}}{\langle g_{\text{tot}}\rangle} \tag{2.93}$$

$$\Sigma^2 = \frac{\sum_{\text{syn }i}\nu_i\left[w_{ik}(E_i^{\text{rev}}-\mu)\right]^2\ \tau_{\text{syn}}}{\langle g_{\text{tot}}\rangle} \tag{2.94}$$

We can illustrate Equations (2.91) to (2.94) in the following way: The first part in Equation (2.91) corresponds to the membrane potential following its effective value with time constant τ_{eff}, as we saw already in Equation (2.74). However, since we only consider noisy Poisson input with fixed rates at the moment, the *average* effective potential is constant. We arrive at Equation (2.93) by inserting Equation (2.81) into Equations (2.78) to (2.75) and u_k is constantly driven towards it because the neuron is in a HCS. The Wiener process in the second part of Equation (2.91) is realized by spikes arriving at random times, corresponding to steps in a random walk in either direction. The variance of such a process (Equation (2.94)) depends on the number of steps taken (the firing rate) and the step length (synaptic weight and the distance to the reversal potential) and is calculated using the result Equation (2.82). It is important to note once again that we are only discussing stochastic diffuse input here and are hence only summing over such synapses.

Activation Function

After calculating the probability density function of the membrane potential $f(u_k, t|u_k^0)$, – in order to finally link the stochastic LIF dynamics to Neural Sampling – we need to derive the activation function.

In order to compute the overall probability to find a corresponding binary RV z_k in the active state $p(z_k = 1)$, we consider the situation where the effective membrane potential just crossed the threshold ϑ and the neuron spikes. We therefore have a peaked membrane potential distribution

$$f(u_k^{\text{eff}}, t = t^{\text{spike}}) = \delta(u_k^{\text{eff}} - \vartheta) \tag{2.95}$$

and $z_k = 1$ in the active state. Even though the neuron is refractory (membrane potential is kept fixed at V_{reset}), the effective membrane potential distribution continues to evolve freely (see Equation (2.88) with parameters (2.92) to (2.94) and initial condition (2.95)). After the refractory period, in the limit $\tau_{\text{eff}} \to 0$ the membrane potential instantly jumps back from V_{reset} to the effective membrane potential – see Figure 2.3. As shown in [Petrovici et al., 2013], after each refractory period, there are two cases to be distinguished: The effective membrane potential can potentially be either above or below the threshold ϑ.

If the effective membrane is below the threshold ϑ, it evolves freely until it reaches the threshold ϑ again – on average after the mean *first passage time* (FPT) – and is again described by the initial condition Equation (2.95). If, on the other hand, it is above the threshold, another spike is immediately elicited and the effective membrane potential distribution evolves for another time period $\tau_{\text{on}} = \tau_{\text{refrac}}$ after which the two cases have to be distinguished again.

Therefore, given that the neuron spiked (Equation (2.95) holds), we generally observe an n-spike-burst (with probability P_n) that lasts $n \cdot \tau_{\text{on}}$ – in which the neuron is active – followed by a time period T_n of subthreshold dynamics – in which the neuron is inactive. This leads to:

$$p(z_k = 1) = \frac{\sum_{n=1}^{\infty} P_n \cdot n \cdot \tau_{\text{on}}}{\sum_{n=1}^{\infty} P_n \cdot (n \cdot \tau_{\text{on}} + T_n)} \tag{2.96}$$

If the n spikes occur at times $\{t_0 = t^{\text{spike}}, t_1 = t_0 + \tau_{\text{on}}, \ldots, t_{n-1} = (n-1) \cdot \tau_{\text{on}}\}$ we get

$$P_1 = p(u_1 < \vartheta | u_0 = \vartheta) = \int_{-\infty}^{\vartheta} \mathrm{d}u_1 p(u_1|u_0 = \vartheta) \tag{2.97}$$

$$T_1 = \int_{-\infty}^{\vartheta} \mathrm{d}u_1 \left\langle T\left(\theta, u\right)\right\rangle p(u_1|u_0 = \vartheta) \tag{2.98}$$

where $u_i := u_k^{\text{eff}}(t_i), p(u_i|u_{i-1}) = f(u_i, \tau_{\text{on}}|u_{i-1})$ and $\left\langle T\left(a, b\right)\right\rangle = \left\langle \arg\min_{t \geq 0} u(t) = a | u(0) = b \right\rangle$ is the average FPT (the time it takes the membrane potential to reach a for the first time when initialized to b) for which no closed form expression is known [Thomas, 1975]:

$$\left\langle T\left(a, b\right)\right\rangle = \frac{\theta}{\Sigma}\sqrt{\frac{\pi}{2}} \int_b^a \mathrm{d}u \exp\left(\frac{(u-\mu)^2}{2\Sigma^2}\right)\left[1 + \operatorname{erf}\left(\frac{u-\mu}{\sqrt{2}\Sigma}\right)\right] \tag{2.99}$$

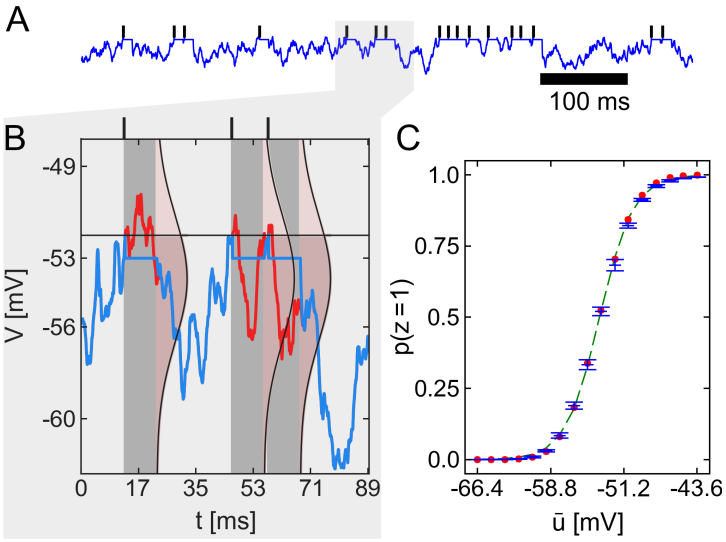

Figure 2.3: **A**: Membrane potential $u_k(t)$ and spikes of a LIF neuron in a noisy environment. **B**: $u_k(t)$ (blue) and $u_k^{\text{eff}}(t)$ (red) in a look almost identical in a HCS, unless the neuron is refractory. After each consecutive spike, the predictive distribution for u_k^{eff} (pink) widens. **C**: Theoretical prediction (red) vs. a sigmoid logistic function $\sigma(\bar{u})$ fitted to simulation results (blue). Taken from: [Petrovici et al., 2013]

All further values can be computed recursively because the effective membrane distribution after the i-th spike is essential the initial distribution for evolution during the $i+1$-st refractory period. [Petrovici et al., 2013] derive the following:

$$P_n = (1 - \sum_{i=1}^{n-1} P_i) \underbrace{\int_{\vartheta}^{\infty} \mathrm{d}u_{n-1}}_{\text{above } \vartheta \text{ up until now}} p(u_{n-1}|u_{n-1} \geq \vartheta) \left[\underbrace{\int_{-\infty}^{\vartheta} \mathrm{d}u_n}_{\text{below } \vartheta \text{ when the burst ends}} p(u_n|u_{n-1}) \right] \tag{2.100}$$

$$T_n = \int_{\vartheta}^{\infty} \mathrm{d}u_{n-1}\, p(u_{n-1}|u_{n-1} \geq \vartheta) \left[\int_{-\infty}^{\vartheta} \mathrm{d}u_n\, p(u_n|\underbrace{u_n < \vartheta}_{\text{renormalization}}, u_{n-1}) \left\langle T\left(u_n, \vartheta\right) \right\rangle \right] \tag{2.101}$$

with

$$p(u_i|u_i \geq \vartheta) = p(u_i|\underbrace{u_i \geq \vartheta}_{\text{renormalization}}, u_{i-1} \geq \vartheta, \ldots, u_0 = \vartheta) \tag{2.102}$$

Furthermore, if a variable appears in both the free as well as conditional part of a probability distribution, it indicates a renormalization of the PDF over the corresponding range.

For additional accuracy, [Petrovici et al., 2013] show that the derivation can be performed for small but non-vanishing τ_{eff} as well. In this case Equations (2.74) and (2.85) represents a system of first-oder ODEs that can be solved by standard techniques (variations of constants). *Post-synaptic potentials* (PSPs), that previously were a linear transformation of the synaptic conductances (Equation (2.84)), are now of difference-of-exponentials shape:

$$u_{\text{PSP}}(t) = \Theta(t - t^{\text{spike}}) \left[\frac{w_i(E_i^{\text{rev}} - \left\langle u^{\text{eff}} \right\rangle)\tau_{\text{syn}}}{\left\langle g_{\text{tot}} \right\rangle} \right] \left[\frac{e^{-\frac{t-t^{\text{spike}}}{\tau_{\text{eff}}}} - e^{-\frac{t-t^{\text{spike}}}{\tau_{\text{syn}}}}}{\tau_{\text{eff}} - \tau_{\text{syn}}} \right] \tag{2.103}$$

Expansion in $\sqrt{\frac{\tau_{\text{eff}}}{\tau_{\text{syn}}}}$ then leads to the corrected mean FPT for the membrane potential to reach the threshold ϑ after being reset:

$$\langle T\left(\vartheta, V_{\text{reset}}\right)\rangle = \tau_{\text{syn}}\sqrt{\pi}\int_{\frac{V_{\text{reset}}-\mu}{\Sigma}}^{\frac{\vartheta^{\text{eff}}-\mu}{\Sigma}} \mathrm{d}x \exp\left(x^2\right)[1+\operatorname{erf}(x)] \tag{2.104}$$

with an effective threshold ϑ^{eff}

$$\vartheta^{\text{eff}} = \vartheta - \zeta\left(\frac{1}{2}\right)\sqrt{\frac{\tau_{\text{eff}}}{2\tau_{\text{syn}}}}\Sigma \tag{2.105}$$

where ζ denotes the Riemann zeta function.

Parameter translation

In order to translate between theoretical (Equations (2.56) and (2.67)) and biological domain (Equation (2.69)), it is easiest to find the parameters for the following relation

$$p(z_k = 1|\mathbf{z}_{\setminus k}) = \sigma(u_k) \overset{!}{=} \sigma\left(\frac{\bar{u_k} - \bar{u}_k^0}{\alpha}\right) \tag{2.106}$$

where u_k denotes the theoretical abstract membrane potential, $\bar{u_k}$ the average free biological one, $\bar{u}_k^0$ is the relative offset at which $p(z_k = 1|\mathbf{z}_{\setminus k}) = \frac{1}{2}$ and α is the lateral dilation that also needs to be accounted for in weight translations. $\bar{u}_k^0$ and α can either be calculated from theory or fitted to simulation data.

Synaptic weights are translated by requiring that the area under the PSP during an active state (with duration τ_{on}) be the same in the theoretical (rectangular shape) and biological (difference-of-exponentials shape) domain.

$$\underbrace{W_{ik}\,\tau_{\text{on}}}_{\text{rectangular PSP}} \overset{!}{=} \frac{1}{\alpha}\int_0^{\tau_{\text{on}}} \mathrm{d}t\; u_{\text{PSP}}(t) \tag{2.107}$$

$$= \frac{w_{ik}\,\tau_{\text{eff}}}{\alpha\left\langle g_{\text{tot}}\right\rangle}\cdot\frac{E_i^{\text{rev}} - \left\langle u_k^{\text{eff}}\right\rangle}{1-\frac{\tau_{\text{syn}}}{\tau_{\text{eff}}}}\left[\tau_{\text{syn}}\left(e^{-\frac{\tau_{\text{on}}}{\tau_{\text{syn}}}}-1\right) - \tau_{\text{eff}}\left(e^{-\frac{\tau_{\text{on}}}{\tau_{\text{eff}}}}-1\right)\right] \tag{2.108}$$

With $\tau_{\text{syn}} = \tau_{\text{on}}$ [Petrovici et al., 2013] obtain the following mapping between theoretical W_{ik} and biological weight w_{ik}:

$$W_{ik} = w_{ik}\cdot\left(\frac{\tau_{\text{eff}}}{\alpha\left\langle g_{\text{tot}}\right\rangle}\cdot\frac{E_i^{\text{rev}} - \left\langle u_k^{\text{eff}}\right\rangle}{1-\frac{\tau_{\text{syn}}}{\tau_{\text{eff}}}}\left[\left(e^{-1}-1\right) - \frac{\tau_{\text{eff}}}{\tau_{\text{syn}}}\left(e^{-\frac{\tau_{\text{syn}}}{\tau_{\text{eff}}}}-1\right)\right]\right) \tag{2.109}$$

$$=: w_{ik}\cdot\frac{1}{f_{\text{theo}\to\text{bio}}} \tag{2.110}$$

Short Term Synaptic Plasticity

Another difference between the theoretical rectangular and the biological double exponential PSP-shape is that the latter has an overshoot ($u_{\text{PSP}}(t^{\text{spike}}+\tau_{\text{on}}) > 0$). If neuron k is burst-firing – corresponding to $z_k = 1$ constantly – the PSPs of subsequent spikes will have a larger influence than the first, distorting the sampling dynamics.

This can be mitigated by employing a *short term plasticity* (STP) model such as the *Tsodyks-Markram* (TM) mechanism [Tsodyks and Markram, 1997; Markram et al., 1998; Tsodyks et al., 1998]. Here, we take into account that neurotransmitters, upon transmitting a spike, are not instantly replenished but instead recovered over time. It is modelled by the following set of differential equations

$$\frac{\mathrm{d}u}{\mathrm{d}t} = -\frac{u}{\tau_{\text{facil}}} + U_{\text{SE}}(1-u)\delta(t-t^{\text{spike}}) \tag{2.111}$$

$$\frac{\mathrm{d}x}{\mathrm{d}t} = \frac{z}{\tau_{\text{rec}}} - u\, x^{-}\, \delta(t-t^{\text{spike}}) \tag{2.112}$$

$$\frac{\mathrm{d}y}{\mathrm{d}t} = -\frac{y}{\tau_{\text{syn}}} + u\, x^{-}\, \delta(t-t^{\text{spike}}) \tag{2.113}$$

$$\frac{\mathrm{d}z}{\mathrm{d}t} = \frac{y}{\tau_{\text{syn}}} - \frac{z}{\tau_{\text{rec}}} \tag{2.114}$$

where $x, y, z \in [0,1]$ denote the fraction of available, active and inactive synaptic vesicles respectively, $x^{-} = x(t^{\text{spike}} - \epsilon)$ is the value of x just prior to spike transmittance and τ_{rec} is the recovery while τ_{facil} is the facilitation time constant. Whenever a spike is transmitted, only a fraction u of the available resources is activated, decays with τ_{syn} and is then recovered via τ_{rec}. This implements *short term depression* (STD). Furthermore, *short term facilitation* (STF) is implemented by the dynamics of u: Each consecutive spike in a short time period ($\sim \tau_{\text{facil}}$) increases the fraction of available resources to be released. If $\tau_{\text{facil}} = 0$, u is simply kept constant at U_{SE}.

Depending on whether the neuron model is current or conductance based, the synaptic current or conductance is then modulated by y as only the active part of neurotransmitters can help in signal transmission:

$$\text{(current-based)} \quad \hat{I}^{\text{syn}}(t) = \sum_{\text{syn } i} w_{ik} \sum_{t_i^s} y_i(t_i^s)\, \exp\left(-\frac{t-t_i^s}{\tau_{\text{syn}}}\right) \Theta\left(t-t_i^s\right) \tag{2.115}$$

$$\text{(conductance-based)} \quad \hat{g}_i^{\text{syn}}(t) = w_{ik} \sum_{t_i^s} y_i(t_i^s)\, \exp\left(-\frac{t-t_i^s}{\tau_{\text{syn}}}\right) \Theta\left(t-t_i^s\right) \tag{2.116}$$

We avoid the overshoot of the exponential PSP by setting $U_{\text{SE}} = 1$, $\tau_{\text{facil}} = 0$ and $\tau_{\text{rec}} = \tau_{\text{syn}}$. The synapse becomes renewing: The inactive fraction (variable z) vanishes and the non-active fraction is always available. This results in the PSP never exceeding the target maximum value – even if the pre-synaptic neuron is constantly spiking.

2.5 Learning

2.5.1 Contrastive Divergence

Contrastive divergence (CD) is a class of algorithms to train RBMs (Section 2.3) to represent arbitrary probability distributions over the visible units [Hinton, 2002, 2010]. As always with *maximum likelihood* (ML) learning (see Section 2.2), the objective function is to minimize the D_{KL} between the *true* distribution $p^*(\mathbf{v})$ and the one represented by the model, $p(\mathbf{v}|\mathbf{W}, \mathbf{a}, \mathbf{b})$. As we know from Equation (2.22) this corresponds to finding:

$$\mathbf{W}^*, \mathbf{a}^*, \mathbf{b}^* = \arg\max_{\mathbf{W},\mathbf{a},\mathbf{b}} \left\langle \ln p(\mathbf{v}|\mathbf{W}, \mathbf{a}, \mathbf{b}) \right\rangle_{p^*(\mathbf{v})} \tag{2.117}$$

$$= \arg\max_{\mathbf{W},\mathbf{a},\mathbf{b}} \left\langle \ln \sum_{\mathbf{h}} p(\mathbf{v}, \mathbf{h}|\mathbf{W}, \mathbf{a}, \mathbf{b}) \right\rangle_{p^*(\mathbf{v})} \tag{2.118}$$

The gradient can be computed as:

$$\frac{\partial}{\partial W_{ij}} \left\langle \ln p(\mathbf{v}) \right\rangle_{p^*(\mathbf{v})} = \left\langle \frac{1}{p(\mathbf{v})} \sum_{\mathbf{h}} \frac{\partial}{\partial W_{ij}} \frac{1}{Z} e^{-E(\mathbf{v},\mathbf{h})} \right\rangle_{p^*(\mathbf{v})} \tag{2.119}$$

$$= \left\langle \frac{1}{p(\mathbf{v})} \sum_{\mathbf{h}} p(\mathbf{v}, \mathbf{h}) \left[v_i h_j \; - \frac{1}{Z} \sum_{\mathbf{v}',\mathbf{h}'} v_i' h_j' e^{-E(\mathbf{v}',\mathbf{h}')} \right] \right\rangle_{p^*(\mathbf{v})} \tag{2.120}$$

$$= \left\langle \sum_{\mathbf{h}} p(\mathbf{h}|\mathbf{v}) v_i h_j \; - \sum_{\mathbf{h}} p(\mathbf{h}|\mathbf{v}) \left\langle v_i' h_j' \right\rangle_{p(\mathbf{v}',\mathbf{h}')} \right\rangle_{p^*(\mathbf{v})} \tag{2.121}$$

$$= \left\langle v_i h_j \right\rangle_{p(\mathbf{h}|\mathbf{v})p^*(\mathbf{v})} \; - \left\langle v_i h_j \right\rangle_{p(\mathbf{v},\mathbf{h})} \tag{2.122}$$

$$=: \left\langle v_i h_j \right\rangle_{\text{data}} - \left\langle v_i h_j \right\rangle_{\text{model}} \tag{2.123}$$

where we omitted out the distributions' (Equation (2.33)) explicit dependency on the parameters. The first term in Equation (2.123) corresponds to an average correlation term between visible and hidden units where the state of visible layer is drawn from the true distribution $p^*(\mathbf{v})$ whereas the second term denotes the average correlation between the two nodes in the current model distribution. Analogously, we find for the biases:

$$\frac{\partial}{\partial a_i} \ln p(\mathbf{v}) = \left\langle v_i \right\rangle_{\text{data}} - \left\langle v_i \right\rangle_{\text{model}} \tag{2.124}$$

$$\frac{\partial}{\partial b_j} \ln p(\mathbf{v}) = \left\langle h_j \right\rangle_{\text{data}} - \left\langle h_j \right\rangle_{\text{model}} \tag{2.125}$$

Parameter updates are hence performed in the direction of the gradients Equations (2.123) to (2.125), modulated by an adaptive learning rate η:

$$\Delta W_{ij} = \eta \cdot \left(\left\langle v_i h_j \right\rangle_{\text{data}} - \left\langle v_i h_j \right\rangle_{\text{model}} \right) \tag{2.126}$$

$$\Delta a_i = \eta \cdot \left(\left\langle v_i \right\rangle_{\text{data}} - \left\langle v_i \right\rangle_{\text{model}} \right) \tag{2.127}$$

$$\Delta b_j = \eta \cdot \left(\left\langle h_j \right\rangle_{\text{data}} - \left\langle h_j \right\rangle_{\text{model}} \right) \tag{2.128}$$

For a fully visible BM with interconnected visible units, but no hidden units, the following learning rule can be derived in an analogue matter:

$$\Delta W_{ij} = \eta \cdot \left(\langle v_i v_j \rangle_{\text{data}} - \langle v_i v_j \rangle_{\text{model}} \right) \tag{2.129}$$

As usual, the true distribution can only be approximated with training samples (often called *input images*) and the updates are calculated by fixing $\mathbf{v}$ and then computing the expectation values with the current model parameters. This is called stochastic steepest ascent.

Unfortunately, calculating the partition function Z in order to determine $\langle \cdot \rangle_{\text{model}}$ scales exponentially with the number of units. It is therefore necessary to approximate the averages $\langle \cdot \rangle_{\text{data}}$ $\langle \cdot \rangle_{\text{model}}$ by drawing samples from the model distribution. This process is called *contrastive divergence* (CD) [Hinton, 2002]. CD_n consists of the following:

- Initialize $\mathbf{v}^{(0)}$ with the training sample.
- Draw a sample $\mathbf{h}^{(0)} \sim p(\mathbf{h}|\mathbf{v}^{(0)})$.
- Draw sample n times:

$$\mathbf{v}^{(i+1)} \sim p(\mathbf{v}|\mathbf{h}^{(i)}) \tag{2.130}$$

$$\mathbf{h}^{(i+1)} \sim p(\mathbf{h}|\mathbf{v}^{(i+1)}) \tag{2.131}$$

- Update weights and parameters according to:

$$\Delta W_{ij} = \eta \cdot \left(v_i^{(0)} h_j^{(0)} - v_i^{(n)} h_j^{(n)} \right) \tag{2.132}$$

$$\Delta a_i = \eta \cdot \left(v_i^{(0)} - v_i^{(n)} \right) \tag{2.133}$$

$$\Delta b_j = \eta \cdot \left(h_j^{(0)} - h_j^{(n)} \right) \tag{2.134}$$

Even with only one sampling step, CD_1, while only crudely approximating the gradient and optimizing a slightly different objective function [Sutskever and Tieleman, 2010], works well in practice.

Further enhancements such as *persistent contrastive divergence* (PCD) [Tieleman, 2008] improve upon CD. Here the model term is essentially sampled from its own Markov chain, independent of the current data sample. If the training images can be labeled somehow, there is usually one such chain for every label. It learns significantly better models than CD_1 or even CD_{10}.

2.5.2 Expectation Maximization

Expectation maximization (EM) is a general technique for finding local maximimum likelihood solutions for probabilistic models with latent variables [Bishop, 2006, chap. 9] in unsupervised or semi-supervised fashion. That is, the model $p(\mathbf{y}, \mathbf{z}|\boldsymbol{\theta})$ includes two distinct kinds of RVs: As usual, we have a set of observed variables $\mathbf{y} = (y_1, y_2, \ldots, y_N)^\top$ which our learning data $\mathcal{Y} = \{\mathbf{y}_1, \mathbf{y}_2, \ldots\}$ is comprised of. Then, there are the latent variables $\mathbf{z} = (z_1, z_2, \ldots, z_K)^\top$. These are not observed and do not appear in the training data and can only be inferred. They

rather correspond to additional information the probabilistic model has about the problem at hand and can help to reduce its dimensionality. Examples include the assignment data points to cluster centers in k-means (see below) or the topics for a given document in probabilistic topic models such as *probabilistic latent semantic analysis* (PLSA) [Hofmann, 1999] or *latent Dirichlet allocation* (LDA) [Blei et al., 2003]. Sometimes they are also referred to as "hidden causes" of the observable data. For instance, when modelling handwritten digits such as the MNIST dataset [LeCun and Cortes, 1998], the latent variables $\mathbf{z}$ may encode which digit is present in each image while the observed variables $\mathbf{y}$ just correspond to the actual pixel values the images are composed of.

As is usually the case with ML learning, the overall objective function to maximize is the expected log-likelihood $\langle \ln p(\mathbf{y}|\boldsymbol{\theta}) \rangle_{p^*(\mathbf{y})}$ where $p^*(\mathbf{y})$ is the true distribution over the observed data, approximated by input samples. However, if direct maximization is difficult while maximizing the *complete* log-likelihood $\ln p(\mathbf{y}, \mathbf{z}|\boldsymbol{\theta})$ is significantly easier, the concept of EM is applicable.

First we note that for any choice of non-vanishing distribution $q(\mathbf{z}|\mathbf{y})$ over the latent variables conditioned on the observed ones, the following decomposition holds:

$$\langle \ln p(\mathbf{y}|\boldsymbol{\theta}) \rangle_{p^*(\mathbf{y})} = \left\langle \sum_{\mathbf{z}} q(\mathbf{z}|\mathbf{y}) \ln p(\mathbf{y}|\boldsymbol{\theta}) \right\rangle_{p^*(\mathbf{y})} \tag{2.135}$$

$$= \left\langle \sum_{\mathbf{z}} q(\mathbf{z}|\mathbf{y}) \ln \left(\frac{q(\mathbf{z}|\mathbf{y})}{q(\mathbf{z}|\mathbf{y})} \cdot \frac{p(\mathbf{y}, \mathbf{z}|\boldsymbol{\theta})}{p(\mathbf{z}|\boldsymbol{\theta}, \mathbf{y})} \right) \right\rangle_{p^*(\mathbf{y})} \tag{2.136}$$

$$= \left\langle \sum_{\mathbf{z}} q(\mathbf{z}|\mathbf{y}) \ln \left(\frac{q(\mathbf{z}|\mathbf{y})}{p(\mathbf{z}|\boldsymbol{\theta}, \mathbf{y})} \right) + \sum_{\mathbf{z}} q(\mathbf{z}|\mathbf{y}) \ln \left(\frac{p(\mathbf{y}, \mathbf{z}|\boldsymbol{\theta})}{q(\mathbf{z}|\mathbf{y})} \right) \right\rangle_{p^*(\mathbf{y})} \tag{2.137}$$

$$= \langle \mathrm{D}_{\mathrm{KL}} \left(q(\mathbf{z}|\mathbf{y}) || p(\mathbf{z}|\mathbf{y}, \boldsymbol{\theta}) \right) \rangle_{p^*(\mathbf{y})} + \langle \mathcal{L}(q, \boldsymbol{\theta}) \rangle_{p^*(\mathbf{y})} \tag{2.138}$$

As can be seen from Equation (2.138) the log-likelihood decomposes into two factors: First we have the D_{KL} between our currently chosen distribution q and the a-posteriori distribution of the model given our current choice of parameters. The second term in Equation (2.138) is a functional $\mathcal{L}(q, \boldsymbol{\theta})$ over the set of all possible distributions $\mathcal{Q}$ over our latent variables. For now this set is unconstrained. Since the D_{KL} is non-negative, it is straightforward to see that $\mathcal{L}(q, \boldsymbol{\theta}) \leq \ln p(\mathbf{y}|\boldsymbol{\theta})$. Therefore $\mathcal{L}(q, \boldsymbol{\theta})$ is a *lower bound* to the log-likelihood.

EM operates in two phases that are repeated until the algorithm convergences to a *local* maximum. Both are detailed below and illustrated in Figure 2.4. In each step, suppose that we start with a current parameter vector $\boldsymbol{\theta}^{\mathrm{old}}$.

Expectation-step

The first thing to note when looking at the decomposition Equation (2.138) is that the left side does not depend on our choice of $q(\mathbf{z}|\mathbf{y})$. Also, the D_{KL} measure is non-negative which

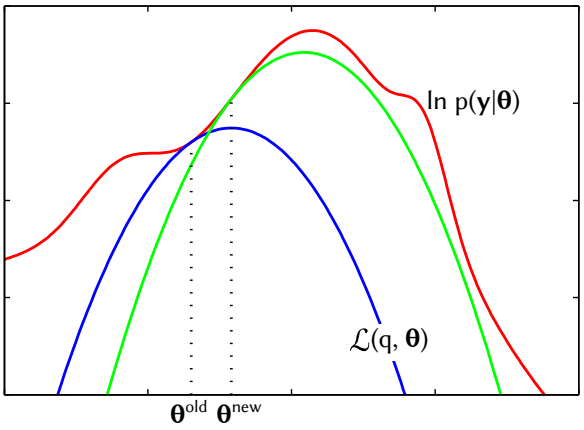

Figure 2.4: Illustration of EM: In the M-step, the parameters $\boldsymbol{\theta}$ are maximized with regard to the current lower bound $\mathcal{L}(q, \boldsymbol{\theta})$ to the log-likelihood $\ln p(\mathbf{y}|\boldsymbol{\theta})$. In the E-step, the distribution $q(\mathbf{z}|\mathbf{y})$ is adjusted by minimizing the D_{KL} between $q(\mathbf{z}|\mathbf{y})$ and the posterior distribution $p(\mathbf{z}|\mathbf{y}, \boldsymbol{\theta})$, ideally setting it to zero. This gives raise to a new lower bound. The two steps are repeated until a local minimum is found. See text for details. Illustration taken from [Bishop, 2006].

means that by minimizing it we are maximizing the lower bound $\mathcal{L}(q, \boldsymbol{\theta})$ while leaving the log-likelihood itself untouched. The *expectation step* (E-step) hence consists of finding:

$$\hat{q}(\mathbf{z}|\mathbf{y}) = \arg\min_{q \in \mathcal{Q}} \left\langle \mathrm{D}_{\mathrm{KL}}(q||p(\mathbf{z}|\mathbf{y}, \boldsymbol{\theta}^{\mathrm{old}})) \right\rangle_{p^*(\mathbf{y})} \tag{2.139}$$

Ideally – if the set of all possible distributions $\mathcal{Q}$ is unconstrained – this simplifies to evaluating the a-posteriori distribution $\hat{q}(\mathbf{z}|\mathbf{y}) = p(\mathbf{z}|\mathbf{y}, \boldsymbol{\theta}^{\mathrm{old}})$ as this reduces the D_{KL} to zero.

Maximization-step

After fixing the distribution $\hat{q}$ we can focus on increasing the lower bound.

$$\begin{aligned}
\boldsymbol{\theta}^{\mathrm{new}} &= \arg\max_{\boldsymbol{\theta}} \left\langle \mathcal{L}(\hat{q}, \boldsymbol{\theta}) \right\rangle_{p^*(\mathbf{y})} && (2.140)\\
&= \arg\max_{\boldsymbol{\theta}} \left\langle \sum_{\mathbf{z}} \hat{q}(\mathbf{z}|\mathbf{y}) \ln\left(p(\mathbf{y}, \mathbf{z}|\boldsymbol{\theta})\right) - \sum_{\mathbf{z}} \hat{q}(\mathbf{z}|\mathbf{y}) \ln\left(\hat{q}(\mathbf{z}|\mathbf{y})\right) \right\rangle_{p^*(\mathbf{y})} && (2.141)\\
&= \arg\max_{\boldsymbol{\theta}} \left\langle \sum_{\mathbf{z}} \hat{q}(\mathbf{z}|\mathbf{y}) \ln\left(p(\mathbf{y}, \mathbf{z}|\boldsymbol{\theta})\right) \right\rangle_{p^*(\mathbf{y})} && (2.142)\\
&= \arg\max_{\boldsymbol{\theta}} \left\langle \ln\left(p(\mathbf{y}, \mathbf{z}|\boldsymbol{\theta})\right) \right\rangle_{p^*(\mathbf{y})\,\hat{q}(\mathbf{z}|\mathbf{y})} && (2.143)
\end{aligned}$$

We see that the *maximization step* (M-step) corresponds to maximizing the average complete log-likelihood under our choice of distribution $\hat{q}(\mathbf{z}|\mathbf{y})$ over the latent variables. As we postulated above, maximizing the complete log-likelihood is significantly easier than maximizing the log-likelihood $p(\mathbf{y}|\boldsymbol{\theta})$ directly.

The change in parameters $\boldsymbol{\theta}^{\mathrm{old}} \rightarrow \boldsymbol{\theta}^{\mathrm{new}}$ obviously changes the a-posteriori distribution, resulting in a now non-zero D_{KL}-term in Equation (2.138). Hence another E-step becomes necessary.

Since the E-step leaves the overall log-likelihood unchanged, and the lower bound is only ever increased in the M-step, the EM algorithm is guaranteed to converge to a *local* maximum.

Example: k-means

A very intuitive example for understanding EM is the k-means algorithm [Steinhaus, 1957; MacQueen, 1967], illustrated in Figure 2.5. It aims to partition a set of observations $\mathcal{Y} = \{\mathbf{y}_1, \mathbf{y}_2, \ldots, \mathbf{y}_n\} \subseteq \mathbb{R}^d$ into k partitions $\mathcal{S} = \{S_1, \ldots, S_k\}$ such that the distance of all points to their respective cluster center $\boldsymbol{\mu}_i$ is minimal

$$\arg\min_{\mathcal{S}} \sum_{i=1}^{k} \sum_{\mathbf{y} \in S_i} ||\mathbf{y} - \boldsymbol{\mu}_i||^2 \qquad \boldsymbol{\mu}_i = \frac{1}{|S_i|} \sum_{\mathbf{y} \in S_i} \mathbf{y} \tag{2.144}$$

The algorithm then operates as follows: In the E-step we simply assign each data point to the closest cluster centers (which are initialized randomly). In the M-step we recompute the positions of each clusters' center from all data points currently assigned to it.

Even though k-means does not operate on probability distributions and instead uses hard assignments, we can still draw parallels to the working principles of EM: The observed variables are the data points, the latent variables correspond to the cluster assignment of each data point, whereas the positions of the cluster centers correspond to the free parameters of the model we want to optimize. The updated assignments in the E-step correspond to a re-evaluation of the a-posteriori distribution given the current cluster positions (the current parameter vector). In the M-step the cluster center are updated as a direct result of our choice of distribution over the assignments (i.e., the latent variables).

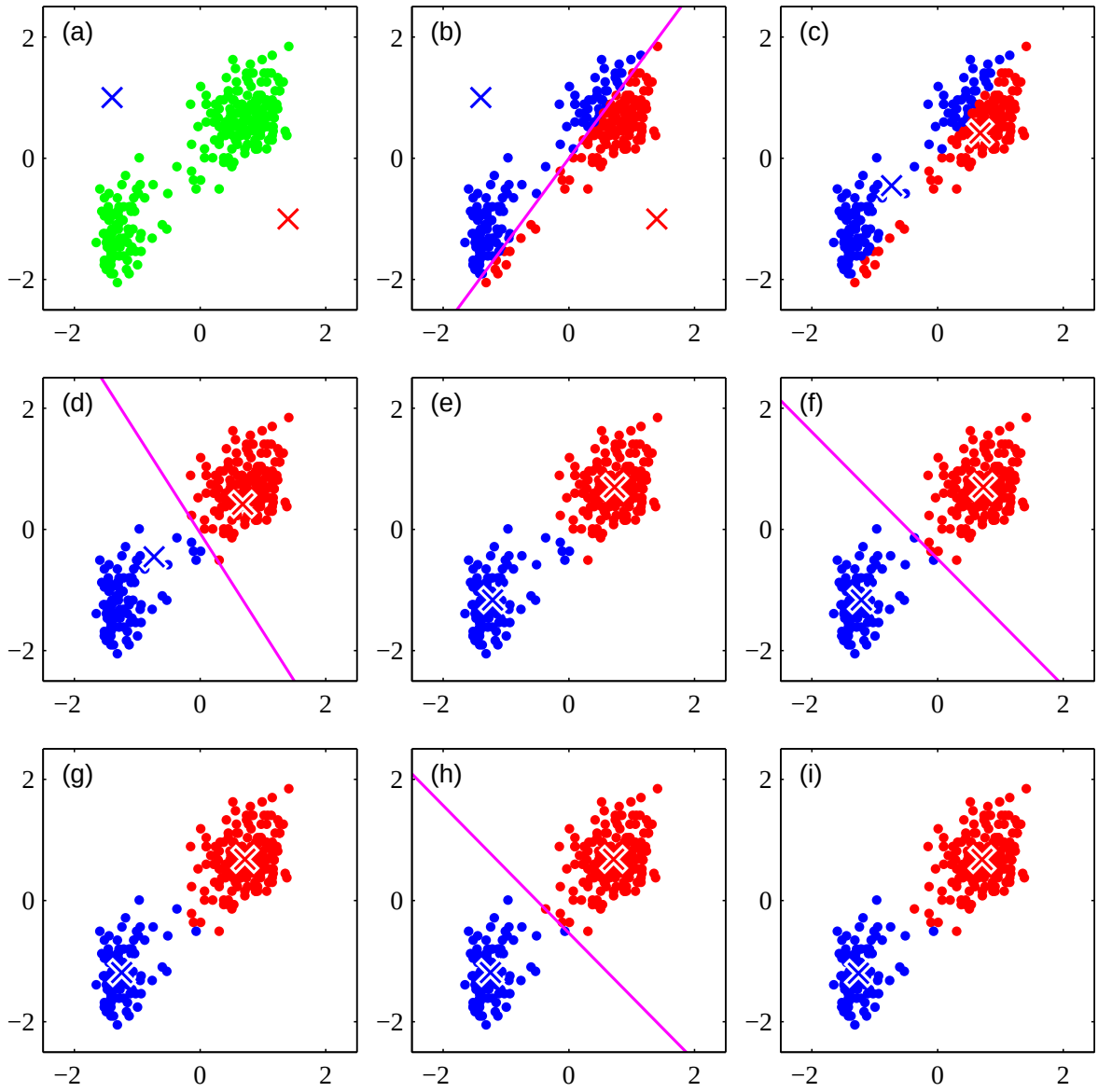

Figure 2.5: Several training steps of the k-means algorithm to illustrate the principle of EM: In the E-step we simply assign each data point to the closest cluster centers (which are initialized randomly). In the M-step we recompute the positions of each clusters' center from all data points currently assigned to it. See text for details. Illustration taken from [Bishop, 2006]

.

2.5.3 Spike-based Expectation Maximization

The concept of EM can be extended to the realm of Neural Sampling, aptly called *spike-based expectation maximization* (SEM). Here a group of stochastic neurons receives structured spiking input. Using a combination of *spike timing dependent plasticity* (STDP) and intrinsic excitability modification, it can be shown that they perform a form of stochastic EM to find the "hidden causes" in their input, effectively reducing its dimensionality [Nessler et al., 2008, 2009, 2013]. Broadly speaking, a stochastic online version of the EM algorithm is applied in the following way: Spikes from the cause layer neurons correspond to a stochastic evaluation of the E-step, as it samples from the current posterior of the underlying generative model $p(\mathbf{z}|\mathbf{y}, \boldsymbol{\theta})$, where $\boldsymbol{\theta} = \{\mathbf{b}', \mathbf{V}, \mathbf{W}\}$ is its parameter vector. The M-step is then realized via local STDP update rules. Both steps will later be detailed further. Among other things, this network motif can be used to learn Bayes-optimal decisions [Nessler et al., 2008], to optimally decode population codes [Nessler et al., 2009, 2013; Habenschuss et al., 2013], be embedded in bigger Bayesian inference tasks [Bill et al., 2015] or even extended to learn *hidden Markov models* (HMMs) when reward-gated STDP learning is added to facilitate rejection sampling [Kappel et al., 2014]. Furthermore, SEM-like update rules are suitable to be implemented on neurally inspired hardware components such as Memristors [Bill and Legenstein, 2014].

The explicit generative model used as a starting point for the translation towards implementation on neuromorphic hardware (see Chapter 5) used for SEM is taken on [Bill et al., 2015]. The general network architecture is outlined in Figure 2.6: The *cause layer*, a group of stochastic neurons, receives structured input in form of Poisson spike trains from an input layer. The cause layer itself forms a BM with very strong negative weights V_{kl}, effectively turning it into a *winner-take-all* (WTA) circuit: Whenever one cause layer neuron fires the others are strongly inhibited, ideally making it impossible for two cause layer neurons to be active at the same time ($V_{kl} \to -\infty$). WTAs circuits are a common network structure when modelling the neocortex [Lundqvist et al., 2006, 2010]. We then introduce the following gen-

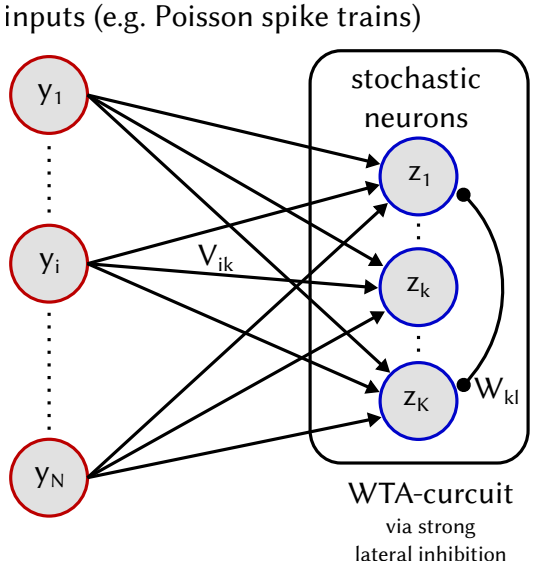

Figure 2.6: Overview of the network architecture. The cause layer – comprised of stochastic theoretical neurons – receives input from an input layer that is modelled via Poisson spike trains. The weights W_{kl} between causelayer neurons are strongly inhibitory to facilitate a WTA-like structure[6]. Only one cause layer neuron should ideally respond to each presented input pattern. The weights V_{kl} between input and cause layer evolve according to update rule Equation (2.163).

erative model:

$$p(\mathbf{y}, \mathbf{z}|\boldsymbol{\theta}) = p(\mathbf{y}|\mathbf{z}, \boldsymbol{\theta}) \cdot p(\mathbf{z}|\boldsymbol{\theta}) \tag{2.145}$$

In accordance with the previous sections, $\mathbf{y} = (y_1, \ldots, y_N)^\top$ denotes the N input variables and $\mathbf{z} = (z_1, \ldots, z_K)^\top$ are the K binary latent variables. Since the cause layer forms a BM we have as prior

$$p(\mathbf{z}|\boldsymbol{\theta}) = \frac{1}{Z_{\text{prior}}} \exp\left[\frac{1}{2}\mathbf{z}^\top \mathbf{W}\mathbf{z} + \mathbf{b'}^\top \mathbf{z}\right] \tag{2.146}$$

which is just a more compact way of writing Equation (2.28). Since we demand that at most one cause layer neuron be active at any time, only one z_k can explain the input pattern currently presented to the network. All input variables y_i are hence independent given the state of the latent variables.

$$p(\mathbf{y}|\mathbf{z}, \boldsymbol{\theta}) = \prod_i p(y_i|\mathbf{z}, \boldsymbol{\theta}) \tag{2.147}$$

Whenever the input neuron i fires, the variables y_i is increased by one for a time period τ_{syn}, corresponding to non-renewing rectangular PSPs. $y_i(t)$ therefore denotes the number of afferent spikes in the time interval $(t - \tau_{\text{syn}}, t]$. This is also called *eligibility trace* in the context of STDP learning. Since the input stream consists of Poisson spike trains, y_i follows a Poisson distribution:

$$p(y_i|\mathbf{z}, \boldsymbol{\theta}) = \underbrace{\text{Pois}\left(y_i \mid \lambda_{i0}\right)^{1-\sum_k z_k}}_{\text{default hypothesis}} \prod_k \text{Pois}\left(y_i \mid \lambda_{ik}\right)^{z_k} \tag{2.148}$$

Where we have the constraint $0 \leq \sum_k z_k \leq 1$ as two latent variables cannot be active at the same time. If z_k is active, we have

$$y_i \sim \text{Pois}\left(y_i \mid \lambda_{ik}\right) := \frac{(\lambda_{ik})^{y_i}}{y_i!} e^{-\lambda_{ik}} \tag{2.149}$$

that is, the input neuron i fires with rate $\nu_{ik} = {}^{\lambda_{ik}}/_{\tau_{\text{syn}}}$. The firing statistics of each input variable are "explained" by the active cause layer neuron alone. If, however, no cause layer neuron is active, the input is said to be explained by a *default hypothesis* – also called *null cause* – in which each neuron i spikes with rate $\nu_{i0} = {}^{\lambda_{i0}}/_{\tau_{\text{syn}}}$. This default hypothesis cannot be learnt but is rather set by the experimenter prior to learning. It follows:

$$p(y_i|\mathbf{z}, \boldsymbol{\theta}) = \text{Pois}\left(y_i \mid \lambda_{i0}\right) \prod_k \left[\frac{\text{Pois}\left(y_i \mid \lambda_{ik}\right)}{\text{Pois}\left(y_i \mid \lambda_{i0}\right)}\right]^{z_k} \tag{2.150}$$

$$= \underbrace{\frac{(\lambda_{i0})^{y_i}}{y_i!} e^{-\lambda_{i0}}}_{=:h(y_i)} \prod_k \left[\left(\frac{\lambda_{ik}}{\lambda_{i0}}\right)^{y_i} e^{-(\lambda_{ik}-\lambda_{i0})}\right]^{z_k} \tag{2.151}$$

$$= h(y_i) \exp \sum_k \left[y_i \underbrace{\ln\left(\frac{\lambda_{ik}}{\lambda_{i0}}\right)}_{=:V_{ik}} z_k - \underbrace{(\lambda_{ik} - \lambda_{i0})}_{=:A_{ik}} z_k \right] \tag{2.152}$$

where $h(y_i)$ is a helper function gathering all terms that will play no role for inference. Please note that the factors $A_{ik} = e^{V_{i0}} \cdot (e^{V_{ik}} - 1)$ (where $V_{i0} = \log \lambda_{i0}$) were only introduced for notational convenience and do not constitute independent parameters. Combining Equations (2.146), (2.147) and (2.152) we arrive at the full joint probability:

$$p(\mathbf{y}, \mathbf{z}|\boldsymbol{\theta}) = p(\mathbf{y}, \mathbf{z}|\mathbf{b}', \mathbf{V}, \mathbf{W}) = \frac{h(\mathbf{y})}{Z_{\text{prior}}} \exp\left[\frac{1}{2} \mathbf{z}^\top \mathbf{W} \mathbf{z} + \mathbf{y}^\top \mathbf{V} \mathbf{z} + \underbrace{(\mathbf{b}' - \mathbf{1}^\top \mathbf{A})^\top}_{=:\mathbf{b}} \mathbf{z} \right] \tag{2.153}$$

$$= \frac{h(\mathbf{y})}{Z_{\text{prior}}} \exp\left[\frac{1}{2} \mathbf{z}^\top \mathbf{W} \mathbf{z} + \mathbf{y}^\top \mathbf{V} \mathbf{z} + \mathbf{b}^\top \mathbf{z} \right] \tag{2.154}$$

where $h(\mathbf{y}) := \prod_i h(y_i)$, Z_{prior} is the normalizing constant of the prior and $\mathbf{1} = (1, \ldots, 1)^\top$. Please note that $\mathbf{b}$ still depends on both $\mathbf{b}'$ and $\mathbf{V}$.

In order to show that our network can perform a stochastic online version of EM [Sato, 1999], we need to investigate two things: First – for the E-step – we need to show that spikes from the cause layer actually represent spikes from the model posterior distribution $p(\mathbf{z}|\mathbf{y}, \boldsymbol{\theta})$. Secondly, the to be derived STDP update rules have to raise the average complete log-likelihood – corresponding to the M-step (compare Section 2.5.2, Equation (2.143)).

Expectation-Step

We employ the NCC, Equation (2.55), to find

$$u_k \stackrel{!}{=} \ln \frac{p(z_k = 1|\mathbf{z}_{\setminus k}, \mathbf{y}, \boldsymbol{\theta})}{p(z_k = 0|\mathbf{z}_{\setminus k}, \mathbf{y}, \boldsymbol{\theta})} \tag{2.155}$$

$$= \ln \left[\frac{p(z_k = 1, \mathbf{z}_{\setminus k}, \mathbf{y}|\boldsymbol{\theta}) / \sum_{z_k} p(z_k, \mathbf{z}_{\setminus k}, \mathbf{y}|\boldsymbol{\theta})}{p(z_k = 0, \mathbf{z}_{\setminus k}, \mathbf{y}|\boldsymbol{\theta}) / \sum_{z_k} p(z_k, \mathbf{z}_{\setminus k}, \mathbf{y}|\boldsymbol{\theta})} \right] \tag{2.156}$$

$$= \ln \left[\frac{p(z_k = 1, \mathbf{z}_{\setminus k}, \mathbf{y}|\boldsymbol{\theta})}{p(z_k = 0, \mathbf{z}_{\setminus k}, \mathbf{y}|\boldsymbol{\theta})} \right] \tag{2.157}$$

$$= \frac{1}{2} \sum_{j \neq k} (1 \cdot W_{kj} z_j + z_j W_{jk} \cdot 1) + \sum_i y_i V_{ik} \cdot 1 + b_k \cdot 1 + \underbrace{\ln(1)}_{\substack{\text{all terms from Equation (2.154) without } z_k \\ \text{appear in numerator and denominator}}} \tag{2.158}$$

$$= \sum_{j \neq k} z_j W_{jk} + \sum_i y_i V_{ik} + b_k \tag{2.159}$$

where we made use of the fact that $W_{kj} = W_{jk}$ and $W_{kk} = 0$. This corresponds to each cause layer neuron k receiving input from the input layer via weights $\mathbf{V}$, from the other cause layer neurons via $\mathbf{W}$, while being driven by an effective bias b_k. We see that our generative model Equation (2.154) does indeed satisfy the NCC. The spiking activity of the cause layer therefore does represent samples from the posterior distribution (see Equation (2.139)), as was required.

Maximization-Step

The gradient of the average complete log-likelihood with the respect to our varying model parameters is (remember that the prior-BM weights $\mathbf{W}$ and biases $\mathbf{b}'$ are kept fixed)

$$\frac{\partial}{\partial V_{ik}} \left\langle \ln p(\mathbf{y}, \mathbf{z}|\boldsymbol{\theta}) \right\rangle_{p^*(\mathbf{y})\hat{q}(\mathbf{z})} = \left\langle \frac{\partial}{\partial V_{ik}} \ln p(\mathbf{y}, \mathbf{z}|\boldsymbol{\theta}) \right\rangle_{p^*(\mathbf{y})\hat{q}(\mathbf{z})} \tag{2.160}$$

$$= \left\langle y_i z_k - z_k \frac{\partial}{\partial V_{ik}} A_{ik} \right\rangle_{p^*(\mathbf{y})\hat{q}(\mathbf{z})} \tag{2.161}$$

$$= \left\langle z_k \left(y_i - e^{V_{ik}+V_{i0}} \right) \right\rangle_{p^*(\mathbf{y})\hat{q}(\mathbf{z})} \tag{2.162}$$

By rescaling with $e^{V_{ik}+V_{i0}}$ we arrive at the following learning rule:

$$\frac{\mathrm{d}V_{ik}}{\mathrm{d}t} = \eta \cdot z_k(t) \cdot \left(y_i(t)\, e^{-(V_{ik}+V_{i0})} - 1 \right) \tag{2.163}$$

Since the weight update points in the direction of the complete average log-likelihood gradient, the lower bound $\mathcal{L}(\hat{q}, \boldsymbol{\theta})$ is always increased (compare Equation (2.143)). It is important to note that, in the current model definition, changing the weights V_{ik} also changes the effective biases, since $b_k = (b'_k - \sum_i A_{ik})$. The sum over all inputs i is a non-local operation as the cause layer neuron k needs to know the synaptic efficacies of all afferent input neurons. This deficit is remedied in Section 2.5.4.

The formalism presented can be applied to all likelihood distributions $p(\mathbf{y}|\mathbf{z}, \boldsymbol{\theta})$ which are members of the natural exponential family

$$p(\mathbf{y}|\mathbf{z}, \mathbf{V}) = h(\mathbf{y}) \exp\left[(\mathbf{V}\mathbf{z})^\top \mathbf{y} - A(\mathbf{V}\mathbf{z}) \right] \tag{2.164}$$

with "arbitrary"[7] $h(\mathbf{y}), \mathbf{V}\mathbf{z}, A(\mathbf{V}\mathbf{z}) = \mathbf{1}^\top \mathbf{A}(\mathbf{V}\mathbf{z})\mathbf{1}$. Each probability distribution (e.g., Bernoulli, Gaussian or – as presented here – Poisson) leads to slightly different update rules [Bill et al., 2015].

2.5.4 Homeostasis

As already discussed in Section 2.5.3, in "regular" SEM, each cause layer neuron k needs to keep track of all its afferent synaptic weights V_{ik} in order to adjust its effective bias $b_k = b'_k + \sum_i A_{ik}(V_{ik})$. Furthermore, all patterns need to be normalized, otherwise some patterns have an intrinsic "advantage" over others to be evoked even though both appear at the same frequency. This can only be somewhat accounted for by adjusting the regular bias $\mathbf{b}'$.

A more plausible approach to account for time varying $\mathbf{A}$ is to apply homeostasis to the activity of the cause layer neurons. Thas is, we impose a posterior constraint on the latent

[7] $p(\mathbf{y}|\mathbf{z}, \mathbf{V})$ still has to be a proper probability distribution.

variables [Habenschuss et al., 2012, 2013; Bill et al., 2015]: Each cause layer neuron is actively trying to maintain an externally set target activity m_k. Theoretically, the set $\mathcal{Q}$ over all possible distributions to chose from in the E-step (see Section 2.5.2) becomes constricted:

$$\mathcal{Q}_{\text{hom}} = \left\{ q : \langle z_k \rangle_{p^*(\mathbf{y})\, q(\mathbf{z}|\mathbf{y})} = m_k \ \forall k \right\} \tag{2.165}$$

where $\sum_k m_k =: p_{\text{net}} \leq 1$ represents the total probability of one cause layer neuron in the network being active at any given time. This means that we expect the null cause to explain the input $1 - p_{\text{net}}$ of the time.

These constraints can be taken into account during the E-step via the use of *Lagrange multipliers*. The Lagrange function Λ is then

$$\Lambda(q) = \underbrace{\langle \mathrm{D}_{\mathrm{KL}}(q(\mathbf{z}|\mathbf{y}) || p(\mathbf{z}|\mathbf{y}, \boldsymbol{\theta})) \rangle_{p^*(\mathbf{y})}}_{\text{unconstrained objective function}} - \sum_k \beta_k \underbrace{\left(\langle z_k \rangle_{q(\mathbf{z}|\mathbf{y})\, p^*(\mathbf{y})} - m_k \right)}_{\substack{\text{constraint on target activity} \\ m_k}} - \lambda \underbrace{\left(\langle 1 \rangle_{q(\mathbf{z}|\mathbf{y})} - 1 \right)}_{q \text{ properly normalized}} \tag{2.166}$$

where β_k, λ are the Lagrange multipliers. Setting the derivative with respect to $q(\mathbf{z}|\mathbf{y})$ to zero for any choice of $\mathbf{z}$ and $\mathbf{y}$ yields the form of the optimal solution $\hat{q}(\mathbf{z})$:

$$\frac{\partial}{\partial q(\mathbf{z}|\mathbf{y})} \Lambda(q) = \left\langle \ln \left(\frac{q(\mathbf{z}|\mathbf{y})}{p(\mathbf{z}|\mathbf{y}, \boldsymbol{\theta})} \right) + \underbrace{q(\mathbf{z}|\mathbf{y}) \, \frac{p(\mathbf{z}|\mathbf{y}, \boldsymbol{\theta})}{q(\mathbf{z}|\mathbf{y})} \, \frac{1}{p(\mathbf{z}|\mathbf{y}, \boldsymbol{\theta})}}_{=1} - \lambda - \sum_k z_k \, \beta_k \right\rangle_{p^*(\mathbf{y})} \stackrel{!}{=} 0 \tag{2.167}$$

Here, we made use of $\langle 1 \rangle_{q(\mathbf{z}|\mathbf{y})} = \langle 1 \rangle_{q(\mathbf{z}|\mathbf{y})\, p^*(\mathbf{y})}$, stating that $p^*(\mathbf{y})$ is normalized. Since Equation (2.167) has to vanish for arbitrary choices of $p^*(\mathbf{y})$, we can conclude that the term in angle brackets over which we compute the average has to vanish.

$$0 \stackrel{!}{=} \ln \left(\frac{q(\mathbf{z}|\mathbf{y})}{p(\mathbf{z}|\mathbf{y}, \boldsymbol{\theta})} \right) + 1 - \lambda - \sum_k z_k \, \beta_k \tag{2.168}$$

$$\implies \hat{q}(\mathbf{z}) = p(\mathbf{z}|\mathbf{y}, \boldsymbol{\theta}) \ \exp \left[\lambda - 1 + \sum_k \beta_k z_k \right] \tag{2.169}$$

$$= \underbrace{\exp\left[\lambda - 1\right]}_{\text{normalization}} \ \exp \left[\frac{1}{2} \, \mathbf{z}^\top \mathbf{W} \, \mathbf{z} + \mathbf{y}^\top \mathbf{V} \mathbf{z} + \underbrace{\left(\mathbf{b}' - \mathbf{1}^\top \mathbf{A} - \boldsymbol{\beta} \right)^\top}_{=: \mathbf{b}^{\text{hom}}} \mathbf{z} \right] \tag{2.170}$$

$$= \exp\left[\lambda - 1\right] \ \exp \left[\frac{1}{2} \, \mathbf{z}^\top \mathbf{W} \, \mathbf{z} + \mathbf{y}^\top \mathbf{V} \mathbf{z} + {\mathbf{b}^{\text{hom}}}^\top \mathbf{z} \right] \tag{2.171}$$

We see that we now sample from modified version of the old posterior (which is normalized due to the free factor $e^{\lambda - 1}$) where the biases are now driven by the factors β_k (still to be determined).

We first solve for λ:

$$1 \stackrel{!}{=} \sum_{\mathbf{z}} \hat{q}(\mathbf{z}) = e^{\lambda - 1} \sum_{\mathbf{z}} p(\mathbf{z}|\mathbf{y}, \boldsymbol{\theta}) \exp\left(\boldsymbol{\beta}^\top \mathbf{z} \right) \tag{2.172}$$

$$\iff \lambda = 1 - \ln \sum_{\mathbf{z}} p(\mathbf{z}|\mathbf{y}, \boldsymbol{\theta}) \ \exp\left(\boldsymbol{\beta}^\top \mathbf{z} \right) \tag{2.173}$$

Inserting Equations (2.169) and (2.173) into Equation (2.166) yields the dual $\Psi(\boldsymbol{\beta})$ where most terms cancel out.

$$\Psi(\boldsymbol{\beta}) = \left\langle \ln \left(\frac{p(\mathbf{z}|\mathbf{y}, \boldsymbol{\theta})}{p(\mathbf{z}|\mathbf{y}, \boldsymbol{\theta})} e^{(\lambda - 1 + \boldsymbol{\beta}^\top \mathbf{z})} \right) \right\rangle_{p^*(\mathbf{y})\, \hat{q}(\mathbf{z})} - \sum_k \beta_k \left(\langle z_k \rangle_{p^*(\mathbf{y})\, \hat{q}(\mathbf{z})} - m_k \right) \tag{2.174}$$

$$= \left\langle (\lambda - 1)\ \langle 1 \rangle_{\hat{q}(\mathbf{z})} \right\rangle_{p^*(\mathbf{y})} + \left\langle \boldsymbol{\beta}^\top \mathbf{z} \right\rangle_{p^*(\mathbf{y})\, \hat{q}(\mathbf{z})} + \boldsymbol{\beta}^\top \mathbf{m} - \left\langle \boldsymbol{\beta}^\top \mathbf{z} \right\rangle_{p^*(\mathbf{y})\, \hat{q}(\mathbf{z})} \tag{2.175}$$

$$= \boldsymbol{\beta}^\top \mathbf{m} - \left\langle \ln \sum_{\mathbf{z}} p(\mathbf{z}|\mathbf{y}, \boldsymbol{\theta})\ \exp\left(\boldsymbol{\beta}^\top \mathbf{z} \right) \right\rangle_{p^*(\mathbf{y})} \tag{2.176}$$

Here we used the fact that $\langle 1 \rangle_{\hat{q}(\mathbf{z})} = 1$ by choice of λ (see Equation (2.173)) twice: To cancel out the last term of Equation (2.166) and to drop the averages over $q(\mathbf{z})$ from Equation (2.175) to Equation (2.176).

Analogous to [Habenschuss et al., 2012; Graca et al., 2007], by performing gradient ascent on the dual $\Psi(\boldsymbol{\beta})$, we get the gradient – and therefore the update rule – for the effective biases (which are only a linear transformation of $\boldsymbol{\beta}$):

$$\frac{\partial}{\partial b_k^{\text{hom}}} \Psi(\boldsymbol{\beta}) = \frac{\partial}{\partial \beta_k} \Psi(\boldsymbol{\beta}) = \frac{\partial}{\partial \beta_k} \left[\boldsymbol{\beta}^\top \mathbf{m} - \left\langle \ln \sum_{\mathbf{z}} p(\mathbf{z}|\mathbf{y}, \boldsymbol{\theta})\ \exp\left(\boldsymbol{\beta}^\top \mathbf{z} \right) \right\rangle_{p^*(\mathbf{y})} \right] \tag{2.177}$$

$$= m_k - \left\langle \frac{\sum_{\mathbf{z}}\ z_k\, p(\mathbf{z}|\mathbf{y}, \boldsymbol{\theta})\ \exp\left(\beta^\top \mathbf{z} \right)}{\sum_{\mathbf{z}'}\ p(\mathbf{z}'|\mathbf{y}, \boldsymbol{\theta})\ \exp\left(\beta^\top \mathbf{z}' \right)} \right\rangle_{p^*(\mathbf{y})} \tag{2.178}$$

$$= m_k - \langle z_k \rangle_{p^*(\mathbf{y})\, \hat{q}(\mathbf{z}|\mathbf{y})} \tag{2.179}$$

$$\implies \frac{\mathrm{d} b_k^{\text{hom}}}{\mathrm{d}t} = \eta_b \cdot \left(m_k - \langle z_k \rangle_{p^*(\mathbf{y})\, q(\mathbf{z}|\mathbf{y})} \right) \tag{2.180}$$

where η_b is the corresponding learning rate. Since we wanted to minimize the D_{KL} in Equation (2.166), we have to maximize the dual and point our update rule in ascending gradient direction. Intuitively, whenever the activity of neuron k is below the target m_k the intrinsic excitability b_k^{hom} is increased. Conversely, if the average activity is too high, b_k^{hom} is reduced. The new bias-adjustments $\boldsymbol{\beta}$ "take care" of the non-localities $\mathbf{A}$. Furthermore, the update rules Equation (2.180) are completely local and therefore do not rely on any distributed information.

3 Neuromorphic Hardware

Given the difficulties of observing neurons and synapses *in vivo*, computer simulations have always been an important tool to investigate neural systems. These artificial neural networks are an invaluable tool to investigate both low- and high-level network behavior and can serve as the basis for new applications in many areas, such as robotics, ambient intelligence or human machine interfaces.

Neural network consist of many units, each computing rather simple functions over their input (i.e., other afferent neurons). This is fundamentally different from the von Neumann architecture employed in the vast majority of information technology today. Here, computation is concentrated in comparatively few cores which can carry out more complex tasks. Unfortunately, this technology is slowly running into physical limitations in terms of computation speed and power consumption [Borkar and Chien, 2011], highlighting the need for alternative technology when the famous yet fully empiric *Moore's Law* [Moore, 1965] can no longer be upheld.

One of these alternatives is *neuromorphic computing* [Mead, 1990; Douglas et al., 1995]. It aims to build upon and mimic the working principles of biological neural networks: Instead of solving a set of differential equations evolving in parallel via numerical methods, a *physical model* – electronic components behaving similarly to their biological counterparts – is *emulated* rather than simulated [Schemmel et al., 2010]. *very-large-scale integration* (VLSI) technology operates on much smaller time scales than typical neural structures in the brain due to smaller capacities. Therefore, emulation in silicon has the benefit that it is much faster compared to biological real time. This allows for long term studies as well as vast parameter sweeps to be conducted in acceptable time frames. Compared to traditional information technology, these massively parallel circuits exhibit a much larger fault tolerance as most components are non-essential and defects can simply be worked around, just like in the brain. It is therefore feasible to do *wafer-scale integration*, that is, to not cut the silicon-wafer into individual dies, but rather to inter-connect the chips in a post-processing step directly on the wafer. Also, it is several orders of magnitude more energy efficient than traditional super-computer simulations [Müller, 2014].

The *Neuromorphic Physical Model System 1* (NM-PM1), in development throughout the successive projects *Fast Analog Computing with Emerging Transient States* (FACETS), *BrainScaleS* (BSS) and *Human Brain Project* (HBP), follows these design goals [HBP SP9 partners, 2014; Brüderle et al., 2011]. As can be seen in Figure 3.1, it is comprised of several wafer-modules, each of which is separated into 48 reticles of which each consists of 8 *high input count analog neural network* (HICANN) chips – the smallest conceptual building block of the system. Each wafer can emulate up to $180\,000$ neurons and $40\,000\,000$ synapses [Schemmel et al., 2010]. On the wafer itself, spikes are routed via a network of asynchronous buses – the so called *layer-1*

(L1) communication network. The system was designed with scalability in mind, allowing for the inter-connection of several wafers via a synchronous packet-based communication network called *layer-2* (L2).

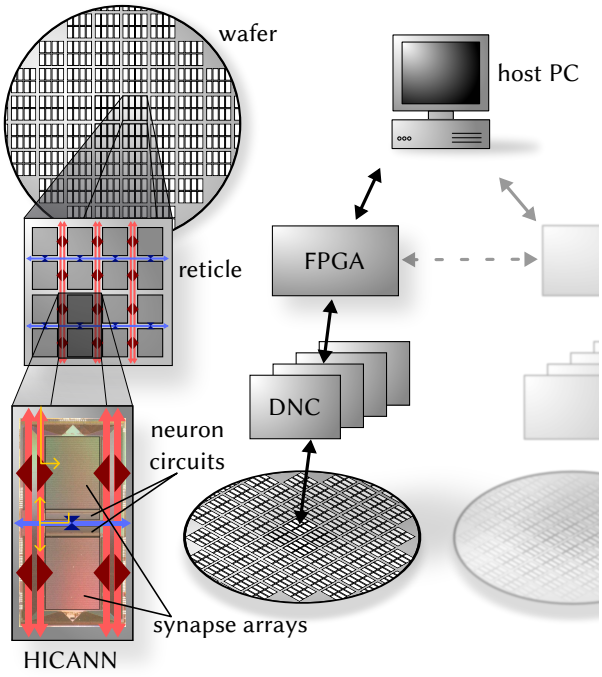

Figure 3.1: Short architectural overview of the NM-PM1. Left: Each wafer consists of 48 reticles, whereas each reticle consists of 8 HICANN chips. Each HICANN has two symmetric halves with neuron circuits and synapse arrays. HICANNs are interconnected via horizontal (blue) and vertical (red) asynchronous buses than span the entire wafer (layer-1). An exemplary spike path is shown in yellow: The incoming spike packet is routed to the synapse drivers. In case a neuron spikes the resulting spike packet is emitted back onto the routing network. Right: The off-wafer communication is realized by a hierarchical packet-based network (layer-2) via Digital Network Chips (DNC) and Field Programmable Gate Arrays (FPGA). Illustration taken from: [Petrovici et al., 2014]

This chapter is organized in the following way: First, the centerpiece of the current hardware generation, the HICANN building block, is introduced. Then the current generation *spike timing dependent plasticity* (STDP) update controller and the planned next-generation *plasticity processing unit* (PPU) are introduced. The complete system specification can be found in [HBP SP9 partners, 2014].

3.1 HICANN Building Block

The smallest conceptual building block of the NM-PM1 is the *high input count analog neural network* (HICANN) building block [Schemmel et al., 2008, 2010]. It was developed within the *BrainScaleS* (BSS) project and has a symmetric structure: Top and bottom half are mirrored versions of each other (see Figure 3.1 or Figure 3.2 left). The central part containing the analog circuitry is aptly named *analog network core* (ANC). It consists of two parts: The 2×256 neuron circuits – called *dendrite membranes* (DenMems) – and the much bigger synapse arrays. Each implements the *adaptive exponential integrate-and-fire* (AdEx) neuron model with conductance-based synapses (discussed in Section 2.4.4) with two synaptic input channels and the possibility for constant external current injection. The DenMems are subdivided into 8 blocks of 64 circuits each. In each block, membrane potentials of several DenMems can be short circuited to create larger neurons or simply combined to form multi-compartment models [Millner, 2012].

The 224×256 synapse array represents the synaptic connections realized on the particular chip. It is fed by 2×56 synapse drivers, each one supplying two synapse rows of the array. Each DenMem receives input from one synaptic column, leading to 224 possible inputs per DenMem and up to $224 \times 64 = 14336$ possible inputs per neuron (assuming a completely connected DenMem block).

In case of a spike, a digital spike pulse packet is generated, consisting only of the 6-bit address of the originating neuron. In the merger tree, this packet is then either time-stamped and sent off-wafer via the L2 network or injected onto one of the 64 horizontal buses. Here it is routed throughout the wafer via sparse crossbar switches (Figure 3.2 left) towards the target synapse driver. There, the digital spike event is translated back to an analog signal (hence the term *mixed-signal hardware*). Based on the top two weights of the afferent neuron's address, the signal is routed onto one of four *strobe lines*. Both the lower 4 address bits and a reference signal of length τ^{STDF} (modelling *short term plasticity* (STP), either depressing or facilitating, see Section 2.4.4) are transmitted into the synapse array. Each synapse is statically connected to one of the four strobe lines. If the four bits match the address stored in a synapse, a 4-bit *digital-analog converter* (DAC) generates the final current signal representing the conductance with height proportional to the maximum conductance $g^{\max}$ multiplied by the 4-bit weight w^{syn} for duration τ^{STDF} so that the total charge transmitted corresponds to the *Tsodyks-Markram* (TM)-modulated synaptic weight of the corresponding synaptic connection. See Figure 3.2 right for an illustration.

The $g^{\max}$ of two adjacent synapse drivers can be configured to be a fixed multiple of each other, thereby increasing the weight resolution to 8-bit while halving the number of synapses for these synapse drivers.

Furthermore, besides the neuron circuits, there are 8 *linear-feedback shift registers* (LFSRs) present on each HICANN. They serve as source of pseudo-randomness by injecting spikes onto the L1 buses.

3.2 STDP Update Controller

Long term learning is incorporated into every synapse via *spike timing dependent plasticity* (STDP) [Morrison et al., 2008]. Due to size constraints, there is always a trade-off between the number of realizable synapses and the complexity of each synapses circuitry [Schemmel et al., 2006, 2010]. Having full-fledged STDP circuitry in every synapse is very costly in terms of die area, therefore a compromise had to be reached: Each synapse only stores local correlation information whereas the synaptic weights are updated periodically by a global update controller[1]. The reasoning behind this is that weight dynamics typically evolve on slower time scales than the immediate neural activity [Morrison et al., 2007; Kunkel et al., 2011] and hence correlation measurements and weight updates can be separated.

[1] One update controller per half of the HICANN chip (224×256 synapses total per controller).

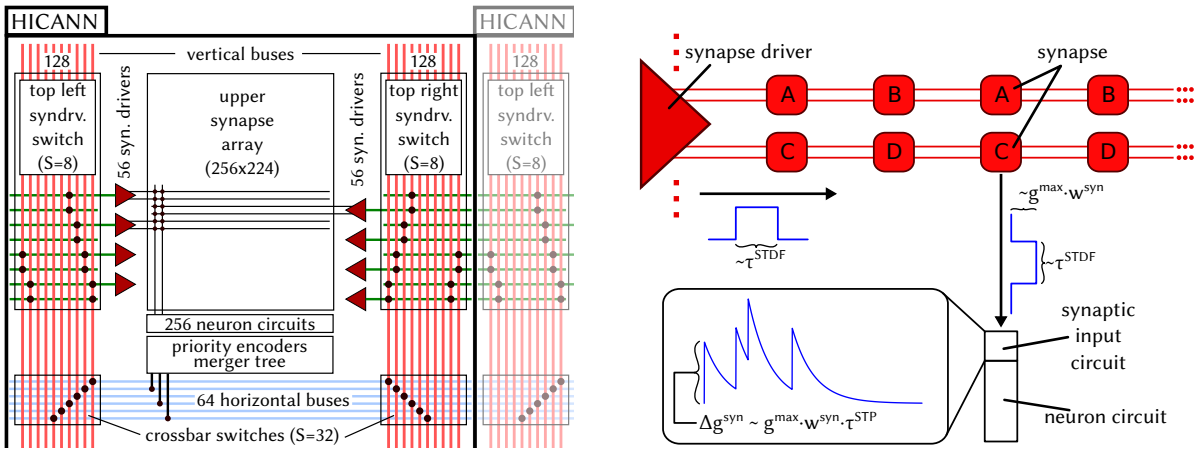

Figure 3.2: Left: Components and connectivity of the HICANN building block. Shown is the upper part of the symmetric chip. The by far largest part is covered by the synapse array with $56 \times 2 \times 2 = 224$ pre-synaptic inputs and 256 outputs, one for each DenMem neuron circuit, located at the center of the chip. Multiple DenMems can be combined to form larger, multi-compartment neurons. Whenever a neuron circuit spikes, a digital asynchronous event is emitted consisting only of the 6-bit neuron address. These pulses are then routed via two statically configurable switches (crossbar rsp. synapse driver switch) in the asynchronous L1 network to the target synapse drivers, operating two synapse rows each. The switches themselves are sparse, meaning that not at every crossing exists a configurable switch (the synapse driver switches only connect every 8-th row resulting in a *sparseness* $S = 8$). It is important to note that the buses do not stop at the HICANN boundaries but extend throughout the whole wafer. Illustration taken from: [Petrovici et al., 2014]. Right: Conceptual overview of the synapse driver. Each synapse row driver listens for incoming spike pulses and routes them onto one of four strobe lines based on top two bits of the afferent neuron's address and emits a pulse packet with length τ^{STDF} (the length is determined by a possibly active TM mechanism). Each synapse is connected to one of the four strobe lines (indicated by A-D). If the lower four bits match, the synapses then reroute the pulse into the corresponding column. The strength of the synaptic conductance is modulated by the window length τ^{STDF}, the maximum conductance $g^{\max}$ (set for the whole row) and the actual 4-bit synaptic weight w^{syn} so that the total charge applied corresponds to the TM-modulated synaptic weight (see Section 2.4.4).

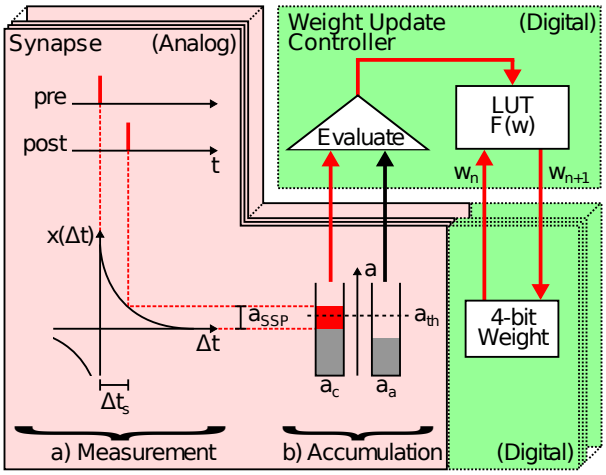

Figure 3.3: Overview of the working principles of the STDP update controller. Red: For each nearest-neighboring spike pair, a charge a_{SSP} – decaying exponentially with the pair's absolute time distance Δt – is accumulated either in the causal (pre-post pairing) or anti-causal (post-pre pairing) capacitor (a_{c} or a_{a} respectively). Green: Upon updating a synaptic weight, the charge in both capacitors is compared against a threshold a_{th} from which it is decided in which out of three possible LUTs a new digital weight value is looked up based on the current one and written back to the synapse. If the weight was updated, *both* capacitors are reset. Illustration taken from: [Pfeil et al., 2012]

The correlation measurement is performed in analog hardware and accumulated between weight updates in each synapse. For this, there are two capacitors within each synapse. As can be seen in Figure 3.3, for every post-synaptic (pre-synaptic) spike we measure the absolute time distance to the latest pre spike (post spike) and apply a charge decaying exponentially with said time distance to the causal (anti-causal) capacitance – see Figure 3.4.

The update controller then periodically reads out the capacitors synapse row by synapse row, compares their charges in an adjustable fashion to controllable thresholds and – based on that – may chose one (or none) of three *look up tables* (LUTs) that are programmed prior to each experiment. In this LUT the new digital 4-bit weight is chosen based on the current one and written back to the synapse. After each successful update, *both* capacitors are reset.

Despite these constraints, this STDP-scheme is able to successfully perform many learning tasks [Pfeil et al., 2012].

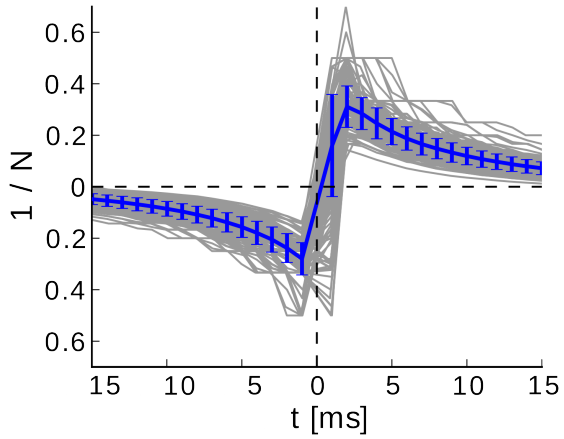

Figure 3.4: STDP curves (grey) measured for 252 neurons (in a single synapse row) on the Spikey chip – a predecessor to the HICANN – along with their mean and standard deviation (blue): Since the STDP window can only be measured indirectly, we measure how many spike pairs with a given time difference t it takes to trigger a causal ($t > 0$) or anti-causal ($t < 0$) weight update. The inverse $1/N$ then corresponds to the STDP window shape. Plot taken from [Pfeil et al., 2012]

3.3 Plasticity Processing Unit

While the NM-PM1 being assembled and tested at the time of writing of this thesis, there are nevertheless improved follow-up components planned and still under active development. One such improvement is the HICANN-DLS [Hartel and Schemmel, 2014], a successor to the current-generation HICANN chip. While the specifications have not been finalized at the time of writing, there are nevertheless some noteworthy concepts worth of discussion in the context of learning.

The first point is that – due to further miniaturization – components shrink in size, allowing the circuits to become more complex. Hence, the weight resolution of synapses can be increased, presumably to 6-bit (corresponding to a fourfold increase in possible weight-values).

The major improvement in terms of STDP is the inclusion of the so-called *plasticity processing unit* (PPU) [Friedmann, 2013]. Instead of fixed LUTs in which the new weight is looked up in a *deterministic* fashion, the PPU features a *microprocessor* capable of computing the updated weights during the experiment. It implements a subset of the PowerISA 2.06 specification for 32-bit architectures [PowerISA, 2010] and can be programmed using standard tools, namely the C programming language and the GNU Compiler Collection. By including an external reward signal, it allows for three-factor STDP learning. Here, the "regular" STDP signal is modulated by an external reward signal that is either computed on the PPU itself or received in a closed-loop manner from the outside [Friedmann et al., 2013].

The PPU also allows for *stochastic weight updates*. Instead of having the next weight be a deterministic function of the current weight and the state of the correlation capacitors, weights can be updated in a non-deterministic, stochastic fashion. By choosing the update probabilities accordingly, this can virtually increase the weight resolution, as many weight updates are attempted, but only few succeed while ensuring that the long-term averages of the weights are the same as if the weight resolution was much higher. This is not possible with fixed LUTs as having the next weight be the current weight causes the weight to be stationary under the same correlation conditions, i.e., whenever this LUT is chosen.

Furthermore, instead of comparing the charge in both capacitors to fixed thresholds, HICANN-DLS will feature the direct readout of both charges via *analog-to-digital converter* (ADC) – presumably with 8-bit resolution. This allows weight updates based on the concrete charge deposited in *both* capacitors instead of just whether a threshold was crossed. In the same vein, the possible problem of reseting *both* capacitors independent of which charge crossed what threshold is alleviated.

4 Simulation Software

Over the last 20 years, a variety of simulation environments have emerged in computational neuroscience [Brette et al., 2007]. Each was born out of a different set of needs – be it the modelling of cellular dynamics in *in-vivo* or *in-vitro* experiments, generating the EEG impression of vast networks of point neurons or the efficient simulation of abstract theoretical models – and hence each have their own individualities in scope, parameter naming conventions, choice of algorithms and programming language.

In this chapter we will quickly introduce the existing simulation software upon which the then developed frameworks (see Chapter 6) are based.

4.1 PyNN

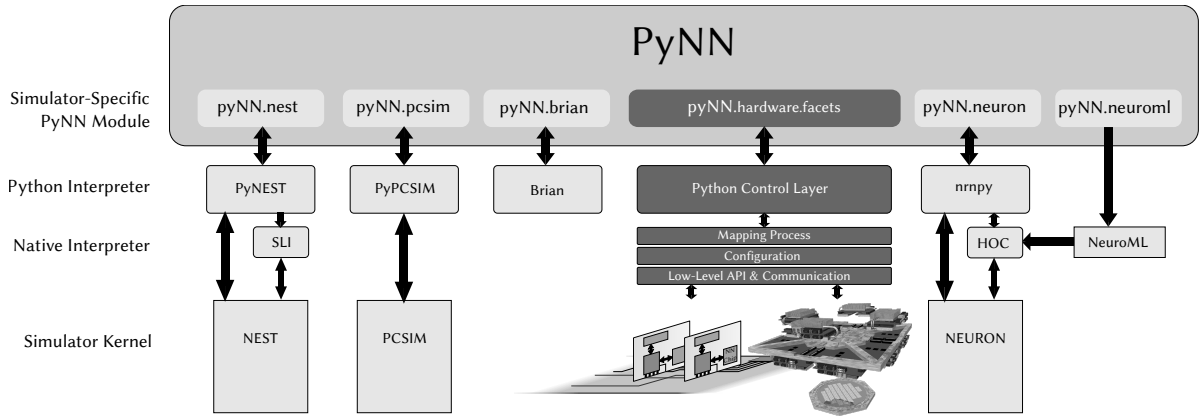

Figure 4.1: Schematic of the simulator-independent modelling language PyNN. See text for details. Taken from: [Brüderle et al., 2011]

PyNN (pronounced 'pine') is a backend-agnostic modelling language for neural networks [Davison et al., 2008]. Written in Python [Rossum, 2000], PyNN aims to unify descriptions of model networks over a large set of simulators. Up until its conception, code written for one simulator – in the worst case – had to be rewritten from scratch in order to be run on a different back-end, severally hindering the reproducibility of simulation results.

PyNN defines a set of standard neuron models and synapse types which need to be supported by each backend. Its *application programming interface* (API) abstraction is two-fold: On the lower end, the user can specify and neurons and connect them directly in a very flexible way, while the high-level API operates on the scale of populations that can be connected

via several different synapse types in a series of connection schemes (all-to-all, one-to-one, random, distance dependent etc.).

After the user has specified his network model, depending on which backend was chosen, PyNN then translates the common parameters and network description into the backend-specific equivalent in an automatic fashion. In the ideal case, switching from one backend to the next can be done by just changing one line of code[1]. The network simulation is then performed in the backend's native environment. Any possible results (e.g., recorded spike trains/voltage traces) are loaded back into the Python environment after the simulation run has completed.

As can be seen in Figure 4.1, PyNN currently supports the following simulation and emulation backends: NEST [Diesmann and Gewaltig, 2002], NEURON [Hines and Carnevale, 2003], PCSIM [Pecevski et al., 2009], Brian [Brette and Goodman, 2008], NeuroML [Gleeson et al., 2010] and the *Neuromorphic Physical Model System 1* (NM-PM1)-backend as described in Chapter 3. Especially in the latter case, having an abstract network description is invaluable since the average hardware user will not have the working knowledge to configure, for example, crossbar switches or translate between parameters from the physical and biological regime.

4.2 NEST

For this thesis, most[2] simulations were carried out using *NEural Simulation Tool* (NEST) [Diesmann and Gewaltig, 2002] as backend for PyNN. Written in C++, it focuses mainly on the simulation of large networks of point neurons with biologically realistic connectivity patterns. Since it is designed to be run on super-computers where performance is of critical issue, supports parallelization in the forms of multi-threading and multi-processing (in the form of message passing), allowing simulations to be distributed over multiple cores on a single machine as well as multiple machines in a network. It is optimized for high performance computing and can easily handle up to 10^5 neurons, making heavy use of the *GNU Scientific Library* (GSL) [Galassi et al., 2009], especially for solving *ordinary differential equations* (ODEs) and random number generation. Memory usage is minimized by storing synapse information only on the node computing the post-synaptic neuron [Morrison et al., 2005]. Furthermore, more complex synapse types store their static common parameters only once for all their instances to conserve even more memory so that each instance can read them from the same memory location.

NEST features several user interfaces: First and oldest, there is *SLI*, a native interpreter using a high level scripting language. This stack-oriented programming language with postfix notation might seem a bit outlandish compared to more modern scripting languages such as Python; it is used to create and interconnect networks, read and set parameters, define and

[1] `import pyNN.<backend> as sim`

[2] As discussed in Section 6.2, only the abstract theoretical models were simulated using a custom Boost.Python-wrapped C++ solution. All other simulations were carried out using NEST.

execute functions (e.g., probabilistic or distance dependent connectivity) and also features exception handling in case an error occurs during simulation. Furthermore there is *PyNEST*, a Python API that has the same feature set as SLI since internally it translates user commands are translated to SLI instructions. Finally, there the already mentioned NEST backend for PyNN, implementing PyNN's API on top of PyNEST (see Figure 4.1).

4.3 Contributions

During this thesis, some programming work has been done on PyNN. Several patches were submitted for the `pyNN.nest`-backend to make it more compatible when handling native NEST synapses: In order to conserve memory, NEST has the notion of *common synapse parameters* that are set for a whole synapse model. This way, on each computing node every model instance can simply look up these common parameters at the same memory location. Each synapse then only has to store local parameters (e.g., its weight or its local state in case of synapse dynamics) resulting in fewer memory used in total.

For the standard models, common parameters were handled correctly. However, when using native nest synapse types – which are only thinly wrapped – PyNN had no notion of local and common synapse parameters. This lead to errors in the execution of simulations since local and common parameters are set via different functions in PyNEST (`SetStatus` versus `SetDefaults`). For this thesis, a lot of custom synapse types were developed to study various aspects of emulating *spike-based expectation maximization* (SEM)-learning on hardware (see Chapter 3). It was therefore necessary to extend the `PyNN.nest`-backend to distinct between both types of parameters.

Furthermore, several new models and synapse types in NEST were developed during this thesis. They are detailed in Section 6.4.

5 Waferscale Neuromorphic SEM: Challenges & Solutions

Learning tasks such as *spike-based expectation maximization* (SEM) are typically conducted over very long time periods and are thus resource intensive to carry out on regular simulation equipment. The very high speed-up factor of $10^3 - 10^5$ on neuromorphic hardware would shorten the time span required for each network's emulation from hours or even days for larger networks to a mere few seconds. This provides many interesting opportunities for conducting large scale real world learning tasks.

Unfortunately, we are not as flexible in regards to what network dynamics we can emulate, as neuron dynamics and the *spike timing dependent plasticity* (STDP)-processing elements are implemented as physical components in hardware and thus fixed. This makes a one-to-one implementation of some learning tasks difficult. Thus, one of the main goals of this thesis was a feasibility study as to whether the theory of spike-based expectation maximization (described in Section 2.5.3) could be implemented in neuromorphic hardware, namely the the waferscale system *Neuromorphic Physical Model System 1* (NM-PM1) (described in Chapter 3).

5.1 Network Setup

The network setup is detailed in Figure 5.1. It closely resembles the setup in the original model (see Figure 2.6): An input layer – whose structured firing characteristics we want to learn – is projecting onto the (hidden) cause (detection) layer. This cause layer forms a BM over binary variables $\mathbf{z}$ with very strong inhibitory weights so that – in a WTA-like fashion – only one neuron may be active at any time. Nevertheless, there are some key differences to the original model: The cause layer neurons – which are intrinsically stochastic in the theoretical model – are now deterministic LIF-neurons (see Section 2.4.4) that are brought into the stochastic regime by both excitatory as well as inhibitory Poisson background stimulus.

The mutual inhibition of the cause layer neurons is realized via an inhibitory population consisting – for computational simplicity – of a single LIF-neuron with parrot-like behavior: Its parameters and the weight of all synapses projecting onto it were chosen in such a way that each spike from the cause layer is enough to elicit a spike from the inhibitory inter-neuron (after roughly $1.2\,\mathrm{ms}$). The inhibitory population projects onto all cause layer neurons with very strong inhibitory weights so that all cause layer neurons are prohibited from spiking while the currently active one is in its refractory period. The choice to implement the WTA-dynamics of the cause layer in such a way was made for two reasons: Firstly, we want to

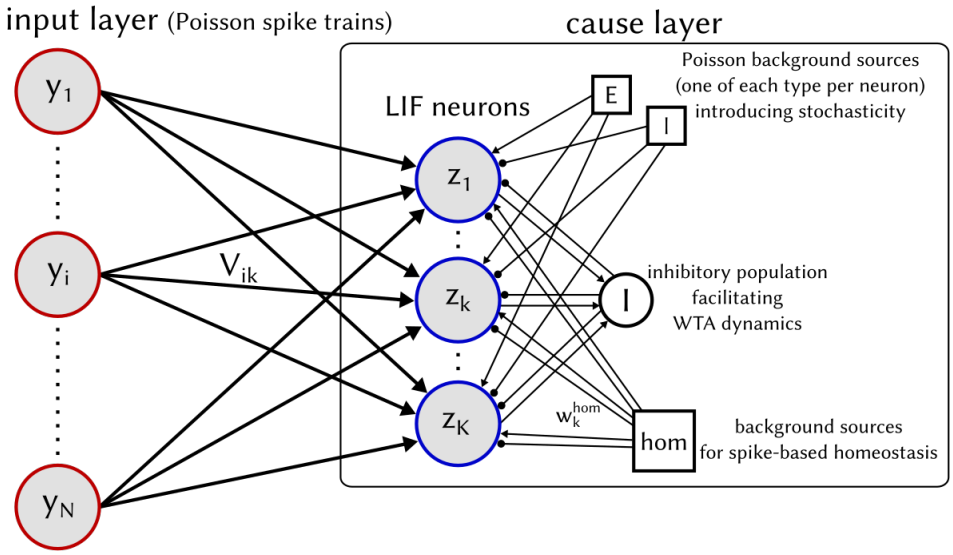

Figure 5.1: Overview of the network architecture. The cause layer – comprised of LIF neurons brought into the stochastic regime by excitatory and inhibitory Poisson input – receives input from an input layer that is modelled via Poisson spike trains. Its aim is to distinguish hidden causes in the presented input stimuli. The cause layer neurons are connected via an inhibitory population with parrot-like behavior: Each spike from a cause layer neuron elicits a spike from the inhibitory population, preventing all other cause layer neurons from firing. The cause layer therefore forms a WTA-like structure (representing a BM with very strong inhibitory weights). Therefore, it follows that only one cause layer neuron can ideally respond to each presented input pattern. The weights V_{ik} between input and cause layer evolve according to update rules detailed in Section 5.3. The activity of each cause neuron is kept at a predetermined value via dynamic synapses, implementing a form of spike-based homeostasis described in Section 5.2.

uphold *Dale's Law* [Eccles et al., 1954] when embedding the excitatory pyramidal cells the cause layer represents into larger network structures and, secondly, implementing mutual inhibition over a common inter-population takes much less routing resources in hardware as we only need to connect each cause layer neuron to the inhibitory population and vice versa (versus connecting every cause layer neuron with every other cause layer neuron). Another benefit is that inhibition becomes symmetric. If the active cause layer neuron is directly inhibiting all others, a new problem arises: The inhibition needs to be strong enough to prevent any other cause layer neuron from spiking. After the refractory period is over, each neuron should have the same initial probability to spike again (before accounting for the current input etc.). This becomes very hard to tune: If the weight is too low, several cause layer neurons can be active at the same time. If it is too high, the currently active cause layer neuron has a distinct advantage to spike again after its refractory period. This is especially crucial in the initial training phase. Both problems are eliminated if *all* neurons receive the same inhibitory signal. Finally, the delay between the cause layer neuron spike and the inhibitory signal was chosen to correspond to the average fixed delay we experience in hardware (on biological time scales), as emulations are executed in real time and the physical signals travel at finite velocity through the silicon substrate. Direct instantaneous inhibition

is hence not possible. See Section 5.4.6 for further information how this affects network dynamics.

Another difference is the way homeostasis is implemented: In the theoretical model the self-regulatory mechanisms were directly integrated into the neuron models whereas on hardware, we have to resort to external solutions. They are detailed in Section 5.2.

Finally, for a variety of reasons, the original update rule Equation (2.163) for the weights V_{ik} between input and cause layer is not directly transferable to the hardware domain and therefore needs to be modified. The adjustments are detailed in Section 5.3. During simulation, the theoretical weight is translated from the Boltzmann-regime to biological synaptic conductances using the weight translation factor $f_{\text{theo}\to\text{bio}}$ (see Equation (2.110)).

5.2 Spike-based Homeostasis

As outlined in Section 2.5.4, we need a way to regulate the effective bias b_k^{hom} of each cause layer neuron (representing binary variable z_k) in order to maintain a set target activity m_k. The derived updated rule was Equation (2.180):

$$\frac{\mathrm{d}b_k^{\text{hom}}}{\mathrm{d}t} = \eta_b \cdot \Big(m_k - \langle z_k \rangle_{p^*(\mathbf{y})\, q(\mathbf{z}|\mathbf{y})} \Big) \tag{5.1}$$

where $p_{\text{net}} = \sum_k m_k \leq 1$ is the network's total probability to explain the observed input (as opposed to the null cause), $p^*(\mathbf{y})$ is the true input distribution, η_b the learning rate and $q(\mathbf{z}|\mathbf{y})$ a distribution from the constrained set $\mathcal{Q}_{\text{hom}}$ evaluated in the *expectation step* (E-step):

$$\mathcal{Q}_{\text{hom}} = \Big\{ q : \langle z_k \rangle_{p^*(\mathbf{y})\, q(\mathbf{z})} = m_k \ \forall k \Big\} \tag{5.2}$$

In computer simulations Equation (5.1) can be implemented immediately – either as a dynamic intrinsic variable when simulating the abstract theoretical model (detailed in Section 2.4.3) or indirectly as a dynamic current injection or shift of the resting membrane potential when simulating with LIF neurons (see Section 2.4.4).

However, as detailed in Section 3.1, adjusting neuron parameters mid-simulation on the NM-PM1 is not feasible. We therefore need another way to modulate the firing activity of each cause layer neuron externally. A straightforward spike-based solution is to connect each neuron z_k to an additional background source. The synaptic weight w_k^{hom} is then adjusted as follows

$$\Delta w^{\text{hom}} = \begin{cases} +\, c^{\text{pre}} & \text{for every pre-synaptic spike} \\ -\, c^{\text{post}} & \text{for every post-synaptic spike} \end{cases} \tag{5.3}$$

where c^{pre} and c^{post} are two adjustable constants. At equilibrium we have:

$$\left\langle w_k^{\text{hom}} \right\rangle \stackrel{!}{=} 0 \tag{5.4}$$

$$\implies c^{\text{pre}} \cdot \nu^{\text{pre}} = c^{\text{post}} \cdot \nu^{\text{post}} \tag{5.5}$$

where ν^{pre} and ν^{post} are the firing rates of the pre- and post-synaptic neuron (i.e., additional background source and cause layer neuron). By rearranging we obtain

$$\nu^{\text{post}} = \frac{c^{\text{pre}}}{c^{\text{post}}} \cdot \nu^{\text{pre}} =: c^{\text{hom}} \cdot \nu^{\text{pre}} \tag{5.6}$$

We can thus adjust the post-synaptic firing rate by choosing the ratio of absolute weight updates accordingly.

In order to translate the update rule for a given target activity m_k, we first note that a response probability of $p_{\text{net}} = 1$ corresponds to the cause layer firing whenever possible. Since the weights $\mathbf{W}$ of the corresponding BM over the cause layer are infinitely negative, no two cause layer neurons can be active at the same time. Hence, the cause layer can emit a spike after every refractory period τ_{refrac}. The highest possible firing rate is therefore $\nu_{\text{net}}^{\text{max}} = {}^{1}/_{\tau_{\text{refrac}}}$. The target firing rate for cause layer neuron k firing with a fraction m_k of the networks total activity is then

$$m_k \cdot \nu^{\text{net}} \overset{!}{=} \nu_k^{\text{pre}} = c_k^{\text{hom}} \cdot \nu^{\text{pre}} \tag{5.7}$$

$$\implies \qquad c_k^{\text{hom}} = m_k \cdot \frac{\nu^{\text{net}}}{\nu^{\text{pre}}} \tag{5.8}$$

The final spike-based homeostatic update rule is therefore

$$\Delta w^{\text{hom}} = \eta_b \begin{cases} +m_k \cdot \frac{\nu^{\text{net}}}{\nu^{\text{pre}}} & \text{for every spike from background source} \\ -1 & \text{for every spike from cause layer neuron } k \end{cases} \tag{5.9}$$

Intuitively, whenever the k-th cause layer neuron is not active enough (or not active at all) the weight is increased on average, whereas if it is firing too often the weights are decreased.

In actual simulations, in order to satisfy *Dale's Law* [Eccles et al., 1954], two separate homeostasis synapses (with two separate sources) are created pre cause layer neuron. One is restricted to positive weights while the other one is restricted to negative ones only.

As background sources for spike-based homeostasis the first and most readily available model was the default Poisson generator implementation. This has some draw backs, due to the irregularity of the emitted spike train, the overall homeostatic influence varies and is not as constant as a directly adjusted bias would be. By using high firing rates, this effect can be reduced. It can be remedied further by using regular spike trains from a more periodic source model (described in Section 6.4.11) implemented later during this thesis. Both source models are compared in Section 5.4.4. At lower rates, they are able to influence network dynamics substantially.

Potential Emulation on Hardware

Equation (5.9) is not the only way to implement spike-based homeostasis. For technical reasons in *NEural Simulation Tool* (NEST), it was the most straightforward one, showing the feasibility of controlling the cause layer's effective biases externally. When emulating on NM-PM1, spike-based homeostasis will have to be implemented slightly differently.

The first option is to use the *plasticity processing unit* (PPU) (see Section 3.3): Since the background source is firing at a high and fixed rate, the charge in both capacitors – causal as well as anti-causal – can serve as an estimate of the cause layer neurons firing rate upon which the synaptic weight is adjusted. By using stochastic updates, we can virtually increase the 4–6 bit weight resolution. This means that while the actual weight value is constantly changing (due to stochastic updates), its time average will correspond to the target value which lies in between the realizable weights (see Section 5.3.4 below). It remains to see of the limited number of possible weight-values has any detrimental effect on the performance of homeostasis.

A second and in the author's opinion more promising option is to perform the network emulation in a closed-loop setup. This means that the cause layer spikes are sent directly to the controlling host computer while the network emulation is still taking place. The host computer is keeping track of the average firing rates, implementing a slightly modified version of Equation (5.9) (linear weight increase instead of after each background source signal). It then sends spikes to the cause layer via the *layer-2* (L2) communication network. Due to the massive speed-up of the NM-PM1, any calculations performed on the host computer have to keep up with the network dynamics evolving in the emulated network. Luckily, calculating the ideal connection strength is not very demanding so that the controlling host computer will be able run alongside the emulated network. Since we only have a limited weight resolution to work with, the needed homeostatic effect will be achieved by both modifying both rate and weight of the regular homeostatic spike train. This can be done by sending homeostatic input over several synaptic input channels, each set to different synaptic weight. Changing the weight then corresponds to switching to a different input channel. By adjusting the rate and weight the overall homeostatic influence can be adjusted smoothly, which is very important. Another potential problem whenever one is dealing with closed-loop setups is the delay it takes for the networks' spikes to reach the controlling host computer and vice versa. For homeostasis, this is a non-issue as the time scale on which it evolves is much slower than the immediate network dynamics. This means that even if there is delay, over time closed loop homeostasis will drive cause layer activities to the desired values.

Translating Homeostasis Weight to effective Bias

We want to translate between the biological weight w^{hom} calculated in each homeostatic synapse and the effective bias the corresponding cause layer neuron receives. At equilibrium, the effective conductance $\tilde{w}^{\text{hom}}$ after each spike from the source is the same:

$$\tilde{w}^{\text{hom}} \overset{!}{=} \tilde{w}^{\text{hom}} e^{-\frac{T_{\text{ISI}}}{\tau_{\text{syn}}}} + w^{\text{hom}} \tag{5.10}$$

$$\iff \quad \tilde{w}^{\text{hom}} = \frac{w^{\text{hom}}}{1 - e^{-\frac{T_{\text{ISI}}}{\tau_{\text{syn}}}}} \tag{5.11}$$

Here $T_{\text{ISI}} = {}^{1}/_{\nu_{\text{hom}}}$ is the average *inter-spike interval* (ISI) between two spikes from the background source firing with a rate of ν_{hom}. The average conductance exacted onto the post-

synaptic neuron is then:

$$\left\langle \tilde{w}^{\text{hom}} \right\rangle = \frac{1}{T_{\text{ISI}}} \int_0^{T_{\text{ISI}}} \mathrm{d}t \; \tilde{w}^{\text{hom}} \; e^{-\frac{t}{\tau_{\text{syn}}}} \tag{5.12}$$

$$= \frac{\tilde{w}^{\text{hom}}}{T_{\text{ISI}}} \; \tau_{\text{syn}} \left[1 - e^{-\frac{T_{\text{ISI}}}{\tau_{\text{syn}}}} \right] \tag{5.13}$$

$$= w^{\text{hom}} \; \nu_{\text{hom}} \tau_{\text{syn}} \tag{5.14}$$

Dividing by $f_{\text{theo}\to\text{bio}}$ (derived in Section 2.4.4), we obtain the final translation rule:

$$b^{\text{hom}} = \frac{\nu_{\text{hom}} \tau_{\text{syn}}}{f_{\text{theo}\to\text{bio}}} \; w^{\text{hom}} \tag{5.15}$$

5.3 Adjusting the synaptic Update Rule

Due to several factors, directly applying the original theoretical update rule Equation (2.163) for the synaptic weight V_{ik} between the input y_i and cause layer neuron k (representing binary *random variable* (RV) z_k) is infeasible. Instead, some adjustments have to be made which will be detailed in this section. Please note that while the discussion in this section focusses on weights in the theoretical domain, in actual simulations all weights are translated to their biological counterpart via a conversion factor (Sections 2.4.4 and 6.1.1).

We begin our discussion by reformulating the original update rule Equation (2.163):

$$\frac{\mathrm{d}V_{ik}}{\mathrm{d}t} = \eta \cdot z_k(t) \cdot \left(y_i(t) e^{-(V_{ik}+V_{i0})} - 1 \right) \tag{5.16}$$

$$= \eta \cdot z_k(t) \cdot \left(\frac{y_i(t)}{\lambda_{i0}} e^{-V_{ik}} - 1 \right) \tag{5.17}$$

$$= \eta \cdot z_k(t) \cdot \left(\frac{y_i(t)}{\nu_{i0} \cdot \tau_{\text{syn}}} e^{-V_{ik}} - 1 \right) \tag{5.18}$$

Here η denotes the learning rate, $y_i(t)$ is the input spike count in the short time interval $[t-\tau_{\text{syn}}, t]$ (corresponding to the eligibility trace of a rectangular STDP-curve), $V_{i0} = \log \lambda_{i0} = \log(\nu_{i0} \cdot \tau_{\text{syn}})$ the default hypothesis – also called *null cause* – and λ_{i0} the corresponding default hypothesis rate. ν_{i0} is the firing rate with which the generative model *assumes* the input is spiking when no cause layer neuron is active, i.e., no hidden latent variable z_k is explaining the input.

The first thing to notice is that the synaptic weight only gets updated whenever the post-synaptic neuron is active (corresponding to the binary variable $z_k = 1$). Otherwise the synapse is static. If the cause layer neuron is active, the weight is updated by the computing the ratio of how many spikes we did observe in the previous interval ($y_i(t)$) to how many spikes we would have expected to see on average if the input was generated by the default cause (λ_{i0}). This ratio is then weighted by the negative exponential of the current weight

and adjusted with a constant offset. We can get an intuition for the weight dynamics by computing the equilibrium distribution:

$$\left\langle \frac{\mathrm{d}V_{ik}}{\mathrm{d}t} \right\rangle_{p^*(\mathbf{y})} = \left\langle \frac{\mathrm{d}V_{ik}}{\mathrm{d}t} \right\rangle_{p(z_k=1|\mathbf{y})\,p^*(\mathbf{y})} \stackrel{!}{=} 0 \tag{5.19}$$

$$\implies \quad \left\langle \frac{y_i(t)}{\lambda_{i0}} e^{-V_{ik}} \right\rangle_{p(z_k=1|\mathbf{y})\,p^*(\mathbf{y})} = 1 \tag{5.20}$$

$$\hat{V}_{ik} := \langle V_{ik} \rangle_{p^*(\mathbf{y})} = \langle V_{ik} \rangle_{p(z_k=1|\mathbf{y})\,p^*(\mathbf{y})} = \log \frac{\langle y_i \rangle_{p(z_k=1|\mathbf{y})\,p^*(\mathbf{y})}}{\lambda_{i0}} := \log \frac{\langle y_i \rangle_k}{\lambda_{i0}} \tag{5.21}$$

Here we twice made use of the fact that the weight can only change when $z_k = 1$. By noting that

$$\nu_i(t) = \frac{y_i(t)}{\tau_{\text{syn}}} \tag{5.22}$$

is the input firing rate we find:

$$\hat{V}_{ik} = \log \frac{\langle \nu_i \rangle_{p(z_k=1|\mathbf{y})\,p^*(\mathbf{y})}}{\nu_{i0}} := \log \frac{\langle \nu_i \rangle_k}{\nu_{i0}} \tag{5.23}$$

Hence, we observe that the equilibrium weight $\hat{V}_{ik}$ encodes the log-odds of the input's firing rate in the active pattern versus the null cause. If the input is firing more than the default case, the weight will become positive whereas it will become negative if the input is firing less frequent in the active pattern. We can compute the learnt input firing rate $\hat{\nu}_{ik}$ via:

$$\hat{\nu}_{ik} := \langle \nu_i \rangle_k = \nu_{i0}\, e^{\hat{V}_{ik}} \tag{5.24}$$

Here, ν_{i0} is an externally set parameter; the network cannot infer the default hypothesis on its own. Also, ν_{i0} does not influence what input rate is learnt in the active pattern. It does however influence the absolute magnitude of the theoretical weights.

Since in hardware synapses can either be strictly positive or strictly negative, the null cause rate was most often chosen to be either the highest or the lowest possible firing rate of the input, respectively.

We will now discuss the challenges and needed adjustments when implementing SEM on neuromorphic hardware. Each of the next sections was implemented as its own synapse type and extensively tested in simulations.

5.3.1 Pair-based Updates

The first challenge arises when comparing the STDP curves of both the theoretical model and hardware (see Figure 5.2). The first difference is that in the ideal update rule $y_i(t)$ is a strict box-filtered signal of the input spike train. However, [Nessler et al., 2013] showed that a biologically more realistic exponentially decaying eligibility trace $y_i(t)$ is sufficient as well.

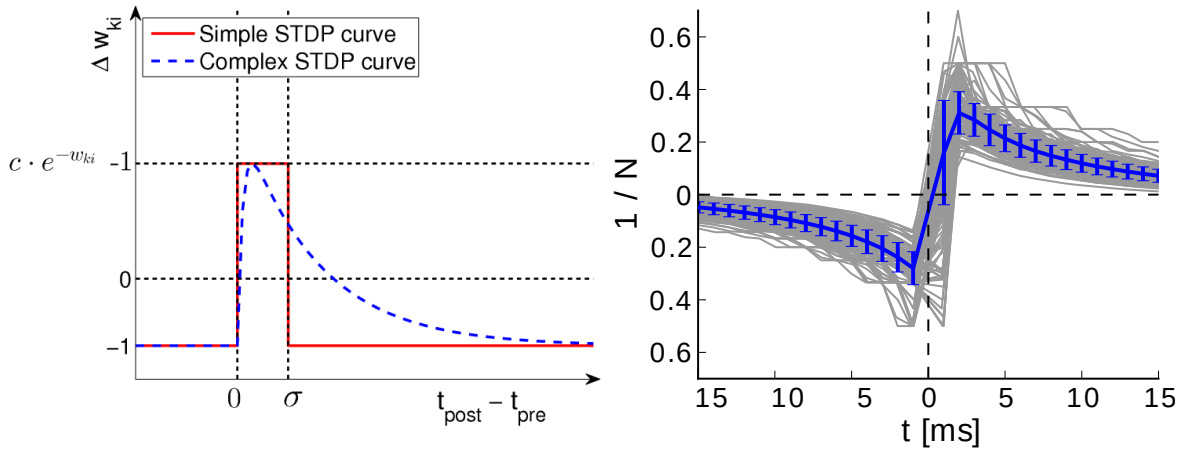

Figure 5.2: Comparison of STDP curves in theory (left) and hardware (right).

Left: Ideal STDP curve according to the theory. In red we have the ideal box-kernel resulting from the theoretical derivation. Potentiation of the synaptic weight w_{ki} (corresponding in notation to V_{ik} in this thesis) only occurs if the time difference between pre- and post-synaptic spike occurs within a narrow time window of length σ (corresponding to τ_{syn} in this thesis). In dashed-blue is a more complex version with a biologically more plausible alpha-kernel that the authors also verified to be working. Taken from: [Nessler et al., 2013].

Right: STDP (grey) curves measured for 252 neurons (in a single synapse row) on the Spikey chip – a predecessor to the HICANN – along with their mean and standard deviation (blue): Since the STDP window can only be measured indirectly, we measure how many spike pairs with a given time difference t it takes to trigger a causal ($t > 0$) or anti-causal ($t < 0$) weight update. The inverse $1/N$ then corresponds to the height of the STDP window at that time difference. Plot taken from [Pfeil et al., 2012].

Comparing the shape of both curves we notice some qualitative differences: The ideal STDP curve is constantly depressing except for a very short causal time window, whereas on hardware the STDP is strictly positive in the causal and strictly negative in the anti-causal part. Furthermore, on hardware the absolute magnitude of the STDP curve is always dependent on the time difference whereas in the ideal box-case we observe almost none (just a binary decision whether or not we are in the causal time window). This makes a direct translation of the update rule difficult.

The bigger problem is the constant shift to negative updates whenever we leave the causal time window of length τ_{syn}. This is a result from updating on every post-synaptic spike: If there are not input spikes, we have a constant linear decay (compare Equation (5.17)).

On hardware, however, we cannot access single pre- or post-synaptic spikes. The only information accessible to the STDP update controller are the causal and anti-causal capacitors that only get charged when there are spike pairs. For the causal capacitor, this corresponds to recording $(y_i z_k)(t)$, i.e., the state of the eligibility trace $y_i(t)$ at the time a post-synaptic spike is emitted. We therefore need to reformulate the original update rule Equation (5.17).

$$\frac{\mathrm{d}V_{ik}}{\mathrm{d}t} = \eta \cdot z_k(t) \cdot \left(y_i(t) \ \frac{1}{\lambda_{i0}} e^{-V_{ik}} - 1 \right) \tag{5.25}$$

$$= \eta \cdot \left(\underbrace{(y_i z_k)(t)}_{\substack{\text{measured in causal capacitance on} \\ \text{hardware}}} \ \frac{1}{\lambda_{i0}} e^{-V_{ik}} - z_k(t) \right) \tag{5.26}$$

Now we capture the first term. For the second term we take a look at the average update

$$\left\langle \frac{\mathrm{d}V_{ik}}{\mathrm{d}t} \right\rangle_{p^*(\mathbf{y})} = \eta \cdot \left(\langle y_i z_k \rangle_{p^*(\mathbf{y})} \ \frac{1}{\lambda_{i0}} e^{-V_{ik}} - \langle z_k \rangle_{p^*(\mathbf{y})} \right) \tag{5.27}$$

Since we employ homeostasis (see Section 5.2), the average activity of every cause layer neuron is kept fixed:

$$\langle z_k \rangle_{p^*(\mathbf{y})} = m_k \tag{5.28}$$

We can incorporate this into the update rule:

$$\frac{\mathrm{d}V_{ik}}{\mathrm{d}t} = \eta \cdot \left((y_i z_k)(t) \ \frac{1}{\lambda_{i0}} e^{-V_{ik}} - m_k \right) \tag{5.29}$$

Our update rule now operates on causal spike pairs only. We replaced the negative offset that was present for every post-synaptic spike in the original update rule with a constant linear decay that is independent of the post-synaptic neuron's spiking activity (since we assume it is fixed).

In figure Figure 5.3 the adjusted synapse dynamics are simulated in a fixed environment, showing that the correct pre-synaptic rates are indeed correctly inferred.

5.3.2 Nearest-Neighbor Spike Pairing

The next difference between the theoretical update rule and the hardware STDP update controller (see Section 3.2) is the spike pairing scheme. Ideally, we would like to have an all-to-all spike matching scheme so that the eligibility trace of the pre-synaptic activity is increased by one for every pre-synaptic spike and decays exponentially. For every post-synaptic spike the current value of the eligibility trace is read and recorded. This poses some problems: Since in

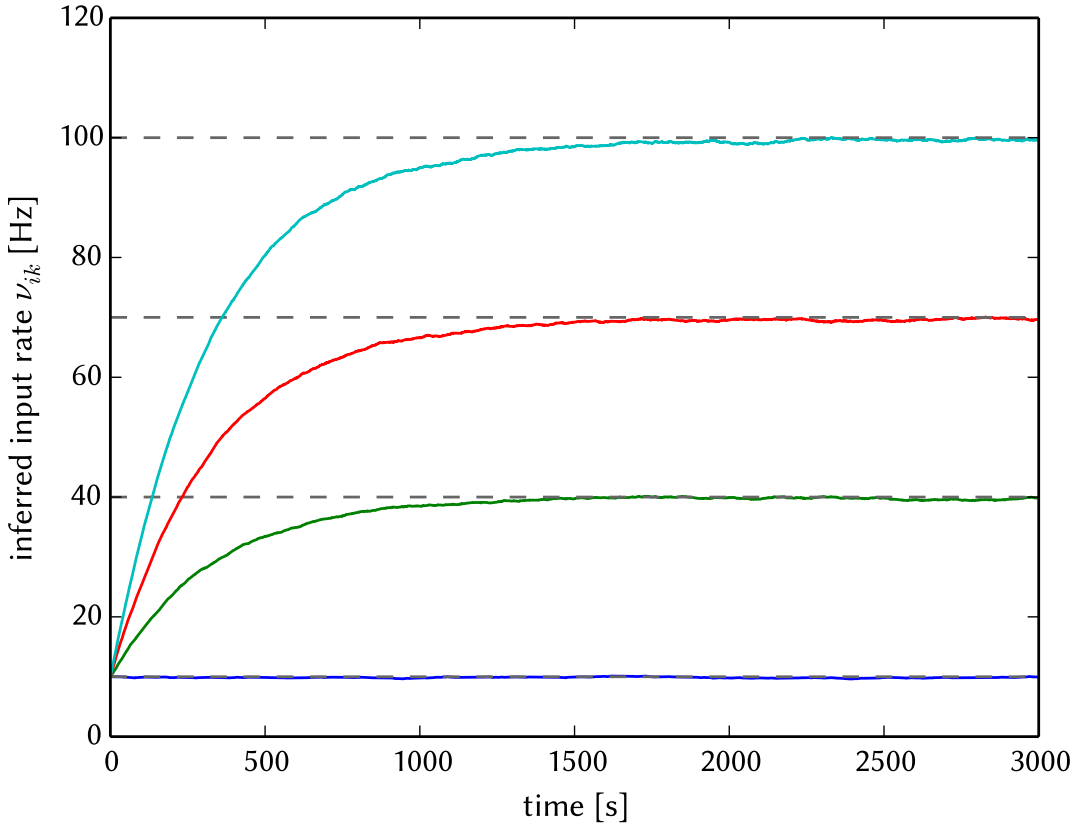

Figure 5.3: We show exemplary weight traces for the adjusted learning rule Equation (5.29). We connect four Poisson generators with rates of 10 Hz, 40 Hz, 70 Hz and 100 Hz (shown in dashed grey) to a selective parrot neuron, that is itself receiving regular spike input at $\nu_{\text{pre}} = 30\,\text{Hz}$, corresponding to almost the highest rate we would expect from a cause layer neuron with $\tau_{\text{refrac}} = 30\,\text{ms}$. Conversely, the average post synaptic activity is $\langle z_k \rangle = \nu_{\text{pre}}/\nu_{\text{net}}^{\text{max}} = 30\,\text{Hz}/(1/30\,\text{ms}) = 0.9$. Accordingly, τ_{syn} was set to 30 ms, while the learning rate η was set to $1 \cdot 10^{-4}$. The post-synaptic selective parrot neuron is set up to not re-emit spikes received via the investigated synapse so that the STDP-dynamics are completely independent from the pre- and post-synaptic activity (see Section 6.4.12 for details). The default hypothesis rate ν_{i0} is set to 10 Hz. The theoretical weight V_{ik} time course is recorded and translated back to the inferred input rate ν_{ik} according to Equation (5.24). We see that all synapses infer the correct rate. Please note that the value range for V_{ik} was not constricted in any way – especially not to be strictly positive.

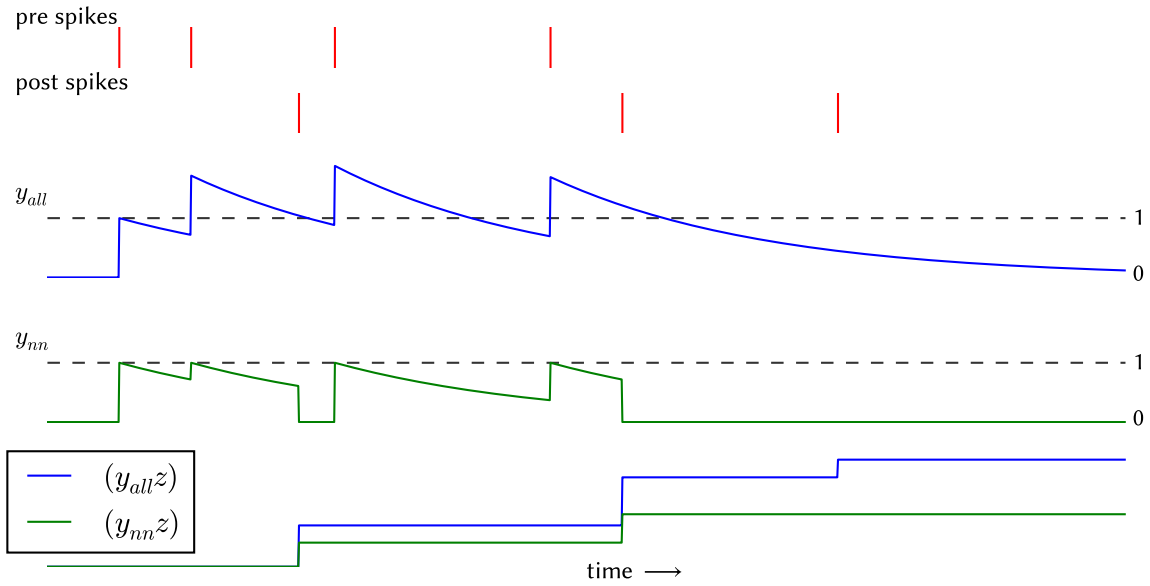

Figure 5.4: Schematic to illustrate the difference between all-to-all and nearest-neighbor spike pairing. We have a set of of pre- (first row) and post-synaptic spikes (second row). Depending on the pairing scheme, pre-synaptic spikes generate different eligibility traces: In the all-to-all scheme (third row) every spike increases the exponentially decaying eligibility trace by one, whereas in the nearest-neighbor pairing scheme (fourth row) every pre-synaptic spike sets the exponentially decaying eligibility back trace to one, effectively erasing the spike history. Furthermore, whenever a post-synaptic spike occurs we read out the pre-synaptic eligibility trace. It therefore gets set to zero (since in hardware the charge is applied to the accumulating capacitor) and subsequent post-synaptic spikes will add no further charge to the capacitor. Overall, the difference between both accumulated $(y_i z_k)$-traces can be significant (bottom row) and need to be accounted for.

hardware, the eligibility trace is modelled as actual charge, it cannot become arbitrarily large (in case of high pre-synaptic activity). Furthermore, upon a post-synaptic spike, this charge is directly applied to the accumulating capacitor. In order to preserve the current eligibility trace the charge would have to be read out and duplicated, requiring additional components in each synapse that would take up more size on the actual chip die. Therefore, the nearest-neighbor spike pairing scheme is a fair compromise. Here, each pre-synaptic spike simply sets the eligibility trace back to one (its maximum value), while a post-synaptic spike sets it to zero since the charge is applied to the accumulating capacitor. Both pairing schemes are compared in Figure 5.4. We see that the nearest-neighbor pairing scheme leads to lower eligibility traces.

We can predict what adjusted rates the synapses will learn in case of nearest-neighbor spike pairing. We first remark that in the equilibrium case (the network has already finished learning) ideally only one cause layer neuron is active for the entire duration we present a pattern. Said neuron will fire whenever possible with a firing rate close to $\nu_{\text{net}}^{\max}$, the ISI will therefore be close to τ_{refrac}. $y_i(t)$ – now restricted to the interval $[0, 1]$ – is only dependent on when the last pre-synaptic spike occurred. We can compute the probability of the last pre spike having

occurred at a time distance Δt by noting that the cumulative distribution of a spike occurring in the interval $[t - \Delta t, t]$ at a given firing rate ν is:

$$p(\text{spike in } [t-\Delta t, t]|\nu) = 1 - \text{Pois}\left(0 \mid \nu\Delta t\right) \tag{5.30}$$
$$= 1 - e^{-\nu\Delta t} \tag{5.31}$$

The probability density of a spike occurring at an exact time difference Δt is thus:

$$p_{\text{pre}}(\Delta t|\nu) := p(\text{spike at } (t-\Delta t)|\nu) = \frac{\mathrm{d}p(\text{spike in } [t-\Delta t, t]|\nu)}{\mathrm{d}\Delta t} \tag{5.32}$$
$$= \nu\, e^{-\nu\Delta t} \tag{5.33}$$

The average input eligibility trace that we observe is thus

$$\langle y_i \rangle_k^{\text{nn}} := \langle y_i \rangle_{p(z_k=1|\mathbf{y})\,p^*(\mathbf{y})}^{\text{nn}} = \int_0^{T_{\text{ISI}}} \mathrm{d}\Delta t\; e^{-\frac{\Delta t}{\tau_{\text{syn}}}}\; p_{\text{pre}}(\Delta t|\nu_{ik}) \tag{5.34}$$
$$= \int_0^{T_{\text{ISI}}} \mathrm{d}\Delta t\; \nu_{ik}\; e^{-\left(\frac{1}{\tau_{\text{syn}}}+\nu_{ik}\right)\Delta t} \tag{5.35}$$
$$=: \int_0^{T_{\text{ISI}}} \mathrm{d}\Delta t\; \nu_{ik}\; e^{-\tilde{\nu}_{ik}\Delta t} \tag{5.36}$$
$$= \left[-\frac{\nu_{ik}}{\tilde{\nu}_{ik}}\; e^{-\tilde{\nu}_{ik}\Delta t}\right]_0^{T_{\text{ISI}}} \tag{5.37}$$
$$= \frac{\nu_{ik}}{\tilde{\nu}_{ik}}\left[1 - e^{-\tilde{\nu}_{ik}T_{\text{ISI}}}\right] \tag{5.38}$$
$$= \frac{1}{1+\frac{1}{\tau_{\text{syn}}\nu_{ik}}}\left[1 - e^{-\left(1+\frac{1}{\tau_{\text{syn}}\nu_{ik}}\right)\nu_{ik}T_{\text{ISI}}}\right] \tag{5.39}$$

where $T_{\text{ISI}} \approx \tau_{\text{refrac}}$ is the average ISI of the active cause layer neuron and ν_{ik} is the input's actual spiking frequency. The equilibrium weight is still described by Equation (5.23), but its absolute value decreases due to

$$\frac{\langle y_i \rangle_k^{\text{nn}}}{\langle y_i \rangle_k} = \frac{1}{\tau_{\text{syn}}\nu_{ik}}\frac{1}{1+\frac{1}{\tau_{\text{syn}}\nu_{ik}}}\left[1 - e^{-\left(1+\frac{1}{\tau_{\text{syn}}\nu_{ik}}\right)\nu_{ik}T_{\text{ISI}}}\right] \tag{5.40}$$
$$= \underbrace{\frac{1}{1+\tau_{\text{syn}}\nu_{ik}}}_{<1}\underbrace{\left[1 - e^{-\left(1+\frac{1}{\tau_{\text{syn}}\nu_{ik}}\right)\nu_{ik}T_{\text{ISI}}}\right]}_{<1} < 1 \tag{5.41}$$

As a direct consequence, the inferred rates – when computed via Equation (5.24) by taking into account Equation (5.22) – also become smaller, but in a predictable way:

$$\hat{\nu}_{ik}^{\text{nn}} = \nu_{i0}\, e^{\hat{V}_{ik}} = \frac{1}{\tau_{\text{syn}}}\;\langle y \rangle^{\text{nn}} \tag{5.42}$$

Unfortunately, inferring the true input rate when all other quantities are known can only be done approximately as there is no analytical solution. If, however, $(\nu_{ik}T_{\text{ISI}}) \gg 1$, we can

approximate

$$\hat{\nu}_{ik} = \frac{1}{\tau_{\text{syn}}} \langle y_i \rangle_k^{\text{nn}} \approx \frac{1}{\tau_{\text{syn}}} \frac{1}{1 + \frac{1}{\tau_{\text{syn}} \nu_{ik}}} \tag{5.43}$$

$$\implies \qquad \nu_{ik} \approx \frac{1}{\frac{1}{\hat{\nu}_{ik}} - \tau_{\text{syn}}} \tag{5.44}$$

Being able to predict the average eligibility has another very practical advantage. The synapses on NM-PM1 are strictly excitatory or inhibitory. This means that if $\langle y_i \rangle_k > \lambda_{i0} > \langle y_i \rangle_k^{\text{nn}}$ an excitatory synapse will be stuck at weight zero. By employing Equation (5.39) we can adjust the null cause accordingly to the eligibility trace an input neuron firing with null cause rate would generate. Analogously to Equation (5.39) we have:

$$\lambda_{i0}^{\text{nn}} = \frac{1}{1 + \frac{1}{\tau_{\text{syn}} \nu_{i0}}} \left[1 - e^{-\left(1 + \frac{1}{\tau_{\text{syn}} \nu_{i0}}\right) \nu_{i0} T_{\text{ISI}}} \right] \tag{5.45}$$

Please note that T_{ISI} remains the same because we are still looking at the cause of one cause layer neuron being active during the whole duration of a pattern. The adjusted update rule for nearest-neighbor spike pairing is thus:

$$\frac{\mathrm{d}V_{ik}}{\mathrm{d}t} = \eta \cdot \left((y_i z_k)(t) \, \frac{1}{\lambda_{i0}^{\text{nn}}} e^{-V_{ik}} - m_k \right) \tag{5.46}$$

Exemplary weight traces are shown in Figure 5.5. As expected, the inferred rates are lower, but nevertheless validate our prediction Equation (5.42).

5.3.3 Accumulated Weight Updates

In NM-PM1, every synapse only stores local causal and anti-causal correlation information (see Section 3.2) – from which for SEM we only need the causal part. The STDP update controller (or the PPU, see Section 3.3) then updates each synapse row in turn with a certain update frequency. This means that we do not have access to the correlation information of single spike pairs $(y_i z_k)$, but rather accumulated information $\sum_l (y_i z_k)_l$ from an unknown number of spike pairs.

The actual weight update computed for a single spike pair ΔV_{ik} is

$$\Delta V_{ik} = \eta \cdot \left((y_i z_k) \, \frac{1}{\lambda_{i0}^{\text{nn}}} e^{-V_{ik}} - T_{\text{ISI}} \, m_k \right) \quad , \tag{5.47}$$

where we integrated the constant part over the course of one ISI. In the limit that a single weight update does not change the actual weight much ($V_{ik} + \Delta V_{ik} \approx V_{ik}$), we can keep V_{ik}

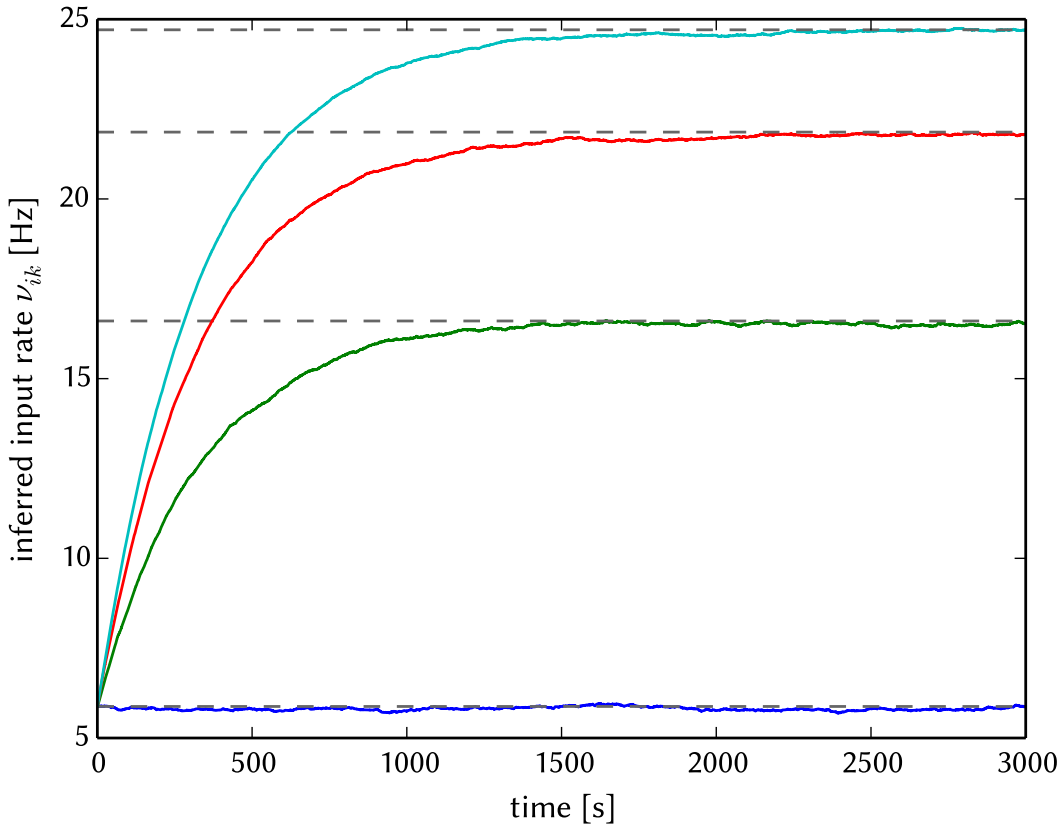

Figure 5.5: We show exemplary weight traces for the adjusted learning rule Equation (5.46) with nearest-neighbor spike pairing. The setup is the same as shown in Figure 5.3. As we can see, while the synapses now infer lower rates, they still match with our prediction according to Equation (5.42) (dashed in grey). While the set of possible weight values was still not restricted, it is important to note that due to our adjustment to the null cause (see Equation (5.45)) all but the blue weight trace remained strictly positive. The blue trace – due to resembling the null cause activity – oscillated around a weight value of zero and therefore sometimes became slightly negative.

constant for a small amount of updates and then update in bulk at regular and pre-determined update times t_{update}

$$\frac{\mathrm{d}V_{ik}}{\mathrm{d}t} = \sum_{t_{\text{update}}} \delta(t - t_{\text{update}}) \sum_{l} \Delta V_{ik,l} \tag{5.48}$$

Since we know the frequency with which updates are performed, we can integrate the update period τ_{update} directly into the update rule

$$\frac{\mathrm{d}V_{ik}}{\mathrm{d}t} = \eta \cdot \sum_{t_{\text{update}}} \delta(t - t_{\text{update}}) \left(\underbrace{\sum_{l} (y_i z_k)_l}_{\substack{\text{read from causal} \\ \text{capacitor in each update} \\ \text{as a whole}}} \frac{1}{\lambda_{i0}^{\text{nn}}} e^{-V_{ik}} - \tau_{\text{update}}\, m_k \right) \quad . \tag{5.49}$$

Exemplary weight traces are shown in Figure 5.6. The longer the update periods and the higher the input rate, the bigger the error when computing the new weight becomes compared to non-accumulated weight updates. Nevertheless, even for rather long update periods the correct input rates are inferred. This robustness for long update periods is important because due to the accelerated time scale of network emulations of $1 \cdot 10^3$–$1 \cdot 10^5$ synapses can only be updated every few seconds biological time, depending on the complexity[1] of the computed weight updates.

It is important to note that update rule Equation (5.49) can only be performed on the PPU as the actual value of the accumulated causal capacitor needs to be read out. The current generation STDP update controller is only able to compare the stored charge to adjustable thresholds that are fixed for the entire emulation run.

5.3.4 Limited Weight Resolution

A last aspect that has to be discussed when implementing SEM on neuromorphic hardware is the limited weight resolution. As discussed in Sections 3.2 and 3.3, the weight in each synapse is a digital 4–6 bit number that gets multiplied with an adjustable weight-factor ΔV_{hw} to form the actual synaptic weight. For the synapses, this means that the actual synaptic weight can be set to 16–64 equidistant values $[V_{ik}]_j$.

$$[V_{ik}]_j = j \cdot \Delta V_{\text{hw}} \tag{5.50}$$

This is a problem, since the weights usually change slowly over long periods of time, as we saw in Figures 5.3, 5.5 and 5.6. Changing the weight deterministically corresponds to increasing the learning rate. This leads to very unreliable learning and rather random network

[1] Since the PPU is essentially a regular CPU at heart, the time it needs per weight update depends on the number of operations it has to perform.

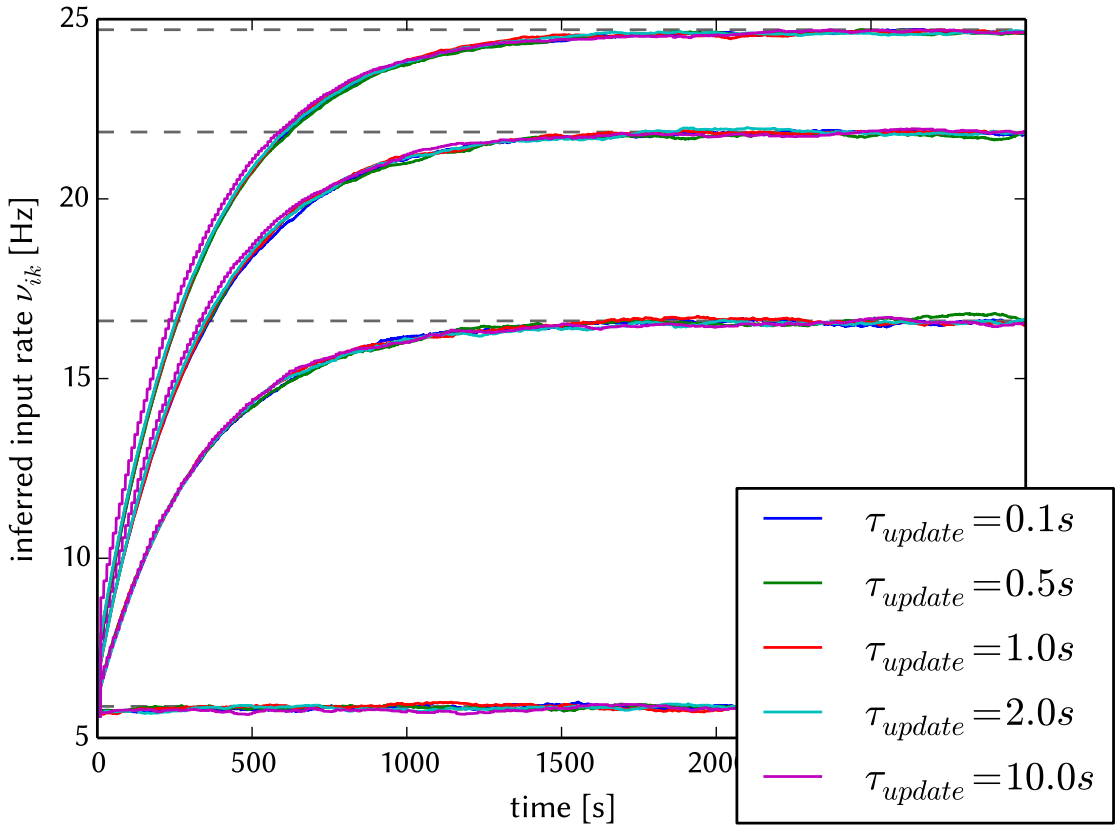

Figure 5.6: We show exemplary weight traces for accumulating weight updates according to Equation (5.49). The setup is the same as shown in Figure 5.3. Additionally, each synaptic connection is simulated several times with different update periods between 0.1–$10\,\mathrm{s}$ biological time. We observe that even for long update periods, the synapses learn to infer the correct rates. The longer the update period and the farther we are away from the final weight, the larger our single weight updates can become, as more spike pairs are weighted with a lower weight-factor $e^{-V_{ik}}$ as they would have been in case of immediate weight dynamics. Overall, accumulated correlation information does not affect synapse dynamics in a significant way.

dynamics. Since the PPU supports stochastic weight updates, by which we can virtually increase the available weight resolution. This means that while the actual weight value is constantly changing (due to stochastic updates), its time average will correspond to the target value which lies in between the realizable weights.

At every weight update the PPU performs, we compute the would-be update as is

$$\tilde{V}_{ik} = V_{ik} + \eta \cdot \left(\sum_l (y_i z_k)_l \ \frac{1}{\lambda_{i0}^{\text{nn}}} e^{-V_{ik}} - \tau_{\text{update}} \ m_k \right) \quad , \tag{5.51}$$

where V_{ik} is the current synaptic weight. We then find the next lower $\left\lfloor \tilde{V}_{ik} \right\rfloor$ and next higher $\left\lceil \tilde{V}_{ik} \right\rceil$ possible discrete weight so that $\left\lfloor \tilde{V}_{ik} \right\rfloor \leq \tilde{V}_{ik} \leq \left\lceil \tilde{V}_{ik} \right\rceil$. The new weight $\hat{V}_{ik}$ is then assigned in the following way:

$$p\left(\hat{V}_{ik} = \left\lceil \tilde{V}_{ik} \right\rceil \middle| \tilde{V}_{ik} \right) = \frac{\tilde{V}_{ik} - \left\lfloor \tilde{V}_{ik} \right\rfloor}{\Delta V_{\text{hw}}} \tag{5.52}$$

$$p\left(\hat{V}_{ik} = \left\lfloor \tilde{V}_{ik} \right\rfloor \middle| \tilde{V}_{ik} \right) = \frac{\left\lceil \tilde{V}_{ik} \right\rceil - \tilde{V}_{ik}}{\Delta V_{\text{hw}}} \tag{5.53}$$

Updating the weights in this way ensures that the long term averages of the weight trace will correspond to the original weight value we would have computed with arbitrary precision. Since weight updates are generally rather small, Equation (5.53) corresponds to a lot of updates not changing the weight value with only a select few succeeding. The average time after which the update succeeds is directly proportional to the proposed weight update value.

Exemplary weight traces with both 6-bit as well as 4-bit weight resolution can be seen in Figures 5.7 and 5.8. We see that even with limited weight resolutions we are able to infer the correct rates.

When performing stochastic weight updates with accumulated correlation information (see Section 5.3.3), we have the additional benefit that longer update periods correspond to more rapid weight change transitions. This serves as a form of "kick-start" for the learning. Long update periods are hence even less of a problem.

When computing with limited weight resolution, the actual weight values become important. By employing prediction developed in Section 5.3.2 we can predict in what range the actual weight values will lie and set the weight-factor ΔV_{hw} accordingly. We therefore compute both the null cause's average eligibility trace Equation (5.45) as well as the one generated by the maximum possible input rate Equation (5.39). The ratio between the two is the maximum (for excitatory synapses) or minimum (for inhibitory synapses) weight our synapses need to be able to achieve.

Lastly, when only dealing with 16–64 possible weight values, we can speed up weight calculation significantly by pre-computing and storing all "expensive" factors from Equation (5.51),

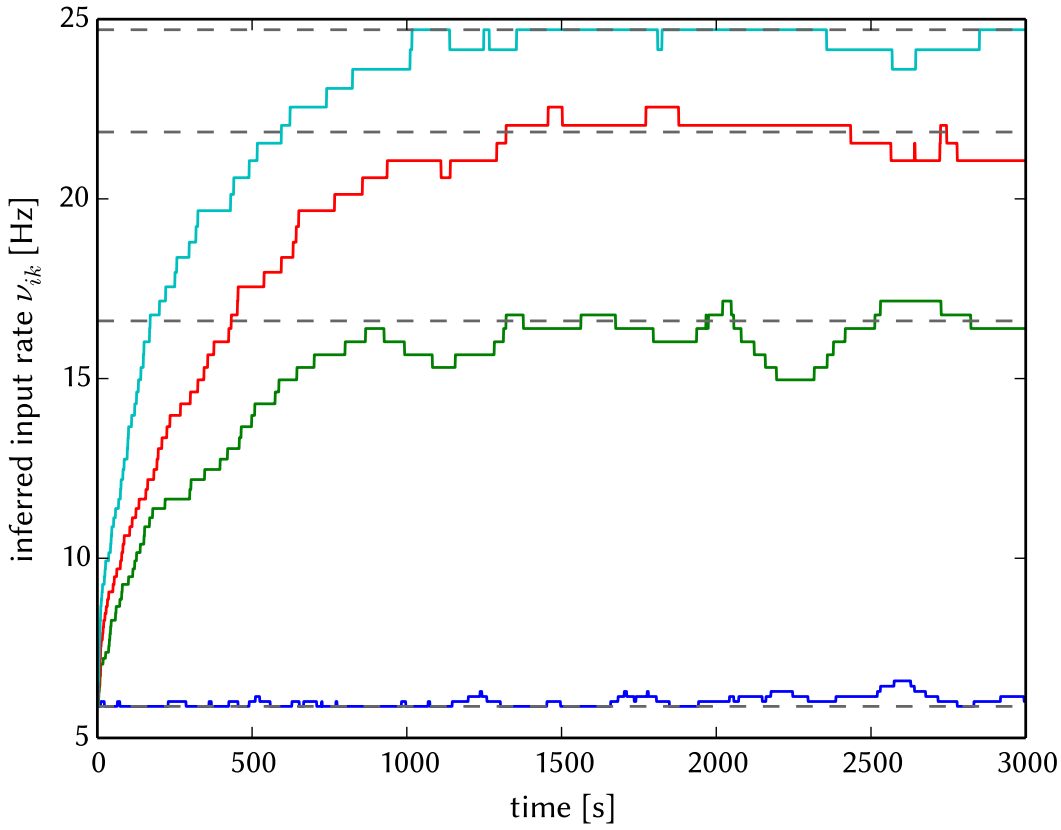

Figure 5.7: We show exemplary weight traces in the case of 6-bit weight resolution and stochastic weight updates according to Equation (5.53). The setup is the same as shown in Figure 5.6 but for each input rate we only show the case $\tau_{\text{update}} = 2\,\text{s}$. The theoretical weight range was set so that the maximum weight corresponds to the log-odds of the average eligibility trace generated by an input rate of 100 Hz and the null cause rate of 10 Hz. Please note that while the actual weights are equidistant, the inferred rates are not due to the exponential translation Equation (5.42). While we do see more weight fluctuation than in the previous cases (Figures 5.3, 5.5 and 5.6), we nevertheless observe clear oscillations around the correct input rates.

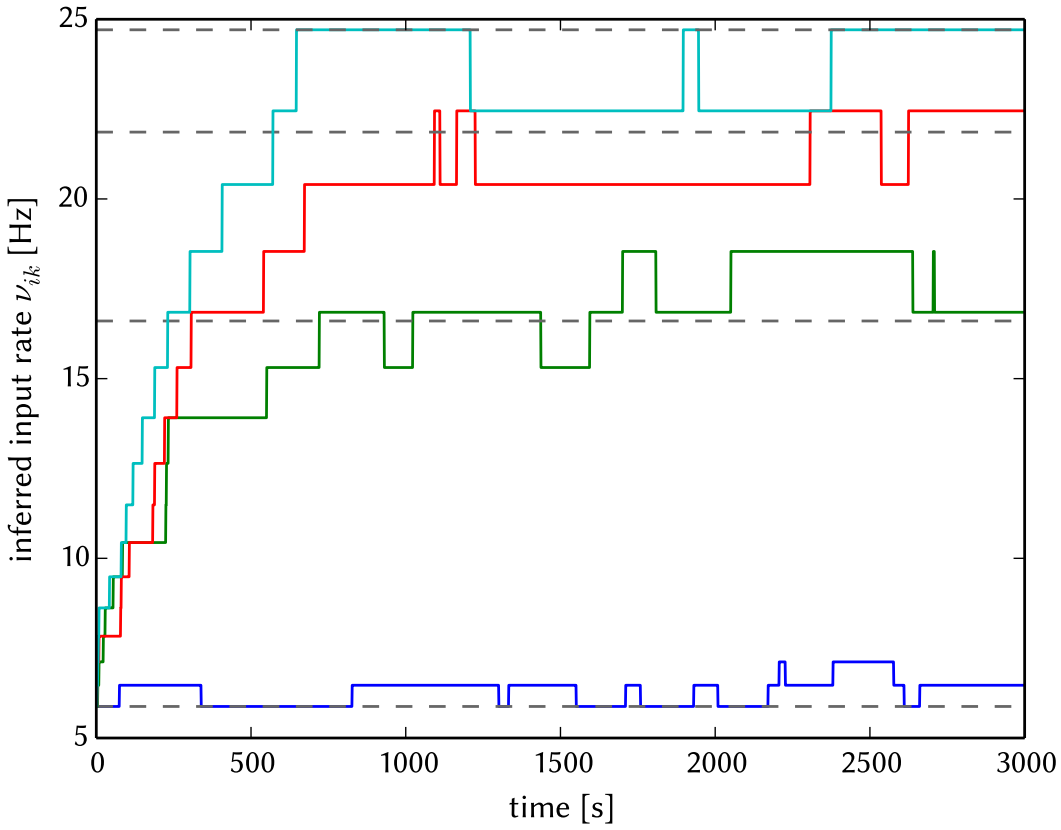

Figure 5.8: We show exemplary weight traces in the case of 4-bit weight resolution and stochastic weight updates according to Equation (5.53). The setup is the same as shown in Figure 5.6 but for each input rate we only show the case $\tau_{\text{update}} = 2\,\text{s}$. The theoretical weight range was set so that the maximum weight corresponds to the log-odds of the average eligibility trace generated by an input rate of 100 Hz and the null cause rate of 10 Hz. Please note that while the actual weights are equidistant, the inferred rates are not due to the exponential translation Equation (5.42). As in Figure 5.7, we observe clear oscillations around the correct input rates, even in the case of only 16 distinct weight values.

most importantly the exponential factors $\frac{1}{\lambda_{i0}} e^{-V_{ik}}$ but also the decay term. The processor then simply has to look up the corresponding factors in a *look up table* (LUT), rather than computing them anew every time. This simplifies the proposed weight calculations to the correlation information readout, one LUT-lookup (two if synapses differ in their target activity and the decay part has to be stored in its own LUT), two multiplications and one addition. The weight update computation time would then be dominated by the stochastic update implementation.

We denote the resulting network model as *neuromorphic spike-based expectation maximization* (NSEM), implementing all adjustments for neuromorphic hardware as well as spike-based homeostasis.

5.4 Simulation Results

5.4.1 Plot Structure

In this subsection, all plots of network dynamics (such as Figure 5.9) follow the same structure, described in the text below.

The top half of the plot depicts the "receptive fields", a colorplot of the inferred rates each cause layer neuron "believes" the input layer is firing with whenever it is active. They are computed from the learnt weights $\tilde{V}_{ik}$ after training via Equation (5.24).

The bottom half is split into three parts: On the left we have the spike response from the network in an additional "test" run. After training, the average over the 5 last snapshots of all weights in the network (input layer $\rightarrow$ cause layer, homeostasis $\rightarrow$ cause layer) is computed. We then initialize a new network in which all weights are kept static and present the same input as during training. For each presented input pattern we note which cause layer neuron spiked how often resulting in the depicted spike response. In order to further quantify how well a network has learnt a given input distribution, we compute either the approximate test accuracy – if the number of cause layer neurons is equal or greater than the number of distinct input labels – or the mutual information between active cause layer neuron and presented input pattern label – if the number of different input labels is greater than the number of cause layer neurons – and print it above the spike response count where applicable. Both are described in detail below.

In the middle part of the bottom half we find a histogram over the computed non-zero final weights in theoretical units (top) as well as the running average of each cause layer neurons firing rate (bottom).

Finally, the right part of the bottom row is comprised of each cause layer neuron's effective bias b_k^{hom} (top) as well as the average input weight seen by each cause layer neuron (bottom). Since we have a strictly excitatory as well as a strictly inhibitory synapse realizing homeostasis for each cause layer neuron, the effective bias is computed separately for each via Equation (5.15) and summed up.

All displayed time-courses are sampled at $25\,\mathrm{s}$ intervals. Since SEM-learning is inherently unsupervised, it is not pre-determined which cause layer neuron will specialize to code for which input pattern. For readability we therefore reorder the indexing of the cause layer neurons based on their activity during training.

Approximate Test Accuracy

The approximate test accuracy is useful when we have as many (or less) hidden causes (labels) in our input patterns as cause layer neurons in the network. Since SEM is inherently unsupervised, we need to identify which label each cause layer neuron codes for. Therefore, from the training step we perform a hard assignment for each cause layer neuron to the input label during which it spiked the most. During testing we then note which cause layer neuron is most active during the presentation of each pattern. We decide that a pattern was correctly classified if its label coincides with the one the most active cause layer neuron was assigned to. The ratio between correctly classified and total patterns presented is then defined as the accuracy. It is denoted as "approximate" because we only limited amount of patterns. Also, when learning the MNIST database (introduced below), we only use input images from the training dataset and not the separate testing dataset.

Mutual Information

When the number of different pattern labels vastly outnumbers the number of cause layer neurons in the network, each cause layer neuron starts to code for more than one label. Assigning it to only one input pattern as in the accuracy calculation would not capture this fact and result in a seemingly poor network performance. We need a way to quantify how well the limited set of cause layer neurons is classifying the input space. Mutual information between two RVs is a measure of their mutual dependence on each other. In this case, the two RVs are which label l we present as well as what cause layer neuron k was active ($p(k) := p(z_k = 1, \mathbf{z}_{\setminus k} = 0)$). It is defined as the *Kullback-Leibler divergence* ($\mathrm{D_{KL}}$) (see Section 2.2) between their joint probability distribution and the product of marginals:

$$\mathrm{D_{KL}}(p(l,k)||p(l)\,p(k)) = \sum_{l,k} p(l,k) \ln \frac{p(l,k)}{p(l)\;p(k)} \tag{5.54}$$

$$= \sum_{l,k} p(l)p(k|l) \ln \frac{p(k|l)}{p(k)} \tag{5.55}$$

$$= \langle \mathrm{D_{KL}}(p(k|l)||p(k)) \rangle_{p(l)} \tag{5.56}$$

Both $p(k)$ as well as $p(k|l)$ are estimated from the spike responses in the testing run. $p(l)$ is fixed by the way we choose to present our patterns during testing (e.g., uniformly, non-uniformly).

If a neuron k' shows no activity for a particular pattern l', we have $p(k'|l') = 0$. In this case we define $0 \cdot \ln 0 := \lim_{x \to 0} x \ln x = 0$.

As one expects, if presenting a pattern does not alter the activity of the cause layer neurons in any way we have $p(k) = p(k|l)$ and the mutual information is zero.

5.4.2 Regular Network Dynamics

In this section we explore the change in network dynamics as we adjust the STDP learning rule step-by-step to the constraints faced when trying to emulate SEM on the NM-PM1 as outlined in Section 5.3.

The general network setup is described in Section 5.1. In this particular implementation, our cause layer is formed by 6 stochastic cause layer LIF-neurons. The input layer consists of 17×17 individual Poisson sources. The input itself is a 3×17 pixels strip that is rotated between $0°$ and $180°$. We define 180 input labels, one for each degree of rotation (0–$179°$). The base image – a rotation of $0°$ – corresponds to a horizontal stripe of width 3 at half height. Pixels on the stripe are set to 1.0 while all others remain at 0.0. All rotations are computed from this image, using the `ndimage.rotate`-function from the SciPy-library [Jones et al., 2001]. The resulting interpolated pixel values are translated into Poisson firing rates between $10\,\mathrm{Hz}$ (pixel value 0.0) and $70\,\mathrm{Hz}$ (pixel value 1.0). To make sure each input label has the same absolute strength, we re-normalize all pixel values again so that the sum of all pixel values is the same as in the base image (rotation of $0°$). Therefore, we have one pattern per input label.

The dynamics of the network operate at different time scales. On the fastest scale is the presented input. Every $0.5\,\mathrm{s}$ we present a randomly chosen new pattern for $0.5\,\mathrm{s}$ and there is no pause[2]. Each unit in the input layer then generates a Poisson spike train of the corresponding input rate.

On the next, slower time scale is the homeostasis takes place. It is implemented using update rule Equation (5.9) with a periodic background source (see Section 6.4.11) firing at $2000\,\mathrm{Hz}$ with a learning rate of $\eta_b = 1.0 \cdot 10^{-3}$. Each cause layer neuron is set to fire with the same activity.

The actual learning takes place on the slowest time scale. The weights V_{ik} between the input and cause layer are set to be strictly excitatory and evolve according to a variety of update rules (discussed in Section 5.3). The learning rate is kept constant at $\eta = 1.0 \cdot 10^{-4}$ during the whole learning process.

The neuron parameters for the cause layer were chosen in a generic fashion as we are have to calibrate our neurons anyway in order to translate between the realm of theoretical Boltzmann-weights and biological synaptic conductances (see Sections 2.4.4 and 6.1.1). The reversal potentials were chosen symmetrically – at $-100\,\mathrm{mV}$ and $0\,\mathrm{mV}$ respectively – around the spike threshold, coinciding with the resting potential at $V_{\text{spike}} = E_L = -50\,\mathrm{mV}$. The membrane capacitance $C_m = 0.2\,\mathrm{nF}$ and time constant $\tau_m = 1.0\,\mathrm{ms}$ to facilitate fast membrane dynamics at comparatively low background input rates. The Poisson background input to make the cause layer stochastic is set to fire $2000\,\mathrm{Hz}$ with a weight of $0.001\,\mu\mathrm{S}$. The corresponding weight translation factor yielded from calibration is $f_{\text{theo}\to\text{bio}} = 1.134\,\mu\mathrm{S}$ for $\tau_{\text{syn}} = 30\,\mathrm{ms}$ ($f_{\text{theo}\to\text{bio}} = 0.456\,\mu\mathrm{S}$ for simulations later on with $\tau_{\text{syn}} = 10\,\mathrm{ms}$ respectively).

[2] Various simulations (not shown) demonstrated no real improvement or decline in learning quality for pauses between patterns. They appear to only prolong the effective learning time as the total time a pattern is presented to the cause layer is reduced.

In order to minimize the time it takes after a refractory period for the membrane potential to converge to the free membrane potential, we set the reset potential close to the threshold ($V_{\text{reset}} - V_{\text{spike}} = -0.001\,\text{mV}$). Therefore, if the free membrane potential is above the spiking-threshold and the neuron can fire again almost immediately. We choose $\tau_{\text{syn}} = \tau_{\text{refrac}} = 30\,\text{ms}$ in accordance with the exemplary weight traces shown in Section 5.3. The choice of τ_{syn} influences both speed and robustness of learning. The larger τ_{syn}, the longer we integrate information from the input layer and can be more confident in our weight updates, but the more simulation time it takes per weight update as the cause layer can only spike with $\nu_{\text{net}}^{\max} = {}^{1}/_{\tau_{\text{syn}}}$.

All simulations are run using the same random seed in order to rule out fluctuations in the input as source for differences in network dynamics. All biases and weights are set to zero at the beginning of learning. Each network is simulated for $10\,000\,\text{s}$.

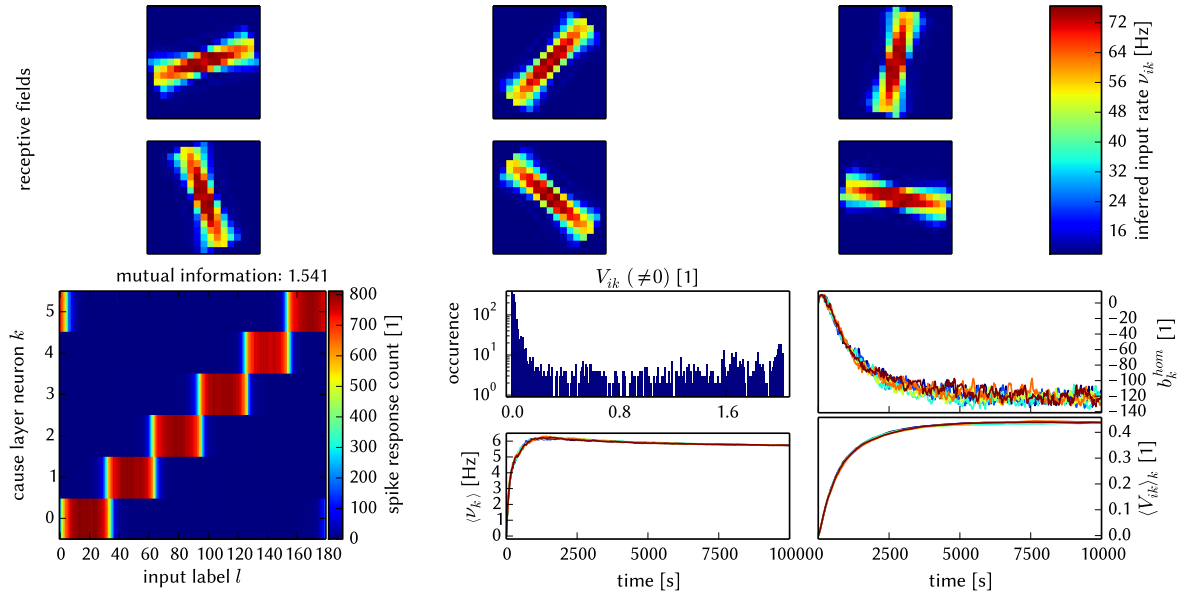

Figure 5.9: SEM learning with stochastic LIF-neurons and original update rule (Equation (2.163)), performing weight updates with rectangular eligibility traces. The activity of cause layer neurons is moderated via spike-based homeostasis (see Equation (5.9)).
The full parameters can be found in Appendix A.1.2 while the plot structure is explained in Section 5.4.1. Please see the text for details.

Ideal Updates with rectangular Eligibility Traces

The first implementation is done using the ideal theoretical update rule Equation (2.163) with rectangular eligibility traces. Here, the input variable y_i always corresponds to the number of pre-synaptic spikes in a time window of length τ_{syn}. Please note that the cause layer neurons still operate with exponentially decaying synaptic conductances.

The network dynamics are shown in Figure 5.9. We see that each cause layer neuron codes for a different set of neighboring orientations. The input space is equally distributed among the cause layer neurons. Despite the constant learning rate, learning stabilizes at the expected

equilibrium points. The receptive fields correspond to the input rate of each input unit averaged over all patterns the cause layer neuron codes for. Input neurons closer to the middle, i.e., active for all relevant orientations, have the correctly inferred rate of $70\,\mathrm{Hz}$. Input units lying further outside are on average not as active whenever the cause layer neuron is, thus their inferred rate is lower.

Furthermore, we see that the spike-based homeostasis manages to keep the activity of all cause layer neurons at the their target value of $\nu_{\text{net}}^{\max}/6 = 1/(30\,\mathrm{ms}\cdot 6) \approx 5.5\,\mathrm{Hz}$. In the beginning (few $100\,\mathrm{s}$), the homoestatic influence gets slightly excitatory in order to encourage the cause layer to spike as the initial biases are set to zero. As the weights V_{ik} evolve the effective biases become strictly negative in order to counteract the increasing input strength.

We do see, however, that during early learning ($\sim 1250\,\mathrm{s}$) the rates slightly exceed their target activities. We are still early in the learning and so the receptive fields are not as pronounced yet and highly overlapping. Furthermore, as the weights increase rapidly in this phase of learning due to the exponential dependence on the current weight in the weight update, the homeostatic adjustment is lagging slightly behind. This means that for any given input pattern several cause layer neurons get stimulated sufficiently to spike (i.e., the synaptic input from the input layer is stronger than the inhibition via homeostasis). If several of them spike within the delay-time window ($\sim 1.3\,\mathrm{ms}$), the inhibitory signal from the inhibitory population will arrive too late. Therefore more than one cause layer neurons can learn from the same input. However, as the effective bias "catches up" to the synaptic input in strength, the cause layer is moderated more effectively, lowering the chance of two neurons being sufficiently stimulated to spike within the delay-time window. Overall, this is a result of the fixed homeostasis learning rate η_b. The quicker we can adjust the homeostatic weights in case of rapid weight changes, the larger the variance of the effective bias when the learning has finished and vice versa. During learning, the are small differences in activity occurring between cause layer neurons during learning. If the homeostasis learning rate becomes too high, these differences are balanced out immediately, thereby driving every unit in the cause layer to learn an averaged superposition of all input patterns.

During testing we see that each cause layer neuron responds to a particular set of neighboring orientations only. When transitioning from one receptive field to the next, we see that the spike responses shift accordingly: For example, when the orientation lies half-way between the preferred orientations of two neurons, both are active roughly for the same amount of time. Overall, we see that the network performs a form of dimensionality reduction by encoding the state of 289 neurons in the activity of 6.

Ideal Updates with exponential Eligibility Traces

Next we exchange the rectangular eligibility traces for exponential ones, so that every spike increases the corresponding input variable y_i by one. y_i decays with time constant τ_{syn}. See the third row in Figure 5.4 for an illustration. The resulting network dynamics are shown in Figure 5.10.

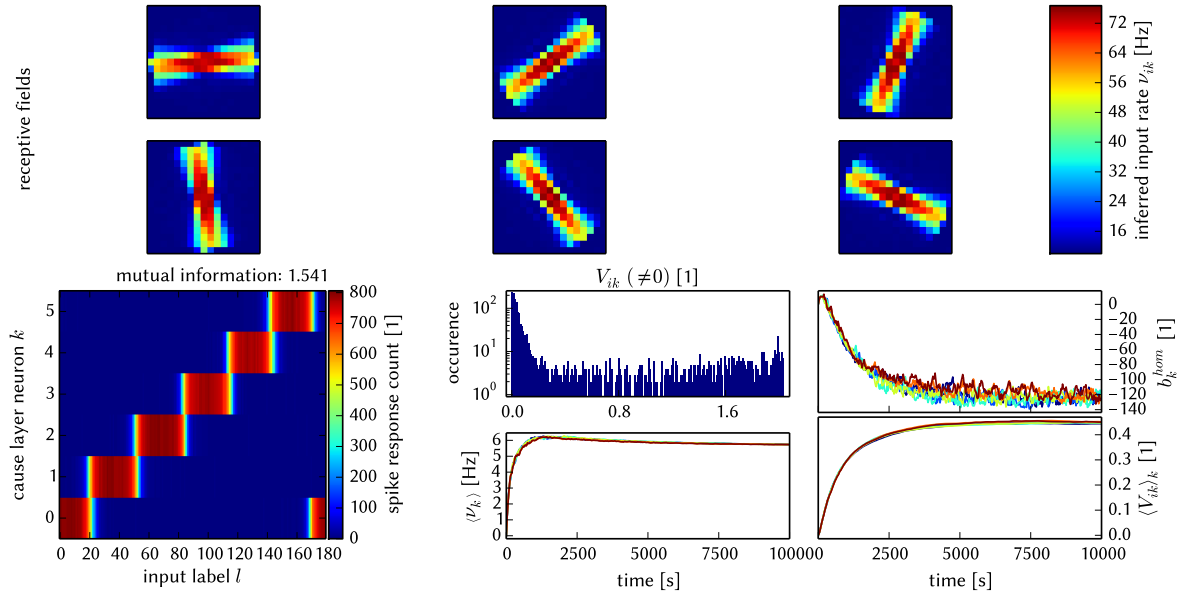

Figure 5.10: SEM learning with stochastic LIF-neurons and original update rule (Equation (2.163)), performing weight updates with exponential eligibility traces (see third row in Figure 5.4). The activity of cause layer neurons is moderated via spike-based homeostasis (see Equation (5.9)).
The full parameters can be found in Appendix A.1.2 while the plot structure is explained in Section 5.4.1. Please see the text for details.

As the eligibility traces now correspond even closer to the exponential-shaped post-synaptic conductances, we see no real change in network dynamics, validating [Nessler et al., 2013].

Pair-based Correlation Measurements

We then introduce pair-based correlation measurements (as in Equation (5.29)). Now the synapse is linearly decaying at a fixed rate, corresponding to the target activity of the corresponding cause layer neuron. The eligibility trace stays exponential (as for all following experiments). The network dynamics are shown in Figure 5.11.

On first inspection we see hardly any difference to the previous two models. The receptive fields are the same, the correct input rates are inferred, the spike response in the subsequent test-run shows that each cause layer neuron is again accounting for the same amount of input patterns, and the distribution of weights also shows no significant differences.

Where we do see a qualitative difference though is the running-mean of the spike activity $\langle \nu_k \rangle$ and the average input weight to each neuron $\langle V_i k \rangle_k$. Due to the fact that the synaptic weight only decays linearly based on the *average* target activity of the post-synaptic neuron, short bursts in activity can increase the weight far more than in the ideal case since the negative update part in the synaptic update is not accounted for in every update. As we see in the average weight plot, this leads to a slight overshoot in learning: While in the ideal case the weights are always increasing on average, we find that in the nearest-neighbor case they

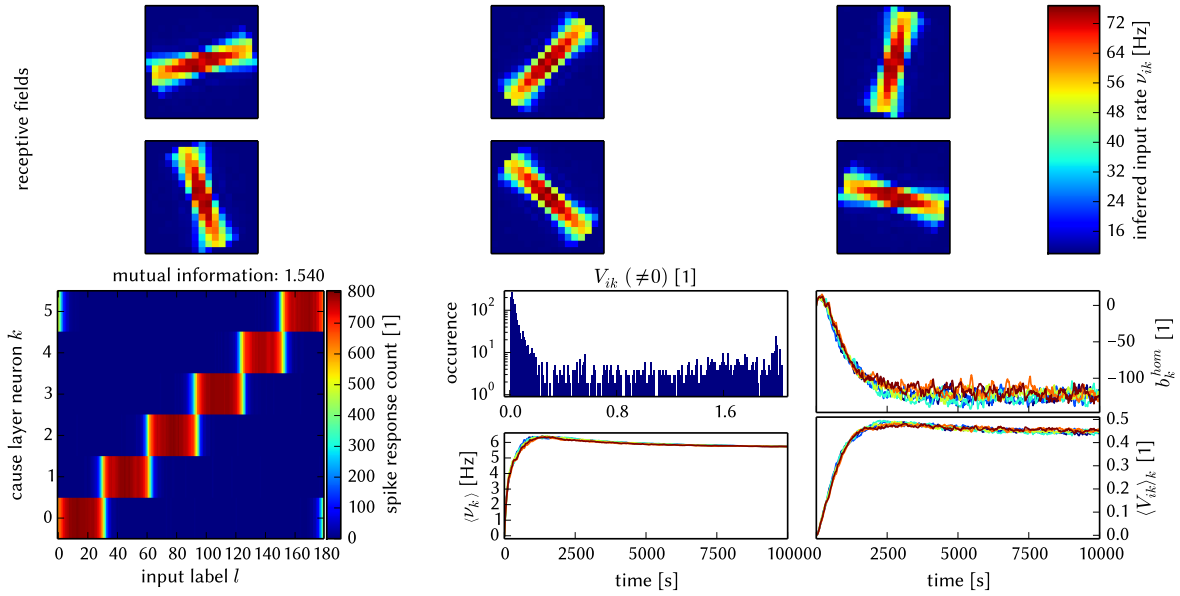

Figure 5.11: SEM learning with stochastic LIF-neurons and a strictly spike pair-based update rule (see Equation (5.29)). The activity of cause layer neurons is moderated via spike-based homeostasis (see Equation (5.9)).
The full parameters can be found in Appendix A.1.2 while the plot structure is explained in Section 5.4.1. Please see the text for details.

reach their maximum after the initial learning phase and then slowly decrease until reaching the equilibrium point. This is due to the fact that the cause layer activity is higher than the target during early learning which is not accounted for in the weight updates. Only when the homeostasis "catches up" we see the approximation towards the equilibrium points.

Nearest-Neighbor Spike Pairing

Next we account for nearest-neighbor spike pairing in the update rule (see Equation (5.46)). For this reason, we adjust the null cause rate ν_{i0} to the average eligibility trace it would generate under a nearest-neighbor spike pairing scheme (see Equation (5.45)). The network dynamics are shown in Figure 5.12.

Here we see some major changes in the network compared to the implementations described above. While qualitatively, the receptive fields seemingly remain unchanged, the corresponding absolute weights are smaller, as evidenced by the weight histogram. This is acceptable, as we had to adjust the null cause for nearest-neighbor spike pairing. If we did not make the adjustment, all synapses with an afferent rate of 20 Hz or below would stay at zero (or become negative, which we do not permit as it is impossible in hardware). While we do argue in Section 5.4.3 that this filtering of low input background can be beneficial when learning strongly overlapping patterns such as MNIST, here we want to maintain as much of the original network dynamics as possible. Ergo, we want all input patterns that fire more strongly than the null cause rate to result in a non-zero input weight. Furthermore, not adjusting the

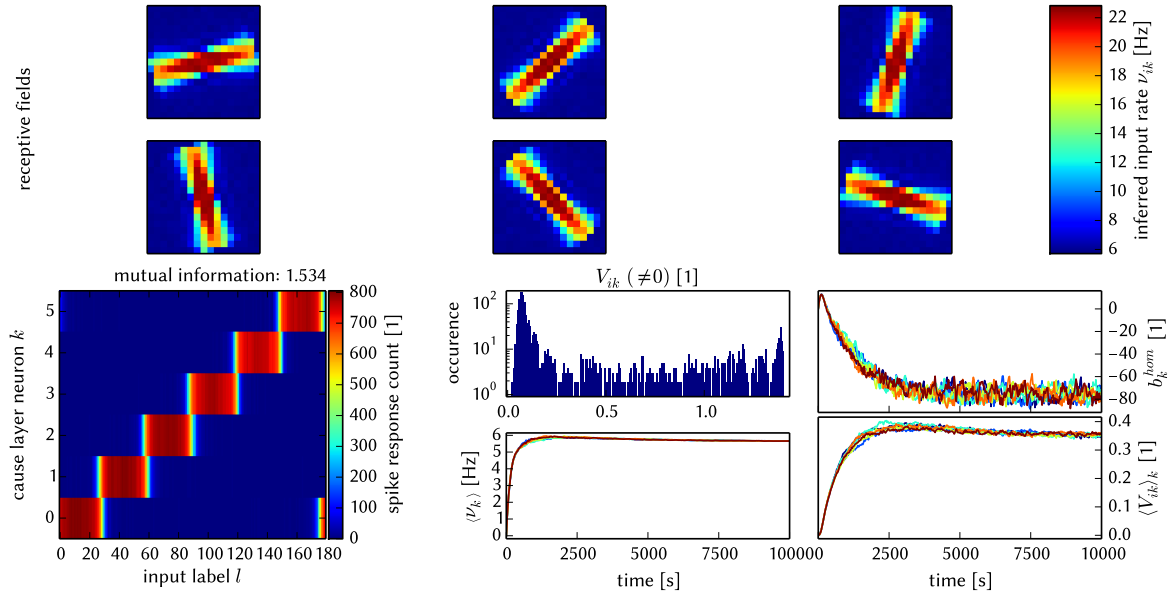

Figure 5.12: SEM learning with stochastic LIF-neurons and nearest-neighbor spike pairing update rule (see Equation (5.46) as well as the fourth row in Figure 5.4). The activity of cause layer neurons is moderated via spike-based homeostasis (see Equation (5.9)).
The full parameters can be found in Appendix A.1.2 while the plot structure is explained in Section 5.4.1. Please see the text for details.

null cause would lead to overall even smaller absolute weights as the weights encode the log-odds between actually observed input rate and null cause rate.

Since average weight for each input neuron is roughly $20\,\%$ lower than for the full eligibility trace, this is also reflected in the effective biases. Conversely, the inferred rates are significantly lower but correspond to what we expect the network to infer (see Equation (5.42) and Figure 5.5).

Nevertheless, from a functional point of view the network dynamics remain the same, as shown by the spike response count as well as the mutual information (bottom left corner in Figure 5.12). Both change only minimally. The weight overshoot that we first experienced when introducing the pair-based update rule also occurs here, but is less pronounced due to the overall lower weights.

Accumulated Weight Updates

Now we introduce accumulated weight updates (see Equation (5.49)). The exemplary weight trace (see Figure 5.6) as well as parameter sweeps Figure 5.16 do not indicate a strong dependency on the update period. τ_{update} was therefore chosen to be $2\,\text{s}$. This is still more than twice of what we actually estimate the update period in actual emulations to be for this kind of weight update[3]. The network dynamics are shown in Figure 5.13. As the PPU updates synaptic weights in a row-wise fashion, we cannot assume all weights to be updated at the

[3]Personal correspondence with Simon Friedmann, designer of the PPU.

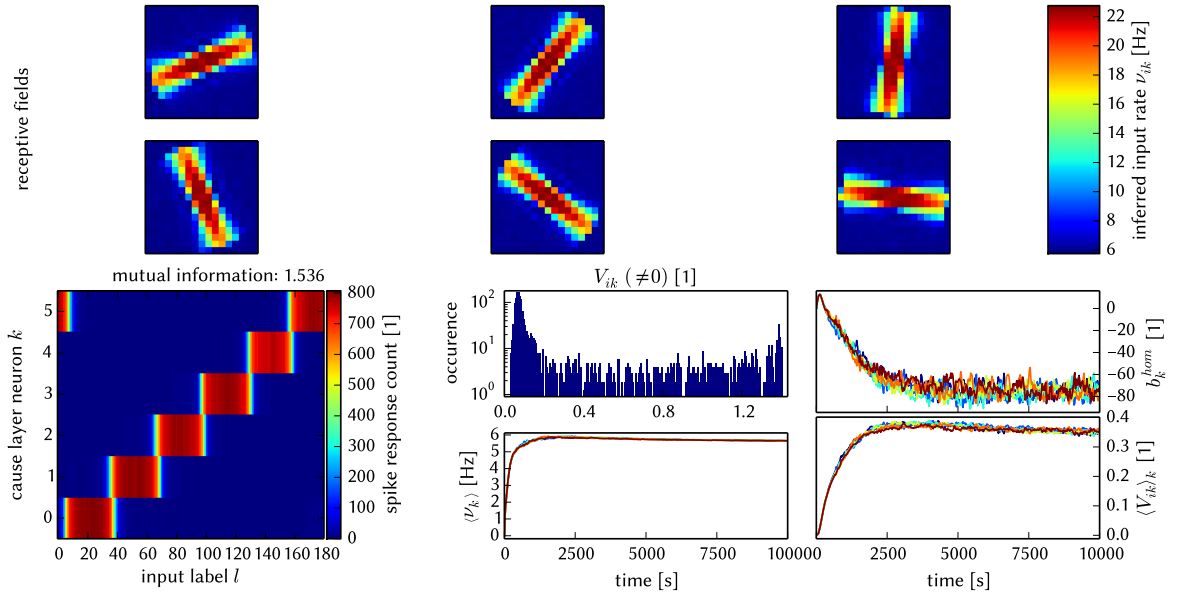

Figure 5.13: SEM learning with stochastic LIF-neurons and accumulating update rule (see Equation (5.49)). The activity of cause layer neurons is moderated via spike-based homeostasis (see Equation (5.9)).
The full parameters can be found in Appendix A.1.2 while the plot structure is explained in Section 5.4.1. Please see the text for details.

same time. To see whether this affects network dynamics, we choose to update the weights in the worst (i.e., most asymmetric) way possible: During the $2\,\mathrm{s}$ update period, we in turn update *all* weights to each cause layer neuron in a round-robin fashion. The updates occur at equidistant points in time so that the time distance between any two cause layer neurons' weight updates is $0.333\,\mathrm{s}$ at the least and $1.666\,\mathrm{s}$ at the most. This is potentially harmful to the network dynamics as already updated neurons might have an "advantage" due to overall slightly higher input weights.

We observe that accumulated weight updates do not alter the network dynamics in any significant way when directly compared to the previous simulation implementing nearest-neighbor spike pairing. The possible disturbance of weight updates for each neuron occurring at different times is mitigated sufficiently by homeostasis. This is in accordance with a larger sweep of simulations found in Figure 5.16.

Limited Weight Resolution

The final constraint we have to adjust our update rule to when transferring the original model to neuromorphic hardware is the limited weight resolution. It is achieved by using the PPU's ability to perform stochastic weight updates (see Equation (5.52)). Since we only have a limited number of weights, we need to set their range accordingly. Here it is essential to use Equation (5.45) as well as Equation (5.39) in order to compute the ideal Boltzmann weight for the highest input rate we expect to see. Since we want to verify that the synapses actually infer the correct rate (and not just assume the highest possible weight), we set the actual

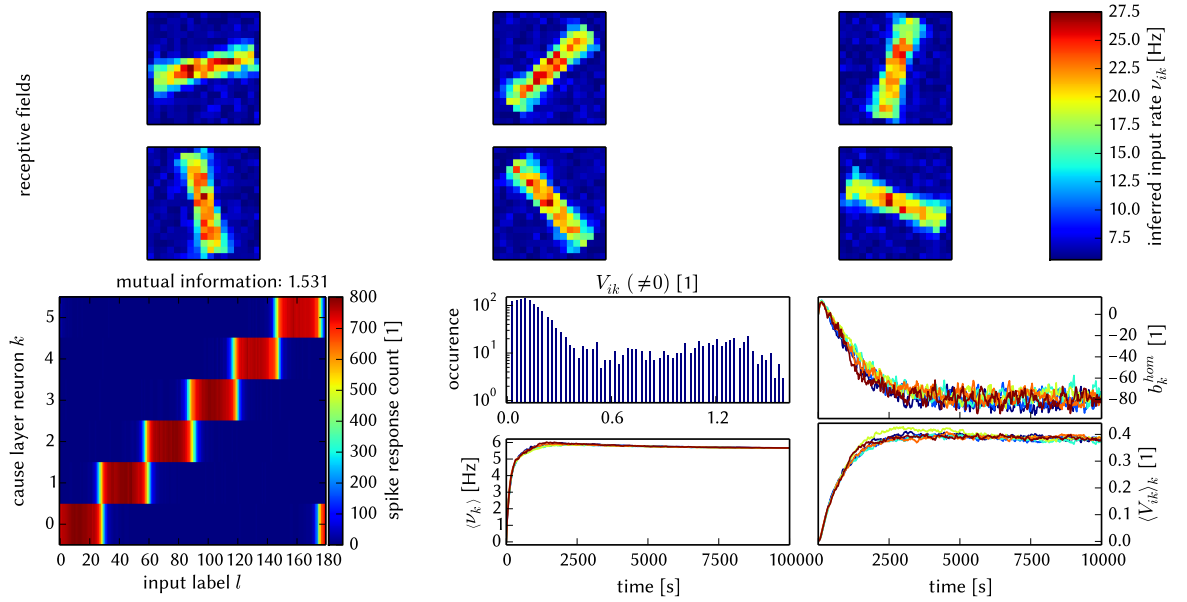

Figure 5.14: NSEM learning with stochastic LIF-neurons and stochastic weight updates (see Equation (5.52)) with 6-bit weight resolution. The activity of cause layer neurons is moderated via spike-based homeostasis (see Equation (5.9)).
The full parameters can be found in Appendix A.1.2 while the plot structure is explained in Section 5.4.1. Please see the text for details.

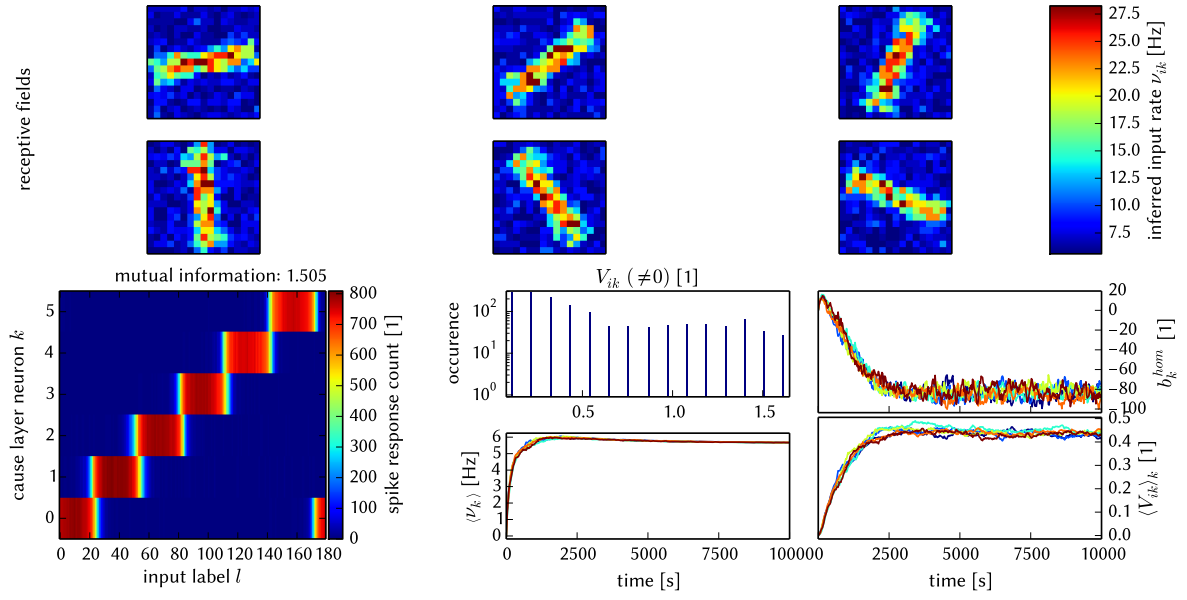

Figure 5.15: NSEM learning with stochastic LIF-neurons and stochastic weight updates (see Equation (5.52)) with 4-bit weight resolution. The activity of cause layer neurons is moderated via spike-based homeostasis (see Equation (5.9)).
The full parameters can be found in Appendix A.1.2 while the plot structure is explained in Section 5.4.1. Please see the text for details.

maximum weight to be $20\,\%$ larger. The resulting network dynamics can be seen in both Figure 5.14 for 6-bit weight resolution as well as in Figure 5.15 for 4-bit weight resolution.

We observe that the receptive fields are now coarser due to the fact that the ideal weights the synapses would naturally assume lie in between the weight values they are able to assume. Hence the weight value will only coincide with the ideal one when averaged over time. Any snapshot of the weights, such as the one shown in Figures 5.14 and 5.15, will differ due to random fluctuations. Nevertheless, we can still recognize the receptive fields.

Furthermore, we see that the weight histograms resemble the non-discretized cases in shape for larger weights (for the 6-bit case more so than the 4-bit case). The peak at small weights is missing as only few of the weights close to zero are actually able to jump to the realtively large possible values. The full weight range range is being utilized with only relatively few weights at maximum, indicating that our predicted target weights are indeed correct. Due to utilization of stochastic weight updates, we find that 64 distinct weight values is indeed enough to represent the receptive fields. For 16 distinct weight values we observe the biggest drop in mutual information compared to all performed adjustments. Nevertheless the network remains functionally intact.

Conclusion

From the plots Figures 5.9 to 5.15 we can safely conclude that NSEM, an implementation of NSEM on neuromorphic platforms, is feasible. Spike-based homeostasis is able to keep each cause layer neuron at its target activity. All adjustments we had to make to the update rules do not disturb network dynamics in any significant way. The only fundamental change from a functional point of view remains the reduced rates that are inferred during learning, as we can only access the last pre-synaptic spike in each synapse.

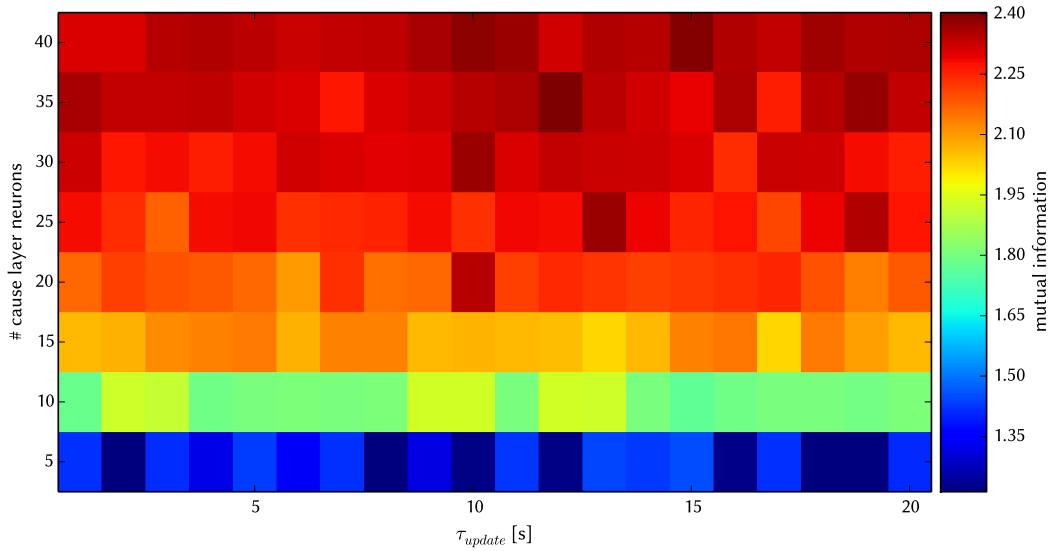

Figure 5.16: Mutual information for varying update periods for accumulation. The network setup corresponds to Figure 5.14, that is the network is learning orientations with fully adjusted update rules at 6-bit weight resolution. We vary the number of cause layer neurons as well as the accumulation time window τ_{update}. As discussed above, weight updates for each cause layer neuron occur one at a time after equidistant steps. The weight updates thus occur asymmetric which could disturb the network's performance. For each network setup we compute the mutual information. Please note that, as the number of neurons in the cause layer increases, so does the mutual information as the network can differentiate between a greater number of orientations. We find that irrespective of the update period, the mutual information remains constant, apart from some acceptable fluctuations. We can therefore conclude that – as long as we can measure and store the causal spike correlations well enough over long periods of time – we are not dependent on frequent weight updates.

5.4.3 Null cause as Contrast-Enhancer in receptive Fields

Learning MNIST

In this section, we shift our attention primarily towards learning MNIST [LeCun and Cortes, 1998], a commonly used database of handwritten digits. For each digit it contains 6000 representations. Each representation is 28×28 pixels in size. We clip the images to 26×26 pixels in order to conserve time.

We begin with the reduced MNIST dataset that only consists of digits 0, 3 and 4. Our input therefore consists of three patterns (the three digits) where for every presentation we randomly draw one of the 6000 representations and convert its pixel values to firing rates of the input layer in the range 10–$40\,\text{Hz}$. Testing is performed the same way as described in Section 5.4.2. We present 300 out of the 6000 representations available during training for every digit.

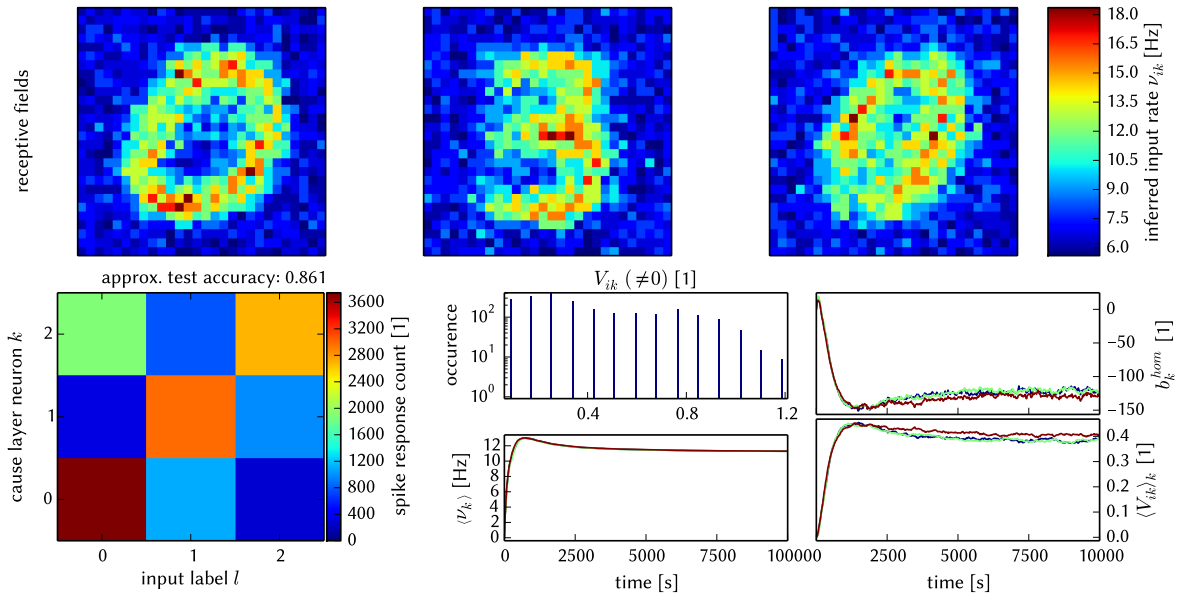

Figure 5.17: Attempt at NSEM learning of the reduced MNIST at a weight resolution of 4-bit. All simulation parameters are the same as in Figure 5.15 except for the input layer. The full parameters can be found in Appendix A.1.3 while the plot structure is explained in Section 5.4.1. Although we can clearly recognize the first neuron coding for the strongest pattern in terms of total firing rate (digit 0), the other two can only be vaguely recognized. This is indicated further by the spike responses. Here the third neuron codes both for 0 and 4 as its receptive field is a superposition of the two.

Using the same parameters as before (Section 5.4.2) and implementing all hardware adjustments, we find that NSEM is unable to sufficiently distinguish between the three digits, as shown in Figure 5.17. The reason for this is the greater variation in label representation. Every example image for each digit is slightly different in shape and orientation. Since white pixels in each image still fire with 10 Hz, their eligibility traces may still induce weight changes in a "spontaneous" way. In the original model, these weight changes are be symmetrically distributed around zero. Weights decay below zero in case a particular input unit is, by chance, silent when the corresponding post-synaptic neuron is active. But, in our case, the lowest possible weight value is zero. Therefore, all spontaneously induced weight changes are positive. Especially in the beginning of learning these spontaneous weights are of the same order in magnitude as "correctly" learning synapses (i.e., synapses that will have a high weight once the network has successfully finished learning). Whenever a sample image for a digit is presented, black pixels will fire with 40 Hz. If the representations vary per label, a particular input pattern might be oriented in such a way that it is unable to activate the "correct" cause layer neuron (i.e., the one randomly chosen neuron that would code for this particular label after successful learning). But, due to spontaneous weights, other cause layer neurons might be activated and adjust their weights towards the – from their point of view – "wrong" input label. In the long run, this "wrongful co-activation" causes all neurons to become randomly activated for all input labels. Since all neurons learn to code for all labels, their receptive fields become a superposition of average rates in all input patterns. Hence, network does

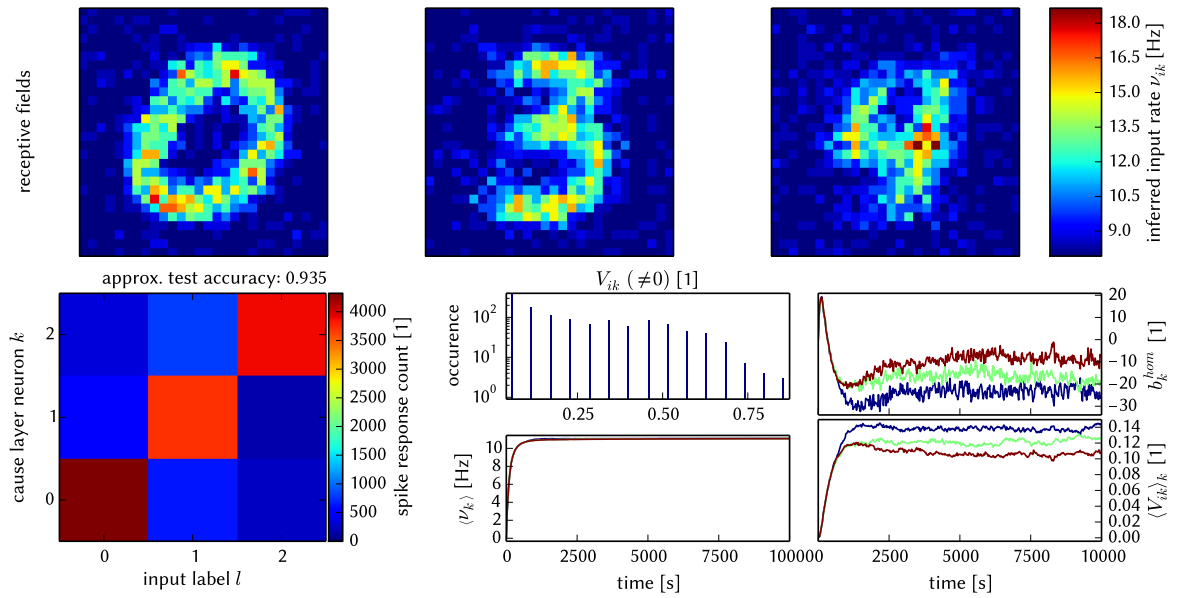

Figure 5.18: NSEM learning of the reduced MNIST set at a weight resolution of 4-bit and adjusted null cause rate. All simulation parameters are the same as in Figure 5.18, except for the null cause rate. While the input still fires with a rate of 10 Hz at the lowest, the weight update rule now assumes it is 15 Hz. All weight ranges are adjusted accordingly. The full parameters can be found in Appendix A.1.3 and the plot structure is explained in Section 5.4.1. The net effect compared to the setup shown in Figure 5.17 is immediately visible: We are able to learn the reduced MNIST set with only 4-bit weight resolution. We can clearly make out the average shape of each digit in the receptive fields. The spike responses also clearly indicate a preferred digit for which each cause layer neuron codes. Still, there is some overlap in the responses left as the spike counts for the other patterns are far from zero. Both the effective bias and the average input weight differ for each cause layer neuron, indicating that the spike-based homeostasis is indeed able to adjust for non-normalized input patterns. As one neuron learns to code for the strongest pattern, it tends to fire for other patterns as well. But in doing so, its homeostatic influence is increased until only the strongest pattern is able to elicit a spike. This way, the other two neurons can learn the remaining patterns in a similar manner. Also, we see the effect of the increased null cause rate in the shape of the weight histogram. Overall, the weights of input units firing with a rate less than the null cause rate are suppressed and kept at zero. Furthermore, the weights of units firing above the null cause rate are reduced as well since the weights represent the log-odds of observed eligibility trace and null cause eligibility trace (now assumed higher).

not learn. This effect is amplified by spike transmission delays discussed in Section 5.4.6, as longer delays correspond to a higher probability for several cause layer neurons spiking simultaneously.

Simply scaling the weights by adjusting the weight conversion factor $f_{\text{theo}\to\text{bio}}$ (see Equation (2.110)) with a scalar factor $0 < s < 1$ does not remedy the situation as the "spontaneous" non-zero weights are still present and continue to disturb learning. Since homeostasis is in place, the problem is only shifted in quantity, not quality.

If, however, we adjust the null cause rate ν_{i0}, we see a vast improvement in network dynamics. The null cause – also called default hypothesis – encodes with what rate the input fires when no cause layer neuron is active (see Sections 2.5.3 and 5.3). For practical purposes, when forcing all afferent synapses from the input layer to be strictly positive it corresponds to the lowest possible input rate the synapses can infer (as then $\ln(\nu_{ik}/\nu_{i0}) = \ln 1 = 0$). As we can see in Figure 5.18, an increase from $10\,\text{Hz}$ to $15\,\text{Hz}$ is sufficient. We are able to learn the reduced MNIST dataset (three digits with 6000 representations each) with only three cause layer neurons and 4-bit weights. The precise value of $15\,\text{Hz}$ is not that strict and was determined using parameter sweeps (see, e.g., Figure 5.29)

Increasing the null cause rate rate has two effects: Firstly, all actual weights V_{ik} in the network are reduced since they encode the log-odds between actual input rate and null cause rate. Secondly, and this is the more important fact, it suppresses the spontaneously emerging weights due to background noise. Comparing Figure 5.17 to Figure 5.18, we notice that while the average weights are lower by a factor of < 4, the effective biases have actually shrunk by a factor of > 4. This seems to support our assumption for the network's inability to learn in Figure 5.17. Increasing the null cause rate – to a reasonable degree – seems to enhance the network's performance to differentiate between input patterns by effectively enhancing the contrast of the receptive fields, thereby limiting the influence of spontaneous non-zero weights of inactive units.

Please note that this does change the generative model. A weight of zero now corresponds to the cause layer neurons "believing" its input fires with a higher rate. If the weights were free to evolve unconstrained they would actually be negative to indicate a firing rate lower than the null cause.

For the full MNIST dataset, we first try to learn with only one exemplary image per digit. As seen in Figure 5.19 we find that we are able to distinguish 9 out of the 10 input images since we are unable separate the two neurons coding for the strongest pattern (digit 0) due to the non-instantaneous inhibitory signal (for more information see Section 5.4.6).

If we try to learn the full MNIST dataset with all 6000 representations per digit, we find that the network is unable to differentiate between all labels to a satisfactory degree. As we can see in Figure 5.20, only some digits can be coded for. The network even interprets different orientations of the same digit as two distinct labels. This indicates that a mere 10 cause layer neurons might be too little to fully capture the complexity of the full MNIST dataset.

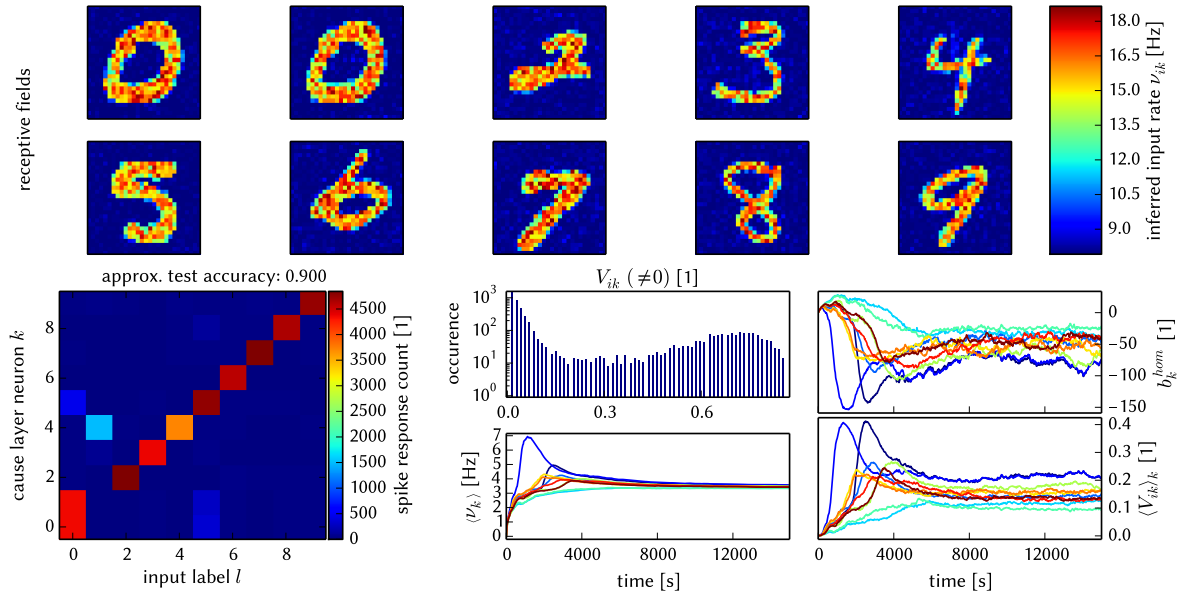

Figure 5.19: Learning of all digits in the MNIST database with NSEM and 6-bit weight resolution. For each digit, we always present the same image. The cause layer consists of 10 neurons. The full list of parameters can be found in Appendix A.1.3. Especially, we adjusted the null cause rate from 10 Hz to 15 Hz as before. The plot structure is explained in Section 5.4.1. We see that during early training, cause layer activity is dominated by the first two cause layer neurons that code for the strongest pattern (digit 0). Due to its constant learning rate, the homeostasis is only able to moderate the situation after the learnt weights have saturated. During this phase, the corresponding cause layer neurons learn a superposition of all patterns (as they are active for all presented input patterns). After their activity is adjusted, the strongest pattern is the only one able to still elicit a spike. Therefore the neurons focus on it, leaving room for the rest of the neurons to learn as well. However, due to the finite delays in the network, one of the first two neurons cannot be discouraged from "unlearning" the 0 digit as they both spike within the delay time-window. Therefore, there is one pattern for which the other neurons do not code, namely the digit 1. It is absorbed by the neuron coding for 4 due to the similarities between the two input patterns. We also observe that the receptive fields are higher in contrast than the corresponding MNIST input images (not shown). This can be attributed to the adjusted null cause rate that is effectively eliminating weights for input units with too low active rates. Overall we see that the neurons are able to distinguish between 9 of the 10 vastly overlapping input patterns.

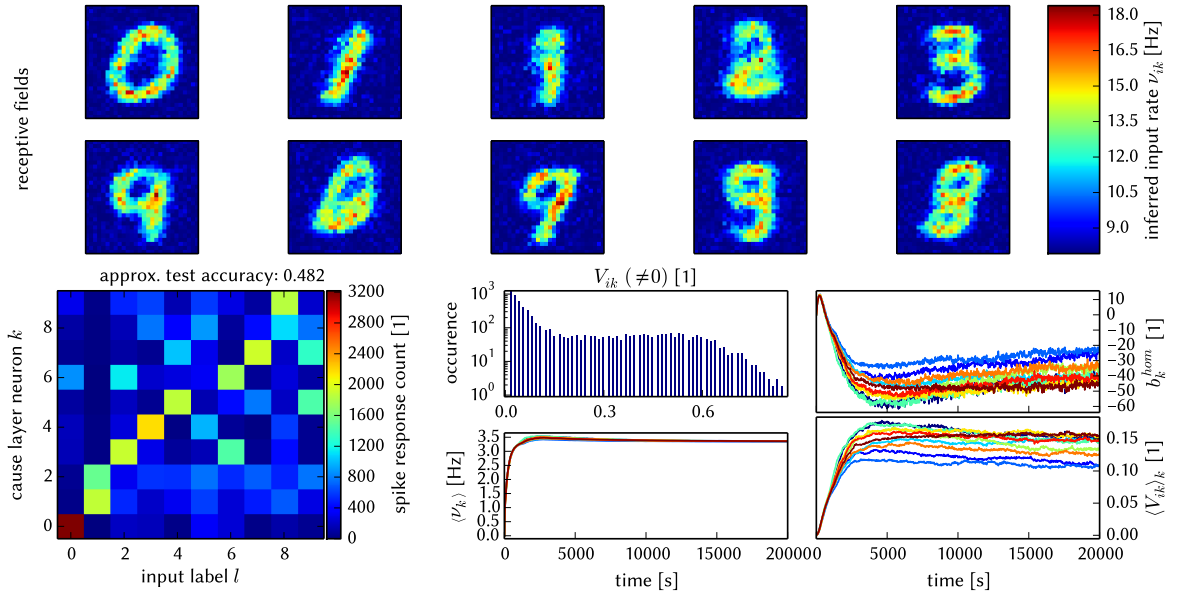

Figure 5.20: Attempt at learning the full MNIST dataset (6000 input images per digit) with NSEM at 6-bit weight resolution. The cause layer consists of 10 neurons. Apart from the increased cause layer neuron number, all other parameters correspond to Figure 5.19. In particular, we adjusted the null cause rate from 10 Hz to 15 Hz as before. The plot structure is explained in Section 5.4.1. We observe that while some neurons are able to specialize on one digit (e.g., 1, 3 or 8), the others tend to learn superpositions of digits or even rather different orientations for the same digits (the second and third code for different orientations of 1). The homeostasis is able to moderate the excitabilities to some extent, however, the classification performance of the network is rudimentary at best. 10 cause layer neurons seem to be too little to fully capture all possible orientations of all possible digits.

Common Box Input Scheme

To try and quantify the effect of modulating the null cause rate we switch to the common box input scheme. It was devised to simulate largely overlapping input labels with high variances in their representations, similar to full MNIST. In this input scheme, we can think of the units in the input layer to arranged in a set boxes. We have one larger common box of 100 units as well as – for every label – one smaller label specific box of 30 units. Since we have 9 patterns total, we have 370 units in our input layer. For every pattern we present, we select a set number of units to be "active". These fire with a high rate of 40 Hz. All other units are said to be "inactive" and fire with 10 Hz. The active units are chosen in the following way: First, we draw 27 out of 30 units from the "active" label box (i.e., the box corresponding to the label being presented). Then we draw a fixed number of units from each other label box. These boxes are denoted as "inactive'. Finally, we draw active units from the common box until we have reached our grand total of 127. In Figure 5.21, we present an example of the input scheme and the resulting network dynamics.

This emulates MNIST learning in the following way: The large overlap between several patterns is realized by the common box from which each label draws the majority if its active

units. The distinctive characteristic of each label is implemented by the active box in which – relatively speaking – most units are active. The units from inactive boxes represent the variations in representations for single labels. They are distributed in a sparse (or "diffuse") manner but nevertheless fire with a high rate. Therefore, if the weights of the corresponding synapses are "spontaneously" non-zero, they can disrupt learning in the same way as for MNIST (discussed above).

We then then vary the number of units drawn from the inactive boxes and try to compensate for this effect by varying the null cause rate. As can be seen in Figure 5.22, adjusting null cause rate within limits enables the network to correctly classify the input labels in case of more diffuse input.

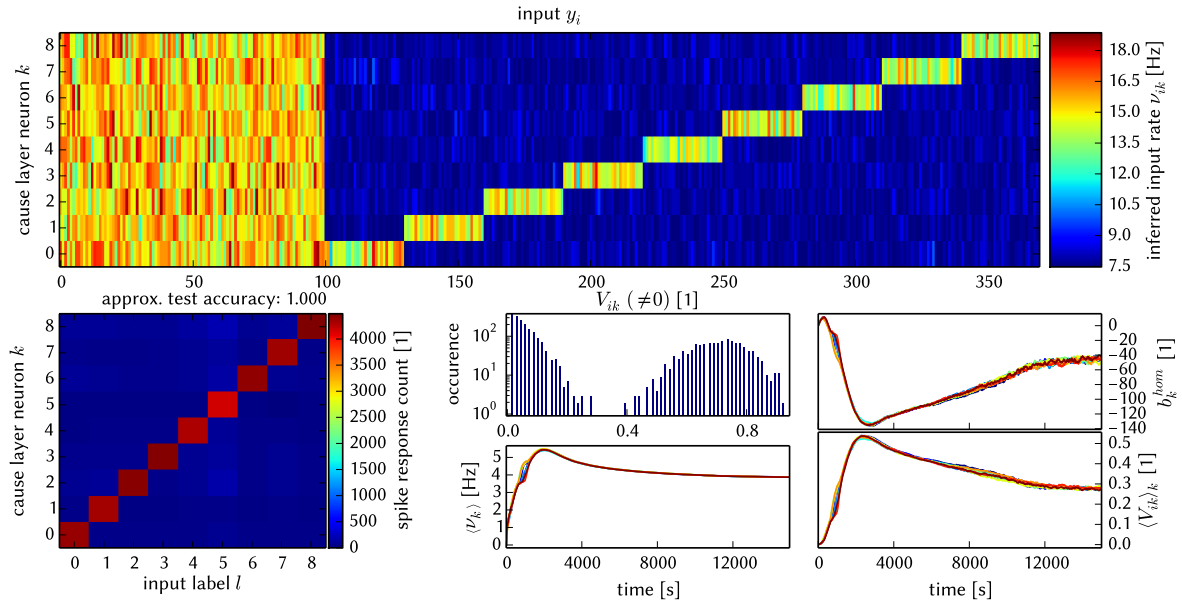

Figure 5.21: Exemplary network dynamics of common box input scheme. The input is comprised of 370 input units, forming a one-dimensional string. The string is separated into a common area – or box – of 100 units and 9 label specific boxes of 30 units. For each pattern, we randomly select 127 units and denote them as "active". All other units are "inactive". Likewise we denote the label specific box of the current pattern as "active" and all other label specific boxes as "inactive". Active units fire with 40 Hz while inactive units fire with 10 Hz. Which 127 units are active is determined in the following way: First, we choose the active box and select 27 out of its 30 units. Then we in turn select all other inactive boxes and choose 2 units each. We then select the remaining 84 units from the 100 units in the common box. In this example the null cause rate is adjusted to 14 Hz. The full parameter set can be found in Appendix A.1.3 and the plot structure is explained in Section 5.4.1. We observe a bimodal weight distribution. The small weights are a result of the active units in inactive boxes while the large weights correspond to the common box as well as the active box. Just like in Figure 5.19, we find that the early learning phase is comprised of a steep increase in weights that all neurons participate in. Only after the increase in synaptic input weight slows down, spike-based homeostasis is able to moderate the network activity. Nevertheless, the network is able to classify all of the nine input labels correctly.

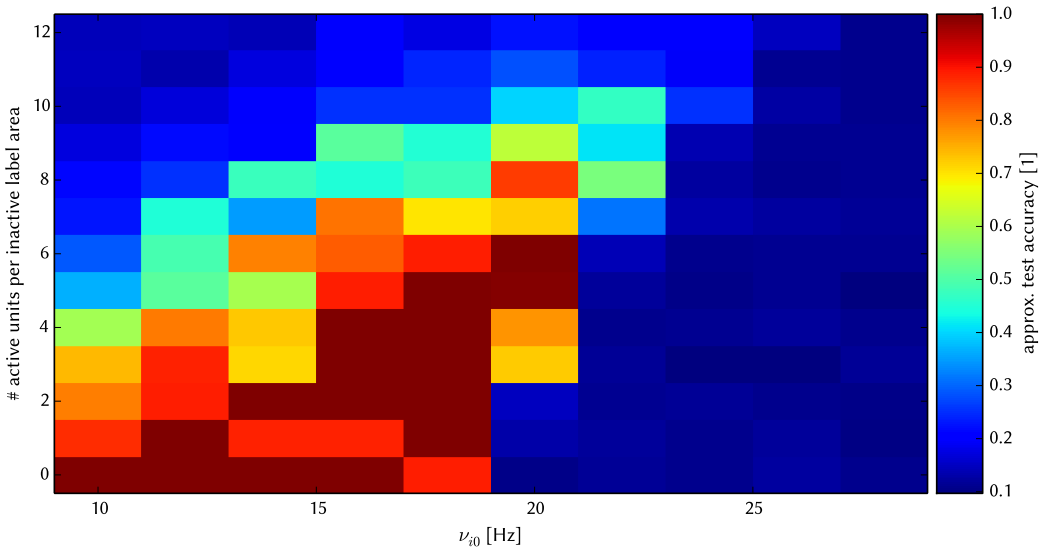

Figure 5.22: Null cause as input rate filter. The network setup corresponds to Figure 5.21. We vary the number of samples drawn from each inactive box to increase the strength of non-specific background noise and try to compensate for it by adjusting the null cause rate ν_{i0}. A higher null cause rate has the effect of suppressing weights from input neurons firing with lower rate, thereby increasing the contrast in the forming receptive fields. This can be seen in the plot as the classification accuracy increases for higher numbers of units in inactive boxes when we increase the null cause rate, but only up to a certain point. At first we see a decrease in performance for low numbers of units in inactive boxes. This is due to the fact that the learnable rate range is now rather compressed. Keep in mind that the total number of active units per pattern remains constant, only their distribution changes. For 0 units in inactive boxes, all 100 common units are *always* active whereas the units in the active boxes are only active in 27 out of 30 pattern presentations on average. As more units are drawn from inactive boxes, the average common box unit becomes less active on average, allowing the cause layer neurons to learn their active boxes. As we increase the null cause rate further, the network ceases to distinguish between the labels. In this case, even when the common box is no longer posing a problem, the active boxes are no longer able to attain enough weights to reliably activate the corresponding cause layer neuron. Conclusively, we find that adjusting the null cause rate can help to increase classification performance by suppressing the influence of both low rates as well as diffuse background input.

5.4.4 Homeostasis Source Type

In this section we briefly compare the two immediate choices for spike-based homeostasis: Poisson and periodic. We find that at high rates ($\geq 1000\,\mathrm{Hz}$), there is no significant difference in network performance. At lower rates, however, periodic generators clearly outperform Poisson background sources. This can be seen in Figure 5.23.

The reason for this is simple: Due to the almost perfectly δ-distributed ISIs, a periodic background generator is able to generate a less variable post-synaptic conductance. As we are trying to emulate a constant current to shift the mean membrane potential, it is modeled more closely by the signal of a periodic generator. If each cause layer neuron is receiving its own Poisson spike train, the difference in membrane potentials is amplified even more. In some cases, this can disturb learning. The differentiation of the receptive fields to code for different patterns starts out slowly in the beginning of learning. If the variance in effective bias for each neuron obscures this differentiation, no learning takes place and the neurons start to code for a general superposition of all input patterns. We observe the same effect if the learning rate for homeostasis is chosen too high. Since at higher rates, the weight of a single spike has a lower absolute value (see Equation (5.15)) and spikes occur in greater succession, the difference is diminished.

In the context of implementing NSEM. If homeostasis is implemented in a closed loop setup, every unnecessary computation on the host computer should be avoided, since it needs to operate in biological real time at the corresponding speed-up factor. This includes pseudo-random operations to generate Poisson spikes, especially if we have to adjust their rate on the fly since the most likely implementation of homeostasis consists of injecting spike trains of variable rate via several synaptic input channels. Each synaptic input channel is set to a different synaptic weight (see Section 5.2). Also, if bandwidth is a concern regarding the amount of spikes we can send to the system during simulation, we can preserve network dynamics at lower homeostatic background rates when using periodic generators.

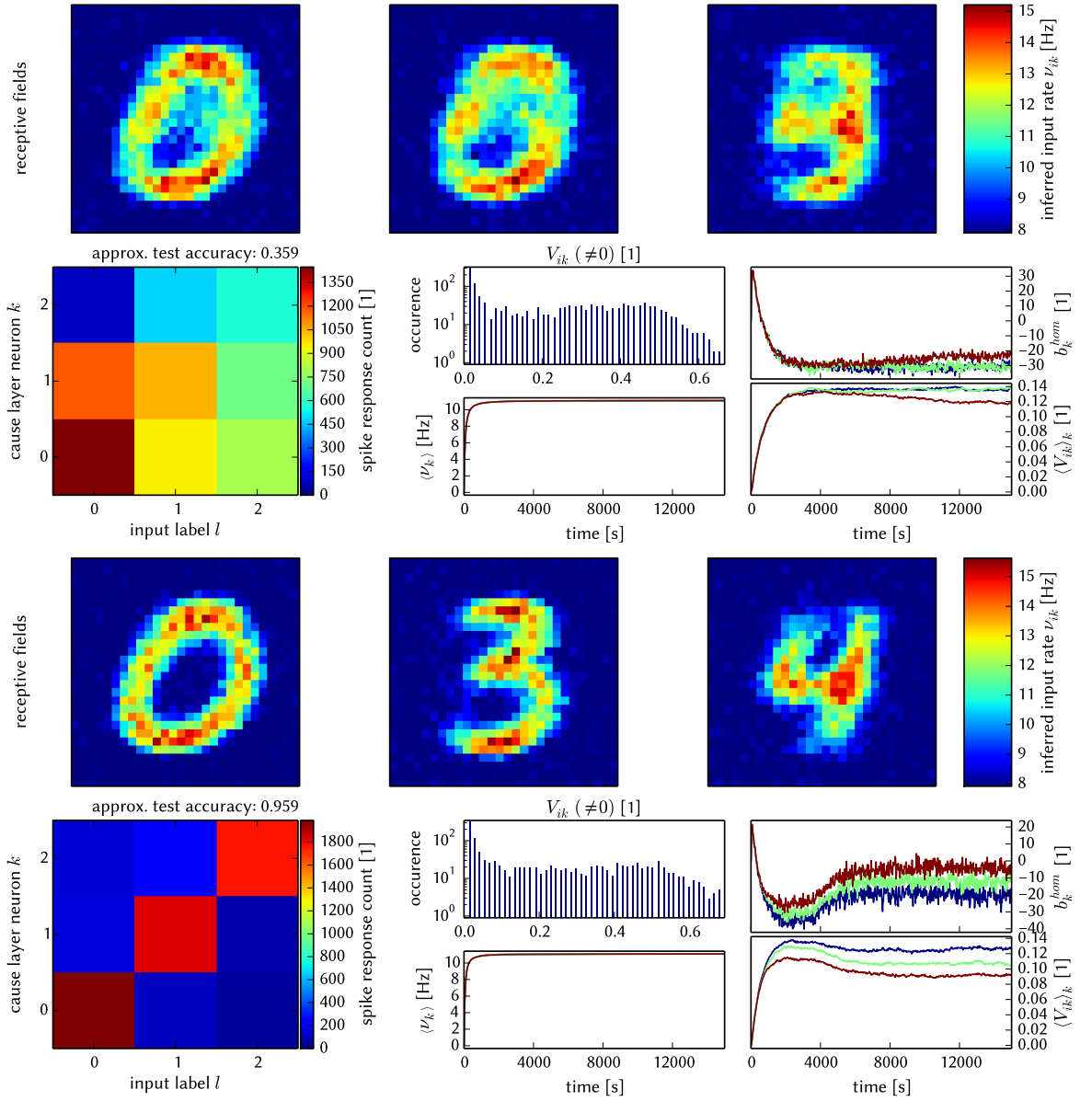

Figure 5.23: Comparison between different models as source for homeostasis: Poisson background sources (top) and periodic background source (bottom). Both background sources fire with 500 Hz while the network is learning the reduced MNIST dataset firing with a rate of 10–40 Hz. The network implements NSEM at a weight resolution of 6-bit. The full parameters can be found in Appendix A.1.4 and the plot structure is explained in Section 5.4.1. From the receptive fields, the spike response counts and the approximate test accuracy we see that the network with periodic homeostasis background is able to distinct the three patterns while the Poisson-supplied network is not. Please see the text for details.

5.4.5 Larger Networks

In this section, we briefly explore the possibilities of NSEM in larger networks regarding the number of cause layer neurons. As these simulations are very time consuming to perform and analyse in any systematic fashion, we leave a thorough investigation to be conducted as further work. In particular, these are the cases where we could benefit the most from the enormous speed-up of neuromorphic hardware.

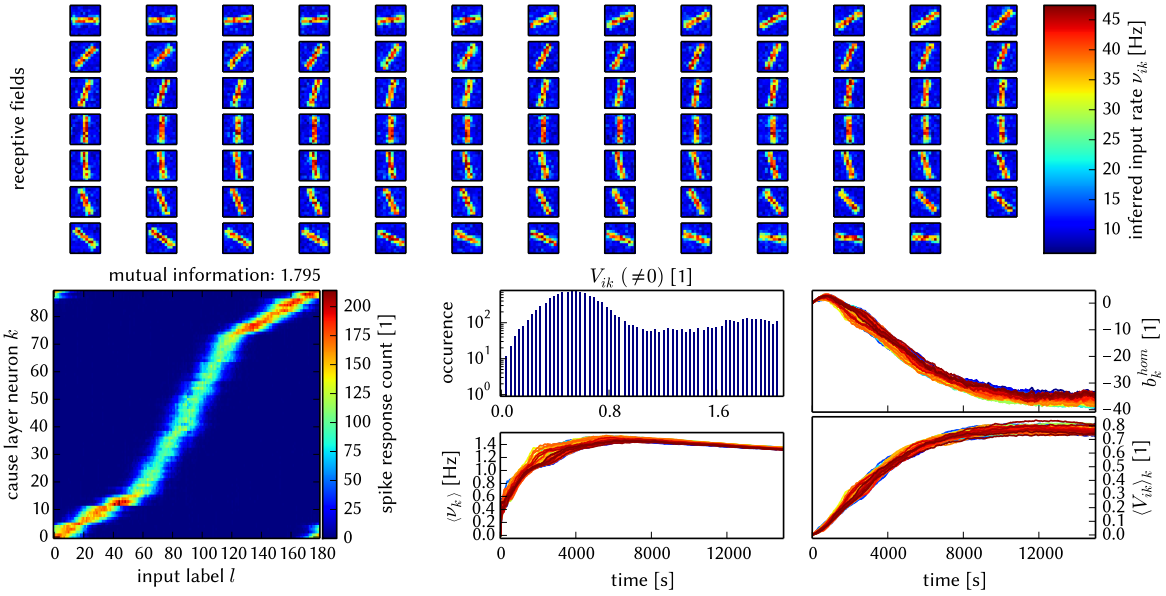

Figure 5.24: Large network simulation for 90 cause layer neurons learning the orientation of a 2 pixel wide strip in a 12×12 image. In order to increase the speed of learning we set $\tau_{\text{syn}} = \tau_{\text{refrac}} = 10\,\text{ms}$. Furthermore, we present orientations in the range 45–135° twice as often as the others on average. The simulation is performed with NSEM weight updates, as in Section 5.3 at a weight resolution of 6-bit. The full parameters can be found in Appendix A.1.5 and the plot structure is explained in Section 5.4.1. We see that also in this rather large setup each cause layer neuron is able to code for a certain range of orientations which is now much smaller than in Section 5.4.2. The inferred input rates are higher which is a result of the shorter synaptic and refractory time constants (at $\tau_{\text{syn}} = 10\,\text{ms}$, the estimated inferred rate for 70 Hz is ∼43 Hz). Furthermore, we see that homeostasis forces more neurons to code for the patterns presented more frequently. Please note that we present all patterns for the same amount of time during the subsequent test run, which lowers the spike response for the more frequently presented patterns (during training). This is due to the fact that the spike-based homeostasis ensures that each neurons activity is the same. Thus, more cause neurons tend to code for labels that are presented more frequently. We also see a larger spread in average rates, input weights and effective biases, indicating that during training some cause layer neurons become too active for short periods of time until the homeostasis is able to moderate their activity.

In Figure 5.24, we revisit the input scheme from Section 5.4.2. We increase the cause layer to 90 units and present half of all input labels twice as often as the rest. Nevertheless, the

cause layer neurons manage code for separate sections of the input space that are much more narrow than in Section 5.4.2. However, they tend to overlap far more. This is due to the time delay it takes for the inhibitory signal to arrive after the cause layer has emitted a spike. As we can see in Figure 5.25, we have a rather large peak at very low ISIs. This coincides perfectly with the inhibition delay of $\sim 1.4\,\mathrm{ms}$. When comparing the raster plots of the network before and after learning in Figure 5.26, we find that several cause layer neurons activate for each presented input pattern. As the inhibitory signal arrives too late, they cannot be trained to segregate the input space even more.

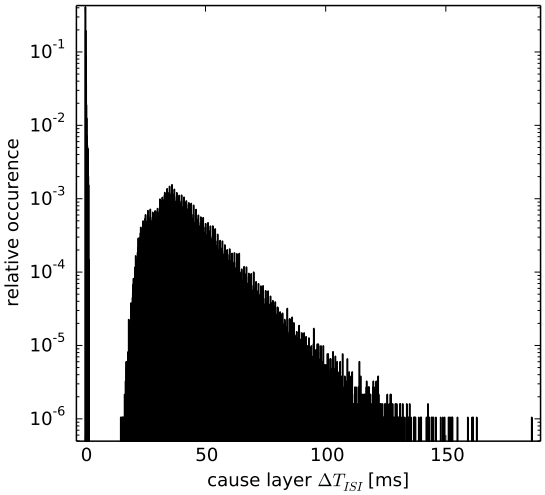

Figure 5.25: ISI distribution for the cause layer spike train during training. The network is the same as in Figure 5.24. Spikes within the delay-time window can not be prevented as evidenced by the concentration of ISIs in the range 0–$1.4\,\mathrm{ms}$. Beyond that, we see that the inhibition is sufficient to inhibit network activity once the inhibitory signal arrives as we observe no ISIs in the range 1.4–$11.4\,\mathrm{ms}$. The observed ISIs are larger than the ideal case as the homeostasis is more inhibiting due to coinciding spikes. Therefore it takes slightly longer for the cause layer to spike again on average.

Finally, we present the current state regarding full MNIST learning with a network consisting of 100 cause layer neurons. The network dynamics are shown in Figure 5.27. From a functional point of view, the network is as good as in Figure 5.20. While the receptive fields of many cause layer neurons bear a clear resemblance to several digits, the classification performance of the network can not be regarded as satisfactory. Figure 5.27 serves as an example for the current state of actively ongoing investigations regarding the ability of the network to learn the full MNIST dataset. Among the investigated parameters are the learning rates and their ratio, synaptic/refractory time constants, overall network activity p_{net} and pauses between pattern presentation to give the network a change to re-establish itself after learning. So far, no satisfactory classification performance has been achieved. Further work is needed in this area.

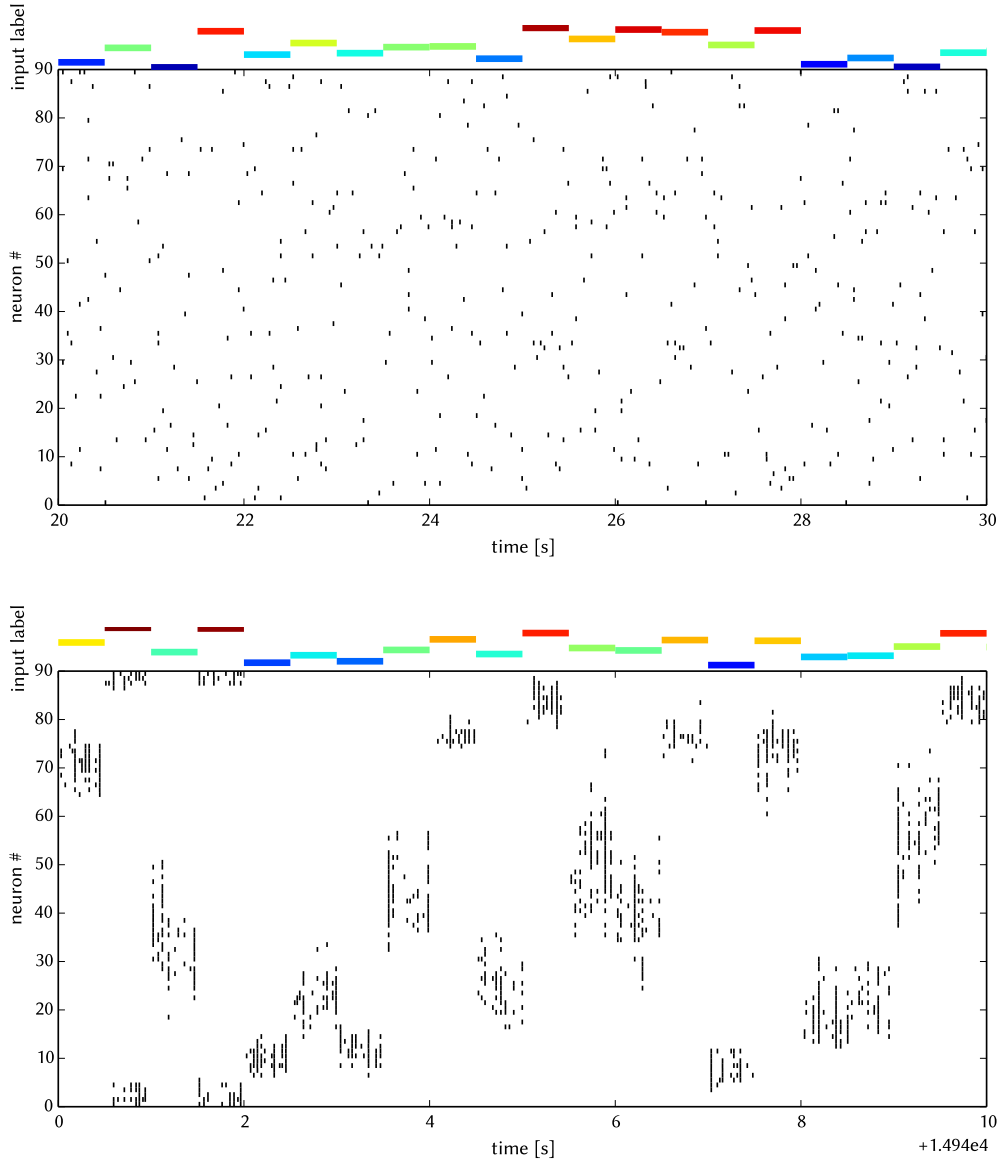

Figure 5.26: Raster plots of cause layer activity before and after training. The network is the same as in Figure 5.24. At the top of both plots we see a color-coded representation of which pattern is currently being presented to the network (dark blue corresponding to an orientation of 0°). Top: Spike trains during early learning phase (time range 10–20 s). We see that all cause layer neurons fire with no clear preference for all presented input patterns. We see that the inhibition is strong enough to discourage spiking activity after each cause layer neuron spike, as already shown in Figure 5.25. Bottom: Spike trains after learning has stabilized (time range 14 940–14 950 s). Each neuron responds to their preferred orientation only. Unfortunately, the stimulus is often sufficient for several neurons to spike within the time-delay window, rendering a further separation of the input space via mutual inhibition ineffective. In accordance with the spike responses from Figure 5.24 we see that more cause layer neurons respond to the more frequent orientations 45–135°.

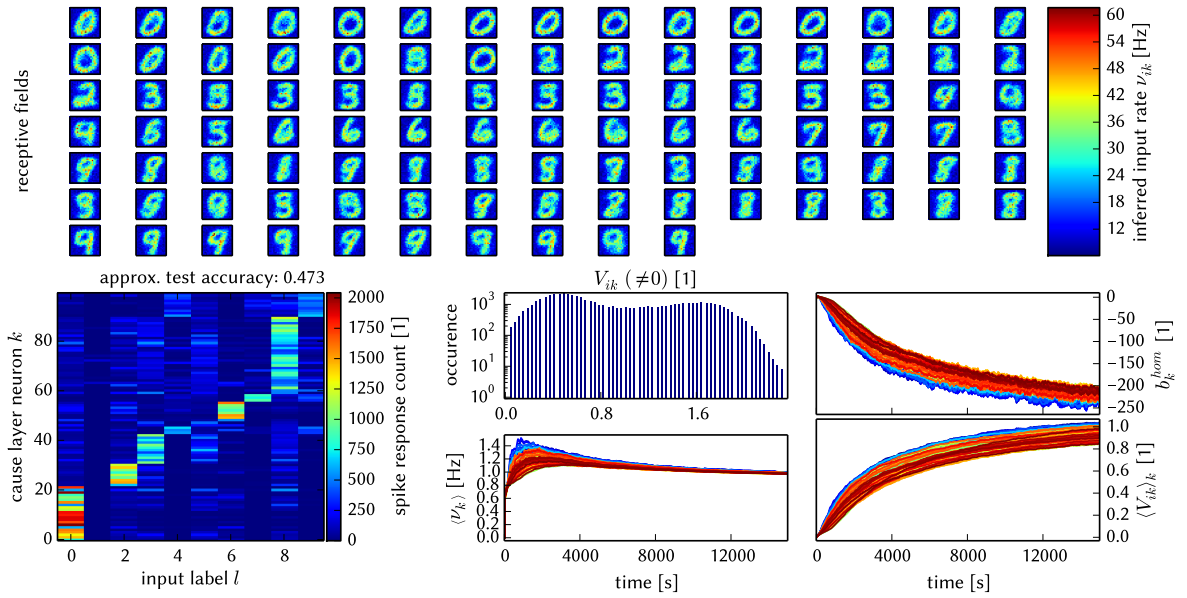

Figure 5.27: Attempt at learning the full MNIST dataset with a network containing 100 cause layer neurons. The input layer is firing with rates of 10–100 Hz. The full parameters are found in Appendix A.1.5 and the plot structure is explained in Section 5.4.1. Same as in Figures 5.19 and 5.20, we observe that during early training the network activity is lead by the cause layer neurons later coding for the strongest pattern (digit 0). While their receptive field do seem to account for several slightly different realizations of digit 0, their overall proportion among the cause layer neurons is far too high with $^{21}/_{100}$ compared to 10 that we would expect. Overall, the classification performance of larger networks in regards to full MNIST remains non-satisfactory.

5.4.6 Non-negligible delays

As delays and their effects were already discussed here and there in the previous sections, we want to recapitulate and illustrate their effects as they turn out to be the hardest challenge when implementing SEM on neuromorphic hardware.

The main objective of SEM-like learning is to find hidden causes in the presented input pattern. Whenever a cause layer neuron spikes, its weights get shifted so that it is more likely to get activated by the same input stimulus next time around. Conversely, other cause layer neurons should be discouraged from coding for the same input because it is already accounted for. For this we need mutual inhibition. The greater the time delay for the inhibitory signal to arrive at the cause layer, the greater the chance of another neuron being activated as well. Co-activated neurons then start to focus on the same input patterns which diminishes classification performance.

As signals are travelling asynchronously on the neuromorphic substrate during emulation, they take an inherent and non-negligible time to arrive. Due to the high speed-up factor, even though signals are as fast as on any other commercially available device, delays will

correspond to biological time frames in the order of $\sim 1\,\mathrm{ms}$. All simulations in this thesis were carried out with an effective time delay of $\sim 1.4\,\mathrm{ms}$ to account for this.

For a low number of neurons (see, e.g., Section 5.4.2 or Section 5.4.3), we find that this delay is less of a problem as each cause layer neuron is able to account for a disjoint set of input labels. The probability for the cause layer to emit a spike at any point in time (given no inhibitory signal) is directly proportional to the number of stochastic neurons it is comprised of. Therefore, we are presented with a different picture for larger cause layers: We can see in Figure 5.26 that after training, each input pattern activates several cause layer neurons. We can rule out a too weak inhibitory signal as source for the overlap in responses after looking at the ISI distribution the spike train generated by the complete cause layer (found in Figure 5.25): We observe a sharp peak in the distribution for times up to the time delay of the inhibitory signal ($1.4\,\mathrm{ms}$). This indicates that once the inhibitory signal arrives, it is indeed able to inhibit all activity. The overlap in pattern responses is therefore caused by the non-negligible delays.

In Figure 5.28, we investigate of higher delays while trying to counter-act their effects by scaling the weight conversion factor $f_{\mathrm{theo}\to\mathrm{bio}}$ (see Equation (2.110)) to moderate the overall input strength each neuron receives. As we already discussed in Section 5.4.3, weight scaling is not effective. Furthermore, we see a clear correlation between the time delay and the number of neurons that start to code for the strongest pattern (digit 0). This supports our hypothesis that delays of the inhibitory signal are the greatest challenge for network dynamics NSEM.

While adjusting the null case rate did improve the classification performance for learning the reduced MNIST dataset (see Section 5.4.3), we find in Figure 5.29 that it is unable to prevent the detrimental effect of too high delays.

The probability of two cause layer neurons activating simultaneously is proportional to the product of general network activity as well as the given time window they have to spike. If we cannot shorten the spike transmission delay (i.e., the time window), we might be able to limit its detrimental effects by lowering the overall network activity. Since weight updates in NSEM are only performed if the post-synaptic neurons fire, this results in effectively longer training durations. The issue of spike transmission delays and possible compensation methods is currently still under active investigation.

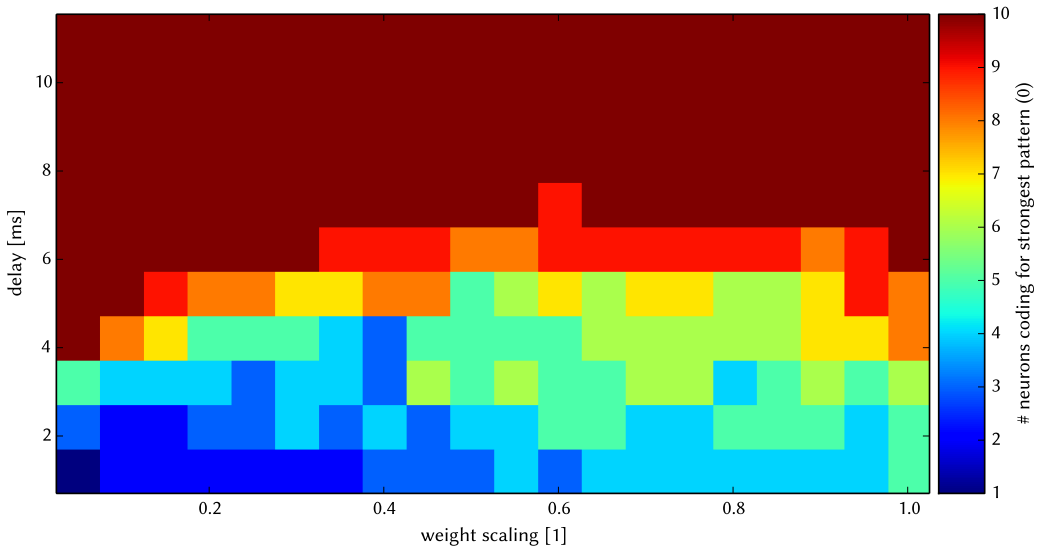

Figure 5.28: Number of patterns coding primarily for the strongest pattern in terms of total input firing rate (digit 0) for different inhibition delays and weight scaling factors (i.e., we arbitrarily scale the weight conversion factor $f_{\text{theo}\to\text{bio}}$, Equation (2.110)). A network consisting of 10 cause layer neurons is presented with the full MNIST label range (digits 0 to 9), but for each digit we always present the same pre-chosen example. Each network implements NSEM at a weight resolution of 6-bit. The full parameters for a single simulation can be found in Appendix A.1.6. For each network we evaluate for which input label (i.e., which digit) each cause layer neuron had the highest activity in the test run. We then count how many of the ten cause layer neurons code for the zero digit, corresponding to the strongest label (i.e., on average, images containing a zero show the most black pixels). The longer the delay time-window of the inhibitory signal, the higher the probability of more than one cause layer neuron spiking and learning from the same input. Therefore, as the delay increases, cause layer neurons are able to learn more independently and hence naturally tend to learn the pattern with the highest synaptic input. We find that scaling the weight conversion factor $f_{\text{theo}\to\text{bio}}$ in order to overall moderate the strength of synaptic input in the network has limited to no effect. This is in accordance to Section 5.4.2. Here we observed that when the absolute weights in the network are decreased, spike-based homeostasis adjusts the effective biases accordingly, thereby preserving network dynamics irrespectively of the absolute synaptic input strength.

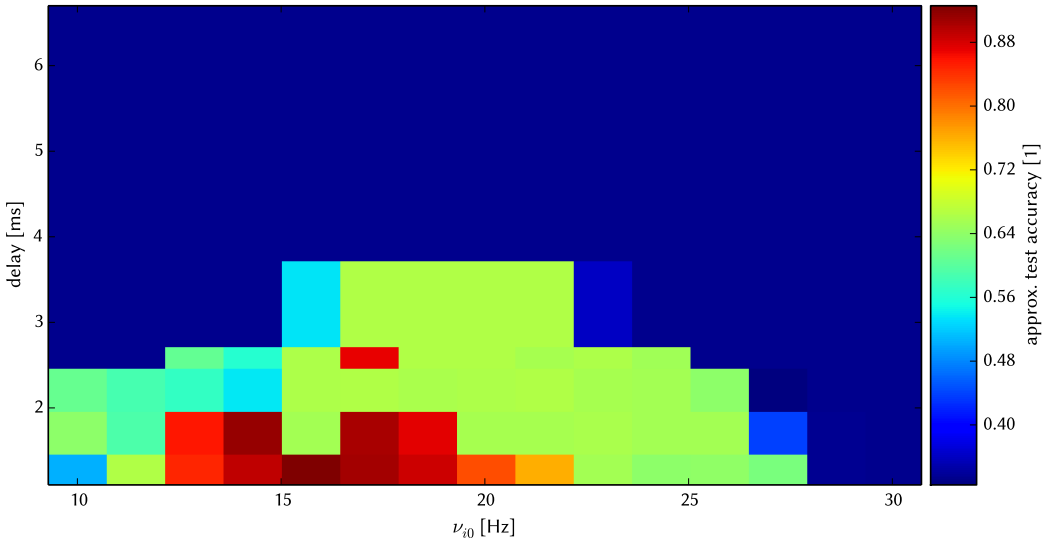

Figure 5.29: Learning the reduced MNIST dataset with varying delay time windows for the inhibitory signal and null cause rates. A network of 3 cause layer neurons is presented with the reduced MNIST set. The network implements NSEM at a weight resolution of 6-bit. The full parameters can be found in Appendix A.1.6. For each network we compute the approximate test accuracy. We observe that for low delays, adjusting the null cause rate is a sufficient way to learn the reduced MNIST dataset, but as the delay increases, the classification performance of the network suffers. Adjusting the null cause rate is hence no adequate method to deal with larger – but unrealistic – delays. Furthermore, we see that adjusting the null cause rate too high becomes detrimental to the network performance. In these cases, the weight range becomes so compressed that regular learning dynamics cannot commence. Too many input weights are driven to be zero, rendering the resulting synaptic input to weak to sufficiently activate any cause layer neurons.

6 A new Software Framework for Spike-based Inference

Proper research in computational sciences can only be conducted with proper software support. In order to speed-up the simulations for both this thesis and other publications, a set of new software libraries was developed. They operate on-top of PyNN and *NEural Simulation Tool* (NEST), make use of numpy [Numpy, 2012] and matplotlib [Hunter, 2007] for plotting. They serve as an abstraction layer for performing spike-based inference.

The first one, *spike-based sampling* (sbs), implements stochastic *leaky integrate-and-fire* (LIF) sampling. It takes care of calibrating LIF neurons for given neuron/input parameters and allows the evaluation of arbitrary Boltzmann-distributions in static networks. It is detailed in Section 6.1. For dynamic weight evolution there are two further libraries: *spike-based expectation maximization framework* (SEMf) and *spike-based learning* (sbl). SEMf implements *spike-based expectation maximization* (SEM)-based feed-forward learning in networks of LIF neurons and was used extensively to study the feasibility of SEM-learning on neuromorphic hardware (see Chapter 5). It is described in Section 6.2. The last library, sbl – at the time of writing still under active development – is aimed at the efficient implementation of *contrastive divergence* (CD)-based algorithms (see Section 2.5.1) in networks of stochastic LIF neurons and can be found in Section 6.3. Both SEMf and sbl make use of custom synapse and neuron models that currently have only been implemented for the NEST simulator. They are detailed in Section 6.4.

6.1 New Library: Spike-based Sampling

spike-based sampling (sbs) is based in spirit on an earlier set of scripts that were developed during the development of Neural Sampling theory (see Section 2.4.4). Since the old set of scripts was written – as is usually the case when prototyping – in organic fashion, it intertwined Boltzmann weight conversion, PyNN commands and plotting. sbs clearly separates the abstract concept of stochastic LIF neurons and *Boltzmann machine* (BM) from network communication code. Its two main conceptual buildings blocks are `LIFsampler` as well as `BoltzmannMachine`.

The `LIFsampler` is described by a neuron model, its corresponding parameters and a background source configuration (typically one excitatory and one inhibitory Poisson source with set rate and synaptic weight). Given this configuration, it is able to automatically calibrate itself to find the weight conversion factors $f^{\text{ext/inh}}_{\text{theo}\to\text{bio}}$ as well as the membrane potential at which the activation function is exactly 0.5 (see Section 2.4.4 and Section 6.1.1). For re-usability, a

complete `LIFsampler`'s configuration is saved as *JavaScript Object Notation* (JSON)-file to allow for easy inspection. After calibrating once, the `LIFsampler` can be created from file.

The `BoltzmannMachine` on the other hand implements – as the name suggests – a BM of interconnected heterogeneously configured `LIFsamplers`. The user can specify either theoretical or biological weight/bias configuration. The `BoltzmannMachine` takes care of automatically translating between the two, taking into account each `LIFsamplers` possibly unique calibration data. The network can then be run to gather spike samples from the corresponding biological network, from which a sample-based approximation of the underlying probability distributions is automatically computed. Renewing synapses with custom *Tsodyks-Markram* (TM) parameters are also possible (see Section 2.4.4). For smaller networks, theoretical distributions can be computed as well as the *Kullback-Leibler divergence* (D_{KL}) computed between the two. Demanding computations regarding probability distributions or state computations from spike trains are implemented using Cython [Behnel et al., 2011], a library that converts type-annotated Python to C that is then pre-compiled and loaded as shared library during execution.

An important feature of sbs is that no PyNN-specific code is run until the user explicitly requests it, e.g., via each objects `create()`-routine. This is necessary due to PyNN's inherent "statefulness". Even though a call to `sim.end()`/`sim.setup(...)` is supposed to wipe the currently used simulator's network state (according to the *application programming interface* (API)-specification) it is not always the case. This way, tasks such as computing theoretical probability distributions or performing weight conversions of already calibrated `LIFsamplers` can be accomplished without involving PyNN at all. Furthermore, as detailed in Section 6.1.2, tasks that involve PyNN – e.g., calibration or the gathering of spikes given a BM-configuration – can be offloaded into subprocesses in a fully transparent manner, allowing for more than one of such tasks to be performed in a single run.

6.1.1 Calibration

Calibrating a stochastic LIF neuron involves computing the average activity of the neuron given a certain leak reversal potential E_L and fitting a logistic function to the result (see Section 2.4.4). The user can either specify the value range in which the calibration should be performed or simply let sbs find the relevant part itself. For additional speed, all different settings of E_L are computed in one network run.

6.1.2 Seamless computation in subprocesses

sbs features a function decorator `@RunInSubprocess` that can be used to offload the execution of arbitrary functions into a subprocess that is terminated after the function has finished executing. The primary use cases are small PyNN simulation such as calibrations and execution of BMs. Especially parameter sweeps for different BMs, when executed in the same script, become inherently slower due to memory leaks. By offloading each simulation into its own subprocess, we avoid these possible memory leaks altogether. The process is completely

transparent to the user. The function can be called regularly and returns whatever it would have returned if run in the same process. For example, this enables evaluation of several BM configurations via the same script in rapid succession without the user having to worry about reusing objects from the previous iteration for the next (in case the network size or `LIFsampler` configuration changes).

Furthermore, the class decorator `@RunClassInSubprocess` can be used to offload the entire life cycle of the decorated class into another subprocess. All calls to methods and attributes are forwarded to the object and the result returned in the parent process. The subprocess can also be terminated and re-created, properly wiping the state of the object contained within.

For both decorators, the only limitation is that the decorated function/class must be defined when the module it resides in gets imported. Furthermore, it should not depend on the global state of the module as we do not `fork` the parent process, but rather execute the module from the start.

6.1.3 On-demand computing via descriptors

Another feature introduced by sbs is the ability to compute attributes of a class on demand. Each attribute computes the values of other attributes it depends on automatically. This is accomplished by decorating each attribute by `@DependsOn(...)` where the arguments are the names of the corresponding dependencies. The whole class object then needs to be decorated with `@HasDependencies`. Values are stored and reused once computed and only discarded when one of the dependencies is changed.

For example, accessing the sampled Boltzmann probability distribution of the `Boltzmann`-object for the first time after automatically computes the distribution from the recorded spike data. Each subsequent access does not lead to a new computation, the probability distribution is stored. If, however, new spike data is gathered, the old distribution is discarded and recomputed once needed. The same relationship exists between the theoretical distribution and the weights set for the network.

The attribute is a simple function accepting up to one argument. Akin to the `properties` concept of Python itself, it has to implement both `get` and `set` operations. If the optional argument is `None` (`get` operation), the function has to compute its current value from its dependencies and return it. If the optional argument is defined (`set` operation), the function has the ability to transform the value before returning what should be stored.

6.1.4 Speed-up

Especially during CD-like training (see Section 2.5.1), when many different networks need to be initialized and evaluated, fast initialization of networks is key. The previous implementation of LIF sampling – for historical reasons – used a single PyNN `Population` per stochastic neuron and correspondingly one `Projection` per synapse between any two `Populations`. sbs streamlines the network creation process by using only a single `Population` and `Projection`

for all neurons and synapse type (excitatory/inhibitory). Furthermore, if there are backend-specific optimizations (e.g., using only a single Poisson source object for all samplers in NEST) they are automatically applied.

To get an indication of the achieved speed-up, we perform an exemplary network initialization for 194 stochastic neurons. For the old implementation this step took between 50–$70\,\mathrm{s}$, depending on the level of optimization. With sbs, the same initialization takes $4.8\,\mathrm{s}$. The result is a speed-up factor of 10–15.

6.2 New Library: Spike-based Expectation Maximization framework

The *spike-based expectation maximization framework* (SEMf) is used for all simulations in Chapter 5. It operates on the principle that a simulation of a certain network configuration should be fully specified by a set parameters (i.e., a data -structure) and *not* a script-file that also describes *how* the simulation should be conducted. In a script file, configuration can theoretically be anywhere, potentially making it very hard to comprehend after some times has passed.

Therefore in SEMf, each simulation is fully specified by a single parameter file. This parameter file can be acted upon through certain `controllers`. Each `controller` performs a specific task, such as network simulation, computation of analytical properties or plotting.

The network dynamics are specified by abstract interacting interchangeable parts. In the case of SEM, for instance, the network is comprised of an input layer, a cause layer, a homeostasis implementation acting upon the cause layer and a connection object facilitating the dynamic synapses between input and cause layer. Each abstract object type effectively defines an interface that subclasses can implement. Which subclass is chosen with what parameters is then defined in the parameter file. This allows very easy testing of several network configurations because each abstract concept can very easily be replaced. For example, each of the different updated rules described in Section 5.3 is implemented as its own connection-subclass. Which connections are chosen during simulation is then completely independent from, say, what homeostatic method was chosen. Each such object type in SEMf – including `controllers` – has a set of default parameters that can only be changed when creating the object and define its complete behaviour.

Apart from simulations in NEST, conducted in Chapter 5, the original theoretical model (detailed in Section 2.5.3) was implemented as `boost.python`-wrapped C++ module. It can be swapped in for the PyNN-based simulations run in NEST. This allows for better comparability between abstract theoretical model and NEST-aided simulations, because the simulation data is then worked on by the same set of analysis and plotting tools, irrespective of origin.

6.2.1 Data Management

Simulation data is stored in *Hierachical Data Format* (HDF) file containers. Since simulations typically take the longest to perform, their results are stored in a separate container that is locked in order to prevent data corruption by any analysis conducted later.

The simulation parameters are stored in *YAML Ain't Markup Language* (YAML)-files. For ease-of-use, SEMf provides a *command line interface* (CLI) that, through the use of Python Meta programming, is able to automatically display the default parameters of each available implementation of every abstract concept class. It does so in YAML format so that the user can directly copy and paste the default parameters into a new file and have a valid network specification.

The default YAML syntax is extended by a custom tag `!ee` (short for: execute expression) that allows the evaluation of strings in Python. In each calculation, the python commands may access two custom objects. First, `np` to access numpy specific functions and then the cache object `cc`. The cache object `cc` is useful for performing minor calculations prior to simulation in the parameter file. Each parameter file may have a custom top-level entry called `cache`, that is a list of dictionaries. The values in each dictionary should be strings to be evaluated in Python. For each key-value pair evaluated, the result of the evaluation of the value string is stored as `cc.`*<key>* and is therefore available for subsequent calculations. The dictionaries are evaluated in order of the list. Therefore, evaluations making use certain attributes of the `cc` cache object need to appear in dictionaries further back in the list.

This way the parameter file can be a bit more expressive as to how parameters are calculated. If only the final result was stored in the parameter file, the parameters would be harder to retrace at a later time. Also, changing part of the computation midway would also not be possible. Finally, parameters such as τ_{syn} that appear several times in the parameter file can be defined once and then simply cross-referenced, thereby minimizing the chance for mistakes when parameters are tweaked.

For examples as to how the cache object is used, please see the attached parameter files in Appendix A.1.

6.2.2 Input Generation

SEM makes heavy use of time varying input. Calculating and assigning the rate changes to each input unit is a tedious and time consuming task. SEMf therefore implements input generation in the following form: Each `Pattern` is defined as a set of parameter changes (for Poisson input we only have the rate) for each input unit at predefined times. This allows for patterns that change over time, but it was not used in Chapter 5. Furthermore, for each presentation, a pattern may execute a different set of parameter changes (e.g., different example images for the same digit in MNIST). `Patterns` can be created in bulk using a set of `Creators` (one for MNIST, one for oriented strips, one for the common box scheme etc).

After the patterns have been created, they can be combined with a variety of schemes (uniformly, non-uniformly, sequentially etc.). Each `Combiner` simply states at which time what `Pattern` is presented to the network. These patterns can explicitly be overlapping. Also, the input trace can be augmented using `Injections` that are one time parameter changes introduced at user-specified times. An example would be the constant $10\,\mathrm{Hz}$ background noise in SEM. If patterns – or parameter changes – overlap, the user can chose between different merging strategies (summing the rates up, using the maximum etc.).

Typically, the user just specifies what kind of patterns he wants to have combined in what way. SEMf then takes care of computing the resulting set of rate changes that are given tot he input implementation.

6.2.3 Sweeping Module

To aid in running parameter sweeps, SEMf provides several features to create all parameter sets needed to sweep over several parameters at once. The user simply specifies the source parameter file and what parameter ranges he wants to sweep over thereby naming each parameter file according to the values of the selected parameters. Filters can be specified to exclude certain parameter combinations from the sweep. Each change in parameters is then logged in the generated parameter files so that they are better to comprehend even when looked at after some time has passed.

Parameters can be specified by a full path, such as `network_params/connection_params/eta` or `cache/0/num_z`. Here, on each level, a string indicates a key in a dictionary, whereas a number selects the corresponding element in a list.

6.3 New Library: Spike-based Learning

spike-based learning (sbl) is based on the same principles as SEMf and is still under active development. It aims to implement efficient CD-like learning algorithms (see Section 2.5.1) to be used for long term training for large scale BMs.

It takes care of setting up `cd_connection`-synapses to be used in a subtype of `sbs.BoltzmannMachine` that focusses on drawing single samples from a BM while slowly adjusting its weights. Unfortunately, NEST was not designed for short simulation runs and constant external updates of network parameters (such as weights). The new `cd_connection` (discussed in Section 6.4.10) alleviates this problem.

The real time needed for a single training step in CD-based algorithms with many stochastic LIF neurons is thereby greatly reduced. Exemplary simulations with a 794×1300 *restricted Boltzmann machine* (RBM) showed a reduction per training step from roughly $3\,\mathrm{min}$ to about $4\,\mathrm{s}$, corresponding to a speed-up factor of ~ 45. Therefore, long term investigation of training large scale BMs in NEST are efficient enough to be conducted in a systematic manner.

6.4 Newly developed NEST-models

Over the course of this thesis, most simulations were performed using NEST (see Section 4.2). Most of the concepts investigated in this thesis lie somewhat outside the main aim of NEST. NEST investigates the interactions of large groups of neurons on super-computers where most synapses are static or have a basic form of *spike timing dependent plasticity* (STDP). Poisson background stimuli to the networks are static and dwarfed in number of actual neurons in the network.

Within this thesis, comparatively fewer biological neurons are simulated, but the input the network receives is much more complicated. Also, the STDP mechanisms are a bit more involved. This takes NEST somewhat out of its primary realm of application. Hence, new models needed to be implemented in order to perform the needed simulations in a timely fashion, thereby broadening NEST's applicability to learning tasks.

6.4.1 Challenges with time-varying Poisson noise

When performing a learning task, input to the network is usually rather structured and varying over time. Most importantly, it is vastly more complicated than diffuse background input in "regular" cortical networks (see, e.g., [Petrovici et al., 2014; Breitwieser, 2011]): Each unit in the input layer y_i (see Figure 5.1) changes the rate ν_i with which its underlying Poisson process is generating events several time per biological (simulated) second. This poses a problem for efficient simulation, as the `poisson_generator` model in NEST, implementing [Ahrens and Dieter, 1972], can only be set to one setting for rate, start- and stop-time.

There are two immediate workarounds we can think of: The first workaround is to pause the simulation whenever a Poisson rate changes, jump back into the controlling Python environment, look up and write the new rates to the Poisson generators and continue the simulation. This causes a lot of computational overhead and is, in general, not very feasible and should be avoided. The second workaround is to pre-generate all needed rate changes and then create one `poisson_generator` for every uniquely occurring combination of start-time, stop-time and rate. This is again unfeasible as the number of needed noise-units increases rapidly: For $1000\,\mathrm{s}$ of MNIST images, already more than $160\,000$ units are needed. Still, simulations are typically run for several thousand seconds biological time. Unfortunately, NEST checks all `poisson_generators` for activity in every simulation step, not taking advantage of the fact that, once deactivated, a generator will not reactivate again.

Yet, there is another potential performance bottle neck. The default `poisson_generator` model operates the following way: In every time step, it emits a pseudo-event that is sent to all units it is connected to. For each target i, a Poisson sample is drawn

$$n_i \sim \mathrm{Pois}\left(n \mid \lambda = \nu \cdot \mathrm{d}t\right) \tag{6.1}$$

where $\mathrm{d}t$ is the simulation time step and ν is the average firing rate. If $n \geq 1$ the spike-event is propagated to the target with the corresponding multiplicity n, but if $n = 0$ the event is

discarded. This procedure is performed independently for every target, ensuring that every target receives a unique spike train[1]. In case of short simulation time steps or low Poisson rates, n will be 0 most of the time, resulting in discarded pseudo-events unnecessarily causing overhead by cluttering the message distribution system. To address both short-comings several new NEST models were developed during this thesis.

6.4.2 Poisson generator with varying rates

The first new model, `poisson_generator_var_rate`, has two settings: `rates` and `times`. Both are arrays, whereas the length of `times` is one greater than the length of `times`. The entries in `times` have to be ascending. The i-th entry in `rates` then specifies with which rate the source emits Poisson spikes in the interval defined by the i-th and $(i+1)$-th entry in `times`. During simulation, the model acts like the default `poisson_generator` implementation – meaning that each connected neuron will receive its own unique spike train – but automatically adjusts its firing rate when the appropriate intervals are reached. A rate of $0\,\mathrm{Hz}$ causes the generator to not emit any spikes in the corresponding interval. While this eliminates the need for one generator per unique combination of start-time, stop-time and rate, it does not yet take care of the possible speed-up by eliminating the overhead of discarded pseudo-events.

6.4.3 Multi-Poisson generator with varying rates for sparse input

The next model, `poisson_generator_var_rate_multi`, combines several sources into one. In addition to the `rates` and `times` arrays it is supplied a third, `indices`, of same length as `rates`. The entries in `times` have to be non-descending. All targets start inactive, $\nu_i = 0\,\mathrm{Hz}$. The j-th entries in each of the three arrays encodes a rate change. The target with index $i =$ `indices[j]` is set to rate $\nu_i =$ `rates[j]` at simulation time `times[j]`. The rate remains set until another rate change occurs for this target or the last entry of `times` is reached, indicating when the total activity should cease altogether (all rates are set to $0\,\mathrm{Hz}$).

For simplicity reasons, the user has to specify the `num_output`-setting to inform the generator about how many targets it will be connected to. This way, buffers and internal arrays can be initialized with the proper size accordingly, rather than resizing everything whenever a new target is added.

Upon simulation, in order to avoid emitting too many pseudo-events, in each time step we first draw the total number of spikes n_{tot} from a Poisson distribution:

$$n_{\mathrm{tot}} \sim \mathrm{Pois}\left(n \,\middle|\, \lambda = \mathrm{d}t \cdot \sum_i \nu_i\right) \tag{6.2}$$

[1] If the user wishes to supply identical spike trains to some neurons, he has to connect the `poisson_generator` to an intermediate `parrot_neuron` – a simple model that re-emits all received spikes immediately – which is then connected to the target neurons.

Subsequently, the total number of spikes is distributed among the individual targets according to a multinomial distribution, where each target is weighted according to its current rate ν_i:

$$(n_1, \ldots, n_i, \ldots, n_n) \sim \text{Multinomial}\left(n_1, \ldots, n_n \,\middle|\, n_{\text{tot}}, p_i = \frac{\nu_i}{\sum_j \nu_j} \right) \tag{6.3}$$

For implementation details, please see below. Only targets receiving spikes in the current time step ($n_i > 0$) are then notified. This bypasses the communication infrastructure in NEST altogether, but is necessary because NEST has no inherent concept of directed messages. Again, this makes sense in its primary realm of application – large scale simulations on super-computers – but is a minor hindrance in our case. Plus, we only draw from a Poisson distribution once instead of N times.

6.4.4 GSL-based random Device for multinomial Distributions

In order to sample from multinomial distributions as required by the new Poisson models, a new random device was added to the `random` sub-library of NEST, `GSL_MultinomialRandomDev`. Its implementation is based on *GNU Scientific Library* (GSL).

We want to sample from the following distribution

$$(n_1, \ldots, n_i, \ldots, n_n) \sim \text{Multinomial}\left(n_1, \ldots, n_n \,\middle|\, n_{\text{tot}}, p_i = \frac{\nu_i}{\sum_j \nu_j} \right) \tag{6.4}$$

The user specifies the total number of targets N as well as initial weight for each, ν_i. These weights do not need to be normalized. First each target i is associated with a renormalized binomial probability $\tilde{p}_i$:

$$\tilde{p}_i = \frac{\nu_i}{\sum_{j=1}^{N} \nu_j - \sum_{j=1}^{i-1} p_j} \tag{6.5}$$

$\tilde{p}_i$ denotes the probability of target i receiving an event, given that there are only $N - i + 1$ other targets left (all targets with an index lower than i being dismissed).

The number of events n_i each target receives is hence implemented as

$$n_i \sim \text{Binomial}\left(n \,\middle|\, \tilde{n}_i := n_{\text{tot}} - \sum_{j=0}^{i-1} n_j, p_i \right) = \binom{\tilde{n}_i}{n} p_i^n \, (1 - p_i)^{\tilde{n}_i - n} \tag{6.6}$$

Since we can stop the distribution process as soon as all events are allotted, the weights are inversely sorted prior to drawing. This way, the targets most likely to receive many events are handled first, thereby maximizing the chance of stopping early because all events have already been distributed, while not changing the sampling statistics in any way.

6.4.5 Lookahead for sparse Input

If saving memory is not of the utmost importance, one can increase the performance of the Poisson generator even more: Instead of only computing how many spikes occur in the current time step only, we can perform a lookahead, meaning that we pre-compute and store all spikes occurring in a longer time interval T_L ($\sim 1\,\mathrm{s}$ biological time).

These new models, `poisson_generator_var_rate_multi_lookahead` and `poisson_generator_var_rate_lookahead`, are based on models presented so far, but additionally takes a setting `steps_lookahead`, specifying for how many steps the lookahead should be performed.

The approach remains mostly unchained. The only difference to the generators without lookahead is that we draw the total number of events as follows

$$n_{\mathrm{tot}L} \sim \mathrm{Pois}\left(n \,\middle|\, \lambda = T_L \cdot \sum_i \nu_i\right) \tag{6.7}$$

Additionally, after distributing the spike events among the targets, the actual spike times have to be drawn for each target. Since all Poisson spike times are independent, we can simply draw them from a uniform distribution over the current lookahead interval.

Finally, the resulting spike times are sorted and inserted into a queue. Whenever the current simulation time step matches the spike time of the next spike, it is taken from the queue and sent to the corresponding target. If the simulation time advances beyond the current lookahead interval, the pre-computation is repeated. Any rate changes during the lookahead interval are automatically taken into account.

6.4.6 Support for Multithreading and Multiprocessing

The final model, `poisson_generator_var_rate_multi_lookahead_mt`, adds support for multi-threading. The only setting differing from the previously described models is that instead of just specifying the number of targets the source is connected to, the user needs to supply `all_gids`, the list of *global IDs* (GIDs) of all targets the generator is connected to globally. NEST uses GIDs to uniquely identify each node in the network.

Since the multinomial Poisson sources generate specific spikes for each of their targets, they need the ability to send directed spike events. This is achieved by storing a pointer to each target unit and directly calling the event-handling method in case of a spike event, bypassing the connection manager. In a multi threading/processing environment the situation is a bit more complicated. Each virtual process (NEST's generalized name for threads and processes) operates on a set of nodes. Devices like Poisson generators – nodes that do not partake in regular network dynamic but rather inject into or record from the network – are instantiated on every virtual process. By using `all_gids`, each instance of the Poisson source model on every virtual process is able to find out which targets it is connected to locally. This allows

for a translation between the global `indices` array and the local target indices. Each instance is only pre-computing spikes for its local targets while ignoring all other rate changes.

For the user, this is completely transparent. The model can be treated the same, independent of the fact that the simulation is performed single or multi threaded or even multi processed.

6.4.7 Benchmark: Poisson Generators

In order to measure the speed-up of the newly deveoloped multinomial Poisson generators, a series of synthetic tests was set up. In each test, we set up a series of Poisson sources firing onto a population of `parrot_neurons` – a simple model that re-emits all received spikes immediately. Each `parrot_neuron` receives its own independent Poisson spike train. This is done to keep the influence of possible neural dynamics in the targets at a minimum. During network execution we only measure the time it takes for the simulation to be conducted, *not* how long it takes to set up. We do include – for obvious reasons – the lookahead-pre-computation of spikes in the corresponding source model. Each benchmark (i.e., each figure) was conducted in whole on one simulating node of the *Human Brain Project* (HBP) high performance cluster. In order to avoid possible differences due to slightly different execution speeds of the nodes, we always compute the relative speed-up for all simulations conducted on that node. For more stable results, each network setup is performed several times ($\sim$100 times in most cases) and only the smallest execution time taken into account. The speed-up factor is then simply the ratio of execution times for default and new implementation.

Firstly, we look at the best-case in terms of the default Poisson model: All targets receive Poisson spikes with the same rate ν so we only have to instantiate `poisson_generator` once.

In Figure 6.1, we compare execution times for different firing rates. A fixed number of targets $n_{\text{targets}} = 500$ is supplied with Poisson noise of varying rates between 1–$1000\,\text{Hz}$. As we can see from the plot, both the model without and with lookahead perform better than the default implementation for low rates. This is due to the fact that – as outlined above – we manage to avoid sending unnecessary pseudo-events. Still, without lookahead the main source of speed-up is when no target receives spikes and hence we have to perform neither multinomial distribution nor delivery of spike-events. On average the ratio of time steps with no spike is:

$$p(n_{\text{tot}} = 0) = \left[\text{Pois}\left(0 \mid \lambda = \nu \cdot \mathrm{d}t\right)\right]^{n_{\text{targets}}} = e^{-n_{\text{targets}} \cdot \nu \cdot \mathrm{d}t} \tag{6.8}$$

Hence, when the rate increases events have to be distributed more often which at some point becomes computationally even more expensive than the default implementation. When performing lookahead, we can maintain a high speed-up factor for higher rates since we sample less from both Poisson and multinomial distributions. Furthermore, since there are always relatively many events per lookahead interval, the computational costs of the multinomial distribution are independent of the total number of events past a certain threshold. If the rate is increased even further, though, more and more spike times have to be drawn from uniform distributions as well as sorted, diminishing the total speed-up achievable as there are relatively less pseudo-events that can be avoided.

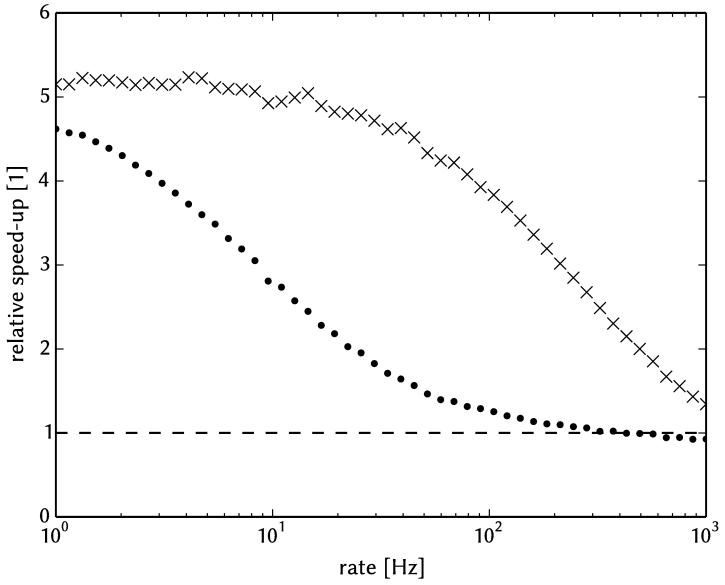

Figure 6.1: Speed-up of the newly implemented multinomial Poisson sources both without (dots) and with (crosses) lookahead of $10\,000$ steps. A single generator of each model is set to varying firing rates and connected to 500 targets. Each target receives a unique Poisson spike train. Each network setup is simulated 100 times for 40 biological seconds (time step $0.1\,\mathrm{ms}$) and the minimal execution time is is noted. The speed-up is then measured as the ratio of execution times of the default Poisson generator implementation (dashed line) and each of the two multinomial models.
As we can see from the plot, the lower the firing rate of the generated Poisson spike trains, the more efficient the multinomial Poisson spike generation is. The implementation without lookahead (dots) decreases in performance for much lower rates whereas the implementation with lookahead (crosses) is able to maintain an almost constant speed-up for larger rates.
This is due to the fact that the implementation without lookahead draws its main speed-up from not distributing any spikes in case all targets do not receive spikes in a time step. If spikes need to be send, the multinomial distribution has to iterate over almost all targets to distribute only a few spikes. The implementation with lookahead (crosses) avoids this by only drawing only once from the total Poisson distribution and once from the multinomial per lookahead interval. After the number of spikes per target in the lookahead interval has been determined, the spike times are uniformly drawn and sorted. As the rate increased, this last part becomes more and more computationally expensive, as more spike times need to be drawn and sorted, resulting in a diminished speed-up. See the text for details.

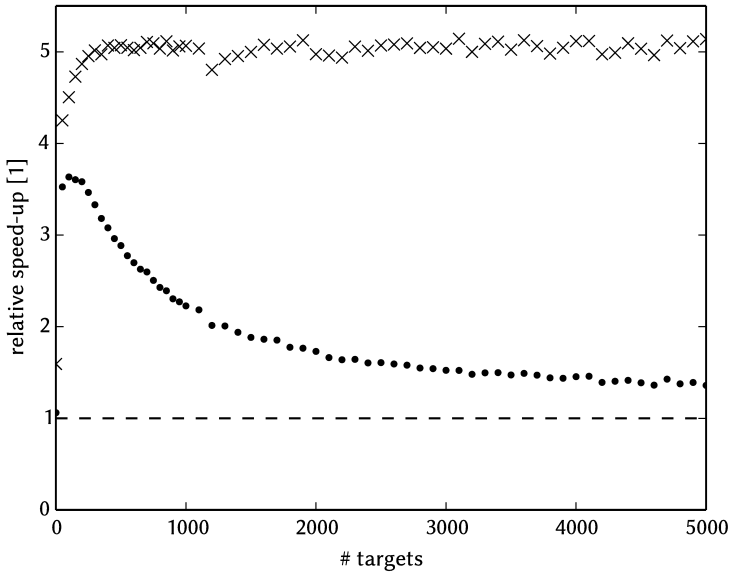

Figure 6.2: Speed-up of the newly implemented multinomial Poisson sources both without (dots) and with (crosses) lookahead of $10\,000$ steps. A single generator of each model is set to $10\,\mathrm{Hz}$ firing rate and connected to varying numbers of targets. Each target receives a unique Poisson spike train. Each network setup is simulated 100 times for 300 biological seconds (time step $0.1\,\mathrm{ms}$) and the minimal execution time is is noted. The speed-up is then measured as the ratio of execution times of the default Poisson generator implementation (dashed line) and each of the two multinomial models.
As we can see from the plot, the implementation without lookahead (dots) approximates the default implementation (dashed line) whereas the implementation with lookahead (crosses) maintains a constant speed-up factor irrespective of the number of targets.
This is due to the fact that without lookahead we need to draw from the multinomial distribution in every time step where at least one target receives a spike, causing additional overhead. With lookahead, on the other hand, we only draw once from both total Poisson as well as multinomial distribution per lookahead interval and temporarily store the input spikes in memory, thereby maintaining a constant speed-up factor. See the text for details.

In Figure 6.2, we repeat the previous experiment, only this time for different numbers of targets while keeping the firing rate fixed ($\nu = 10\,\mathrm{Hz}$). We see a slightly different picture: While without lookahead the speed-up decreases as the number of targets increases (again mainly due to Equation (6.8)), it manages to outperform the default implementation, albeit only slightly. With lookahead, however, the speed-up remains constant, independent of the number of targets. This is due to the fact that the ratio of avoidable pseudo-events remains unchanged when adding new targets with the same rate.

In Figure 6.3, we investigate the lookahead interval. Here we fix the number of targets ($n_{\text{targets}} = 500$) and the firing rate ($\nu = 10\,\mathrm{Hz}$) and vary the number of steps we perform lookahead for at once. We see that the speed-up factor is diminished for small lookup intervals since we approach the case without lookup. It plateaus as soon as the average number of events per target get reasonably large (> 0.1). Beyond that point generating and reordering

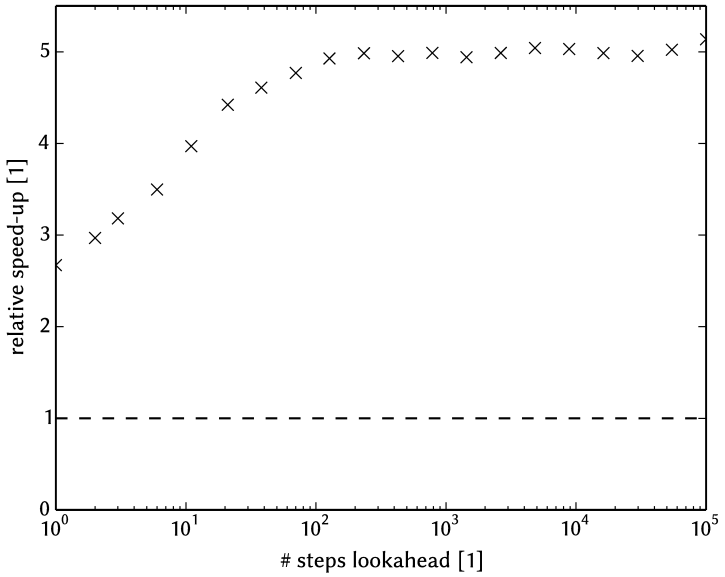

Figure 6.3: Speed-up of the newly implemented multinomial Poisson sources with lookahead of varying step numbers. A single generator of both the default Poisson generator model as well as the sparse multinomial one is set to a firing rate of $10\,\mathrm{Hz}$ and connected to 500 targets. Each target receives a unique Poisson spike train. Each network setup is simulated 100 times for 300 biological seconds (time step $0.1\,\mathrm{ms}$) and the minimal execution time is noted. The speed-up is then measured as the ratio of execution times of the default Poisson generator implementation (dashed line) and the multinomial model with lookahead. As we can see from the plot, the speed-up increases as the number of lookahead increases. This is due to the fact that the larger the lookup interval the fewer times we have to draw from both total Poisson as well as the multinomial distribution. If we increase the lookahead interval even further, the speed-up reaches a plateau: The longer the lookahead interval, the more spikes will fall into it given a constant Poisson rate. The sorting and storing of these spikes then counteracts the benefits of drawing fewer samples from Poisson and multinomial distributions. See the text for more details.

the spike-times is as computationally expensive as the savings obtained by performing less Poisson and multinomial lookups. This shows that the memory overhead per supplied target is not too much if the lookahead interval size is chosen accordingly.

Secondly, in Figure 6.4 we look at the worst-case in terms of the default Poisson model: We have a set of n targets, each receiving a different rate from $1\,\mathrm{Hz}$ for the first and n Hz for the last. Since each instance of the default generator can only spike with a single rate, we need n instances to supply all targets, whereas we still only need one instance for the newly developed models. This has a dramatic effect on the achieved speed-ups, since now – in the default model's case – more nodes need to be queried for updates which causes additional overhead. Since the average rate in the network still increases linearly with the number of neurons[2], the overall speed-up decreases since the sparseness is reduced (see above).

[2] In this benchmark, the average rate for n neurons with rates from $1\,\mathrm{Hz}$ to $n\,\mathrm{Hz}$ is $\langle\nu\rangle = \frac{1}{2}\,(n+1)\,\mathrm{Hz}$.

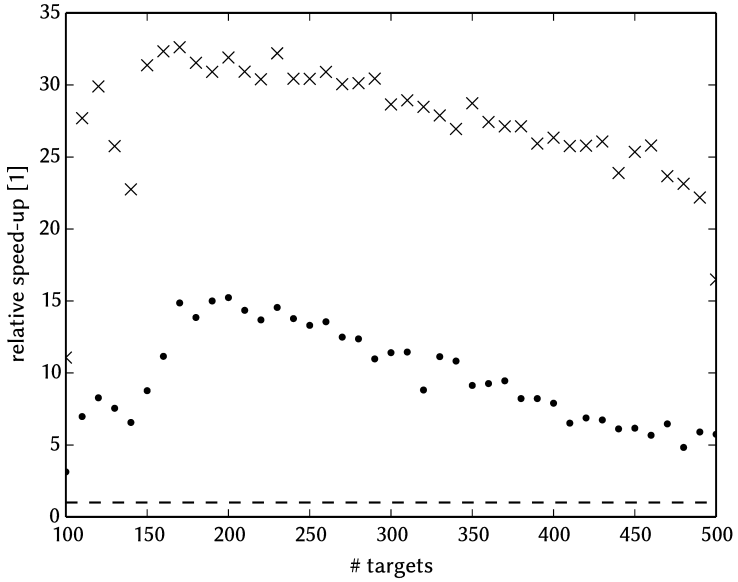

Figure 6.4: Speed-up of the newly implemented multinomial Poisson sources both without (dots) and with (crosses) lookahead of $10\,000$ steps when simulating several sources with different rates. We simulate n Poisson sources with rates ranging from $1\,\mathrm{Hz}$ till n Hz. Since the default model can only fire with one rate, we need to create n generators, whereas the multinomial Poisson sources emulate all n sources in one instance. Each network setup is simulated 100 times for 40 biological seconds (time step $0.1\,\mathrm{ms}$) and the minimal execution time is noted. The speed-up is then measured as the ratio of execution times of the default Poisson generator implementation (dashed line) and each of the two multinomial models. We see a much greater speed-up than for one fixed rate due to the fact that we now need more than one instance of the default model. Yet again, as the rates increase, the speed-up decreases because the input becomes less sparse and therefore there are less pseudo-events that could be avoided. We see that by pre-computing spikes in a lookahead window, we achieve much greater speed-ups. See the text for more details.

Third, in Figure 6.5, we investigate the speed-up when the rates are not fixed but change over time. We therefore take a set of rate courses generated from the input generation module of SEMf (as described in Section 6.2.2) realize both with the default generator model where we create one instance for each tuple of start-time, stop-time and rate and the newly implemented source models. The input itself consists of images from the MNIST database [LeCun and Cortes, 1998]. Each of the $28 \times 28 = 784$ pixels is represented by an input source encoding its intensity between 0 and 255 as a firing rate between 10–$100\,\mathrm{Hz}$. Every $0.5\,\mathrm{s}$, a new image is presented for $0.4\,\mathrm{s}$ after which there is a pause for $0.1\,\mathrm{s}$ in which the input sources all fire with $10\,\mathrm{Hz}$. The duration for which the rate courses are generated is varied. We see that the longer the duration for which we pre-generate input the greater the speed-up. This is due to the fact that the number of default generator instances increases with the amount of tuples of start-time, stop-time and rate. Unfortunately, the scheduler in NEST does not make use of the fact that disabled devices will not reactivate again during the same simulation run and thus can be eliminated from scheduling. Sorting all devices by start and stop time and storing them

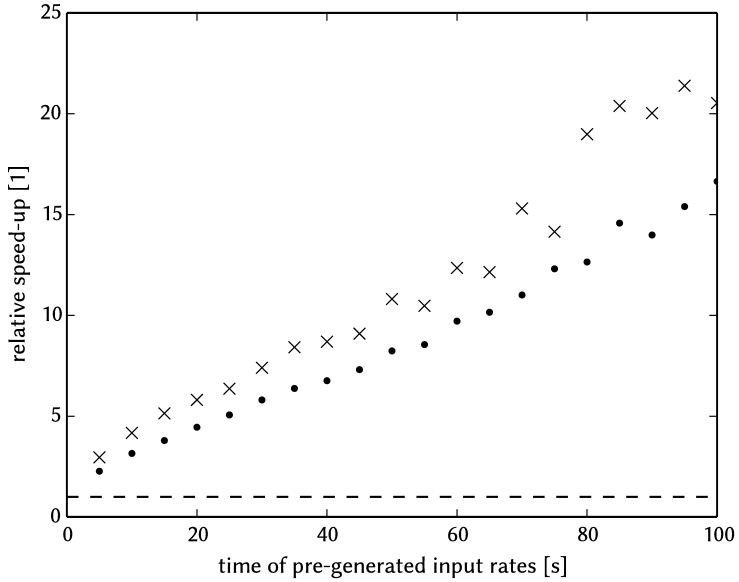

Figure 6.5: Speed-up of the newly implemented multinomial Poisson sources both without (dots) and with (crosses) lookahead of $10\,000$ steps when generating MNIST images. Each of the 784 targets encodes the pixel intensity between 0 and 255 as a rate between 10 and $100\,\mathrm{Hz}$. Every $0.5\,\mathrm{s}$ a new image is encoded in the firing rates for $0.4\,\mathrm{s}$, after which all targets fire with $10\,\mathrm{Hz}$ background rate only for $0.1\,\mathrm{s}$. The rate changes for all targets are precomputed – as explained in Section 6.2.2 – for varying biological durations. Each network setup is simulated 100 times (time step $0.1\,\mathrm{ms}$) and the minimal execution time is noted. The speed-up is then measured as the ratio of execution times of the default Poisson generator implementation (dashed line) and each of the two multinomial models. Since the default Poisson generators can only store a single start, stop and rate value, each unique combination of those three values has to be realized by a single Poisson generator. This causes unnecessarily many generators to be created, slowing down execution times considerably. The sparse multinomial Poisson sources store all rate changes by the individual targets and change their rates on the fly during the simulation without any callbacks to the Python interpreter controlling the simulation. This results in a much greater speed-up than previously and makes network simulations with sophisticated input characteristics feasible.

in a queue could improve the performance as only the next activating device would need to be checked per time step. In actual simulations (see Chapter 5), input is pre-generated and sent to the sources every $500\,\mathrm{s}$ of biological time.

source model	# generators	execution time	speed-up
multinomial model with lookahead	1	3 min 14 s	–
one default model per source	784	23 min 55 s	~ 7.4
pre-generated input with default model	160 147	13 h 28 min 4 s	~ 250

Table 6.1: Exemplary comparison of actual execution time of full SEM-like simulations – discussed in Chapter 5. A small network with three samplers (to minimize the influence of other parts in the network) is receiving MNIST input for $1000\,\mathrm{s}$ biological time. In the first case we have the one instance of the multinomial Poisson source model emulating all 784 units in the input layer. In the second case we have one NEST default Poisson generator per unit in the input layer, but stop the simulation every $0.5\,\mathrm{s}$ of biological times and change their rates accordingly. In the last – and worst – case we pre-generate the complete input for the full duration and create one default Poisson generator for each unique set of start-time, stop-time and rate. Each simulation was only run a single time and is therefore only a rough indication of speed. From the execution time it is clear that the last option is completely unfeasible, while stopping the simulation and resetting the rates every $0.5\,\mathrm{s}$ leaves much room for improvement. Please also note that usually simulations are run for several thousand seconds biological time. See the text for details.

Lastly, we do an exemplary comparison by executing the same simulation network with three different source implementations. The three source implementations are:

- the new multinomial source model with lookahead
- one default Poisson generator per input neuron the rate of which is changed from the controlling Python process whenever a new pattern is presented
- one default Poisson generator is instantiated for every tuple of start-time, stop-time and rate from the pre-generated rate courses ($160\,147$ in total)

We use a small network consisting of three sampling neurons, receiving MNIST input that has been clipped to 26×26 image size. For simplicity, there is no pause between different images, so that input rates are constant for $0.5\,\mathrm{s}$ biological time. The network is run for $1000\,\mathrm{s}$ biological time in total. Please note that regular simulations are run for a multitude of that. The full parameters can be found in Appendix A.1.1. We once again see that implementing varying input rates as a plethora of default Poisson generators is highly infeasible. Also, manually[3] updating all rates from the controlling Python process is slower than the newly developed models. Furthermore, from a developer's standpoint, they are easier to deal with because they can be set once and then simply supply their targets with Poisson noise with no need for stopping the simulation specifically because of them.

[3]This involves stopping the NEST run, switching back to the controlling Python context, adjusting the rates via Python-calls and restarting the NEST run.

6.4.8 Spike-based homeostasis synapses

For spike-based homeostasis, two new synapses have been developed: `stdp_homeostasis_linear` and `stdp_homeostasis_linear_modulating`. Both implement the update rule discussed in Section 5.2. The latter additionally implements a time varying learning rate that oscillates in log-space (not shown in Chapter 5).

6.4.9 SEM-like synapses

For SEM, each of the adjustments discussed in Section 5.3 was implemented in its own synapse model. Each model computes the weight change in the theoretical regime for easier read out and analysis. The actual biological weight is then set by multiplication with the set conversion factor $f_{\text{theo}\to\text{bio}}$ (see Equation (2.110)).

6.4.10 CD-based synapses

When implementing any kind of CD-based algorithm (see Section 2.5.1), one generally has to draw a set of samples from the BM/RBM and then update the synaptic weights according to update rules such as Equation (2.132). When going to large-scale BMs we encounter a performance bottle-neck: The Cython-based interface for NEST was not designed for such rapid parameter updates. Hence, for RBMs on the order of 784×1300 (for learning MNIST) we have $1\,019\,200 \times 2$ synapses in the network. Writing a new weight into all of those takes a relatively long time (close to 15–$20\,\mathrm{s}$ on the currently used machines). Since this operation has to be performed at least once for every training epoch, and we have tens of thousands of training epochs, this severally hinders our ability to investigate CD-learning at this network size.

The `cd_connection`-synapse was implemented to alleviate this problem. It is based on the simple fact that the sample-based CD-like weight update Equation (2.132) is split into four factors and the learning rate:

$$\Delta W = \eta \cdot (d_{\text{pre}}\, d_{\text{post}} - m_{\text{pre}}\, m_{\text{post}}) \tag{6.9}$$

where d_{pre}, d_{post} denote the data-term samples of the pre- and post-synaptic neuron and m_{pre}, m_{post} the model-term samples correspondingly. Symmetry in the products ensures that both synapses between the same neurons perform the same weight update and their BM-weight therefore stays symmetric.

The samples can be computed in the parent Python process that is controlling the simulation. All update factors are then written from Python to a memory-mapped buffer from which they will be read by all synapses during simulation. This effectively sets the time needed to write the weights to the network to zero so that only the simulation time per step (a few seconds) remains.

Simulations that took weeks before can now be conducted in under a day.

6.4.11 Periodic Generator

In principle, if we want to generate regular spike trains we have two options in NEST: We can pre-generate a regular spike train and load it into a `spike_generator` instance. Especially for long simulations, this is very memory consuming. Alternatively, we can configure a biological realistic neuron model to spike regularly (e.g., by injecting current or setting the reversal potential above the firing threshold), but this is not straightforward (we have to translate our desired *inter-spike interval* (ISI) into corresponding neuron parameters, i.e., current strength) and we needlessly compute membrane dynamics we are not interested in, wasting computational capacities.

The `periodic_generator` is therefore a very simple model to generate regular spike trains. Apart from the `start` and `stop` times that are common for all NEST devices, the user specifies both the `isi` as well as `offset`. The generator then emits spikes at the following time steps:

$$t^{\text{spike}} = \texttt{offset} + \texttt{isi} \cdot i \qquad i = 0, 1, 2, \ldots \tag{6.10}$$

All emitted spikes have a fixed ISI, while specifying the offset allows the user to shift the whole spike train by an arbitrary amount.

6.4.12 Selective Parrot Neuron

The `selective_parrot_neuron` behaves almost exactly like the regular `parrot_neuron` already present in NEST with one important exception: It only re-emits spike-events with a weight different from zero. This allows for the investigation of synapse dynamics in fixed environments. Most synapse models developed in this thesis (see Section 6.4.9 and Section 5.3) evolve a theoretical weight value that, upon spike transmission, is multiplied with the conversion factor (see Section 2.4.4 and Section 6.1.1) to obtain its biological equivalent. This has two distinct advantages: Firstly, the theoretical weight value is often more intuitive for the experimenter to interpret. Secondly, by setting the conversion factor to zero, the resulting biological weight will be zero as well, irrespectively of how the theoretical weight evolves. This means a selective parrot neuron as post-synaptic neuron does not re-emit the transmitted spike-event while a regular parrot neuron does. As in the latter case the post-synaptic neuron fires synchronously with the pre-synaptic neuron, no STDP dynamics can be investigated.

By then connecting two selective parrot neurons via the synapse model to be investigated, we can impose any firing dynamics onto the two nodes by stimulating them with non-zero weighted spike-events. The synapse then evolves according to the spike history of the two nodes, but none of the transmitted spikes elicit additional post-synaptic spikes which would disturb the STDP mechanics as these spikes would appear in the spike history of the post-synaptic parrot neuron.

6.4.13 Last Spike Detector

The `last_spike_detector` is a simpler variant of the existing `spike_detector`. For each neuron connected to it, the time of its latest spike is recorded. This is useful whenever we are not interested in the network's complete spike history (which would be recorded by the default `spike_detector`) but rather in the state the network was encoding at the particular time the simulation was halted.

When read out, the `last_spike_detector` returns both a `times` as well as an `indices` array. The i-th entry in both arrays denote both latest spike time and global id (uniquely identifying a node in NEST) of the i-th afferent neuron. By comparing the time distance between latest spike times and current simulation time to the length of an active state τ_{on} we can immediately compute the binary state encoded by the network. This is especially useful in CD-like learning (see Sections 2.5.1, 6.1 and 6.3) as here we perform many small simulation steps after which we are only interested in the current binary state of the simulation and not the entire spike history.

6.5 Source Code

All source code can be found in several Git repositories at:

`https://gitviz.kip.uni-heidelberg.de`

sbs

The source code to sbs can be found in the `code/v2/sbs` subdirectory of the `model-nmsampling` repository. It can be installed via the provided `setup.py` script.

SEMf

The source code to SEMf can be found in the `src/combined` subdirectory of the `model-sem` repository. It can be installed via the provided `waf` install script.

sbl

The source code to sbl be found in the `code/v2/sbl` subdirectory of the `model-nmsampling` repository. It can be installed via the provided `setup.py` script.

Discussion

In this thesis, we demonstrate the fundamental feasibility of NSEM, a neuromorphic implementation of *spike-based expectation maximization* (SEM) with stochastic *leaky integrate-and-fire* (LIF) neurons on state-of-the-art neuromorphic hardware. In particular, we use LIF neurons with exponential synapses, which are a de-facto standard for neuromorphic devices, as well as double-exponential *spike timing dependent plasticity* (STDP). Furthermore, we study the effects of a finite weight resolution and of the specific implementation of STDP in the *Neuromorphic Physical Model System 1* (NM-PM1) system.

We use a so-called cause layer of mutually inhibiting neurons to detect "hidden" causes (i.e., salient features) in the spike patterns emitted by a forward projecting input layer. Following [Nessler et al., 2013], we show that by using a specific *spike timing dependent plasticity* (STDP) rule for the dynamic synapses between input and cause layer, the network dynamics can be understood as a form of online *expectation maximization* (EM), a general class of unsupervised *maximum likelihood* (ML) learning algorithms (Section 2.5.3). After training, each cause layer neuron is self-tuned to respond to a specific input label. The receptive fields then encode the activity pattern of the input layer most likely to be responsible for each cause layer neuron's activation.

Building upon [Petrovici et al., 2013], we implement NSEM with a cause layer comprised of stochastic LIF neurons. As it is difficult for synapses on the NM-PM1 to switch reversal potentials during ongoing emulation and are therefore set to be excitatory only, the cause layer neurons' individual excitabilities have to be moderated during training. Since adjusting neuron parameters at runtime is unfeasible, we implement a spike-based version of homeostasis [Habenschuss et al., 2012] (described in Sections 2.5.4 and 5.2). We show that even with a simple update rule – suitable for closed-loop implementation – we are able to maintain the activity of each cause layer neuron at a pre-defined target value.

The dynamic homeostasis-implementing synapses are driven by background generators. In Section 5.4.4, we demonstrate that, at comparatively lower firing rates, a regular background spike train is preferable to Poisson noise as it is able to better approximate a constant bias current.

Due to technical constraints regarding the implementation of STDP in the *Neuromorphic Physical Model System 1* (NM-PM1), the original weight update rule is not amenable to a straightforward implementation. We therefore addressed each constraint in order and modified the original weight update rule as needed.

The original update rule performs a weight update for every post-synaptic spike, whereas on the NM-PM1 we are only able to operate on spike pairs. This especially means that we cannot perform weight updates if the pre-synaptic neuron is silent. However, since homeostasis fixes

the average activity of each cause layer neuron, we are able to incorporate this information into an adjusted update rule for causal spike pairs only (see Section 5.3.1).

Furthermore, for each post-synaptic spike we effectively only "see" the eligibility trace of latest pre-synaptic spike as opposed to all pre-synaptic spikes. This influenced the underlying generative model that was formulated for full eligibility traces. As we observe less pre-synaptic activity, the resulting weight updates as well as the learnt weights turn out to be smaller. Conversely, the receptive field of each cause layer neuron then corresponds to seemingly lower activity of the input layer. As shown in Section 5.3.2, these influences are predictable and can thus be compensated for to a limited degree, thereby almost restoring the weight distribution in the network.

On the NM-PM1, STDP-based weight updates only happen periodically and are then based on accumulated eligibility traces. We show in Section 5.3.3 that as long as the causal spike pair correlation information can be stored sufficiently (i.e., it is not altered, either by decaying or reaching a maximum value), the update rule is not sensitive to the duration of the update period.

Finally, we showed in Section 5.3.4 how the limited weight resolution of 4–6 bit in neuromorphic systems can be circumvented by employing stochastic weight updates.

The learning ability of the final NSEM model is demonstrated in Section 5.4.2. Here, we illustrate the cause layer neurons' ability to learn classes of a simple input pattern – a randomly oriented bar. Moving to more complicated learning problems, in Section 5.4.3 we show that the network is able to learn and distinguish between all digits in a modified subset of the MNIST dataset, even at 4-bit weight resolution. In order to achieve this goal, we have to adjust the minimally inferable rate of the underlying generative model, corresponding to an increase in contrast in the learnt receptive fields. When used as a discriminative model, the post-learning classification performance on the reduced MNIST dataset approached $94\,\%$ with only three cause layer neurons and 4-bit weight resolution.

In Section 5.4.5, we briefly explore the possibilities of NSEM in larger networks. Here we observe that each cause layer neuron codes for a smaller part of the input space (i.e., fewer orientations in the randomly oriented bar example). Furthermore, we present first steps to learn the full MNIST dataset as in larger networks we are able to account for more variations in the presented input images. Due to the relatively long simulation times needed for these types of networks, emulation in neuromorphic hardware would be primarily useful here.

A major concern for the successful implementation of NSEM in a neuromorphic environment are spike transmission delays. The only time critical spike signals in the entire network are the ones to and from the inhibitory population facilitating the *winner-take-all* (WTA)-structure of the cause layer. The inhibitory signal is needed to discourage several cause neurons from spiking and thereby learning from the same input. The longer it takes to arrive, the more likely it is that several cause layer neurons spike and thus learn overlapping regions of the input space. In Section 5.4.6, we show that long spike transmission delays in the order of several milliseconds diminish the classification performance of the network.

The entire NSEM framework was implemented as a set of interacting *NEural Simulation Tool* (NEST) modules. Therefore, NSEM is available to be combined with other models or integrated into larger network motifs. Furthermore, the software packages developed during this thesis (described in Chapter 6) were designed with extensibility in mind, so that the next steps, outlined below, can be carried out with minimal effort.

Outlook

With the fundamental feasibility of a neuromorphic implementation of SEM – denoted as NSEM – demonstrated in this thesis, we have taken the first steps towards wafer-scale unsupervised learning.

In the immediate future we need to verify the applicability of spike-based homeostasis implemented in a real closed loop setup. This study can be conducted independently of other concepts such as LIF sampling or learning. The setup is simple: The activity of a set of intrinsically firing neurons (either by current injection or background stimulation) is moderated via externally injected spike trains. The needed rate and weight of such spike trains is computed on a host computer running in parallel. These experiments may serve as relatively low-demand benchmarks for the ability of the NM-PM1 to operate in a closed-loop environment as it evaluates key aspects such as available bandwidth for spike readout/injection as well as the permissible model complexity for real-time execution on the host computer.

In parallel, the concept of spike-based homeostasis may be studied further. As was demonstrated in this thesis, a constant homeostatic learning rate for the homeostasis has to balance between two extremes: If it is too small, cause layer neurons with a temporarily high activity cannot be sufficiently moderated as their synaptic input from the input layer increases faster than the homeostatic inhibition. Due to the nature of the update rule, weights can not grow arbitrarily large. Therefore homeostasis is able to overtake the synaptic input in strength and moderate the activity of the "runaway" neurons. However, a lot of learning time is wasted and sometimes the network state remains unrecoverable. If the homeostatic learning rate is too high, however, even slight differences in the activity of cause layer neurons – occurring on smaller time scales when the receptive fields start to differentiate – are immediately balanced out. The neurons are unable to learn distinct input patterns and instead are driven to code for a superposition of all input patterns. A new approach would be to dynamically adjust the homeostatic learning rate during training in the same spirit as the variance tracking approach in [Nessler et al., 2008]. By taking into account how much we needed to adjust the effective biases in recent history, we in- or decrease the homeostatic learning rate during simulation. This would suppress "runaway" behavior of cause layer neurons while still allowing for differences in activity on smaller time scales that are needed for successful NSEM.

As discussed in Section 2.5.3, the formalism of SEM was recently shown to be compatible to all distributions in the natural exponential family [Bill et al., 2015]. As we demonstrated, the underlying generative model used in this thesis – modelling the input as samples from Poisson distributions – is disturbed primarily by the nearest-neighbor spike pair correlation information. NSEM should therefore include modelling the input as Bernoulli processes. In this adjusted generative model, the theoretically optimal weight update takes into account the latest pre-synaptic spike only, resulting in a more faithful representation in hardware.

The renewing synapses required by this model can be realized via the *short term plasticity* (STP) mechanisms available in hardware (discussed in Section 2.4.4) as done in [Petrovici et al., 2013, 2014].

Further studies on the effect of parameter mismatch are also needed. We need to investigate accuracy of the correlation capacitors in the synapse, the decay of the correlation information charge due to leakage currents, fixed pattern noise in synaptic weights and neuron parameters or a non-uniform delay structure in the network. These aspects are likely to have an impact on the classification performance and thus should be investigated in future work.

Finally, we can also expand SEM to larger networks, such as presented in [Bill et al., 2015], where cause layer neurons compete to be local experts regarding hidden causes in small patches of the complete input layer. Sparse excitatory interconnections then allow similarly tuned neurons to integrate information beyond the receptive field of a single unit.

Another such possibility would be – similar to deep learning in *restricted Boltzmann machines* (RBMs) [Hinton, 2010] or *convolutional neural networks* (CNNs) [Lecun et al., 1998] – to stack multiple layers with several WTA-like structures on top of each other. While the neurons in the lowest-level learn local salient features in the input, higher-level neurons integrate this information to detect higher-order features which has the potential to significantly improve the classification capabilities of NSEM. The *spike-based expectation maximization framework* (SEMf) (discussed in Section 6.2) already has limited support for multilayer architectures built in, although it was not presented in this thesis.

Since spike transmission delays were identified as one of the major concerns for successful NSEM learning, strategies for compensation have to be developed. One possible approach is to reduce target activities of the cause layer neurons via spike-based homeostasis. This way, it is less likely for several cause layer neurons to spike simultaneously, potentially making the network more resistant to transmission delays. As learning only occurs when neurons in the cause layer spike, reducing their activity corresponds to a reduction in learning speed. The proposed method would therefore have the downside of prolonging simulations times which is less of a concern on neuromorphic hardware.

Evidently, the ultimate goal is to implement NSEM on neuromorphic hardware, in particular on the NM-PM1. Firstly, its speed-up factor due to physical emulation of neural dynamics in analog circuitry will shorten the time needed for training considerably. Secondly, its inherent parallelism make it ideal for larger integrated network structures. Network sizes are in the order of thousands of units for state-of-the-art software implementations for learning handwritten digits [Lecun et al., 1998; Deng and Yu, 2011; Cireşan et al., 2012]. These can be implemented on single wafer with no increase in simulation time.

When going to potentially even larger problems, the aforementioned multi-layer architecture could be spread among sets of several interconnected wafer modules. Since delays are only relevant within each small WTA-circuit, we have to ensure that each local learning unit is located on the same wafer. Furthermore, since each cause layer sparsely encodes its observed input, we do not need a lot of communication bandwidth between the wafer modules. This allows for unsupervised learning in an online fashion at very large scales, rivaling state-of-the-art software implementations.

Finally, the training and operation of autonomously acting agents, such as self-driving cars or flying drones, has become more relevant in recent years [Kim et al., 2013]. These mobile agents are limited in how much energy they are able to store. Since neuromorphic hardware is more energy efficient than conventional computing architectures, it is a prime candidate to serve as computational substrate for these agents. Of course, a substrate by itself is not enough. Models such as NSEM, employing the core ideas discussed in this thesis (stochastic sampling, Bayesian inference and unsupervised learning), would thus bolster and profit from the development of neuromorphic computing.

Bibliography

Ackley, D. H., Hinton, G. E., and Sejnowski, T. J. (1985). A learning algorithm for Boltzmann machines. *Cognitive Science*, 9:147–169.

Ahrens, J. H. and Dieter, U. (1972). Computer methods for sampling from the exponential and normal distributions. *Commun. ACM*, 15(10):873–882.

Alais, D. and Blake, R. (2005). *Binocular Rivlary*. MIT Press, Cambridge, MA, USA.

Behnel, S., Bradshaw, R., Citro, C., Dalcin, L., Seljebotn, D., and Smith, K. (2011). Cython: The best of both worlds. *Computing in Science Engineering*, 13(2):31 –39.

Berkes, P., Orbán, G., Lengyel, M., and Fiser, J. (2011). Spontaneous cortical activity reveals hallmarks of an optimal internal model of the environment. *Science*, 331(6013):83–87.

Bill, J., Buesing, L., Habenschuss, S., Nessler, B., Maass, W., and Legenstein, R. (2015). Distributed bayesian computation and self-organized learning in sheets of spiking neurons with local lateral inhibition. *(submitted)*.

Bill, J. and Legenstein, R. (2014). A compound memristive synapse model for statistical learning through stdp in spiking neural networks. *Frontiers in Neuroscience*, 8(412).

Bishop, C. M. (2006). *Pattern Recognition and Machine Learning*. Springer.

Blei, D. M., Ng, A., and Jordan, M. (2003). Latent dirichlet allocation. *JMLR*, 3:993–1022.

Borkar, S. and Chien, A. A. (2011). The future of microprocessors. *Communications of the ACM*, 54(5):67–77.

Breitwieser, O. (2011). Investigation of a cortical attractor-memory network. Bachelor thesis, Ruprecht-Karls-Universität Heidelberg. HD-KIP 11-173.

Brette, R. and Gerstner, W. (2005). Adaptive exponential integrate-and-fire model as an effective description of neuronal activity. *J. Neurophysiol.*, 94:3637 – 3642.

Brette, R. and Goodman, D. (2008). Brian. A simulator for spiking neural networks based on Python.

Brette, R., Rudolph, M., Carnevale, T., Hines, M., Beeman, D., Bower, J. M., Diesmann, M., Morrison, A., Goodman, P. H., Harris Jr, F. C., Zirpe, M., Natschlager, T., Pecevski, D., Ermentrout, B., Djurfeldt, M., Lansner, A., Rochel, O., Vieville, T., Muller, E., Davison, A. P., Boustani, S. E., and Destexhe, A. (2007). Simulation of networks of spiking neurons: A review of tools and strategies. *Journal of Computational Neuroscience*, 23(3):349–398.

Brüderle, D., Petrovici, M. A., Vogginger, B., Ehrlich, M., Pfeil, T., Millner, S., Grübl, A., Wendt, K., Müller, E., Schwartz, M.-O., de Oliveira, D., Jeltsch, S., Fieres, J., Schilling, M., Müller, P., Breitwieser, O., Petkov, V., Muller, L., Davison, A., Krishnamurthy, P., Kremkow, J., Lundqvist, M., Muller, E., Partzsch, J., Scholze, S., Zühl, L., Mayr, C., Destexhe, A., Diesmann, M., Potjans, T., Lansner, A., Schüffny, R., Schemmel, J., and Meier, K. (2011). A comprehensive workflow for general-purpose neural modeling with highly configurable neuromorphic hardware systems. *Biological Cybernetics*, 104:263–296.

Brunel, N. (2000). Dynamics of sparsely connected networks of excitatory and inhibitory spiking neurons. *Journal of Computational Neuroscience*, 8(3):183–208.

Buesing, L., Bill, J., Nessler, B., and Maass, W. (2011). Neural dynamics as sampling: A model for stochastic computation in recurrent networks of spiking neurons. *PLoS Computational Biology*, 7(11):e1002211.

Bytschok, I. (2011). *From shared input to correlated neuron dynamics: Development of a predictive framework*. PhD thesis, Diploma thesis, University of Heidelberg.

Cireşan, D., Meier, U., Masci, J., and Schmidhuber, J. (2012). Multi-column deep neural network for traffic sign classification. *Neural Networks*, 32:333–338.

Davison, A. P., Brüderle, D., Eppler, J., Kremkow, J., Muller, E., Pecevski, D., Perrinet, L., and Yger, P. (2008). PyNN: a common interface for neuronal network simulators. *Front. Neuroinform.*, 2(11).

Deng, L. and Yu, D. (2011). Deep convex network: A scalable architecture for speech pattern classification. In *Interspeech*. International Speech Communication Association.

Destexhe, A., Rudolph, M., and Pare, D. (2003). The high-conductance state of neocortical neurons in vivo. *Nature Reviews Neuroscience*, 4:739–751.

Diesmann, M. and Gewaltig, M.-O. (2002). NEST: An environment for neural systems simulations. In Plesser, T. and Macho, V., editors, *Forschung und wisschenschaftliches Rechnen, Beiträge zum Heinz-Billing-Preis 2001*, volume 58 of *GWDG-Bericht*, pages 43–70. Ges. für Wiss. Datenverarbeitung, Göttingen.

Douglas, R., Mahowald, M., and Mead, C. (1995). Neuromorphic analogue VLSI. *Annu. Rev. Neurosci.*, 18:255–281.

Doya, K., Ishii, S., Pouget, A., and Rao, R. (2011). *Bayesian Brain: Probabilistic Approaches to Neural Coding*. Computational Neuroscience Series. MIT Press.

Eccles, J. C., Fatt, P., and Koketsu, K. (1954). Cholinergic and inhibitory synapses in a pathway from motor-axon collaterals to motoneurones. *J. Physiol.*, 126:524–562.

Friedmann, S. (2013). *A New Approach to Learning in Neuromorphic Hardware*. PhD thesis, Ruprecht-Karls-Universität Heidelberg.

Friedmann, S., Frémaux, N., Schemmel, J., Gerstner, W., and Meier, K. (2013). Reward-based learning under hardware constraints - using a RISC processor in a neuromorphic system. *Frontiers in Neuromorphic Engineering*. submitted.

Galassi, M. et al. (2009). *GNU Scientific Library Reference Manual*. 3rd ed. edition.

Geman, S. and Geman, D. (1984). Stochastic relaxation, gibbs distributions, and the bayesian restoration of images. *IEEE Trans. Pattern Anal. Mach. Intell.*, 6(6):721–741.

Gerstner, W. and Kistler, W. (2002). *Spiking Neuron Models: Single Neurons, Populations, Plasticity*. Cambridge University Press.

Gleeson, P., Crook, S., Cannon, R. C., Hines, M. L., Billings, G. O., Farinella, M., Morse, T. M., Davison, A. P., Ray, S., Bhalla, U. S., Barnes, S. R., Dimitrova, Y. D., and Silver, A. (2010). Neuroml: A language for describing data driven models of neurons and networks with a high degree of biological detail. *PLoS Computational Biology*, 6(6).

Graca, J. V., Inesc-id, L., Ganchev, K., Taskar, B., Graça, J. V., Inesc-id, L. F., Ganchev, K., and Taskar, B. (2007). Expectation maximization and posterior constraints. In *In Advances in NIPS*, pages 569–576.

Griffiths, T. L., Kemp, C., and Tenenbaum, J. B. (2008). Bayesian models of cognition. *Cambridge Handbook of Computational Cognitive Modeling*.

Griffiths, T. L. and Tenenbaum, J. B. (2006). Optimal predictions in everyday cognition. *Psychological Science 17*.

Habenschuss, S., Bill, J., and Nessler, B. (2012). Homeostatic plasticity in bayesian spiking networks as expectation maximization with posterior constraints. *Advances in Neural Information Processing Systems*, 25.

Habenschuss, S., Puhr, H., and Maass, W. (2013). Emergence of optimal decoding of population codes through stdp. *Neural Computation*, 25(6):1371–1407.

Hartel, A. and Schemmel, J. (2014). *Specification of the HICANN-DLS ASIC*. Human Brain Project internal documentation.

Hastings, W. (1970). Monte Carlo sampling methods using Markov chains and their applications. *Biometrika*, 57:97–109.

HBP SP9 partners (2014). *Neuromorphic Platform Specification*. Human Brain Project.

Hines, M. and Carnevale, N. (2003). *The NEURON simulation environment.*, pages 769–773. M.A. Arbib.

Hinton, G. (2010). A practical guide to training restricted boltzmann machines. *Momentum*, 9(1).

Hinton, G. E. (2002). Training products of experts by minimizing contrastive divergence. *Neural computation*, 14(8):1771–1800.

Hinton, G. E. (2007). Boltzmann machine. *Scholarpedia*, 2(5):1668. revision 91075.

Hofmann, T. (1999). Learning the similarity of documents: An information-geometric approach to document retrieval and categorization. In *Advances in Neural Information Processing Systems 12, [NIPS Conference, Denver, Colorado, USA, November 29 - December 4, 1999]*, pages 914–920.

Hunter, J. D. (2007). Matplotlib: A 2D graphics environment. *IEEE Computing in Science and Engineering*, 9(3):90–95.

Jones, E., Oliphant, T., and Peterson, P. (2001). SciPy: Open source scientific tools for Python.

Kappel, D., Nessler, B., and Maass, W. (2014). Stdp installs in winner-take-all circuits an online approximation to hidden markov model learning. *PLoS Comput Biol*, 10(3):e1003511.

Kim, J., Kim, H., Lakshmanan, K., and Rajkumar, R. R. (2013). Parallel scheduling for cyber-physical systems: Analysis and case study on a self-driving car. In *Proceedings of the ACM/IEEE 4th International Conference on Cyber-Physical Systems*, ICCPS '13, pages 31–40, New York, NY, USA. ACM.

Knill, D. C. and Pouget, A. (2004). The bayesian brain: the role of uncertainty in neural coding and computation. *TRENDS in Neurosciences*, 27(12):712–719.

Körding, K. and Wolpert, D. (2004). Bayesian integration in sensorimotor learning. *Nature*, 427(6971):244–247.

Kunkel, S., Diesmann, M., and Morrison, A. (2011). Limits to the development of feed-forward structures in large recurrent neuronal networks. *Frontiers in Computational Neuroscience*, 4(160).

Lecun, Y., Bottou, L., Bengio, Y., and Haffner, P. (1998). Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324.

LeCun, Y. and Cortes, C. (1998). The mnist database of handwritten digits.

Leng, L. (2014). Deep learning architectures for neuromorphic hardware. Masterthesis, Universität Heidelberg.

Lundqvist, M., Compte, A., and Lansner, A. (2010). Bistable, irregular firing and population oscillations in a modular attractor memory network. *PLoS Comput Biol*, 6(6).

Lundqvist, M., Rehn, M., Djurfeldt, M., and Lansner, A. (2006). Attractor dynamics in a modular network of neocortex. *Network: Computation in Neural Systems*, 17:3:253–276.

MacQueen, J. (1967). Some methods for classification and analysis of multivariate observations. Proc. 5th Berkeley Symp. Math. Stat. Probab., Univ. Calif. 1965/66, 1, 281-297 (1967).

Markram, H., Gupta, A., Uziel, A., Wang, Y., and Tsodyks, M. (1998). Information processing with frequency-dependent synaptic connections. *Neurobiol Learn Mem*, 70(1-2):101–112.

McManus, I. C., Freegard, M., Moore, J., and Rawles, R. (2010). Science in the making: Right hand, left hand. ii: The duck–rabbit figure. *Laterality: Asymmetries of Body, Brain and Cognition*, 15(1-2):166–185.

Mead, C. A. (1990). Neuromorphic electronic systems. *Proceedings of the IEEE*, 78:1629–1636.

Metropolis, N. and Ulam, S. M. (1949). The Monte Carlo Method. *Journal of the American Statistical Association*, 44(247):335–341.

Millner, S. (2012). *Development of a Multi-Compartment Neuron Model Emulation*. PhD thesis, Ruprecht-Karls University Heidelberg.

Moore, G. E. (1965). Cramming more components onto integrated circuits. *Electronics*, 38(8).

Morrison, A., Aertsen, A., and Diesmann, M. (2007). Spike-Timing-Dependent Plasticity in Balanced Random Networks. *Neural Comp.*, 19(6):1437–1467.

Morrison, A., Diesmann, M., and Gerstner, W. (2008). Phenomenological models of synaptic plasticity based on spike timing. *Biological Cybernetics*, 98(6):459–478.

Morrison, A., Mehring, C., Geisel, T., Aertsen, A., and Diesmann, M. (2005). Advancing the boundaries of high connectivity network simulation with distributed computing. *Neural Comput.*, 17(8):1776–1801.

Müller, E. C. (2014). *Novel Operation Modes of Accelerated Neuromorphic Hardware*. PhD thesis, Ruprecht-Karls-Universität Heidelberg.

Nessler, B., Pfeiffer, M., Buesing, L., and Maass, W. (2013). Bayesian computation emerges in generic cortical microcircuits through spike-timing-dependent plasticity. *PLoS Computational Biology*, 9(4):e1003037.

Nessler, B., Pfeiffer, M., and Maass, W. (2008). Hebbian learning of bayes optimal decisions. In Koller, D., Schuurmans, D., Bengio, Y., and Bottou, L., editors, *Advances in Neural Information Processing Systems 21*, pages 1169–1176.

Nessler, B., Pfeiffer, M., and Maass, W. (2009). Stdp enables spiking neurons to detect hidden causes of their inputs. In *NIPS*, pages 1357–1365.

Numpy (2012). Website. `http://numpy.scipy.org`.

Oaksford, M. and Chater, N. (2007). *Bayesian Rationality: The Probabilistic Approach to Human Reasoning*. Oxford Cognitive Science Series. OUP Oxford.

Pearl, J. (1988). *Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

Pecevski, D., Buesing, L., and Maass, W. (2011). Probabilistic inference in general graphical models through sampling in stochastic networks of spiking neurons. *PLoS Comput Biol*, 7(12):e1002294.

Pecevski, D. A., Natschläger, T., and Schuch, K. N. (2009). Pcsim: A parallel simulation environment for neural circuits fully integrated with Python. *Front. Neuroinform.*, 3(11).

Petrovici, M. A., Bill, J., Bytschok, I., Schemmel, J., and Meier, K. (2013). Stochastic inference with deterministic spiking neurons. *arXiv preprint arXiv:1311.3211*.

Petrovici, M. A., Vogginger, B., Müller, P., Breitwieser, O., Lundqvist, M., Muller, L., Ehrlich, M., Destexhe, A., Lansner, A., Schüffny, R., et al. (2014). Characterization and compensation of network-level anomalies in mixed-signal neuromorphic modeling platforms. *PloS one*, 9(10):e108590.

Pfeil, T., Potjans, T. C., Schrader, S., Potjans, W., Schemmel, J., Diesmann, M., and Meier, K. (2012). Is a 4-bit synaptic weight resolution enough? - constraints on enabling spike-timing dependent plasticity in neuromorphic hardware. *Frontiers in Neuroscience*, 6(90).

PowerISA (2010). PowerISA version 2.06 revision b. Technical report, power.org. Available at `http://www.power.org/resources/reading/`.

Probst, D. (2014). A neural implementation of probabilistic inference in binary probability spaces. Master thesis, Ruprecht-Karls-Universität Heidelberg.

Probst, D., Petrovici, M. A., Bytschok, I., Bill, J., Pecevski, D., Schemmel, J., and Meier, K. (2015). Probabilistic inference in discrete spaces can be implemented into networks of lif neurons. *Frontiers in Computational Neuroscience*, 9(13).

Ricciardi, L. M. (1977). *Diffusion Processes and Related Topics in Biology*. Springer Berlin Heidelberg.

Rossum, G. V. (2000). *Python Reference Manual: February 19, 1999, Release 1.5.2*. iUniverse, Incorporated.

Sato, M.-a. (1999). Fast learning of on-line em algorithm. Technical report, ATR Human Information Processing Research Laboratories.

Schemmel, J., Brüderle, D., Grübl, A., Hock, M., Meier, K., and Millner, S. (2010). A wafer-scale neuromorphic hardware system for large-scale neural modeling. In *Proceedings of the 2010 IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 1947–1950.

Schemmel, J., Fieres, J., and Meier, K. (2008). Wafer-scale integration of analog neural networks. In *Proceedings of the 2008 International Joint Conference on Neural Networks (IJCNN)*.

Schemmel, J., Grübl, A., Meier, K., and Muller, E. (2006). Implementing synaptic plasticity in a VLSI spiking neural network model. In *Proceedings of the 2006 International Joint Conference on Neural Networks (IJCNN)*. IEEE Press.

Steinhaus, H. (1957). Sur la division des corps matériels en parties. *Bull. Acad. Pol. Sci., Cl. III*, 4:801–804.

Sutskever, I. and Tieleman, T. (2010). On the convergence properties of contrastive divergence. In *International Conference on Artificial Intelligence and Statistics*, pages 789–795.

Thomas, M. U. (1975). Some mean first-passage time approximations for the ornstein-uhlenbeck process. *Journal of Applied Probability*, pages 600–604.

Tieleman, T. (2008). Training restricted boltzmann machines using approximations to the likelihood gradient. In *Proceedings of the 25th international conference on Machine learning*, pages 1064–1071. ACM.

Tsodyks, M. and Markram, H. (1997). The neural code between neocortical pyramidal neurons depends on neurotransmitter release probability. *Proceedings of the national academy of science USA*, 94:719–723.

Tsodyks, M., Pawelzik, K., Markram, H., and Tsodyks, M. (1998). Neural networks with dynamic synapses. *Neural Computation*, 10:821–835.

Uhlenbeck, G. E. and Ornstein, L. S. (1930). On the theory of the brownian motion. *Phys. Rev.*, 36:823–841.

Wit, E., van den Heuvel, E., and Romeijn, J.-W. (2012). 'all models are wrong...': an introduction to model uncertainty. *Statistica Neerlandica*, 66(3):217–236.

Acronyms and Technical Terms

ADC . . . . . analog-to-digital converter . . . . . (p. 42)
AdEx . . . . . adaptive exponential integrate-and-fire . . . . . (pp. 17, 38)
ANC . . . . . analog network core . . . . . (p. 38)
API . . . . . application programming interface . . . . . (pp. 43, 45, 96)
BM . . . . . Boltzmann machine . . . (pp. 1, 9, 10, 16, 17, 26, 31, 32, 34, 47, 48, 50, 95–97, 100, 112)
BSS . . . . . BrainScaleS . . . . . (pp. 2, 37, 38)
CD . . . . . contrastive divergence . . . . . (pp. 25, 26, 95, 97, 100, 112, 114)
CLI . . . . . command line interface . . . . . (p. 99)
CNN . . . . . convolutional neural network . . . . . (p. 120)
DAC . . . . . digital-analog converter . . . . . (p. 39)
DenMem . dendrite membrane . . . . . (pp. 38–40)
D_{KL} . . . . . Kullback-Leibler divergence . . . . . (pp. 7, 8, 25, 27, 28, 67, 96)
E-step . . . . expectation step . . . . . (pp. 28–31, 33, 35, 49)
EEG . . . . . electroencephalography . . . . . (p. 43)
EM . . . . . expectation maximization . . . . . (pp. 2, 26–31, 33, 115)
FACETS . . . Fast Analog Computing with Emerging Transient States . . . . . (pp. 2, 37)
FPT . . . . . first passage time . . . . . (pp. 21, 23)
GID . . . . . global ID . . . . . (p. 104)
GSL . . . . . GNU Scientific Library . . . . . (pp. 44, 103)
HBP . . . . . Human Brain Project . . . . . (pp. 2, 37, 105)
HCS . . . . . high-conductance state . . . . . (pp. 17, 18, 20, 22)
HDF . . . . . Hierachical Data Format . . . . . (p. 99)
HICANN . . high input count analog neural network . . . . . (pp. 37–42, 54)
HMM . . . . . hidden Markov model . . . . . (p. 31)
ISI . . . . . inter-spike interval . . . . . (pp. 51, 57–59, 85, 88, 91, 113)
JSON . . . . . JavaScript Object Notation . . . . . (p. 96)
L1 . . . . . layer-1 . . . . . (pp. 37, 39, 40)
L2 . . . . . layer-2 . . . . . (pp. 38, 39, 51)
LDA . . . . . latent Dirichlet allocation . . . . . (p. 27)
LFSR . . . . . linear-feedback shift register . . . . . (p. 39)
LIF . . . . . leaky integrate-and-fire . . (pp. 2, 3, 5, 17, 18, 21, 22, 47–49, 68, 69, 71–75, 95–97, 100, 115, 119)
LUT . . . . . look up table . . . . . (pp. 41, 42, 66)

M-step . . . . maximization step . . . . . (pp. 28–31, 33)
MAP . . . . . maximum a-posteriori probability . . . . . (p. 9)
MCMC . . . . Markov chain Monte Carlo . . . . . (pp. 10, 12, 13, 16)
ML . . . . . maximum likelihood . . . . . (pp. 1, 8, 25, 27, 115)
NCC . . . . . neural computability condition . . . . . (pp. 14, 16, 17, 33)
NEST . . . . . NEural Simulation Tool . . . . . (pp. 44, 45, 50, 95, 98, 100–104, 109, 111–114, 117)
NM-PM1 . . Neuromorphic Physical Model System 1 . . (pp. 2, 16–18, 37, 38, 42, 44, 47, 49–51, 59, 68, 115, 116, 119, 120)
NSEM . . . . . neuromorphic spike-based expectation maximization (pp. 2, 66, 75, 76, 78, 79, 81, 82, 85–87, 91–93, 115–117, 119–121)
ODE . . . . . ordinary differential equation . . . . . (pp. 19, 22, 44)
OU . . . . . Ornstein-Uhlenbeck . . . . . (pp. 17, 19, 20)
PCD . . . . . persistent contrastive divergence . . . . . (p. 26)
PDF . . . . . probability density function . . . . . (pp. 5, 22)
PLSA . . . . . probabilistic latent semantic analysis . . . . . (p. 27)
PPU . . . . . plasticity processing unit . . . . . (pp. 38, 42, 51, 59, 61, 63, 73, 74)
PSP . . . . . post-synaptic potential . . . . . (pp. 1, 22–24, 32)
RBM . . . . . restricted Boltzmann machine . . . . . (pp. 10, 25, 100, 112, 120)
RV . . . . . random variable . . . . . (pp. 1, 5–7, 9, 10, 12–16, 21, 26, 52, 67)
sbl . . . . . spike-based learning . . . . . (pp. 95, 100, 114)
sbs . . . . . spike-based sampling . . . . . (pp. 95–98, 114)
SEM . . . . . spike-based expectation maximization . . (pp. 2, 3, 5, 31, 34, 45, 47, 53, 59, 61, 67–69, 71–74, 90, 95, 98–100, 111, 112, 115, 119, 120)
SEMf . . . . . spike-based expectation maximization framework . . . (pp. 95, 98–100, 109, 114, 120, 133)
STD . . . . . short term depression . . . . . (p. 24)
STDP . . . . . spike timing dependent plasticity . (pp. 3, 31–33, 38, 39, 41, 42, 47, 52–56, 59, 61, 68, 101, 113, 115, 116)
STF . . . . . short term facilitation . . . . . (p. 24)
STP . . . . . short term plasticity . . . . . (pp. 24, 39, 120)
TM . . . . . Tsodyks-Markram . . . . . (pp. 24, 39, 40, 96)
VLSI . . . . . very-large-scale integration . . . . . (p. 37)
WTA . . . . . winner-take-all . . . . . (pp. 31, 47, 48, 116, 120)
YAML . . . . . YAML Ain't Markup Language . . . . . (pp. 99, 133)

A Parameters

A.1 Simulations with SEMf

This chapter lists all *YAML Ain't Markup Language* (YAML) parameter files used for full network simulations with SEMf which can be found at `https://gitviz.kip.uni-heidelberg.de` in the `model-sem` git repository. The `waf` installation scripts are located in the `src/combined` subfolder.

All parameters are specified either unit-less, in theoretical units or – in case of neuron parameters – in PyNN default units. Due to the relatively long time scales on which simulations are performed, the default unit for time was chosen to be seconds instead of milliseconds.

A.1.1 Background Source Comparison

See Section 6.4.7.

```
cache:
    - tau_syn: 10.0e-3
      acf: 1.
      rate_low : 10.
      rate_high: 90.
      nn_only: false
      activity_ratio: 1.
      pattern_length: 0.5
      pattern_pause: 0.0
      pattern_step: 0.1
      weight_bits: 6
      max_weight_factor: 1.2
      update_period: 2.0
      num_z: 3
      dt: 0.1e-3
      labels_to_create: [0,3,4]
      rate_nc: 28.
    - rate_total: cc.rate_high + cc.rate_low
    - rfactor: 1+1./(cc.tau_syn*cc.rate_total)
    - rfactor_nc: 1+1./(cc.tau_syn*cc.rate_nc)
    - lambda_target_ideal_nn: (1.-np.exp(-cc.rfactor*cc.rate_total*cc.tau_syn))/cc.rfactor
    - lambda_target_ideal: cc.lambda_target_ideal_nn if cc.nn_only else cc.rate_total * cc.tau_syn
    - lambda_nc_nn: (1.-np.exp(-cc.rfactor_nc*cc.rate_nc*cc.tau_syn))/cc.rfactor_nc
    - lambda_nc: cc.lambda_nc_nn if cc.nn_only else cc.rate_nc * cc.tau_syn
    - ideal_max_weight: np.log(cc.lambda_target_ideal/cc.lambda_nc)
    - pattern_total: cc.pattern_length + cc.pattern_pause
    - prob_spike_net: cc.pattern_length / cc.pattern_total * cc.activity_ratio

combiner_params:
        steps_per_pattern: !ee int(np.around(cc.pattern_total/cc.pattern_step))
combiner_types: uniform_unique
creator_params:
    inputparams:
        rate: !ee cc.rate_high
    inputtype: AdditivePoisson
    length: !ee cc.pattern_length
    num_repr_per_label: 0
    labels_to_create: !ee cc.labels_to_create
    clip_edge_pixel: 2
creator_types: mnist
injector_types: background
injector_params:
    inputtype: AdditivePoisson
    inputparams:
        rate: !ee cc.rate_low
duration: 5000.0
input_time_step:  500.0
process_all_input_prior: false
manager_params:
    step: !ee cc.pattern_step
network_type: SimTrain
network_params:
    dt: !ee cc.dt
    sim_step: 25.
    num_z: !ee cc.num_z
    record_input: false
    source_type: SimulationLookaheadMultiPoissonVarRateSource
    statefactory_params:
        initializer_params:
            W:
                value: 0.
            b:
                value: 0.
    causelayer_type: BoltzMann_WTA_Inhibition
    causelayer_params:
        saturating_synapses: false
        database: ~/data/sbs/calibration/default-02-10ms
        record_voltages: false
        delays: 0.1e-3
        inh_weight: 100.
        inh_neuron_params:
            cm: 0.2
            e_rev_E: 0.0
            e_rev_I: -100.0
            i_offset: 0.0
            tau_m: 1.0
            tau_refrac: 5.0
            tau_syn_E: 5.0
            tau_syn_I: 5.0
```

```
            v_reset: -50.0
            v_rest: -50.0
            v_thresh: -40.0
        neuron_params_ids: 0
    connection_type:
          coincidence_limited_weight_resolution_aggregating
    connection_params:
        set_y_accum_factor: false
        nullcause: !ee np.log(cc.lambda_nc)
        eta: 1.0e-4
        tau_syn: !ee cc.tau_syn
        tau_onset: 0.0e-3
        receptor_type: excitatory
        prob_spike_net: !ee cc.prob_spike_net
        update_period: !ee cc.update_period
        prob_weight_update: true
        num_weights: !ee 2**cc.weight_bits
        max_weight_theo: !ee cc.ideal_max_weight * cc.
             max_weight_factor
        nearest_neighbors_only: !ee cc.nn_only
        adjust_conversion_factor: !ee cc.acf
        discretize_weights: true
    homeostasis_type: linear
    homeostasis_params:
        prob_spike_net: !ee cc.prob_spike_net
        source_model: periodic_generator
        source_model_kwargs:
            offset: 1.
        create_exc: true
        eta: 1.0e-3
        rate: 1000.0
    take_snapshots: true
    record_spikes: true
    steps_per_snapshot: 1
    steps_per_report: 4
numpy_seed: 42
```

A.1.2 Regular Network Dynamics

Ideal Update

```
cache:
    - tau_syn: 30.0e-3
      rate_low : 10.0
      rate_high: 60.0
      nn_only: false
      activity_ratio: 1.
      pattern_length: 0.5
      pattern_pause: 0.0
      pattern_step: 0.1
      num_z: 6
      dt: 0.1e-3
      rate_nc: 10.
    - rate_total: cc.rate_high + cc.rate_low
    - lambda_nc: cc.rate_nc * cc.tau_syn
    - pattern_total: cc.pattern_length + cc.pattern_pause
    - prob_spike_net: cc.pattern_length / cc.pattern_total *
         cc.activity_ratio

combiner_params:
        steps_per_pattern: !ee int(np.around(cc.pattern_total
             /cc.pattern_step))
combiner_types: uniform_unique
creator_types: orientedstripv2
creator_params:
    inputparams:
        rate: !ee cc.rate_high
    inputtype: AdditivePoisson
    length: !ee cc.pattern_length
    width_image: 17
    width_strip: 3
injector_types: background
injector_params:
    inputtype: AdditivePoisson
    inputparams:
        rate: !ee cc.rate_low
duration: 10000
input_time_step:  500.0
process_all_input_prior: false
manager_params:
    step: !ee cc.pattern_step
network_type: SimTrain
network_params:
    dt: !ee cc.dt
    sim_step: 25.
    num_z: !ee cc.num_z
    record_input: false
    source_type: SimulationLookaheadMultiPoissonVarRateSource
    statefactory_params:
        initializer_params:
            W:
                value: 0.
            b:
                value: 0.
    causelayer_type: BoltzMann_WTA_Inhibition
    causelayer_params:
        saturating_synapses: false
        database: ~/data/sbs/calibration/default-02-30ms
        record_voltages: false
        delays: 0.1e-3
        inh_weight: 100.
        inh_neuron_params:
            cm: 0.2
            e_rev_E: 0.0
            e_rev_I: -100.0
            i_offset: 0.0
            tau_m: 1.0
            tau_refrac: 5.0
            tau_syn_E: 5.0
            tau_syn_I: 5.0
            v_reset: -50.0
            v_rest: -50.0
            v_thresh: -40.0
        neuron_params:
            cm         : .2
            tau_m      : 1.
            e_rev_E    : 0.
            e_rev_I    : -100.
            v_thresh   : -50.
            tau_syn_E  : 10.
            v_rest     : -50.
            tau_syn_I  : 10.
            v_reset    : -50.001
            tau_refrac : 10.
            i_offset   : 0.
        neuron_params_ids: 0
    connection_type: ideal
    connection_params:
        nullcause: !ee np.log(cc.lambda_nc)
        eta: 1.0e-4
        tau_syn: !ee cc.tau_syn
        receptor_type: excitatory
    homeostasis_type: linear
    homeostasis_params:
        prob_spike_net: !ee cc.prob_spike_net
        source_model: periodic_generator
        source_model_kwargs:
            offset: 1.
        create_exc: true
        create_inh: true
        eta: 1.0e-3
        rate: 2000.0
    rng_seed: 424242
    take_snapshots: true
    record_spikes: true
    steps_per_snapshot: 1
    steps_per_report: 4
numpy_seed: 42
```

Exponential Eligibility Traces

In order to conserve space, we only show the differences to the previous dataset.

```
84c84
<     connection_type: ideal
---
>     connection_type: exp
```

Pair-based Correlation Measurements

In order to conserve space, we only show the differences to the previous dataset.

```
84c84
<     connection_type: exp
---
>     connection_type: coincidence
89a90,92
>         tau_onset: 0.
>         prob_spike_net: !ee cc.prob_spike_net
>         nearest_neighbors_only: !ee cc.nn_only
```

Nearest-Neighbor Spike Pairing

In order to conserve space, we only show the differences to the previous dataset.

```
5c5
<       nn_only: false
---
>       nn_only: true
14c14,17
<     - lambda_nc: cc.rate_nc * cc.tau_syn
---
>     - rfactor_nc: 1+1./(cc.tau_syn*cc.rate_nc)
>     - lambda_nc_nn: (1.-np.exp(-cc.rfactor_nc*cc.rate_nc*cc.tau_syn))/cc.rfactor_nc
>     - lambda_nc_all: cc.rate_nc * cc.tau_syn
>     - lambda_nc: cc.lambda_nc_nn if cc.nn_only else cc.lambda_nc_all
```

Accumulated Weight Updates

In order to conserve space, we only show the differences to the previous dataset.

```
11a12
>       update_period: 2.0
87c88
<     connection_type: coincidence
---
>     connection_type: coincidenceaggregating
95a97,99
>         set_y_accum_factor: false
>         update_period: !ee cc.update_period
>         update_offset_per_sampler: !ee cc.update_period / cc.num_z
```

Limited Weight Resolution

In order to conserve space, we only show the differences to the previous dataset.

6-bit weight resolution:

```
12a13,14
>       weight_bits: 6
>       max_weight_factor: 1.2
14a17
>     - rfactor: 1+1./(cc.tau_syn*cc.rate_total)
15a19,21
>     - lambda_target_ideal_nn: (1.-np.exp(-cc.rfactor*cc.rate_total*cc.tau_syn))/cc.rfactor
>     - lambda_target_all: cc.lambda_target_ideal_nn if cc.nn_only else cc.rate_total * cc.tau_syn
>     - lambda_target: cc.lambda_target_ideal_nn if cc.nn_only else cc.lambda_target_all
18a25
>     - ideal_max_weight: np.log(cc.lambda_target/cc.lambda_nc)
20a28
```

```
>     - max_weight_theo: cc.ideal_max_weight * cc.max_weight_factor
88c96
<     connection_type: coincidenceaggregating
---
>     connection_type: coincidence_limited_weight_resolution_aggregating
99a108,111
>         num_weights: !ee 2**cc.weight_bits
>         max_weight_theo: !ee cc.max_weight_theo
>         discretize_weights: true
>         prob_weight_update: true
```

4-bit weight resolution:

```
13c13
<       weight_bits: 6
---
>       weight_bits: 4
```

A.1.3 Null cause as input rate filter

Parameters used for simulations presented in Section 5.4.3.

Reduced MNIST, 4-bit, default Null Cause Rate

Parameters used for Figure 5.17

```
# Lower inhibitory weight
# Higher sigma
# Fixed error in real nc calculations
cache:
    - tau_syn: 30.0e-3
      rate_low : 10.0
      rate_high: 30.0
      nn_only: true
      activity_ratio: 1.
      pattern_length: 0.5
      pattern_pause: 0.0
      pattern_step: 0.1
      labels_to_create: [0, 3, 4]
      num_z: 3
      dt: 0.1e-3
      update_period: 2.0
      weight_bits: 4
      max_weight_factor: 1.2
      rate_nc: 10.
    - rate_total: cc.rate_high + cc.rate_low
    - rfactor: 1+1./(cc.tau_syn*cc.rate_total)
    - rfactor_nc: 1+1./(cc.tau_syn*cc.rate_nc)
    - lambda_target_ideal_nn: (1.-np.exp(-cc.rfactor*cc.
        rate_total*cc.tau_syn))/cc.rfactor
    - lambda_target_all: cc.lambda_target_ideal_nn if cc.
        nn_only else cc.rate_total * cc.tau_syn
    - lambda_target: cc.lambda_target_ideal_nn if cc.nn_only
        else cc.lambda_target_all
    - lambda_nc_nn: (1.-np.exp(-cc.rfactor_nc*cc.rate_nc*cc.
        tau_syn))/cc.rfactor_nc
    - lambda_nc_all: cc.rate_nc * cc.tau_syn
    - lambda_nc: cc.lambda_nc_nn if cc.nn_only else cc.
        lambda_nc_all
    - ideal_max_weight: np.log(cc.lambda_target/cc.lambda_nc)
    - pattern_total: cc.pattern_length + cc.pattern_pause
    - prob_spike_net: cc.pattern_length / cc.pattern_total *
        cc.activity_ratio
    - max_weight_theo: cc.ideal_max_weight * cc.
        max_weight_factor

combiner_params:
        steps_per_pattern: !ee int(np.around(cc.pattern_total
            /cc.pattern_step))
combiner_types: uniform_unique
creator_params:
    inputparams:
        rate: !ee cc.rate_high
    inputtype: AdditivePoisson
    labels_to_create: !ee cc.labels_to_create
    length: !ee cc.pattern_length
    num_repr_per_label: 0 # corresponds to all samples
creator_types: mnist
injector_types: background
injector_params:
    inputtype: AdditivePoisson
    inputparams:
        rate: !ee cc.rate_low
duration: 10000
input_time_step:  500.0
process_all_input_prior: false
manager_params:
    step: !ee cc.pattern_step
network_type: SimTrain
network_params:
    dt: !ee cc.dt
    sim_step: 25.
    num_z: !ee cc.num_z
    record_input: false
    source_type: SimulationLookaheadMultiPoissonVarRateSource
    statefactory_params:
        initializer_params:
            W:
                value: 0.
            b:
                value: 0.
    causelayer_type: BoltzMann_WTA_Inhibition
    causelayer_params:
        saturating_synapses: false
        database: ~/data/sbs/calibration/default-02-30ms
        record_voltages: false
        delays: 0.1e-3
        inh_weight: 100.
        inh_neuron_params:
            cm: 0.2
            e_rev_E: 0.0
            e_rev_I: -100.0
            i_offset: 0.0
            tau_m: 1.0
            tau_refrac: 5.0
            tau_syn_E: 5.0
            tau_syn_I: 5.0
            v_reset: -50.0
            v_rest: -50.0
            v_thresh: -40.0
        neuron_params:
            cm         : .2
            tau_m      : 1.
            e_rev_E    : 0.
            e_rev_I    : -100.
            v_thresh   : -50.
            tau_syn_E  : 10.
            v_rest     : -50.
```

```
            tau_syn_I  : 10.
            v_reset    : -50.001
            tau_refrac : 10.
            i_offset   : 0.
        neuron_params_ids: 0
    connection_type:
         coincidence_limited_weight_resolution_aggregating
    connection_params:
        nullcause: !ee np.log(cc.lambda_nc)
        eta: 1.0e-4
        tau_syn: !ee cc.tau_syn
        receptor_type: excitatory
        tau_onset: 0.
        prob_spike_net: !ee cc.prob_spike_net
        nearest_neighbors_only: !ee cc.nn_only
        set_y_accum_factor: false
        update_period: !ee cc.update_period
        num_weights: !ee 2**cc.weight_bits
        max_weight_theo: !ee cc.max_weight_theo
        discretize_weights: true
        prob_weight_update: true
    homeostasis_type: linear
    homeostasis_params:
        prob_spike_net: !ee cc.prob_spike_net
        source_model: periodic_generator
        source_model_kwargs:
            offset: 1.
        create_exc: true
        create_inh: true
        eta: 1.0e-3
        rate: 2000.0
    rng_seed: 424242
    take_snapshots: true
    record_spikes: true
    steps_per_snapshot: 1
    steps_per_report: 4
numpy_seed: 42
```

Reduced MNIST, 4-bit, adjusted Null Cause Rate

Parameters used for Figure 5.18 In order to conserve space, we only show the differences to the previous dataset.

```
19c19
<       rate_nc: 10.
---
>       rate_nc: 15.
```

Reduced MNIST, 4-bit, adjusted Null Cause Rate

Parameters used for Figure 5.20 In order to conserve space, we only show the differences to the previous dataset.

```
13,14c13,14
<       labels_to_create: [0, 3, 4]
<       num_z: 3
---
>       labels_to_create: range(10)
>       num_z: 10
17c17
<       weight_bits: 4
---
>       weight_bits: 6
50c50
< duration: 10000
---
> duration: 20000
```

Full MNIST with one representation, 6-bit, adjusted Null Cause Rate

Parameters used for Figure 5.19.

```
cache:
    - tau_syn: 30.0e-3
      rate_low : 10.0
      rate_high: 30.0
      nn_only: true
      activity_ratio: 1.
      pattern_length: 0.5
      pattern_pause: 0.0
      pattern_step: 0.1
      labels_to_create: range(10)
      num_z: 10
      dt: 0.1e-3
      update_period: 2.0
      weight_bits: 6
      max_weight_factor: 1.2
      rate_nc: 15.
    - rate_total: cc.rate_high + cc.rate_low
    - rfactor: 1+1./(cc.tau_syn*cc.rate_total)
    - rfactor_nc: 1+1./(cc.tau_syn*cc.rate_nc)
    - lambda_target_ideal_nn: (1.-np.exp(-cc.rfactor*cc.
        rate_total*cc.tau_syn))/cc.rfactor
    - lambda_target_all: cc.lambda_target_ideal_nn if cc.
        nn_only else cc.rate_total * cc.tau_syn
    - lambda_target: cc.lambda_target_ideal_nn if cc.nn_only
        else cc.lambda_target_all
    - lambda_nc_nn: (1.-np.exp(-cc.rfactor_nc*cc.rate_nc*cc.
        tau_syn))/cc.rfactor_nc
    - lambda_nc_all: cc.rate_nc * cc.tau_syn
    - lambda_nc: cc.lambda_nc_nn if cc.nn_only else cc.
        lambda_nc_all
    - ideal_max_weight: np.log(cc.lambda_target/cc.lambda_nc)
    - pattern_total: cc.pattern_length + cc.pattern_pause
    - prob_spike_net: cc.pattern_length / cc.pattern_total *
        cc.activity_ratio
    - max_weight_theo: cc.ideal_max_weight * cc.
        max_weight_factor

combiner_params:
```

```
        steps_per_pattern: !ee int(np.around(cc.pattern_total/cc.pattern_step))
combiner_types: uniform_unique
creator_params:
    inputparams:
        rate: !ee cc.rate_high
    inputtype: AdditivePoisson
    labels_to_create: !ee cc.labels_to_create
    length: !ee cc.pattern_length
    num_repr_per_label: 1
creator_types: mnist
injector_types: background
injector_params:
    inputtype: AdditivePoisson
    inputparams:
        rate: !ee cc.rate_low
duration: 15000
input_time_step:  500.0
process_all_input_prior: false
manager_params:
    step: !ee cc.pattern_step
network_type: SimTrain
network_params:
    dt: !ee cc.dt
    sim_step: 25.
    num_z: !ee cc.num_z
    record_input: false
    source_type: SimulationLookaheadMultiPoissonVarRateSource
    statefactory_params:
        initializer_params:
            W:
                value: 0.
            b:
                value: 0.
    causelayer_type: BoltzMann_WTA_Inhibition
    causelayer_params:
        saturating_synapses: false
        database: ~/data/sbs/calibration/default-02-30ms
        record_voltages: false
        delays: 0.1e-3
        inh_weight: 100.
        inh_neuron_params:
            cm: 0.2
            e_rev_E: 0.0
            e_rev_I: -100.0
            i_offset: 0.0
            tau_m: 1.0
            tau_refrac: 5.0
            tau_syn_E: 5.0
            tau_syn_I: 5.0
            v_reset: -50.0
            v_rest: -50.0
            v_thresh: -40.0
        neuron_params_ids: 0
    connection_type:
          coincidence_limited_weight_resolution_aggregating
    connection_params:
        nullcause: !ee np.log(cc.lambda_nc)
        eta: 1.0e-4
        tau_syn: !ee cc.tau_syn
        receptor_type: excitatory
        tau_onset: 0.
        prob_spike_net: !ee cc.prob_spike_net
        nearest_neighbors_only: !ee cc.nn_only
        set_y_accum_factor: false
        update_period: !ee cc.update_period
        num_weights: !ee 2**cc.weight_bits
        max_weight_theo: !ee cc.max_weight_theo
        discretize_weights: true
        prob_weight_update: true
    homeostasis_type: linear
    homeostasis_params:
        prob_spike_net: !ee cc.prob_spike_net
        source_model: periodic_generator
        source_model_kwargs:
            offset: 1.
        create_exc: true
        create_inh: true
        eta: 1.0e-3
        rate: 1000.0
    rng_seed: 424242
    take_snapshots: true
    record_spikes: true
    steps_per_snapshot: 1
    steps_per_report: 4
numpy_seed: 42
```

Common Box Input Sweep

Parameters used for Figure 5.21 as well as in the sweep depicted in Figure 5.22.

```
cache:
-   acf: 1.0
    activity_ratio: 1.0
    dt: 0.0001
    max_weight_factor: 1.2
    nn_only: true
    num_z: 9
    pattern_length: 0.5
    pattern_pause: 0.0
    pattern_step: 0.1
    rate_high: 30.0
    rate_low: 10.0
    rate_nc: 14.0
    tau_syn: 0.03
    update_period: 2.0
    weight_bits: 6
-   rate_total: cc.rate_high + cc.rate_low
-   rfactor: 1+1./(cc.tau_syn*cc.rate_total)
-   rfactor_nc: 1+1./(cc.tau_syn*cc.rate_nc)
-   lambda_target_ideal_nn: (1.-np.exp(-cc.rfactor*cc.rate_total*cc.tau_syn))/cc.rfactor
-   lambda_target_ideal: cc.lambda_target_ideal_nn if cc.nn_only else cc.rate_total
       * cc.tau_syn
-   lambda_nc_nn: (1.-np.exp(-cc.rfactor_nc*cc.rate_nc*cc.tau_syn))/cc.rfactor_nc
-   lambda_nc: cc.lambda_nc_nn if cc.nn_only else cc.rate_nc * cc.tau_syn
-   ideal_max_weight: np.log(cc.lambda_target_ideal/cc.lambda_nc)
-   pattern_total: cc.pattern_length + cc.pattern_pause
-   prob_spike_net: cc.pattern_length / cc.pattern_total * cc.activity_ratio
combiner_params:
    steps_per_pattern: !ee int(np.around(cc.pattern_total/cc.pattern_step))
combiner_types: uniform_unique
creator_params:
    box_sizes:
        common: 100
        pattern: 30
    inputparams:
        rate: !ee cc.rate_high
    inputtype: AdditivePoisson
    length: !ee cc.pattern_length
    num_patterns: !ee cc.num_z
    num_samples:
        active: 27
        common: 84
        inactive: 2
creator_types: commonboxsamples
duration: 15000.0
injector_params:
    inputparams:
        rate: !ee cc.rate_low
    inputtype: AdditivePoisson
injector_types: background
input_time_step: 500.0
manager_params:
    step: !ee cc.pattern_step
network_params:
    causelayer_params:
        database: ~/data/sbs/calibration/default-02-30ms
        delays: 0.0001
        inh_neuron_params:
            cm: 0.2
            e_rev_E: 0.0
            e_rev_I: -100.0
            i_offset: 0.0
            tau_m: 1.0
            tau_refrac: 5.0
            tau_syn_E: 5.0
            tau_syn_I: 5.0
            v_reset: -50.0
            v_rest: -50.0
            v_thresh: -40.0
```

```
        inh_weight: 100.0
        neuron_params:
            cm: 0.2
            e_rev_E: 0.0
            e_rev_I: -100.0
            i_offset: 0.0
            tau_m: 1.0
            tau_refrac: 30.0
            tau_syn_E: 30.0
            tau_syn_I: 30.0
            v_reset: -50.001
            v_rest: -50.0
            v_thresh: -50.0
        record_voltages: false
        saturating_synapses: false
    causelayer_type: BoltzMann_WTA_Inhibition
    connection_params:
        adjust_conversion_factor: !ee cc.acf
        eta: 0.0001
        max_weight_theo: !ee cc.ideal_max_weight * cc.
              max_weight_factor
        nearest_neighbors_only: !ee cc.nn_only
        nullcause: !ee np.log(cc.lambda_nc)
        num_weights: !ee 2**cc.weight_bits
        prob_spike_net: !ee cc.prob_spike_net
        prob_weight_update: true
        receptor_type: excitatory
        set_y_accum_factor: false
        tau_onset: 0.0
        tau_syn: !ee cc.tau_syn
        update_period: 2.0
    connection_type:
         coincidence_limited_weight_resolution_aggregating
    dt: !ee cc.dt
    homeostasis_params:
        create_exc: true
        create_inh: true
        eta: 0.001
        prob_spike_net: !ee cc.prob_spike_net
        rate: 1000.0
        source_model: periodic_generator
        source_model_kwargs:
            offset: 1.0
    homeostasis_type: linear
    num_z: !ee cc.num_z
    record_input: false
    record_spikes: true
    sim_step: 25.0
    source_type: SimulationLookaheadMultiPoissonVarRateSource
    statefactory_params:
        initializer_params:
            b:
                value: 0.0
    steps_per_report: 4
    steps_per_snapshot: 1
    take_snapshots: true
network_type: SimTrain
numpy_seed: 42
process_all_input_prior: false
```

A.1.4 Homeostasis background source

Parameters used for simulations in Section 5.4.4.

Reduced MNIST Dataset with Poisson Homeostasis Background Source

```
cache:
-   acf: 1.0
    activity_ratio: 1.0
    dt: 0.0001
    labels_to_create:
    - 0
    - 3
    - 4
    max_weight_factor: 1.2
    nn_only: true
    num_z: 3
    pattern_length: 0.5
    pattern_pause: 0.0
    pattern_step: 0.1
    rate_high: 30.0
    rate_low: 10.0
    rate_nc: 15.0
    tau_syn: 0.03
    update_period: 2.0
    weight_bits: 6
-   rate_total: cc.rate_high + cc.rate_low
-   rfactor: 1+1./(cc.tau_syn*cc.rate_total)
-   rfactor_nc: 1+1./(cc.tau_syn*cc.rate_nc)
-   lambda_target_ideal_nn: (1.-np.exp(-cc.rfactor*cc.
      rate_total*cc.tau_syn))/cc.rfactor
-   lambda_target_ideal: cc.lambda_target_ideal_nn if cc.
      nn_only else cc.rate_total
        * cc.tau_syn
-   lambda_nc_nn: (1.-np.exp(-cc.rfactor_nc*cc.rate_nc*cc.
      tau_syn))/cc.rfactor_nc
-   lambda_nc: cc.lambda_nc_nn if cc.nn_only else cc.rate_nc
      * cc.tau_syn
-   ideal_max_weight: np.log(cc.lambda_target_ideal/cc.
      lambda_nc)
-   pattern_total: cc.pattern_length + cc.pattern_pause
-   prob_spike_net: cc.pattern_length / cc.pattern_total * cc
      .activity_ratio
combiner_params:
    steps_per_pattern: !ee int(np.around(cc.pattern_total/cc.
         pattern_step))
combiner_types: uniform_unique
creator_params:
    inputparams:
        rate: !ee cc.rate_high
    inputtype: AdditivePoisson
    labels_to_create: !ee cc.labels_to_create
    length: !ee cc.pattern_length
    num_repr_per_label: 0
creator_types: mnist
duration: 15000.0
injector_params:
    inputparams:
        rate: !ee cc.rate_low
    inputtype: AdditivePoisson
injector_types: background
input_time_step: 500.0
manager_params:
    step: !ee cc.pattern_step
network_params:
    causelayer_params:
        database: ~/data/sbs/calibration/default-02-30ms
        delays: 0.0001
        inh_neuron_params:
            cm: 0.2
            e_rev_E: 0.0
            e_rev_I: -100.0
            i_offset: 0.0
            tau_m: 1.0
            tau_refrac: 5.0
            tau_syn_E: 5.0
            tau_syn_I: 5.0
            v_reset: -50.0
            v_rest: -50.0
            v_thresh: -40.0
        inh_weight: 100.0
        neuron_params:
            cm: 0.2
            e_rev_E: 0.0
            e_rev_I: -100.0
            i_offset: 0.0
            tau_m: 1.0
            tau_refrac: 30.0
            tau_syn_E: 30.0
            tau_syn_I: 30.0
            v_reset: -50.001
            v_rest: -50.0
            v_thresh: -50.0
        record_voltages: false
```

```
        saturating_synapses: false
    causelayer_type: BoltzMann_WTA_Inhibition
    connection_params:
        adjust_conversion_factor: !ee cc.acf
        eta: 0.0001
        max_weight_theo: !ee cc.ideal_max_weight * cc.
             max_weight_factor
        nearest_neighbors_only: !ee cc.nn_only
        nullcase: !ee np.log(cc.lambda_nc)
        num_weights: !ee 2**cc.weight_bits
        prob_spike_net: !ee cc.prob_spike_net
        prob_weight_update: true
        receptor_type: excitatory
        set_y_accum_factor: false
        tau_onset: 0.0
        tau_syn: !ee cc.tau_syn
        update_period: 2.0
    connection_type:
         coincidence_limited_weight_resolution_aggregating
    dt: !ee cc.dt
    homeostasis_params:
        create_exc: true
        eta: 0.01
        prob_spike_net: !ee cc.prob_spike_net
        rate: 500.0
        source_model: lookahead_poisson_generator
        source_model_kwargs:
            steps_lookahead: 10000
    homeostasis_type: linear
    num_z: !ee cc.num_z
    record_input: false
    record_spikes: true
    sim_step: 25.0
    source_type: SimulationLookaheadMultiPoissonVarRateSource
    statefactory_params:
        initializer_params:
            b:
                value: 0.0
    steps_per_report: 4
    steps_per_snapshot: 1
    take_snapshots: true
network_type: SimTrain
numpy_seed: 42
process_all_input_prior: false
```

Periodic Homeostasis Background Source

In order to conserve space we only print the differences to the Poisson background source case.

```
105c105
<         source_model: lookahead_poisson_generator
---
>         source_model: periodic_generator
107c107
<             steps_lookahead: 10000
---
>             offset: 1.0
```

A.1.5 Large networks

Oriented Strip

Parameters used for simulations in Figure 5.24. To conserve space we only show the differences to the 6-bit limited weight resolution case in Appendix A.1.2.

```
0a1,3
> # Lower inhibitory weight
> # Higher sigma
> # Fixed error in real nc calculations
2c5
<     - tau_syn: 30.0e-3
---
>     - tau_syn: 10.0e-3
10c13
<       num_z: 6
---
>       num_z: 90
32c35,36
< combiner_types: uniform_unique
---
>         pattern_weights: !ee np.r_[np.ones(60), np.ones(60) * 2, np.ones(60)]
> combiner_types: nonuniform
39,40c43,44
<     width_image: 17
<     width_strip: 3
---
>     width_image: 12
>     width_strip: 2
46c50
< duration: 10000
---
> duration: 15000.0
67c71
<         database: ~/data/sbs/calibration/default-02-30ms
---
>         database: ~/data/sbs/calibration/default-02-10ms
102c106
<         tau_onset: 0.
```

```
---
>         tau_onset: 1.0e-03
107d110
<         update_offset_per_sampler: !ee cc.update_period / cc.num_z
121c124
<         rate: 2000.0
---
>         rate: 1000.0
```

MNIST

Parameters used in Figure 5.27.

```
cache:
-   acf: 1.0
    activity_ratio: 1.0
    dt: 0.0001
    labels_to_create:
    - 0
    - 1
    - 2
    - 3
    - 4
    - 5
    - 6
    - 7
    - 8
    - 9
    max_weight_factor: 1.2
    nn_only: true
    num_z: 100
    pattern_length: 0.5
    pattern_pause: 0.1
    pattern_step: 0.1
    rate_high: 90.0
    rate_low: 10.0
    rate_nc: 10.0
    tau_syn: 0.01
    update_period: 2.0
    weight_bits: 6
-   rate_total: cc.rate_high + cc.rate_low
-   rfactor: 1+1./(cc.tau_syn*cc.rate_total)
-   rfactor_nc: 1+1./(cc.tau_syn*cc.rate_nc)
-   lambda_target_ideal_nn: (1.-np.exp(-cc.rfactor*cc.
      rate_total*cc.tau_syn))/cc.rfactor
-   lambda_target_ideal: cc.lambda_target_ideal_nn if cc.
      nn_only else cc.rate_total
        * cc.tau_syn
-   lambda_nc_nn: (1.-np.exp(-cc.rfactor_nc*cc.rate_nc*cc.
      tau_syn))/cc.rfactor_nc
-   lambda_nc: cc.lambda_nc_nn if cc.nn_only else cc.rate_nc
      * cc.tau_syn
-   ideal_max_weight: np.log(cc.lambda_target_ideal/cc.
      lambda_nc)
-   pattern_total: cc.pattern_length + cc.pattern_pause
-   prob_spike_net: cc.pattern_length / cc.pattern_total * cc
      .activity_ratio
combiner_params:
    steps_per_pattern: !ee int(np.around(cc.pattern_total/cc
        pattern_step))
combiner_types: uniform_unique
creator_params:
    clip_edge_pixel: 2
    inputparams:
        rate: !ee cc.rate_high
    inputtype: AdditivePoisson
    labels_to_create: !ee cc.labels_to_create
    length: !ee cc.pattern_length
    num_repr_per_label: 0
creator_types: mnist
duration: 15000.0
injector_params:
    inputparams:
        rate: !ee cc.rate_low
    inputtype: AdditivePoisson
injector_types: background
input_time_step: 500.0
manager_params:
    step: !ee cc.pattern_step
network_params:
    causelayer_params:
        database: ~/data/sbs/calibration/default-02-10ms
        delays: 0.0001
        inh_neuron_params:
            cm: 0.2
            e_rev_E: 0.0
            e_rev_I: -100.0
            i_offset: 0.0
            tau_m: 1.0
            tau_refrac: 5.0
            tau_syn_E: 5.0
            tau_syn_I: 5.0
            v_reset: -50.0
            v_rest: -50.0
            v_thresh: -40.0
        inh_weight: 100.0
        neuron_params_ids: 0
        record_voltages: false
        saturating_synapses: false
    causelayer_type: BoltzMann_WTA_Inhibition
    connection_params:
        adjust_conversion_factor: !ee cc.acf
        eta: 0.0001
        max_weight_theo: !ee cc.ideal_max_weight * cc.
              max_weight_factor
        nearest_neighbors_only: !ee cc.nn_only
        nullcause: !ee np.log(cc.lambda_nc)
        num_weights: !ee 2**cc.weight_bits
        prob_spike_net: !ee cc.prob_spike_net
        prob_weight_update: true
        receptor_type: excitatory
        set_y_accum_factor: false
        tau_onset: 0.0
        tau_syn: !ee cc.tau_syn
        update_period: 2.0
    connection_type:
          coincidence_limited_weight_resolution_aggregating
    dt: !ee cc.dt
    homeostasis_params:
        create_exc: true
        eta: 0.01
        prob_spike_net: !ee cc.prob_spike_net
        rate: 1000.0
        source_model: periodic_generator
        source_model_kwargs:
            offset: 1.0
    homeostasis_type: linear
    num_z: !ee cc.num_z
    record_input: false
    record_spikes: true
    sim_step: 25.0
    source_type: SimulationLookaheadMultiPoissonVarRateSource
    statefactory_params:
        initializer_params:
            b:
                value: 0.0
    steps_per_report: 4
    steps_per_snapshot: 1
    take_snapshots: true
network_type: SimTrain
numpy_seed: 42
process_all_input_prior: false
```

A.1.6 Non-negligible delays

Weight Scaling Sweep

An example parameter set for the sweep shown in Figure 5.28. All other parameter set differ only by the inverse arbitrary weight conversion factor (`acfi`) and the delay between inhibitory population and cause layer. Please note that a spike from the cause layer takes $\sim 1.2\,\mathrm{ms}$ to elicit a spike in the inhibitory population.

```
cache:
-   acfi: 1.5384615384615383
    activity_ratio: 1.0
    dt: 0.0001
    eta_b: 0.01
    labels_to_create:
    - 0
    - 1
    - 2
    - 3
    - 4
    - 5
    - 6
    - 7
    - 8
    - 9
    max_weight_factor: 1.2
    num_z: 10
    pattern_length: 0.5
    pattern_pause: 0.0
    pattern_step: 0.1
    period_per_sampler: 0.5
    rate_high: 90.0
    rate_low: 10.0
    tau_syn: 0.03
    update_period: 1.5
    weight_bits: 6
-   rate_nc: cc.rate_low
-   rate_total: cc.rate_high + cc.rate_low
-   rfactor: 1+1./(cc.tau_syn*cc.rate_total)
-   rfactor_nc: 1+1./(cc.tau_syn*cc.rate_nc)
-   lambda_target_ideal: (1.-np.exp(-cc.rfactor*cc.rate_total
      *cc.tau_syn))/cc.rfactor
-   lambda_nc: (1.-np.exp(-cc.rfactor_nc*cc.rate_nc*cc.
      tau_syn))/cc.rfactor_nc
-   ideal_max_weight: np.log(cc.lambda_target_ideal/cc.
      lambda_nc)
-   pattern_total: cc.pattern_length + cc.pattern_pause
-   prob_spike_net: cc.pattern_length / cc.pattern_total * cc
      .activity_ratio
combiner_params:
    steps_per_pattern: !ee int(np.around(cc.pattern_total/cc
         pattern_step))
combiner_types: uniform_unique
creator_params:
    inputparams:
        rate: !ee cc.rate_high
    inputtype: AdditivePoisson
    labels_to_create: !ee cc.labels_to_create
    length: !ee cc.pattern_length
    num_repr_per_label: 1
creator_types: mnist
duration: 5000.0
injector_params:
    inputparams:
        rate: !ee cc.rate_low
    inputtype: AdditivePoisson
injector_types: background
input_time_step: 500.0
manager_params:
    step: !ee cc.pattern_step
network_params:
    causelayer_params:
        database: ~/data/sbs/calibration/default-02-30ms
        delays: 0.0001
        inh_neuron_params:
            cm: 0.2
            e_rev_E: 0.0
            e_rev_I: -100.0
            i_offset: 0.0
            tau_m: 1.0
            tau_refrac: 5.0
            tau_syn_E: 5.0
            tau_syn_I: 5.0
            v_reset: -50.0
            v_rest: -50.0
            v_thresh: -40.0
        inh_weight: 100.0
        neuron_params:
            cm: 0.2
            e_rev_E: 0.0
            e_rev_I: -100.0
            i_offset: 0.0
            tau_m: 1.0
            tau_refrac: 30.0
            tau_syn_E: 30.0
            tau_syn_I: 30.0
            v_reset: -50.001
            v_rest: -50.0
            v_thresh: -50.0
        record_voltages: false
        saturating_synapses: false
    causelayer_type: BoltzMann_WTA_Inhibition
    connection_params:
        adjust_conversion_factor_inv: !ee cc.acfi
        eta: 0.001
        latest_pre_spike_only: false
        max_weight_theo: !ee cc.ideal_max_weight * cc.
             max_weight_factor
        nearest_neighbors_only: true
        nullcause: !ee np.log(cc.lambda_nc)
        num_weights: !ee 2**cc.weight_bits
        prob_spike_net: !ee cc.prob_spike_net
        prob_weight_update: true
        receptor_type: excitatory
        set_y_accum_factor: false
        tau_onset: 0.0
        tau_syn: !ee cc.tau_syn
        update_period: 2.0
    connection_type:
          coincidence_limited_weight_resolution_aggregating
    dt: !ee cc.dt
    homeostasis_params:
        create_exc: true
        eta: !ee cc.eta_b / cc.acfi
        prob_spike_net: !ee cc.prob_spike_net
        rate: 3000.0
    homeostasis_type: linear
    num_z: !ee cc.num_z
    record_input: false
    record_spikes: true
    sim_step: 25.0
    source_type: SimulationLookaheadMultiPoissonVarRateSource
    statefactory_params:
        initializer_params:
            b:
                value: 0.0
    steps_per_report: 4
    steps_per_snapshot: 1
    take_snapshots: true
network_type: SimTrain
numpy_seed: 42
process_all_input_prior: false
```

Null Case Rate Sweep

An example parameter set for the sweep shown in Figure 5.29. All other parameter set differ only by the set null cause rate in each synapse and the delay between inhibitory population and cause layer. Please note that a spike from the cause layer takes $\sim 1.2\,\mathrm{ms}$ to elicit a spike in the inhibitory population.

```
cache:
-   acf: 1.0
    activity_ratio: 1.0
    dt: 0.0001
    labels_to_create:
    - 0
    - 1
    - 2
    max_weight_factor: 1.2
    nn_only: true
    num_z: 3
    pattern_length: 0.5
    pattern_pause: 0.0
    pattern_step: 0.1
    rate_high: 30.0
    rate_low: 10.0
    rate_nc: 25.714285714285715
    tau_syn: 0.03
    update_period: 2.0
    weight_bits: 6
-   rate_total: cc.rate_high + cc.rate_low
-   rfactor: 1+1./(cc.tau_syn*cc.rate_total)
-   rfactor_nc: 1+1./(cc.tau_syn*cc.rate_nc)
-   lambda_target_ideal_nn: (1.-np.exp(-cc.rfactor*cc.
      rate_total*cc.tau_syn))/cc.rfactor
-   lambda_target_ideal: cc.lambda_target_ideal_nn if cc.
      nn_only else cc.rate_total
        * cc.tau_syn
-   lambda_nc_nn: (1.-np.exp(-cc.rfactor_nc*cc.rate_nc*cc.
      tau_syn))/cc.rfactor_nc
-   lambda_nc: cc.lambda_nc_nn if cc.nn_only else cc.rate_nc
      * cc.tau_syn
-   ideal_max_weight: np.log(cc.lambda_target_ideal/cc.
      lambda_nc)
-   pattern_total: cc.pattern_length + cc.pattern_pause
-   prob_spike_net: cc.pattern_length / cc.pattern_total * cc
      .activity_ratio
combiner_params:
    steps_per_pattern: !ee int(np.around(cc.pattern_total/cc.
         pattern_step))
combiner_types: uniform_unique
creator_params:
    clip_edge_pixel: 2
    inputparams:
        rate: !ee cc.rate_high
    inputtype: AdditivePoisson
    labels_to_create: !ee cc.labels_to_create
    length: !ee cc.pattern_length
    num_repr_per_label: 0
creator_types: mnist
duration: 15000.0
injector_params:
    inputparams:
        rate: !ee cc.rate_low
    inputtype: AdditivePoisson
injector_types: background
input_time_step: 500.0
manager_params:
    step: !ee cc.pattern_step
network_params:
    causelayer_params:
        database: ~/data/sbs/calibration/default-02-30ms
        delays: 0.001
        inh_neuron_params:
            cm: 0.2
            e_rev_E: 0.0
            e_rev_I: -100.0
            i_offset: 0.0
            tau_m: 1.0
            tau_refrac: 5.0
            tau_syn_E: 5.0
            tau_syn_I: 5.0
            v_reset: -50.0
            v_rest: -50.0
            v_thresh: -40.0
        inh_weight: 100.0
        neuron_params:
            cm: 0.2
            e_rev_E: 0.0
            e_rev_I: -100.0
            i_offset: 0.0
            tau_m: 1.0
            tau_refrac: 30.0
            tau_syn_E: 30.0
            tau_syn_I: 30.0
            v_reset: -50.001
            v_rest: -50.0
            v_thresh: -50.0
        record_voltages: false
        saturating_synapses: false
    causelayer_type: BoltzMann_WTA_Inhibition
    connection_params:
        adjust_conversion_factor: !ee cc.acf
        eta: 0.0001
        max_weight_theo: !ee cc.ideal_max_weight * cc.
             max_weight_factor
        nearest_neighbors_only: !ee cc.nn_only
        nullcause: !ee np.log(cc.lambda_nc)
        num_weights: !ee 2**cc.weight_bits
        prob_spike_net: !ee cc.prob_spike_net
        prob_weight_update: true
        receptor_type: excitatory
        set_y_accum_factor: false
        tau_onset: 0.0
        tau_syn: !ee cc.tau_syn
        update_period: 2.0
    connection_type:
         coincidence_limited_weight_resolution_aggregating
    dt: !ee cc.dt
    homeostasis_params:
        create_exc: true
        eta: 0.001
        prob_spike_net: !ee cc.prob_spike_net
        rate: 1000.0
        source_model: periodic_generator
        source_model_kwargs:
            offset: 1.0
    homeostasis_type: linear
    num_z: !ee cc.num_z
    record_input: false
    record_spikes: true
    sim_step: 25.0
    source_type: SimulationLookaheadMultiPoissonVarRateSource
    statefactory_params:
        initializer_params:
            b:
                value: 0.0
    steps_per_report: 4
    steps_per_snapshot: 1
    take_snapshots: true
network_type: SimTrain
numpy_seed: 42
process_all_input_prior: false
```

Acknowledgments (Danksagungen)

Professor Meier für die Betreuung meiner Arbeit.

Meinen Eltern für die Unterstützung all die Jahre.

Mihai for general awesomeness.

Vitali für die langen Nächte im Büro and otherwise.

Eric für die Verteidigung des Clusters vor meinesgleichen…

Christian, Ilja, Mihai, Roman und Vitali für das Korrekturlesen dieser Thesis.

Der TMA-Gruppe für die interessanten Diskussionen.

Der gesamten Electronic Vision(s) Gruppe für die einzigartige (Arbeits-)Atmosphäre.

Statement of Originality (Erklärung):

I certify that this thesis, and the research to which it refers, are the product of my own work. Any ideas or quotations from the work of other people, published or otherwise, are fully acknowledged in accordance with the standard referencing practices of the discipline.

Ich versichere, daß ich diese Arbeit selbständig verfaßt und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.

Heidelberg, 29th April 2015

.....
(signature)