RUPRECHT-KARLS-UNIVERSITÄT HEIDELBERG

Venelin Petkov

Toward Belief Propagation on Neuromorphic Hardware

Diplomarbeit

KIRCHHOFF-INSTITUT FÜR PHYSIK

Faculty of Physics and Astronomy

University of Heidelberg

Diploma thesis

in Physics

submitted by

Venelin Petkov

born in Varna, Bulgarien

Toward Belief Propagation on Neuromorphic Hardware

This diploma thesis has been carried out by Venelin Petkov at the
Kirchhoff Institute for Physics
Ruprecht-Karls-Universität Heidelberg
under the supervision of
Prof. Dr. Karlheinz Meier

Toward Belief Propagation on Neuromorphic Hardware

The inference of probabilistic quantities from noisy sensory data is one of the most important tasks performed by biological neural networks. The computations performed in such networks can be investigated with belief propagation methods. In this thesis, preparatory studies toward implementing this algorithm on neuromorphic hardware are presented. Firstly, a theoretical treatment of statistical inference models is introduced. Furthermore, software simulations aimed at validating this approach are presented. Finally, calibration methods necessary to prepare the utilized FACETS hardware system for the emulation of these models are provided.

Implementierung von Belief Propagation auf Neuromorpher Hardware

Eine der wichtigsten Aufgaben biologischer neuronaler Netze ist die Abschätzung von Groessen aus verrauschten sensorischen Daten. Die Informationsverarbeitung solcher Netze kann durch Methoden der *Belief Propagation* untersucht werden. Die vorgestellte Arbeit zeigt Vorstudien auf, die zur Implementierung jener Algorithmen auf neuromorpher Hardware dienen. Zunächst werden theoretische Betrachtungen statistischer Inferenzmodelle vorgestellt. Desweiteren werden Softwaresimulationen durchgeführt, um diesen Ansatz zu validieren. Abschliessend werden die erforderlichen Kalibrationsmethoden für das in dieser Arbeit verwendete FACETS Hardwaresystem beschrieben.

Contents

1 Introduction

One of the most exciting scientific endeavours of our time is the study of the operational principles of the brain. Not until very recently was it even possible to hope that the arguably most intricate system known to humankind would be amenable to understanding in a sufficiently detailed manner. A truly interdisciplinary field, neuroscience employs a great variety of theoretical and experimental concepts that are usually found in the domain of disciplines ranging from physics and mathematics to biology and electrical engineering. This great confluence of ideas is driven by necessity, since the research of neural information processing requires the examination of many different facets of the underlying phenomena, as they happen on varying scales, starting from individual neuron compartments up to the large networks comprising an organism's nervous system.

Since the very first comprehensive biophysical model of neural signaling was presented by Hodgkin and Huxley in 1952, there has been an extensive study of the properties of the neural cell. The evidence amassed so far has led to a very detailed understanding of the electro-chemical basis for signal generation and propagation *in vivo* that can be verified in minute details by modern experimental techniques such as the patch-clamp method, which is used for the investigation of single ion channels (a word of caution is in order, however, since a major feature of neuron signaling, namely the back-propagation of dendrite spikes, has been uncovered only recently). Although this confidence in how individual neurons are functioning paves the way for theoretical studies of neural microcircuits, the comprehensive investigation of large networks of neurons is inaccessible by the currently available research tools used in both theoretical and empirical neuroscience. Under these circumstances, a different approach is necessary for the understanding of neural information processing.

As the biochemical details of neuron operation are often irrelevant in the context of larger networks, neural dynamics can be described by abstract mathematical neuron models that emulate most of the features of their biological counterparts with sufficient accuracy. This representation enables the development of mathematical theories concerning the operational principles of *artificial neural networks*. A number of these theories are statistical in nature and are often inspired by the information-theoretical discipline of *machine learning*. Still, their mathematical analysis, as currently understood, often reaches its limits when faced with large ensembles of complex non-linear units. Therefore an approach which involves the computational modeling of networks of spiking neurons, is often used as a powerful alternative.

Computer simulations are now firmly established as the third pillar of science, be-

sides theory and experiment, and often provide a cost-effective alternative to time-consuming empirical studies. Moreover, they enable the testing of theoretical computational paradigms *in silico*, in particular the neural implementation of the belief propagation algorithm discussed in this thesis. In addition, large-scale simulations of biologically inspired models are often used to investigate the behavior of large constituents of the nervous system, such as patches of the human neo-cortex.

The performance of modern computing facilities often reaches its limits when used in conjunction with parameter studies involving large networks of neurons. An important side-effect of such simulations is the cost that is incurred by digital electronics in terms of power consumption. Additionally, the run-time of even a single trial used to explore a large parameter space is often considerably longer. Clearly, a very different approach is needed if such experiments are to be performed within reasonable time span.

Instead of *computing* the equations that describe a neural network model mathematically, it is possible to build a device that actually *behaves* like a network of idealized neurons. A analog VLSI implementation of such a device would allow it to operate much faster than biological systems due to the different order of magnitude of the involved neuron time constants which are realized as electronic components. If the individual neuron and synapse circuits are designed to operate asynchronously, with no central clock governing their dynamics, the time needed for the completion of an experiment becomes largely independent of the number of artificial neurons that it contains. A prototype system of this type has been developed by the Electronic Vision(s) group at the University of Heidelberg in the framework of the FACETS [14] collaboration. It represents a stepping stone toward much larger wafer-scale device, currently in development, which is expected to be delivered within the BrainScaleS [4] project.

Thesis Outline

The work described in this thesis is concerned with implementing statistical inference models in artificial neural networks that are emulated on neuromorphic hardware. The theoretical cornerstone of the involved studies is *belief propagation*, which is an efficient parallel algorithm for the marginalization of joint probability distributions. The ultimate goal of testing statistical inference models on neuromorphic hardware is motivated by the insight provided into the operation of the brain, since the inference of various quantities from noisy sensory data is a major computational task performed by the nervous system. The presented work draws upon several different research areas, including statistical theory, software simulation of neural networks, and the operation of neuromorphic hardware.

Chapter 2 provides the theoretical background needed to understand the proposed models (based on [37]) and the algorithm that underlies the belief propagation framework in general. It discusses a novel model that is well-suited for implementation on the available neuromorphic hardware and concludes with a discussion about how statistical inference can be embedded in a network of spiking neurons.

Chapter 3 describes the details of the neural implementation of the presented models, as well as simulation results that serve to validate the ability of neural networks to implement the belief propagation algorithm.

Chapter 4 is concerned with the operation of the neuromorphic hardware platform FHW-1v4, developed by the Electronic Vision(s) group at the University of Heidelberg. Prior to performing scientifically meaningful experiments on this device, a calibration of the analog neuromorphic circuit is needed. The corresponding methods, that have been developed and implemented by the author, as well as their results, are discussed in detail.

The final chapter provides a short summary of the previously presented results and discusses remaining issues that need to be addressed in the future.

2 Statistical Inference and Neural Networks: A Theoretical Background

The ability to estimate physical variables such as velocities and relative distances is a crucial element of the survival skill set utilized by many biological organisms to thrive in a complex and uncertain environment. Since the assessment of these quantities from noisy and unreliable data is thus fostered by natural selection, evolution has lead to the development of *statistical inference* mechanisms that are performed by biological neural network circuits in the larger context of animal nervous systems.

Unfortunately, there is no universally agreed upon view of how statistical inference operates *in vivo* [37], despite the existence of a variety of advanced mathematical methods dealing with uncertainty and probability. A promising avenue of research is provided by the discipline of *Bayesian inference*, which has been used recently for solving many scientific and engineering problems. In contrast to the frequentist approach to probability, Bayesian inference starts with an assumption about the distribution of the quantity of interest and then iteratively updates this distribution, based on the observed data. Since a close analogy with learning in nervous systems is immediately obvious, Bayesian inference forms an appealing theoretical framework for the description of probabilistic events in biological systems. An open question in neuroscience is therefore the characterization of possible algorithms performing this task in biological neural networks.

The general form of any candidate algorithm carrying out statistical inference *in vivo* is constrained by the topological and physical properties of the underlying neural circuits. A minimal set of rules for the evaluation of possible algorithms can be easily derived. First of all, only local computation must be allowed for, since no biologically plausible mechanisms for keeping track of global variables in a neural network exist. This leads in a natural way to the parallelization of the algorithm, an inherent feature of biological nervous systems. Second, the processing of information should be based on message passing among neurons, thus respecting the main task of a neural cell: the generation and propagation of action potentials. Finally, the algorithm should ideally operate on graphs, since these are the most natural algebraic structures describing neural networks.

One powerful algorithm that satisfies the above criteria is the *sum-product algorithm*. It is used extensively in fields as diverse as machine learning, signal processing, and digital communication, and operates on a very general class of graphical models known as *factor graphs*. Due to its generality, it can be used to derive a number of well-known schemes such as the forward/backward algorithm, the Viterbi algorithm, the Kalman

filter, and even special cases of the Fast Fourier Transform [25].

The main focus of this thesis is the investigation of the plausibility of implementing statistical inference by the sum-product algorithm on neuromorphic hardware. This chapter aims at providing the necessary background information for the understanding of the performed neural network simulations in software and the consecutive discussion of the operation of the hardware device.

2.1 Bayesian Inference

The Bayesian interpretation of probability arises when considering uncertain events. In contrast to the frequentist view, these events are not repeatable (this is not known *a priori*). Still, some initial assumption about their distribution is needed, often provided by prior experience or the form of the problem. Given a new set of data, the distribution is updated to reflect the increase in knowledge.

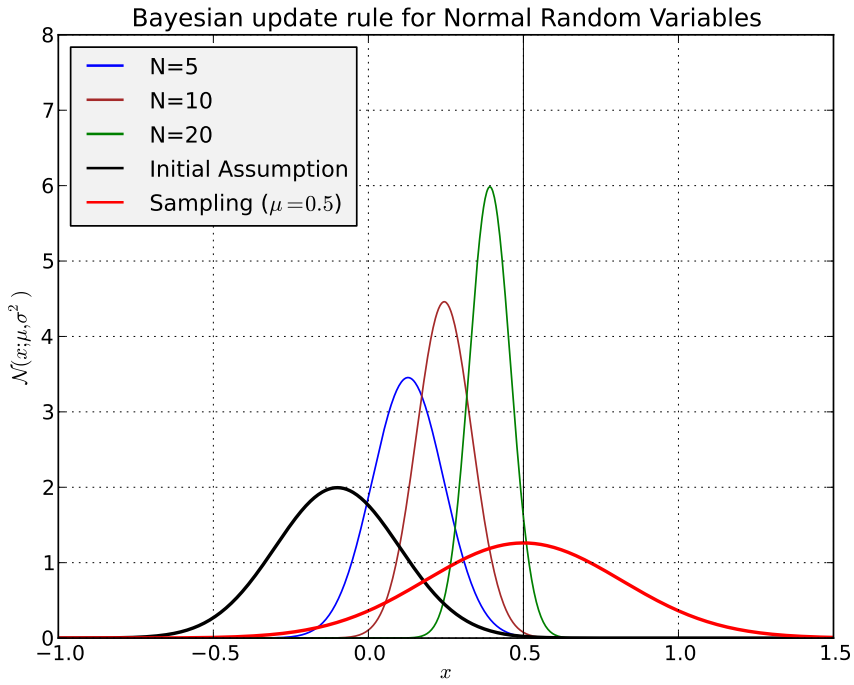


Figure 2.1: Iterative update of a Gaussian distribution using Bayes' rule. A sampling distribution with a known variance σ^2 has an unknown mean μ. It generates N data points X that are used to update the prior distribution $\mathcal{N}(\mu; \mu_0, \sigma_0^2)$ which is constructed by assuming the mean μ_0. The posterior distribution $p(\mu|X) = \mathcal{N}(\mu|\mu_N, \sigma_N^2)$ is then updated with each sample and approaches the unknown mean μ.

On more formal terms, a set of parameters θ describes a statistical model by the *prior* distribution $p(\theta)$. The effect of the observed data y on the model is described by the *likelihood function* $p(y|\theta)$. The *posterior* distribution $p(\theta|y)$ is proportional to the product of the prior distribution and the likelihood: $p(\theta|y) \propto p(y|\theta)p(\theta)$. This state of affairs is expressed by the Bayes theorem, which describes the update of the "belief" $p(\theta)$ given a new set of data:

$$p(\theta|y) = \frac{p(y|\theta)p(\theta)}{p(y)} \tag{2.1}$$

The normalization factor $p(y)$ can be calculated when both sides of the equation are integrated with respect to θ:

$$p(y) = \int p(y|\theta)p(\theta)d\theta \tag{2.2}$$

It should be pointed out that in the frequentist view, the estimation provides a particular value for θ, so that the uncertainty depends on the distribution of possible data sets, while Bayesian inference deals with a single data set and the uncertainty in the parameters is expressed through the distribution $p(\theta)$ [3].

A further important feature of the Bayesian approach is that it allows for making predictions based on the the current uncertainty about the parameters θ. Starting with the posterior distribution $p(\theta|y)$ and a quantity z that depends on θ through a sampling distribution $p(z|\theta)$, the distribution of future values of z, based on the current knowledge y can be derived as:

$$p(z|y) = \int p(z|\theta)p(\theta|y)d\theta \tag{2.3}$$

(Provided that y and z are conditionally independent on θ: $p(y, z|\theta) = p(y|\theta)p(z|\theta)$)

An immediate application of the Bayesian paradigm is the Kalman filter, which is used to estimate the time series of true system states from noisy measurement data. The algorithm uses a prediction-correction method based on equations 2.3 and 2.1. Its starting assumption (leading to the prior distribution) is derived from the physical laws that govern the dynamics of the system.

Real-world problems usually imply complex relations between probability distributions that may pose significant difficulties for the calculation of the quantities of interest. Graphical models allow for the straightforward application of Bayesian inference to such problems by casting them in a representation that is both intuitive and general. Quite often, seemingly unrelated algorithms have been discovered to be a particular instance of a more general graphical concept [17]. An important class of such models are Bayesian networks.

2.1.1 Bayesian Networks

A probabilistic graphical model represents the factorization of a joint probability distribution over a set of random variables into a product of distributions, each of which depends on a subset of these variables. Each node (*vertex*) is identified with a random variable, while (graph) *edges* express the relations among them. Since a graph can be either *directed* or *undirected*, there are two possible formulations of probabilistic graphical models:

- Bayesian networks (directed graphs)
- Markov random fields (undirected graphs)

Directed graphs are used to represent causal relations between random variables, while undirected graphs pose soft constraints between them [3]. Under specific conditions, Bayesian networks can be converted to Markov random fields and vice versa, but there exists a more general representation known as a factor graph that is especially well-suited for expressing statistical inference problems and underlies a more general framework that goes beyond probabilistic relations.

Any joint probability $p(x, y)$ over two variables can be factorized into a conditional probability $p(x|y)$ and a *marginal* probability $p(x)$ by the product rule:

$$p(x, y) = p(y|x)p(x) \tag{2.4}$$

In a directed probabilistic graphical model, the edges point toward random variables conditioned on other random variables. This rule, together with equation 2.4, forms the basis for constructing any Bayesian network from a joint probability distribution. For instance, the joint distribution $p(x, y)$ forms the simplest possible directed graph:

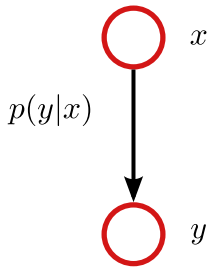


Figure 2.2: Bayesian network representing $p(y|x)p(x)$

Conversely, the factorization of a given Bayesian network (see figures 2.3 and 2.4 for larger graphs) can be found by examining each node and noticing which of the remaining random variables point to it. The latter are known as *parent nodes* and are often denoted with $\mathrm{pa}(x)$, where x is the examined variable. Since this relation describes the conditional

probability $p(x|\text{pa}(x))$, the graphical model depicts the product:

$$p(x_1, \ldots, p_N) = \prod_{i=1}^{N} p(x_i|\text{pa}(x_i)) \tag{2.5}$$

Starting with equation 2.4, any joint probability distribution can be factorized and thus represented by a Bayesian network:

$$p(x_1, \ldots, x_n) = p(x_n|x_1, \ldots, x_{n-1}) \ldots p(x_2|x_1)\, p(x_1) \tag{2.6}$$

If all conditional dependencies are fulfilled, the resulting directed graph is *fully connected* and *acyclic*.

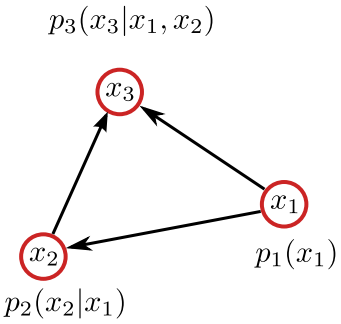


Figure 2.3: Factorization and Bayesian network representation of the fully connected model $p(x_1, x_2, x_3) = p_3(x_3|x_1, x_2)p_2(x_2|x_1)p_1(x_1)$

In most practical problems, however, the Bayesian network is usually *sparse*, so that each conditional distribution does not depend on all variables provided by the factorization rule as in figure 2.4

Such sparse graphs may provide important insights into the structure of the problem under investigation and their conditional independence properties lead to efficient algorithms for calculating marginal distributions, as it will be discussed in the following.

2.1.2 Belief Propagation

The central question in statistical inference is finding the marginal distribution of some random variable of interest, provided some initial assumption ("belief") and data samples. The most obvious approach to this problem is to sum over the the joint distribution $p(x_1, \ldots, x_n)$ for all possible values of all the other random variables:

$$p_k(X_k) = \sum_{x_1 \in X_1} \cdots \sum_{x_{k-1} \in X_{k-1}} \sum_{x_{k+1} \in X_{k+1}} \cdots \sum_{x_n \in X_n} p(x_1, x_{k-1}, x_{k+1}, x_n) \tag{2.7}$$

Note that this calculation scales exponentially with the number of variables and is thus untenable for real-world networks with more than a few nodes. As an example, let X be an i.i.d. multinomial with K states. It follows that the total cost of calculation

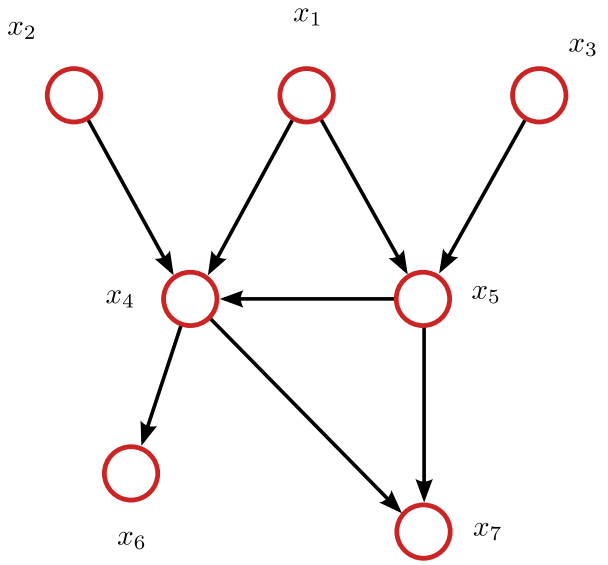


Figure 2.4: A sparse Bayesian network representing the factorization:
$p(x_1)p(x_2)p(x_3)p(x_4|x_1,x_2,x_3)p(x_5|x_1,x_3)p(x_6|x_4)p(x_7|x_4,x_5)$

is $\propto K^{n-1}$. Is there a more efficient computational model that ideally scales linearly in the number of variables? It turns out that the conditional independence properties of the graphical model allow not only for many different efficient computational schemes, but also for an important interpretation of the underlying algorithms.

The easiest approach to belief propagation is to consider Bayesian inference on a chain:

Figure 2.5: Bayesian inference on a chain

The factorization of its joint probability distribution can be read directly from the graph (see equation 2.5) and is given by $p(x_n|x_{n-1})p(x_{n-1}|x_{n-2})\dots p(x_2|x_1)p(x_1)$ since each node depends only on the preceding one. If we are interested in the marginal probability $p(x_n)$, then we have to sum over all variables $x_{n-1}\dots x_1$:

$$p(x_n) = \sum_{x_{n-1}} \cdots \sum_{x_1} p(x_n|x_{n-1})p(x_{n-1}|x_{-2})p(x_2|x_1)p(x_1) \tag{2.8}$$

A simple reordering of the product of summation operations affords a new insight into

the marginalization procedure:

$$p(x_n) = \sum_{x_{n-1}} p(X_n|x_{n-1}) \left[\underbrace{\sum_{x_{n-2}} p(x_{n-1}|x_{n-2}) \left[\ldots \left[\underbrace{\sum_{x_1} p(x_2|x_1)p(x_1)}_{\mu_{x_1 \to x_2}(x_2)} \right] \right]}_{\mu_{x_{n-1} \to x_n}(x_n)} \right] \tag{2.9}$$

Each node follows a *local* update rule $\mu_{x_{k-1} \to x_k}(x_k) = \sum_{x_{k-1}} p(x_k|x_{k-1}) \mu_{x_{k-2} \to x_{k-1}}(x_{k-1})$ which marginalizes the conditional probability at this node, multiplied by the marginal calculated for the preceding one. If the local marginal distributions μ are viewed as messages sent from one node to another, then the propagation of these probability distributions along the chain with each summation can be interpreted as a *message passing algorithm*. It is obvious that when these summations are performed sequentially, the complexity of the algorithm increases only linearly with the number of variables, since the computations at each node are the same.

The generalization of this scheme for more general types of Bayesian networks, as well as the invention of robust local computation rules has been pioneered by Judea Pearl [34], [35], whose work is widely known as *belief propagation*, since in his framework the prior distributions are designated as belief in the corresponding quantity. The idea of reordering summation and product factors to achieve efficient computational algorithms, optimized for the structure of the graph has been exploited in a number of different fields such as statistical physics, digital communication, machine learning, etc. A unified representation of all these ideas has been achieved by transforming various types of graphical models in a more general representation known as *factor graphs*. The latter are undirected and consist of two types of nodes, which represent either random variables or the relationships between them.

2.2 Factor Graphs

Not only probabilistic graphical models, but also many well-known methods such as the Kalman filter, Tanner graphs or Fast Fourier transforms can be cast into factor graphs on which the *sum-product algorithm* provides a unified procedure for performing efficient marginalization tasks. This generality, as well as the distributed nature of the algorithm underpin an elegant theoretical paradigm for viewing statistical inference in neural networks.

2.2.1 Bipartite Factor Graphs

Leaning on the **definition** from [26], let $x_1, x_2, \ldots, x_n$ be a collection of variables so that each variable x_i takes on values in a domain A_i. Let $g(x_1, \ldots, x_n)$ be an R-valued function of these variables, $g : S \mapsto R$, where the domain S of g is designated as the *configuration space* and each element $s \in S$ represents a particular *configuration* of the variables:

$$S = A_1 \times A_2 \times \cdots \times A_n \tag{2.10}$$

Identifying R with the set of real numbers does not lead to significant loss of generality [26] and since summation in this field is well-defined, for each variable x_i of g there exists a *marginal* function $g_i(x_i)$. The value of the marginal function $g_i(a)$ for a particular value $x_i = a$ with $a \in A_i$ is calculated by summing $g(x_1, \ldots, x_n)$ of a configuration subspace of S where $x_i = a$. This operation can be expressed by:

$$g_i(x_i) := \sum_{\{x_k; k \in n\} \setminus x_i} g(x_1, \ldots, x_n) \equiv \sum_{\sim\{x_i\}} g(x_1, \ldots, x_n) \tag{2.11}$$

The function $g(x_1, \ldots, x_n)$ is often designated as the *global* function and may factorize into a product of *local* functions, each of which is a function of a subset $X_j \subset \{x_1, \ldots, x_n\}$ of the variables of g. If J is a discrete index set, this product can be written as:

$$g(x_1, \ldots, x_n) = \prod_{j \in J} f_j(X_j) \tag{2.12}$$

The representation of this factorization by a graph is achieved using the following definition.

Definition: A factor graph is a *bipartite* graph (in which there may be no edges connecting nodes of the same type). It has a *variable node* for each variable x_i and a *factor node* for each local function f_j. An edge connecting variable node x_i to factor node f_j exists if and only if x_i is an argument of f_j.

The visualization of such graphs is straightforward and the following standard example will be used to illustrate the sum-product algorithm:

$$g(x_1, x_2, x_3, x_4, x_5) = f_A(x_1) f_B(x_2) f_C(x_1, x_2, x_3) f_D(x_3, x_4) f_E(x_3, x_5) \tag{2.13}$$

2.2.2 The Sum-Product Algorithm

For cycle-free graphs, the sum-product algorithm provides a recipe to calculate exactly all marginal functions $g_i(x_i)$ associated with a global function $g(x_1, \ldots, x_n)$. In some cases, it can be used iteratively without modification even for factor graphs with cycles, where it delivers an approximation of the exact solution [39]. A formal derivation of the

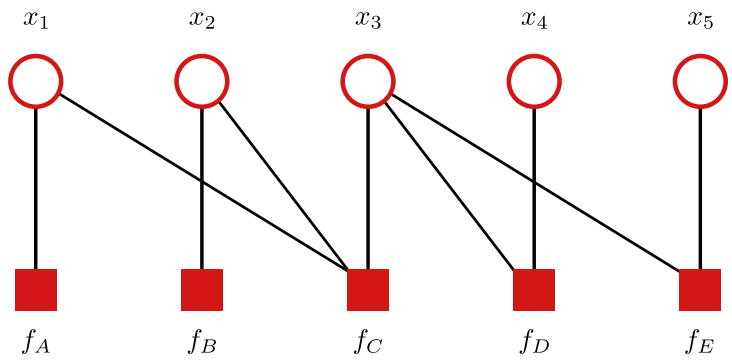


Figure 2.6: Example bipartite factor graph

algorithm from first principles is beyond the scope of this thesis, but it can be found in [3], [25], [26], and [39].

The sum-product algorithm relies strongly on the tree structure of a graph, as in example 2.13. The variable x_i for which the marginal function $g_i(x_i)$ is computed is first designated as the *root node* of the tree. Thereafter the calculation proceeds similarly to the Bayesian inference on a chain presented previously (equation 2.9). Starting from example 2.13, the marginal value $g_1(x_1)$ can be computed according to:

$$g_1(x_1) = f_A(x_1)\left[\sum_{x_2} f_B(x_2)\left[\sum_{x_3} f_C(x_1,x_2,x_3)\left[\sum_{x_4} f_D(x_3,x_4)\right]\left[\sum_{x_5} f_E(x_3,x_5)\right]\right]\right] \tag{2.14}$$

The summation notation from 2.11 can be used to convert the expression to a more convenient form:

$$g_1(x_1) = \underbrace{f_A(x_1)}_{\mu_{f_A\to x_1}(x_1)} \sum_{\sim\{x_1\}} \underbrace{\left[f_C(x_1,x_2,x_3)\ \underbrace{f_B(x_2)}_{\mu_{f_B\to x_2(x_2)}} \underbrace{\left[\sum_{\sim\{x_3\}} f_D(x_3,x_4)\right]}_{\mu_{f_D\to x_3}(x_3)} \underbrace{\left[\sum_{\sim\{x_3\}} f_E(x_3,x_5)\right]}_{\mu_{f_E\to x_3}(x_3)} \right]}_{\mu_{f_C\to x_1}(x_1)} \tag{2.15}$$

In analogy to equation 2.9, the nesting of summation operations that produce local marginals can be viewed as passing of messages from one node to the next. In this case all messages converge on node x_1, for which the marginal function $g_i(x_i)$ is the product of these messages: $g_i(x_i) = \mu_{f_A\to x_1}(x_1) \times \mu_{f_C\to x_1}(x_1)$.

The computation starts at the leaf nodes. Each leaf variable node sends a unit function $\mu_{x_j\to f_k}(x_j) = 1$ to its parent factor node. The leaf factor nodes, which depend only on

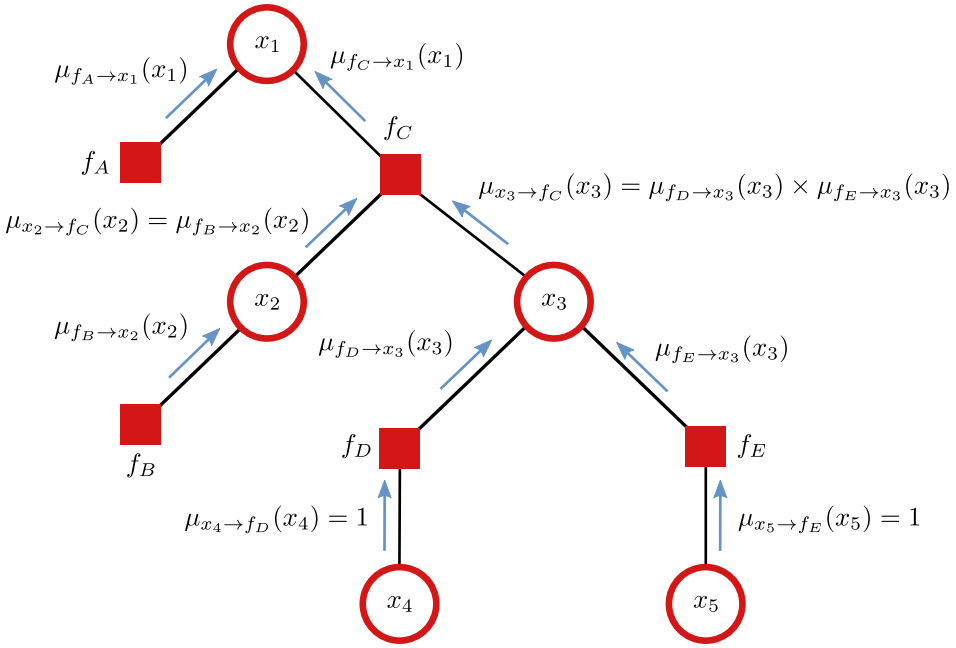


Figure 2.7: Visualization of message passing on a tree-structured graph according to the single-i-algorithm.

one variable, send a description of the factor function to that parent variable.

Similarly, the computations performed by non-leaf nodes of a factor graph are of two varieties, each specific for the type of node:

- A variable node x_j simply forms the product of the s incoming messages $\prod_{k=1}^{s} \mu_{f_k \to x_j}(x_j)$ and passes it to its parent factor node f_k.
- A factor node f_k receives q messages from a set of variable nodes $X = \{x_1, \ldots, x_q\}$ and multiplies their product $\prod_{i=1}^{q} \mu_{x_i \to f_k}(x_i)$ with the function $f_k(x_j, X)$ that describes it. Then it calculates the sum $\sum_{\sim\{x_j\}} f_k(x_j, X) \prod_{i=1}^{q} \mu_{x_i \to f_k}(x_i)$ and passes the result $\mu_{f_k \to x_j}(x_j)$ to its parent variable node x_j.

The described version of the sum-product algorithm may be referred to as the "single-i-algorithm" [26], since it results in the calculation of a single marginal $g_i(x_i)$. It might be sufficient for the forward computation of belief updates on a time-series chain like a hidden Markov model or a Kalman filter, but in other models it might be desirable to have all marginals calculated simultaneously.

Using Pearl's [35] metaphor, each node in a factor graph may be viewed as a processor that waits for incoming messages and performs a well-defined operation on them before sending them away to other nodes. Since in such a case there is no centralized supervision (no global variables), the computations are entirely local and distributed. Unlike the

simple tree presented above, the general version of the sum-product does not demand parent-child relations between nodes. Messages are sent in both directions along an edge, as soon as updates are available. For convenience, let the algorithm operate in discrete time, so that each step can be isolated. The calculation presented below is then known as *flooding schedule*.

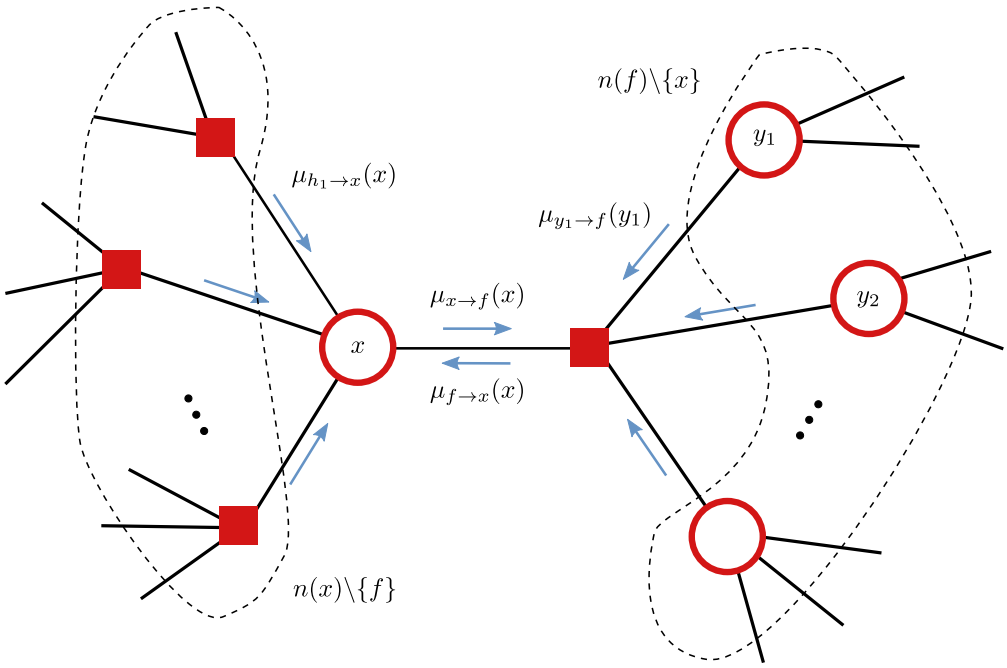


Figure 2.8: Propagation of sum-product messages using the flooding schedule regime.

As in the single-i case, the leaf nodes initiate message passing. Each non-leaf node waits for incoming messages until it has received a message on each but one edge. Designating the neighbouring node along this edge as a temporary parent, the node performs its computations, depending on its type (see above) and then waits for a message to come along this same edge. Once it has arrived, the node is free to perform its computation and send the resulting marginals over each of the remaining edges. The algorithm terminates when two messages have passed along each edge in opposing directions. At the end of the calculation, all marginals $g_i(x_i)$ are available at each variable node x_i.

Formal Definition of the Sum-Product Rule:

Let $\mu_{x\to f}(x)$ be the message from a variable node x to a factor node f. Let $\mu_{f\to x}(x)$ denote the message sent from node f to node x. Let $n(v)$ denote the set of neighbors of a given node v (regardless of its type) in a factor graph. The local computation rules, together with the flooding schedule described previously constitute the sum-product algorithm:

Sum-product update rule

A node v in a factor graph sends a message along an edge e that consist of a product of the local factor function at v (which is the unit function if the node v is a variable node) with all messages collected at v on edges other than e according to the following rules:

Variable to local function

$$\mu_{x\to f}(x) = \prod_{h\subset n(x)\setminus\{f\}} \mu_{h\to x}(x) \tag{2.16}$$

Local function to variable

$$\mu_{f\to x}(x) = \sum_{\sim\{x\}} f(X) \prod_{y\in n(f)\setminus\{x\}} \mu_{y\to f}(y) \tag{2.17}$$

As an illustration of the algorithm, consider the standard example 2.13 used so far and figure 2.9 that depicts each calculation step. The time sequence of the computations is initiated by the leaf nodes.

Step 1 (initiation):

$$\begin{aligned}
\mu_{f_A\to x_1}(x_1) &= \sum_{\sim\{x_1\}} f_A(x_1) = f_A(x_1)\\
\mu_{f_B\to x_2}(x_2) &= \sum_{\sim\{x_2\}} f_B(x_2) = f_B(x_2)\\
\mu_{x_4\to f_D}(x_4) &= 1\\
\mu_{x_5\to f_E}(x_5) &= 1
\end{aligned}$$

Step 2:

$$\begin{aligned}
\mu_{x_1\to f_C}(x_1) &= \mu_{x_A\to x_1}(x_1)\\
\mu_{x_2\to f_C}(x_2) &= \mu_{x_B\to x_2}(x_2)\\
\mu_{f_D\to x_3}(x_3) &= \sum_{\sim\{x_3\}} \mu_{x_4\to f_D}(x_4) f_D(x_3, x_4)\\
\mu_{f_E\to x_3}(x_3) &= \sum_{\sim\{x_3\}} \mu_{x_5\to f_E}(x_5) f_E(x_3, x_5)
\end{aligned}$$

Step 3:

$$\mu_{f_C \to x_3}(x_3) = \sum_{\sim\{x_3\}} \mu_{x_1 \to f_C}(x_1)\mu_{x_2 \to f_C}(x_2) f_C(x_1, x_2, x_3)$$
$$\mu_{x_3 \to f_C}(x_3) = \mu_{f_D \to x_3}(x_3)\mu_{f_E \to x_3}(x_3)$$

Step 4:

$$\mu_{f_C \to x_1}(x_1) = \sum_{\sim\{x_1\}} \mu_{x_3 \to f_C}(x_3)\mu_{x_2 \to f_C}(x_2) f_C(x_1, x_2, x_3)$$
$$\mu_{f_C \to x_2}(x_2) = \sum_{\sim\{x_2\}} \mu_{x_3 \to f_C}(x_3)\mu_{x_1 \to f_C}(x_1) f_C(x_1, x_2, x_3)$$
$$\mu_{x_3 \to f_D}(x_3) = \mu_{f_C \to x_3}(x_3)\mu_{f_E \to x_3}(x_3)$$
$$\mu_{x_3 \to f_E}(x_3) = \mu_{f_C \to x_3}(x_3)\mu_{f_D \to x_3}(x_3)$$

Step 5:

$$\mu_{x_1 \to f_A}(x_1) = \mu_{f_C \to x_1}(x_1)$$
$$\mu_{x_2 \to f_B}(x_2) = \mu_{f_C \to x_2}(x_2)$$
$$\mu_{f_D \to x_4}(x_4) = \sum_{\sim\{x_4\}} \mu_{x_3 \to f_D}(x_3) f_D(x_3, x_4)$$
$$\mu_{f_E \to x_5}(x_5) = \sum_{\sim\{x_5\}} \mu_{x_3 \to f_E}(x_3) f_E(x_3, x_5)$$

Termination:

$$g_1(x_1) = \mu_{f_A \to x_1}(x_1)\mu_{f_C \to x_1}(x_1)$$
$$g_2(x_1) = \mu_{f_D \to x_2}(x_2)\mu_{f_C \to x_2}(x_2)$$
$$g_3(x_1) = \mu_{f_C \to x_3}(x_3)\mu_{f_D \to x_3}(x_3)\mu_{f_E \to x_3}(x_3)$$
$$g_4(x_1) = \mu_{f_D \to x_4}(x_4)$$
$$g_5(x_1) = \mu_{f_E \to x_5}(x_5)$$

After termination, the marginals $g_i(x_i)$ are available for all variables x_i.

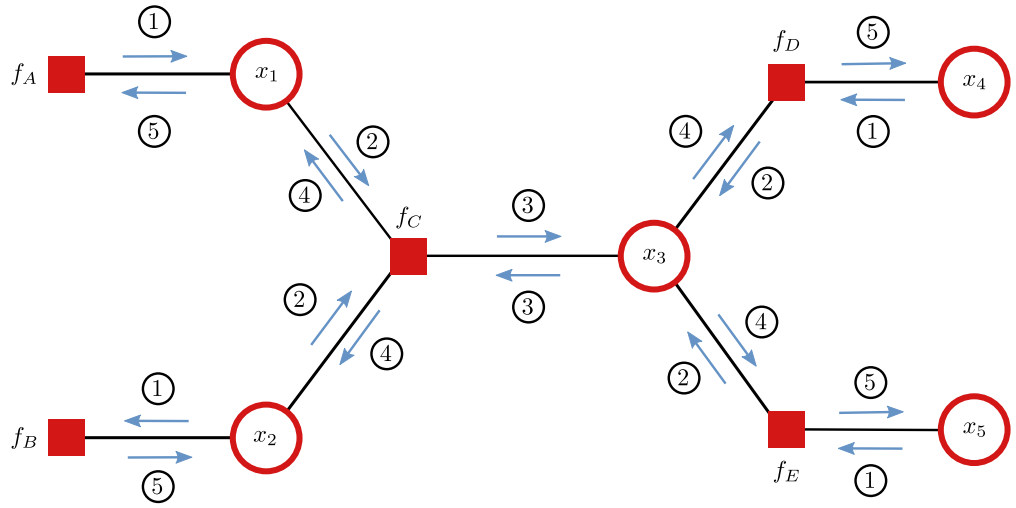


Figure 2.9: Message passing schedule visualization example

2.2.3 Forney Factor Graphs

The first step in implementing factor graph models in neural networks is to choose a convenient representation that lends itself more naturally to this task. One such representation is known as a *normal graph* [17], or, after the name of its discoverer, as a *Forney factor graph* (an FFG, [27]). Just like conventional factor graphs, FFGs represent graphically the factorization of a function of many variables. In contrast to them, however, the nodes represent local functions only, while the edges are interpreted as the variables. Moreover, this arrangement allows for the existence of half-edges which are connected to one factor node only. They are used to describe the observed variables in these graphical models. The following definition, according to Loeliger [27], puts this concept on a more formal footing:

- For every factor there exists a node in the graph
- There is a unique edge or half-edge for each variable
- The node representing some factor g is connected with the edge representing some variable x if and only if g is a function of x

As it is immediately obvious from the definition, a variable in an FFG can only be connected to 2 factors. This does not drastically limit the applicability of the graph, however, since the introduction of a special node, describing an equality constraint, can be used to "clone" any variable, thus eschewing a lack of generality (at the cost of introduction of an additional node, see figure 2.11 a)).

Forney factor graphs possess a number of advantages over the conventional type, many of them quite relevant to the task of describing statistical inference in neural networks:

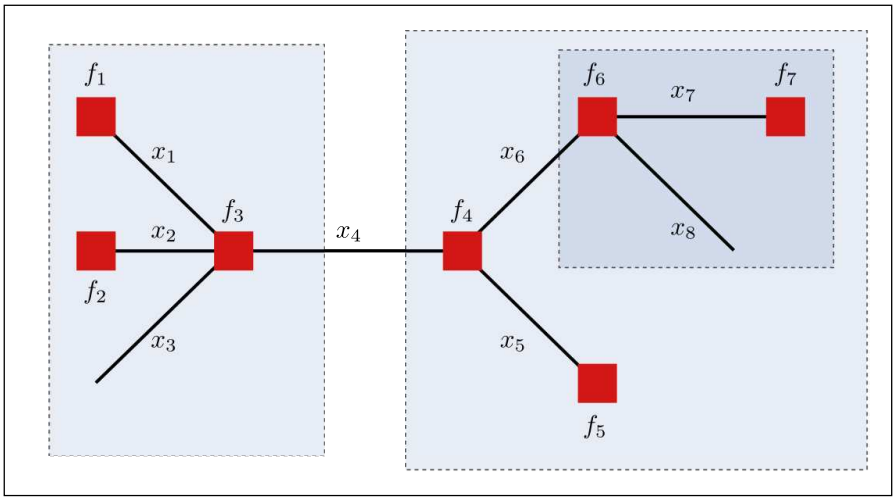


Figure 2.10: An example Forney factor graph. The shaded boxes depict sub-graphs that can be used to build recursive constructs.

Forney Factor Graph Advantages

- The sum-product algorithm assumes it simplest form in its FFG representation, as will be shown later.
- FFGs are well-suited for hierarchical modeling, so that recursive constructs ("Boxes within boxes", compare 2.10) are possible [27].
- They are directly compatible with general block diagrams [27], thus enabling a straightforward conversion of the problem from one representation to the other.

Two very useful types of factor node are the equality constraint and the zero-sum constraint. Their brief description at this point is germane to some of the models discussed later on. As mentioned previously, each edge in an FFG connects to at most two factors. In block diagrams, such "shortcomings" are dealt with by using *branching points*, see fig 2.11 a), top. In an FFG, on the other hand, such constraints are expressed in a natural manner as factor nodes. The corresponding functions are as follows:

- Equality constraint among 3 variables: $\delta(x_1 - x_3)\delta(x_2 - x_3) \Leftrightarrow x_1 = x_2 = x_2$
- Zero-sum constraint: $\delta(x_1 + x_2 + x_3) \Leftrightarrow x_1 + x_2 + x_3 = 0$

where $\delta(x - x_0)$ represents either the Kronecker symbol if the variable is discrete or the Dirac delta function otherwise. As a side remark, the zero-sum constraint may represent a sum constraint if one of the variables has a negative sign: $\delta(x_1 + x_2 - x_3) \Leftrightarrow x_1 + x_2 = x_3$, see figure 2.11, b), bottom.

The sum-product algorithm on a Forney factor graph operates much in the same way as in the conventional case if the flooding schedule (section 2.2.2) update regime is used.

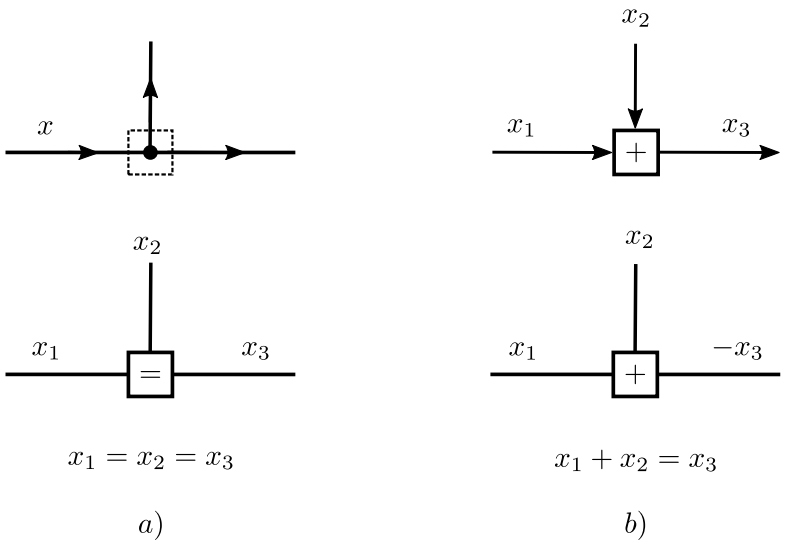


Figure 2.11: Constraint nodes: a) Branching point (top) and the corresponding equality constraint $\delta(x_1 - x_3)\delta(x_2 - x_3)$ (bottom) b) Block diagram summation (top) and the corresponding factor $\delta(x_1 + x_2 - x_3)$ (bottom)

The only difference is the update rule, which is now of a single type. The message from a node $g(x, y_1, \ldots, y_n)$ to a variable x on an FFG is calculated by multiplying the factor function g with the messages collected from all neighboring nodes along edges $y_1, \ldots, y_n$ and subsequently summarized over those variables:

$$\mu_{g \to x}(x) = \sum_{y_1} \cdots \sum_{y_n} g(x, y_1, \ldots, y_n) \mu_{y_1 \to g}(y_1) \ldots \mu_{y_n \to g}(y_n) \tag{2.18}$$

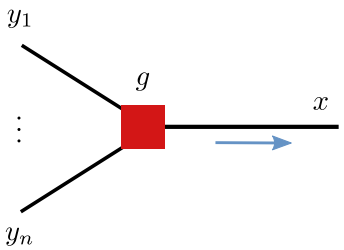


Figure 2.12: Sum-product update rule at an FFG factor node.

For demonstration purposes, the factorization of the standard example 2.13 in the previous section can be graphically represented as in figure 2.13:

2.3 Kalman Filter: A Classical Statistical Inference Model

An important application of statistical inference is a recursive minimum mean-square error method known as the *Kalman filter*[23]. It allows for the sequential estimation of

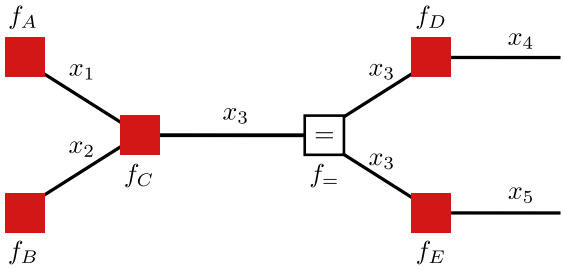


Figure 2.13: Normal graph realization of the function $f(x_1, ..., x_5)$

states of a noise-perturbed *linear dynamical system* (LDS) where the observations are also corrupted by noise and each measurement is a linear transformation of the corresponding system state. The algorithm rests on the fundamental assumption that all noise processes affecting the system and the measurements are normally distributed. Depending on the relation between time t_{k+1} at which the system state estimate is made and time t_k of the last known observation, the Kalman filter addresses 3 possible problems:

(i) **State prediction for $t_{k+1} > t_k$:** Determines the most likely future state, based on the observation sequence up to and including t_k.

(ii) **Smoothing for $t_j < t_k$:** Reduces the noise in a signal of interest.

(iii) **Filtering at t_k:** Determines the most likely current state from the knowledge of the system dynamics and the observation sequence up to and including t_k.

It should be mentioned that the algorithm is applicable to both discrete-time and continuous-time problems, but due to the computational simplicity of the former and the discrete nature of the presented model (section 2.4), this thesis concerns itself exclusively with the discrete-time Kalman filter.

The utility of the method is reflected by its widespread use in a number of technical and scientific areas such as electronic filters (FM receivers), orbit determination (Apollo program), and satellite navigation (GPS). In a neuroscientific context, a particularly interesting application of the Kalman filter is in visual object tracking by artificial neural networks. Some recent advances in brain-machine interfaces [12] suggest that it can be successfully used as the statistical inference model for the estimation of velocities by neural networks.

2.3.1 Linear Dynamical Systems and Space State Representation

The inference of system states from noisy measurement data requires an exact or approximate knowledge of the dynamics of the investigated system. If the set of generalized coordinates $\{q_1, \ldots, q_l, \dot{q}_{l+1}, \ldots, \dot{q}_n\}$ is represented by a *state vector* $\mathbf{x} \in \mathbb{R}^n$, then a linear

dynamical system (LDS) can be described by the differential equation

$$\frac{d\mathbf{x}(t)}{dt} = \mathbf{\Phi}(t)\mathbf{x}(t) \tag{2.19}$$

where $\mathbf{\Phi}$ is a linear transformation $\mathbf{\Phi} : \mathbb{R}^n \mapsto \mathbb{R}^n$ describing the dynamics of the system. In the discrete-time domain the successive state updates are governed by the difference equation $\mathbf{x}_{k+1} = \mathbf{F}_k\mathbf{x}_k$ where $\mathbf{F}_k$ is generally defined for each time step.

The state-space representation of an LDS has had a crucial role in the development of the Kalman filter [23]. Since the actual state $\mathbf{x}$ can not be determined directly or with an infinite precision, the system is modeled by a set of coupled difference (differential) equations of the observed and state variables $\mathbf{z} \in \mathbb{R}^m$ and $\mathbf{x} \in \mathbb{R}^n$ (the input or control variables are not relevant for the following discussion). The state-space model consists of a *process model* describing the evolution of the system and a *measurement model* for the observation sequence. It also incorporates error sources such as the *process noise* $\mathbf{w}$ and the *measurement noise* $\mathbf{v}$. The corresponding update equations are

$$\mathbf{x}_{k+1} = \mathbf{F}_k\mathbf{x}_k + \mathbf{w}_k \tag{2.20}$$
$$\mathbf{z}_k = \mathbf{H}_k\mathbf{x}_k + \mathbf{v}_k \tag{2.21}$$

If the state perturbation vector $\mathbf{w}$ is assumed to be normally distributed, then it can be represented by a zero-mean white-noise process $\mathbf{w}_k \sim \mathcal{N}(0, \mathbf{Q}_k)$ where its *process covariance* $\mathbf{Q}_k$ may generally depend on each time step. The measurement model for the observable $\mathbf{z}$ is represented by a linear transformation $\mathbf{H}_k : \mathbb{R}^n \mapsto \mathbb{R}^m$ of the system state $\mathbf{x}_k$ with an additional zero-mean noise $\mathbf{v}_k \sim \mathcal{N}(0, \mathbf{R}_k)$ where $\mathbf{R}_k$ is the (generally time-dependent) *measurement covariance*. The covariance matrices are given by the following equations when the processes $\mathbf{v}$ and $\mathbf{w}$ are uncorrelated:

$$E\left[\mathbf{w}_k\mathbf{w}_j^T\right] = \mathbf{Q}_k\delta_{kj} \tag{2.22}$$
$$E\left[\mathbf{v}_k\mathbf{v}_j^T\right] = \mathbf{R}_k\delta_{kj} \tag{2.23}$$
$$E\left[\mathbf{v}_k\mathbf{w}_j^T\right] = 0 \quad \forall k, j \in N \tag{2.24}$$

A block diagram of a state-space model presents visually the relations among its variables and can be used to construct a factor graph representation:

2.3.2 Discrete Kalman Filter

The Kalman filter is a prediction-correction method that recursively calculates an estimate $\hat{\mathbf{x}}_k$ for the system state $\mathbf{x}_k$ from the previous estimate $\mathbf{x}_{k-1}$ and the current observation $\mathbf{z}_k$. Since the previous estimate is based on the one preceding it, the algorithm implicitly includes the whole observation sequence up to the current time point without involving all measurements explicitly in each calculation[36]. The algorithm is

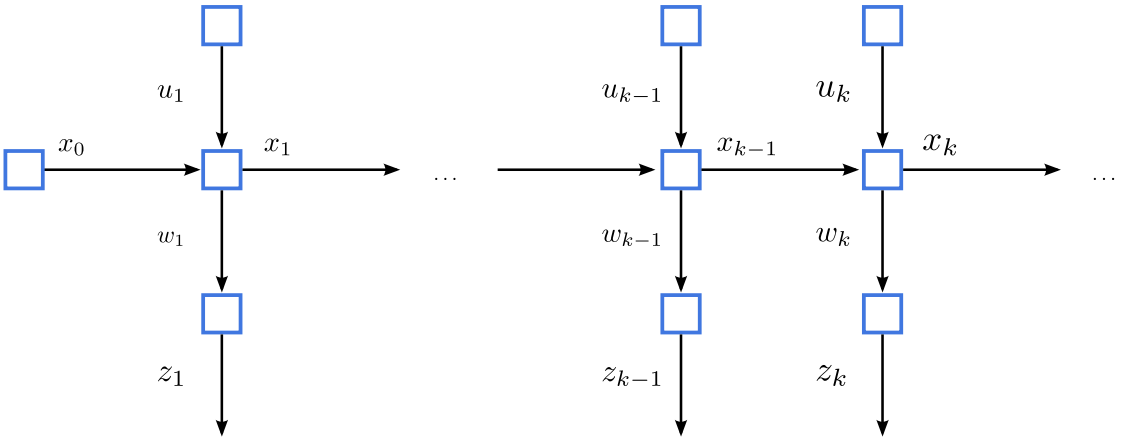


Figure 2.14: Block diagram of a linear state space model

therefore tailor-made for real-time applications involving optimal estimation problems.

It is instructive to clearly delineate the *prediction* step which produces an *a priori* estimate $\hat{\mathbf{x}}_k^{(-)}$ based only on the known dynamics of the system and the consequent *correction* step (using observation $\mathbf{z}_k$) that results in the *a posteriori* estimate $\hat{\mathbf{x}}_k^{(+)}$ for the same state $\mathbf{x}_k$. Quantities calculated in these steps will be denoted by the corresponding superscripts $^{(-)}$ and $^{(+)}$.

Prediction

The a priori estimate is calculated solely using the state transition matrix $\mathbf{F}_k$:

$$\hat{\mathbf{x}}_k^{(-)} = \mathbf{F}_k \hat{\mathbf{x}}_{k-1}^{(+)} \tag{2.25}$$

The associated estimation error of the prediction is given by the difference between the state $\mathbf{x}_k$ and the a priori estimate:

$$\mathbf{e}^{(-)} = \mathbf{x}_k - \hat{\mathbf{x}}_k^{(-)} \tag{2.26}$$

It leads to the following definition of the a priori error covariance matrix of the estimate, $\mathbf{P}_k^{(-)}$, which describes the "quality" of the prediction:

$$\mathbf{P}_k^{(-)} = E\left[\mathbf{e}_k^{(-)}\mathbf{e}_k^{(-)T}\right] = E\left[(\mathbf{x}_k - \hat{\mathbf{x}}^{(-)})(\mathbf{x}_k - \hat{\mathbf{x}}^{(-)})^T\right] \tag{2.27}$$

Correction

The a posteriori estimate $\hat{\mathbf{x}}_k^{(+)}$ is a linear combination of the a priori estimate $\hat{\mathbf{x}}_k^{(-)}$ and the *innovation* (measurement residual) $\mathbf{y}_k := \mathbf{z}_k - \mathbf{H}_k\hat{\mathbf{x}}_k^{(-)}$. The latter is weighted by the

Kalman gain matrix $\mathbf{K}_k$ which determines how much to correct the a priori estimate, depending on the "quality" of the observation:

$$\hat{\mathbf{x}}_k^{(+)} = \hat{\mathbf{x}}_k^{(-)} + \mathbf{K}_k \mathbf{y}_k = \hat{\mathbf{x}}_k^{(-)} + \mathbf{K}_k \left(\mathbf{z}_k - \mathbf{H}_k \hat{\mathbf{x}}_k^{(-)} \right) \tag{2.28}$$

The error covariance matrix for the a posterior estimation $\mathbf{P}_k^{(+)}$ is defined in the same way as the a priori covariance matrix:

$$\mathbf{e}^{(+)} = \mathbf{x}_k - \hat{\mathbf{x}}_k^{(+)} \tag{2.29}$$

$$\mathbf{P}_k^{(+)} = E\left[\mathbf{e}_k^{(+)} \mathbf{e}_k^{(+)T} \right] = E\left[(\mathbf{x}_k - \hat{\mathbf{x}}^{(+)})(\mathbf{x}_k - \hat{\mathbf{x}}^{(+)})^T \right] \tag{2.30}$$

In order to complete the recursion step, $\mathbf{P}_k^{(+)}$ has to be expressed in terms of the a priori covariance matrix $\mathbf{P}_k^{(-)}$. Although the derivation is comparatively straightforward, it goes beyond the scope of this thesis (see [5]). It suffices to mention that $\mathbf{P}_k^{(+)}$ resolves to

$$\mathbf{P}_k^{(+)} = (\mathbf{I} - \mathbf{K}_k \mathbf{H}_k) \mathbf{P}_k^{(-)} (\mathbf{I} - \mathbf{K}_k \mathbf{H}_k)^T + \mathbf{K}_k \mathbf{R}_k \mathbf{K}_k \tag{2.31}$$

for any (possibly suboptimal) $\mathbf{K}_k$. The optimization procedure entails the minimization of the estimation error variances along the diagonal of the matrix with respect to the Kalman gain:

$$\frac{d(\mathrm{tr}\mathbf{P}_k^{(+)})}{d\mathbf{K}_k} = 0 \tag{2.32}$$

This leads to the *optimal* Kalman gain that minimizes the mean of the estimation error:

$$\mathbf{K}_k = \mathbf{P}_k^{(-)} \mathbf{H}_k^T \left(\mathbf{H}_k \mathbf{P}_k^{(-)} \mathbf{H}_k^T + \mathbf{R}_k \right)^{-1} = \mathbf{P}_k^{(-)} \mathbf{H}_k^T \mathbf{S}_k^{-1} \tag{2.33}$$

where $\mathbf{S}_k := \mathbf{H}_k \mathbf{P}_k^{(-)} \mathbf{H}_k^T + \mathbf{R}_k$ is also known as the *innovation covariance*. For the optimal gain $\mathbf{K}_k$ the a posteriori covariance matrix takes on a particularly simple form:

$$\mathbf{P}_k^{(+)} = (\mathbf{I} - \mathbf{K}_k \mathbf{H}_k) \mathbf{P}_k^{(-)} \tag{2.34}$$

The calculation of $\hat{\mathbf{x}}_k^{(+)}$ and $\mathbf{P}_k^{(+)}$ completes the recursion step in the Kalman filter, since they are used to compute the next a priori estimate and its covariance matrix.

Kalman Filter Example

As an illustration of the action of the Kalman filter, the following plot represents a tracking problem in the two-dimensional plane. A particle with coordinates $\mathbf{x}_k = (x_k, y_k)^T$ and a constant velocity $\dot{\mathbf{x}}_k$ experiences random acceleration $\mathbf{a}_k$ at each t_k. The observations $\mathbf{z}_k$ are corrupted by noise $\mathbf{v}_k \sim \mathcal{N}(0, \mathbf{R})$ while the process noise $\mathbf{w}_k \sim \mathcal{N}(0, \mathbf{Q})$ is derived from the acceleration distribution $\mathbf{a}$.

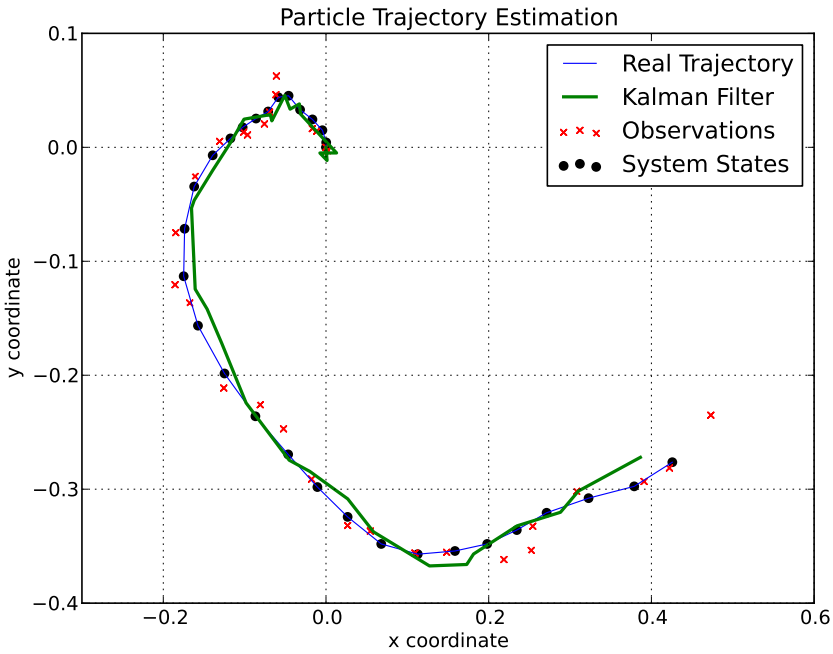


Figure 2.15: Kalman filter example. A point particle experiences a random acceleration at each time step. Its real position is given by the blue trajectory with system states determined by its (x,y) coordinates. The noisy measurements are represented by the red markers, while the prediction of the Kalman filter is shown as a green trace.

2.3.3 Bayesian Representation of the Kalman Filter

As with many other statistical inference methods, the Kalman filter can be cast into a probabilistic graphical model in a natural way. The following graphical representation of a state space system generalizes both the LDS and another important structure known as the *Hidden Markov Model* (HMM). In contrast to the LDS, the state variables in an HMM are discrete.

In the Bayesian framework the state variables $\mathbf{x}_k$ are known as *hidden* or *latent* variables, whereas the measurements $\mathbf{z}_k$ are the *observable* variables. The latent variables form a Markov chain such that each $\mathbf{x}_k$ is conditionally independent of all but the preceding variable $\mathbf{x}_{k-1}$. Under this assumption the *transitional probabilities* then take the simple form:

$$p(\mathbf{x}_k|\mathbf{x}_{k-1}, \ldots, \mathbf{x}_1) = p(\mathbf{x}_k|\mathbf{x}_{k-1}) \tag{2.35}$$

The observables $\mathbf{z}_k$ are produced from the state variables by *emission probabilities*. They are likewise conditionally independent from all states $\mathbf{x}_k$ with $k \leq k-1$:

$$p(\mathbf{z}_k|\mathbf{x}_k, \mathbf{x}_{k-1} \ldots, \mathbf{x}_1) = p(\mathbf{z}_k|\mathbf{x}_k) \tag{2.36}$$

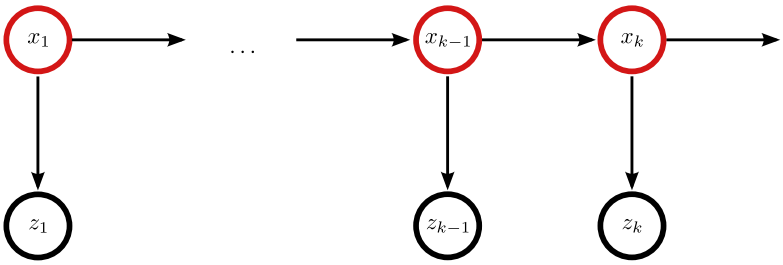


Figure 2.16: Bayesian graphical model of a state space system

This results in a joint probability distribution

$$p(\mathbf{x}_1, \ldots, \mathbf{x}_k, \mathbf{z}_1, \ldots, \mathbf{z}_k) = p(\mathbf{z}_1) \prod_{i=2}^{k} p(\mathbf{z}_i|\mathbf{z}_{i-1}) \prod_{j=1}^{k} p(\mathbf{x}_j|\mathbf{x}_{j-1}) \tag{2.37}$$

It should be noted that this model automatically absorbs the process and measurement noise into the transition and emission distributions accordingly and thus allows a simpler representation of the problem.

2.3.4 Kalman Filter Factor Graph

The calculation of a particular distribution $p(\mathbf{x}_k|\mathbf{z}_k)$ of probable states given a measurement requires the marginalization of the joint probability 2.37. The prediction step of the Kalman filter can then be expressed by the marginalization

$$p(\mathbf{x}_k|\mathbf{z}_{k-1}) = \int p(\mathbf{x}_k|\mathbf{x}_{k-1}) p(\mathbf{x}_{k-1}|\mathbf{z}_{k-1}) d\mathbf{x}_{k-1} \tag{2.38}$$

if the quantities up to t_{k-1} are already known. The prediction then follows by Bayes's theorem 2.1:

$$p(\mathbf{x}_k|\mathbf{z}_k) = \frac{p(\mathbf{z}_k|\mathbf{x}_k) p(\mathbf{x}_k|\mathbf{x}_{k-1})}{p(\mathbf{z}_k|\mathbf{z}_{k-1})} = \alpha p(\mathbf{z}_k|\mathbf{x}_k) p(\mathbf{x}_k|\mathbf{x}_{k-1}) \tag{2.39}$$

The calculation of the normalization factor α can be done at each node when the sum-product algorithm is used for the computation of the marginalization.

The factor graph representation of the space state system can be used to calculate all marginal probabilities simultaneously by the sum-product algorithm. A simple HMM model that is well-suited for neural network implementation is presented in the next section.

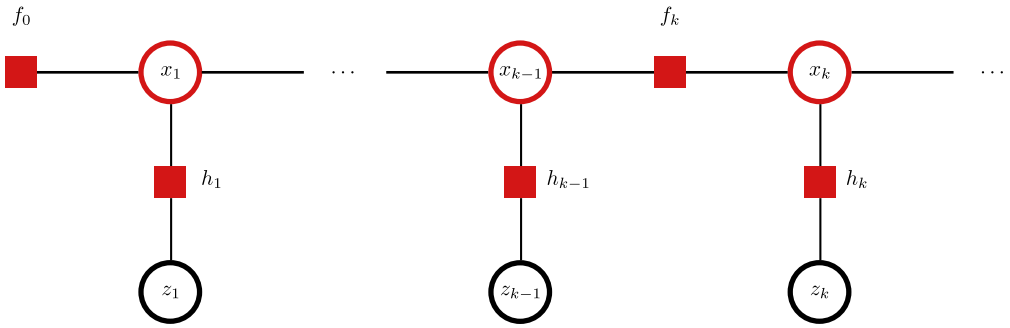


Figure 2.17: Factor graph representation of the Kalman filter

2.4 A Simplified Kalman Filter-Like Model for Inference on a Chain

The Kalman filter is a statistical inference model that is particularly well-suited for an implementation on neuromorphic hardware. Due to the reasons explained in [37], messages that are produced by factor nodes that are not in direct contact with external input sources tend to be less correlated with the theoretically expected messages, since they represent the result of secondarily processed inputs (see also figure 3.19). Moreover, the limited number of neurons and synaptic connections available on the device pose certain limits on the complexity of the model. The nodes in a Kalman filter, on the other hand, avoid these problems since they receive input directly from the observables and can be represented by relatively simple factor functions when the variables used in the graph are binary.

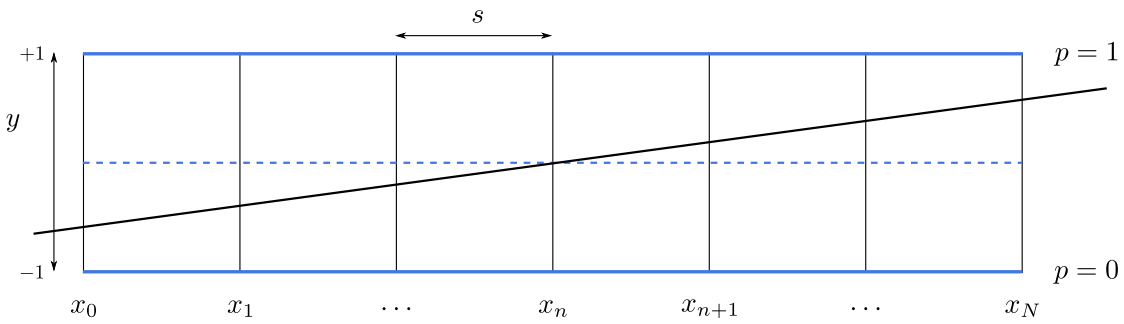


Figure 2.18: Schematic depiction of the simplified Kalman filter model

Figure 2.18 represents a schematic depiction of the model. A particle enters a detector array with N sensors that are equidistantly located along its length $L = Ns$, where s is the distance between them. Each sensor has a limited spatial resolution and can only determine whether the passage of the particle was on the left or on the right side of the line at $y = 0$ shown in the figure, leading to a state variable z_n with 2 possible values where $p = 0$ if the particle can be found in the region $y \in [0, -1]$ and $p = 1$ if it is in $y \in [0, 1]$.

Since any physical measurement has an associated uncertainty, it is given by a conditional distribution $p(z_n|x_n)$ where x_n is the observable. The problem, then, is to determine the transitional probabilities that are needed to construct a factor graph of the model.

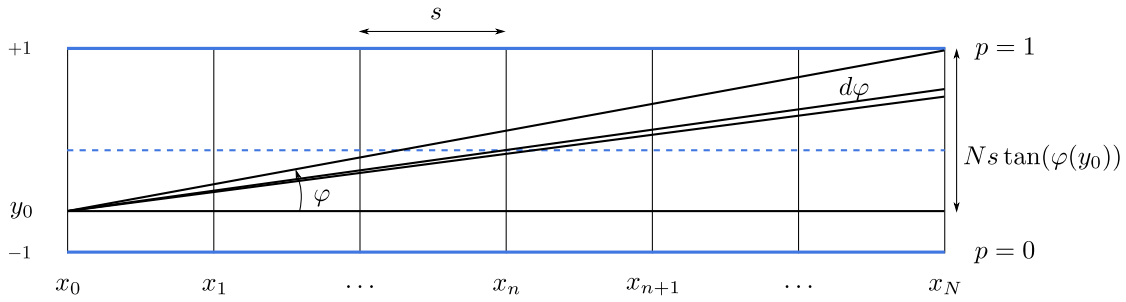


Figure 2.19: Transitional probability calculation sketch for the simplified Kalman filter

The calculations necessary to find these are only sketched schematically in this publication, since the model has not yet been utilized for statistical inference purposes yet.

A single particle that enters the detector array at the y_0 coordinate shown in figure 2.19 can only exit it for the range spanned by the angle

$$\varphi(y_0) \in \left[-\arctan\left(\frac{1+x_o}{Ns}\right), \arctan\left(\frac{1-x_o}{Ns}\right)\right] \tag{2.40}$$

which generally depends on the entry point y_0. The infinitesimal probability for the particle being in this range is given by

$$p(\varphi(x_0)) = \frac{d\varphi}{\arctan\left(\frac{1+x_0}{Ns}\right) + \arctan\left(\frac{1-x_0}{Ns}\right)} \tag{2.41}$$

with $p(x_0) = \frac{1}{2}dx_0$. The transitional probability between two identical states (as an example) $P_{0\to 0}$ depends on the range of possible locations $y_0 \in [y_{min}, y_{max}]$ that the particle may have entered the detector from, which can be determined from the condition that it has to remain in $-1 \leq x_n \leq 0$ ($y_0 \in [A, B]$) and $-1 \leq x_{n+1} \leq 0$ ($\varphi \in [C(y_0), D(y_0)]$). The transitional probability is then given by

$$P_{0\to 0} = \frac{1}{Z}\int_A^B dy_0 \int_{C(y_0)}^{D(y_0)} d\varphi p(y_0)p(\varphi(y_0)) \tag{2.42}$$

with a normalization factor

$$Z = \int_{-1}^{1} dy_0 \int_{-\arctan\left(\frac{1+y_0}{Ns}\right)}^{\arctan\left(\frac{1-y_0}{Ns}\right)} d\varphi p(y_0)p(\varphi(y_0)) \tag{2.43}$$

The Forney factor graph representation of the model is depicted in figure 2.20.

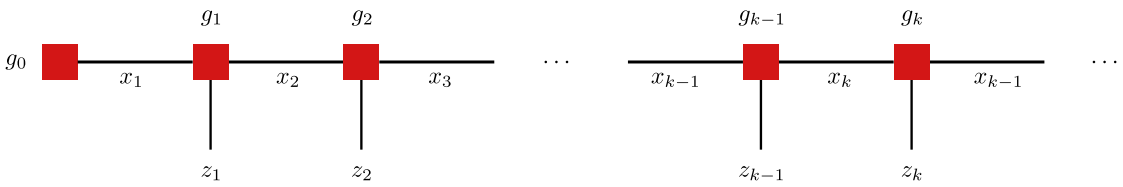


Figure 2.20: Forney factor graph representation of the simplified Kalman filter

2.5 Belief Propagation in Artificial Neural Networks

If a particular statistical inference model is to be embedded in an artificial neural network, the central problem consists in the arrangement of the probabilistic computations necessary for the execution of the sum-product algorithm. Since Bayesian inference provides a good foundation for neuroscientific computing [16], there have been studies aiming at explaining how it is performed by biological neurons [29]. Such studies are concerned mostly with single neuron computation and even with the biophysical foundations of Bayesian inference *in vivo* [15], but their results are not general enough or depend on rough approximations and are therefore unsuitable for the task at hand[37].

An alternative approach is to devise a network architecture that implements the inference algorithm and thus show the feasibility of the postulated probabilistic computations by construction. This method has been used successfully by *Steimer et al.* [37] to demonstrate belief propagation in a spiking neural network. Key to the experimental setup are the encoding of the messages and the implementation of the factor nodes.

2.5.1 Network Architecture

The first step toward the realization of the sum-product algorithm in an artificial neural network is to choose a suitable factor graph representation and consequently to identify all of its nodes and edges with elements of the network.

The Forney factor graph (section 2.2.3) may be considered the most natural class of graphical models that can be implemented as an ANN, since there is a direct mapping of factor nodes to populations of neurons and another one from factor graph variables to the projections between these populations in the network. Observable variables that usually deliver sensory input to biological neural networks can be identified with the half-edges of an FFG and can therefore be implemented by inhomogeneous Poisson spike train generators in their artificial counterparts.

In order to perform belief propagation, each population has to represent a particular mathematical factor function and to marginalize its product with incoming messages over a set of variables. This requires a computational model that approximates the theoretical calculations with sufficient accuracy. Such a model is the *liquid state machine* (LSM) [31]. Under particular conditions (section 2.5.4) it represents a filter with a fading memory that can perform a broad class of analog calculations on incoming streams of data.

The ANN realization of an LSM comprises a pool of recurrently connected neurons and a set of *readout* populations of neurons that are (linearly) connected to it but not among each other. When input spike trains representing messages from other factor nodes excite time-varying high-dimensional states in the *liquid pool*, the resulting dynamics can be viewed as a non-linear transformation of the inputs. Each readout captures a particular calculation result from the liquid as a linear combination of states via its connection with the recurrent network. Weight adjustment of the corresponding synapses forces the readout to extract a different informational content from the liquid and thus leads to the possibility of training it to perform a classification task on the liquid states. This method can be used to implement the mathematical function at each FFG node and the associated message update rule by training a readout assigned along an edge (a variable) connecting the LSM to another factor node.

As already described (see figure 2.9), the flooding schedule allows for the simultaneous calculation of all marginal probability distributions on a factor graph representing a complex factorized joint probability density function. It requires that information flows in both directions along each edge (but unidirectionally on half-edges) where each message is the marginalized product of the factor function with the messages coming along all other edges at the factor node. Since readout populations are tasked with emitting these messages, it follows that in an ANN implementation of an FFG there have to be at least two readouts along an edge connecting two factors. This completes the blueprint for a neural network implementation of a Forney factor graph. Figure 2.21 depicts the stepwise transformation of a bipartite factor graph representing a Kalman filter to a neural network model of the same statistical inference method.

2.5.2 Neural Encoding of Messages

After establishing the overall graphical structure of the ANN implementation of an FFG, it is time to consider the encoding of the probability messages in the network. There are are no constraints on the type of variables or factor functions used in an FFG. In a statistical context, the probability distributions of these variables may therefore be discrete or continuous, or they can even represent deterministic variables.

In practice, binary random variables afford the easiest approach to the encoding problem, since the time-series of the corresponding random process encodes the probability with which such a variable takes the value 1 at each time point. Due to normalization, the probability of it taking value 0 is then automatically derived from the encoded message. All models described in this thesis are based on Bernoulli random variables due to the simplicity of the encoding problem. Theoretically it is possible to assign K readouts, each for a particular realization of a K-valued discrete random variable, but such an approach suffers from obvious computational scaling problems and can not be trivially generalized for continuous distributions.

Choosing binary probability messages leads to a simple encoding scheme that is based on the population activity rate of the readout neurons. As it is described in the next

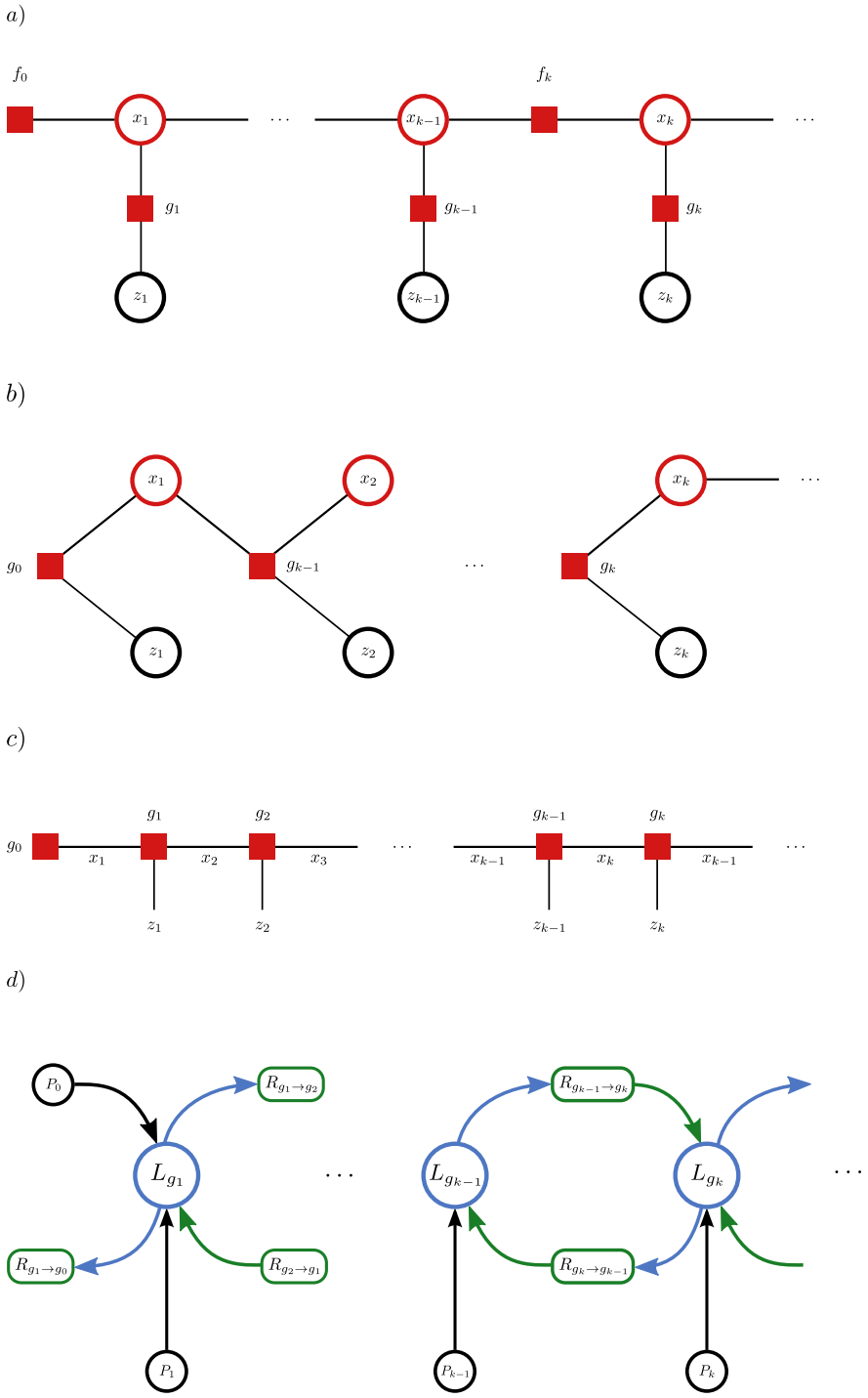


Figure 2.21: From a bipartite factor graph to a neural network model: *a*) Kalman filter factor graph *b*) Kalman filter factor graph that can be easily converted to an FFG *c*) FFG representation of the Kalman filter *d*) Neural network representation of the FFG Kalman filter

section, the discrete message update rule of the sum-product algorithm can be cast into a continuous-time differential equation whose solution results in a function of time representing the message emitted along some edge of the network. Such a solution describes the time-varying rate of an inhomogeneous Poisson process that underlies the firing of the readout neurons.

The calculation of the instantaneous firing rate is a non-trivial neuroscientific problem that is described for instance in Nawort et al [28]. A common way to determine it includes the convolution of a spike train represented by Dirac delta functions with a suitably chosen kernel, thus resulting in a continuous rate function of time:

$$r(t) = \sum_{t_k} \int_0^T \delta(t' - t_k) K(t - t') dt' = \sum_{t_k} (t - t_k), \quad t_k < T \tag{2.44}$$

The shape of $K(t)$ can be chosen in terms of computational efficiency, depending on the application of interest (For instance a triangle-shaped kernel may be as good as a Gaussian one [28]). The work in this thesis uses a Gaussian filter for determining instantaneous firing rates.

2.5.3 Sum-Product Calculation

So far the sum-product message update rule on an FFG at a factor node has been described only in the discrete-time domain, see equation 2.18. In a neural network, on the other hand, the dynamics are continuous in time and there is no global clock signal to steer the computation of messages. If the differential version of the flooding schedule is regarded as the extreme case of an infinitesimal update dm in time step dt, however, it is possible to construct a differential equation that represents the algorithm in continuous time. Moreover, it belongs to the class of time-invariant filters with fading memory [37], and it can therefore be approximated by an LSM [31].

If message $\mu_{i \to j}(x_j)$ is to be passed from node i to node j over the edge x_j in an FFG, the differential equation that enables its calculation takes on the following form:

$$\tau \dot{\mu}_{i \to j}(t) + \mu_{i \to j}(t) = \frac{1}{Z(t)} \sum_{X \in \{0,1\}^m} f_i(x_j = 1, X) \prod_{k \in \mathcal{N}/j} p_{k \to j}(x_k, t - D) \tag{2.45}$$

$$p_{k \to i}(x_k, t) = \begin{cases} \mu_{k \to i}(t), & \text{if } x_k = 1 \\ 1 - \mu_{k \to i}(t), & \text{otherwise} \end{cases} \tag{2.46}$$

The quantities that take part in the equation are defined as follows:

- $\mathcal{N}(i)/j$ - Set of all neighbors of node i except for node j
- n - Cardinality of $\mathcal{N}(i)/j$

- $X = (x_k)_{k \in \mathcal{N}(i)/j}$ - Vector of binary variables x_k each linking node k to node i
- f_i - Factor function associated with node i
- D - A fixed synaptic delay
- τ - A time constant that determines the computational dynamics of the system
- $Z(t)$ - A normalization term ensuring $0 \leq \mu_{i \to j}(t) \leq 1$

This equation can be used to calculate the time series of a message along a full edge (half edges are the observables) from the incoming messages on all other edges at a factor node. If the input messages are generated and then fed into a factor node, then the knowledge of the calculated message can be use to train the LSM to approximate the same computation as in 2.45. How this is possible is subject of the next section.

2.5.4 Liquid State Machine Paradigm

As it has been discussed in section 2.5.1, the central question of approximating the sum-product update rule by a population of neurons can be addressed by training an ANN representing a liquid state machine. According to the underlying mathematical theory of an LSM, it can perform universal computations with fading memory on analog functions in continuous time [31]. Central to that theory is the notion of a high-dimensional *liquid* medium that is disturbed by the action of input sequences (e.g. complex spatio-temporal sensory data). The perturbed state of such a liquid is defined at any point in time and generally reflects the dynamics of the medium. Since it is transient, it stands in stark contrast to a Turing machine where the states and their transition functions are stable, well-defined, and task-specific. The state of an LSM, on the other hand, can be mathematically described as a time-parametrized point trajectory in a high-dimensional Euclidean space.

As the dynamics of the liquid is affected by past and present inputs, it has to produce sufficiently different states when faced with disparate data input streams. This ability is an intrinsic characteristic of the system and can be formalized by the *separation property* which describes the divergence of two state trajectories caused by different input streams.

While the liquid can generally be regarded as performing a complex nonlinear transformation on its inputs, the question of how this can be used to calculate a specific output function $y(t)$ from the inputs $u(.)$ remains open. To this end, an additional component of the LSM framework has to be introduced, namely the *readout*. It maps a liquid state $x^M(t)$ to the target function $y(t)$ and thus acts as a memoryless *filter*. An interesting feature of the LSM model is that a given liquid can have a number of readouts that produce different functions $y_i(t)$ from the same input $u(.)$ in real time. The characterization of the ability of an readout to produce the desired output function is done using the *approximation property* of the LSM. In general the separation property depends on the

complexity of the liquid medium and its relaxation time constants, while the approximation property is defined mostly by the readout mechanism used to produce the output function [31].

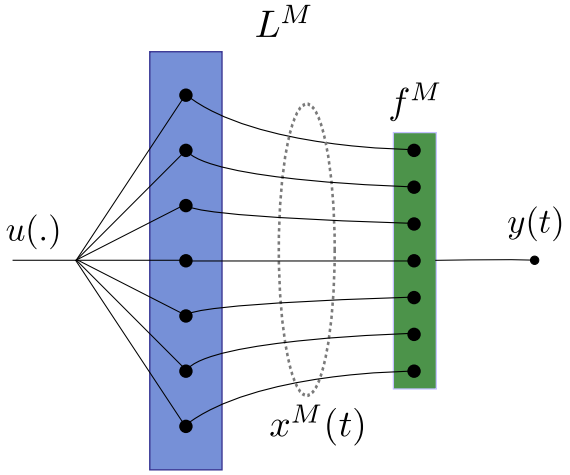


Figure 2.22: Liquid State Machine

A more rigorous definition of these ideas can be achieved by starting with a vector of input functions $\mathbf{u}(t) \in \mathcal{U}^n$ that is transformed into an k-dimensional liquid state $\mathbf{x}^k(t)$ by the action of the *liquid filter* L^k

$$\mathbf{x}^k(t) = (L^k\mathbf{u})(t) \tag{2.47}$$

where L^k is map (an operator) $\mathcal{U}^n \mapsto (\mathbb{R}^{\mathbb{R}})^k$ from the n-dimensional vector space of real-valued functions in $\mathcal{U} \subset \mathbb{R}^{\mathbb{R}}$ to a k-dimensional vector space of all real-valued functions $(\mathbb{R}^{\mathbb{R}})^k$ and represents the dynamics of the system.

Furthermore, a memoryless readout map $f^k : (\mathbb{R}^{\mathbb{R}})^k \mapsto \mathbb{R}$ transforms the state $\mathbf{x}^k(t)$ to an output function $y(t)$ and has to be chosen in a application-specific manner.

As already mentioned in section 2.5.1, the definition of the readout filter f^k in an ANN representation of an LSM can be identified with a collection of synaptic weights $\{w_1, \ldots, w_N\}$ for the projection connecting the recurrent liquid pool population with a readout population of neurons that are not connected among themselves. The determination of these weights can be achieved by training the readout as a classifier for liquid states.

2.5.5 Readout Training

The approximation of a factor function $f(t)$ with a neural network-based LSM can be achieved by a *supervised learning* procedure. This method relies on the ability of the network to generalize its behavior for unknown input functions when it is trained to

respond correctly to a set of of input training messages $\{\mathbf{u}_1(t), \ldots, \mathbf{u}_N(t)\}$.

Since the time-varying values of an input function $u(t)$ are mapped to the instantaneous firing rate $r(t)$ of a population of neurons (section 2.5.2), the spike trains corresponding to such a message can be generated by a collection of inhomogenous Poisson processes, one for each neuron, whose rates during each step dt correspond to the value of $u(t)$ in that infinitesimal interval. Similarly, the instantaneous population rate $o(t)$ of the readout corresponds to the target function $f(t)$.

In a current-based synapse, each input spike $\delta(t - t_k)$ at time t_k injects a current $\int \delta(t'-t_k)K(t-t')dt'$ into the post-synaptic neuron. Typically, the kernel $K(t)$ represents an exponential decay

$$PSC(t) = w \ \exp\left(-\frac{t-D}{\tau_{\text{syn}}}\right) \tag{2.48}$$

where w is the synaptic weight, D the synaptic delay, and τ_{syn} the synaptic time constant. The total current $I'(t)$ injected into a readout neuron by all pre-synaptic liquid neurons (assuming an all-to-all connection between liquid and readout) in response to a training input $\mathbf{u}(t)$ can therefore be calculated simply as the weighted sum of current courses $L_i(t)$, each gained by the convolution of an individual spike train $S_i(t)$ with the kernel $K(t)$:

$$I'(t) = \sum_{i=1}^{N} w_i L_i(t) = \sum_{i=1}^{N} w_i \int_0^T S_i(t')K(t-t')dt', \quad t \in [0,T] \tag{2.49}$$

If the time-series of the target function $f(t)$ has been computed for $\mathbf{u}(t)$, then the corresponding population rate $o(t)$ of the readout would be produced by some current $I(t)$ that can not be calculated directly due to the spiking dynamics of the readout neurons. It is possible, however, to experimentally determine a monotonic function $R(I)$ by measuring the instantaneous firing rates of the readout for different values of an injected total mean current (with noise). The inverse function can be used to calculate the current $I(t)$ that is used as a target for training the readout.

A linear readout generally represents a linear combination of states $f(t) = \mathbf{w}\mathbf{x}(t)$ that obviously translates to $I(t) = \mathbf{w}\mathbf{L}(t)$ for the current-based synapse model discussed above. The synaptic weights $\mathbf{w}$ can be computed by linear regression to minimize the mean squared error $(\mathbf{I}(t) - \mathbf{w}\mathbf{L}(t))^2$ [30] for a theoretically calculated target current $\mathbf{I}(t)$. The least-squares problem can be formally solved by the following equation:

$$\mathbf{w} = (\mathbf{L}\mathbf{L}^T)^{-1}\mathbf{L}^T\mathbf{I} \tag{2.50}$$

The performance of the readout can be evaluated by examining the cross-correlation between the target message $m(t)$ (encoded as a rate) and the measured instantaneous output rate $o(t)$ for novel input messages.

The ability of the LSM to approximate a factor function $f(t)$ depends on a number of parameters that govern its separation and approximation properties. In general they have to be optimized for each such function and therefore are factor-specific. As a guiding principle, a "default" set of parameters can be derived from biological data, since a liquid pool may represent the computations occuring in a cortical mini-column [30].

2.5.6 Factor Graphs

The possibility for training a liquid state machine implemented by an artificial neural network that has to approximate a particular factor function is the fundamental prerequisite for embedding the sum-product algorithm in an ANN. Since belief propagation is only the means by which the marginalization of a complex probability distribution is performed, a suitable model for statistical inference is required in order to demonstrate the computational abilities of such networks. A publication by Steimer et al. [37] establishes the feasibility of the presented approach, based on two simple models that use binary random variables.

Binary Channel

The first can be regarded as a proof-of-concept for the theoretical considerations discussed so far and represents a channel for the transmission of 4-bit binary data. As with any physical medium, such a channel does not propagate the information with absolute reliability and bits may flop with a certain probability on their way to the receiver. The latter is interested in estimating the probability for such an event occuring, based on a priori assumptions and the received data.

Explaining Away Phenomenon

A more advanced model[24] that has a biological significance describes a visual illusion that involves the perception of the surface curvature and luminance of two attached objects. It is known as an *explaining away* phenomenon and has been used as a standard example in other publications.

The details of the software implementation of these models is subject to the next chapter.

3 Belief Propagation in Artificial Neural Networks

The ultimate aim of achieving belief propagation on a neuromorphic device can only be realized if a number of prerequisite steps are fulfilled. The most important advance toward this goal consists in validating the approach described in section 2.5 by means of a software simulation of an artificial neural network performing statistical inference. The results obtained in this fashion do not only establish the method, but also provide important guidance for the adaptation of the investigated network models to the unique characteristics of neromorphic devices. This chapter is dedicated to the details of the implementation and simulation of three statistical inference models briefly described in section 2.5.6. In that, it follows closely the pioneering work of Steimer et al. [37] which demonstrates belief propagation in networks of spiking neurons.

3.1 Simulation Methods

The simulation of large artificial neural networks (on the order of thousands of neurons and more) generally relies on a trade-off between the computational efficiency of the chosen neuron model and the range of the neuroscientific phenomena that it can emulate. Consequently, *leaky integrate-and-fire* (LIF) neuron models [33], [18] of varying complexity are particularly popular with modelers due to their simplicity and biological plausibility. In such models the cellular membrane is represented by a capacitance C_m and a leakage conductance g_l that combines the action of various types of ionic channels, thus allowing for the existence of a *leakage current* across the membrane when its potential V_m differs from its resting potential E_l. The membrane potential dynamics of the most basic LIF neuron are given by the following equation:

$$C_m \frac{dV_\mathrm{m}}{dt} = -g_\mathrm{l}(V_\mathrm{m} - E_\mathrm{l}) + I(t) \tag{3.1}$$

where the total current $I(t)$ is injected by all synapses connected to the neuron and therefore depends on the temporal distribution of incoming spike trains. Since this is a point neuron model, it leaves out the issue of passive (and possibly active) propagation of the signal across dendrites to the membrane region of the *soma* where the spike generation actually takes place.

Furthermore, there are two types of *synaptic model*: current based and conductance based. The former has already been presented in section 2.5.5 (see equation 2.49) and describes the generation a predefined current time course in response to an incoming *spike* (a delta function in time, $\delta(t - t_o)$). The latter is closer to biological reality and

emulates the conductance change at the postsynaptic membrane triggered by the release of neurotransmitters by the presynaptic terminal. This *conductance course* is typically described by an exponential decay or an *alpha function* and the corresponding current depends on the difference between the membrane potential V_m and a *reversal potential* E_syn that is synapse-specific and can be used to differentiate between excitatory and inhibitory synapses. The uncertainty associated with the quantal release of transmitter-filled vesicles is sometimes represented by a multiplicative probability factor.

The LIF model is completed by the specification of a *threshold potential* V_thresh that is necessary for the generation of action potentials (spikes in the LIF model) in the vicinity of the axonal hillock. When $V_\mathrm{m}(t)$ reaches that threshold, it immediately drops to a *reset potential* V_reset and then reaches back to the resting potential E_l in the absence of input stimuli. In addition, there is an *absolute refractory period* τ_r immediately after the generation of an action potential, during which the membrane potential remains clamped to V_reset.

In contrast to more fundamental biophysical studies involving the modeling of ion channels or individual neurons with multiple compartments, the simulation of large artificial neural networks usually involves stereotypical calculations of the type described above, so that the problem consists mainly in managing signal propagation across the system. In order to avoid the repetitious task of setting up the same type of differential equations over and over again for each particular network configuration, researchers have developed several software packages that automate the process to the point where end users only have to specify the connectivity of the network and the parametrization of its neuron and synapse models. Such *spiking neural network simulators* then execute the low-level calculations underlying the simulation and usually provide access to its results in the form of spike, voltage, and conductance trace recordings. In contrast to neuromorphic devices that are usually more rigid in terms of emulable neuron models and network structures, there are but few constraints on the type of neural networks that can be investigated via software simulators *in silico*, therefore the latter are well-suited for preliminary studies of network models that are to be ported to the hardware.

Two such artificial neural network simulators that have been used extensively in preparatory studies and for the results obtained for this chapter are NEURON [9] and NEST [19]. The NEURON project focuses more on the biophysical details of the simulations and provides more complex neuron models (including multi-compartment neurons), while NEST is geared more toward computational efficiency and enables the simulation of large neural networks on HPC clusters.

One drawback of using such software simulators or neuromorphic hardware is that the corresponding programming interfaces are simulator-specific and usually involve the utilization of a custom programming language. Such lack of standardization may often stymie attempts to compare results across simulators such as NEURON, NEST, and Spikey v4. A major effort toward resolving this problem has produced the PyNN [1] modeling interface that not only handles all software and (FACETS) hardware backends

in a unified way, but also provides many high-level data structures and algorithms that simplify the definition of complex neural networks. It is based on the Python [38] programming language and has been used for the implementation of all neural network models presented in this chapter.

Many of the calculations involved in the preparation and analysis of data derived from the presented statistical inference models, such as spike train convolution and various signal generation functions, are based on routines defined in the *NeuroTools* [6] package. Its goal is to provide well-established neuroscientific methods for dealing with experimental results and to simplify some aspects of simulation development.

3.2 Factor Training

The performance of the sum-product algorithm, when implemented by a network of spiking neurons, relies on the proper training of individual factors to perform the continuous-time update rule as defined in equation 2.45. According to the method described in section 2.5.2, each factor node f_i receives time-varying messages $\{\mu_{k\to i}(t), k \in \mathcal{N}(i)/j\}$ which represent the instantaneous probabilities of the associated edges x_k to take on the value 1 when the random variables are Bernoulli-distributed (see figure 3.1). In an artificial neural network this probability is represented by the (linearly scaled) population firing rate $R(t)$ of readouts and spike sources that project into a liquid pool of neurons (section 2.5.1). A readout $R_{i\to j}$ that belongs to the LSM representing f_i and projects to another factor node f_j along edge x_j has to produce a population rate $R(t)$ that corresponds to the correct message $\mu_{i\to j}(t)$ in response to its inputs $\mu_{k\to i}(t)$. Since the readout can not be trained for all possible combinations of values $\mu^N, \mu \in [0,1] \subset \mathbb{R}$ of its N inputs at each instant dt, it is necessary to use supervised learning that is based on randomly generated training messages and let the readout generalize the update rule for novel input.

The training method presented in section 2.5.5 uses multivariate linear regression, but in order to arrive at the correct readout post-synaptic current $I(t)$ used in the procedure, several experimental steps have to be completed. These are outlined in figure 3.1 and their detailed description is subject of the following sections. Each factor is trained separately, since its input while learning is necessarily different from that in an actual statistical inference simulation (partly or wholly supplied by other factors in that case).

Firstly, a set of training messages must be generated. Since they have to be normalized and need to cover a wide range of values, an adequate procedure to this end is provided by a random walk process with reflecting bounds. Thereafter, using equation 2.45, the time series of the target output message $\mu_{i\to j}(t)$ (for the corresponding factor function f_i) can be computed numerically from the data. Furthermore it needs to be linearly scaled to a target readout rate $R_{i\to j}(t)$ that describes the population firing rate of the readout. Finally, the target current $I(t)$ can be obtained from $R_{i\to j}(t)$ from an empirically

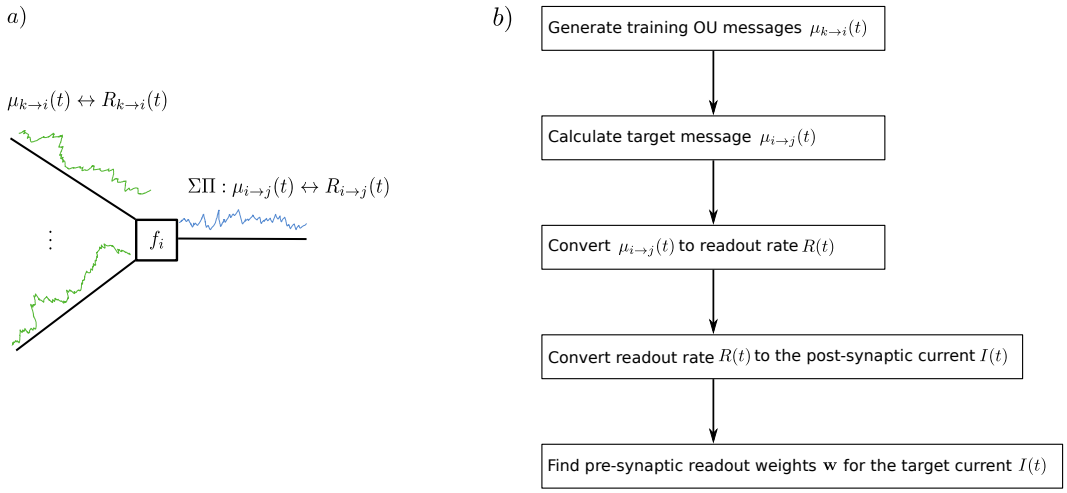


Figure 3.1: An overview of the training procedure for individual readouts

derived relation between the total mean current injected into a readout and its firing rate.

When the synapse model used in the simulation is current-based, the convolution of the spike train output from the liquid pool with the current time course kernel, which is parametrized by a default weight w and a synaptic time constant τ_{syn}, leads to the matrix $\mathbf{L}(t)$ described in section 2.5.5. The weights $\mathbf{w}$ that would produce the target current $I(t)$ are then obtained by the application of equation 2.50.

3.2.1 Generation of Training Messages

Since the supervised learning method relies on generalization from training samples, particular care must be taken in considering how these messages are generated. Ideally, the time series of a training message should be a superposition of all possible frequencies in a way that they are distributed uniformly, which are properties of the zero-mean Gaussian white noise process $w(t)$. It can also be defined as the derivative of the time-continuous *Wiener* process

$$\frac{dW(t)}{dt} = \sigma w(t) \tag{3.2}$$

that is used to describe Brownian motion and thus a plausible candidate for the generation of random inputs for training by the simple rule

$$w_{n+1} = w_n + \sigma\sqrt{h}z, \quad z \sim \mathcal{N}(0,1) \tag{3.3}$$

where h is the integration step. Unfortunately, since the individual realizations of the Gaussian distribution are independent, the autocorrelation of a white noise process is $< w(t)w(t') >= \delta(t-t')$ so that its intensity changes sharply in time. Clearly, this is an implausible assumption, since candidate random processes with this property would not

lead to the desired population firing rate due to the dynamics of the integrate-and-fire neuron.

In fact, most white noise processes encountered in reality are filtered by measurement devices and the physical processes that generate them produce a "white" band in a limited range of the frequency spectrum [13]. In an artificial neural network the membranes of the neurons have a low-pass filtering effect on the currents entering these cells (see equation 3.1), consequently cutting off higher frequencies. This means that an *Ornstein-Uhlenbeck* (OU) process with reflecting bounds is especially well-suited for the generation of training messages since it can be regarded as a low-pass filtered white noise. In addition it can be synchronized with the relaxation times of the system by the adjustment of its time constant.

The OU process represents the solution of the Langevin equation [13]

$$dU = \left(-\frac{U}{\tau} + \mu\right) dt + \sigma dW \tag{3.4}$$

that consists of a deterministic and a stochastic element. The parameter $\tau \geq 0$ is the time constant of the process, $\sigma \geq 0$ its diffusion coefficient, while μ is the drift coefficient and also a (long-term) mean $< U(t) >$. The stochastic part is represented by the continuous-time random walk W, which is a Wiener process.

The solution of equation 3.4 for discrete times $t_n = nh$ leads to an iterative form that can be used directly to generate the necessary data:

$$U_{t_n} = U_{t_{n-1}} e^{-\frac{h}{\tau}} + \mu(1 - e^{-\frac{h}{\tau}}) + Z_n \tag{3.5}$$

The stochastic part $Z_n = \sigma e^{-\frac{t_n}{\tau}} \int_{t_{n-1}}^{t_n} e^{-\frac{s}{\tau}} dW_s$ is a sequence of i.i.d normal random variables [13] with an expectation $\mathbb{E}(Z_n) = 0$. The successive values of an OU process are correlated exponentially with a variance $\mathrm{Var}(Z_n) = \sigma_z^2$ that is given by:

$$\sigma_z = \frac{\sigma^2 \tau}{2}(1 - e^{-2\frac{h}{\tau}}) \tag{3.6}$$

The correlation time $\tau = 100$ ms used for message generation has been chosen such that it is on the order of the membrane time constant and so that the probability values in the successive steps do not change too abruptly. The standard deviation has been set at $\sigma = 0.05$. The full algorithm listing is available in the appendix.

3.2.2 Target Message Calculation

All supervised learning procedures require a data set representing the desired outcome that must be reached by adjusting the parameters of the model. In the case of training individual readouts $R_{i\to j}$ at some factor f_i, the multivariate linear regression is based on the availability of a target current $I(t)$ that evokes a population firing rate $R_{i\to j}(t)$ which in turn is the linearly scaled time series of the message $\mu_{i\to j}(t)$ passed along the

edge x_j. Therefore, once the training input is available, this message can be calculated numerically using equation 2.45 where f_i and $p_{k\to i}(x_k, t)$ are substituted for this particular factor, as it is shown below.

A minor practical difficulty that arises in the context of the integration of equation 2.45 for f_i is that the standard *odeint* routine provided by Scipy [22] (the scientific computing library used for most of the calculations in this thesis) implements a variable-step integration procedure that is ill-suited for working with equidistantly sampled data such as the input messages $\mu_{k\to i}(t)$ that have been generated by the Ornstein-Uhlenbeck process. This necessitated the implementation of a custom integration routine, one that would use directly arrays of data.

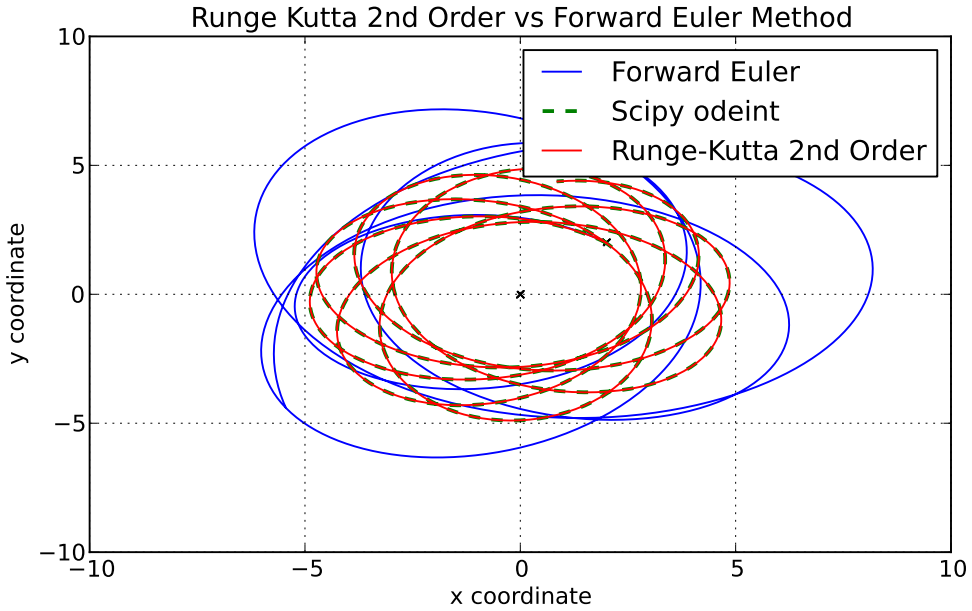


Figure 3.2: A test for examining the accuracy of the RK2 implementation. A mass point with a randomly generated initial velocity is attracted by an inverse-square force to the origin. The integration of the corresponding system of first-order differential equations is performed by 3 different integration routines, resulting in the trajectories shown in the figure.

The forward Euler method is the simplest solution to the problem and has been used by Steimer et al. [37]. Unfortunately, it is often numerically unstable and inaccurate, therefore a different fixed-step method, the 2nd order Runge-Kutta (RK2) integrator has been used in this thesis. It has the advantage of being less computationally complex in contrast to the more advanced Scipy algorithm, while retaining better numerical stability in comparison to the forward Euler method (the midpoint values are easily obtained by linear interpolation, see appendix). Figure 3.2 shows a comparison of all three methods and can be regarded as a validation for this particular implementation of

the RK2 algorithm.

For the purpose of training individual factors as well for comparing the performance of statistical inference carried out by the investigated models, it is necessary to derive specific versions of the sum-product update rule for each factor f_i. This is done by the substitution of the factor function f_i in the general form described by equation 2.45. An example derivation should further clarify the method.

Consider for instance a factor $f_i(x_1, x_2, x_3)$ that describes the joint probability density function of three binary variables x_1, x_2, and x_3:

		$x_1 = 0$
	$x_3 = 0$	$x_3 = 1$
$x_2 = 0$	0.8	0.2
$x_2 = 1$	0.2	0.9

		$x_1 = 1$
	$x_3 = 0$	$x_3 = 1$
$x_2 = 0$	0.2	0.8
$x_2 = 1$	0.8	0.1

Table 3.1: Definition of $f(x_1, x_2, x_3)$

If the message $\mu_{i\to 1}(t)$ along x_1 describes the probability that $x_1 = 1$, then the sum-product update rule from equation 2.45 (D is left out for simplicity) for m incoming messages is given by

$$M(t,1) := \sum_{X\in\{0,1\}^m} f_i(x_j = 1, X) \prod_{k\in\mathcal{N}/j} p_{k\to j}(x_k, t) \tag{3.7}$$

and in this case is reduced to

$$M(t,1) = \sum_{X\in\{0,1\}^2} f_i(x_1 = 1, x_2, x_3) p_{2\to i}(x_2, t) p_{3\to i}(x_3, t) \tag{3.8}$$

where $p_{k\to i}(x_k = 1, t) = \mu_{k\to i}(t)$ and $p_{k\to i}(x_k = 0, t) = 1 - \mu_{k\to i}(t)$. The final expression is the sum over all realizations of X and can be arrived at very efficiently in a tabular format:

The resulting message $M(t,1)$ is then the sum of all expression in the last column:

$$\begin{aligned} M(t,1) \;=\; & 0.2(1-\mu_{2\to i}(t))(1-\mu_{3\to i}(t)) + \\ & 0.8(1-\mu_{2\to i}(t))\mu_{3\to i}(t) + \\ & 0.8\mu_{2\to i}(t)(1-\mu_{3\to i}(t)) + \\ & 0.1\mu_{2\to i}(t)\mu_{3\to i}(t) \end{aligned} \tag{3.9}$$

$M(t,1)$		
$0,0$	$f_i(x_1=1,0,0)p_{2\to i}(0,t)p_{3\to i}(0,t)$	$0.2(1-\mu_{2\to i}(t))(1-\mu_{3\to i}(t))$
$0,1$	$f_i(x_1=1,0,1)p_{2\to i}(0,t)p_{3\to i}(1,t)$	$0.8(1-\mu_{2\to i}(t))\mu_{3\to i}(t)$
$1,0$	$f_i(x_1=1,1,0)p_{2\to i}(1,t)p_{3\to i}(0,t)$	$0.8\mu_{2\to i}(t)(1-\mu_{3\to i}(t))$
$1,1$	$f_i(x_1=1,1,1)p_{2\to i}(1,t)p_{3\to i}(1,t)$	$0.1\mu_{2\to i}(t)\mu_{3\to i}(t)$

Table 3.2: Calculation of $M(t,1)$

In order to obtain the normalizing factor $Z(t) = \frac{1}{M(t,0)+M(t,1)}$, the complementary values $M(t,0)$ have to be calculated too, but this time for $x_1 = 0$:

$M(t,0)$		
$0,0$	$f_i(x_1=0,0,0)p_{2\to i}(0,t)p_{3\to i}(0,t)$	$0.8(1-\mu_{2\to i}(t))(1-\mu_{3\to i}(t))$
$0,1$	$f_i(x_1=0,0,1)p_{2\to i}(0,t)p_{3\to i}(1,t)$	$0.2(1-\mu_{2\to i}(t))\mu_{3\to i}(t)$
$1,0$	$f_i(x_1=0,1,0)p_{2\to i}(1,t)p_{3\to i}(0,t)$	$0.2\mu_{2\to i}(t)(1-\mu_{3\to i}(t))$
$1,1$	$f_i(x_1=0,1,1)p_{2\to i}(1,t)p_{3\to i}(1,t)$	$0.9\mu_{2\to i}(t)\mu_{3\to i}(t)$

Table 3.3: Calculation of $M(t,0)$

$$\begin{aligned} M(t,0) &= 0.8(1-\mu_{2\to i}(t))(1-\mu_{3\to i}(t)) + \\ &\quad 0.2(1-\mu_{2\to i}(t))\mu_{3\to i}(t) + \\ &\quad 0.2\mu_{2\to i}(t)(1-\mu_{3\to i}(t)) + \\ &\quad 0.9\mu_{2\to i}(t)\mu_{3\to i}(t) \end{aligned} \tag{3.10}$$

Finally, the probability $\mu_{i\to 1}(t)$ is obtained by integrating the differential equation

$$\tau\dot{\mu}_{i\to 1}(t) + \mu_{i\to 1}(t) = \frac{M(t,1)}{M(t,0)+M(t,1)} \tag{3.11}$$

where $\mu_{2\to i}(t)$ and $\mu_{3\to i}(t)$ are either OU-generated training messages or some other time series used to test the approximation performance of the LSM representing the factor.

Due to the non-ideal nature of statistical inference when implemented by an artificial neural network, as well as other distortive effects that are introduced on neuroscientific grounds, it is necessary to build a theoretical model of the network for comparison purposes. Such a model takes the form of a linear system of differential equations where each equation describes the sum-product update rule of an individual factor as given in equation 3.11 and where the connections to some factor f_i are represented by incorporating the instantaneous values of the output messages of its peers into the messages $M(t,1)$ and $M(t,0)$.

3.2.3 Population Rate Encoding

The relation between the calculated normalized probability message $\mu_{i\to j}(t)$ and the actual instantaneous firing rate $R(t)$ of the readout population depends on the utilized encoding scheme. The simplest possibility is a linear transformation between these quantities, a method used by Steimer et al [37]. In order to reproduce the results from that work, the same scaling factor of $R(t) = 90$ Hz $\times\mu(t)$ has been used for the simulations presented in this chapter.

Conversely, the instantaneous population firing rate has also to be determined from experimental data in order to evaluate the performance of each trained readout. This leads to two important questions concerning the experimental setup:

- Which are the optimal parameters for population firing rate measurements?
- What is the optimal readout population size?

As already outlined in section 2.5.2, the rate can be measured by applying a smoothing filter to the spike trains produced by a population of neurons. Assuming that the Gaussian kernel used for the convolution is appropriately normalized in the computations, the single parameter governing the resulting instantaneous firing rate is its width (defined as $2\times$ its standard deviation σ).

Plots a) and b) in Figure 3.3 depict qualitatively the agreement between theoretically calculated firing rates and the ones measured from the population spike output. Both black curves are created from pre-generated probability time-series that have been linearly scaled to rates $R(t)$ in the range $[0, 90]$ Hz used in all simulations. Each spike train is produced by an individual inhomogeneous Poisson processes whose underlying rate is given by $R(t)$. Overlain on each black curve is the corresponding experimentally determined instantaneous population firing rate (red curves) with a filter width of 24ms. Plot a) represents a training message generated by an OU process, while b) shows the output of a simple factor describing a conditional probability $p(y|x)$ with the message from a) as its input. The performance measure is given by the correlation between the generating rate and the one determined experimentally.

A quantitative study of the rate measurement method is presented in plot c) where a sweep over a wide range of filter widths results in a maximum correlation when its standard deviation is about 14-15 ms. In order to avoid excessive smoothing of the data, the slightly lower value of 12 ms has been chosen for all further experiments presented in this chapter.

One reason for using a population of neurons instead of single synaptic connections for the purpose of encoding of probability messages is founded on the fact that the instantaneous firing rates of individual neurons do not correlate well with their target rates, as shown in the last plot of figure 3.3. The relatively high number of neurons used for the readout populations in Steimer et al. [37] (343) is justified when considering that

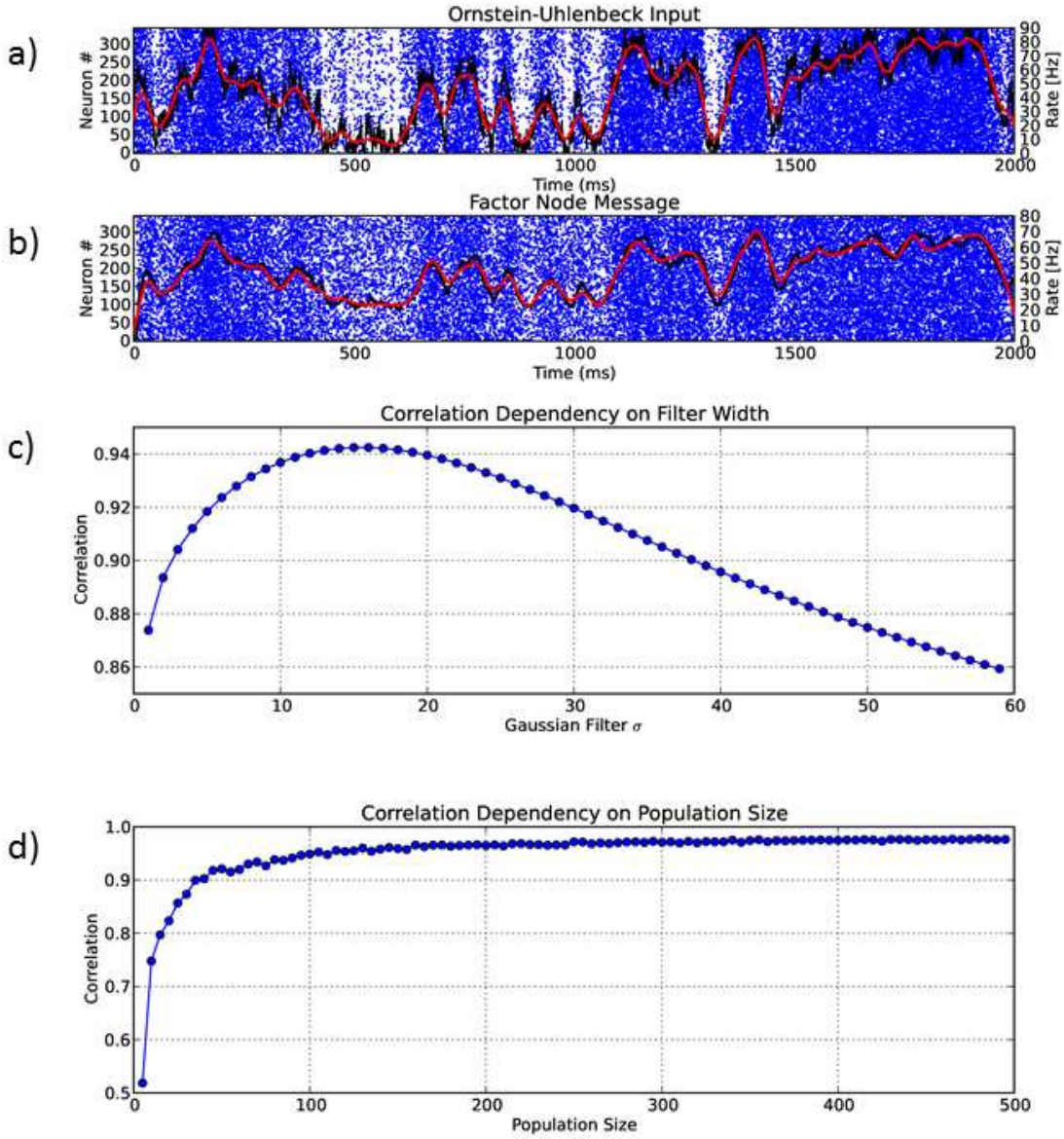


Figure 3.3: Measurement of the impact of different filter widths on the correlation performance measure, as well as its dependence on the readout population size used for the purpose.

the correlation stabilizes at about that population size. In addition, liquid pools of more than a thousand neurons require significant stimulation, which can be provided by larger readouts.

In addition, this relation can also be used for population size optimization which is especially important in the context of the bandwidth constraints of neuromorphic hardware, hence readouts with 50 to 100 neurons may present a reasonable performance trade-off considering that the correlation in that range is still well above 0.9.

3.2.4 Empirical Current-Rate (IR) Relation

The last step prior to the calculation of readout weights by linear regression is to determine an empirical relation between the total mean current entering a population of neurons and the resulting firing rate $R(t)$. The inverse function can then be used to determine the target current $I(t)$ which is a linear combination of the currents entering individual readout neurons.

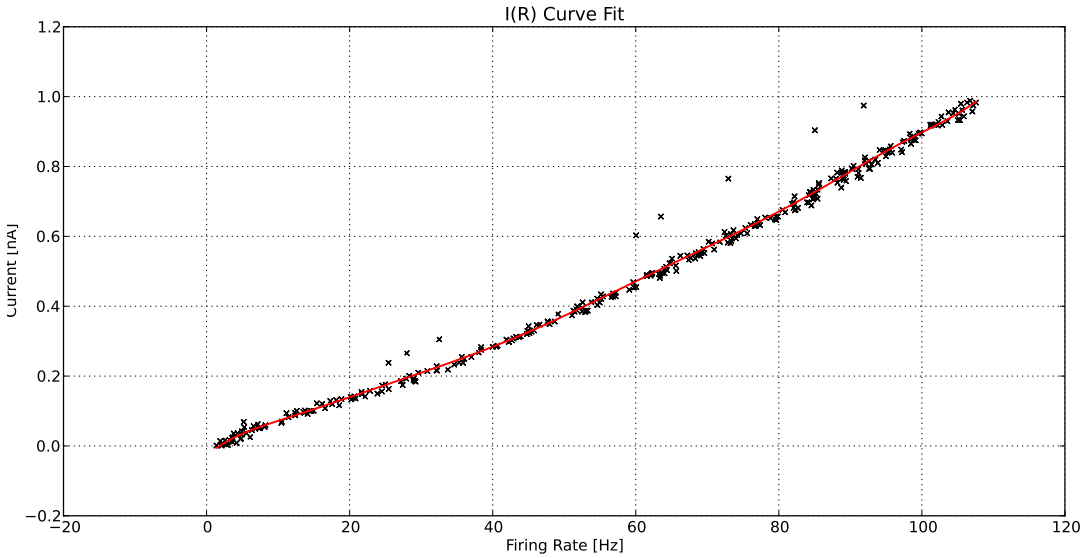


Figure 3.4: Measurement of the relation between the total injected current I and the resulting induced firing rate R. The data from 10 trials, each using 32 steps for 250 ms provided a total of 320 data points. The red line shows a polynomial fit on the data.

Characterization of the Readout Population

The readout populations form the crucial part of all neural network models described in this thesis and share the same architecture and parametrization except for their individually generated random thresholds. An instance of such a standard readout has also been used to determine the $I(R)$ relation and therefore merits a brief description.

Each readout population consists of 343 mutually unconnected LIF neurons with a resting potential of 0 mV and a membrane time constant $\tau_{\mathrm{m}} = 30$ ms. The reset potential is set at 10 mV and the absolute refractory time to $\tau_{\mathrm{r}} = 2$ ms. Due to linear regression used in the readout training, the synaptic time constants of $\tau_{\mathrm{syn}} = 6$ ms are the same for both the excitatory and the inhibitory inputs, since a designated "excitatory" neuron in the liquid pool may be assigned a negative weight by the procedure and thus has to project an inhibitory connection to a readout cell. All thresholds of the readout neurons are drawn from a uniform distribution in the range of $[15, 20]$ mV. This is necessary to avoid synchronous spiking of the whole population when innervated by the same input and also may play an important role in reducing the dependency of the readout response on past values of the total main current $I(t)$ [37].

The other characteristic that differentiates individual instances of the readout population is the zero-mean white noise current injected into every cell. An individual readout neuron thus receives a separate Gaussian white noise process whose realization at each time step is parametrized by a standard deviation $\sigma = 0.14$ nA. This is about 10% of the expected maximum stimulus (synaptic current at 90 Hz).

A constant background current of 0.5 nA (technically realized as the mean of the noise current) is necessary to avoid distortions in the low range of the firing rate (below 20 Hz) and also partially de-correlates the readout response. Without a background current, the matrix product $\mathbf{L}^T\mathbf{L}$ (see section 2.5.5) that needs to be inverted is often singular, due to empty rows when there is no input current.

Measurement of the I(R) Curve

The measurement of the I(R) relation is done by injecting a step current into a readout population with randomly generated thresholds (an instance of the standard readout). Each step has a duration of 250 ms and is drawn from a uniform distribution with a range of $[0, 1]$ nA. As it is shown on figure 3.5, this current evokes Poissonian firing rates between 0 and 100 Hz.

The relation between current and population firing rate is determined by the time average $< R(t) >$ over each step for that particular magnitude of the current. The width of the smoothing Gaussian filter used in the process is 24 ms.

For the evaluation of the IR curve, a polynomial function of degree 9 has been fitted on the data from 10 trials where each experiment consists of 32 discrete current steps. The resulting relation is presented in Figure 3.4.

3.2.5 Training Performance

Once that the target current $I(t)$ has been determined, the spike output of a liquid pool stimulated by the same messages, as those used in the theoretical computation, is needed to complete the calculation of the synaptic weights for connections between liquid and readout. The necessary simulation is done within the general software framework described in section 3.3 by instantiating a liquid using a set of random number generator

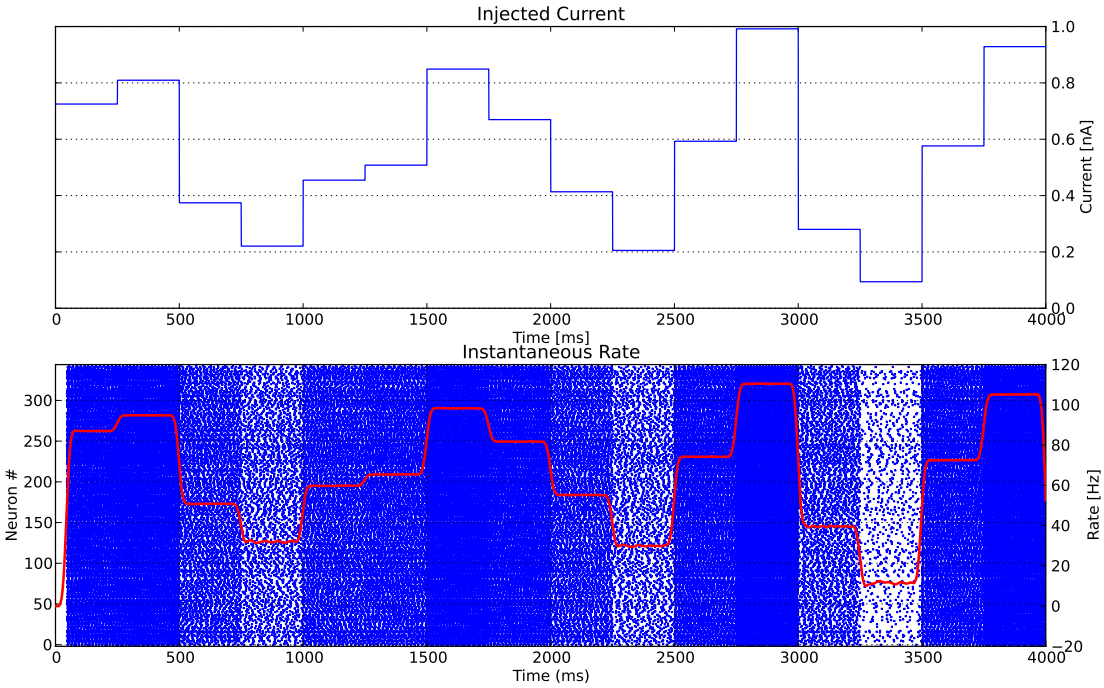


Figure 3.5: IR Curve measurement. Shown in a) is the current injected into each readout. The raster plot in b) depicts the firing activity induced by the current, while the red line shows the population activity firing rate calculated from the data.

seeds describing the stochastic aspects of its architecture and by connecting to it external spike sources that correspond to all of its static inputs (half-edges on an FFG) and readouts from other LSMs. The inputs are loaded with population spike trains generated by the training OU rates using an inhomogeneous Poisson process, as described in section 3.2.3. A decisive role in the performance of the training method is played by the architecture of the liquid pools.

Liquid Pool Architecture

The liquid state machine is a computational paradigm that does not pose strict requirements on the architecture of its physical implementation. It follows that similar structural units of a larger system that can be identified with liquid pools may be involved in completely unrelated calculations. This is indeed the case for the microcircuit anatomy of the human neocortex which provided much of the information necessary to build a liquid medium based on an artificial neural network [31], especially its connectivity patterns and the spatial relations among individual cells. Moreover, the recurrently connected neurons that belong to a cortical mini-column exhibit the separation property required by the LSM model, while their complex dynamics is still subject of much research.

The simulations used in this thesis use liquid pools whose neurons are arranged on a three-dimensional cuboid lattice (see figure 3.6). 20% of these neurons are chosen randomly to project inhibitory connections to the rest of the pool and also recurrently to themselves. They belong to the *inhibitory population*. The remaining 80%, on the other hand, project excitatory connections to the inhibitory population and also recurrently, thereby forming the *excitatory population*.

The connectivity of the liquid also follows the (biologically inspired) small-world topology, so that neurons are exponentially less likely to be connected to more distant ones than to their neighbors. The stochastic rule used in generating these connections is given by the probability $p = C \cdot \exp\left(\frac{-||(\mathbf{a}-\mathbf{b})||^2}{\lambda^2}\right)$ that any two neurons with coordinates $\mathbf{a}$ and $\mathbf{b}$ will be connected. The amplitude $C = 1$ was kept the same for all simulations, while the density parameter λ was empirically adjusted for each factor pool, since it exerts a heavy influence on the training performance and is therefore dominantly governing the separation property of the liquid.

Technically, the pool is constructed by forming four types of projections between both populations of neurons as shown in figure 3.6 a). The following parameters are used in Steimer et al. [37] and are based on [30]. The weights for the connections belonging to each projection are drawn from a gamma distribution with mean μ_{w} and coefficient of variation CV_{w} and are presented in table 3.4. The synaptic delays are normally distributed with mean μ_{D} and coefficient of variation CV_{D}.

Connection Type	μ_{w} [nA]	CV_w	μ_D [ms]	CV_D	τ_{syn} [ms]
Exc $\rightarrow$ Exc	30	0.7	1.5	0.1	3
Exc $\rightarrow$ Inh	60	0.7	0.8	0.1	3
Inh $\rightarrow$ Exc	-19	0.7	0.8	0.1	6
Inh $\rightarrow$ Inh	-19	0.7	0.8	0.1	6

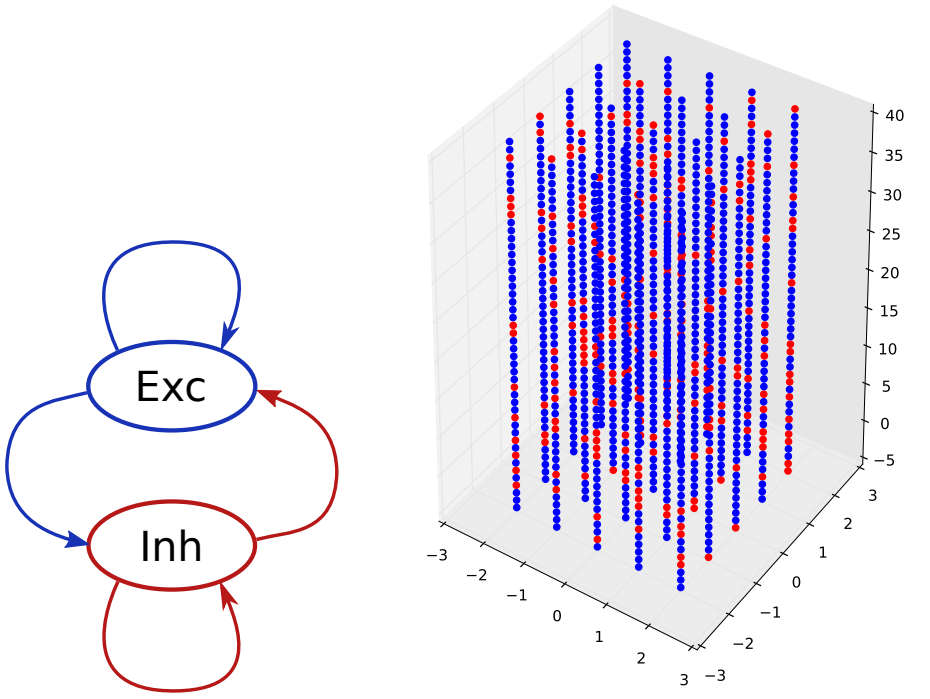
Table 3.4: Liquid pool connection parameters

The gamma distribution is given by

$$p(x; k, \theta) = \frac{1}{\Gamma(k)\theta^k} x^{k-1} e^{-\frac{x}{\theta}} \quad \text{for } x \geq 0 \text{ and } k, \theta > 0 \tag{3.12}$$

Its shape parameter k and scale parameter θ are related to its mean μ and variance σ^2 trough the relations $\mu = k\theta$ and $\sigma^2 = k\theta^2$. The main utility of the gamma distribution is founded on the fact that it is positive-valued and also that it can take various asymmetric or symmetric shapes, depending on the parameter k.

The LIF models of all liquid neurons are parametrized according to [30] and are presented in table 3.5. Parameters which are drawn from a uniform distribution are indiciated by the corresponding range of values.



(a) Connectivity in the liquid pool. Excitatory neurons project excitatory synapses to the inhibitory pool and also recurrently, while inhibitory neurons project inhibitory connections to the excitatory populations and also recurrently.

(b) Spatial arrangement of the liquid pool. The red dots represent inhibitory neurons that are selected at random. They comprise 20% of the total number of neurons in the column.

Figure 3.6: Liquid pool architecture

In addition, each cell receives a constant background current that is drawn from a uniform distribution in the range $[13.5, 14.5]$ nA. No background noise is applied to the liquid neurons.

Factor Node Architecture

For training purposes, an isolated factor node must be constructed in terms of a liquid pool of neurons, a set of readouts, and static spike inputs representing incoming probability messages. The arrangement of these neural populations are based on the FFG encompassing the factor node and its immediate neighbors.

Since observed variables are represented by half-edges, their network manifestation

Population Type	V_{rest} [mV]	V_{reset} [mV]	V_{thresh} [mV]	τ_{m} [ms]	τ_{refrac} [ms]
Exc	0	$[13.8, 14.5]$	15	30	3
Inh	0	$[13.8, 14.5]$	15	30	2

Table 3.5: Liquid pool connection parameters

is a population of spiking neurons that emits pre-generated spike trains with a population rate corresponding to the value of the corresponding variable (like I, P_2 on figure 3.7 a)). Each full edge, on the other hand, designates a bidirectional transfer of information and consequently a readout projecting to the liquid pool as well as an outgoing one. There is an important difference in the neural implementation between neighboring factor nodes that are leaves on the FFG and those that are not. The former are represented by a static input mirroring the values of the associated factor function and an outgoing readout projecting into its liquid pool. The latter project their readouts into the liquid and receive the same number of matching outgoing readouts. An example that includes all of the connection types outlined above is given in figure 3.7.

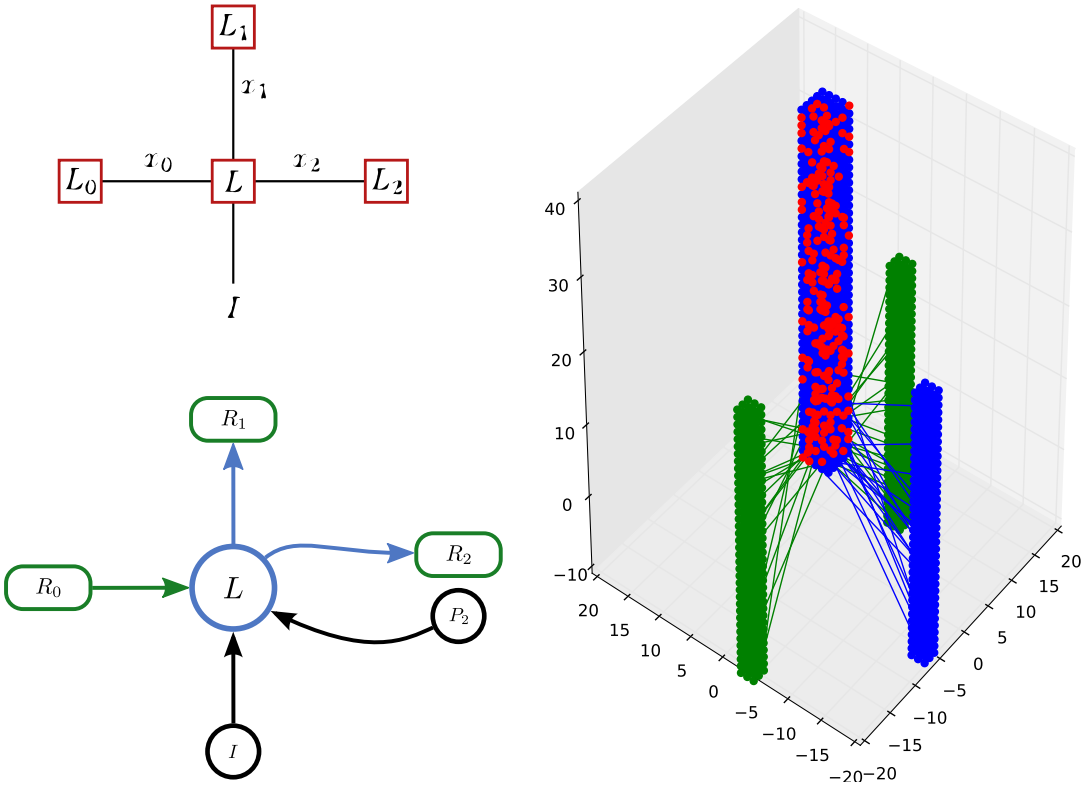
It is important to recall that in the flooding schedule regime the calculations of the output function along each edge are done simultaneously. Therefore, given an edge x_j along which one such output is to be transmitted, the theoretical calculations include only the incoming messages along $x_k,\ k \in \mathcal{N}(i)/j$ while in reality the liquid also receives input from the matched readout along x_j. Obviously, this has an influence on the computations performed by the LSM and in order to avoid mixing of information, each input (static or readout) projects into a different region of the liquid pool [37], as shown in figure 3.7 b).

Each readout/input forms two separate projections to sub-populations of the excitatory and inhibitory liquid neurons that are confined to its assigned region of the column. These projections are parametrized differently, as presented in table 3.6 (The readout/input neurons are all excitatory). The weights and delays use the same types of probability distributions as those used in the liquid pool interconnections.

Projection Type	μ_{w} [nA]	CV_{w}	μ_{D} [ms]	CV_D	τ_{syn} [ms]
Exc $\rightarrow$ Exc	4.5	0.7	1.5	0.1	3
Exc $\rightarrow$ Inh	9	0.7	0.8	0.1	3

Table 3.6: Readout and input connection parameters

Each readout neuron in an LSM factor node is connected to all neurons in the liquid. The synaptic time constant for both excitatory and inhibitory synapses is $\tau_{\text{syn}} = 6$ ms and the synaptic delays are fixed at 1.5 ms for all connections. Since their weights are determined by linear regression, excitatory liquid neurons may inhibit readout neurons and inhibitory neurons may excite them. All readout neurons use the same set of weights.



(a) FFG diagram and an example factor node **(b)** Liquid pool with inputs projecting to the same part

Figure 3.7: Factor architecture

Training Method Validation

The training method can be validated by first applying it to an individual factor node, followed by a comparison between the theoretically calculated time-series of the function and the actual LSM output when the system is stimulated by newly generated random probability messages.

Figure 3.8 presents the training performance of an equality constraint factor $\delta(z - x_1)\delta(z-x_2)$ (see section 2.2.3) that achieves a correlation with the theoretically calculated message higher than 0.92 for a novel input generated by the same OU process. It is defined by a time constant $\tau = 100$ ms and a variation of $\sigma = 0.05$ Hz. Readout projections are substituted with static inputs using exactly the same connections to the liquid.

3.3 Simulation Framework

A general software framework that allows for the definition and training of arbitrary FFG-based belief propagation models has been developed in conjunction with the factor

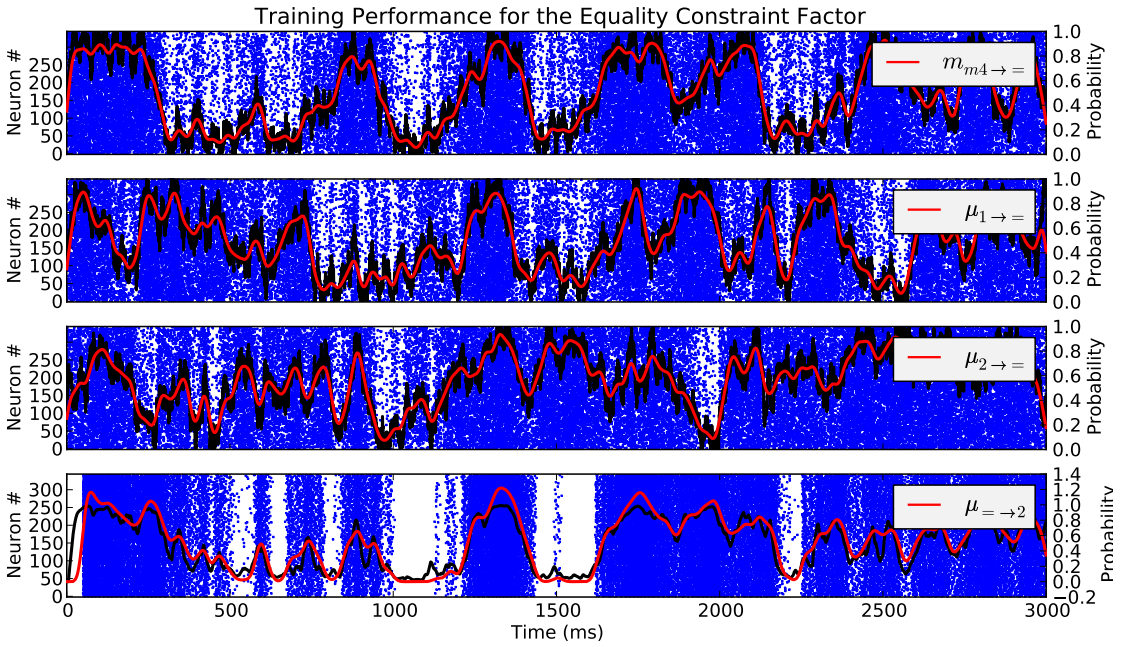


Figure 3.8: Training performance of the readout that connects the equality constraint factor in the explaining away model (see section 3.4.2) with factor P_2. The correlation between the readout output and the theoretically calculated message is 0.93.

graphs investigated in this thesis. It implements a simple command-line interface that enables the automatic execution of all training steps described above and includes some useful data plotting utilities that are indispensable when analysing simulation outcomes. All of the software libraries mentioned in section 3.1 such as Scipy, PyNN, and NeuroTools are used by the framework to perform the many preparatory and simulation tasks necessary in building a successful neural statistical inference model. Figure 3.9 depicts the typical workflow for inference model simulations.

Central to the ability to define such models easily is the ParameterSet class provided by NeuroTools. Objects of this type are used to store the great number of parameters necessary to define a complex PyNN-based neural network such as the binary channel described below in a consistent and hierarchical manner. The framework uses a text-based serialization routine to store the parametrization of particular instances of network objects in an accessible human-readable format.

A particular inference model is defined in a global read-only parameter file that describes the graph hierarchically, starting from its comprising factors, down to the readouts and inputs associated with them. The overall connectivity of the graph (edges in the FFG translated to neural populations) is also stored in the parameter file, as well as the parameters necessary to describe the excitatory and inhibitory populations and

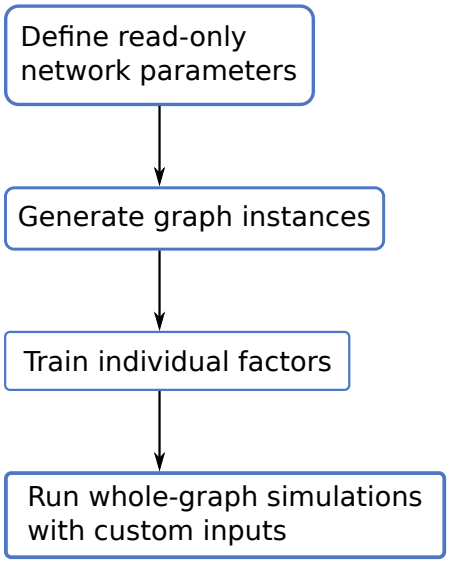


Figure 3.9: Workflow in the simulation framework

projections.

The first step toward the simulation of belief propagation using the framework is to instantiate the inference model, based on its defining parameters. They are used to generate all neuron populations (liquid pools, readouts) and the projections among them with the PyNN interface to a simulator (the simulations in this thesis have been performed with NEST as the back-end simulator). All parameters describing this particular instance of the graphical model (such as random number generator seeds for connections and other stochastically generated objects) are saved in a tree-like directory structure reflecting the organization of the FFG. This allows the deterministic instantiation of exactly the same network when it is used to perform inference with particular values of its observed variables.

The training step involves the usual step of generating training messages using the Ornstein-Uhlenbeck process, computation of theoretical values and the calculation of synaptic weights for the readouts. The stimulation of isolated factors with training input is done by instantiating the corresponding liquid pool using data from the pre-generated graph and by substituting readouts from other factors with static inputs that, however, use exactly the same connections as those readouts and project to the same part of the liquid. The calculated weights are finally written to connectivity files in the graphical model directory that are used to instantiate the factors.

The trained graphical model can perform statistical inference using arbitrary custom inputs, for instance to evaluate its performance.

3.4 Models

After the required training steps have been applied to all factors belonging to the corresponding neural FFG model, these artificial neural networks approximate the computations involved in the belief propagation algorithm. Two such models are considered next and their performance is evaluated by comparing their output to the expected response given the same input messages.

3.4.1 Noisy Binary Channel

Since the implementation of the sum-product algorithm in spiking neural networks is an emerging field and has so far been demonstrated mainly in the pioneering work of Steimer et al. [37], it is beneficial to start with a theoretical model that is widely used in the belief propagation literature as a proof-of-concept for the methods used in this work. Such a model, which is both simple and realistic, is the *noisy binary channel*, described in Löliger 2004 [27]. It represents an imperfect parallel digital transmission channel that is n-bit wide and where each line has an unreliability that is modeled by a small probability of flipping the bit while in transmission, so that the receiving side detects its opposite value. This is a problem of great practical importance in digital electronics where it is necessary to estimate the probability with which the received signal is the originally sent one, based on the data from the whole transmission.

Many error-correcting schemes have been developed over time and the sum-product algorithm provides an efficient framework for delivering optimal estimates. Graphs are a convenient representation of error-correcting codes because they allow for the straightforward derivation of many such algorithms.

There are two main types of error-correcting code: block codes and convolutional codes. The former require a fixed-size communication channel and have had a fundamental importance in the development of information theory. The first *linear* block code was described by Hamming and is probably the most famous error-correcting algorithm.

Linear Error-Correcting Block Codes

When a block code is used to transfer data over an unreliable channel, the stream is broken down into *messages* of length k which are then encoded into *blocks* (*codewords*) of length n by the block code C. These blocks are sent out either serially or in parallel and the receiver uses an error-correcting scheme to recover the original message from the codeword, even if individual bits have been corrupted during transmission.

Formally, an error correcting block code C of length n over some alphabet $\mathcal{A}$ is defined as an injective mapping $C : \mathcal{A}^k \mapsto \mathcal{A}^n$. It is linear if the alphabet $\mathcal{A}$ is a field $\mathbb{F}$ and the code C is a subspace of the corresponding vector space $\mathbb{F}^n$ [27]. For a binary code this field is $\mathbb{F}_2$ (the field of the binary numbers). The linear block mapping any linear code

can then be represented by its encoding action over the messages:

$$C = \{uG : u \in \mathbb{F}^k\} \tag{3.13}$$

and the correcting operation:

$$C = \{x \in \mathbb{F}^n : Hx^T = 0\} \tag{3.14}$$

where H and G are matrices over $\mathbb{F}$. The $k \times n$ matrix G is the *generator matrix* for the code C and maps a message $u \in \mathbb{F}^k$ of information symbols with length k to a codeword (block) $x = Gu$. The matrix H is called the *parity check matrix* for C and is essential in recovering the original message.

The *Hamming code* is a well-known linear block code. The parity check matrix for such a code of length $n = 7$, dimension $k = 4$, and a minimum *Hamming distance* $d = 3$ is given by:

$$H = \begin{pmatrix} 1 & 1 & 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 1 & 1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 1 & 1 & 0 & 1 \end{pmatrix} \tag{3.15}$$

A membership indicator function $I_C : \mathbb{F}^n \mapsto \{0, 1\}$ that determines whether a codeword belongs to the code C is defined by

$$I_C(\mathbf{x}) = \begin{cases} 1, & \text{if } x \in C \\ 0, & \text{else} \end{cases} \tag{3.16}$$

The exact form of the membership indicator can be easily derived from equations 3.13 and 3.15:

$$\begin{aligned} I_C(x_1, \ldots, x_n) = \quad & \delta_1(x_1 \oplus x_2 \oplus x_3 \oplus x_5) \cdot \\ & \delta_2(x_2 \oplus x_3 \oplus x_4 \oplus x_6) \cdot \\ & \delta_3(x_3 \oplus x_4 \oplus x_5 \oplus x_7) \end{aligned} \tag{3.17}$$

where $\delta(w) = \{1 \text{ if } w = 0;\ 0 \text{ otherwise }\}$ is the (discrete) Kronecker delta function and $\oplus$ denotes addition modulo 2 over the field $\mathbb{F}_2$ (XOR binary logical bit operator).

Since the membership function effectively determines if a codeword belongs to C, a factor graph of such a code represents the factorization of the associated indicator function I_C.

Figure 3.10 shows the factor graph of function I_C. Any linear binary block code can be represented by a function of the same type, which is based on the parity check matrix [27].

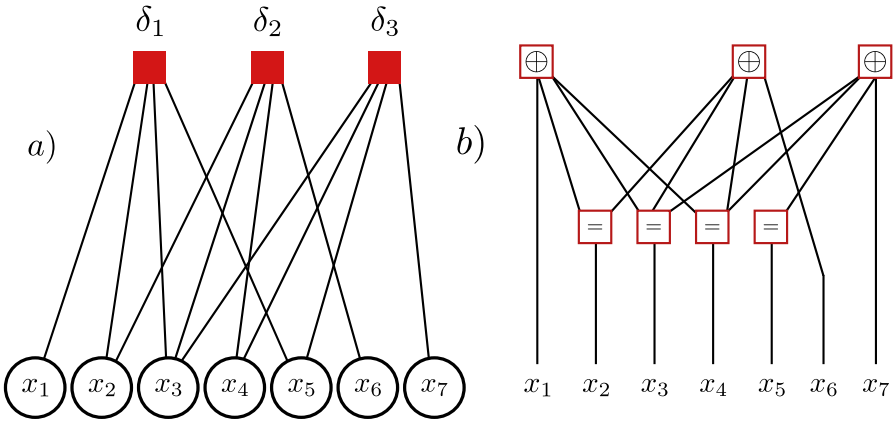


Figure 3.10: Code factor graph representation: a) Bipartite factor graph, b) Forney factor graph

Memoryless Channel Model

Any physical information transfer medium is non-ideal in the sense that signals propagating in it are typically corrupted by noise. If binary data is transmitted in a parallel fashion through an n-bit wide channel (see figure 3.11), there is always a *crossover probability* ε that the value of any bit is reversed on the receiving end. In simple devices (such as wires) this probability does not depend on the previous values of the data and they are correspondingly designated as *memoryless*.

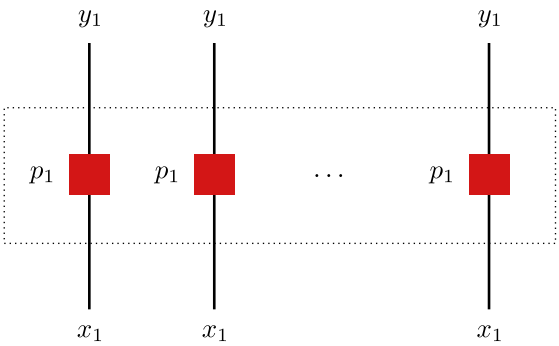


Figure 3.11: Memoryless Channel

The formal representation of such *memoryless channels* comprises a family $p(y|x)$ of probability distributions over a block $y = (y_1, \ldots, y_n)$ of channel output symbols given any block $x = (x_1, \ldots, x_n)$ of channel input symbols. The conditional probabilities $p_l(y_l|x_l)$ represent the error likelihood in each channel line and if the individual lines are not correlated with each other, the joint probability function describing the channel

consists of their product:

$$p(y|x) = \prod_{k=1}^{n} p(y_k|x_k) \tag{3.18}$$

Since the receiver is interested in assessing the probability that the received data is the one originally sent, it is possible to construct a joint code/channel factor graph as shown in figure 3.12 by feeding in the received codeword to the membership function.

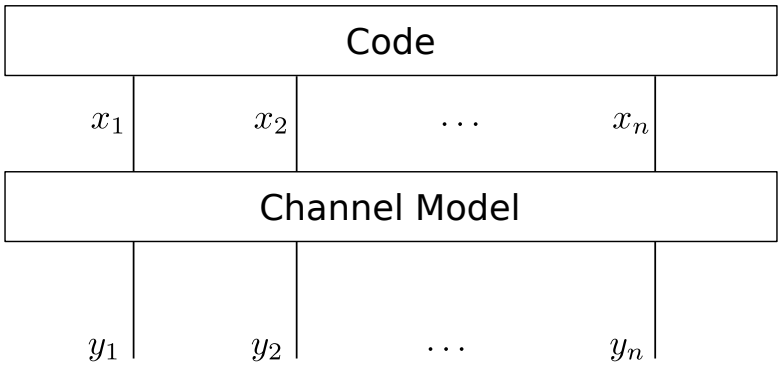


Figure 3.12: Joint Code/Channel FFG

This corresponds to the joint likelihood function $p(y|x)I_C(x)$. If the codewords are equally likely to be transmitted, for any fixed received block y, the a posteriori joint probability of the coded symbols $x_1, \ldots, x_n$ is:

$$p(x|y) = \frac{p(y|x)p(x)}{p(y)} \propto p(y|x)I_C(x) \tag{3.19}$$

A Four-Bit Noisy Binary Channel Model

The concrete noise binary channel chosen to as the first statistical inference model is 4-bit wide and has been described in Löliger et al. [27]. Its code is relatively simple, and can be given directly as a set of bit tuples:

$$C = \{(0,0,0,0), (0,1,1,1), (1,0,1,1), (1,1,0,0)\} \tag{3.20}$$

The crossover probabilities are equal for each line and are defined as follows:

$$p(y_l|x_l) = \begin{cases} 0.8, & \text{if } y_l = x_l \\ 0.2, & \text{if } y_l \neq x_l \end{cases} \tag{3.21}$$

The FFG of the joint binary channel represents the factorization of the a posteriori probability $p(x_1, x_2, x_3, x_4, z|y_1, y_2, y_3, y_4)$ where z is an auxiliary variable. The graphical model consists of three distinct types of factors, as shown in figure 3.13:

By definition, a variable in an FFG links only two factors and since the last bits x_3 and x_4 of the code are always equal, a succinct version of the parity check function takes the

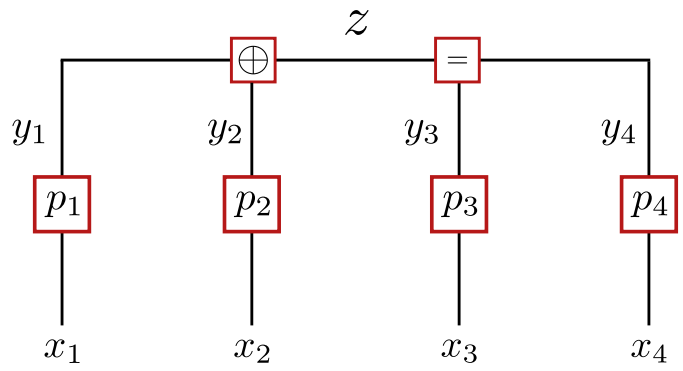


Figure 3.13: Binary Channel FFG

form $f_{\oplus}(x_1, x_2, z) = \delta(x_1 \oplus x_2 \oplus z)$ which necessitating the introduction of an auxiliary variable of the equality constraint function $f_{=}(x_1, x_2, z)$. The a posteriori probability thus factorizes as in the following way:

$$p(x_1, x_2, x_3, x_4, z|y_1, y_2, y_3, y_4) \propto f_{\oplus}(x_1, x_2, z) f_{=}(x_3, x_4, z) \prod_{i=1}^{4} p_l(y_l|x_l) \tag{3.22}$$

Belief propagation using the flooding schedule regime delivers the marginal a posteriori probability function $p(x_i|y_1, y_2, y_3, y_4)$ for each bit in the channel simultaneously and can be read from the population rate of the readouts projecting to the p_i factors in the neural network. The architecture of the latter is presented in the next section.

Training Results

The network corresponding to the FFG shown in figure 3.13 has been constructed according to the rules presented in section 3.2.5. Its architecture is depicted in figure 3.14 and was implemented in the software framework discussed in section 3.3.

All readouts and external input sources in the graph consist of 343 neurons and are connected with the corresponding liquid pools according to the description given in section 3.2.5. Each factor was trained individually with OU messages that were 45 s long. The sizes of the liquid pools are based on the supplementary data given in [37] and are summarized in table 3.7 (the small size of liquids 1-4 reflects their simple factor function, as they describe the conditional probability $p(y_l|x_l)$).

It has been found empirically that an optimal connection density parameter for the liquids is $\lambda = 4.5$. For most of the factors, the stimulation of their liquids with novel input generated by the same OU process as the one used for training, resulted in correlation coefficients between the readout probability messages and the theoretically calculated ones in excess of 0.9.

Unfortunately, it was not possible to produce correlation higher than 0.7 for the addition-modulo-2 (XOR) factor that is involved with the parity check. Higher values of λ did not lead to the expected improvement of its performance, as neither did

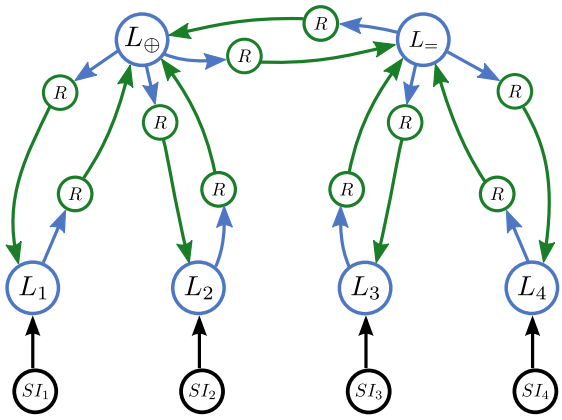


Figure 3.14: Neural network architecture of the binary channel model

Liquid Name	Column Structure [width × length × height]	Total Number
$L_{1,2,3,4}$	$5 \times 5 \times 12$	300
$L_{=}$	$5 \times 5 \times 40$	1000
$L_{\oplus}$	$5 \times 5 \times 78$	1950

Table 3.7: Number of neurons used for liquid pools in the explaining away model

a significant increase in the number of neurons of its liquid pool (2500). It is possible that longer training times might have exerted a positive influence on the training performance, but due to the algorithmic complexity of the matrix multiplication involved in the training procedure (equation 2.50), they were not feasible on the available hardware. This leads to the conclusion that an additional parameter search study is necessary to correct this problem.

Figure 3.15 shows that after the training procedure, the XOR factor produces a messages that shows little correlation with the theoretically expected output.

Since it is not possible to perform meaningful statistical inference experiments with an untrained factor, the binary channel model has to be scrutinized further, in order to achieve its target performance.

3.4.2 Explaining Away Model

A psycho-physical inference problem in the area of human visual perception, first described in [24], lends itself well to a Bayesian interpretation in the form of a probabilistic graphical model. Since it has a biological significance, it is highly relevant to the effort of implementing statistical inference in networks of spiking neurons, and, as it will be presented in this section, has been successfully tested in a software simulation. From now on it will be referred to as the "Explaining Away" model, the reasons for which shall be immediately obvious from its description.

The Explaining Away Model is concerned with two alternative hypotheses about the

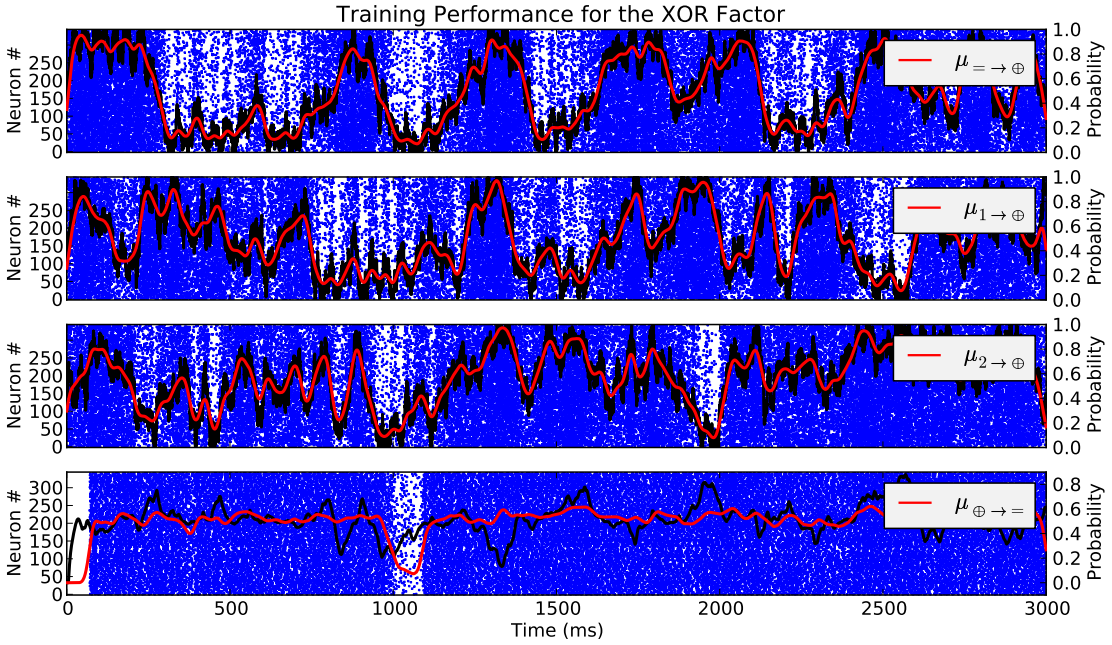


Figure 3.15: Training performance evaluation of XOR factor in the binary channel graph. The correlation with the expected output is 0.38.

properties of adjacent three-dimensional objects in a specially arranged visual scene. For a particular set of observables, one of these hypothesis prevails, thus explaining away the other one. When two cylinders with an identical photo-metrical luminance profile are attached to each other, they appear the same to the observer. When they are replaced by two cubes with the same luminance profile, however, the objects are perceived differently, as it is shown in figure 3.16 a).

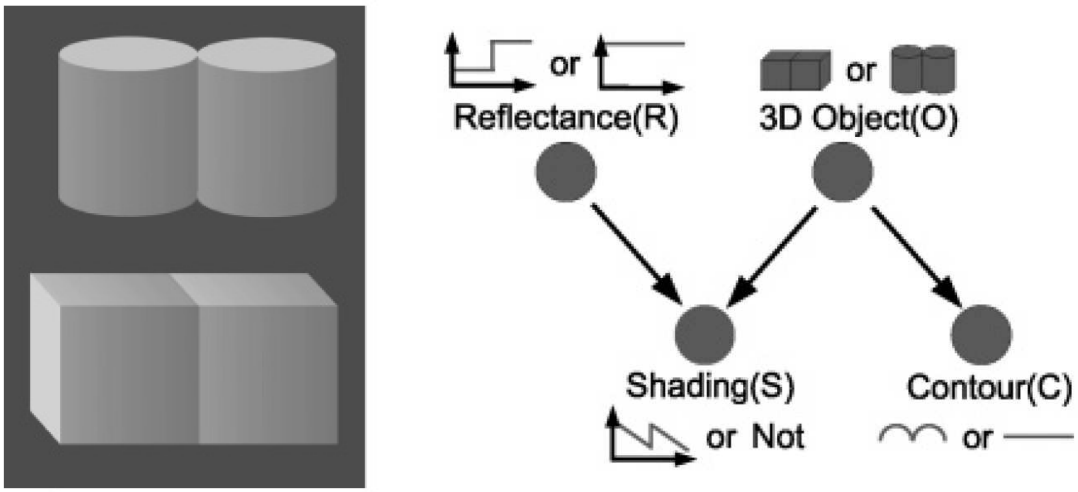
The first hypothesis postulates that the material reflectance (random variable $R = 0$, see figure 3.16 b)) of both objects is equal and dominates when they have round contours (observable C in the same figure). This is explained by the fact that cylindrical shapes make it more likely that the source of illumination is located in the left part of the scene [37].

If, on the other hand, the contours are straight, then the second hypothesis prevails, since it assumes uniform illumination and a higher reflectance of the right object ($R = 1$), as it is the cases for the cubes.

The Bayesian network shown in figure 3.16 b) includes the conditional probabilities $P_1(S|R, O)$ and $P_2(C|O)$ that are defined in tables 3.8, 3.9:

It corresponds to a joint probability distribution $P(S, R, O, C)$ that factorizes according to:

$$P(S, R, O, C) = P_1(S|R, O)P_2(C|O)P_3(R)P_4(O) \tag{3.23}$$



(a) Knill & Kersten's lightness illusion, picture taken from [37].

(b) Bayesian network representation of the explaining away problem

Figure 3.16: Visual demonstration of the explaining away problem and the corresponding Bayesian network. The binary variable R, denoting material reflectance, determines which of both hypotheses prevails. It is either $R = 0$ when the reflectance is homogeneous, or $R = 1$ when the right object reflects more light. The *observed* luminance profile S can be as shown in b) ($S = 1$) or not ($S = 0$). The type of the objects is given by O, which takes on value $O = 0$ when they are cylinders, or $O = 1$ when they are cubes. Finally, the other observable, C, determines whether the objects are curved ($C = 1$) or straight ($C = 0$). The definition of the conditional probabilities $P_1(S|R, O)$ and $P_2(C|O)$, as well as the factorization of the joint probability distribution, are given in the main text.

The probability for the reflectance, which determines the confidence in the second hypothesis ($R = 1$), is given by the marginal probability $P(R|S, C)$, which can be obtained using the sum-product algorithm. This requires the construction of the factor graph shown in figure 3.17 a). As it is evident from equation 3.23, the variable O is an argument to more than two functions, which necessitates the introduction of an equality constraint factor $\delta(O-O')\delta(O-O'')$ that is linked to $P_1(S|R, O)$, $P_2(C|O)$, and $P_4(O)$.

The details of the neural implementation of the graphical model are presented in figure 3.17 b). It has been developed using the software framework described above (section 3.3) and the results from the performed experiments are going to be discussed next.

Simulation Results

All factors shown in figure 3.17 b) have been trained separately for 45 s simulation time with stochastically generated input messages, produced by an Ornstein-Uhlenbeck

		$S = 0$
	$O = 0$	$O = 1$
$R = 0$	0.8	0.2
$R = 1$	0.2	0.9

		$S = 1$
	$O = 0$	$O = 1$
$R = 0$	0.2	0.8
$R = 1$	0.8	0.1

Table 3.8: Definition of the explaining away factor $P_1(S|R, O)$

	$O = 0$	$O = 1$
$C = 0$	0.8	0.2
$C = 1$	0.2	0.8

Table 3.9: Definition of the explaining away factor $P_2(C|O)$

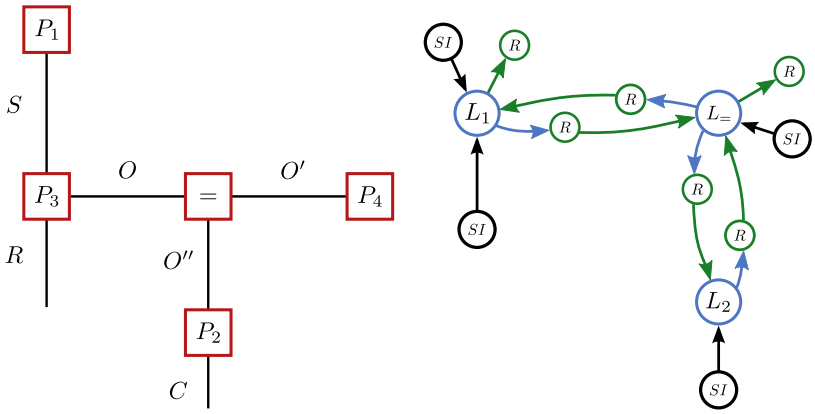
process, as described in section 3.2.5. It should be noted, that, since P_3 and P_4 do not perform any calculations, being leaf nodes on an FFG, they are replaced by externally generated output corresponding to the values of the variables R and O. All readouts and inputs consist of 343 neurons and the number of neurons comprising each liquid pool are given in table (3.10):

Liquid Name	Column Structure [width × length × height]	Total Number
L_1	$5 \times 5 \times 78$	1950
L_2	$5 \times 5 \times 42$	1050
$L_=$	$5 \times 5 \times 42$	1050

Table 3.10: Number of neurons used for liquid pools in the explaining away model

In the course of the training simulations it has been found that the optimal connection density parameter leading to the best training performance is $\lambda = 4.5$ for P_2 and $P_=$, and $\lambda = 7.0$ for P_1.

After training, the performance of the statistical inference model has been evaluated from the results of an experimental series comprising 600 experimental trials involving the same instance of the neural network implementation. Each simulation was run for 1 s biological time with 4 randomly generated inputs, corresponding to the input messages $\mu_{P_3 \to P_1}(t)$, $\mu_{P_4 \to P_=}(t)$, $\mu_{S \to P_1}(t)$, and $\mu_{C \to P_2}(t)$. These messages were constant in time, with a value drawn from a uniform distribution. In addition to the probability



(a) Forney Factor Graph implementation of the explaining away problem

(b) Neural network implementation of the FFG representing the explaining away problem

Figure 3.17: Factor graph models corresponding to the explaining away problem

time-series recorded from each readout, a theoretical calculation based on the same input has been performed for each trial for evaluation purposes.

The performance analysis is based on the difference between the calculated steady-state solutions for the readout probability messages, μ_{correct}, and those obtained experimentally from the simulation. Since the inputs are constant in time, the network reaches a steady-state in about 200 ms simulation time. Therefore the average value $\mu_{\text{spiking}} = \langle \mu_{\text{spiking}}(t) \rangle$ over the remaining 800 ms was subtracted from μ_{correct} to obtain the error histograms shown in figure 3.19:

As a control for each experiment, a randomly generated probability μ_{random} (from a uniform distribution) was subtracted from each μ_{correct}, leading to the distributions shown as a background in each histogram. For each trial, the Kullback-Leibler divergence between the experimentally determined error $\mu_{\text{correct}} - \mu_{\text{spiking}}$ and a random binary distribution has been calculated, whereby the results have been averaged to produce the expected values shown besides each histogram. The same calculation has also been applied to the errors $\mu_{\text{correct}} - \mu_{\text{random}}$. While messages $\mu_{2\to=}$ and $\mu_{=\to 1}$ show a relatively stable performance when compared to randomly generated values, many of the other readouts produce error distributions that appear rather close to their random counterparts. The correlation coefficients that have been obtained from a single-trial experiment using Ornstein-Uhlenbeck generated input are presented in table 3.11 and show a similar pattern.

Clearly, the simplest factor node, P_2, which is also directly supplied with external input, emits a highly correlated message, since this corresponds exactly to the setup that is used for training. More complex nodes, on the other hand, generally exhibit weaker performance, since they have to process secondary messages and are therefore located deeper in the network. In fact, this result is one of the main motivations for the

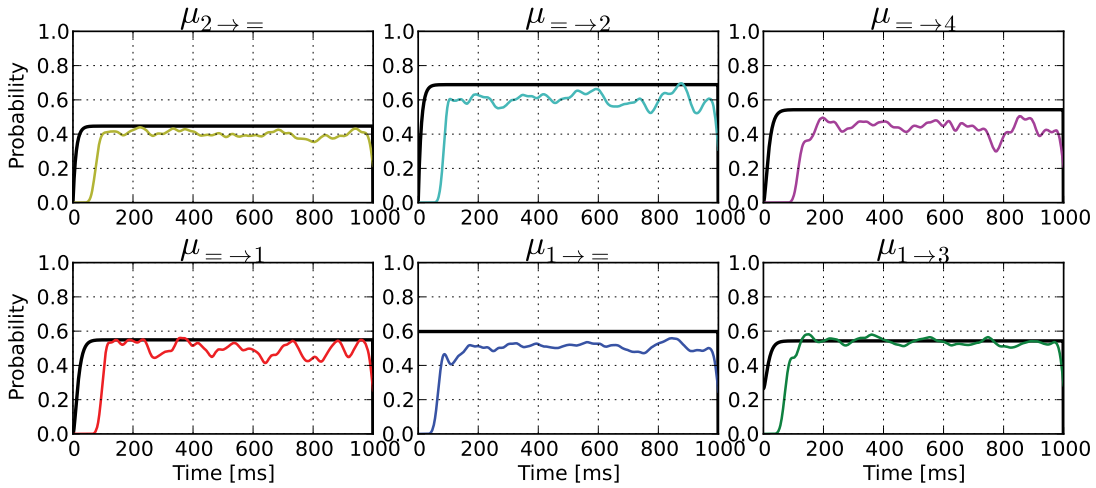
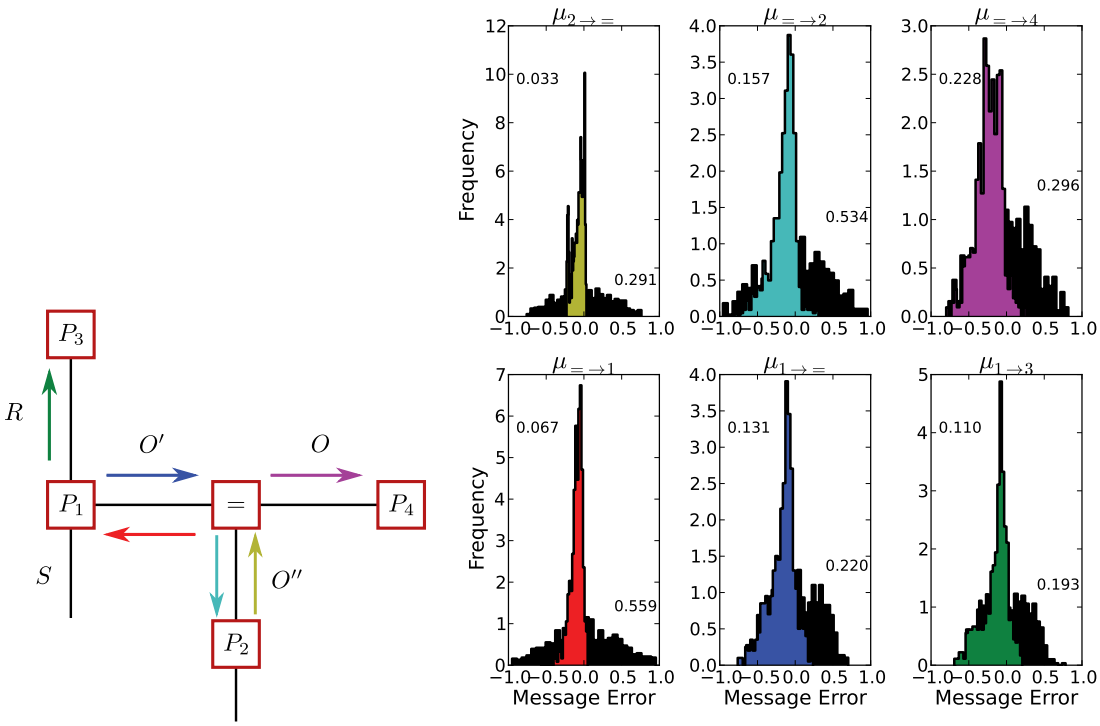


Figure 3.18: Theoretically calculated messages (black) and color-coded traces corresponding to the experimentally determined readout probability messages. The figure represents the results from a single trial of the experimental series described in the main text.

Message	$\mu_{1\to3}$	$\mu_{1\to=}$	$\mu_{=\to1}$	$\mu_{=\to2}$	$\mu_{=\to4}$	$\mu_{2\to=}$
CC	0.22	0.21	0.93	0.90	0.65	0.90

Table 3.11: Correlation between readout messages and theoretically calculated values for a single 2000 ms trial where the input messages were generated by an Ornstein-Uhlenbeck process.

development of the simplified Kalman filter model, which is constructed in such a way, that each node receives external input.



(a) Messages passed along the edges of the explaining away FFG

(b) Normalized performance histograms (30 bins) for individual messages. The evaluation method is described in the main text. Their color codes are identical to those used in a). The black areas represent the difference $\mu_{\text{correct}} - \mu_{\text{random}}$.

Figure 3.19: Performance analysis of the explaining away model

4 Calibration of Neuromorphic Hardware

The work described in this chapter is aimed at implementing belief propagation models on neuromorphic hardware. The neuromorphic device used for this purpose has been developed within the FACETS[1] [14] research project and represents a prototype device (FACETS stage1) for a larger wafer-scale system that is expected to be operational within the time frame of the BrainScaleS[4] initiative. A *mixed-signal* ASIC[2] chip called "Spikey" forms the core of the system. Its analog circuits comprise a highly configurable artificial neural network (ANN) of leaky integrate-and-fire neurons, while its digital electronics are responsible for spike event routing, parameter setting, and communication with external devices.

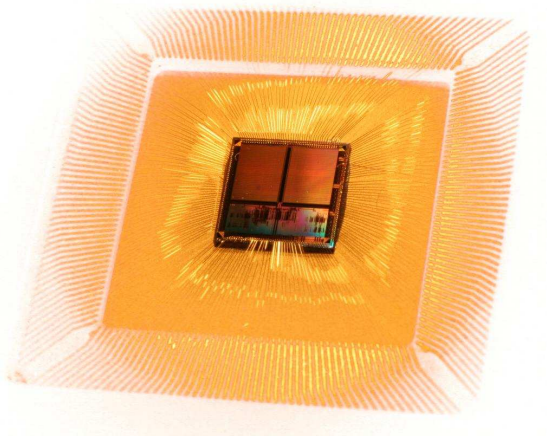


Figure 4.1: The Spikey chip. Wires radiating out of the device connect its pin terminals to the "Recha" board (described below).

Spikey v4 is the latest generation of this device at the time of this writing. It represents a significant improvement over its predecessors due to the elimination of a number of defects that interfere with the intended mode of operation, outlined in its design specification[7, 2]. The chip is produced by a standard 180 nm CMOS[3] process and occupies a square silicon die with an area of 25 mm^2. A single FACETS stage1 version 4 (FHW-1v4)[4] system runs with a speedup factor of about 10^4 compared to real time and provides a network of up to 384 neurons with a maximum number of 2x256x192 synapses, 98304 in total. However, due to a technical problem, only half of these are

[1] Fast Analog Computing with Emergent Transient States
[2] Application-Specific Integrated Circuit
[3] Complementary Metal Oxide Semiconductor
[4] Stage 2 is the wafer-scale device, while version 4 refers to the generation of the chip

available in the currently available version[21]. The synapses of Spikey v4 implement STP[5] and STDP[6], but these capabilities are not needed for this thesis and are not going to be considered further.

In addition to the chip, there are many hardware and software components comprising the system that will be described in the following sections and which are necessary for the seamless operation of the device for modeling purposes.

Neural Network Emulation on Neuromorphic Hardware

Experiments on neuromorphic hardware are fundamentally different from computer simulations, because the artificial network in question undergoes physical changes while it is in operation and is thus more akin to its biological counterparts than to the idealized mathematical models used in software simulators (like those described in section 3.1). While a naive use of the chip is certainly possible due to the availability of a PyNN front-end, there are a number of implications following from the nature of the neuromorphic device that must be explored if it is to be used in any meaningful way.

First of all, there are various limitations imposed by the design specifications of the hardware. Such are the fixed maximum number of neurons and synapses and the resulting topological constraints on the utilized network architectures. In addition, the values of most model parameters are confined to relatively narrow ranges which, however, have been specifically selected by the hardware designers for their biological significance and should therefore be compatible with most modeling scenarios. Synaptic weights, for instance, allow only 16 different settings, since they are represented by 4-bit numbers, but due to the stochastic mapping described in section 4.2.5 and the freedom to define their range through synapse driver calibration (section 4.2.4), they can be used to emulate a well-defined statistical behaviour of larger networks.

A further implication of the physical nature of the device is that a completely deterministic reproduction of simulation results is not possible due to the presence of various types of noise, an intrinsic feature of all analog electronic circuits. The analog components of the chip are also particularly sensitive to parasitic effects such as capacitance or conductance fluctuations under the influence of neighboring circuits that are often impossible to predict in the design phase. Thus they can only be discovered after production, while operating the device.

One of the most important factors determining the non-ideal behavior of the neuromorphic hardware is the photolithographic manufacturing process. Spikey v4 is a VLSI [7] device comprised of thousands of elementary building blocks such as by transistors, capacitors, resitors, diodes, etc. The CMOS production at this scale (180 nm) inevitably introduces imperfections to individual components that are reflected in deviations with a

[5] Short Term Plasticity
[6] Spike-Time Dependent Plasticity
[7] Verly Large Scale Integration

certain magnitude (summarized in [7]) of their characteristics from the intended target values. Higher-level circuits consisting of these elements compound the errors, ultimately leading to systematic variations in the parameters of neurons and synapses, in stark contrast to software neural networks. It should be noted that even though such variations can be partly compensated for by the calibration methods presented below, the circuits that convert the digital representation of the target parameter values in the analog domain are also affected by variations, as are the storage cells holding those values during simulations. Since absolute precision is impossible, a certain amount of stochastic variation of any model parameter will always persist across populations of neurons and synaptic drivers.

While severe process defects may lead to the loss of functionality of the digital circuitry, a great strength of the analog design is that it is resilient against such occurrences. Individual neurons or synapses on an affected chip can be ignored while the rest of the network can still be used for simulation purposes. This significantly increases the *yield* ratio of the production process.

Finally, the readout channels that allow membrane potentials of individual neurons to be monitored and recorded using an oscilloscope are also subject to systematic offsets due to impedance mismatches and amplifier process variations that must be accounted for. In contrast, the acquisition of spike events is generally much more precise (with a temporal resolution of about 0.3 ns on the hardware time scale[7]), because it is handled by the digital part of the chip. In the course of development of the calibration routines, it has been discovered by the author that spike times are not completely synchronized with membrane trace recordings. The difference typically has a magnitude of $\Delta t \approx 1$ ms (in simulated biological time) and varies from neuron to neuron.

Given all of the above constraints and the knowledge of process variations in the FACETS stage1 hardware, there are two possible ways of enabling scientifically meaningful experiments on the system: a) Devise (or modify existing) models that are robust against the noises and parameter distributions present in the system, and b) Develop calibration routines that attempt to minimize these variations within acceptable bounds.

While there exist some neural network models that are highly resilient against large variations in their parameters, often motivated by empirical evidence that biological neurons operate in a very noisy environment, it should be kept in mind that the leaky integrate and fire model on which the FHW-1v4 system is based is a very idealized representation of the properties of biological systems which possess many self-regulation mechanisms for their parameters. Moreover, even such models may not function properly when faced with parameter deviations out of a certain range. The subject of this chapter, therefore, is the development of realiable methods for compensating process variations using the facilities provided by Spikey v4.

Compensation of Process Variations

One of the advantages of the FACETS stage1 architecture is the ability to adjust nearly all modeling parameters and also to implement almost arbitrary network topologies, thus bringing it close to a universal simulation platform for conductance-based LIF neural networks. This configurability is the cornerstone of all calibration procedures, since they are based on the application of certain offsets when setting target parameter values. This approach depends mostly on the granularity with which the corresponding parameters can be set for populations of neurons or blocks of synaptic drivers and leads to four classes of parameters according to this criterion:

- **Non-adjustable** parameters. They are derived from the physical characteristics of the object. A good example are the capacitors representing the neuron membrane. The parameter C_{m} for an individual neuron is directly proportional to its area, which is obviously affected by manufacturing imprecision.
- **Globally-adjustable** parameters. There are only a few, such as the rising time constant for the synaptic conductance course and the offset of the voltage ramp that is translated to a conductance course at the synaptic drivers (described in more detail later).
- **Shared** parameters. The neurons on Spikey are divided into two populations. In each population the odd or even-indexed neurons share their voltage parameters such as their threshold potentials and resting potentials. Therefore there are 4 independent sets of neurons that can have different neuron voltage parameter values. These constraints are mostly based on design decisions for the layout of the chip, based on the optimal utilization of the die area.
- **Individually adjustable** parameters can be different for all neurons or synaptic drivers and include the membrane leakage conductance and the parameters governing the shape of the synaptic conductance courses.

If any calibration routines are to be developed by the user of the FACETS stage1 neuromorphic hardware, a working knowledge of its architecture and operation principles is required. The next sections provide a more detailed description of the FHW-1v4 system and outlines the software workflow for configuration and simulation on the chip.

4.1 FACETS stage1 Neuromorphic Platform

The ability to use FHW-1v4 for neuroscientific studies is made possible by the supporting infrastructure of the Spikey v4 VLSI device. It consists of a number of additional hardware and software components that effectively shield the user from the low-level implementation details of the neuromorphic hardware. Figure 4.2 provides an overview of the system.

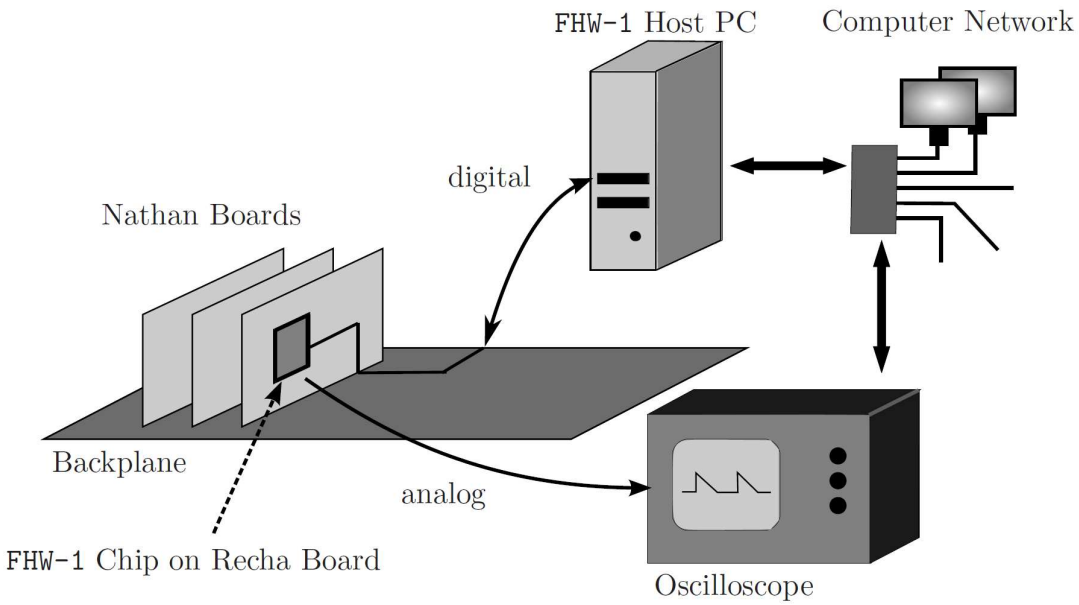


Figure 4.2: An overview of the FHW-1v4 system (taken from [7])

In a Multi-Chip environment, the *backplane* is a carrier PCB[8] which provides 16 slots for individual FHW-1v4 devices. These can be operated individually, or as a unit (multi-chip mode). The backplane is connected to a Linux host PC (simulation server) that contains all of the software needed for the communication with the system via a Gigabit Ethernet interface. PyNN simulations are run locally on the server, to which end users can connect via a standard TCP/IP network. The same network is also used to control an advanced oscilloscope that is used as an acquisition device for membrane potential data. The oscilloscope is connected directly to the *ibtest* pin of one of the chips, since only one neuron can be monitored at a time. Spike event data, on the other hand, are relayed digitally by the backplane. All Spikey chips are bonded to *Recha* boards and are protected by clear plexiglas covers. These in turn are mounted on the larger *Nathan* PCBs which can be added to the available slots on the backplane.

The individual components of FHW-1v4 are annotated on the photograph shown in figure 4.3. Backplanes used in production are placed in special-purpose cases that rest in 19-inch racks, so that the system is kept in a relatively protected environment, due to the temperature sensitivity of the analog electronics [2].

4.1.1 FHW-1v4 Supporting Hardware

As is the case with other VLSI devices (such as computer processors), an isolated Spikey chip can not operate by itself and is supplied with the necessary currents and voltage signals by its supporting hardware. Some of these reach back as far as the backplane on

[8] Printed Circuit Board

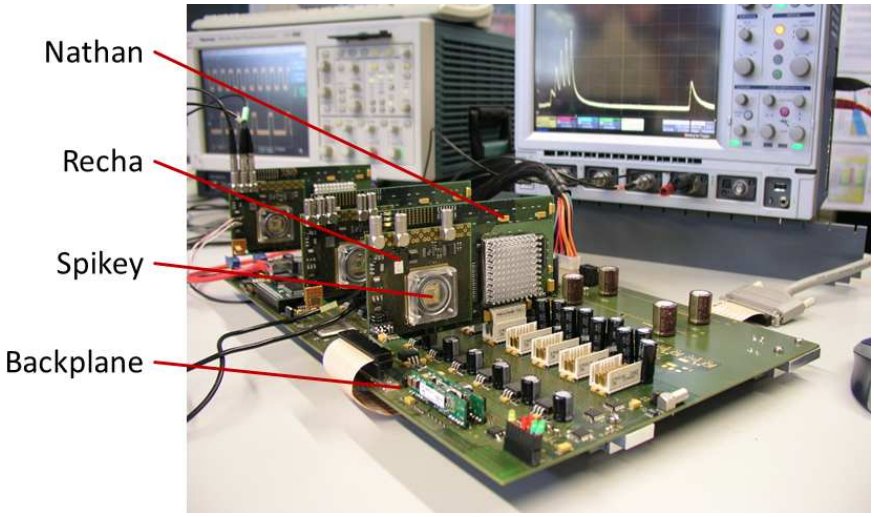


Figure 4.3: Experimental setup showing the FHW-1v4 system

which the whole system is mounted. The relation among these components is considered next.

Backplane

A major task of the backplane is to serve as a gateway between the FHW-1v4 hardware and the simulation environment on the PC host connected to it. In addition, it provides power and physical support to the devices, as well as the clock signal controlling them. The necessary logic is programmed in an FPGA[9] module on the board, representing a flexible solution when additional functionality, such as multi-chip operation, is needed.

Nathan Board

The most important components of the Nathan board are an FPGA and a DDR-SRAM module, since they are necessary for its function as a controller of the ANN on the Spikey chip. It also carries the Recha board which is mounted in a custom socket, designed specifically for an ANN ASIC[21]. The FPGA contains the modules *spikey-sei* and *spikey-control* enabling the real-time access to the chip using the *SlowControl* (SC) protocol [21]. One rather important feature of the FPGA on the Nathan board is the *playback memory* that is used to send event packets with a controlled delay between them. This enables the stimulation of the ANN with input spike trains with well-defined interspike intervals. The input data are externally generated and stored in the RAM module for the duration of the simulation. In addition, the Nathan board contains digital-to-analog converters (DAC) that generate some of the control voltages for the chip as well as the readout circuit of the analog-to-digital-converter (ADC) on the Recha board, and a temperature control mechanism [21].

[9] Field-Programmable Gate Array

Recha Board

The Recha board carries the Spikey ASIC (figure 4.3) and supplies many of the low-level control signals for the chip. Some of these are also global parameters such as the reference current I_{refdac} (controlled by the Nathan) and the voltages V_{rest}, V_{start} [21] controlling the voltage ramp generated by the synaptic drivers (described below). All 9 analog voltage output pins of Spikey are connected to terminals on the top of the Recha board, so that they can be attached to an oscilloscope if a *lemo jack* is soldered at the corresponding place on the PCB. This is often not necessary for all 9 outputs, because each of them can be selected by a multiplexer on the Recha board whose output is available at such a jack. In addition, its output can be directed to a four-channel 10-bit ADC on the Recha board so that the values of the analog output pins (parameter voltages, voltage traces, etc) can be transmitted digitally back to the controlling software on the PC (the rapidly evolving membrane traces are read by oscilloscope, however).

4.1.2 The Spikey Chip

As it can be seen in figure 4.4, synaptic circuits occupy most of the die area on a Spikey ASIC, since they represent the connection matrix for the ANN. It consists of two equal network blocks that are located symmetrically about the vertical axis of the chip. The innermost strip on each network block is comprised of 256 horizontally oriented *synaptic drivers* that are instrumental in controlling the conductance courses impinging on the neurons and ultimately the resulting post-synaptic potentials (PSPs). The bottom row of each block is taken up by 192 artificial neurons whose both ingoing and outgoing connections with other neurons are routed through the corresponding synapse array.

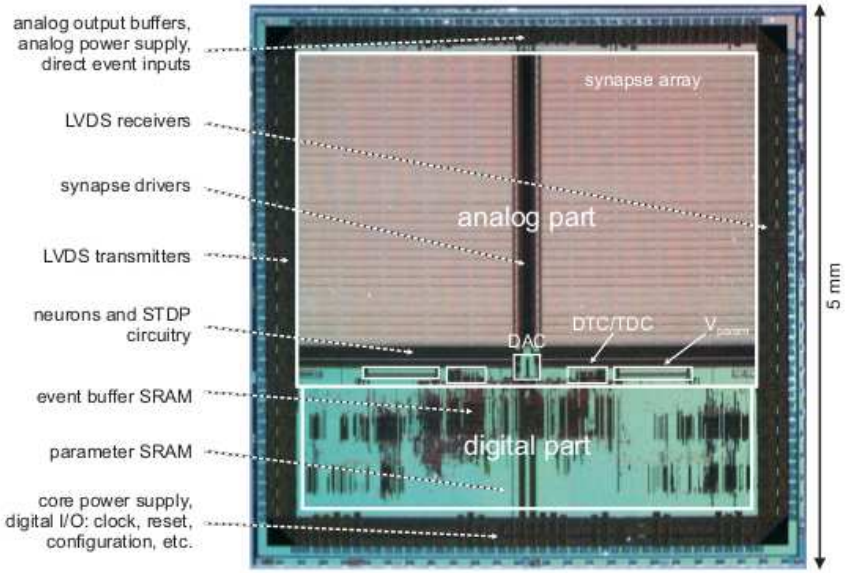


Figure 4.4: Spikey micro photograph

Except for the wires necessary for the connectivity matrix, the area of each synapse array is also occupied by 256 x 192 *synapse nodes* representing physical synapses, each containing a 4-bit static weight memory. A vertical column of such nodes controls the total synaptic conductance at each neuron. The exact mechanism is highly relevant to the calibration of the synapse drivers and is described in more detail below.

At this point it should be mentioned that the close proximity of the wires carrying controlling signals for conductance courses and those feeding back spike events inevitably leads to the occurrence of parasitic effects when neighboring neurons are actively stimulated.

Individual circuits in the ANN are controlled by either currents or voltages. Many of these controlling signals correspond to (directly or indirectly) model parameters used in neuroscientific experiments and therefore have to be stored on-chip in the analog domain during the configuration stage of a simulation. This has lead to the development of a custom current memory cell that functions as a voltage controlled current source/sink[20]. Since that voltage is stored in a capacitor within the cell and continuously diminishes due to leakage currents, it requires constant refreshing that is accomplished by the digital-to-analog converter shown on figure 4.4. There are 2967 such parameters, all connected to the output of the DAC.

Each analog voltage parameter is likewise generated by a circuit containing a current memory cell (voltage generator) that converts the current to a voltage via a 10 kΩ poly-silicon resistor. In this way one DAC can refresh all parameters regardless of their type, thus saving costly die area. Two voltage generator blocks consisting of 24 cells each are found on the interface between the digital and the analog part, as indicated on figure 4.4.

All values used by the DAC are stored in a static *parameter RAM* module located in the digital part of the chip that is controlled externally by the Nathan FPGA. The allowed analog parameter currents range within 0 and 2.5 μA while the parameter voltages can be set between 0 V and 1.8 V[21]. The resolution of the 10-bit DAC divides those intervals in 1024 discrete adjustable values with an increment of $\frac{1}{1024}\times$ upper bound.

Neuron Model and its Physical Implementation

The artificial neurons in FHW-1v4 emulate a leaky integrate-and-fire neuron model with conductance-based synapses. It is defined by equation

$$-C_\mathrm{m}\frac{dV_\mathrm{m}(t)}{dt} = g_\mathrm{l}(V_\mathrm{m}(t) - E_\mathrm{l}) + \sum_j p_j(t)g_j(t)(V_\mathrm{m}(t) - E_\mathrm{e}) + \sum_k p_k(t)g_k(t)(V_\mathrm{m}(t) - E_\mathrm{i}) \tag{4.1}$$

with the additional condition

$$V_\mathrm{m}(t) = \begin{cases} V_\mathrm{reset}(t), & \text{if } V_\mathrm{m} > V_\mathrm{thresh} \\ V_\mathrm{m}(t), & \text{otherwise} \end{cases} \tag{4.2}$$

since action potentials are not modeled explicitly. A summary of the role of each quantity figuring in equation 4.1 follows:

- $V_{\mathrm{m}}(t)$ - Time-evolving neuron membrane potential
- C_{m} - Membrane capacitance
- g_{l} - Fixed membrane leakage conductance
- E_{l} - Membrane leakage reversal potential
- $p_{j,k}(t)$ - Conductance courses for the excitatory and inhibitory synapses
- $g_{j,k}(t)$ - Multiplicative factor describing the evolution of synaptic weights in time, due to synaptic plasticity
- E_{e}, E_{i} - Excitatory and inhibitory synaptic reversal potentials

Their physical implementation in the hardware is schematically depicted in figure 4.5 and is discussed in the next two sections.

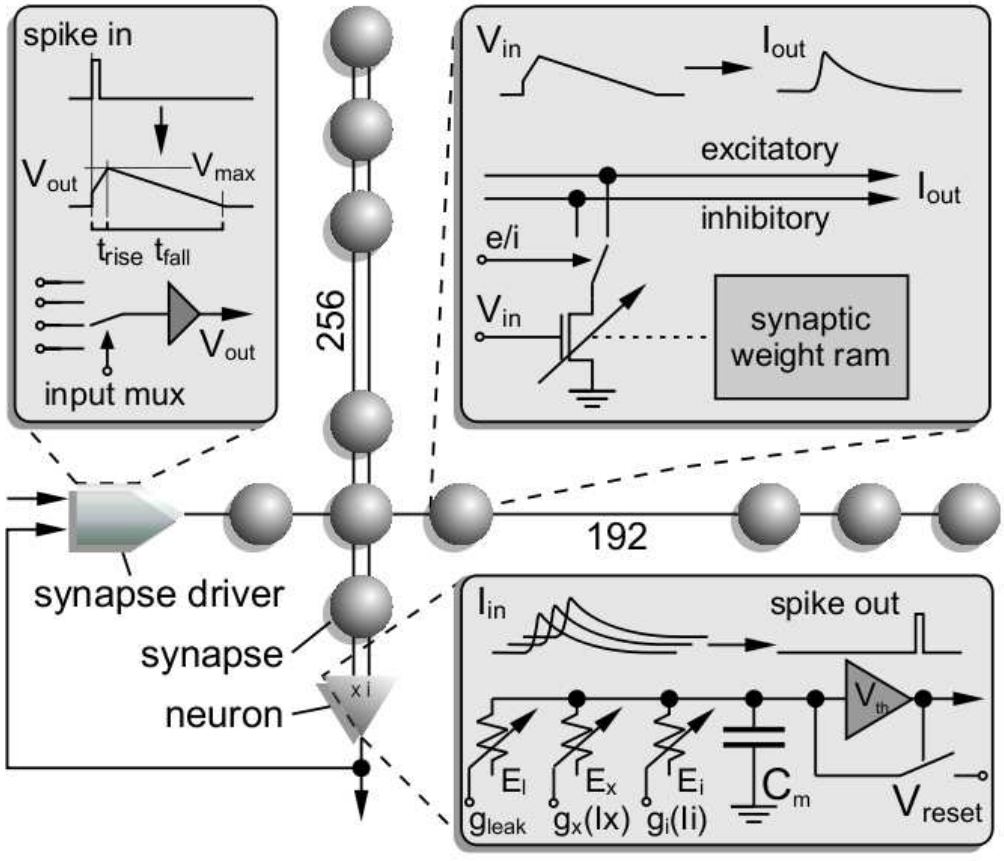


Figure 4.5: Physical implementation of neurons and synapses

A neuron in Spikey is a capacitance C_m, the value of which depends on its surface area. The size of the area is chosen in conjunction with the 10^4 hardware speedup to produce the correct low-pass filtering properties of the neural membrane. The inevitable systematic variations caused by the production process require some mechanism enabling the user to set the value of the membrane time constant $\tau_\mathrm{m} = C_\mathrm{m}/g_\mathrm{l}$, a neuron parameter of crucial importance. This task can be accomplished by changing the membrane *leakage conductance* g_l which is controlled by the individually adjustable current parameter *ileak_cond* (the relation between *ileak_cond* and g_l is not linear and has to be determined empirically for each neuron).

In the absence of external stimuli, the membrane potential V_m tends to reach the *resting potential* V_rest ($\equiv$ *membrane leakage reversal potential* E_l). Its value is controlled by a voltage generator, but V_rest represents a shared parameter for all neurons of the same parity in a neuron block. The same applies to the *threshold potential* V_thresh, the *reset potential* V_reset, and the *inhibitory reversal potential* E_i. The *excitatory reversal potential* E_e is fixed at 0 mV in the biological voltage domain. The calibration of the voltage generators (section 4.2.1) ensures a linear relationship between the biologically relevant parameter and the actual potential at the neuron. Of course, systematic variations also affect these voltages at the individual neuron (e.g. variation in the 10 $k\Omega$ resistances), but the shared nature of the parameters prevents any possibility for their relative adjustment. Still, the method described in section 4.2.2 mitigates the effect of these variations.

Another important component of the hardware neuron is the threshold comparator V_th(figure 4.5), which provides the signal for lowering the membrane potential V_m to the reset level V_reset when it exceeds the threshold potential V_thresh. As it is necessarily of limited precision, a membrane trace of a regularly spiking neuron (figure 4.6) shows that variations also take place in the time domain.

Physical Synapses

In a biological setting, the arrival of an action potential at the synaptic terminal of the presynaptic neuron triggers the release of neurotransmitter molecules in the synaptic cleft by exocytosis. The fusion of each synaptic vesicle with the cellular membrane is a complex process that occurs with a certain probability depending on time. Neurotransmitter molecules that reach the surface of the postsynaptic neuron bind to ligand-gated ion channels whose opening changes the conductivity of the corresponding membrane area, thus leading to the generation of a PSP. This whole process can be described in more abstract terms by a conductance course $p_\mathrm{s}(t)$ that is unique for each synapse. It is most often modeled as an exponential decay or an *alpha function*[33]. In equation 4.1 the conductance course is scaled by an additional factor

$$g_\mathrm{s}(t) = \omega_\mathrm{s}(t) g_\mathrm{s}^\mathrm{max}(t) \tag{4.3}$$

that describes changes in the synaptic strength due to the plasticity mechanisms (STP,

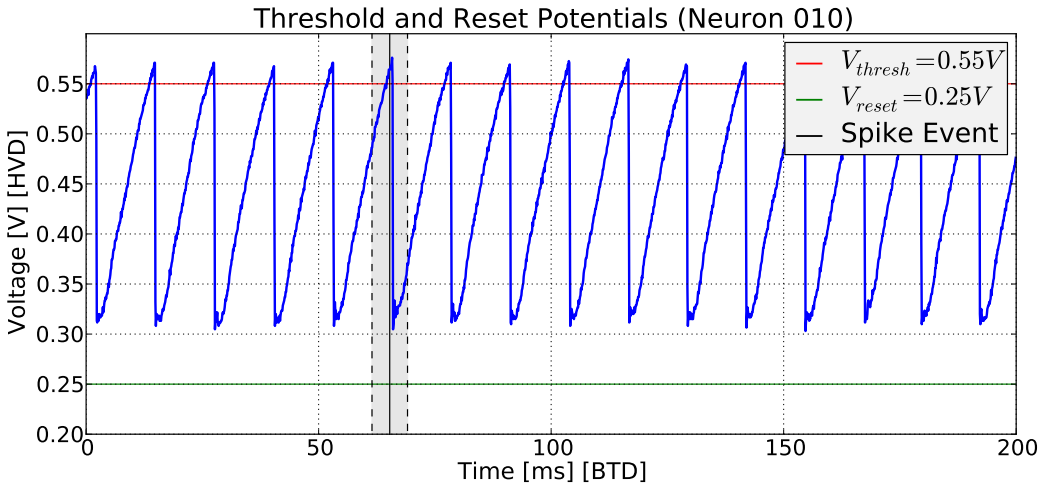


Figure 4.6: Measurement of the threshold and reset potentials. The target values in BVD are $V_{\text{thresh}} = -55$ mV and $V_{\text{reset}} = -80$ mV. Based on the recorded membrane trace and the spike train produced by the neuron, it is possible to find regions around each spike event, whose maximum and minimum values are the threshold and reset levels, respectively.

STDP) implemented by the chip. Since these are of no consequence for the following work, it should be assumed that $g_{\text{s}} = \omega_{\text{s}} = const$ for all synapses from now on.

The generation of post-synaptic potentials on Spikey is a three-step process that is controlled by several analog voltage and current parameters. FHW-1v4 does not provide any direct means of measuring the synaptic conductances, therefore parameters governing the corresponding conductance courses have to be inferred from membrane trace potentials and spike event data. An in-depth understanding of the process is required for the calibration of the time constants and heights of these conductance courses.

Figure 4.7 depicts a schematic view of the right network block on the chip. Its 192 neurons are situated on the bottom row of the synapse array while the 256 synaptic drivers occupy its left edge. At each row there are 192 synapse nodes that control two separate excitatory and inhibitory lines leading to the corresponding neuron. The great number (256 x 192) of these synapse nodes means that they occupy much of the die area and necessitates the relegation of some of the synapse functionality to the synaptic drivers.

Conductance Course

The generation of the conductance course starts by the arrival of a digital pulse at a synaptic driver. This spike event may come from a neuron in the network block, a neuron from the other network block, or from the playback memory on the Nathan

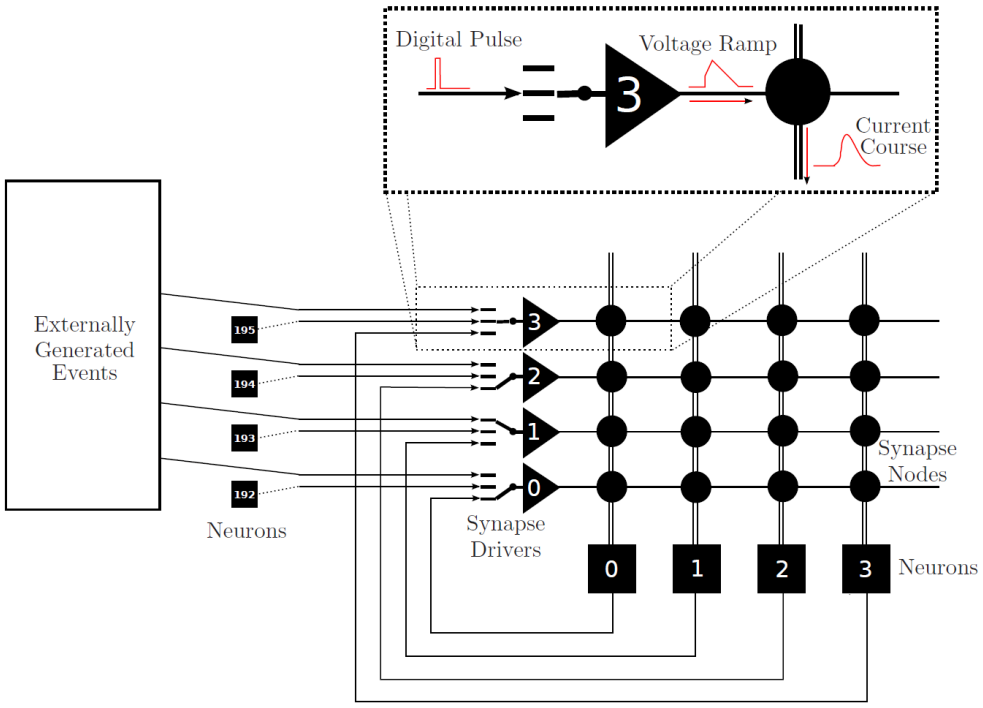


Figure 4.7: A schematic depiction of the synapse array. The inset depicts the generation of the conductance courses.

where external spike input is stored and fed into the chip. In response, the synaptic driver generates a voltage ramp that is parametrized by one voltage and three currents, as shown on figure 4.8 and in table 4.1:

Parameter	Name	Configurability	Value Range
V_{start}	`v_start`	global	$[0, 1.8]$ V
I_{amp}	`drviout`	individual	$[0, 2.5]$ μA
I_{rise}	`drvirise`	individual	$[0, 2.5]$ μA
I_{fall}	`drvifall`	individual	$[0, 2.5]$ μA

Table 4.1: Readout and input connection parameters

This *pre-synaptic* voltage signal[20] propagates horizontally across a synapse row and is transformed by the synaptic nodes into the double-exponential *post-synaptic* current shown in figure 4.9. Since the positive slope of the voltage ramp is converted to an exponential, the resulting PSPs for longer rising times t_{rise} would distort the PSP shape significantly. Hence, the V_{start} parameter that controls the onset of the current can be used to produce more realistic (faster rising) conductance courses. The exponential decay is generated by a current sink consisting of 15 n-type metal oxide transistors[20]. They are controlled by a 4-bit static weight memory located within the synapse in a way

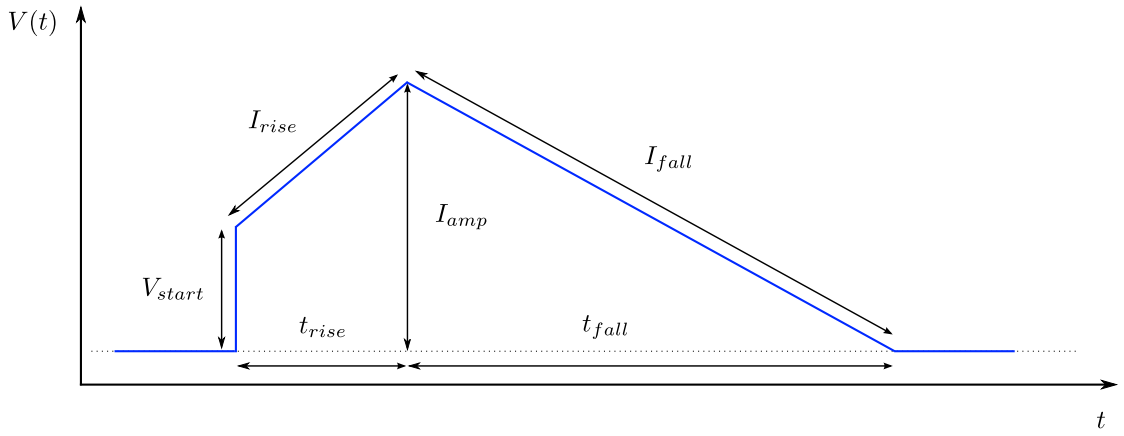


Figure 4.8: Voltage ramp generated by the synaptic driver

that the number of activated transistors depends on the weight of the connection.

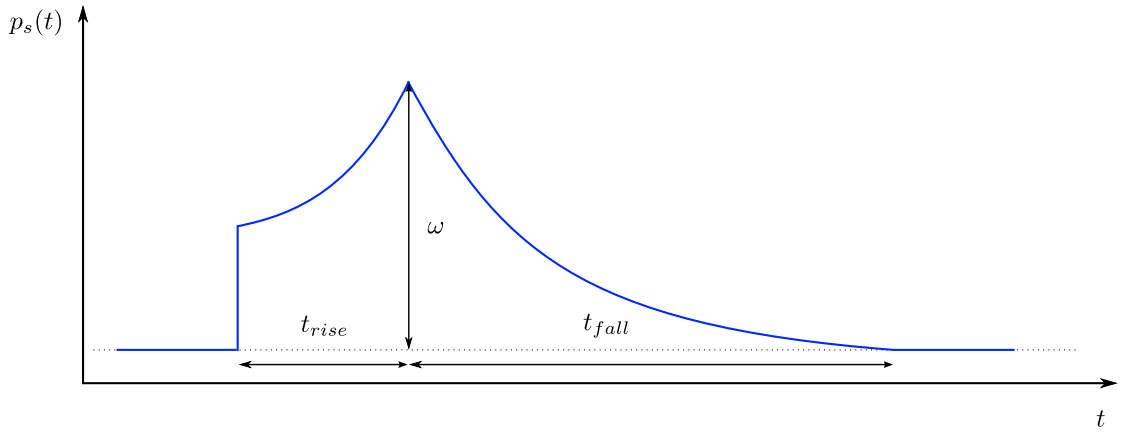


Figure 4.9: Conductance course

As it can be seen in figure 4.7, the synaptic nodes are connected to the neurons via two separate wires for the excitatory and the inhibitory PSPs. It must be mentioned that a control line connected to the synapse driver sets the complete row to add its current to either wire, thus reserving it for only one type of post-synaptic signal[20].

All currents generated by synapse nodes belonging to the same column are summed up in the corresponding excitatory or inhibitory line and arrive at an operational *transconductance* amplifier which belongs to the neuron circuit and which changes the conductance between its membrane potential V_{m} and the corresponding synaptic reversal potential $E_{\mathrm{x}}/E_{\mathrm{i}}$. When the onset of the controlling current is set high (using V_{start}), the resulting conductance course is very close to the standard exponential decay kernel used in neuroscientific simulations. The falling voltage time period t_{fall}, controlled by the

`drvifall` current, consequently determines the synaptic time constant τ_{syn}.

4.1.3 Software Stack

An important advantage of the FACETS stage1 platform is that the development of neural network models running on FHW-1v4 relies on the same tools (PyNN, SciPy, etc.) that are used to construct software simulations. Since the resulting code is compatible with a number of different platforms (see figure 4.10), this opens up the possibility to compare the performance of the hardware with expected results delivered by other means. Such a seamless operation of the FHW-1v4 hardware is only possible due to the multilayer architecture of its software stack, where each layer is concerned with a particular level of detail in controlling the configuration and execution of the simulation on the chip. The calibration of Spikey utilizes these components, therefore it is necessary to briefly discuss the architecture of the software stack.

FHW-1v4 Neuroscientific Modeling Software

The top-level programming interface that is utilized for neuroscientific modeling in conjunction with Spikey v4 is the PyNN modeling layer which has already been introduced in section 3.1. It represents the top level of the software stack and communicates with a custom-designed *stage1* backend called *pynnhw* that relays the abstract data structures and algorithms used in an artificial neural network model to lower-level components of the controlling software which execute the proper configuration and simulation commands on the hardware. Figure 4.10 shows the interrelations among many of the PyNN components and, more specifically, the pynnhw interface to the chip.

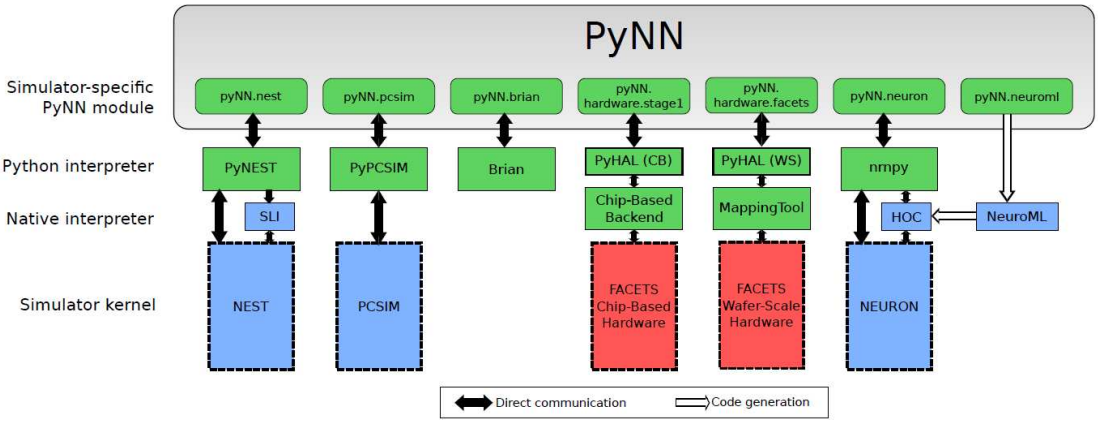


Figure 4.10: The hierarchical architecture of PyNN and its back-ends.

It is necessary to point out that PyNN does not automatically take into consideration *all* hardware constraints of the FHW-1v4 system and its user must be aware of possible problems when the neural network model is improperly defined. For instance, it is impossible to set different values of an electric potential parameter ($E_{\text{l}}, E_{\text{i}}, V_{\text{tresh}}, V_{\text{reset}}$) for

different neurons with the same index parity belonging to a network block, because they use the same voltage generator. Such an attempt would, however, fail silently, so that the last value attempted to be set overrides the previous one for the whole population. Moreover, there are possible topological constraints when using the full capabilities of the ANN (especially in a multi-chip setup with external input), thus precluding the naive use of the chip, however easy it is to perform simulations on it.

PyHAL and Hardware Configuration

A neural network simulation in PyNN is a stereotypical process, typically consisting of a configuration, simulation, and a data acquisition phase. Once the corresponding Python script is executed on the Linux simulation server (host PC to FHW-1v4), its pynnhw back-end uses the low-level routines of the PyHAL layer to configure the ANN on the chip by sending commands to the Nathan. It is a wrapper around a C++ component that contains a model of Spikey's ANN, a configuration, and a communication module (see figure 4.11). Most notably, data arrays comprising the calibration data and some associated functions are also present in the PyHAL layer.

PyHAL does not, however, communicate with the hardware directly, but through the mediation of the *Darkwing server*(*dwserver*) daemon which runs on the host PC and relays the commands using the SC protocol. The need for the dwserver arises out of an earlier version of the FHW-1v4 system that used a SCSI cable and a PCI card as the communication channel between simulation server and backplane[21]. Moreover, since one instance of dwserver manages a backplane, several users with different allotted chips on that backplane may use the FHW-1v4 system simultaneously.

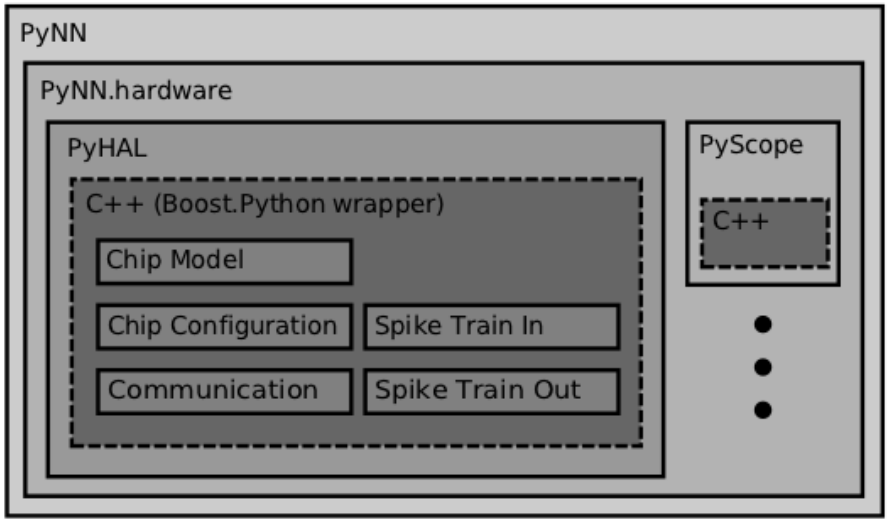


Figure 4.11: Software stack necessary for the operation of the FHW-1v4 system (from [2]).

Spike output data generated by an ANN simulation is gathered digitally over the same

channel. Membrane potential traces, on the other hand, are not mentioned by the design specification of the chip and have to be recorded in a more circuitous way. The network can be configured to multiplex the membrane potential of a neuron to the *ibtest* pin mentioned in section 4.1.1. If the corresponding lemo jack at the Recha board is connected to an oscilloscope, the latter can be used for data acquisition. Sophisticated oscilloscopes, such as the ones used in the FHW-1v4 platform are network-enabled devices. Consequently, pynnhw contains a dedicated module that fetches neuron membrane potential recordings over TCP/IP. It should be emphasized that this acquisition mechanism is completely independent of the FHW-1v4 supporting hardware, except for the PC host.

4.2 Calibration Routines

The parameters describing a neural network model refer to biologically relevant quantities whose values are usually inspired by empirical neuroscientific measurements. They obviously have to be translated to the adjustable voltages and currents on the chip that control the corresponding properties of its physical ANN network. Since the FHW-1v4 neuromorphic hardware operates on a time scale far different from biochemical processes *in vivo* and is by construction very dissimilar in terms of on-chip parameters to biological neurons, it is to be expected that the ranges of those values on the chip would also fall in a different order of magnitude. Moreover, not all of the modeling parameters may have their analogous counterparts on the chip and even if they do, they could be described by a quantity of a different type (e.g. synaptic time constants are controlled by a voltage, membrane leakage conductance by a current). In addition, while some of these pairs of parameters are linearly related (neuron potentials $E_\mathrm{l}, V_\mathrm{reset}, E_\mathrm{i}, V_\mathrm{thresh}$), the interdependence of others reflects the architecture of the underlying electronic circuits.

On one hand this state of affairs means that the mapping of biological parameters on the hardware is a non-trivial task, but on the other it allows for the free interpretation of hardware quantities in the biological domain. Table 4.2 gives an overview of the available configuration parameters:

The following sections employ a special notation to distinguish clearly between the biological and the hardware quantities by utilizing the descriptors [BVD] for the biological voltage domain, [BTD] for biological time domain, [HVD] for the hardware voltage domain, and [HTD] for the hardware time domain.

The mapping of model parameters to the hardware domain is described extensively in[7] and is performed by the PyHAL layer. After the configuration of the ANN their digital representation is held in the parameter SRAM on the chip which is then used to constantly refresh the analog voltage generators and current memory cells that ultimately control the physical properties of the network. Due to the imperfections in the creation of elementary electronic components, which is introduced by the manufacturing process, the deviations from their expected electrical characteristics leads to variations in the analog values of the controlling voltages and currents, even when the correct mapping routine is used. The next section outlines how such effects can be compensated for.

Neuron Parameter	Symbol	Configurability	Value Range
Membrane Capacitance	C_{m}	Fixed	0.2 nF
Resting Potential	V_{rest}	Shared	$[0, 1.8]$ V
Threshold Potential	V_{thresh}	Shared	$[0, 2.5]$ V
Reset Potential	V_{reset}	Shared	$[0, 1.8]$ V
Inhibitory reversal potential	E_{i}	Shared	$[0, 1.8]$ V
Leakage Conductance	g_{l}	Individual	$[0, 2.5]$ μA
Synaptic Parameter			
Starting Voltage	V_{start}	Global	$[0, 1.8]$ V
Rise Time	t_{rise}	Global	$[0, 2.5]$ μA
Voltage Ramp Maximum	V_{max}	Individual	$[0, 2.5]$ μA
Fall Time	t_{fall}	Individual	$[0, 2.5]$ μA

Table 4.2: Neuron parameters

Calibration Process

When a particular value of a biological parameter is used in a model, the corresponding analog control signal on the chip changes the characteristics of a neuron or a synapse in the ANN. In some cases this physical quantity can not be measured directly, but has to be determined from the available recorded data for the simulation. This necessitates the development of methods for the measurement of the variables presented in table 4.2 from neuron membrane potential data and spike output. While some of these can be inferred with sufficient accuracy, the values of others, more specifically the synaptic parameters, have to be estimated from secondary effects, such as the shape of the PSPs. Since the magnitude of the control and process variables are both in the BVD and BTD, it should be possible to find a transfer function that can be used to tune the expected value of the parameter by offsetting it with the application of an *offset* voltage or a *bias* current. This is indeed the method behind most calibration routines presented in this chapter, but the determination of the transfer function proves all but trivial for the synaptic conductance course, an issue addressed in section 4.2.4. All bias current and voltage values generated by the calibration are stored in data arrays in the PyHAL module, which enables the complete separation of the calibration work from the FHW-1v4 software stack, thus enabling the user of the platform to develop task-specific calibration methods when needed. An overview of the ones used in this thesis are portrayed in figure 4.12 and are partly based on the procedures presented in[7].

Most basic of all is the calibration of the voltage generators on the chip, some of which are used for other tasks besides governing neuron potential parameters in the ANN. Due to process variations, the target values that the DAC writes to the memory cells are not precisely reproduced as analog voltages. Since the relation is linear, however, a simple measurement of the linear range for each cell is sufficient to find the slope and offset

needed for the calibration.

Once the actual voltages are known to be those intended by the parameter mapping process, it is necessary to calibrate the dynamic range of each neuron individually (described in section 4.2.2). This is crucial for all further measurements that rely on the utilization of membrane potential traces in the biological domain, since it determines the translation HVD $\rightarrow$ BVD by minimizing possible error sources.

The membrane time constant τ_{m} determines the low-pass filtering properties of the neural membrane. This is a physical quantity that can not be set directly, in contrast to software simulations. Because of systematic variations in the fixed membrane capacity C_{m} , the only calibration facility is provided by the membrane leakage conductance g_{l}. It is controlled by a current and has a non-linear transfer function. The importance of τ_{m} is reflected in the fact that g_{l} is individually adjustable for each neuron.

Synaptic conductance courses are governed by the voltage ramps described in section 4.1.2. Such ramps, as well as the conductances, are not directly measurable in Spikey. Their time constants and magnitudes have to be determined from the PSPs which are low-pass filtered conductances (hence the priority of the τ_{m} calibration). In addition, the network has to be in a *high conductance state*, the necessity for which is given in section 4.2.4 and the transfer functions governing the mapping of the parameters are non-linear.

Due to their different *physical* dynamic range, the impact of a conductance course with the same characteristics on different neurons would produce different PSPs, a problem that is not found in either software simulations or biological cells. The only possibility to compensate for this effect is to provide a multiplicative constant for each column in a synaptic array that scales the aforementioned impact in accordance with the dynamic range of neuron.

These calibration steps are now going to be considered more closely.

4.2.1 Voltage Generators

The 10 kΩ poly-silicon resistors used to convert memory cell currents to voltages (section 4.1.2) have mismatch rates of up to 30%, thus requiring the calibration of all 46 voltage generators. Spikey v4 can be programmed to connect any of these *vouts* to its *ibtest* analog readout pin, which in turn can be read from the 10-bit ADC on the Recha board (section 4.1.1), allowing for the digital transmission of the data back to the calibration program (This stands in contrast to membrane potential data which must be read by oscilloscope from the *ibtest* pin. Due to reasons clarified in the next section, the data delivered by the ADC can not be used to calibrate out the systematic variations in the neuron voltage potentials).

An equidistant sampling over the whole range of the 10-bit DAC for two voltage parameters is shown in figure 4.13. An automatic plateau detection algorithm by E.Müller[32]

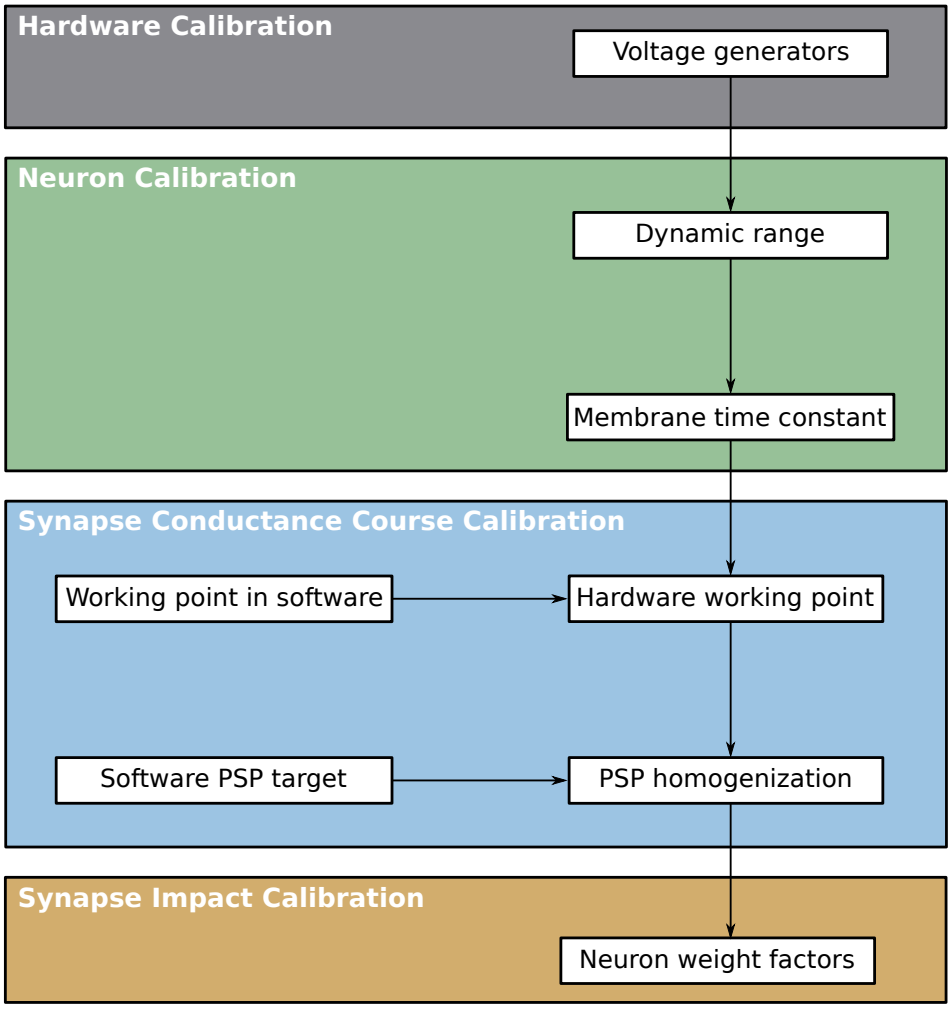


Figure 4.12: Calibration process overview

determines the linear range of the transfer function and finds the slope and offset parameters needed for the proper programming of the voltage generator. Unlike the rest of the calibration routines, it is implemented in C++ and is built into the HAL layer.

4.2.2 Dynamic Range

The mapping of the neuron voltage parameters ($E_\mathrm{l}, V_\mathrm{thresh}, V_\mathrm{reset}, E_\mathrm{i}$, and E_x) to the HVD defines the dynamic range of the physical neuron. It is determined by several factors, including the full range of the voltage generators ($[0, 1.8]$ V), a design-inherent upper limit of 1.2 V, and the choice of lower and upper neuron potentials spanning the dynamic range ($[V_\mathrm{reset}$ or E_i, $E_\mathrm{x}]$). The method used by the FHW-1v4 software stack for parameter translation is described in[7].

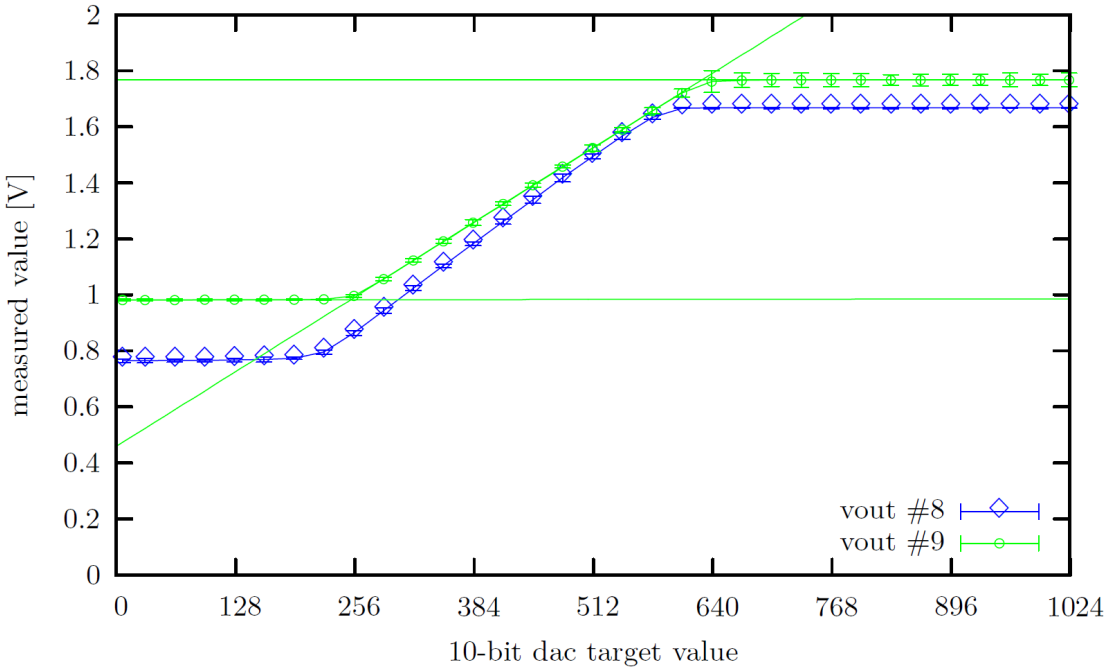
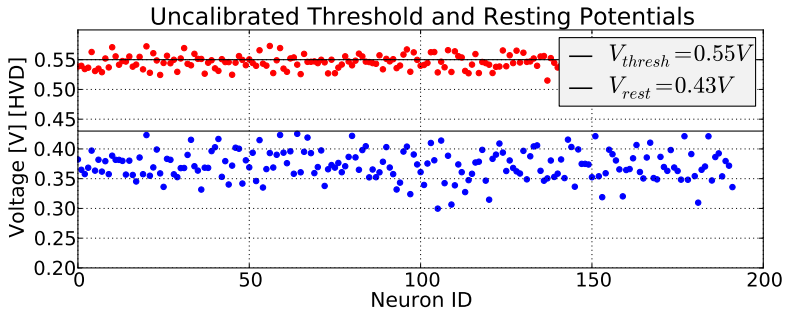


Figure 4.13: Voltage generator calibration (from [32])

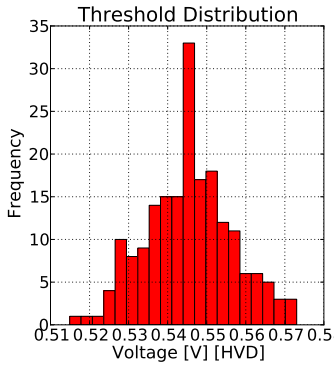
When the biologically plausible resting and threshold potentials of $E_\mathrm{l} =: V_\mathrm{rest} = -65$ mV and $V_\mathrm{thresh} = -55$ mV respectively, are set for a population of 192 neurons that occupies the whole right network block of the ANN, the corresponding values at the voltage generators, measured by the ADC readout channel are 0.86 HVD and 1.1 HVD. These are accurate readings of the voltage generators and can be used to find systematic variations in the readout of the membrane traces by oscilloscope, since both parameters can be easily determined from the membrane potentials. Taking into account the impedance at the oscilloscope input which effectively halves the measured voltage (leading to 0.43 V and 0.55 V respectively), it should be expected that any systematic readout error ΔV at a chosen neuron has the same magnitude for its threshold and resting potentials. Their distances from the target values, portrayed in figure 4.14 a) which displays the result of such a measurement, however, behave contrary to this prediction, and show weak correlation.

Obviously, the voltage at a parameter memory cell is not directly connected to the capacitance C_m, but is instead inputted by a rather different process which is susceptible to manufacturing mismatches. Indeed, the value of each voltage parameter is written to the neuron membrane by a separate *operational amplifier*, as shown in figure 4.15, thus leading to a *write error* $\Delta Write$ which is assumed to be normally distributed (J.Schemmel, personal communication).

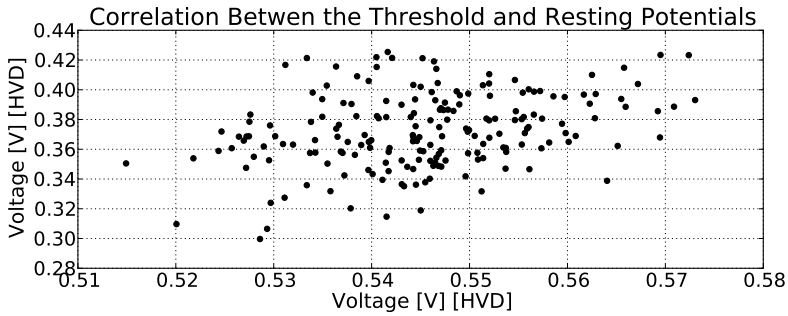
Similarly, there is an operational amplifier dedicated to the reading membrane traces, thus introducing an uncertainty ΔRead for their values.



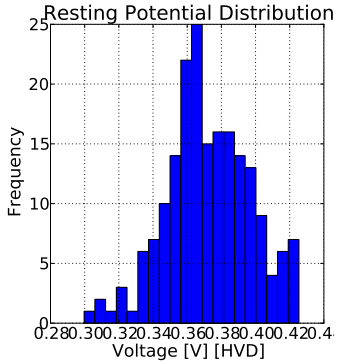
(a) Threshold and resting potentials



(b) Threshold distribution



(c) Correlation between the resting potential and the threshold potential



(d) Resting potential distribution

Figure 4.14: Measurement of threshold and resting potentials of all neurons on an uncalibrated device.

A further source of error is contributed by the mechanism used for reading membrane traces by oscilloscope. Four analog readout lines per network block (see figure 4.16) are utilized to monitor the membrane potential of neurons with index $i\%4 + j$ where $j \in [0, 3]$ is the number of the line. The signal is then either multiplexed to the *test* pin or is available directly over one of the 8 analog outputs at the Recha board if a lemo jack is soldered to the corresponding terminal (pynnhw controls only the multiplexer, so that the other analog terminals are not used with the default configuration of the FHW-1v4 system). The membrane potential signal undergoes a final transformation due to the aforementioned input impedance of the oscilloscope channel.

Summarily, the membrane potentials of a quarter of all neurons per network block are measured with the same systematic error ΔLine.

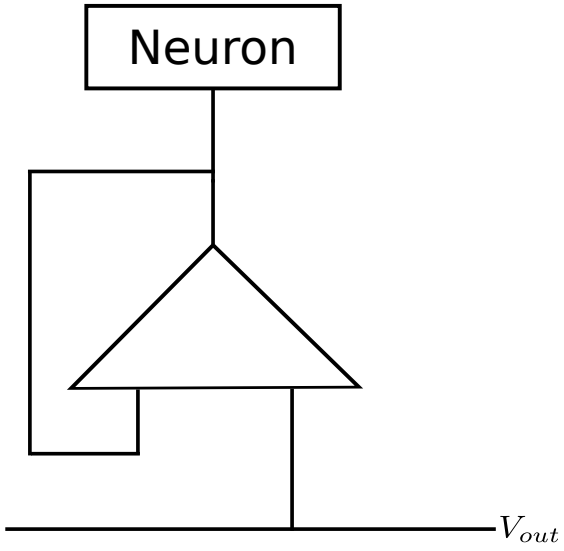


Figure 4.15: An operational amplifier is used to write the voltages for each potential parameter

When all of the above error sources are added, it follows that each voltage parameter of a neuron can be determined from the membrane potential trace with an uncertainty of

$$\Delta V = \Delta\text{Write} + \Delta\text{Read} + \Delta\text{Line} \tag{4.4}$$

thus leading to the conclusion that this intrinsic systematic variation can not be corrected for by the usual method of using bias values.

The significant offset (see figure 4.14) of the membrane potential caused by ΔV is ultimately translated to the corresponding large deviations in BVD. While some neuroscientific models are resilient against such large variations of their parameters, the utility of FHW-1v4 as a universal simulation platform is clearly diminished by this problem. It is possible, however, at the expense of absolute precision, to redefine the dynamic range of the neurons in a way that homogenizes the neuron parameter voltages in the biological domain.

This option is afforded by the freedom of choosing a custom HVD $\rightarrow$ BVD translation rule individually for each neuron and by the fact that the threshold potential, unlike like the resting one, can be determined without the $\Delta Write$ error.

First, the dynamic range of a neuron is chosen as the region between its resting potential (lower bound) and its typical threshold value (-55 mV BVD) since FHW-1v4 does not model the supra-threshold dynamics (action potentials) of spiking neurons. In a regularly spiking neuron (whose resting potential is set above the threshold potential for measuring purposes, see figure 4.6), the maximum of the membrane potential in a

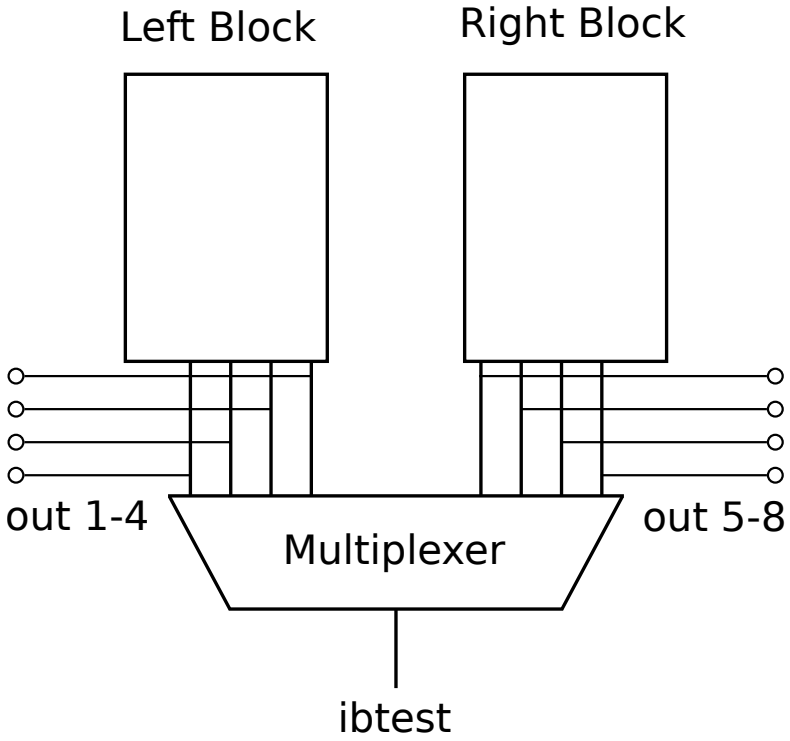
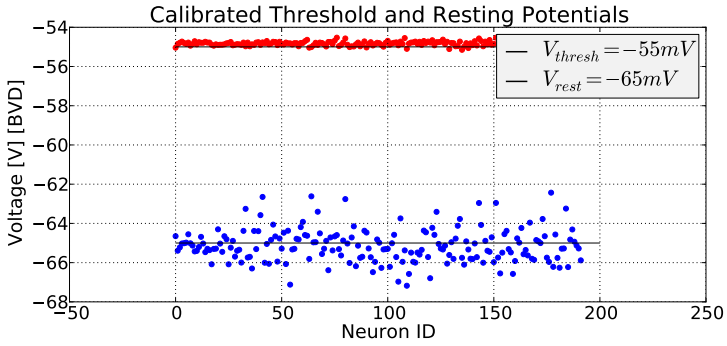


Figure 4.16: Analog readout lines for the Spikey chip.

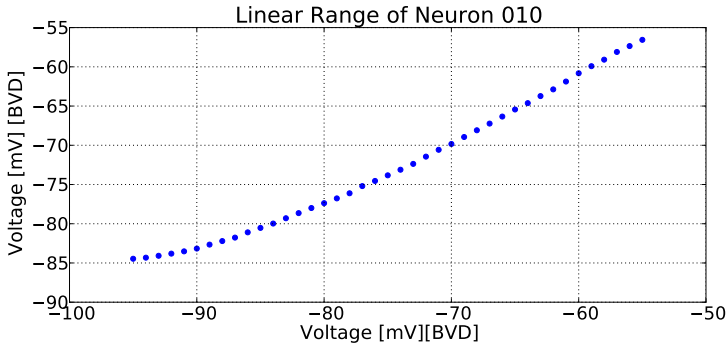
region about the spiking time reflects the correct value of its threshold (assuming that the comparator error is small). Since the readout error $\Delta\text{Read} + \Delta\text{Line}$ is the same for both the resting potential and the threshold potential, the only uncertainty remains $\Delta\text{Write}(V_{\text{rest}})$. When the HVD reading of a -55 mV target threshold is translated back to BVD, its value can be *defined* as the -55 mV level. The same applies to V_{rest}, where the error $\Delta\text{Write}(V_{\text{rest}})$ can not be avoided and represents an accuracy trade-off. Using these two points for the definition of a linear mapping HVD $\rightarrow$ BVD for *each* neuron, the situation is now significantly improved, as it is evident from the calibrated measurement shown in figure 4.17. Moreover, the linear range covers an extensive region between $[-80, -55]$ mV, despite the error in the resting potential.

4.2.3 Membrane Time Constant

An important precondition for the operation of the FHW-1v4 neuromorphic hardware is the calibration of the membrane time constants $\tau_{\text{m}} = C_{\text{m}}/g_{\text{l}}$ of the ANN neurons on the chip. This is not only necessary when the simulated neuroscientific models are sensitive to the systematic variations introduced by the manufacturing process, but is also required for the calibration of the synapse drivers due to the low-pass filter action of the neural membrane on synaptic input (conductance courses). The averaging of a PSP produced by a particular driver across different neurons is indeed doubtful when the distribution of τ_{m} in the uncalibrated state of the chip is considered (figure 4.18):



(a) Dynamical range of the neurons in a calibrated state



(b) Linearity in the dynamic range of the neuron. The resting potential has been set as a target value between -85 and -55 mV with a 1 mV and has been measured in a calibrated device. The linear range spans the usual interval between the threshold potential at -55 mV and the reset potential at -80 mV

Figure 4.17: Demonstration of the calibrated dynamic ranges on FHW-1v4

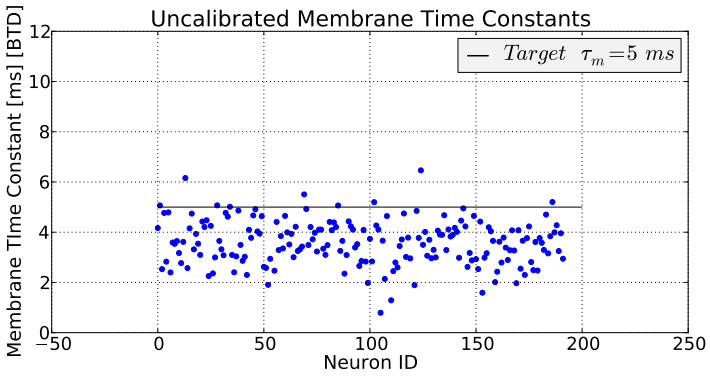


Figure 4.18: Uncalibrated membrane time constants. Neurons where the measurement failed have the corresponding values set to 0.

As mentioned before, the membrane capacitance is a fixed parameter and the only possibility for controlling the membrane time constant, therefore, is provided by the individually adjustable leakage term. Ideally, it is possible to calculate the conductance $g_\mathrm{l} = C_\mathrm{m}/\tau_\mathrm{m}$ for a particular value of the time constant when the capacitance is assumed to be 0.2 nF. Setting it via the PyNN interface, however, inevitably leads to the variations shown in figure 4.18 since both the capacitance C_m and the conductance g_l contribute to the deviation from the target value. Assuming a Gaussian distribution for both quantities, the compound error is given by:

$$\Delta\tau_\mathrm{m} = \sqrt{\frac{1}{g_\mathrm{l}^2}\Delta C_\mathrm{m}^2 + \frac{C_\mathrm{m}^2}{g_\mathrm{l}^2}\Delta g_\mathrm{l}^2} \tag{4.5}$$

The membrane time constant is not a simulation parameter, as in the neuron models used in computational neuroscience, but a physical quantity that has to be measured from the available data. Its calibration depends on a reliable method for determining its value based on the availability of the membrane potential trace and the spike train output produced by a neuron.

Membrane Time Constant Measurement

The measurement of the membrane time constant τ_m is partly based on a method suggested in[7] and is depicted in figure 4.19:

It follows from the solution of the first-order differential equation that models a leaky integrate-and-fire neuron with missing synaptic input:

$$-C_\mathrm{m}\frac{dV_\mathrm{m}(t)}{dt} = g_\mathrm{m}(V_\mathrm{m}(t) - E_\mathrm{l}) \;\rightarrow\; \frac{dV_\mathrm{m}(t)}{dt} = -\frac{g_\mathrm{m}}{C_\mathrm{m}}(V_\mathrm{m}(t) - E_\mathrm{l}) = -\frac{1}{\tau_\mathrm{m}}(V_\mathrm{m}(t) - E_\mathrm{l}) \tag{4.6}$$

As it describes the charging of a capacitor C_m with $R_\mathrm{m} = 1/g_\mathrm{l}$, the membrane potential always reaches the resting potential (leakage reversal potential) E_l asymptotically for an initial condition $V_\mathrm{m}(0)$ different from E_l:

$$V_\mathrm{m}(t) = E_\mathrm{l} + (V_0 + E_\mathrm{l})e^{-t/\tau_\mathrm{m}} \tag{4.7}$$

If the threshold potential is chosen to be less than the resting potential, the neuron will fire regularly, since $V_\mathrm{m}(t)$ can not reach E_l before crossing the threshold, thus producing equidistantly spaced spikes. The reset potential V_reset can then be chosen as the initial condition $V_0 = V_\mathrm{m}(0)$ in equation 4.7. Now it is possible to find a particular threshold potential, $V(\tau_\mathrm{m})$, so that the time elapsed during the charging of the membrane corresponds to the membrane time constant:

$$V(\tau_\mathrm{m}) = V_\mathrm{thresh} = E_\mathrm{l} + (V_\mathrm{reset} + (V_0 - E_\mathrm{l})e^{-1} \tag{4.8}$$

$$\tau_\mathrm{m} = -\frac{T}{\log\left(\frac{V_\mathrm{thresh}-E_\mathrm{l}}{V_\mathrm{reset}-E_\mathrm{l}}\right)} = -\frac{\mathrm{ISI} - \tau_\mathrm{ref}}{\log\left(\frac{V_\mathrm{thresh}-E_\mathrm{l}}{V_\mathrm{reset}-E_\mathrm{l}}\right)} \tag{4.9}$$

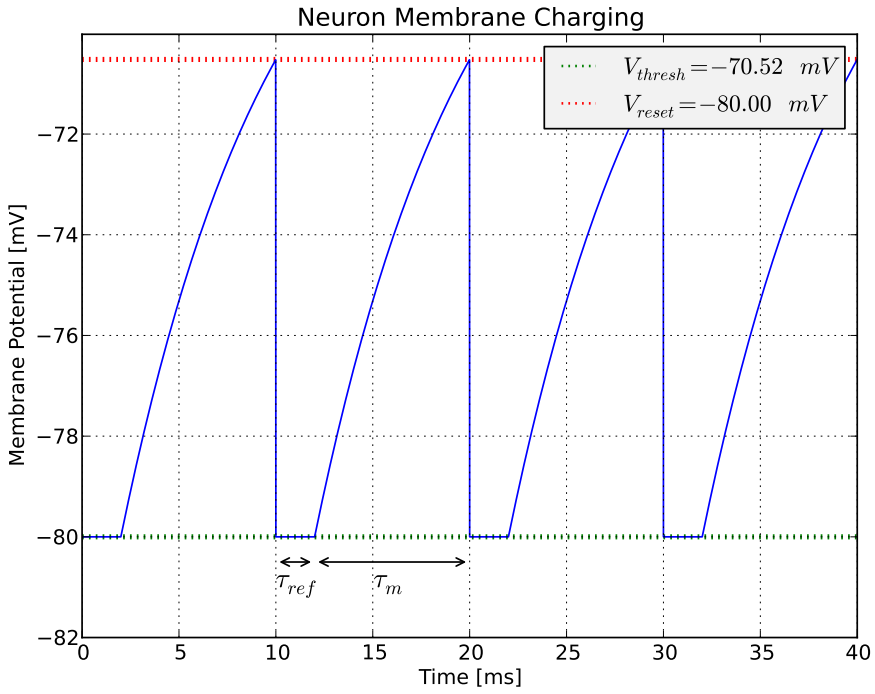


Figure 4.19: A schematic depiction of the measurement of the membrane time constant.

It is warranted to assume (see figure 4.14) that the variations in all parameters follow a normal distribution. The total error in the calculation of the membrane time constant can therefore be calculated by using Gaussian error propagation. The result of that calculation is given by the following expression:

$$
\begin{aligned}
L &:= \log\left(\tfrac{V_{\text{thresh}}-E_{\text{l}}}{V_{\text{reset}}-E_{\text{l}}}\right) \\
T &:= V_{\text{thresh}} - E_{\text{l}} \\
R &:= V_{\text{reset}} - E_{\text{l}} \\
\Delta\tau_{\text{m}}^2 &= \tfrac{1}{L^2}\Delta T^2 + \tfrac{T^2R^4}{T^2L^4}\Delta V_{\text{thresh}}^2 + \tfrac{T^2}{R^2L^4}\Delta V_{\text{reset}}^2 + \tfrac{T^2R^2}{L^4}\Delta E_{\text{l}}^2
\end{aligned}
\tag{4.10}
$$

In summary, the following steps are necessary to measure the membrane time constant on the FACETS stage1 hardware:

1. Calculate the threshold potential $V_{\text{thresh}} = V_{\text{m}}(\tau_{\text{m}})$ from equation 4.8.
2. Measure the actual values of V_{thresh}, V_{reset}, and E_{l} for the chosen target values.
3. Measure the interspike intervals for these settings
4. Calculate τ_m from equation 4.9

The presented routine was used in producing figure 4.18 which shows the distribution of all 192 time constants for neurons in the right network block with a target value of $\tau_{\mathrm{m}} = 5$ ms BTD.

Membrane Time Constant Calibration

The measurement routine can be used to find the correct settings of g_{l} that would produce the desired target value of τ_{m} on individual neuron basis. A sweep over all 1024 possible values of the controlling current is possible and has been used, in conjunction with linear interpolation for measurement purposes, but applying this method for all 192 neurons is rather impractical, due a number of reasons:

- A great number of measurements (192x1024)
- A non-linear relation between the controlling current and τ_{m} that confines the available range of values to a small region of this one-dimensional parameter space

A more efficient means for the calibration of τ_{m} for a particular value is the binary search algorithm [10] (see appendix for more details). It bisects recursively the available parameter range and quickly converges toward the required value. It has been implemented in the calibration framework and it takes less than 10 measurements, on average, to arrive at the correct setting for the current *ileak_cond*. Figure 4.20 shows the calibrated time constants of all 192 neurons belonging to the right network block of the chip:

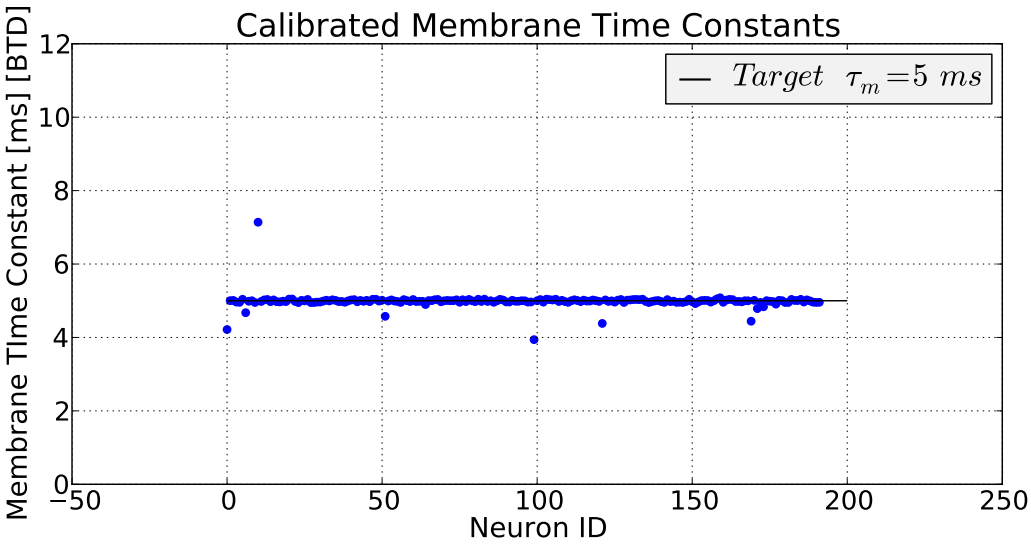


Figure 4.20: Calibrated membrane time constants

It should be emphasized that the suggested method of calibration works only for one particular target value of the membrane time constant. Thus, the system has to be

re-calibrated when a different τ_{m} is necessary for modeling purposes. A more general solution is to perform the parameter sweeps discussed above for all neurons and use the data for arbitrary values of τ_{m}. Due to the short time necessary for binary-search recalibration, however ($\sim$ 20 min), and because of the practical problems involving the handling of failure-tolerance issues and possible hardware defects, the trade-off provided by the binary search method seems a good alternative to the "brute-force" sweeping approach.

4.2.4 Synapse Drivers

As it was shown in section 4.1.2, the generation of post-synaptic potentials on the FHW-1v4 system is a multi-step process involving a number of intermediary electronic circuits. Particularly important is the post-synaptic signal path that the controlling current for the transconductance amplifier has to traverse between the synaptic node and the target neuron. The presence of parasitic capacitances along this line attenuates the current[7], resulting in smaller PSPs with a larger exponential decay constant τ_{fall}. These differ for drivers targeting the same post-synaptic neuron, since the signal propagates along a wire of varying height, depending on the vertical placement of the driver. It follows that when the corresponding synapse column is used in a basis activity regime, i.e. a number of drivers targeting the same neuron generate conductance courses sharing the same line , the parasitic capacitances are pre-charged, thus leading to PSPs of a different shape. Figure 4.21 illustrates the discrepancy for an excitatory PSP generated by the same driver in both circumstances:

Spikey does not allow for the superposition of conductance courses by the same driver in time and consequently the basis activity regime has to be generated by a group of other drivers, thus increasing the total conductance of the line. The total sum of their PSPs will be referred from now on as a *background noise*, whose sole purpose is to remove the dampening effect and to increase the conductance of the synapse column (a useful way to increase the synaptic conductance artificially, described in[2], is no more available in Spikey v4, since the corresponding controlling parameter is fixed due to technical problems with the previous version of the chip). The only possibility for inferring the shape of the PSP produced by the driver of interest in the basis activity regime remains the STA method described below.

If a large enough number of high-frequency inputs are connected to the target neuron via low synaptic weights, its membrane enters a high conductance state which reflects some changes of the neural properties that are useful for calibration purposes.

High Conductance State

Since FHW-1v4 uses conductance-based synapses, the total conductance at the membrane is a sum of the following terms

$$g_{\mathrm{tot}}(t) = g_{\mathrm{l}} + g_{\mathrm{exc}}(t) + g_{\mathrm{inh}}(t) \tag{4.11}$$

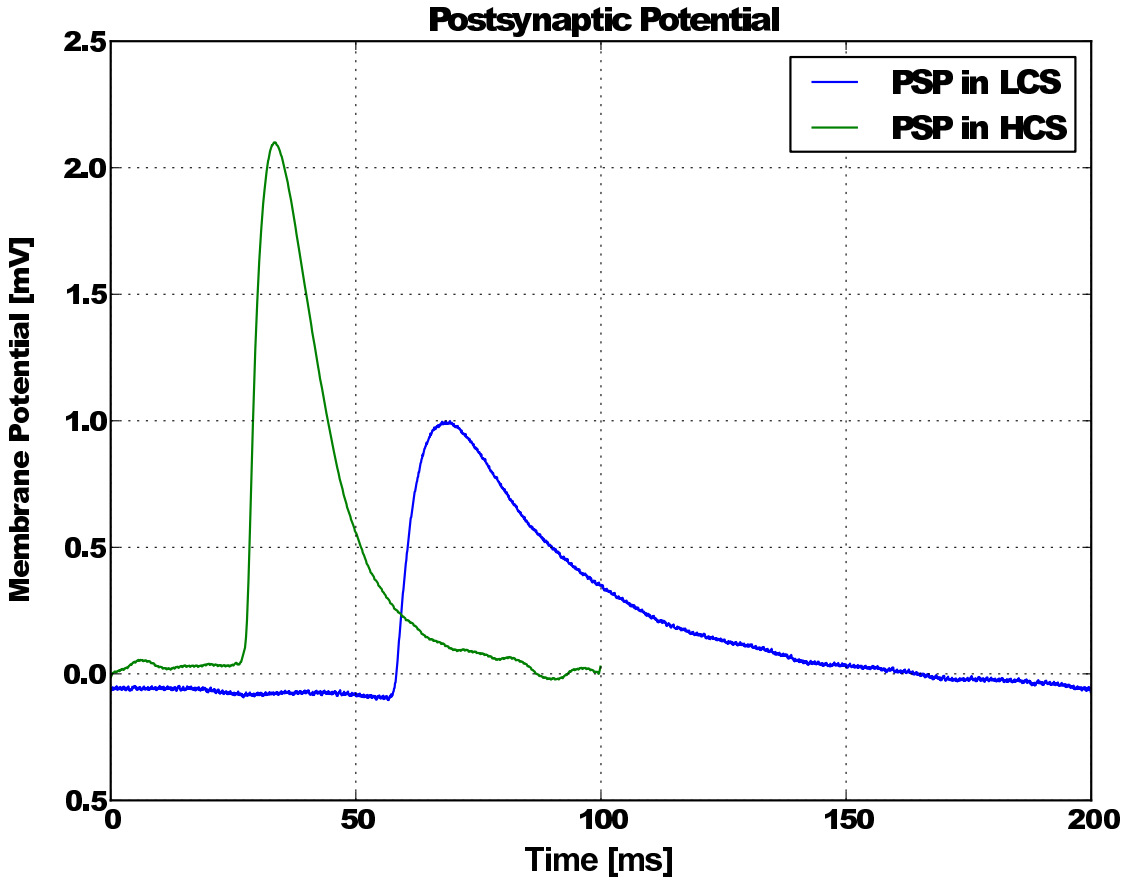


Figure 4.21: Post-synaptic potential generated by the same driver on a neuron when it is dampened and in the basis activity regime (measured by STA described below). The time offset is due to the measurement.

where $g_{\text{exc}}(t)$ denotes the sum of all conductance courses coming from the excitatory line and $g_{\text{inh}}(t)$ - those arriving from the inhibitory one. Under the conditions mentioned above the total synaptic conductance may exceed the leakage conductance g_{l}, leading to an activated state of increased network activity known as the *high-conductance state* (HCS). Experimental studies of neocortical neurons *in-vivo*[11] show that the typical ratio of the average synaptic conductance to the leakage conductance is $\langle g_{\text{exc}} \rangle / g_{\text{l}} = 0.73$ for excitatory synapses and $\langle g_{\text{inh}} \rangle / g_{\text{l}} = 3.67$ for inhibitory ones.

For conductance courses with an exponentially decaying kernel, the average total conductance can be estimated by the following formula[8]

$$\langle g_{\text{tot}}(t) \rangle = \sum_{s \in \{\text{exc,inh}\}} \nu_s w_s \tau_s + g_{\text{l}} \tag{4.12}$$

where the rate ν_s, the weights w_s, and the synaptic time constants τ_s are fixed for all excitatory and for all inhibitory inputs.

Since the membrane time constant τ_{m} is directly proportional to the total conductance due to its fixed capacitance C_{m}, it attains smaller values during intensive stimulation of the neuron:

$$\tau_{\mathrm{m}}(t) = \frac{C_{\mathrm{m}}}{g_{\mathrm{l}} + g_{\mathrm{exc}}(t) + g_{\mathrm{inh}}(t)} \tag{4.13}$$

This accounts for the improved resolution of the neuron when it responds to temporally close excitatory PSPs (also known as coincidence detection property) that has important neuroscientific implications.

At this time it is useful to introduce the concept of the *effective reversal potential* $V_{\mathrm{eff}}(t)$ whose time-average can be determined experimentally from the mean of the membrane potential trace of the neuron under stimulation[8] and is used in the fitting procedure described in section 4.2.4:

$$V_{\mathrm{eff}}(t) = \frac{1}{\langle g_{\mathrm{tot}}(t) \rangle} \left(g_{\mathrm{l}} + \sum_{s \in \{\mathrm{exc,inh}\}} g_s(t) \right) \tag{4.14}$$

The effects of the high-conductance state on neural behavior can be summarized in the following points[7]:

- Low input resistance
- Increase in the temporal resolution of quasi-concurrent input stimuli
- Large membrane potential fluctuations due to the intensive input
- Stochastic firing pattern due to proximity of the average effective potential $\langle V_{\mathrm{eff}}(t) \rangle$ to the threshold potential V_{thresh}
- Dominant inhibitory conductances

The relevance of the high-conductance state to the calibration of the FHW-1v4 system is three-fold:

1. The neurons used for synaptic driver calibration must operate in a basis activity regime which can be achieved by using HCS.
2. Neurons in HCS exhibit a stable mean of the effective potential V_{eff} which can be used to find the working point for HCS and also to measure the parameters of a PSP.
3. The shape of the conductance course is influenced by the total conductance $\langle g_{\mathrm{tot}}(t) \rangle$ and $V_{\mathrm{m}}(t)$ follows the synaptic reversal potential more closely when the neuron is in HCS due to the small $\langle \tau_{\mathrm{m}}(t) \rangle$.

The practical difficulty associated with using high-conductance state in Spikey is its characterization (achieving HCS ≡ finding the working point, see[7]) and the extraction of the PSPs from the fluctuating membrane potential.

Measurement Methodology

FHW-1v4 does not provide any facilities for the direct recording of synaptic conductances. Even then, they would be distorted by attenuation if the chip is not used in a basis activity regime. It is necessary, therefore, to devise an indirect method for determining the height and the synaptic time constants (τ_{rise} and τ_{fall}) for a conductance course from the corresponding PSP shape in a high-conductance state. This implies finding the solution of two problems:

a) How to extract the PSP generated by a single driver from the background noise

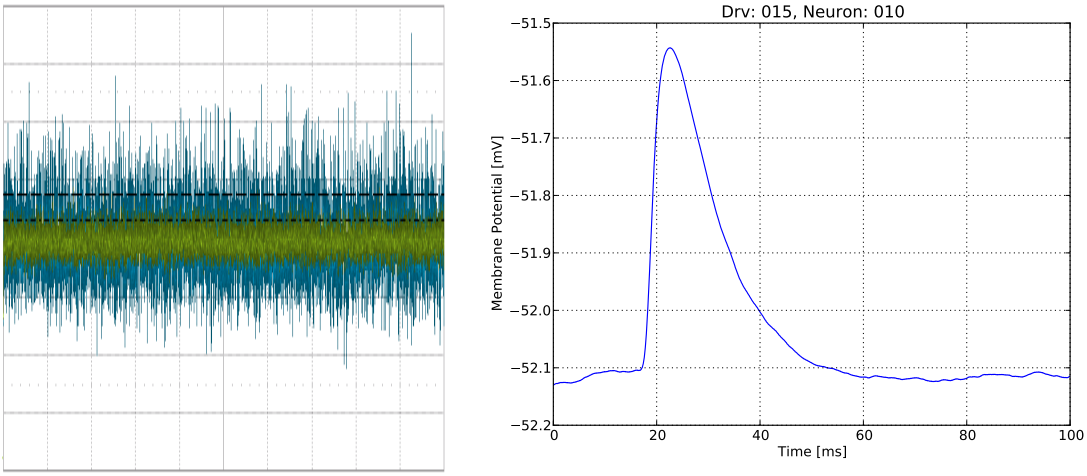
b) How to measure the conductance course parameters from the PSP

The following sections assume that the neuron is in HCS and that the measurement objective is a single-driver PSP. It is important to note that all measurement involving post-synaptic potentials are performed when the spiking mechanism is disabled, to avoid distortions in the neural dynamics. This is done both in software and in hardware by setting the threshold potential to unusually high level that can not be reached even by large fluctuations of the membrane potentials.

Spike-Triggered Averaging

The variation in a neuron's membrane potential $\sigma^2 = Var\left[V_{\text{m}}(t)\right]$ in HCS is typically larger than the magnitude of the post-synaptic potentials and attains white noise characteristics for a large number of active synapses, high input frequencies, and low synaptic weights. The membrane trace data contained in non-overlapping time intervals therefore represent different realizations of the same random process. According to the central limit theorem, the mean sample variance of these sub-traces behaves as $\sigma^2_{mean} = \sigma^2/N \to 0$, for $N \to \infty$, so that the background noise essentially disappears for a high enough sample number N. If each membrane trace slice contains a PSP that is positioned with exactly the same time offset relative to its start, the lowering of the background noise gradually reveals its shape, which represents the low-pass filtered mean conductance course produced by the synaptic driver of interest (figure 4.22). This method is known as *Spike-Triggered Averaging*(STA) and is used extensively in experimental neuroscience [7].

Crucial for STA is the availability of both spike event data and the membrane potential trace of the neuron that is used to perform the averaging. Custom algorithms are needed for the correct determination of the sample intervals around the time at which a conductance course arrives at a neuron from the corresponding spike train which is typically Poisson distributed. An additional hurdle is the fact that the membrane trace recordings and the spike times are not perfectly synchronized (see section 4.3.1), leading



(a) The averaging of many membrane traces on the oscilloscope reduces the background noise. The blue trace represents a single trial, while the green one is the average over 10 experiments.

(b) A post-synaptic potential extracted from the trace shown on the left. A total number of 10000 samples were used for the purpose.

Figure 4.22: The extraction of a PSP from the background noise.

to inaccurate averaging of the PSP.

The simple version of STA developed by the author tries to circumvent these problems by sending a regularly spaced spike train from the investigated driver, so that the membrane potential can be divided up in regular slices that are averaged very efficiently using the facilities provided by the utilized Numpy[22] programming library. In this way any systematic offsets between the spiking time and the PSP trace can be accounted for by adjusting the slice points on the membrane trace and no spike events are needed for the procedure. In fact, the averaging itself is much faster than the other operations involved in setting up the experiment, so that the actual bottleneck becomes the simulation on the chip and the membrane trace acquisition by oscilloscope rather than the processing of the data.

Determining the Parameters of the Conductance Course

The shape of the conductance course described in section 4.1.2 differs significantly from the standard kernels (exponential decay or alpha function) typically used in software simulations modeling conductance-based synapses by the rising exponential flank and the onset controlled by the V_{start} parameter. It is to be expected that the resulting PSP may behave differently and would not be amenable to the same fitting methods that are used in the standard case. Therefore, a model of the low-pass filtering action of the

membrane on this conductance course that describes the PSP would provide important insight into the behavior of the hardware.

The starting point is a simplified form of equation 4.1 that accounts for only one input by a synapse driver

$$-C_{\mathrm{m}}\frac{dV_{\mathrm{m}}(t)}{dt} = g_{\mathrm{l}}(V_{\mathrm{m}}(t) - E_{\mathrm{l}}) + g(t)(V_{\mathrm{m}}(t) - E_{\mathrm{syn}}) \tag{4.15}$$

where $g(t)$ is the conductance course shown in figure 4.9 and E_{syn} stands for the excitatory (E_{x}) or the inhibitory (E_{i}) reversal potential, depending on the synapse type. This equation is not solvable analytically for the function $g(t)$ and has to be simplified. The approximation (suggested by M. A. Petrovici) is founded on the fact that the magnitude of the PSPs in the high-conductance state is typically much smaller than the difference between the membrane potential and the reversal potential and can therefore be taken as the constant R:

$$V_{\mathrm{m}}(t) - E_{\mathrm{syn}} \approx const =: R \tag{4.16}$$

$$-C_{\mathrm{m}}\frac{dV_{\mathrm{m}}(t)}{dt} = g_{\mathrm{l}}(V_{\mathrm{m}}(t) - E_{\mathrm{l}}) + g(t) \cdot R \tag{4.17}$$

The approximation decouples the conductance from the membrane potential and allows equation 4.17 to be solved piece-wise for each of the following regions of the conductance $g(t)$

$$g(t) = \begin{cases} 0, & \text{for } 0 \leq t < t_0 \\ m \cdot t, & \text{for } t_0 \leq t < t_s \\ e^{-\frac{t}{\tau_\mu}} + g_s - 1, & \text{for } t_s \leq t < t_g \\ g_{\max} \cdot e^{-\frac{t}{\tau_\lambda}}, & \text{for } t \geq t_g \end{cases} \tag{4.18}$$

that is delineated by the time points $0, t_0, t_s, t_g$ and parametrized by the slope m, the time constants τ_μ and τ_λ, as well as the maximum conductance $g_{\max}$ and the onset conductance g_s that are shown in figure 4.23 a):

As it is evident from the comparison with the numerical solution for the same conductance course $g(t)$ (figure 4.23 d)), the approximation is valid even for much wider range than the typical PSP heights of $\approx 1 - 2$ mV in HCS.

While the solution has been shown for a single PSP for didactic purposes, the solution which results in for the high-conductance state, according to the theory presented in [8] has the same form and can therefore be adapted for the measurement of PSPs extracted by the STA method in the basis activity regime required for such measurements.

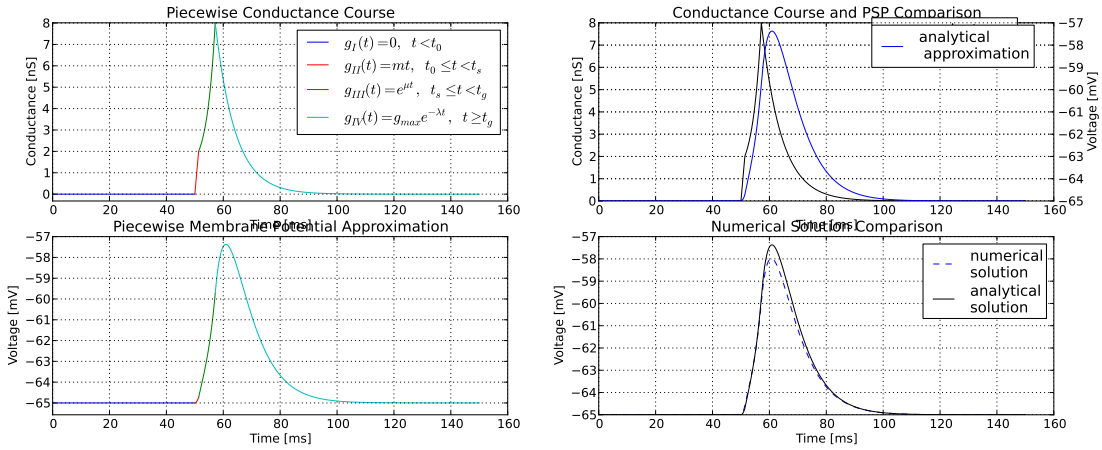


Figure 4.23: Conductance Course

Synapse Driver Calibration

An accurate measurement of the post-synaptic potentials can only be made in a basis activity regime that requires significant background stimulation. In the uncalibrated state the synapse drivers generate conductance courses with varying parameters due to process variation. When a particular subset of all drivers is chosen, however, they tend to produce a stable average membrane potential $\langle V_\mathrm{m}(t) \rangle$ that can be used to measure the impact of the total synaptic background activity on the neuron.

Since the distribution of the synaptic parameters is unknown in advance, it is useful to employ a software simulator for the purpose of finding a working point [7] $\langle V_\mathrm{m}(t) \rangle^{wp} = V_\mathrm{rest} + 2/3 \cdot (V_\mathrm{thresh} - V_\mathrm{rest})$ that defines the required basis activity regime and which can be reached on the hardware by the adjustment of the current I_amp that controls the height of the voltage ramp discussed previously, thus leading to a change in the height of the conductance course impinging on the neuron.

A biologically meaningful activity regime can be achieved when about 20% of the input is chosen to be inhibitory, and when high enough activity rates are used together with low synaptic weights. The search for the working point in software uses 208 excitatory inputs and 48 inhibitory inputs that stimulate one neuron. It consists of two steps:

- First, all inhibitory weights are set to zero, so that only excitatory input is available. Starting from 0, the weights of the excitatory synapses are eventually increased until the threshold level V_thresh is reached.
- In the second step the excitatory weights are kept, while the inhibitory ones are slowly increased, so that the average membrane potential reaches the target of $\langle V_\mathrm{m}(t) \rangle^{wp}$.

The described procedure delivers target weight values for both the excitatory and the

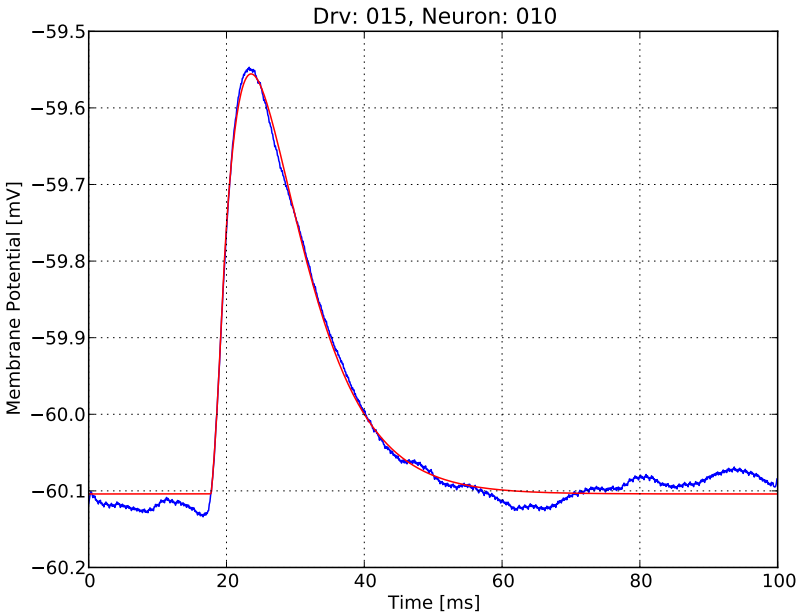


Figure 4.24: The fitting procedure is applied to an STA-extracted PSP from Spikey. Note that in contrast to software simulations, the rise of the PSP is not instantaneous.

inhibitory synapses that can not be applied directly in hardware when the synapse drivers are uncalibrated. They are still set, but the working point is found by a different method.

Since an overall increase of I_{amp} for all excitatory or inhibitory drivers does not necessarily produces the same result as it would do in a software simulation with comparable weights (due to process variations leading to distributed values of I_{amp}), the method for finding the working point in hardware is applied using the previously mentioned binary search algorithm:

- The weights of all inhibitory inputs is set to zero and the values of I_{amp} that lead to an average membrane potential equal to the threshold V_{thresh} are found within a certain tolerance by the bisection of the one dimensional parameter space of I_{amp}.
- Keeping the excitatory controlling currents, the values of I_{amp} for the inhibitory synapses that produce the desired average potential $\langle V_{\text{m}}(t)\rangle^{wp}$ are found within a certain tolerance using the same method.

Once that the working point has been established both in software and hardware, it is possible to compare the characteristics of individually extracted post-synaptic potentials using the STA method.

A PSP extracted from a software simulation provides a target height and a synaptic time constant that can be reached by the repeated adjustment of the controlling currents

I_{amp} and $I_{\tau_{fall}}$. This is again accomplished with the binary search algorithm, which converges sufficiently quickly toward the target values.

While the described method has been used for the calibration of individual synapse drivers, an automated calibration routine is subject to a further investigation, since for the number of samples, typically utilized for the STA method (10000) the characteristics of individual synapse drivers still show significant variability. A further increase in the number of samples significantly slows down the calibration process, therefore it might be necessary to implement an additional optimization of the fitting algorithm.

4.2.5 Per-Neuron Weight Factors

As it was shown previously, individual neurons have a widely ranging dynamic range, compared to each other, which can not be calibrated away because even homogenized PSPs would have a different impact on their dynamics. The only possibility for dealing with this problem is to find a weight factor that scales each conductance course impinging on the neuron in a way that it its comparable to other neurons with a different dynamic range. A method for achieving this goal has been implemented by the author in the calibration software framework developed for this thesis and was shown to function properly when a particular set of drivers are used for the procedure.

It is based on the fact that the weights defined in the model are mapped stochastically (as shown in [7]) to the available 4-bit values on the hardware, so that large numbers of inputs can achieve a mean effect with a higher resolution than individually utilized synapse drivers. It uses the binary search algorithm to find a particular multiplicative factor so that the average membrane potential $\langle V_{\text{m}}(t) \rangle$ of the neuron can reach a predefined value corresponding to the correct dynamics.

4.3 Encountered Hardware Imperfections

Due to the prototype nature of the FHW-1v4 system, there are inevitable hardware problems that are not known in advance and can be encountered by the users of the chip. Several such imperfections have been discovered by the author in the course of this thesis.

4.3.1 Synchronization of Voltage Trace and Spike Times

One of the first problems that are evident when using the hardware is that there is a fixed time interval between events shown on the membrane trace of each neuron and the corresponding spike events which are delivered digitally to the host PC. Figure 4.25 shows such a measurement for a single neuron. The difference is specific for each neuron is typically on the order of several milliseconds [BTD]. Since this is a systematic variation, it is possible to calibrate it by offseting the spiking times so that they coincide with the maxima used to measure threshold potentials when the neuron fires (see figure 4.25).

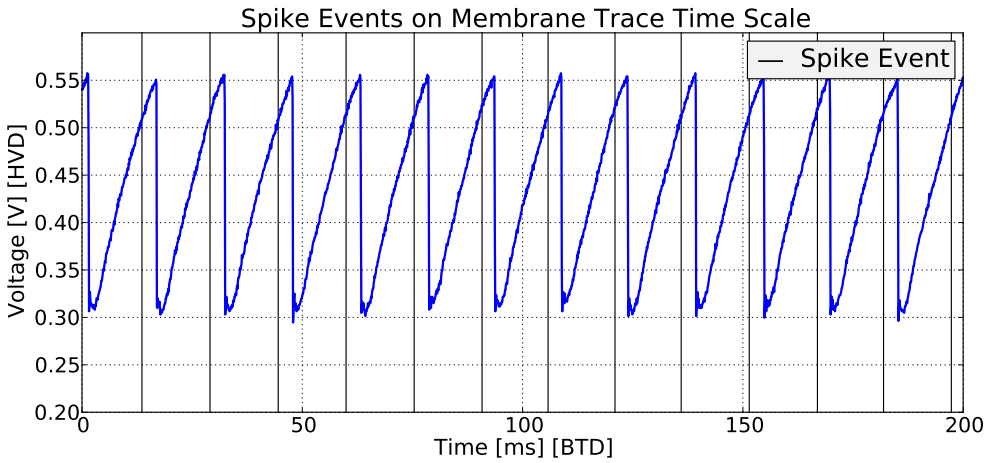


Figure 4.25: Membrane trace and spike times are not perfectly synchronized.

4.3.2 Fixed Synaptic Delays

Even when the spikes and traces were apparently synchronized, fixed synaptic delays that can not be calibrated away were encountered. Since the best way to detect such effects is to rely on measurements using only the membrane potential, the following setup was utilize to demonstrate unequivocally the presence of delays.

A single external spike is sent to a neuron over a synaptic connection that is strong enough to induce firing. The neuron is connected to itself via an inhibitory synapse. When its membrane potential drops to the reset potential after it spikes, it should become even more negatively polarized due to the inhibition. As it can be seen on figure 4.26, however, this does not happen immediately, but there is a delay of about 1.5 ms. Such delays were found to be neuron-specific and generally of the same order as the measurement presented here.

4.3.3 Distribution of Refractory Times

The refractory times of the neurons can not be adjusted in network models in the current version of the stage1 system. Instead, a particular parameter has to be adjusted manually. Still, there is the question of how such refractory times are distributed in time when a single neuron is used in an experiment.

Since the membrane trace of regularly spiking neuron (with a threshold potential lower than its resting potential) is clamped to its reset value V_{reset} during the refractory period, it is possible to measure this interval by fitting a function that describes the charging of a capacitor, $\theta(t-t_0)C(1-\exp(-\frac{t-t_0}{\tau_{\text{m}}})$ to the membrane trace between two spike events. The distribution of such a measurement for the default values on the hardware system is shown in figure 4.28

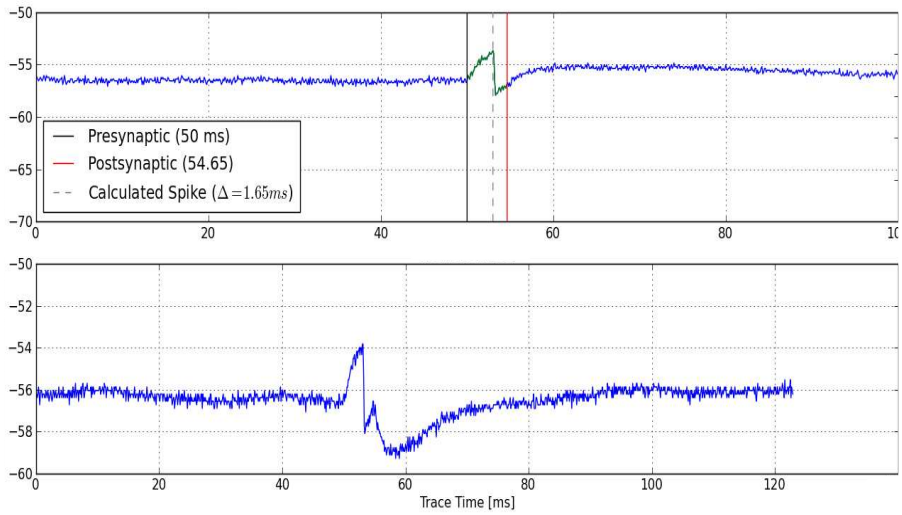
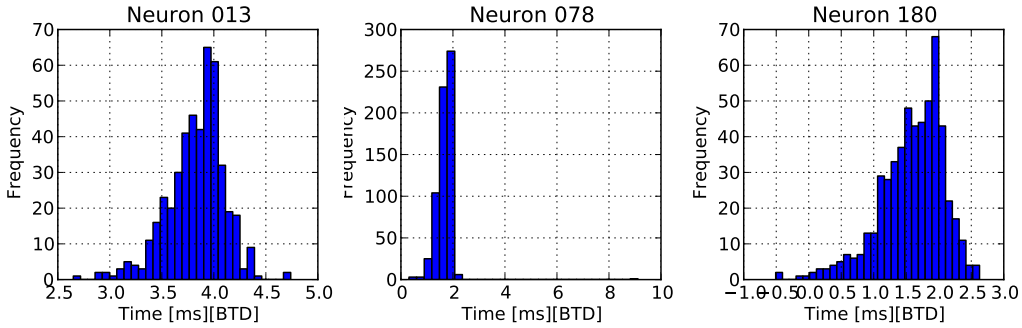


Figure 4.26: Measurement of a fixed synaptic delay at one neuron. The top trace shows that the spiking time (dashed line) is synchronous with the voltage trace.



(a) neuron 13 **(b)** neuron 78 **(c)** neuron 3

Figure 4.27: Distribution of refractory times at 3 randomly chosen neurons.

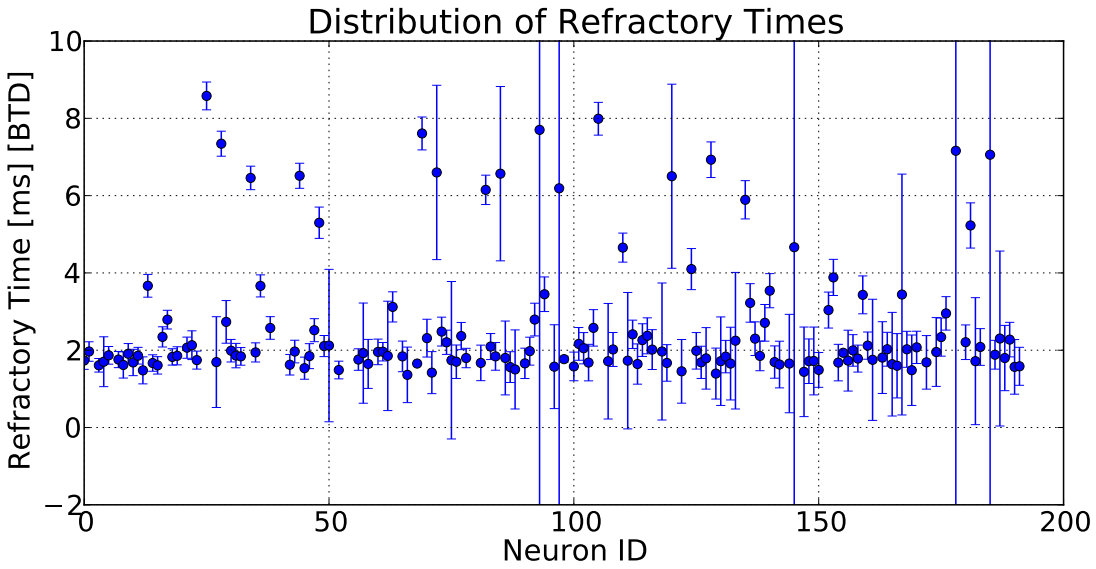


Figure 4.28: Distribution of the refractory times for the whole chip

5 Discussion and Outlook

The implementation of belief propagation on neuromorphic hardware is an endeavour that involves a multi-stage effort, entailing the investigation of various theoretical, experimental, and engineering problems. The groundwork toward achieving this goal has been laid in this thesis, which describes essential preparatory steps, including statistical inference models, software simulation results, and calibration methods.

The first step toward implementing statistical inference in neural networks concerned the development of a software framework that can be used to perform simulation experiments for any chosen model. It provides the theoretical calculations that are necessary to compare the performance of the neural network, as well as an implementation of the training procedure that is applied to individual factors. Simulations with current-based synapses were deemed necessary, in order to reproduce the results provided in [37], since the method has not been applied to conductance-based models so far. Almost all parameters in the corresponding simulations followed closely the ones presented in the pioneering work of [37], but the training performance differed significantly, depending on the complexity of the implemented factor function. While simple factors like binary conditional probabilities $p(x|y)$ and equality constraint factors $\delta(x-y)\delta(x-z)$ achieved very high correlations with the theoretically expected output in the course of their training (> 0.93), some seemingly uncomplicated functions like binary addition $\oplus$ (XOR) could not be trained above chance level. This has led to the inability to test the performance of the binary channel model in a fully implemented form, due to the distorted dynamics of the parity check node. The reasons for the subpar training results of particular factor functions remain unclear and need to be further investigated.

It is clearly possible to improve upon the training method for individual factors, which is currently based on the linear regression methodology described in section 2.5.5. Better results might also be achieved if the projections from readouts and external input sources to the liquids are arranged in a different manner, as it is currently done (they project to disjoint regions of the liquid to avoid mixing of information, according to [37]). A systematic parameter search study may result in a better training performance by optimizing the size of the liquids in relation to their connection density λ.

The model corresponding to the explaining away problem, on the other hand, was constructed successfully from individual factors that showed a very high input correlation with randomly generated novel input and was used in a long experimental series of individual trials lasting for 1 s where each simulation used a different set of randomly generated inputs. As it is shown in figure 3.19, factors that were directly connected with the input sources and represent simpler functions generally perform better than the rest,

but there is still a significant distortion of the expected network dynamics.

Because of inevitable error propagation, the quality of inference in the inherently noisy implementation of liquid factor graphs evolves inversely with network depth (i.e. maximum distance of any node in the graph to the closest leaf node). A type of network that avoids this problem is a chain, since messages are updated at every step from external sources. An example is provided by the proposed simplified Kalman filter (section 2.4) that was developed specifically for an application on neuromorphic hardware, since its chain-like structure also entails a sparse connectivity matrix, thus taking the pressure off bandwidth constraints between individual neuromorphic chips.

If statistical inference performed by this model can be successfully demonstrated, the next step towards achieving belief propagation in hardware is the introduction of a new training method for conductance-based synapses. While a superficial similarity between the synapse-injected currents and the conductance courses does exist, the non-linear dynamics that the latter induce may invalidate the linear regression method used for the current-based case.

A single-chip FHW-1v4 does not provide enough neurons and synaptic connections to be able to emulate statistical inference models without further modification. Therefore, the next step in implementing them in hardware should involve the utilization of the multi-chip system, which has only recently been in operation, since the FPGA code required for multi-Spikey operation was not fully developed during the time of the work presented in this thesis. It may still contain unknown technical problems, because it has not been used for modeling purposes so far. A long-term perspective would, of course, involve using the waferscale device, as it is vastly superior to the multi-Spikey chip both in terms of available neurons and regarding the available connectivity.

The calibration of a single FHW-1v4 chip was presented in chapter 4. It described a method for ameliorating the large process variation in the dynamic range of the neurons, as well as a calibration routine for homogenizing their membrane time constants. A new measurement method that is capable of determining the characteristics of the post-synaptic potentials and their underlying conductance courses was used to calibrate the synapse drivers on the chip. For most synaptic drivers, the fit can provide a reliable results, however, there exist some drivers which show highly atypical PSP shapes for which the method obviously can not work. For exceptionally weak drivers, long STA runs need to be performed to achieve an acceptable signal-to-noise ratio.

It is possible, however, to make the involved data fitting procedure more robust against such problems by the introduction of additional failure-tolerance checks in its implementation. Since it is a purely membrane trace-based method, the measured quantities are entirely in the biological domain (BVD, BTD), so it can be used to test the performance of spike-based methods for measuring the synaptic time constant. Which one of the two methods turns out to be the most efficient is a high-priority question that needs to be addressed in further studies.

The PSP characterization routine has also been used successfully in measuring the parameters of post-synaptic potentials on the HICANN chip, which is part of the wafer-scale

hardware the Electronic Vision(s) group is currently developing (personal communication M. A. Petrovici).

In the course of operating the FHW-1v4 system, several imperfections were encountered. Firstly, it was found out that the membrane potential traces are not perfectly synchronized with the spiking times delivered by the digital electronics. This has led to the discovery of fixed synaptic delays on the order of 2-3 ms in biological time which are probably due to signal processing at either the synapse drivers or the synapse nodes. In addition, the refractory times of individual neurons (a parameter which can not yet be controlled directly via the PyNN interface) are distributed randomly in time, with a neuron-specific distribution shape and variance (mostly on the order of 1 ms).

In summary, while significant progress was made toward achieving belief propagation on neuromorphic hardware (development of the necessary software, calibration methods), more work is required if this goal is to be realized. First of all, the simplified Kalman filter model has to be tested for current-based synapses. The development of a conductance-based training method is required to test the compatibility of method with the leak integrate-and-fire neuron model used in the hardware. Parameter optimization, especially for the size of the involved neural populations is the next step in achieving full compatibility with the FHW-1v4 system.

On the hardware side, the completion of the synapse driver calibration process for individual FHW-1v4 devices is a necessary prerequisite for having a fully operational multi-chip system that can be used for emulating neuroscientific models. The methods described here are already being used for the calibration of the next-generation hardware, the HICANN chip on the BrainScaleS waferscale device.

Bibliography

[1] Davison AP, Brüderle D, Eppler JM, Müller E, Pecevski Da, Perrinet L, and Yger P. Pynn: a common interface for neuronal network simulators. *Front. Neuroinform.*, 2, 2008.

[2] Johannes Bill. Self-stabilizing network architectures on a neuromorphic hardware system. Master's thesis, University of Heidelberg, 2008.

[3] Christopher M. Bishop. *Pattern Recognition and Machine Learning (PRML).* Springer, Springer Science+Business Media, LLC, 233 Spring Street, New York, NY 10013, USA, 2006.

[4] Brainscales. Brainscales project, 2011.

[5] Robert Brown and Patrick Hwang. *Introduction to Random Signals and Applied Kalman Filtering, 3rd ed.* Addison-Wesley, 1997.

[6] Daniel Bruederle, Andrew Davison, Jens Kremkow, Eric Mueller, Eilif Muller, Martin Nawrot, Michael Pereira, Laurent Perrinet, Michael Schmuker, and Pierre Yger. Neurotools, 2010.

[7] Daniel Brüderle. *Neuroscientific Modeling with a Mixed-Signal VLSI Hardware System.* PhD thesis, University of Heidelberg, 2009.

[8] Ilja Bytchok. From shared input to correlated neuron dynamics: Development of a predictive framework. Master's thesis, University of Heidelberg, 2011.

[9] N.T. Carnevale and M.L. Hines. *The NEURON Book.* Cambridge University Press, 2006.

[10] D.E.Knuth. *The Art of Computer Programming.* Addison-Wesley, 1997.

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

[12] Julie Dethier. Spiking neural network decoder for brain-machine interfaces. *Proceedings of the 5th International IEEE EMBS Conference oin Neural Engineering, Cancun, Mexico*, 2011.

[13] Bibbona E., Panfilo G., and Tavella P. The ornstein–uhlenbeck process as a model of a low pass filtered white noise. *Metrologia*, 45:117–126, 2008.

[14] FACETS. Fast analog computing with emergent transient states, 2009.

[15] C.D. Fiorillo. Towards a general theory of neural computation based on prediction by single neurons. *PLoS ONE*, 3(e3298), 2008.

[16] C.D. Fiorillo. A neurocentric approach to bayesian inference. *Nature Reviews Neuroscience*, 11(605), 2010.

[17] G. David Forney. Codes on graphs: Normal realizations. *IEEE Transactions on Information Theory*, 47(2):520–548, 2001.

[18] W. Gerstner. *Spiking Neuron Models*. Addison-Wesley, 2002.

[19] Marc-Oliver Gewaltig and Markus Diesmann. Nest (neural simulation tool). *Scholarpedia*, 2(4):1430, 2007.

[20] Andreas Grübl. *VLSI Implementation of a Spiking Neural Network*. PhD thesis, University of Heidelberg, 2007.

[21] Andreas Hartel. Improving and testing a mixed-signal vlsi neural network chip. Master's thesis, University of Heidelberg, 2010.

[22] Eric Jones, Travis Oliphant, Pearu Peterson, et al. SciPy: Open source scientific tools for Python, 2001–.

[23] R.E. Kalman. A new approach to linear filtering and prediciton problems. *Transactions of the ASME-Journal of Basic Engineering*, 82 (Series D):35–45, 1960.

[24] D.C. Knill and D. Kersten. Apparent surface curvature affects lightness perception. *Nature*, 351:228–230, 1991.

[25] Frank. R. Kschischang, Brendan J. Frey, and Hans-Andrea Loeliger. Factor Graphs and the Sum-Product Algorithm (FGSPA)). *IEEE Transactions on Information Theory*, 47:498–519, 1998.

[26] Frank. R. Kschischang, Brendan J. Frey, and Hans-Andrea Loeliger. Factor Graphs and the Sum-Product Algorithm (FGSPA)). *IEEE Transactions on Information Theory*, 47(2):498–519, 2001.

[27] Hans-Andrea Loeliger. An Introduction to Factor Graphs (AIFG)). *IEEE Signal Processing Magazine*, 21(1):28–41, 2004.

[28] Nawrot M., Aersten A., and S. Rotter. Single-trial estimation of neuronal firing rates: From single-neuron spike trains to population activity. *Journal of Neuroscience Methods*, 94(1):81–92, 1999.

[29] Wei Ji Ma. Bayesian inference with probabilistic population codes. *Nature Neuroscience*, 9:1432–1438, 2006.

[30] W. Maass and Prashant J. Computational aspects of feedback in neural circuits. *PLoS Computational Biology*, 3(1):15–34, 2007.

[31] W. Maass, T. Natschlaeger, and H. Markram. Real-time computing without stable states: A new framework for neural computation based on perturbations. *Neural Computation*, 14:2531–2560, 2002.

[32] Eric Müller. Operation of an imperfect neuromorphic hardware device. Master's thesis, University of Heidelberg, 2008.

[33] Dayan P and Abott L.F. *Theoretical Neuroscience*. MIT Press, Cambridge, Massachusetts, 2001.

[34] Judea Pearl. Reverend Bayes on inference engines: A distributed hierarchical approach. *AAAI-82 Proceedings*, pages 133–136, 1982.

[35] Judea Pearl. *Probabilistic Reasoning in Intelligent Systems, 2nd ed.* Morgan-Kaufmann, 1988.

[36] H.W. Sorenson. Least-squares estimation: from gauss to kalman. *IEEE Spectrum*, 7:63–68, 1970.

[37] Andreas Steimer, Wolfgang Maas, and Rodney Douglas. Belief Propagation in Networks of spiking Networks (BPNN)). *Neural Computation*, 21:2502–2523, 2009.

[38] G. van Rossum. Python tutorial. *Technical Report CS-R9526, Centrum voor Wiskunde en Informatica (CWI)*, 1995.

[39] Jonathan S. Yediddia. Constructing free-energy approximations and generalized belief propagation algorithms. *IEEE Transactions on Information Theory*, 51(7):2282–2312, 2005.

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, March 19, 2012

.....
(signature)