Master's Thesis

Neural Clustering Applied to Neuromorphic Hardware and Population Coding

---

Helmut Puhr

Graz, 2011

Institute for Theoretical Computer Science
University of Technology Graz

Graz University of Technology

Institute for Neuroinformatics
ETH Zürich

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Supervisor/First reviewer: O.Univ.-Prof. Dipl.-Ing. Dr.rer.nat. Wolfgang Maass
Second reviewer: Dr. Shih-Chii Liu

Abstract (English)

The goals of this work are twofold: In the first part a more application driven framework is developed to show the applicability of a previously developed method to object recognition and basic tracking using a recently developed hardware retina. In computer vision, the task of object recognition has been around for several decades, and has been approached numerous times with different objectives and methods. However, most approaches used methods that are difficult, if not impossible to implement using biologically plausible neural structures. A different approach is to explore possibilities in a bottom-up approach, using relatively simple neuron models to build networks which are, in principle, capable of recognizing and classifying any given input. While the simplicity of the approach presented here limits the complexity of the tasks that can be solved, it also shows a general approach to how a relatively basic neural network can be used to solve non-trivial tasks in a computationally efficient way.

In the second part, the same method is used in a more biological context. In the brain, a stable method to store information is a population code, where a population of neurons responds to a given input (e.g. a single variable) by particular spike trains encoding its value. In recent papers, the task of optimally computing said value from a population code was solved using a likelihood decoding principle. This decoding network consisted of neurons with specific, fixed synaptic connectivity to the encoding population. In this master's thesis this static weight connectivity will be replaced by a spike-timing dependent learning mechanism, and it will be shown that the resulting network features the same optimality while also adding a more general applicability in biologically plausible systems.

Abstract (German)

In der hier vorliegenden Arbeit werden zwei Ziele angestrebt: Zur Lösung des ersten wurde eine anwendungsbezogene Softwareumgebung erstellt, welche die Anwendung einer schon existierenden Methode für Objekterkennung und einfaches Tracking auf Daten einer vor kurzem entwickelten, hardware-basierenden Retina aufzeigen soll. Bei der visuellen Mustererkennung besteht die Problematik der Objekterkennung bereits seit mehreren Jahrzehnten und wurde zur Genüge mit verschiedenen Herangehensweisen und Methoden zu lösen versucht. Bei den dafür verwendeten Methoden treten oft schwere, wenn nicht unlösbare Probleme bei Implementationen mit biologisch plausiblen neuronalen Elementen auf. Ein alternativer Zugang zum Problem besteht darin, Lösungen mit einem bottom-up Ansatz zu entwickeln: Netzwerke mit relativ einfachen Neuronenmodellen für einfache, jedoch allgemeine Objekterkennung und Klassifikation umzusetzen. Obwohl die Benutzung von einfachen Netzwerkelementen die Komplexität der lösbaren Problemstellungen einschränkt, wird gezeigt, wie ein elementares neuronales Netzwerk nicht-triviale Problemstellungen auf effiziente Art und Weise lösen kann.

Im zweiten Teil der hier präsentierten Arbeit wird dieselbe Lernmethode in einem biologischeren Kontext angewandt. Im Gehirn wird wichtige Information mittels eines population codes störungssicher repräsentiert. Dabei wird eine Population von Neuronen von einem angelegten Eingangssignal (z.B. einer einzelnen Variablen) zur Ausgabe von spike trains angeregt, welche dadurch den Wert der Variable kodieren (population code). In bereits publizierten Arbeiten wird die optimale Möglichkeit, den originalen Wert der Variablen aus dem population code zu berechnen, mit dem sogenannten likelihood decoding Prinzip beschrieben. Das dekodierende Netzwerk besteht aus Neuronen mit spezieller, fixierter synaptischer Konnektivität zu der kodierenden Population. In dieser Masterarbeit wird diese statische Konnektivität durch einen vom spike-Zeitpunkt abhängigen Lernalgorithmus ersetzt und es wird gezeigt, wie das daraus resultierende Netzwerk dieselbe Optimalität wie auch allgemeinere Anwendbarkeit in biologisch plausiblen Systemen aufweist.

Acknowledgements

The project was designed to be a co-operation between the IGI at the TU Graz and the INI at the ETH Zuerich, and was mainly developed during a semester at the INI. The resulting software from the first part of this thesis was co-developed by Stefan Habenschuss, who also contributed theoretical extensions, which proved to be necessary for the task solution. The author wishes to thank all contributing persons, foremost Stefan Habenschuss, for his devotion to the project and readiness to happily participate in the weirdest discussions. Furthermore Dr. Shih-Chii Liu, for her patient explanations of the hardware retina and reluctance against the unsolvability of any task. Special thanks are in order for Prof. Maass, who not only made this fruitful co-operation possible and suggested the used method, but also proved to be a never-ending source of paper recommendations. Furthermore, too many people to be mentioned here (from both the INI and the IGI) gave contributing suggestions, provided a comfortable working environment and showed interest in the author's infinitesimal steps towards a scientific publication. Last but not least, the author wants to thank his parents for their loving support.

Helmut Puhr
Graz, 2011

Statutory Declaration

I declare that I have authored this thesis independently, that I have not used other than the declared sources / resources, and that I have explicitly marked all material which has been quoted either literally or by content from the used sources.

Graz,	
Place, Date	Signature

Eidesstattliche Erklärung

Ich erkläre an Eides statt, dass ich die vorliegende Arbeit selbstständig verfasst, andere als die angegebenen Quellen/Hilfsmittel nicht benutzt, und die den benutzten Quellen wörtlich und inhaltlich entnommene Stellen als solche kenntlich gemacht habe.

Graz, am	
Ort, Datum	Unterschrift

Contents

List of Figures

List of Tables

Part I.

Introduction

General Notes

In the vast field of computational intelligence, scientists strive to find the foundational building blocks of intelligence and reasoning. Put more accurately, one tries to formulate real-world problems in mathematical and machine-based descriptions and uses sophisticated algorithms to find solutions. Approaches differ in manifold ways, one differentiated field in the community focuses on building bridges between neuroscientific research (scientific study of the nervous system) and abstract, theoretically founded learning algorithms. Since the human brain is seen as the most advanced and universal learning system known, the interest in studying and reverse-engineering the central nervous system is obvious.

While many small-scale building blocks have been identified, due to their diversity and complexity, abstract generalization is delicate. Also, although many well-founded theories about large-scale behavior and the resulting view about learning and memory exist, various main questions cannot be explicitly answered yet.

The main contribution of the work presented here is the novel application of a previously developed learning mechanism to two different problem domains. The learning method, called Spiking Expectation Maximization (SEM), is theoretically well-founded and can be implemented using biologically plausible elements. The term **neural clustering** was chosen due to the fact that the behavior of the SEM algorithm could be interpreted as a clustering method. Clustering is an unsupervised learning procedure, where groups of similar examples are discovered in the input data. These so-called clusters are equivalent to neurons in a SEM architecture, as it will be demonstrated by later results.

In the first part, mainly during a stay at the ETH Zürich, the SEM method was integrated in a pre-existing software framework that utilizes neuromorphic hardware, in particular a spiking retina camera, as input devices. Using an implementation of the SEM algorithm in a neural simulator (PCSIM), several problems were investigated. Please note that the resulting software was in part co-developed by Stefan Habenschuss, who also contributed essential theoretical extensions to the SEM learning algorithm.

In the second part of this work, previous work that focused on optimal information representation and readout using neural structures was reproduced using the SEM method. However, since prior work was founded on several assumptions which limited biological applicability, the findings presented later show a more general and flexible approach, which still features the same optimality.

In Chapter I, a short introduction into the terms used and necessary context information is made, in Section 0.5 a short overview over the SEM and its extension is provided. In Chapter II the first main part of this thesis is developed, describing the hardware used, the software developed (and the environment used), task descriptions and results. The second main part is depicted in Chapter 0.25, starting with a detailed description, reproduction and discussion of previous work, followed by the results of SEM application and a short comparison between the original and extended SEM performance in this task. At last, a final discussion which summarizes the previous points and gives a short outlook on future work is provided in Chapter IV. Please note that due to the large quantity of information in some parts of this thesis a detailed view was sacrificed to allow a shorter and more general overview of the presented work.

Biological Neurons and Learning

0.1. Short History of Neuroscience

The history of neuroscience presented here is not deemed to be complete but is simply meant to be a coarse outline of historic findings which the author considers important and is based on discussions and presentations of historic papers in the lecture "Foundational Literature of Neuroscience" held by K.A. Martin and R.J. Douglas at the ETH Zürich in 2009.

In 1837 J.E. Purkinje first described neurons from the central nervous system (cerebellum) and identified a neuron's nucleus, later the development of the silver impregnation staining technique by C. Golgi (1873) made it possible to determine neurons as cells with a cell body and filamentous extensions (which he believed to be fused). Later in 1889 S.E. y Cajal found that nerve cells are independent elements, in 1897 C.S. Sherrington introduced the term synapse. In 1921, the pharmacologist O. Loewi confirmed chemical synaptic transmission and found the first neurotransmitter (which he called "Vagusstoff"). In an "accidental discovery", A.D. Adrian in 1928 proved the presence of electricity within nerve cells using a capillary electro-meter and a cathode ray tube as amplifier. In 1949 the author D.O. Hebb published the book "The Organization of Behavior", in which he described the "Hebbian theory" on learning: *When an axon of cell A is near enough to excite cell B and repeatedly or persistently takes part in firing it, some growth process or metabolic change takes place in one or both cells such that A's efficiency, as one of the cells firing B, is increased.* In a series of papers in 1952 A.L. Hodgkin and A.F. Huxley showed the dependence of axonal membrane potential on ions and were able to reconstruct the so called "action potential", the electrical entity of information in the brain, quantitatively.

0.2. Regarding Neurons and Synapses

This small summary of neural elements is based on (Gerstner, 2002). A typical neuron can be divided into three parts: dendrites, soma, and axon. Roughly speaking, the dendrites are the 'input device' that connects to other neurons, receives signals and forwards them to the soma. The soma can be seen as the 'central processing unit' that performs a non-linear processing step: If the total input exceeds a certain threshold, then an output signal is generated. The output signal is propagated by the 'output device', the axon, which forwards the signal to other neurons. The junction between two neurons is called a synapse, the sending one is called the presynaptic neuron and the receiving one accordingly the postsynaptic neuron. The basic entity of information that is electrically exchanged between neurons is a short pulse of an amplitude of roughly 100 mV and a duration of a few milliseconds. The so called *action potential* or *spike* does not change in form while it is propagated and its amplitude is about constant. When considering how information is encoded in a series of spikes (which is referred to as *spike train*), since the amplitude or duration should not change, only the number of spikes or the times of occurrence are of interest. Also since, after firing, neurons have a so called refractory period, in which they are almost unable to spike even if subjected to high input, action potentials are in general well separable.

The site where the axon of a presynaptic neuron makes contact with the dendrite (or soma) of a postsynaptic cell is the synapse. The most common type of synapse in the vertebrate brain is a chemical synapse, which transmits a presynaptic action potential chemically (using neurotransmitters) by triggering a change in the postsynaptic membrane potential due to ionic flux (later referred to as postsynaptic potential). Such potential changes can be positive (excitatory postsynaptic potential - EPSP) or negative (inhibitory - IPSP), depending on the neurotransmitters and receptors involved.

To generalize neuron behavior is hard, since different types of neurons exists and an extensive model that incorporates such diverse behavior is rather complex and requires time-consuming computation. For example, the model by Hodgkin-Huxley (from (Hodgkin, 1952)) includes electro-physiological properties of a giant squid axon, which makes it quite accurate but also implies that various parameters have to be set correctly to ensure sensible behavior. However, cortical neurons in vertebrates exhibit a much richer repertoire of electro-physiological properties than the squid axon, which is mostly due to a larger variety of different ion channels (Koch,1999).

Detailed neuron models which can reproduce such behavior to a high degree of accuracy exist, but because of their intrinsic complexity these models are difficult to analyze and to compute. For these reasons, simple phenomenological spiking neuron models are more popular, which can incorporate accurate spiking behavior using less complex parametrization.

One of the more common models is the *leaky integrate-and-fire* model (LIF), in which an input current $I(t)$ drives a parallel resistor and capacitor (RC) circuit. The input current (which can be interpreted as a synaptic input, which can be weighted) charges the capacitor C and, if the voltage crosses a certain threshold, an output spike is generated. Formally, one has to calculate the membrane potential $u(t)$ from a differential equation, where $\tau_m = RC$ is the membrane time constant.

$$\tau_m \frac{du}{dt} = -u(t) + RI(t) \tag{0.1}$$

Note that the integrate-and-fire model does not explicitly describe the form of an action potential, spikes are only events characterized by the firing point in time $t^{(f)}$, which is defined by a threshold criterion (exceed threshold δ).

$$t^{(f)} : u(t^{(f)}) = \delta \tag{0.2}$$

There exist multiple important extensions of the LIF neuron model, but a detailed discussion was omitted since this text should only give a general introduction to the topic. Please note that the LIF model is mostly chosen due to its simplicity and is inferior to others on account of biological realism, please refer to (Izhikevich, 2004) for a detailed discussion. A commonly used generalization of the LIF model is the so called *spike response model* (SRM), in which the formulation of the equations differs by not using differential equations but instead formulating the membrane potential $u(t)$ as an integral over the past. Also, it models a refractory period in a detailed way, in contrast to the LIF model.

In the SRM, the state of a neuron i is described by single variable u_i, which is the resting potential $u_{rest} = 0$ in the absence of spikes. An incoming spike alters u_i, which will return to the resting potential over time. If several incoming spikes drive u_i over a threshold δ, an output spike is triggered. After firing at time $\hat{t}_i$, the evolution of u_i is calculated using the following equation.

$$u_i(t) = \eta(t - \hat{t}_i) + \sum_i w_{ij} \sum_f \varepsilon_{ij}(t - \hat{t}_i, t - t_j^{(f)}) + \int_0^\infty \kappa(t - \hat{t}_i, s) I^{ext}(t - s) ds \tag{0.3}$$

The function η describes the form of the action-potential an after-potential, w_{ij} is the synaptic weight of current neuron i and presynaptic neuron j. Using the function ε one describes the response to an incoming spike, κ is the linear response of the membrane potential to an input current, while I^{ext} is an external driving current. Using the response kernels η, ε and κ one can describe the effect of spike emission and spike reception on the variable u_i and closely fit the model to experimental data. For performance benchmarks of the model, please refer to (Jolivet, 2008).

0.3. Learning Through Synaptic Plasticity

As mentioned earlier in the historic outline, the Hebbian learning theory states that a change in synaptic weights occurs when pre- and postsynaptic neurons fire simultaneously in a given time window, which is often abbreviated inaccurately as “Cells that fire together, wire together”. However, pure Hebbian learning is unstable, since negative weight changes (synaptic depression) are also necessary to avoid self-reinforcing cyclic connections in recurrent networks.

To extend the basic idea from Hebbian learning, one needs to incorporate the exact timing between pre- and post-synaptic spikes in the learning algorithm, which is called *spike-timing dependent plasticity* (STDP). To be more precise, a pre-synaptic spike occurrence before post-synaptic spike induces a positive change in synaptic efficacy, while in the opposite order, leads to a negative weight change. This relationship, with $\triangle t = t_{post} - t_{pre}$, is shown in Figure 0.1.

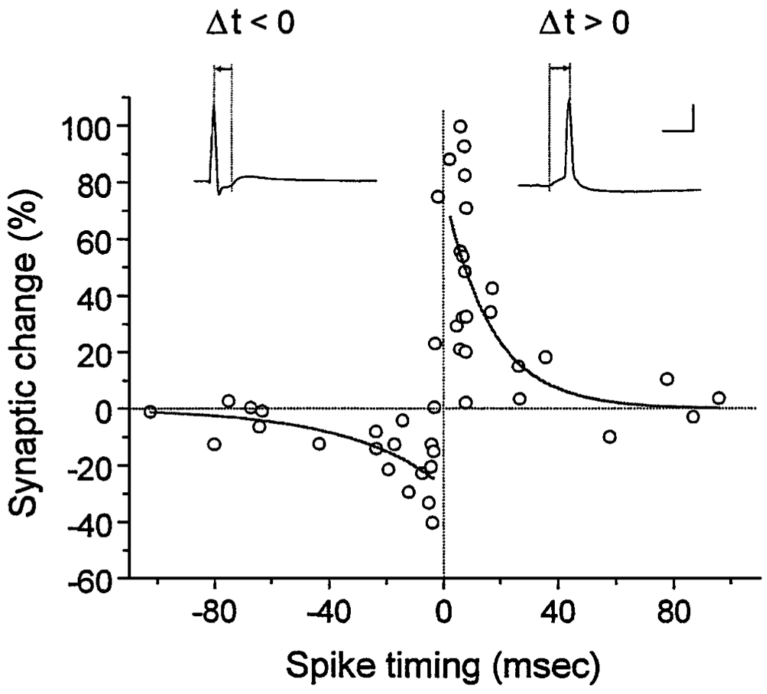


Figure 0.1.: Weight change over spike time relation from (Bi, 2001)

However, recent research suggests not only that different STDP relations exist, but can also be modulated by other mechanisms (please refer to (Dan, 2006) for a summary).

0.4. Winner-Take-All - Function Through Competition

A group of neurons can be formed into a Winner-Take-All (WTA) circuit, which ensures if one neuron fires, all others are suppressed. In its purest form, WTA means that only the neuron with the strongest response is active and all others are silenced (e.g. using lateral inhibition). Other forms exist, notably soft-WTA (where the other neurons are only subdued and may also fire), k-WTA (where k neurons are allowed to fire) or probabilistic WTA (where the activation strength of each neuron is used to form a probability distribution, from which the winner is sampled). By forcing neurons to compete with each other for activation one builds a powerful computational module, which was proven to be able to approximate arbitrary continuous functions (using soft winner-take-all) (Maass, 2000). Although it is still disputed how an exact realization of an WTA circuit in the brain can be created, WTA mechanisms are viewed as components of principal features of cortical circuits (Douglas, 2004).

0.5. Population Coding

The motivation behind population coding arises from the simple question how information is encoded in the brain. A single neuron can, since its firing is noisy and error-prone, hardly be a reliable basis of representation of information. Therefore, considering a population of neurons would lead to a more stable representation (Pouget, 2000). When considering the importance of the general notion of a variable encoded in a response from a population of neurons, one should remember that sensible processing method requires a stable information representation. Experimental results suggest that the brain makes extensive use of this coding strategy, e.g. encoding movement direction in the motor cortex (Georgopoulos, 1986) or orientation in the striate cortex (Vogels, 1990).

Theory

0.6. Probability Theory

In (Jaynes, 2003), probability theory is described as an extended theory of logic, making it possible to describe logical but uncertain relationships of events. According to (Bishop, 2006) (on which this short summary is based), probability theory provides a consistent framework for the quantification and manipulation of uncertainty and forms one of the central foundations of pattern recognition. A random variable is a variable whose value is unknown, representing for example the outcome of a later experiment or potential values of a variable whose already-existing value is uncertain. Assume two random variables X, with possible outcomes being x_i where $i = 1,...,M$, and Y with according y_j where $j = 1,...,L$. From a number of trials N where both X and Y are sampled, calculations using the following rules can be made. The sum rule describes how in a joint distribution one random variable can be summed out (marginalized).

$$p(X = x_i) = \sum_{j=1}^{L} p(X = x_i, Y = y_j) \tag{0.4}$$

For the second rule, conditional probability needs to be defined. If one wants to describe only the events where $X = x_i$, then the fraction of such instances for which $Y = y_j$ is written as $p(Y = y_j|X = x_i)$. The following product rule describes the relationship between joint and conditional probability distributions.

$$p(X = x_i, Y = y_j) = p(Y = y_j|X = x_i)p(X = x_i) \tag{0.5}$$

Using these two fundamental rules one can obtain a relationship between conditional probabilities, the *Bayes theorem*, which plays a central role in pattern recognition and machine learning.

$$p(Y = y_j|X = x_i) = \frac{p(X = x_i|Y = y_j)p(Y = y_j)}{p(X = x_i)} \tag{0.6}$$

Probabilistic reasoning was introduced by (Pearl, 1988) to the field of machine learning. Its main contribution was the theory of a Bayesian network, which is a representation of random variables and their conditional dependencies by a probabilistic graphical model. Nowadays, machine learning heavily relies on statistics and probability theory, the interested reader is referred to (Bishop, 2006) for a usage-oriented introduction.

0.7. Probability Distributions

Probability distributions are used to describe a range of possible values that a random variable can take and the probability that the value of a random variable is within a subset of that range. Using an appropriate probability distribution one can model or at least approximate a random variable using only a few parameters.

0.7.1. The Gaussian Distribution

Also called the normal distribution, in case of a single real-valued variable x, the Gaussian distribution is defined by mean μ and variance σ^2.

$$P(x) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(x-\mu)^2}{2\sigma^2}} \tag{0.7}$$

0.7.2. The Poisson Distribution

This discrete distribution defines probability of a number of events x occurring in a given period of time, if these events occur independently with a known average rate λ.

$$P(x|\lambda) = \frac{\lambda^x e^{-\lambda}}{x!} \tag{0.8}$$

0.8. Expectation Maximization

A parametric model is a distribution which is controlled by a number of finite parameters, e.g. mean and variance in a Gaussian or rate in a Poisson case. Latent variables are unobservable variables which are inferred from directly observed ones. Using this terms, the expectation-maximization (EM) algorithm can be described as a method for finding maximum likelihood or maximum a posteriori (MAP) estimates of parameters of a parametric model, where the model depends on latent variables. EM consists of two steps, which are iteratively executed: The expectation step, which uses the current estimate of the latent variables to compute the expectation of the log-likelihood, and the maximization step, which computes the parameters which maximize the previously calculated expected log-likelihood. For a solid presentation of this topic please refer to (Kay, 1993).

0.9. SEM

In previous work of (Nessler, 2010) it was shown how spike-timing-dependent plasticity (STDP) is able to approximate a stochastic on-line Expectation-Maximization (EM) algorithm. STDP in conjunction with a stochastic soft winner-take-all (WTA) circuit induces spiking neurons to represent the input spike patterns in form of their synaptic weights. In this specific algorithm, STDP is able to approximate a stochastic online Expectation-Maximization (EM) algorithm for modeling the input data. This method was titled Spiking Expectation Maximization (SEM) to emphasize the implementability using biologically plausible network elements.

All “external“ input variables, denoted by $x_1,...,x_m$, are encoded in binary input values $y_1,...,y_n$, which ensures the number of input spikes to the network being a constant. If x_i is binary, two y variables are required for this encoding, one associated with the value of x_i, the other with its inverse. In each time step the input spikes are processed by each of the z-neurons $z_1,...,z_K$ by calculating its membrane potential u_k. From this distribution of u_k's, a winner is selected and is allowed to fire and thus updates its synaptic weights according to the learning rule. Figure 0.2 shows the described SEM network.

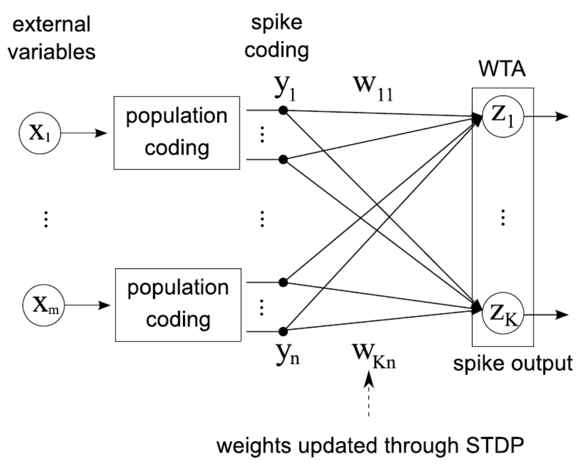


Figure 0.2.: SEM network

The membrane potential of a neuron z_k is defined as

$$u_k(t) = \sum_{i=i}^{n} w_{ki}\hat{y}_i(t) + w_{k0} \tag{0.9}$$

with $\hat{y}_i(t)$ being 1 if y_i had fired in the defined previous interval, the weights $w_{ki}(t)$ denote the current synaptic weight from input i to output k, and $w_{k0}(t)$ constitutes a bias weight. A Poisson process facilitates the firing of one of the z-neurons, which was selected from a soft-max distribution defined as

$$p(z_k \text{ fires at time } t|\{y_i(t)\}) = \frac{e^{u_k(t)}}{\sum_{l=1}^{K} e^{u_l(t)}}. \tag{0.10}$$

If one assumes a simple STDP learning curve as in Figure 0.3 one can implement the learning process with a Hebbian learning rule and use the SEM algorithm in a non-spiking network without loss of exactness.

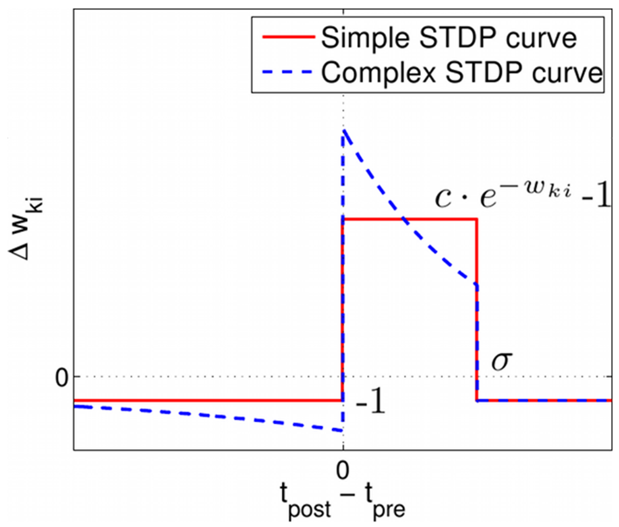


Figure 0.3.: SEM STDP learning curve

The selected output neuron emits a single spike and updates its synaptic weights according to a weight-dependent Hebbian learning rule.

$$\Delta w_{ki} = \begin{cases} \eta(e^{-w_{ki}} - 1) & \text{if } y_i = 1 \text{ and } z_k = 1 \\ -\eta & \text{if } y_i = 0 \text{ and } z_k = 1 \\ 0 & \text{if } z_k = 0 \end{cases} \tag{0.11}$$

$$\Delta w_{k0} = \begin{cases} \eta(e^{-w_{k0}} - 1) & \text{if } z_k = 1 \\ -\eta & \text{if } z_k = 1 \end{cases} \tag{0.12}$$

The log-likelihood of a given the input x and synaptic weights w can be calculated using the following formula.

$$\log p(\mathbf{x}|\mathbf{w}) = -\log K + \log\left(\sum_k \exp(\mathbf{w}_k^T \mathbf{x})\right) \tag{0.13}$$

0.10. SEM Extension

One problem using the original SEM method was that in most cases the input is not binary, therefore requiring an additional pre-processing step. In a simple case, a threshold operation solves the problem, but will introduce additional input distortion. Also, doubling the number of input variables is a computationally unsound practice in any case, if it can be avoided.

This problem arises from the property that SEM is limited to categorical variables (variables whose values are limited to categories), but a more natural choice would be continuous variables or modeling Poisson distributed inputs (Gerstner, 2002).

As it was shown in (Habenschuss, 2010), it was possible to extend the SEM method to any input distribution which belongs to the class of the exponential family (e.g. Gaussian, Poisson) by introducing an additional normalization term in the calculation of the membrane potential and altering the learning rule accordingly. Also, extensions to biologically realistic EPSP shapes are now covered in the theory, but were not used in this thesis.

0.10.1. Poisson Based SEM

Calculation of the membrane potential:

$$u_k = \mathbf{w}_k^T \mathbf{x} - \sum_i e^{w_{ki}} \tag{0.14}$$

Learning rule:

$$\Delta w_{ki} = \eta(x_i e^{-w_{ki}} - 1) \tag{0.15}$$

Calculation of the log-likelihood given input and weights:

$$\log p(\mathbf{x}|\mathbf{w}) = -\log K - \sum_i \log(x_i!) + \log\left(\sum_k \exp(\mathbf{w}_k^T \mathbf{x} - \sum_i e^{w_{ki}})\right) \tag{0.16}$$

0.10.2. Gauss Based SEM

Calculation of the membrane potential:

$$u_k = \mathbf{w}_k^T \mathbf{x} - \sum_i \frac{w_{ki}^2}{2} \tag{0.17}$$

Learning rule:

$$\Delta w_{ki} = \eta (x_i - w_{ki}) \tag{0.18}$$

Part II.

Application to Neuromorphic Hardware

General Notes

In machine learning and computer vision, the task of object recognition is to find objects in a given image or video sequence. Object recognition focuses on an object's identity, tracking uses the given identity to estimate the current or future positions of the object. While humans and animals are able to solve such tasks in a highly proficient way, computer based systems are easily troubled when objects are shown from different viewpoints, are occluded or transformed by rotation, translation or scaling. While the brain seems to be able to easily generalize few pictures of an object into a stable invariant representation, how this can be done using neural structures is yet unclear. Still, in machine vision, a large quantity of robust methods to solve such tasks exist (e.g. Edge matching, SIFT, Bag of words).

Since a large complexity in the given problem requires sophisticated learning algorithms, implementing such algorithms using locally constricted and noisy neurons is a difficult problem. For this reason, the work presented here focuses on comparatively small tasks but shows under which conditions the chosen method can solve a number of tasks in a biologically plausible context. Another point which ruled out most established methods was a real-time and on-line computation requirement, on-line in this context being the capability of the method to produce immediate results after each input step rather than only after a long training period using a large quantity of training data.

Using sophisticated object classification generally requires large quantities of training data, so that all transformations that the target object can be exposed to are covered by input data to extract a sufficiently complex model that reflects such behavior. Here, a crucial distinction can be made between supervised and unsupervised training data. Supervision in this context requires the existence of an additional meta-information describing the content of the training data, e.g. object position and identity in each input frame. From this information, complex features (properties of the input) can be extracted and are used later for classification of input that lacks supervised information. Since in supervised classification large quantities of training data have to be correctly labeled, which requires significant effort, another approach is to use unsupervised training data. Being closely related to density estimation in statistics, numerous methods have been developed (e.g. Clustering, ICA, Artificial Neural Networks).

At the INI, especially in the hardware group, the main focus is hardware development and applications using such hardware. The requirements were that the method used should be integrated into the already existing software framework (jAER), use online computation and, if possible, should be able to run in real-time on a standard workstation.

The main task was to investigate methods for object classification and tracking using neuromorphic hardware, mainly the spiking retina camera, using biologically plausible learning algorithms. Unsupervised methods were to be preferred due to the large effort of labeling large amounts of training data.

As input device, the hardware retina (DVS128) was used to generate spike trains, where each spike corresponds to a luminance change of an input pixel. This input was processed by a sophisticated software system, which was developed partly in Java, partly in C++.

Hardware

0.11. DVS128

Contents of this chapter have been published in (Delbruck, 2008) unless explicitly noted.

Conventional camera systems are typically frame-based, meaning that color/luminance of the input area are measured for a time interval and the measured frame is sent to later stages. Another approach was taken in development of the dynamic vision sensor (DVS) which was conceived during the CAVIAR project. Basically an array of 128x128 pixels measures the change of luminance I and, if the change succeeds a threshold T, produces an output-event. The change is compared to a threshold in the log-space:

$$|\Delta log I| > T \tag{0.19}$$

These output events consist of x,y addresses, a time-stamp, and a flag denoting a positive (ON) or negative (OFF) change in luminance.

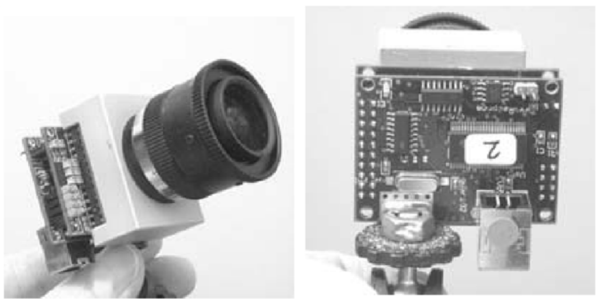


Figure 0.4.: DVS128 from (Lichtsteiner, 2008)

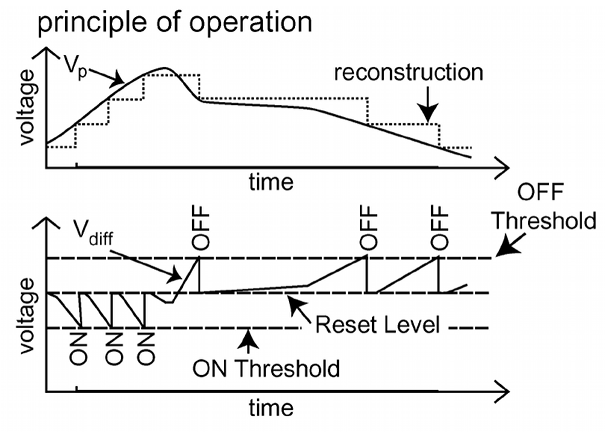


Figure 0.5.: DVS128 principle of operation

Using this event-based input for this project had several advantages, e.g. the wide dynamic range of operation (less dependence on scene illumination), the timing accuracy and the reduced input quantity. Also the output of the DVS128 can be quite easily processed by a standard workstation using the already developed jAER framework.

Setup

0.12. Overview

The spiking retina camera is connected to a workstation via USB, and using an existing driver the resulting spike trains are processed in jAER. Since it was necessary to integrate into an already existing Java environment (jAER), any used or developed software had to be callable from within the Java VM. For several reasons the implementation of the learning algorithm was not done in Java. For one it seemed futile to implement yet another neural simulator in the little time available, and existing simulators in Java were either unsuitable or poorly documented.

Based on (Brette, 2007), the choice of a software framework for the implementation of the learning algorithm fell on PCSIM for several reasons. The most important reason was its efficient processing core, which allowed the simulation of large neuron populations even on small workstations. Also the usage from the Python programming language was convenient, since it would not only make the software accessible to other software packages but would also speed up testing of implementations (fast development cycles, extensive math and visualization library).

0.13. Java Framework: jAER

Section is based on (Delbruck, 2008).

Implemented in Java, the jAER project allows processing, capturing and playback of address-events from multiple hardware sources. These events, which are organized in packages for efficiency reasons, are processed by filters (serial or parallel). An automatically-generated software GUI interface allows online-configuration of hardware or filter parameters as well as visualization of the output. Various existing filters can be used for pre-processing or evaluation (e.g. background activity filter for noise reduction, cluster tracker as translation invariance filter).

0.14. PCSIM

Section is based on (Pecevski, 2009).

0.14.1. Introduction

The *P*arallel *C*ircuit *Sim*ulator is a neural simulator for large scale neuron networks. PCSIM derives its core features from CSIM, which was written in C++ and used via a Matlab interface for convenient usage. When using Matlab, though advantages like excellent result plotting abilities exist, some main features of sophisticated programming languages are missing. Especially computational efficiency (e.g. call by reference), true object-oriented design and interfacing with already existing software frameworks are difficult, if not impossible, to achieve. When using Python as a new high-level programming language for PCSIM functionality, several of these issues can be avoided.

The C++ core functionality of PCSIM is exposed to Python programs by generation of wrappers, which allows usage of C++ classes in Python. Also a port of PCSIM functionality into Java has been made available by Dipl.Ing. Natschläger, which makes parts of the functionality usable from a Java environment via JNI interfaces. Basically, JNI-compatible wrappers of the exposed C++ classes are generated. From these wrappers, Java counterparts are generated and compiled, which can be used if the "libjpcsim" library is loaded previously.

0.14.2. Architecture

The main compartment of PCSIM consists of the C++ part with the core simulation engine, the simulation objects (e.g. neurons, synapses) and the network generation functions. This environment is confined in a C++ library called 'libpcsim'. Using the boost.python library (and others) parts of 'libpcsim' are made accessible to be used within Python code as the Python extension module 'pypcsim'. Please refer to (Pecevski, 2009) for a more detailed discussion and usage.

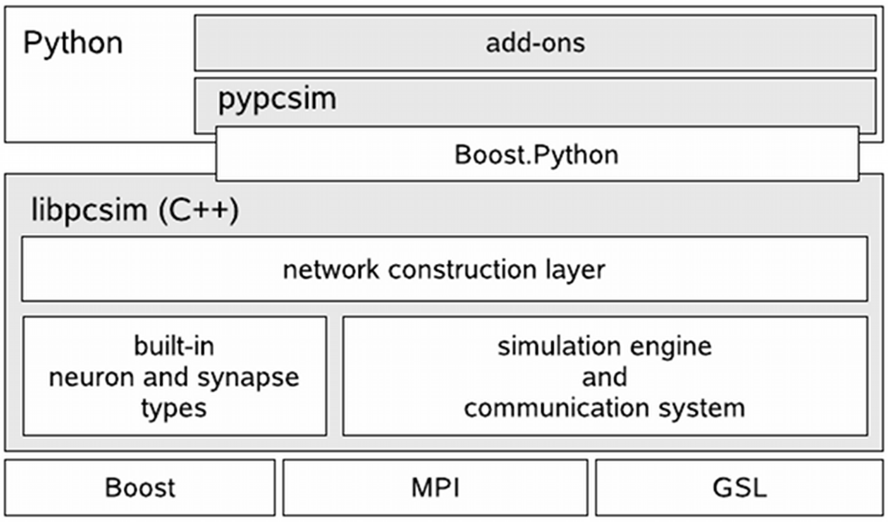

Python
add-ons
pypcsim
Boost.Python
libpcsim (C++)
network construction layer
built-in neuron and synapse types
simulation engine and communication system
Boost
MPI
GSL

Figure 0.6.: PCSIM overview

0.14.3. Python Interface

Using the Boost.Python library, C++ functionality can be interfaced with Python and, if desired, overwritten by Python code. Using the Py++ package, the repetitive step of generating the Boost.Python code can be automated by writing a set of high-level rules which define the latter conversion process. Py++ builds on a GCC-XML parser which outputs a XML representation of the C++ code, which is used to create a Python program and the required Boost.Python code. The resulting code is compiled and linked with the 'libpcsim' library to create the Python extension module 'pypcsim'.

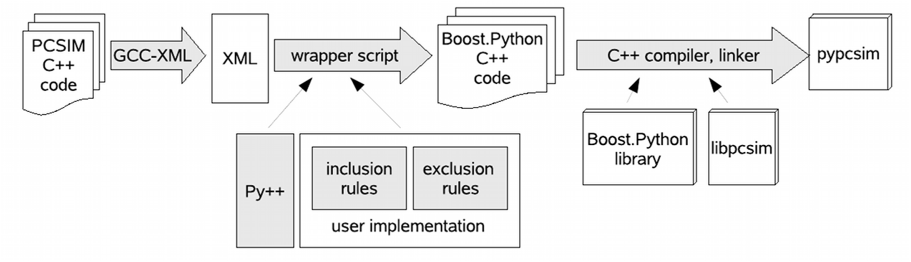


Figure 0.7.: PCSIM Python generation

0.14.4. Build Guide

Since the current PCSIM environment depends on several open-source libraries, the installation process is non-trivial under current linux systems. When PCSIM was developed, these libraries formed closely interlinked dependencies which required specific, and now outdated, library versions. Since open-source libraries are not always backward compatible, using newer versions almost always corrupts the build process. Additionally, these errors almost always occur at a very late stage, foremost when the pcsim_test target is linked to generated library. Therefore, the installation requirements have to be fulfilled, otherwise almost uninterpretable errors will occur.

bison-2.4.tar.gz
boost_1_34_1.tar.gz
cmake-2.6.4.tar.gz
cppunit-1.12.1.tar.gz
doxygen-1.5.3.src.tar.gz
elementtree-1.2.6-20050316.tar.gz
flex-2.5.35.tar.gz
gccxml_cvs.tar.gz
gsl-1.10.tar.gz
mpich2-1.2.1.tar.gz
Py++-0.9.5.zip
pygccxml-0.9.5.zip
tinyxml_2_5_3.zip
xerces-c-src_2_8_0.tar.gz

Table 0.1.: Necessary libraries for PCSIM

Installing the required libraries can be quite painstaking, since a system-wide installation via an easy-to-use packet manager is out of the question. Even if such outdated libraries could be installed, conflicts would arise when other programs require newer versions (e.g. boost). The option of manually installing the source code, compiling and installing to system folders by creating appropriate symlinks is possible, but most users would prefer an easier way.

For these reasons, the author chose a completely local installation in a user directory, with rather small requirements to the underlying linux system. This "sink included" installation creates a complete PCSIM environment in a local folder and changes the necessary system paths to prefer the local installation over the system-wide paths using a shell script. The basic idea is to resolve as many dependencies as possible by compiling all necessary libraries from scratch and setting up a complete environment in a local user folder. This way, the only requirements to the Linux OS are a compiler version 4.1 or 4.2 and Python 2.5.

PCSIM Environment Build

By calling “env_setup.sh“ the user switches the path variable so that programs from the pcsim environment installation path are preferred to system programs, also library paths are changed so that the cmake build system is able to find the appropriate libraries in the local directory structure. This script **must** be executed prior to any pcsim usage.

ENV_DIR	base directory of the PCSIM environment
ENV_INSTALL_DIR	installation directory (substitute for /usr)
LD_LIBRARY_PATH	C++ library paths
PYTHON_PATH	Python packages paths
JAVA_HOME	Java JRE path
PCSIM_HOME	PCSIM branch path

Table 0.2.: Variables for local PCSIM installation

All necessary libraries are supplied as compressed archives (folder archive) and can be extracted into the ”ENV_DIR” folder with the shell script ”extract_env.sh“. The environment can later be built using ”build_env.sh“, and can be deleted by executing ”clean_env.sh“.

It is recommended to build each library by executing the appropriate lines from the shell script ”build_env.sh“ rather than to execute the complete script, to detect if for some reason the build process of a library failed. If it succeeded, all libraries and binaries are installed into the appropriate ”ENV_INSTALL_DIR” subfolder.

PCSIM Build

The build system makes use of a open-source make system called CMake. All necessary information about a specific build process is stored in platform and compiler independent configuration files, using this information CMake creates platform native make files for a specific compilation environment.

In each directory a CMakeLists.txt file is required, which defines the complete build project configuration such as subdirectories, source files, required libraries, variables, resource paths (include files, libraries) and target configuration (executables, libraries).

When “env_setup.sh” has been called, a branch of PCSIM can be built by the script “build.sh“:

```
env CPPFLAGS=”-I${ENV_INSTALL_DIR}/include” LDFLAGS=”-L${ENV_INSTALL_DIR}/lib”
        python build.py install –prefix ${ENV_INSTALL_DIR}
```

Note that if changes were made to the Java interface, executing the already supplied Python file ”generate_java_wrapper.py“ in folder ”java“ is required, as stated in the supplied PCSIM documentation. To clear execute the command ”python build.py wipe“, which should be done after an incomplete build process or if new classes have been added to the C++ code.

In every folder where cmake should perform an action a file called ”CMakeLists.txt“ resides which defines the operations to be performed. In a branch base directory, this file defines the project name, include a file ”config.cmake“, defines the directories to be used and a custom build target.

In the rather large file ”config.cmake“ **every** required dependency is checked, all libraries are searched in defined system paths (which are masked in the ”sink-included“ environment) and the correct paths are set via ”LINK_DIRECTORIES“. Also the Python and Java paths are checked and set to the system specific values. If a requirement is not fulfilled the build process stops and prints an error message.

0.15. Visual Simulator: SpikeSim

One of the first problems of the project was lack of high-quality input data from the retina camera. Since the effort of producing such input was quite time-consuming, the need for a visual simulator that could produce such input arose. While this generated input could of course not substitute real-world input, the means to produce large quantities of specified, uncorrupted input data proved to be advantageous, especially during testing of new implementations.

As first tasks, one or multiple objects (bitmaps) should translate in a 2D plane, and be reflected at defined borders. To ensure the easy setup of new experiments without changing C++ code, all configurations are defined in XML files. One of these files is read in at startup in the simulator, the objects are created and moved for a specific number of time steps. After each step, a memory-screenshot is taken and stored in a buffer. After simulation, according to the configuration, the collected frames are processed (e.g. scaled or differentiated in time) and saved as file (either as Matlab file or as AER spike file). An example of 2 objects (cross and circle) with accentuated movement is shown in Figure 0.8.

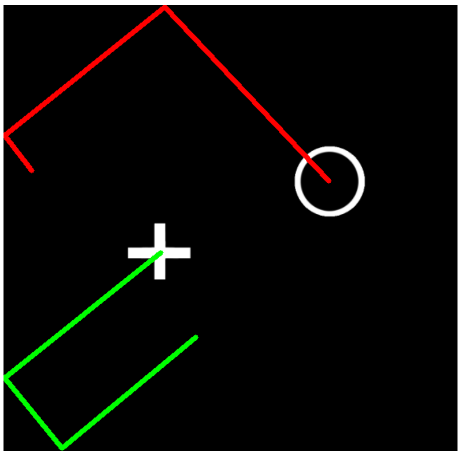


Figure 0.8.: SpikeSim frame with 2 objects

0.15.1. Implementation

SpikeSim was written in C++, using the open source 3D engine Ogre. Its first implementation was to translate objects in 2D, but for later purposes a solid 3D engine seemed appropriate. Objects are implemented as simple sprites with a texture, consisting of a PNG bitmap with transparent background, which is moved in 3D space. The viewpoint camera is placed “on top“ of the scene, face downwards. Therefore an object has initial position, scale, velocities, rotation and rotation speed and so on. Movement in the 3rd dimension (y in Ogre) changes the size of an object from the camera’s point of view. It would be an easy matter to extend SpikeSim to real 3D objects, but for this project’s toy tasks planar objects were sufficient. The simulation is performed for a specific number of time steps. In each step, the objects in the scene are moved according to the current movement speed and reflected (inverse speed) if a border is hit. Note that the borders form a cube. From this scene a snapshot it taken and, since it resides in the GPUs memory, transferred to main memory and stored in a buffer. After simulation, this buffer is processed, and if stated in the configuration, the frames are differentiated over time. Also, they can be scaled (with or without interpolation) and noise can be added.

After this, the frames are either stored in a Matlab file (MatFileWriter) or as a AER file (SpikeFileWriter). As a Matlab file, the resulting frames are simply stored in a matrix and written. As an AER file, a more complicated process is necessary: For each frame, in a defined time window ”frame_length“ the resulting number of spikes is generated by a Poisson process, then each spike is randomly placed within the current time window. The resulting list of events is written as an AER file.

0.15.2. Configuration

The implemented simulation capabilities are best reflected by inspection of the SpikeSim program parameters.

%Spike simulation configuration file	comment
<SimEnvironment>	begin of environment configuration
write_file=true	write to file flag
filewriter=SpikeFileWriter	used file writer
%filewriter=MatFileWriter	commented file writer
<SimTimer>	world movement timer
time_step=0.5	time step
frame_update=1	update every n-th frame
</SimTimer>	end world movement timer
<MatFileWriter>	Matlab file configuration
num_frames=10000	number of frames
w=128	resolution width
h=128	resolution height
filename=/path/frames.mat	filename
interpolate=false	interpolation flag
differentiate=false	if frames should be substracted
</MatFileWriter>	end of Matlab file configuration
<SpikeFileWriter>	AER file configuration
num_frames=10000	number of frames
frame_length=10000	length of one frame in us
w=128	resolution width
h=128	resolution height
filename=/path/data.dat	
interpolate=false	interpolation flag
threshold=1.1	change of luminance threshold
noise_prob=0.001	uniform random noise probability
noise_min=0	noise minimum
noise_max=15	noise maximum
</SpikeFileWriter>	end of AER file configuration
<SimWorld>	world configuration
<SimObjectBorder>	border configuration
size=100	size for each coordinate
epsilon=1.0	reflection threshold
interaction=reflect	reflect or teleport
</SimObjectBorder>	end of border configuration
%#include=conf/bear.conf	include several possible object configurations
%#include=conf/penguin.conf	
%#include=conf/noise.conf	
#include=conf/circle.conf	
%#include=conf/cross.conf	
%#include=conf/rectangle.conf	
</SimWorld>	end of world configuration
</SimEnvironment>	end of environment configuration

Table 0.3.: Simulation.conf

One possible configuration of an object is included in an optional configuration file, the sprite texture must be supplied as an Ogre material resource. Multiple objects can be added by including several definitions of these SimObjects. The circle, cross and rectangle objects consist of the according geometrical structures in solid white, while bear and penguin are toy bitmaps. All bitmaps are black-white (color could be added, but was, in conformity with the spiking retina structure, omitted for this project) bitmaps with transparent background.

<SimObjectCross>	SimObject definition
name=cross	object name
material=cross	Ogre material name
pos_x=20	starting position
pos_y=10	
pos_z=10	
vel_x=-2	starting velocity
vel_y=0	
vel_z=1.8	
size=1	object size
rot=0	starting velocity
rotvel=0	change of velocity
vel_rand_min_x=0	random velocity parameters
vel_rand_max_x=0	
vel_rand_min_y=0	
vel_rand_max_y=0	
vel_rand_min_z=0	
vel_rand_max_z=0	
</SimObjectCross>	SimObject definition end

Table 0.4.: cross.conf

With these parameter values, an AER file was generated and played back using jAERViewer, the resulting (low-pass filtered) output images is shown in Figure 0.9. The on-events are shown as white, the off-events as black pixels. Note that, according to the movement direction of the object, the front edges show as on-events while back edges generate off-events, and background noise generates both types.

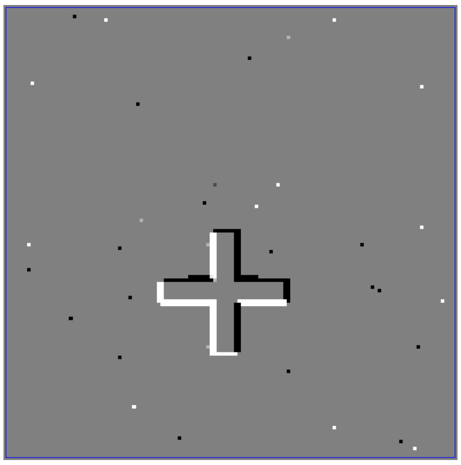


Figure 0.9.: SpikeSim AER result in jAERViewer

Implementation

0.16. Introduction

The implementation of the developed software consists of a Java and a C++ part. In the Java part, the already existing jAER framework was used as a basis for running an event filter, which used the C++ functionality to process the events collected from the retina camera. In C++, the entire network is constructed, simulated for each time step, and observed using several measures (e.g. output, synaptic weights). While processing, the filter read out output and resulting internal values for display purposes in a graphical user interface (GUI) in Java.

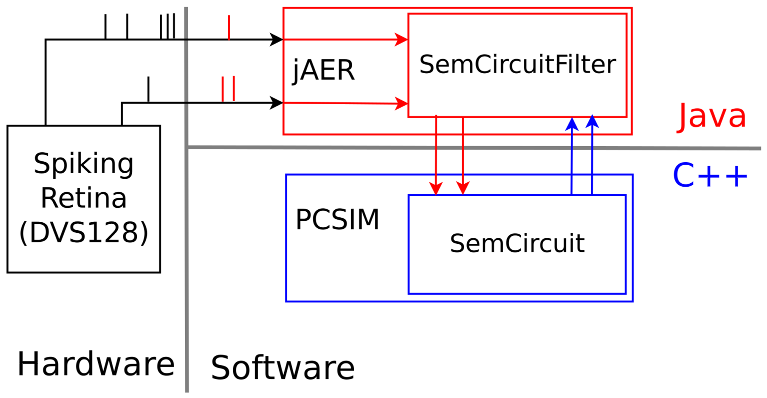


Figure 0.10.: General architecture

Two different SEM implementations exist, which serve different purposes. The first implementation (SemFilter) uses the native PCSIM SimObject structure for each neuron and synapse and is therefore, in principle, usable in a mixture with other PCSIM objects. The main problem with this implementation lies in the immense memory usage and its therefore limited applicability in real-time tasks. Especially for larger networks, which consist of more than a hundred neurons, memory of a common workstation (e.g. 2 GB) is not enough to keep the entire network in RAM.

For this reason, an optimized implementation (SemCircuit) was developed, which features faster processing (without loss of exactness) and less memory consumption. One SemCircuit encapsulates an entire SEM network consisting of several types of SemCircuitElements (neuron types, synapses, spike response functions, spiking mechanisms), layers of neurons (input layers, layers of z-neurons) and a defined connection structure. Each network definition resides in a XML document, where all of the previously mentioned components are defined and used to assemble the later network behavior. One of most striking features of the implementation is the modular definition of all network components, which allows easy generation of sophisticated experiment setups just by changing the defined XML structure of the network.

For this filter, a rich GUI was developed to control the simulation, view and modify parameters and to visualize synaptic weights, performance measures, output spikes et cetera. Also a filter called SemCircuitFilterRL was developed which uses a reinforcement-learning performance measure to modulate the learning rate of neurons to impress desired behavior.

0.17. C++

This section should give a general notion of the implementation of the SemCircuit components, as well as a more detailed discussion of the interaction of these components, which is required for a more sophisticated understanding of the simulation process. Considering presentation structure, a bottom-up approach was chosen. One of the main goals of the implementation was to develop a modularly extend-able framework, which should allow the creation and configuration of a network in a startup procedure using an XML configuration document.

Interfacing with the Java counterpart, the main class is called SemNetworkManager, which loads an XML configuration file and creates, manages and destroys all network elements and supporting data structures. Depending on the XML file, several SemNetworkElement instances are created which interact to form a SemCircuit, as depicted in Figure 0.11.

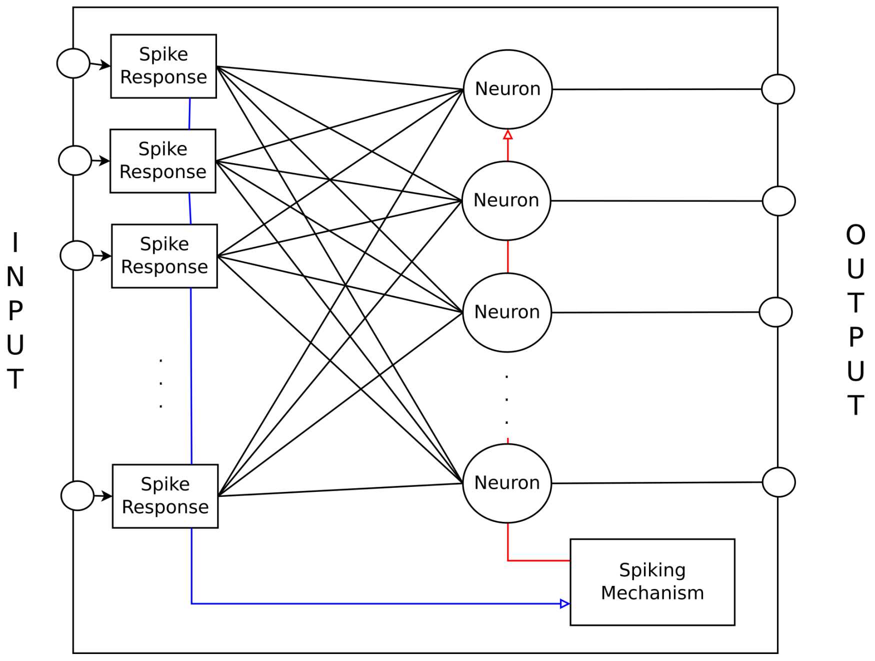


Figure 0.11.: SemCircuit architecture

0.17.1. SemConfigurable

The basic idea was to develop a class that could store and manage parameters, such as booleans, integers, floating point numbers and strings. Each parameter has a string identifier, and is stored, in accordance with data type, in one of four hash-maps. Each parameter had to be registered and could later be accessed by getter and setter functions. Each class that can be configured using an XML file is derived from SemConfigurable or at least holds an instance of SemConfigurable and is required to register its parameters in the constructor.

When using a floating point number, it is possible to add a variable schedule SemVarScheduleDouble, which changes the value each time a step function is called. It contains an initialization value, flags for operation behavior, possible end values, factors for step modification (addition, multiplication). This was necessary for changing e.g. learning rates or noise components during simulation.

0.17.2. SemCircuitElement

Each SemCircuitElement holds a member of SemConfigurable, can initialize using a configuration, and most importantly create an instance of the same type as its own. It also holds a long and a short type definition in a character pointer. Using this structure, a singleton factory SemCircuitElementFactory is used to register new SemConfigurable types and to create elements during runtime according to a short type definition, the long type is only used in the GUI for display purposes.

0.17.3. Spike Responses

A spike response is, generally speaking, a filter function that operates on finite spike trains and outputs a single floating point number. Calculating a spike response to a specific input spike train poses a more efficient way than calculating the membrane potential in every synapse that is connected to this input.

SemCircuitSpikeResponse

It forms the base class for all spike responses. The main function are the spikeHit function, which processes incoming input spikes, the advance function which is used to e.g. reset or decay internal values, and getValue, which returns the current membrane potential.

SemCircuitBufferResponse

Accumulates number of spikes until reset function is called and can be used to generate only binary output if appropriate flag is set.

binary_	bool	Binary output flag

Table 0.5.: SemCircuitBufferResponse parameters

SemCircuitSquarePulseResponse

This is a rectangular filter which counts the number of spikes in a specific time window.

a_	double	Amplitude of the square pulse
tau_	double	Width in secs of the square pulse spike response

Table 0.6.: SemCircuitSquarePulseResponse parameters

SemCircuitBufferDecayResponse

Set to 1 if spike hits and decays value. Output is 0 if membrane potential is smaller than cutoff parameter, otherwise output is membrane potential or 1 (if binarize flag is set).

decay_	double	Decay constant
cutoff_	double	A lower value is set to 0
reset_factor_	double	Reset multiplicative factor
binarize_	bool	Binary output flag

Table 0.7.: SemCircuitBufferDecayResponse parameters

SemCircuitCapacitorResponse

Uses exponential decay and the following spikeHit function (v is internal value, c is charge parameter).

$$v = v + (1 - v) * c$$

decay_	double	Decay constant
cutoff_	double	A lower value is set to 0
reset_factor_	double	Reset multiplicative factor
charge_	double	Charge parameter

Table 0.8.: SemCircuitCapacitorResponse parameters

SemCircuitAlphaResponse

Double exponential filter, uses AlphaFunctionSpikeResponse.

tau_	double	Decay time constant

Table 0.9.: SemCircuitAlphaResponse parameters

0.17.4. Neuron Types

SemCircuitNeuron

This class is the base class for all neuron types. Each neuron is connected to a 2D patch (defined by width, height) of inputs by synapses, which are stored in a 2 dimensional array using pointers. It allows addition, initialization and resetting of synapses, as well as readout functions for visualization purposes. Also present are functions for input processing (calculation of membrane potential as sum of input multiplied with synaptic weights), learning and possible log-likelihood calculation.

SemCircuitBiasNeuron

Neuron as in Equation 0.12, which holds the w_{k0} value in a bias variable. This bias is added in membrane potential calculation and adapted during learning.

min_bias_	double	Minimum bias
max_bias_	double	Maximum bias
init_bias_	double	Starting value
eta_mult_bias_	double	Multiplicative constant during learning

Table 0.10.: SemCircuitBiasNeuron parameters

SemCircuitRMHNeuron

Same parameters as SemCircuitBiasNeuron, but uses a calculated error which is the difference between a given target and the membrane potential. It is used in reward modulated hebbian learning experiments.

min_bias_	double	Minimum bias
max_bias_	double	Maximum bias
init_bias_	double	Starting value
eta_mult_bias_	double	Multiplicative constant during learning

Table 0.11.: SemCircuitRMHNeuron parameters

SemCircuitCWNeuron

Derived from SemCircuitBiasNeuron, it serves as the super-class for Extended SEM implementations (e.g. Poisson, Gauss).

min_bias_	double	Minimum bias
max_bias_	double	Maximum bias
init_bias_	double	Starting value
eta_mult_bias_	double	Multiplicative constant during learning
use_approx_	bool	Flag if approximation or exact calculation
init_cw_	double	Weight initialization
eta_mult_cw_	double	Learning rate multiplicative factor

Table 0.12.: SemCircuitCWNeuron parameters

SemCircuitPoissonNeuron

Poisson distribution based neuron.

min_bias_	double	Minimum bias
max_bias_	double	Maximum bias
init_bias_	double	Starting value
eta_mult_bias_	double	Multiplicative constant during learning
use_approx_	bool	Flag if approximation or exact calculation
init_cw_	double	Weight initialization
eta_mult_cw_	double	Learning rate multiplicative factor
mcorr_	double	Correction term for approximation

Table 0.13.: SemCircuitPoissonNeuron parameters

SemCircuitGaussNeuron

Normal distribution based neuron.

min_bias_	double	Minimum bias
max_bias_	double	Maximum bias
init_bias_	double	Starting value
eta_mult_bias_	double	Multiplicative constant during learning
use_approx_	bool	Flag if approximation or exact calculation
init_cw_	double	Weight initialization
eta_mult_cw_	double	Learning rate multiplicative factor
sigma_sq_	double	Variance constant

Table 0.14.: SemCircuitGaussNeuron parameters

0.17.5. Synapse Types

SemCircuitSynapse

Base class, holds weight value and a learning function. Since the number of synapses in a network is conventionally very large (up to millions), this implementation tries to keep the synapse classes as lightweight as possible.

w_init_	double	Weight initialization value

Table 0.15.: SemCircuitSynapse parameters

SemCircuitShiftedPoissonSynapse

Synapse implementing learning function from Equation 0.11, with negative weights shifted into positive domain (by adding a constant).

w_init_	double	Weight initialization value
m_	double	Positive weight shift constant
max_theta_	double	Maximum weight value
min_real_theta_for_learning_	double	Minimum weight value
eta_mult_theta_	double	Eta multiplication factor

Table 0.16.: SemCircuitShiftedPoissonSynapse parameters

SemCircuitShiftedPoissonVTSynapse

Same as SemCircuitShiftedPoissonSynapse, but with implemented variance tracking. The general idea of variance tracking is to model the variance of the estimator and adjust the learning rate appropriately. This should remedy the great dependence of the learning rate during learning, which is often a problem considering certain classes of input.

SemCircuitGaussSynapse

Synapse implementing a learning function, based on Gaussian distribution with constant, predefined variance.

SemCircuitGaussVTSynapse

Same as SemCircuitGaussSynapse, but with variance tracking.

w_init_	double	Weight initialization value
m_	double	Positive weight shift constant
max_theta_	double	Maximum weight value
min_real_theta_for_learning_	double	Minimum weight value
eta_mult_theta_	double	Eta multiplication factor
init_lr_	double	Learning rate initialization
min_lr_	double	Minimum learning rate
debug_display_	bool	Debug flag

Table 0.17.: SemCircuitShiftedPoissonVTSynapse parameters

w_init_	double	Weight initialization value
sigma_sq_	double	Variance constant
min_mean_	double	Minimum mean value
max_mean_	double	Maximum mean value
eta_mult_theta_	double	Eta multiplication factor

Table 0.18.: SemCircuitGaussSynapse parameters

w_init_	double	Weight initialization value
sigma_sq_	double	Variance constant
min_mean_	double	Minimum mean value
max_mean_	double	Maximum mean value
eta_mult_theta_	double	Eta multiplication factor
init_lr_	double	Learning rate initialization
min_lr_	double	Minimum learning rate
debug_display_	bool	Debug flag

Table 0.19.: SemCircuitGaussVTSynapse parameters

SemCircuitRMHSynapse

For reward modulated learning.

w_init_	double	Weight initialization value
eta_mult_w_	double	Eta multiplication factor

Table 0.20.: SemCircuitRMHSynapse parameters

0.17.6. Spiking Mechanisms

A spiking mechanism defines the conditions as well as the point in time when an output spike should be generated and selects the neuron that is allowed to produce an output spike.

SemCircuitSpikingMechanism

Base class for all spiking mechanisms, consists of a function to connect to a SemCircuit and an advance function.

SemCircuitWTASpikingMechanism

Simple soft winner-takes-all mechanism, selects one neuron based on the membrane potential distribution.

spiking_rate_	double	Number of spikes per time
eta_	double	Global learning rate
sharpness_	double	Determines “softness“ of WTA

Table 0.21.: SemCircuitWTASpikingMechanism parameters

SemCircuitRMHSpikingMechanism

Extension of SemCircuitWTASpikingMechanism, allows a reward modulation of learning in the network.

spiking_rate_	double	Number of spikes per time
eta_	double	Global learning rate
sharpness_	double	Determines “softness“ of WTA
method_	double	Method selection
sigma_	double	Reward normalization parameter
current_reward_	double	Current reward

Table 0.22.: SemCircuitRMHSpikingMechanism parameters

SemCircuitWTALowCutoff

spiking_rate_	double	Number of spikes per time
eta_	double	Global learning rate
sharpness_	double	Determines “softness“ if WTA
hard_cut_off_	double	Minimum input activity spiking condition
soft_cut_off_	double	Soft input activity condition

Table 0.23.: SemCircuitBufferWTALowCutoff parameters

SemCircuitWTALocalCompet

Extension of SemCircuitWTALowCutoff.

spiking_rate_	double	Number of spikes per time
eta_	double	Global learning rate
sharpness_	double	Determines “softness“ if WTA
hard_cut_off_	double	Minimum input activity spiking condition
soft_cut_off_	double	Soft input activity condition
compet_w_	int	Width of local competition
compet_h_	int	Height of local competition

Table 0.24.: SemCircuitBufferWTALocalCompet parameters

```
<?xml version="1.0" ?>
<SpikingExperiment>
  <Global>
    ...
  </Global>
  <SemCircuitElement type="..." id="...">
    ...
  </SemCircuitElement>
  <Layers>
    <InputLayer id="...">
      ...
    </InputLayer>
    <SemCircuit id="..." spikeresponse="..." neuronmodel="..."
                      synapsemodel="..." spikingmechanism="...">
      ...
    </SemCircuit>
  </Layers>
  <Connections>
    <SimpleConnection>
      ...
    </SimpleConnection>
  </Connections>
</SpikingExperiment>
```

Table 0.25.: SpikingExperiment configuration schema

0.17.7. SemConfigReader

Using the open-source library tinyxml, an XML file is read in and the configuration is stored in SemConfigurable classes, which are later used to dynamically generate the simulation environment.

Parameters are defined using a <Parameter*Type* name="value" />tags, where *Type* can be Bool, String, Integer or Double. Additionally, a write flag (write="1") can be set if the value should be changeable during runtime.

Each configuration is embedded in *SpikingExperiment* tags, the *Global* tags contain some general simulation parameters. Then, multiple *SemCircuitElements* can be defined, which define **all** neuron types, synapse types, spike responses and spiking mechanisms used, each identified by an unique id. In *Layers*, all input and SEMCircuit layers are defined, for each SemCircuit the identifiers of *SemCircuitElements* used must be given. In *Connections*, the connection structure between layers is defined, note that connections between layers only forward spikes without any additional behavior.

Since it is often useful to use parameters which change over simulation time, a variable schedule was implemented. In the parameter *target_id*, one defines the SemCircuitElement in which the variable resides, *target_var* is the variables id, and other parameters control the lifetime values of the scheduled variable (init value, min/max value, additive factor, multiplicative factor). The value modification takes place each time *advance_step* time steps have come to pass.

0.17.8. SemCircuitNetworkManager

This class, which is also a SemConfigurable, holds an instance of SemConfigReader and stores the simulation information during the XML parsing process. It also provides all functionality to set up, run and destroy a simulation environment, holds all SemCircuit elements including the SemCircuitElementFactory and is the main interface for the Java counterpart.

```
<VarScheduleDouble id="schedule1">
  <ParameterString target_id="wta2" />
  <ParameterString target_var="eta" />
  <ParameterInt advance_step="100" write="1"/>
  <ParameterDouble add_factor="0" write="1"/>
  <ParameterDouble mult_factor="0.985" write="1"/>
  <ParameterDouble min="0.0001" write="1"/>
  <ParameterDouble max="1" write="1"/>
  <ParameterDouble init="0.1" write="1"/>
  <ParameterBool run_at_start="true"/>
</VarScheduleDouble>
```

Table 0.26.: Variable schedule example

0.17.9. List of files

0.18. Java

In the jAER framework, filters can be added which can receive asynchronous events from hardware devices, such as the DVS128. These filters process the events and can relay or generate events to later filters (filter chain). In this pre-existing framework, new filters were added, which feed events to the java interface of PCSIM, read out and visualize the results.

0.18.1. SemCircuitFilter

A filter itself can define parameters, which are automatically displayed in the jAER GUI, are stored during shutdown and reloaded at start of the filter.

The SemCircuitFilter extends the jAER EventFilter2D and thus inherits a filterPacket function, which is called with a packet of input events and returns the same object (possible output events). As startup the function initFilter is called, which loads and initializes the jpcsim library and creates the GUI and its controller singleton.

When the SemCircuitFilter is added to the filter chain, an additional window is created which manages the high-level control functions of the simulation. Once a correct simulation configuration is loaded, check-boxes for single layer view and control are added at the bottom of the dialog. The init button re-initializes the complete experiment and can be used if the experiment configuration has changed. Note that the filter is not active until the start button is pressed.

SemCircuit.cpp
SemCircuit.h
SemCircuitElement.cpp
SemCircuitElement.h
SemCircuitNeuron.cpp
SemCircuitNeuron.h
SemCircuitSpikeResponse.cpp
SemCircuitSpikeResponse.h
SemCircuitSynapse.cpp
SemCircuitSynapse.h
SemConfigReader.cpp
SemConfigReader.h
SemConfigurable.cpp
SemConfigurable.h
SemInputLayer.cpp
SemInputLayer.h
SemNetworkElement.cpp
SemNetworkElement.h
SemNetworkManager.cpp
SemNetworkManager.h
SemNeuron.cpp
SemNeuron.h
SemOtherCircuit.cpp
SemOtherCircuit.h
SemPopulation.cpp
SemPopulation.h
SemSpikingMechanism.cpp
SemSpikingMechanism.h
SemSynapse.cpp
SemSynapse.h
SemUtils.h
SemVarSchedule.cpp
SemVarSchedule.h
SemWTA.cpp
SemWTA.h

Table 0.27.: List of files (C++)

experiments_folder_	Path to XML experiment configurations
xml_editor_	Name of favorite xml text editor
input_id_	Name of the input neuron layer
ignore_off_events_	Omits off events when true
input_w_	Number of pixels in width
input_h_	Number of pixels in height
cluster_w_	Width of cluster window
cluster_h_	Height of cluster window
use_tracker_	Flag if cluster tracker is used
last_trace_decay_	Decay constant

Table 0.28.: SemCircuitFilter parameters

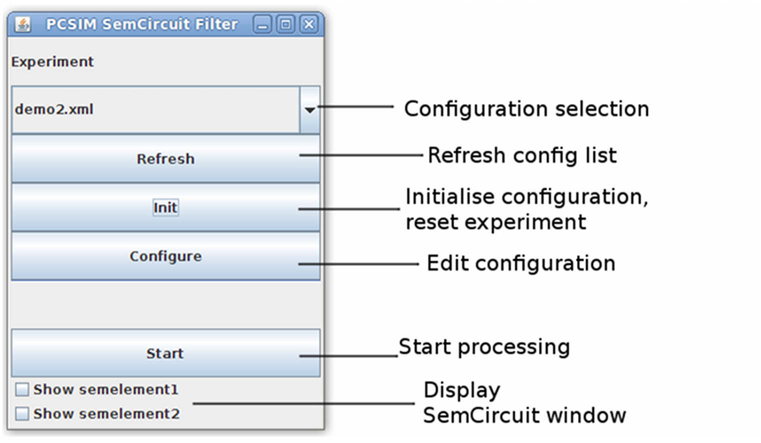
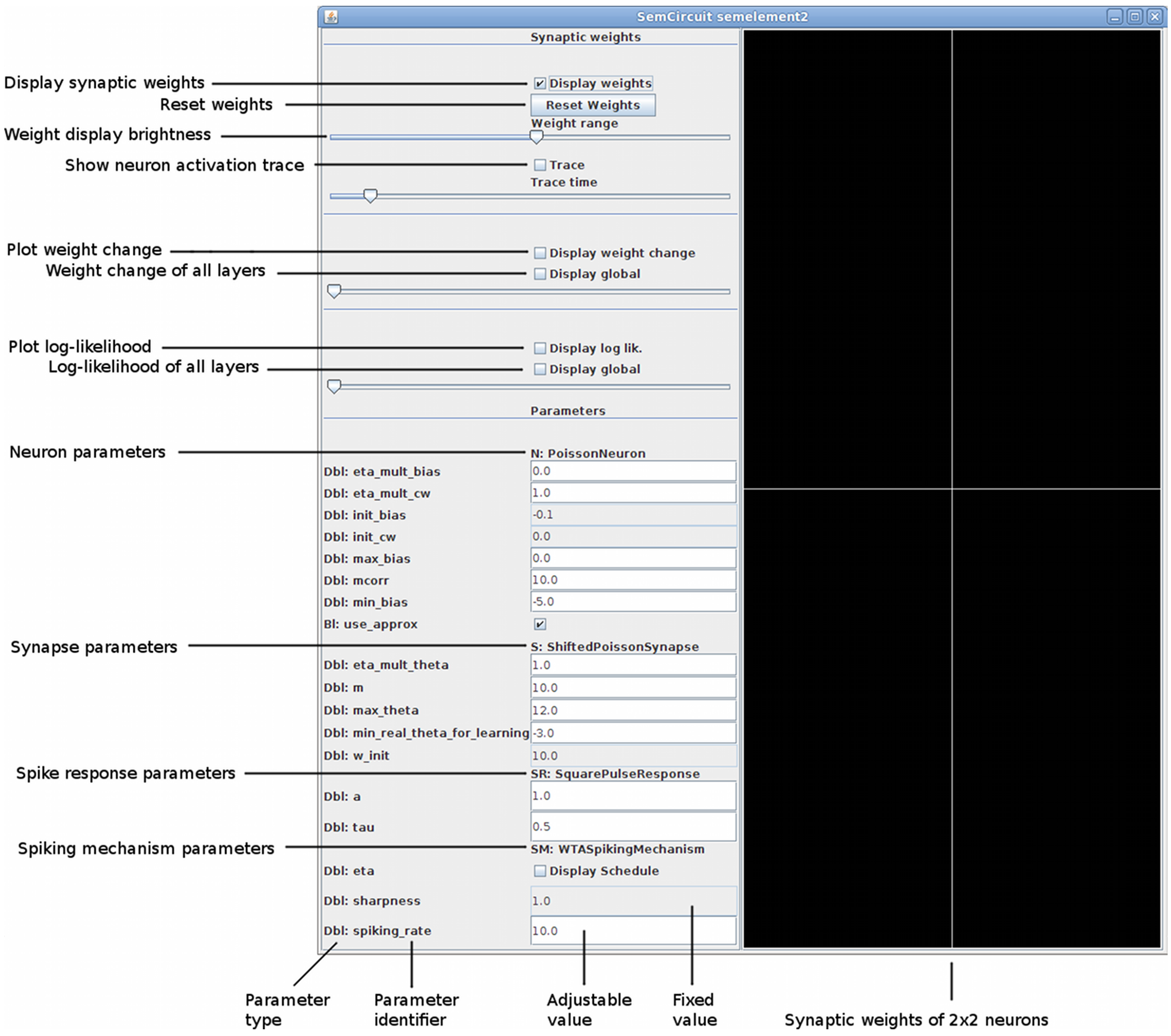


Figure 0.12.: SemCircuitFilter main GUI

When a layer is displayed, all layer elements are read out during runtime and a GUI is dynamically generated. All parameter values represent the actual current value during the simulation, therefore one can easily change a value (if permitted) and see the immediate changes during the simulation.


Display synaptic weights
Reset weights
Weight display brightness
Show neuron activation trace
Plot weight change
Weight change of all layers
Plot log-likelihood
Log-likelihood of all layers
Neuron parameters
Synapse parameters
Spike response parameters
Spiking mechanism parameters
Parameter type
Parameter identifier
Adjustable value
Fixed value
Synaptic weights of 2x2 neurons

SemCircuit semelement2

Synaptic weights

[✓] Display weights
Reset Weights
Weight range
☐ Trace
Trace time

☐ Display weight change
☐ Display global

☐ Display log lik.
☐ Display global

Parameters

N: PoissonNeuron

Dbl: eta_mult_bias	0.0
Dbl: eta_mult_cw	1.0
Dbl: init_bias	-0.1
Dbl: init_cw	0.0
Dbl: max_bias	0.0
Dbl: mcorr	10.0
Dbl: min_bias	-5.0
Bl: use_approx	✓

S: ShiftedPoissonSynapse

Dbl: eta_mult_theta	1.0
Dbl: m	10.0
Dbl: max_theta	12.0
Dbl: min_real_theta_for_learning	-3.0
Dbl: w_init	10.0

SR: SquarePulseResponse

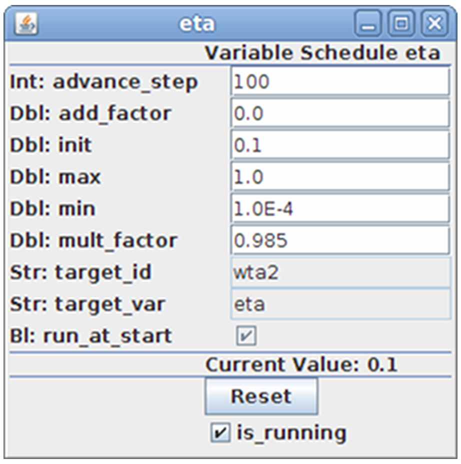
Dbl: a	1.0
Dbl: tau	0.5

SM: WTASpikingMechanism

Dbl: eta	☐ Display Schedule
Dbl: sharpness	1.0
Dbl: spiking_rate	10.0

Figure 0.13.: SemCircuitFilter layer GUI

When multiple layers are used, each can reside in a separate window. Synaptic weights can be viewed using a double-click, a left-click selects a neuron, which causes a labeling of neurons on a lower layer. When one or several neurons of a layer are selected, the synaptic weights of synapses which connect from a lower layer are drawn as green borders of different brightness (according to normalized weight). Another useful feature is the variable schedule dialog, which allows adjustment of a variable which changes in each time-step. Please refer to the C++ implementation for a more detailed discussion.



Variable Schedule eta

Int: advance_step	100
Dbl: add_factor	0.0
Dbl: init	0.1
Dbl: max	1.0
Dbl: min	1.0E-4
Dbl: mult_factor	0.985
Str: target_id	wta2
Str: target_var	eta
Bl: run_at_start	✓

Current Value: 0.1

Reset

✓ is_running

Figure 0.14.: SemCircuitFilter variable schedule dialog

0.18.2. HybridClusterTracker

One of the problems encountered was that objects move in the input space, and are stored in synaptic weights at all possible locations. While this can be resolved by using a large number of neurons, for some tasks it was necessary to focus on other aspects than translation invariance. In the jAER implementation, a filter ClusterTracker was already implemented, as mentioned in (Delbruck, 2008). This filter, generally speaking, was written to find the rectangular area (of defined size) where maximal spiking activity occurs, with some side constraints which should avoid too erratic movement. When using this filter to identify the position of an object in the input, several problems made an extension of the ClusterTracker necessary. For example, the maximum spike count indicates primarily the most active region of an object, but not a good approximation of the center position. Most of the other problems did not arise because of problems in the ClusterTracker, but because of the necessity of integration of a tracker component with the later filter stage, which simply required an extended class. To resolve these problems, a HybridClusterTracker was developed, which used a number of embedded ClusterTracker filters (e.g. 8) and used their positions to calculate a more stable approximation of the object center. The activity centers were grouped together using k-means, and a group center was the center of mass of all group members.

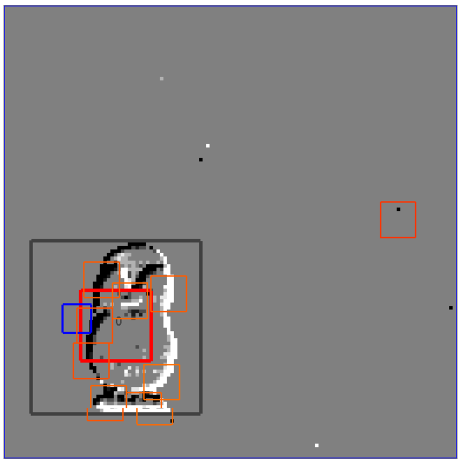


Figure 0.15.: SemCircuitFilter HybridClusterTracker

Experiments

0.19. Experiment 1: 1 Layer

In the first experiment, the input was generated with the Simulator SpikeSim (refer to Section 0.15 for a detailed discussion) using a bitmap of a toy penguin and additional uniform noise. One low-pass filtered example of the input (shown in jAERViewer as usual) is shown in Figure 0.16.

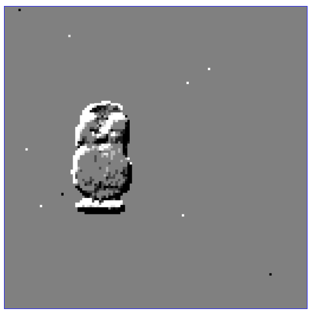


Figure 0.16.: Experiment 1: Example input

The simulated network consisted of 8x8 GaussNeurons, which were all-to-all connected (using GaussSynapses) to the input layer (128x128 standard resolution). As spike response a SquarePulseResponse was used, the spiking mechanism was a WTALowCutoff (refer to Appendix .1 for the exact configuration).

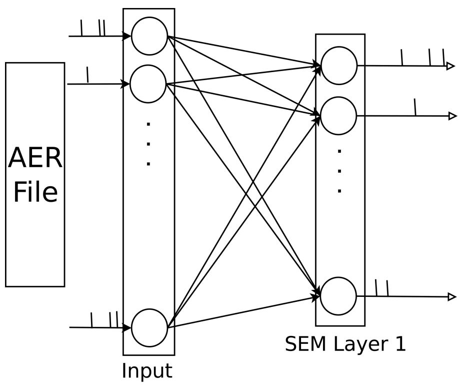


Figure 0.17.: Experiment 1: Network architecture

After simulating the network, the synaptic weights of each of the neurons show a blurry version of the input, with the object at different locations, as shown in Figure 0.18. Each image is a graphical representation of the synaptic weights of one neuron (in 128x128 pixels).

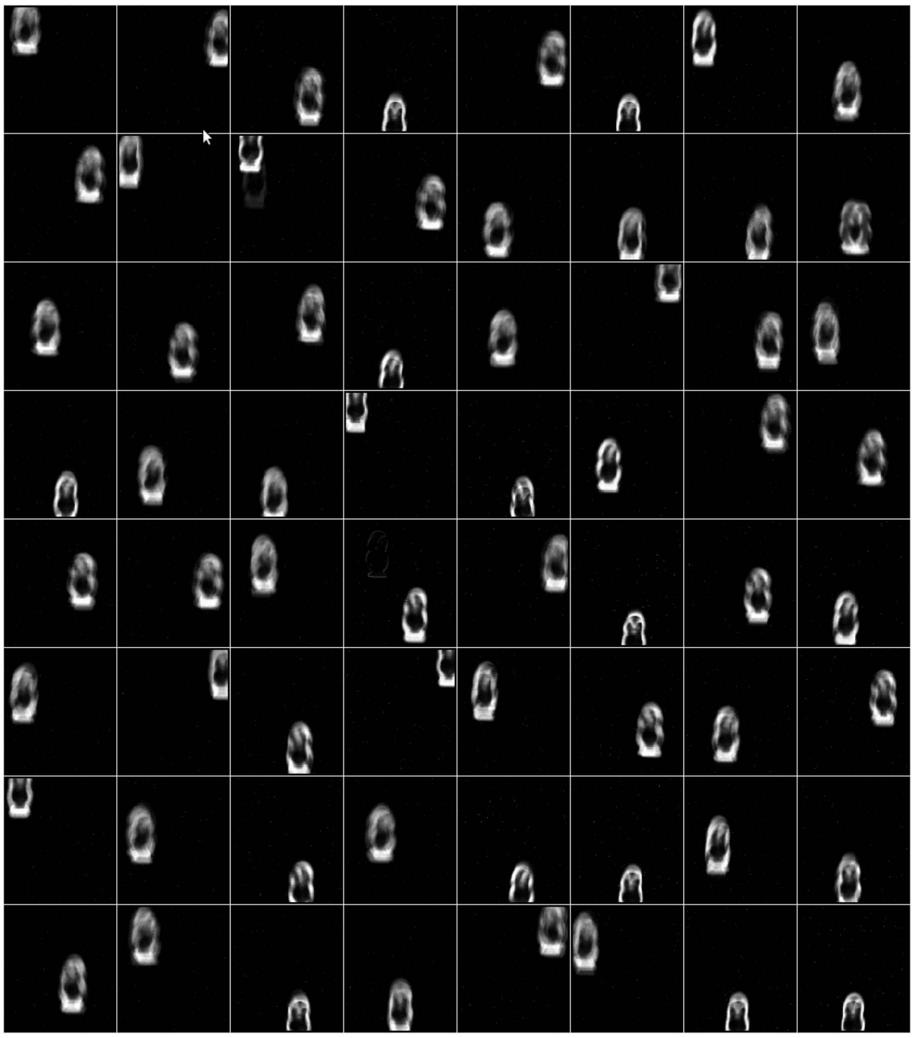


Figure 0.18.: Experiment 1: Synaptic weights

During the learning process, each neuron specializes on one specific input pattern and always gets activated if a similar pattern is shown, thus adapting its weights towards the current input. If the quantity of neurons is large enough, compared to the complexity of the input, each neuron would specialize on exactly one specific input. When, as in this setup, the number of neurons is relatively small (64), each neuron represents an average of the distribution of all input patters with strong spiking activity in roughly the same area.

0.20. Experiment 2: 2 Layers - Classification

As shown in experiment 1, when choosing the number of neurons in a layer, one has to make a significant trade-off between generalization (small number of neurons) and exactness (large number of neurons). If one wants to classify objects using the output of a neuron layer, few neurons might suffice if the objects do not change significantly over time, if that is the case, one can simply label each neuron (by hand or with appropriate supervised training data) according to object representation. If however, objects change (e.g. translate, scale, rotate), few neurons lead to a poor input representation and objects cannot be distinguished properly.

If one uses a large number of neurons, one can circumvent this problem, but is forced to label a large number of neurons, which may be possible if one has acceptable quantities of supervised input data. If this is not the case, a second neuron layer can be used as label substitute, if certain assumptions about the input hold. In experiments with slow feature analysis, one assumes that the most important features of the input vary slowly over time. If one extends this notion to the assumption that only one object is present over a sufficient amount of time, one can use the accumulated spiking activity of the first layer to group together neurons that only fire if one particular object is present.

For this reason a second neuron layer was introduced, which uses a rather “wide” square pulse response (tau is 0.5 as opposed to 0.05 in layer 1). This large time constant ensures that the second layer receives a histogram of the spiking activity of the first layer. After the first layer has converged (neurons fire only if one specific object is present) neurons in the second layer represent the accumulated spiking activity and, after sufficient training, group together neurons which are likely to fire as one and, if the assumption about the input holds, can be used to classify input objects.

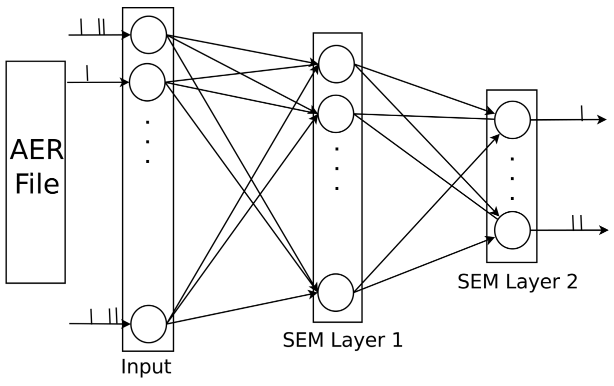


Figure 0.19.: Experiment 2: Network architecture

In the described architecture 256 neurons are used in layer 1, but only 4 neurons in layer 2 (refer to Appendix .2 for the exact configuration). Input generated with SpikeSim using 2 objects (bear and penguin), which are alternately present (with a short pause of no spiking activity in between), as shown in Figure 0.20.

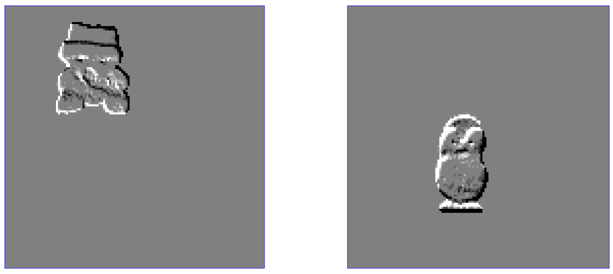


Figure 0.20.: Experiment 2: Alternating input objects

After learning, the synaptic weights of neurons in the first layer represent one object, while in the second layer, each neuron specializes in one common spiking pattern of the first layer. Each second layer neuron specializes on one of the objects, the number of neurons per objects and order varies from trial to trial. If one uses the synaptic weights of the neurons on the second layer which have specialized on object 'bear' and projects the (normalized) connection strength to the first layer, one can see which neurons on the first layer are commonly activated by that specific object. The results are shown in Figure 0.21, with red borders for object 'bear' and green for object 'penguin'. The brightness of the borders denotes the activation frequency. Using the described method proves to be a natural way to gain good object representation while avoiding elaborate labeling of neurons according to object class.

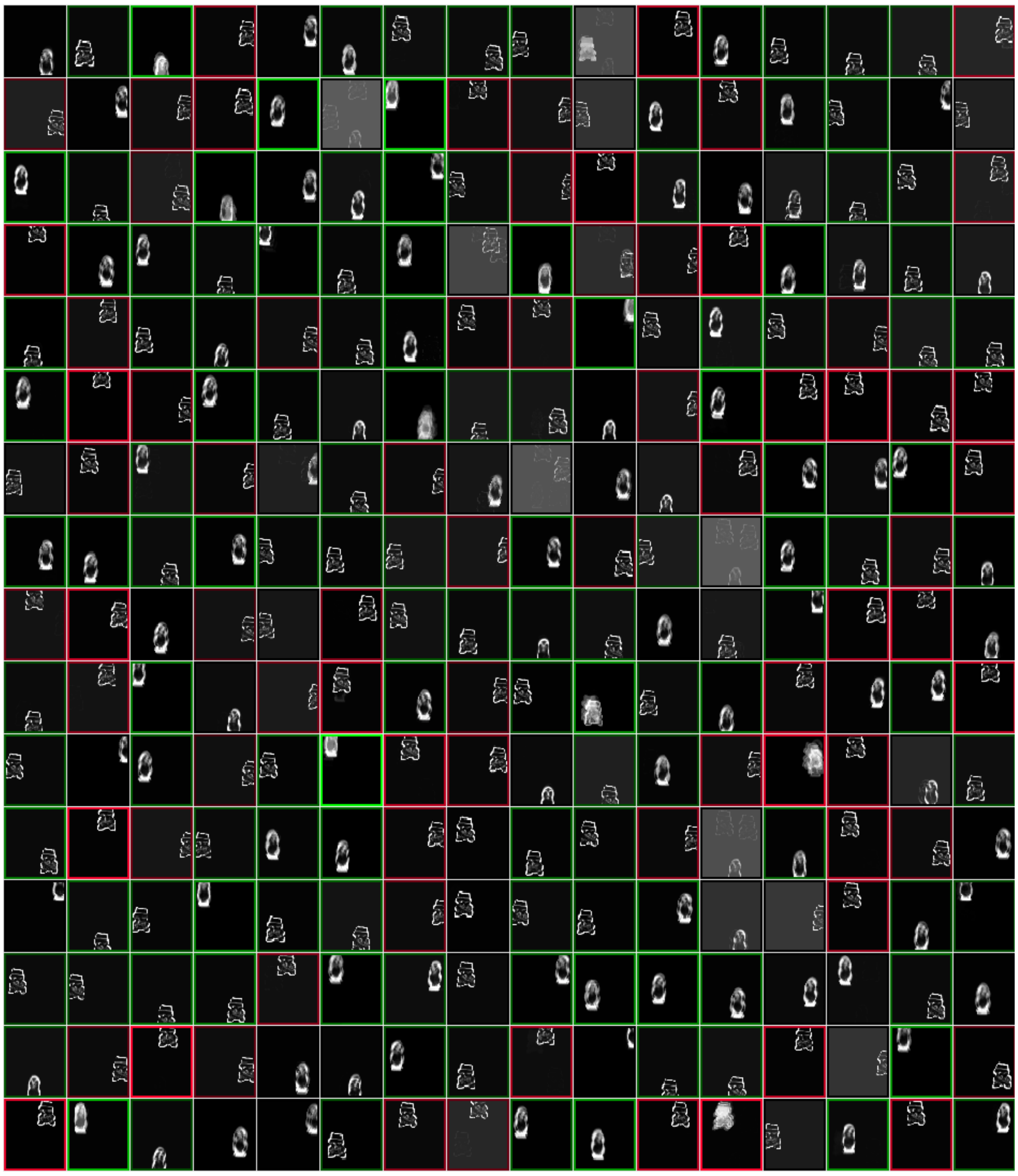


Figure 0.21.: Experiment 2: Labeled synaptic weights

From a more detailed inspection of the results, one notices that some neurons have wrong class label. This can be explained by considering the input of the second layer during the time of object switching.

During this time period, some neurons representing the prior object can become associated with the later object, and vice versa. One of the solutions can be a longer pause between the objects, this, however, fabricates another problem.

If no input spikes occur and the SEM layer spikes during this period, some neurons will store this null pattern and will become associated with one object class, or will be represented as an additional class. While a pause reduces the intersection between the object classes, it simply adds more “transition neurons” (with empty input) and thus complicates the input-class association. For this reason, the WTALowCutoff spiking mechanism was used, which cuts off learning when weak input is present.

Another solution would be decreasing the “histogram length” of the input of the second layer by reducing the time constant tau and adding more neurons to the second layer. This would result in more accurate class labels, but also require more effort concerning the second layer neurons. In all cases described one has to choose a reasonable trade-off, depending on the goal of the task.

0.21. Experiment 3: 2 Layers - Tracking and Classification

In the previous experiments, a large number of neurons in the first layer was required to sufficiently represent translating objects, by storing each different location in a different neuron. To circumvent this problem, a very basic attention mechanism was used to process only the most active parts of the current input. The original input size of 128x128 pixels was reduced to two smaller input windows of 48x48, approximately matching the object size. Only input from these windows was fed as input to the neural network. Also, in this experiment, two objects were present in the input at the same time, therefore two of these input windows were moved, each focusing on one object. As attention mechanism, the previously described HybridClusterTracker (refer to Section 0.18.2) was used to move each input window to the location of maximum spiking activity while avoiding extensive erratic movement.

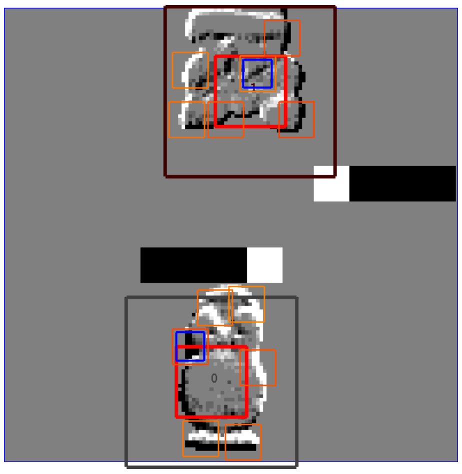


Figure 0.22.: Experiment 3: Input objects with HybridClusterTracker

Each of the HybridClusterTracker windows is annotated with a graphical representation of the spiking activity of the second SEM layer, consisting of 4 boxes (for 4 neurons) with color ranging from white (high activity) to black (no activity). From the two tracker windows, the input channels are forwarded as input to the neural network, as depicted in Figure 0.23. Using this simple attention mechanism, the object’s translation is strongly reduced, therefore a smaller number of neurons in the first layer suffices. Also, a number of objects can be present as input, but with an equal number of HybridClusterTracker windows each part of the network only receives input from one of the objects, therefore the same assumption holds as in the previous examples. If, as in these experiments, the objects are allowed to overlap and mask one another, the HybridClusterTracker windows may switch objects, and the objects are not separable for a short amount of time.

Another necessary extension of the previous network architecture was the weight sharing between the two first SEM layers, as well as between the two second SEM layers. The input to each first layer is input generated by one of the objects, and is represented by one of the neurons. Since the object type in the input may switch, it is advantageous to use shared weights and to represent object appearances only once. The output of each first SEM layer is then fed into a similarly constructed second layer. In each spiking mechanism activation, one neuron in each layer fires and adapts their weights according to the learning function.

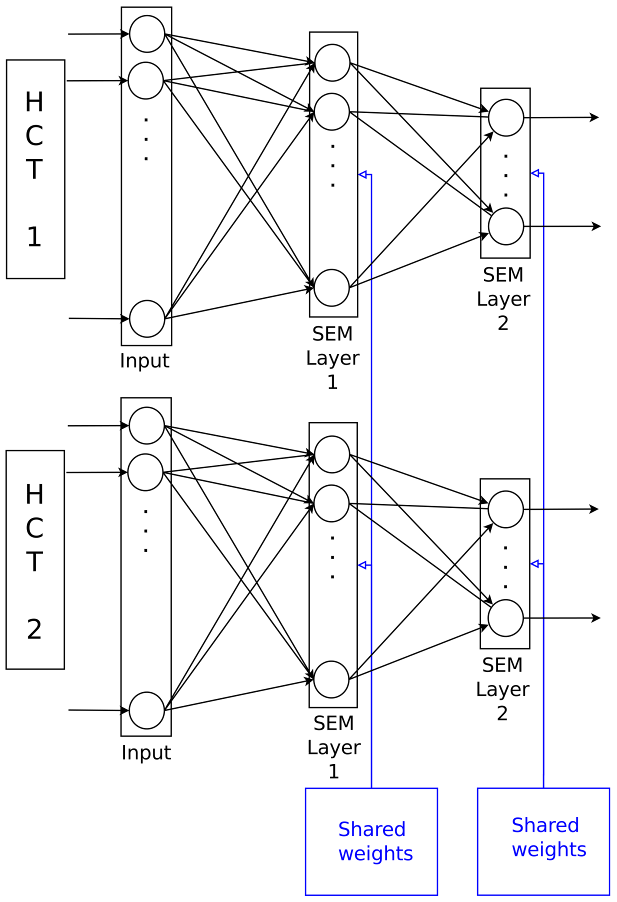


Figure 0.23.: Experiment 3: Network architecture

In the first layer 12x12 neurons were used and 4 neurons in the second layer, the resulting weights of first the layer are shown in Figure 0.24, again labeled by the synaptic strength of the 2nd layer neurons (refer to Appendix .3 for the exact configuration). As difference to the results from Experiment 2 should be noted, that less neurons where used to and even so only a small number is regularly activated.

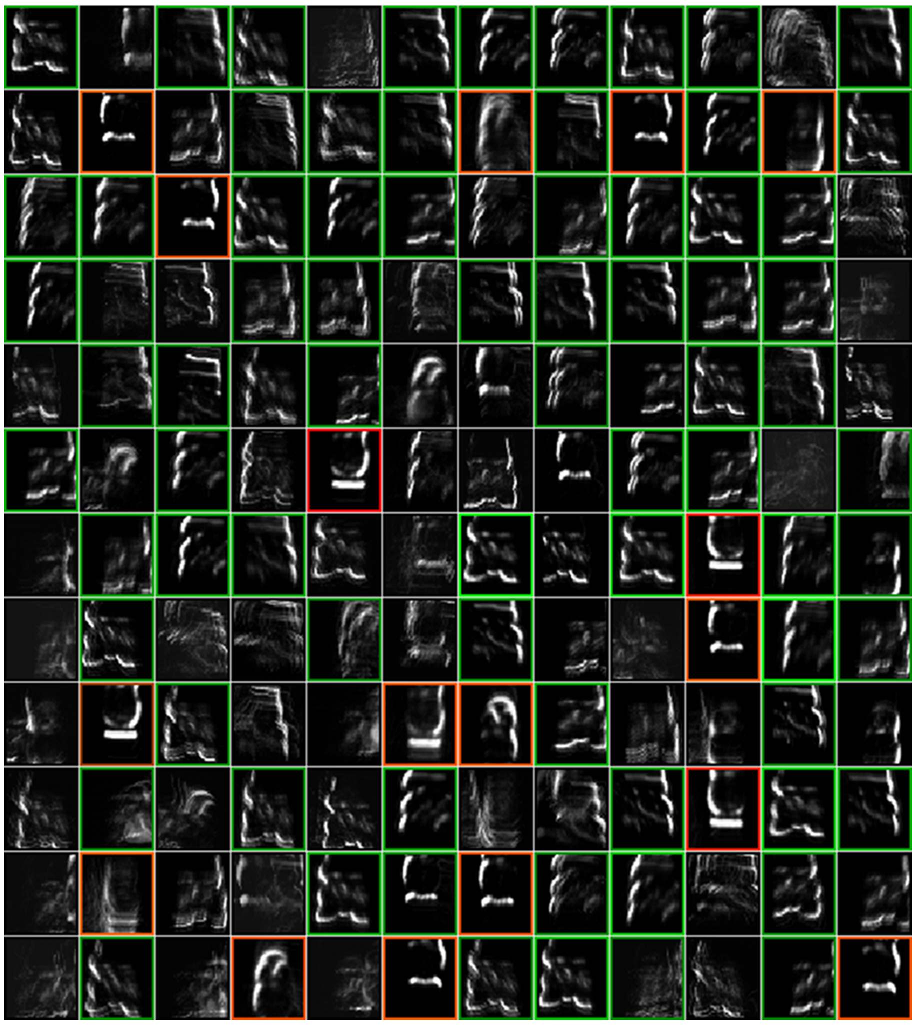


Figure 0.24.: Experiment 3: Synaptic weights of layer 1

0.22. Experiment 4: Digit Task

As an example of a real-world object recognition and tracking task, a scene was recorded consisting of different, colored objects which are moved over a (partly similar) colored background (as shown in Figure 0.25). In this case, digits were used as objects, but any distinguishable shape that generates a recognizable change in luminance would be possible. The background consisted of colored distractor shapes which are in part of the same color as the digits. Due to the nature of the DVS128 sensor, static objects and parts of moving objects, when positioned over similar colored background, do not generate spikes.

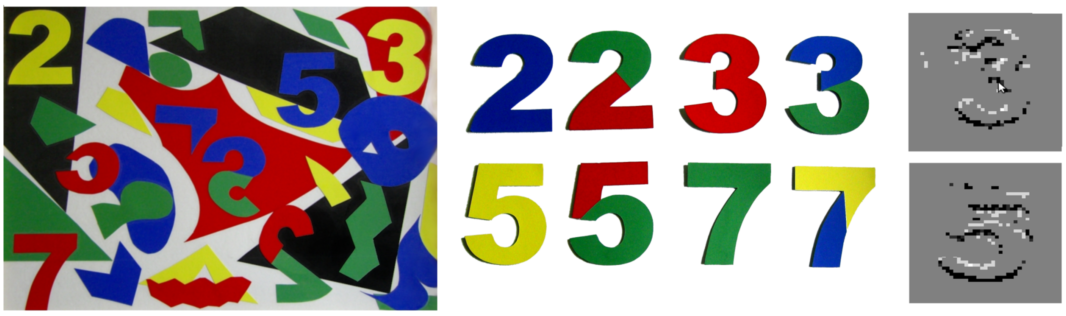


Figure 0.25.: Model setup: Background, objects, and resulting low-pass filtered retina output

When solving this task, two SEM layers with bipartite all-to-all connectivity were used, similar to experiment 2 (Figure 0.19). The first SEM layer contains a large number of neurons (50), to ensure a reasonable representation of the input distribution, while the second layer has significantly fewer neurons (2). The input spikes cover an area of 48x48 pixels and is generated by HyridClusterTracker window, giving a total number of 52 neurons and 115300 synapses (refer to Appendix .4 for the exact configuration).

As shown in Figure 0.26, the synaptic weights of the neurons in the first layer approximate and represent the input distribution. Each neuron represents a specific instance of a digit, and fires when a similar input pattern occurs.

The neurons in the second layer receive the spiking output of the first layer and represent histograms of its spiking activity in a short time window. Due to the learning scheme, first-layer neurons which are likely to fire together within a period of time are grouped together in the same second-layer neuron. Therefore, each neuron in the second layer represents a common spiking pattern of the first layer output. If a certain object is present long enough in the input, the spiking activity of the first layer during that period of time is dominated by neurons which represent this object. Then, at least one neuron of the second layer specializes in the resulting spike histogram and can be used as a class label for neurons in the first layer. Note that, on a functional level, this bears some resemblance to previous work on Slow Feature Analysis (Wiskott, 2002) which extracts slowly varying features from the input.

Projecting the synaptic weights of second layer neurons to the first layer, one can see in Figure 0.26 that Neuron 1 (red) labels most of the neurons specialized in the input digit 5, accordingly Neuron 2 (green) labels digit 3. The synaptic strength of the label corresponds to the spiking probability of the first layer neuron in the learned temporal sequence, which explains why some neurons have no label.

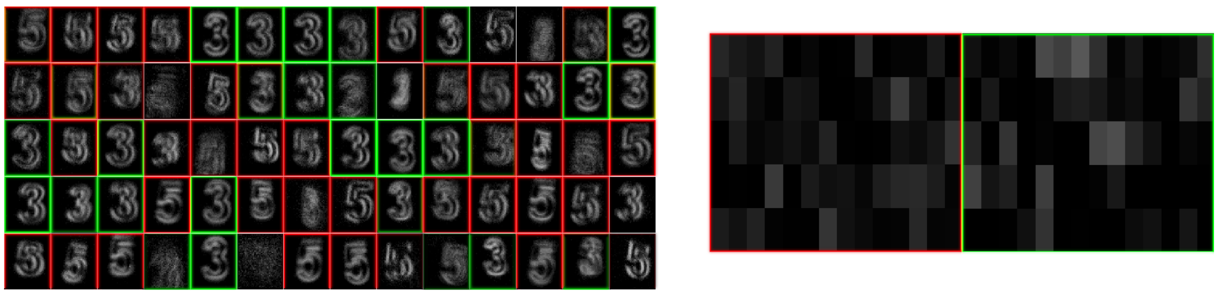


Figure 0.26.: Experiment 4: Synaptic weights with labels

0.23. Experiment 5: 2 Layer - Reward Modulation

In this experiment, a two-layer network was used and the HybridClusterTracker was substituted by a different mechanism. Still, a movable input window was used, but moved according to the spiking behavior of neurons from the second layer. Each neuron represented a movement in a different direction with different speed. As a basic reward function the sum of the input spikes in the input window was used, which modulated the learning rate of the second layer. Using this approach, after sufficient training the network was able to cover the one moving object with the window at all times, at least in a one-dimensional case. When two dimensions were used, the network required exceedingly long training before a good solution was found, but was also able to solve the task. Unfortunately, due to limited time, analysis and further studies of this task had to be omitted.

0.24. Hardware Requirements

Using the efficient SemCircuit implementation made it possible to simulate all of the previous experiments on an outdated laptop, with a dual-core 2.0 GHz processor with 2 GB of RAM. During simulation, most processor time was used by the visualization of the synaptic weights, since for each update and for each neuron an image has to be generated, forwarded to the GUI and rescaled according to display size. When using a few hundred neurons, one core was used to capacity by the GUI threads on the test computer. The network simulation itself, with extensive display computations omitted, allowed simulations up to a thousand neurons in real-time on the test machine. However, during development it was convenient to run pre-recorded training data well over real-time for smaller test cycles. For such purposes the author would recommend either smaller network sizes or up-to-date workstations. In the current implementation, multi-core processors are not used by the SemCircuit simulation, but jAER and the GUI benefit from dual-core processors.

0.25. Discussion

When summarizing the previous experiments, one can conclude that basic object recognition is possible using SEM in the described environment. In the results from Experiment 1 one can see how one object that translates over the input positions is stored on all possible locations. This already shows a strong limitation when using such a simple method on a task where the input is transformed and SEM has no possibility to compensate for the transformation except using an excessive number of neurons to store every possible state. Still using this brute-force approach, in Experiment 2 a two-layered network can classify two different objects correctly, if the assumption holds that the active object's identity varies slowly over time. In the third Experiment, a relatively simple pre-processing method is introduced, which keeps two movable input windows focused on the most active input regions. Using two two-layered networks with weight sharing, the object's identity can be classified as before and fewer neurons are needed for representation of the objects. In the previous experiments, only simulator-generated training data was used, in Experiment 4 a real-world recording was used, and correct object classification was achieved. In Experiment 5 a simple reward mechanism was used to imprint favorable behavior, but due to a lack of time adequate research on this interesting field was omitted.

Part III.

Application to Population Coding

General Notes

A prominent way to store important information in the brain using a population of neurons is a population code (introduced in Section 0.5). In the work presented here a single, analog input variable is assumed. This input is processed by a population of neurons (called encoding population), which for example can be sensory neurons. Each of these sensory neurons responds to the input stimulus according to its *activation function* with particular strength, and accordingly outputs a number of spikes drawn from a Poisson distribution.

An interesting question in this context is how a good estimate of a single variable that has given rise to a certain noisy population response can be made. Several models have been proposed, most importantly the population vector and maximum likelihood decoding strategies (Seung, 1993). A more general approach than maximum likelihood (which finds the most likely variable value) is the computation of a full likelihood function: to compute the likelihood of each possible value of the variable, as shown in (Movshon, 2006). This relatively simple and biologically plausible model that can realize the likelihood function by computing a weighted sum of sensory neuron responses, will be the starting point for the work presented here.

A Detailed View

0.26. Previous Work

According to (Movshon, 2006), the optimal decoding strategy of sensory information is to compute the likelihoods of the stimuli that could have given rise to an observed sensory population response. The proposed solution of representing such a likelihood function is similar (but not equivalent) to a population vector model in that activity of sensory neurons is pooled in a simple additive feed-forward architecture. In essence, a two-layered neural network is proposed, the first layer consisting of sensory neurons (encoding population), each of which receive a single stimulus Θ as input. Each of the sensory neurons possesses a preferred stimulus and an activation function, based on which a response is calculated. According to the response strength, a noisy number of spikes is generated by a Poisson distribution. In a second layer (readout population) the likelihood function is computed, based on a weighted sum of the first layer outputs. These static synaptic weights are calculated as the logarithm of the activation function used in the first layer. The complete process is shown in Figure 0.27.

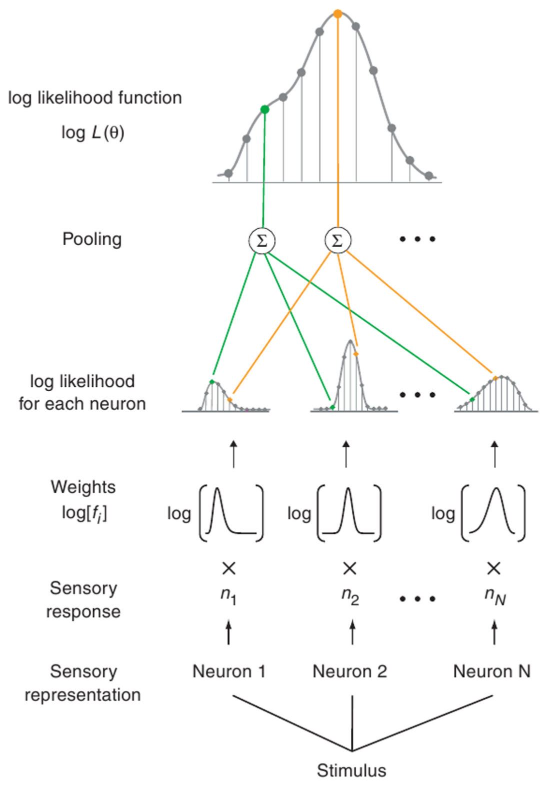


Figure 0.27.: Network architecture and function in (Movshon, 2006)

When computing the likelihood function with neurons, a stimulus Θ causes a sensory neuron (indexed by i, tuned to Θ_i) to fire n_i spikes in any given time window. It is assumed that such firing statistics can be described by a Poisson process, therefore n_i is Poisson-distributed and has a mean of $f_i(\Theta)$.

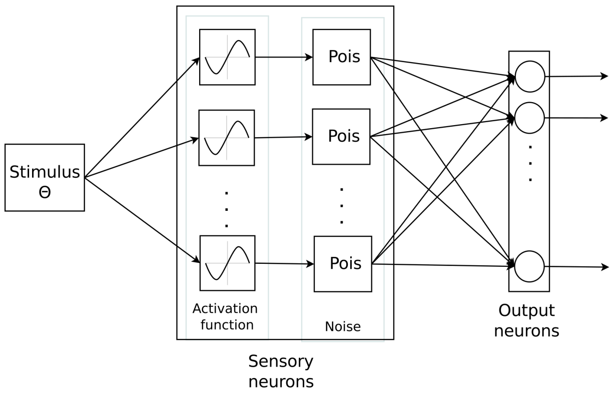


Figure 0.28.: Abbreviated Movshon network architecture

A sensory neuron tuning function $f_i(\Theta)$ uses tuning center Θ_i between $[0, 2\pi]$ and concentration parameter k and scale parameter c.

$$f_i(\Theta) = c * e^{k*cos(\Theta-\Theta_i))} \tag{0.20}$$

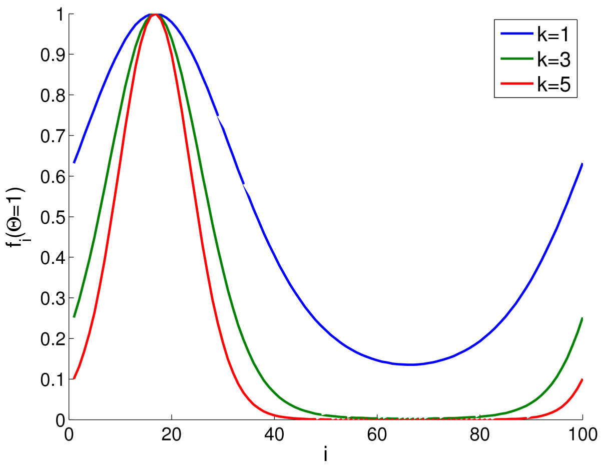


Figure 0.29.: Activation of sensory neurons with different concentration factors k generated by one stimulus $\Theta = 1$

The likelihood of a stimulus Θ, denoted $L_i(\Theta)$, is the unnormalized probability that a neuron would fire n_i spikes when presented with said stimulus.

$$\begin{aligned} \log L(\Theta) &= \sum_{i=0}^{N} \log L_i(\Theta) \\ &= \sum_{i=0}^{N} n_i \log f_i(\Theta) - \sum_{i=0}^{N} f_i(\Theta) - \sum_{i=0}^{N} \log(n_i!) \\ &= \sum_{i=0}^{N} n_i \log f_i(\Theta) + C \end{aligned} \tag{0.21}$$

The authors argue that in Equation 0.21 both of the last two terms can be ignored, since both are constant regarding Θ. This assumption is clearly true for the last term, for the other it can be assumed for certain specific sensory neuron populations where the overall response is constant for all stimuli, which is the case in their architecture.

With exact knowledge of the sensory neuron tuning function in Equation 0.21 the log-likelihood calculation can be further simplified.

$$\log L(\Theta) = \sum_{i=0}^{N} n_i \log(c * e^{(k(cos(\Theta-\Theta_i)))}) \tag{0.22}$$

0.27. Example Calculation

As an example, a complete calculation of the proposed method, as shown in Figure 0.27, was done for demonstration purposes. In the network 100 sensory neurons form the first layer, with Θ_i values equally spaced between $[0, 2\Pi]$ and 20 neurons in the second layer. The tuning functions use parameters $c = 5$ (for a higher spiking rate in the Poisson distribution) and concentration parameter $k = 1$. Input to the first layer Θ was uniformly set to 1, which excited a response as shown in Figure 0.30.

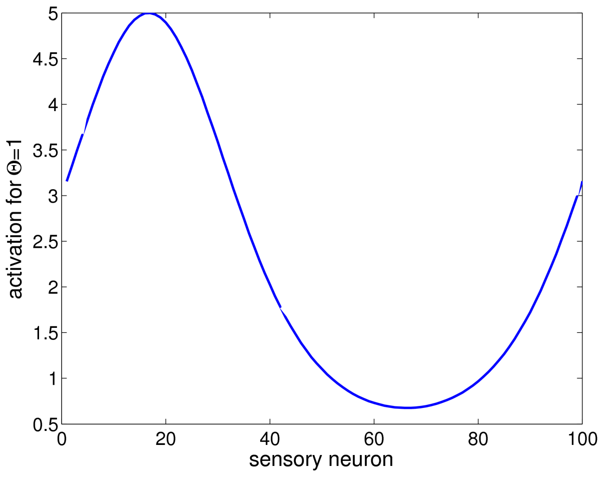


Figure 0.30.: Activation of sensory neurons generated by one stimulus $\Theta = 1$

Using these sensory neuron activation values as Poisson rates, for each sensory neuron a number of generated spikes n_i was calculated, as shown in Figure 0.31. These spike rates form the output of the first neuron layer.

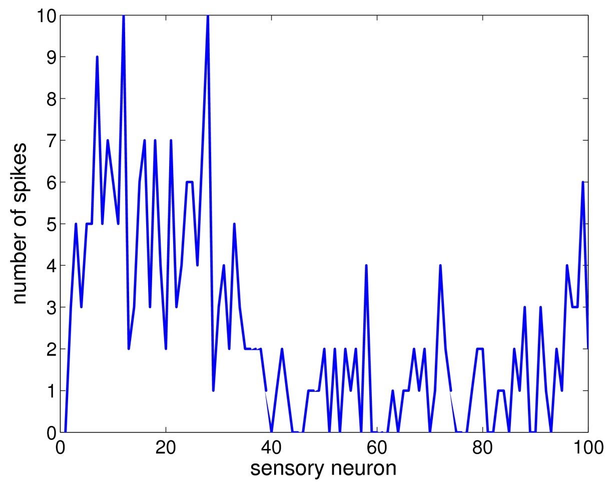


Figure 0.31.: Sensory neurons response to one stimulus $\Theta = 1$

Each of the neurons in the readout population is all-to-all connected to neurons in the first layer, and sum up all inputs with weights according to Equation 0.21, as depicted in Figure 0.32.

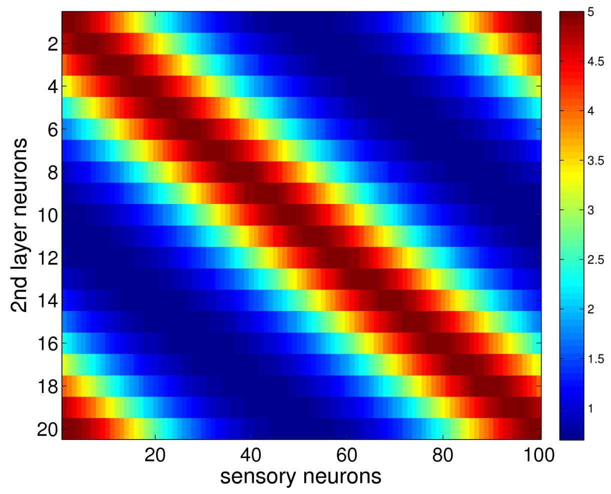


Figure 0.32.: Synaptic weights in matrix illustration

The resulting log-likelihood for stimulus Θ_0, calculated using Equation 0.22, is shown in Figure 0.33.

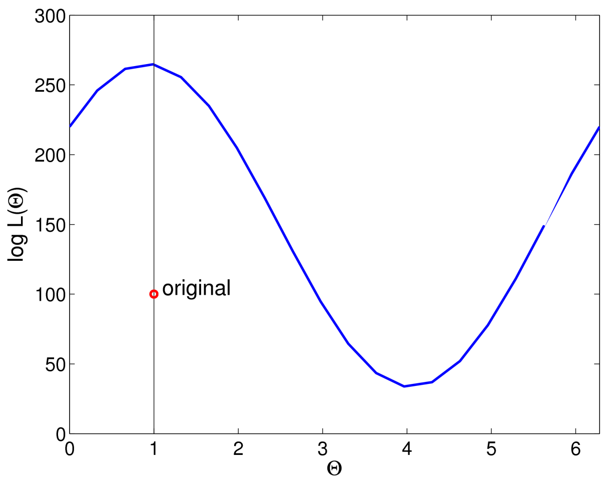


Figure 0.33.: Log-likelihood for stimulus $\Theta = 1$

Discussion

0.28. Implicit Assumptions

Using this simple architecture, a noisy representation of a measured stimulus can be transformed into a stable estimation of its likelihood function. However, this network implicitly assumes several characteristics which may be hard to achieve in grown, biological networks. For example, perfect tuning functions would require an exact number of sensory neurons with equally spaced Θ_i parameters, which is not changed during the lifetime of the network. Also, the synaptic weights from the second to the first layer require the knowledge of the exact tuning function of the pre-synaptic neuron, which may also be disputable.

If, for example, the tuning centers Θ_i of the sensory neurons are not as perfect as assumed, the assumptions made in Equation 0.21, which require a constant response of the sensory neuron population to every possible stimulus, would be violated. As shown in Figure 0.34, if Θ_i is subject to noise this also is reflected in the population response, an even more problematic case would be if Θ_i's are not equally spaced, which heavily distorts the balance of the sum of the overall response.

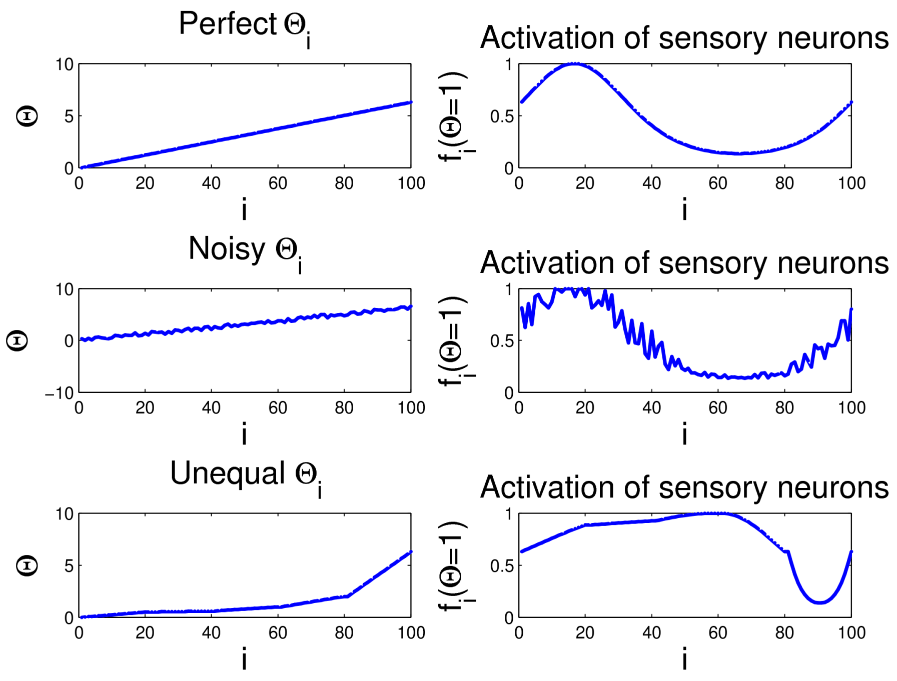


Figure 0.34.: Example activations of with different Θ_i vectors

Especially for the “unequally spaced Θ_i's” case, sensory activations with different Θ inputs, as shown in

Figure 0.35, no longer sum up to a constant, as shown in Figure 0.36.

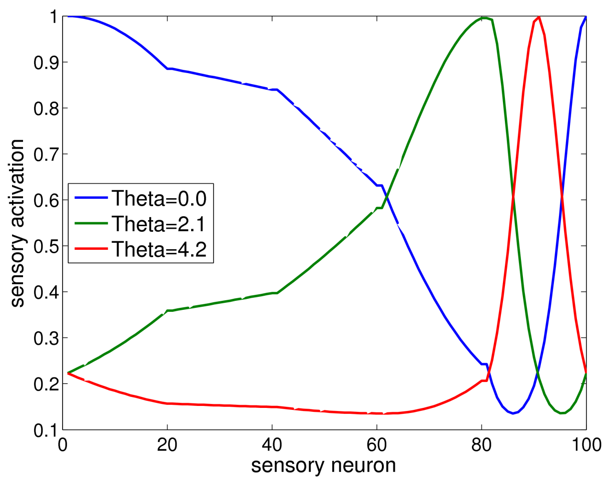


Figure 0.35.: Sensory activations for different Θ stimuli with unequally spaced Θ_i's

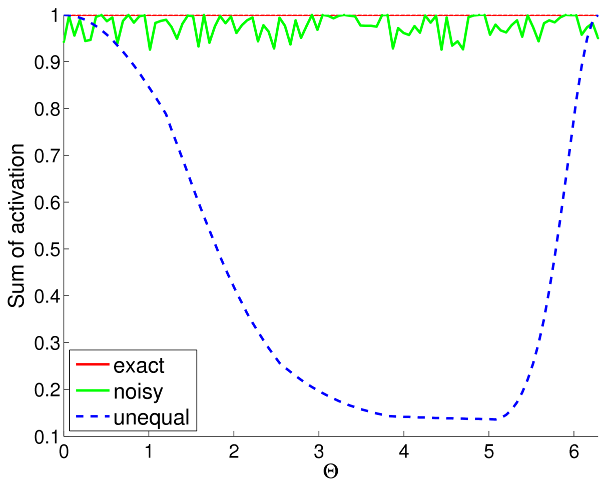


Figure 0.36.: Summation of activations with different Θ_i vectors

In the “noisy Θ_i” case, the estimation of the stimuli becomes imprecise (depending on the noise magnitude). However, for the “unequally spaced Θ_i” case, for some domain of Θ_0 the estimation fails completely due to the higher synaptic strength for other areas of Θ which dominate the calculation. Using the same calculation as before, but with a stimulus $\Theta = \pi$ and unequally spaced weights, one can show that the log-likelihood estimation is incorrect.

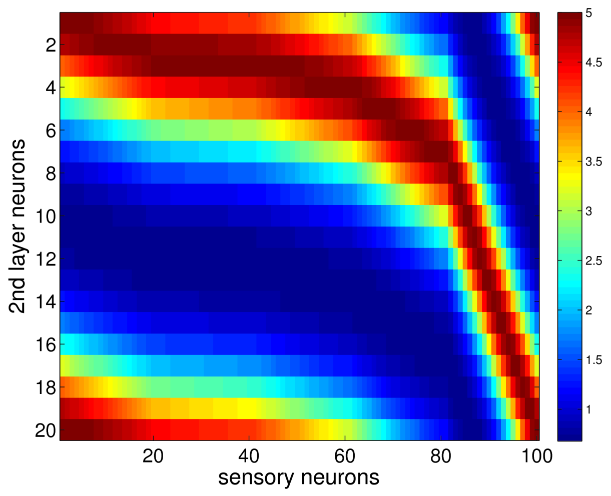


Figure 0.37.: Erroneous synaptic weights in matrix illustration

The resulting log-likelihood for stimulus $\Theta = \pi$, calculated using Equation 0.22, is shown in Figure 0.39.

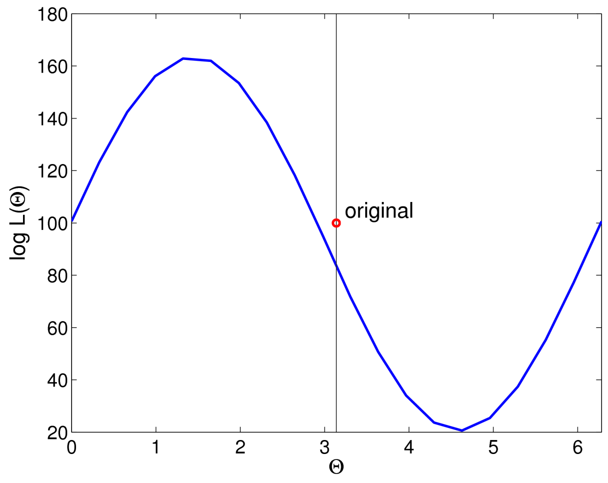


Figure 0.38.: Erroneous log-likelihood for stimulus $\Theta = \pi$

0.29. Motive for Considering Unequally Spaced Θ_i's

Studies as (Blasdel, 1992) in the primary visual cortex (V1) of macaque monkeys have shown that in maps of orientation preference and selectivity, linear organizations are present in patches. In such patches with high orientation selectivity, preferred orientations rotate linearly along one axis while remaining constant along the other.

It was argued that upper-layer neurons are likely to have similarly sized dendritic fields in all regions, and those in the linear zones should receive precise information about narrowly constrained orientations, while others near existing orientation fractures should receive coarse information about all orientations.

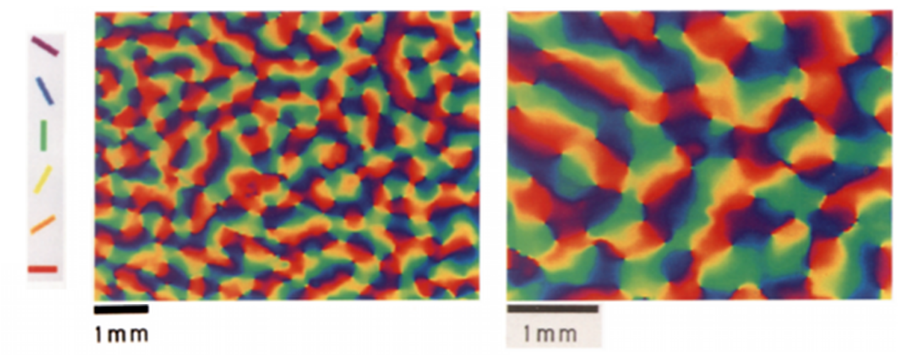


Figure 0.39.: Orientational fields in V1 of macaque monkey

Continuing this argument, one can argue that the assumption of a constant sensory neurons population response, is very likely to be violated in grown neural networks with connectivity limited to small lower-layer areas. Another argument against adherence of this assumption is that during cortical development the tuning centers would have to be set according to the size of the encoding population, which seems a complex task when considering naturally developing networks. Another implication of the assumption would be that cell death of a single sensory neuron would not only leave a “blind spot” but degrade the overall performance of the stimulus computation. Using a dynamic network could correct such erroneous behavior relatively easy and is supported by (Gerstner,2000), where it was argued that most cortical neurons show adaptation.

Weight Plasticity

0.30. Extended Applicability Through Learning

Static weights limit, as discussed in the previous chapter, plausibility and applicability of the proposed architecture. Therefore, the author proposes an extension by replacing the fixed, constructed weights of the synaptic connections of the second layer with a learning method as discussed in section 0.5. While the extended neural network requires preceding training before correct functionality, fewer constraints are required and addition or removal of neurons is no longer problematic.

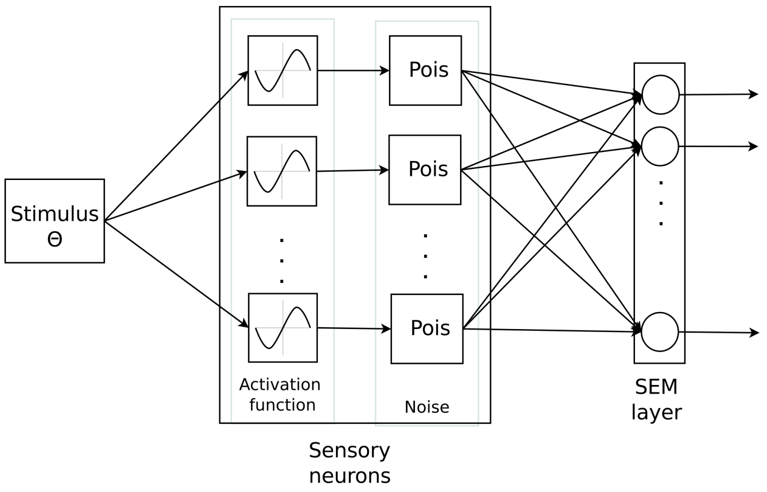


Figure 0.40.: Extended network

Using basically the same architecture as before, the second layer was substituted with a SEM layer consisting of 15 neurons with parameters winit= 4 and a constant learning rate $\eta = 0.03$. As input training sequence 10000 stimuli where drawn from a uniform distribution of $[0, 2\pi]$.

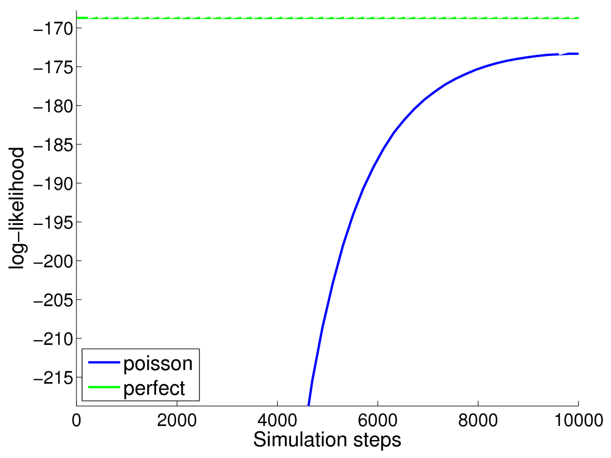


Figure 0.41.: Log-likelihood over time

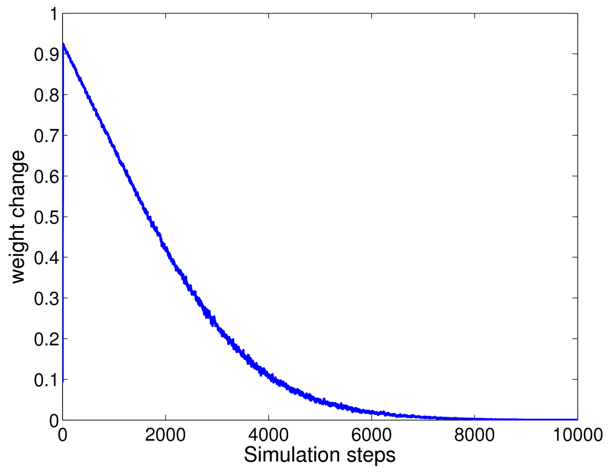


Figure 0.42.: Weight change over time

As one can deduct from the previous figures, the neuron weights in the SEM layer are already converged to sound values and only suffer minor changes over additional training steps. This behavior can of course be altered by changing the learning rate to either a lower constant or introducing a variable schedule, but for this simulation these steps were omitted.

Using the correlation with shifted tuning functions, the maximum center of response for each z-neuron can be calculated during learning, and the sum over the correlation gives the similarity between the current neuron weights and the optimal ones. The results of this calculation are plotted in Figure 0.43, where the marker size denotes the correlation strength. Please note that the y-axis is periodical to 2π, therefore neurons may swap sides from time to time.

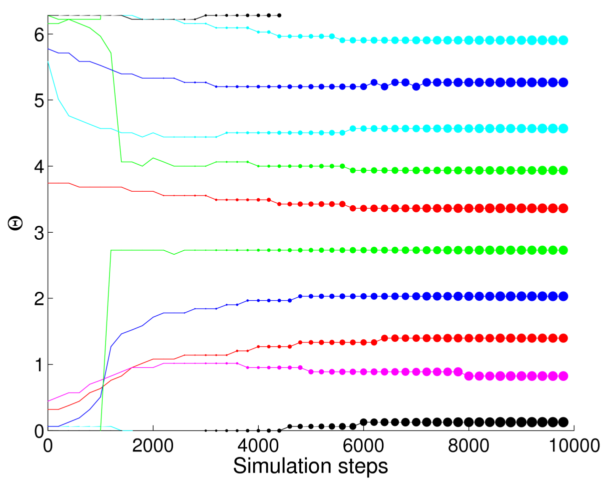


Figure 0.43.: Centers over time

After training, the weights of each neuron show strong centers of preferred stimuli, and after sorting according to center order, the weight matrix strongly resembles the optimal one, as shown in Figure 0.44.

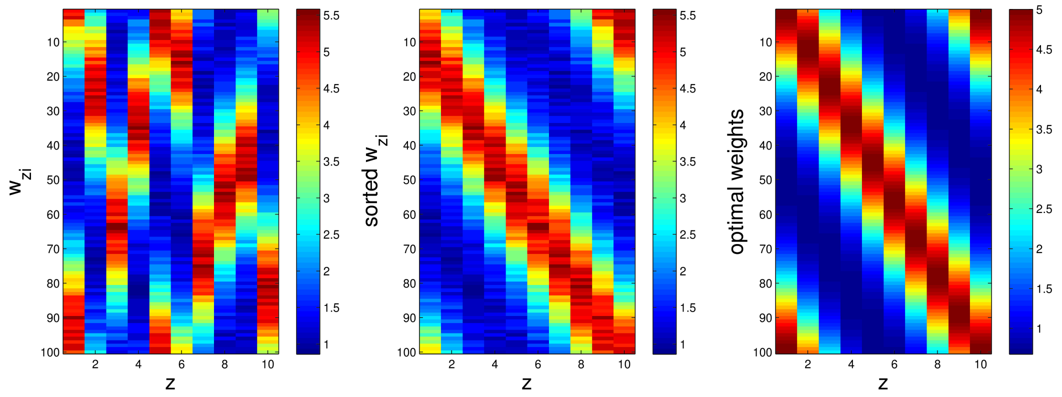


Figure 0.44.: Resulting weights of the z-neurons

Using the calculated, sorted weights and the firing sequence of the output neurons, one can reconstruct original input stimuli with high precision. Note that this should only serve as a performance measure, as does the calculated overall mean-square-error of 0.1877. Using 20000 training steps, 100 z-neurons and a learning rate schedule, the mean-square-error can be lowered to 0.1667.

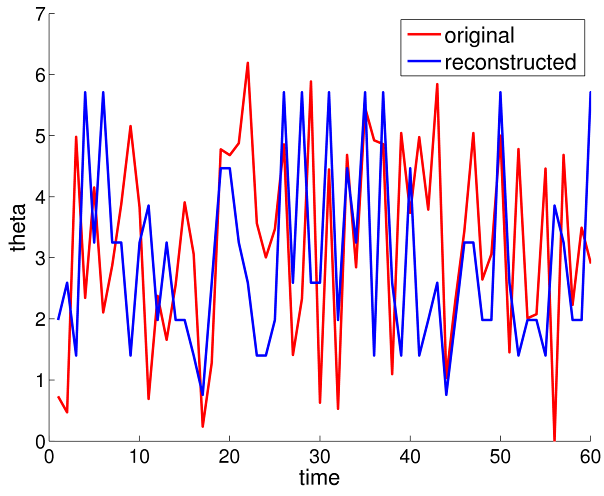


Figure 0.45.: Reconstructed input signal at the beginning of training

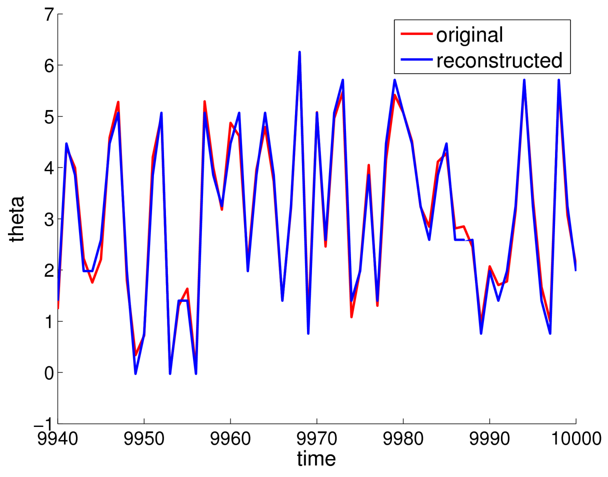


Figure 0.46.: Reconstructed input signal at the end of training

0.31. Dynamic Network Size

Learning the weights in the second layer also allows a dynamic network size, which changes not only the maximum likelihood that can be achieved, but also allows for a more natural growth pattern (new neurons may either grow or join the network due to synaptic growth). Also, if neurons die, the network performance is hardly affected due to adaptation of the other neurons.

The simulations where done using 20000 time steps (again drawn from a uniform distribution), with parameters winit= 4 and a constant learning rate $\eta = 0.03$.

0.31.1. Stalling Network

In this simulation of a stalling network, 10 neurons get trained for 2500 steps, then 5 neurons get removed and the training continues. As can be seen from the following plots, the log-likelihood drops when the network is reduced, but adapts by shifting the centers of the remaining neurons and settles after about 1000 time steps to a lower but stable value.

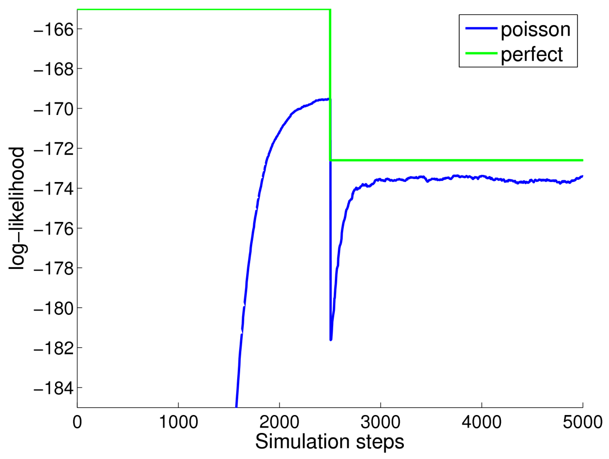


Figure 0.47.: Stalling network: Log-likelihood over time

During the adaptation, the weight change is slightly increased but, since the changes are made gradually, not significant.

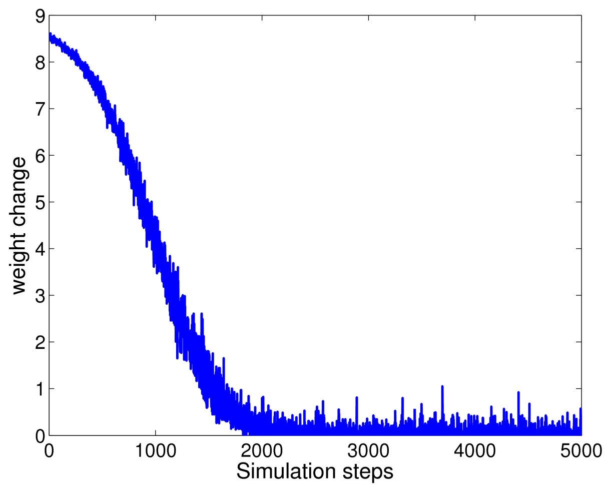


Figure 0.48.: Stalling network: Weight change over time

From the neuron center plot in the next Figure one can see that at time step 2500 the network is sufficiently adapted with roughly equally spaced neuron centers. After removal of some of the neurons, the remaining ones adapt their weights to make up for lost accuracy.

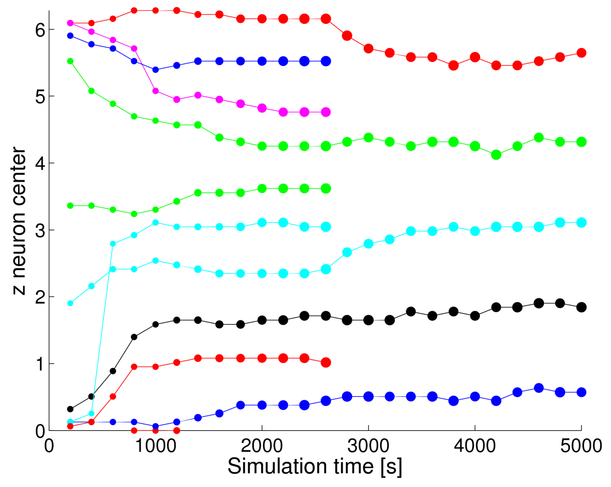


Figure 0.49.: Stalling network: Center development over time

0.31.2. Growing Network

In this simulation of a growing network, 5 neurons get trained for 2500 steps, then 5 neurons are added and the training continues. As can be seem from the following plots, the log-likelihood suffers a minor drop when the new neurons are joined (since the constant weight initialization affects the performance), but settles at a higher level after about 1000 time steps.

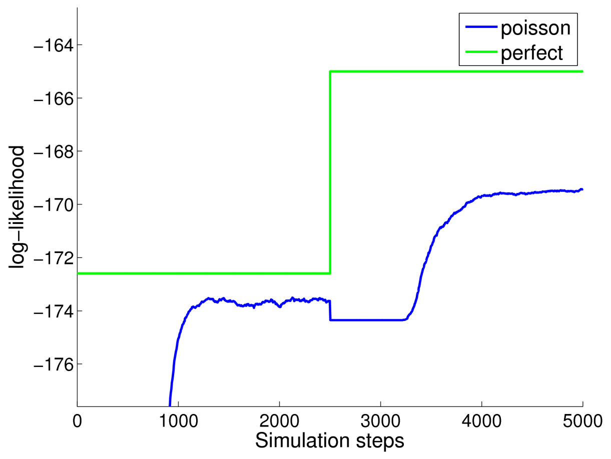


Figure 0.50.: Growing network: Log-likelihood over time

During the adaptation, the weight change shows a definite peak, since the weight initialization introduces very high weights so that new neurons are very likely activated until their weights adjust to correct values.

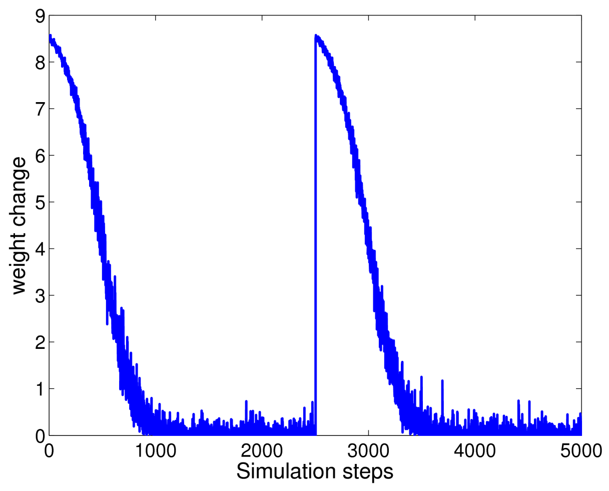


Figure 0.51.: Growing network: Weight change over time

As can be seen from the following plot, when the new neurons are added at time-step 2500, the original neurons's centers remain the same, but each new neuron specializes on a center in between.

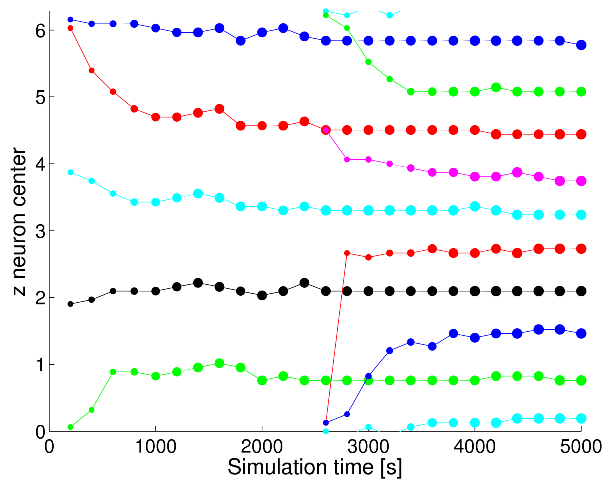


Figure 0.52.: Growing network: Center development over time

0.32. Unequally Distributed Θ_i's

Using the same distorted sensory neurons as in section 0.27, training is performed in exactly the same way as in the previous section, but with the introduction of a normalization term that compensates the varying population response sum. Please refer to section 0.5 for a discussion. As parameters a weight initialization of 3.6 was used, with a constant learning rate of 0.01, again over 10000 time steps. As one can see from the following plots, training takes longer than in the previous simulations, which is caused by the additional complexity of estimating said normalization factor.

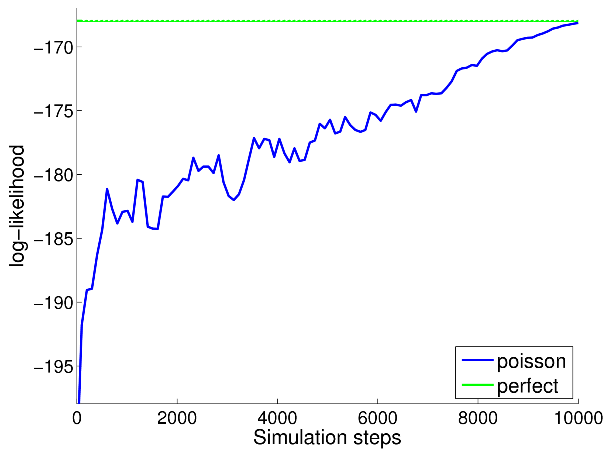


Figure 0.53.: Unequal Θ_i's: Log-likelihood over time

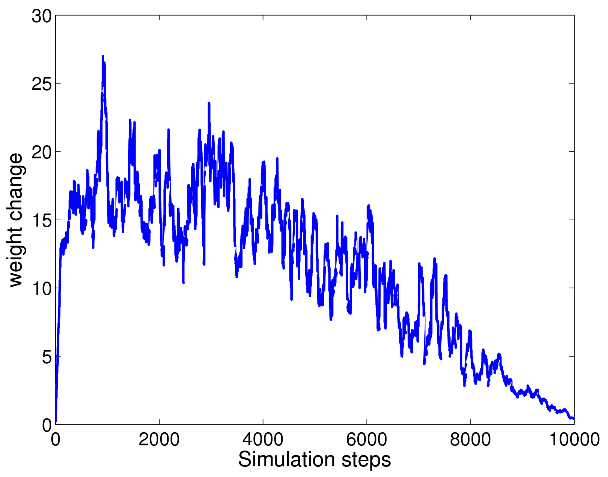


Figure 0.54.: Unequal Θ_i's: Weight change over time

The same weights as presented in the static case in Figure 0.37 evolved.

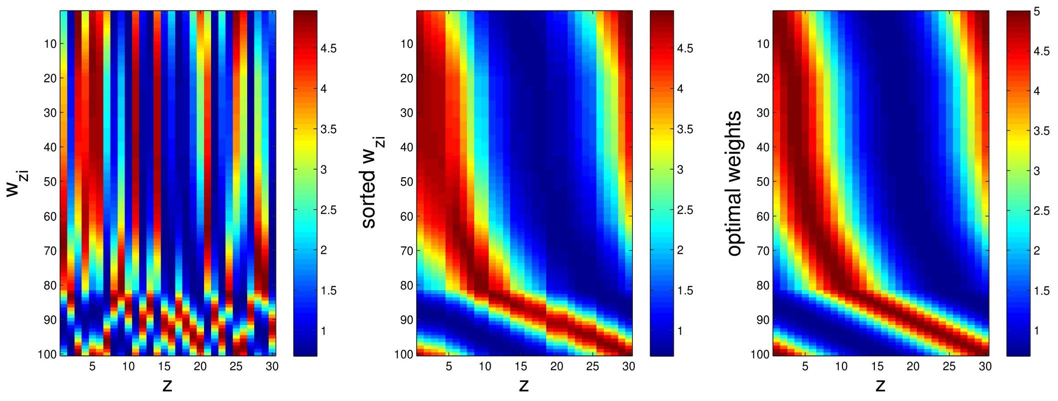


Figure 0.55.: Unequal Θ_i's: Weights of the z-neurons

However, the log-likelihood calculation differs (please refer to Section 0.5 for exact formulation) and a correct estimate is made, as can be seen in the following figure.

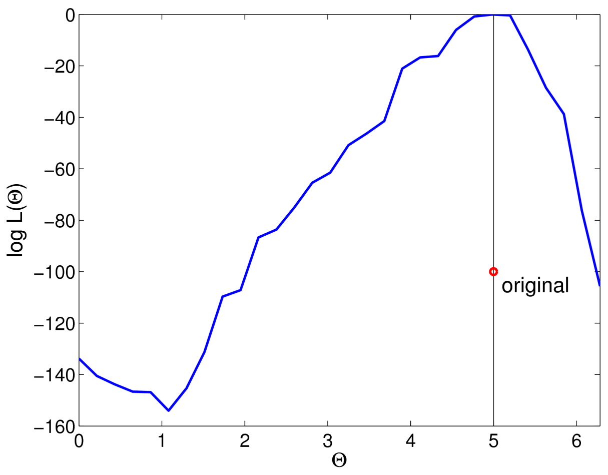


Figure 0.56.: Unequal Θ_i's: Calculated log-likelihood of specific stimulus

Optimality

To find a network and configuration best suited to solve the task chosen, both of the SEM learning algorithms in Section 0.5 where applied and evaluated.

0.33. Poisson SEM Evaluation

Using 20 neurons in the SEM layer, 10000 simulation steps and $c = 5$ as parameters, the log-likelihood of networks with different parameters of learning rate eta and weight initialization winit was calculated after training, as shown in Figure 0.57. The optimal likelihood was calculated to -167.74, the best found by learning -169.48 at winit 3.63 and eta 0.06. However, from the results one can conclude that finding a reasonable performance is relatively easy to achieve since the predominant part of the configurations tried achieved passable performance.

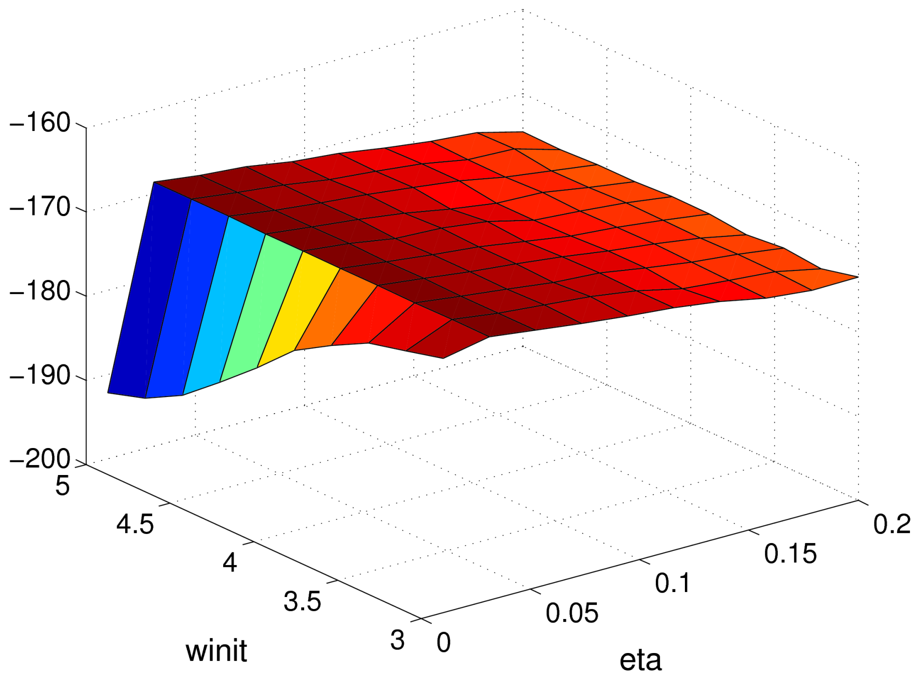


Figure 0.57.: Performance depending on eta and winit

Using parameters winit= 5, eta= 0.05 and 10000 simulation steps, different quantities of neurons (between 5 and 45) in the SEM layer were evaluated, the results are shown in Figure 0.58. One can see that larger network size, as expected, increases performance while requiring longer training. A reasonable choice of neuron quantity could be 20, which should be converged (in this task) after 5000 simulation steps.

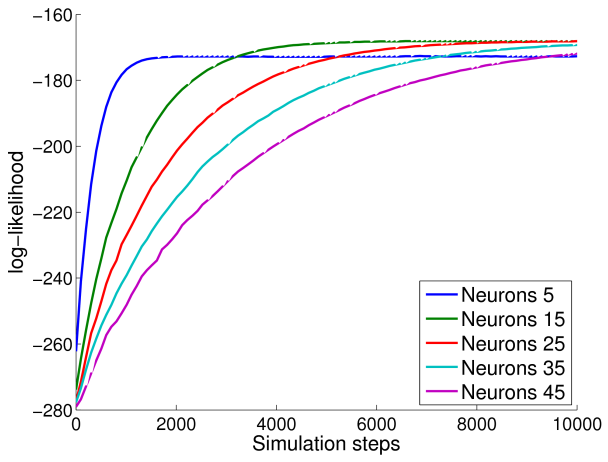


Figure 0.58.: Performance depending on neuron quantity

0.34. Binary SEM Evaluation

Using a binary version of the task requires the introduction of a threshold, which was the first parameter evaluated. Using winit= 0 (which is always the case), 20 z-neurons, a learning rate eta of 0.01 and 20000 simulation steps, different thresholds ranging between $[0,5]$ were evaluated, as shown in Figure 0.59. From the simulation results one can see that a threshold of 2 seems to be the best choice.

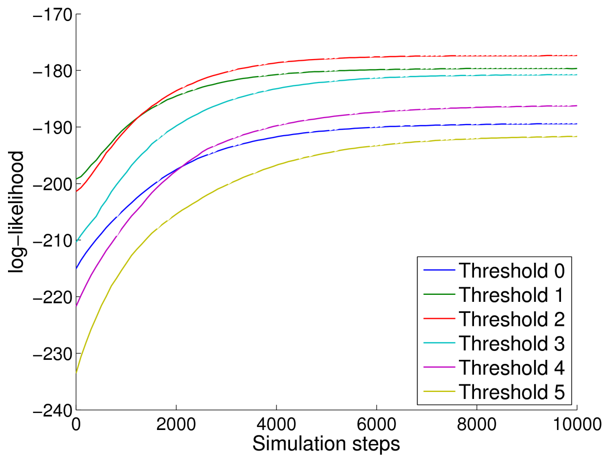


Figure 0.59.: Performance depending on threshold

Using the same parameters as in the previous simulation, the performance of different network sizes was evaluated, as depicted in Figure 0.60.

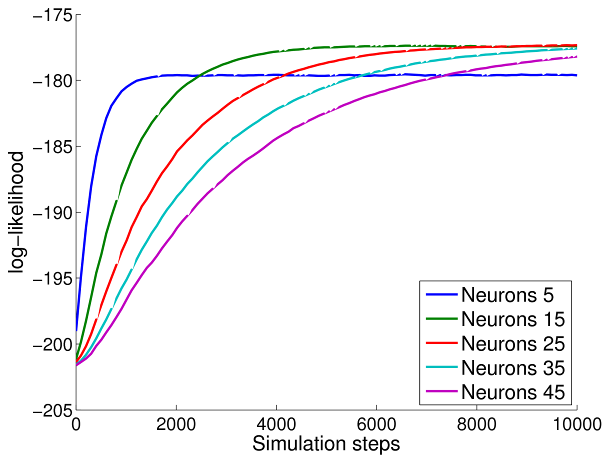


Figure 0.60.: Performance depending on network size

The best performance found was -177.42 using winit -0.150000 and eta 0.0147, which was used for the later comparison of the algorithms.

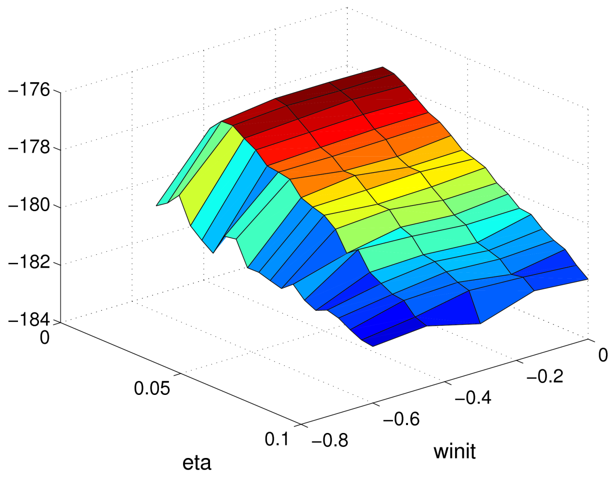


Figure 0.61.: Performance depending on eta and winit

0.35. Comparison

Using the previously found optimal parameters, a comparison of both the original and the extended SEM (without normalization) was made. As one can see, in this task (with Poisson noise) the extended SEM algorithm converges faster and to a higher log-likelihood than the original SEM method. However, it should be noted that this could be seen as an unfair comparison since the original SEM method's performance suffers from the introduction of the simple threshold mechanism. The author supports this view, but since most real-world tasks are not binary a similar pre-processing is unavoidable.

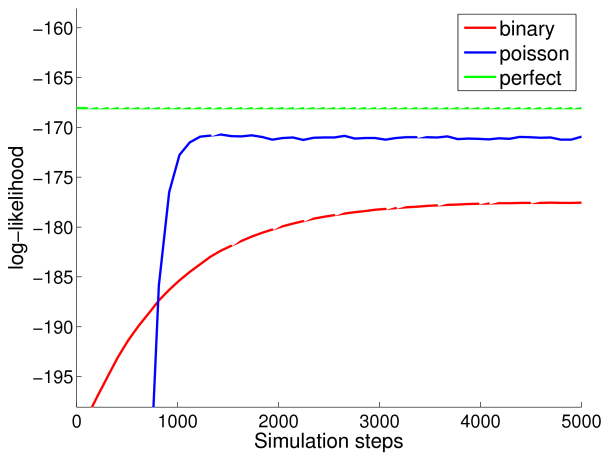


Figure 0.62.: Comparison of binary and Poisson method

Part IV.

Conclusion and Outlook

In this thesis the SEM learning method was applied to two relatively different fields of research. After a short introduction to the general background in Chapter I, the theoretical foundations are described in Section 0.5. In Chapter II, it was shown how the extended SEM method can be used to solve a number of tasks (unsupervised object recognition and, in combination with a simple pre-processing mechanism, basic tracking) using both simulation-generated (Section 0.21) and real-world data (Section 0.22). The requirements of online computation and real-time computation have been fulfilled, the latter with regard to the size of the network used (Section 0.24). Another requirement, the integration into an existing Java framework was fulfilled in combination with the pre-existing PCSIM neuron simulator (to avoid code duplication and allow further usage). From experience with the described tasks, the author argued that more complex tasks can be solved with the described framework, with possible need of appropriate pre-processing (depending on the problem) and multiple SEM layers (Section 0.25).

However, using SEM in the current architecture has swiftly shown its limitations. When objects were transformed over time (e.g. by translation, scaling or similar) either a large number of neurons were required to brute-force model each possible state of the object (which required large training effort), or a pre-processing mechanism which compensated the transformation at large. In its present state, SEM can not compete with specialized, complex state-of-the-art computer vision methods.

The previous argument does not necessarily have to be a drawback: Due to the relative simplicity of the method used, an implementation on special-purpose hardware (e.g. (Khan, 2008)) or small-scale embedded systems might be worth considering. Future work in an application-wise context could be manifold, since the software developed allows easy integration of other neuromorphic hardware (e.g. a hardware cochlea (Chan, 2007)), possibly with sensor fusion of multiple input devices. Even when using only one hardware retina, the author argues that it would be possible to build a basic gesture recognition system (mostly limited by the low resolution of the DVS128), let alone if multiple sensors would be used. Also, further studies of imprinting favorable behavior using reinforcement modulation of the SEM parameters come to mind. For this purpose the author would propose an extension of the existing simulator SpikeSim, which could with reasonable effort be extended to stereo-vision and complex 3D tasks, if desired with acoustic output.

Chapter 0.25 contains the second main part, where previous results of (Movshon, 2006) have been introduced (Section 0.26) and verified (Section 0.27). Although previous work was focused on sensory information processing, the method presented here can be used as a general information decoding strategy when a population code is used, e.g. when encoding movement direction in the motor cortex(Georgopoulos, 1986) or orientation in the striate cortex (Vogels, 1990). It was shown how a SEM based approach can reproduce a previously pre-calculated optimal readout method using a biologically plausible learning mechanism (Section 0.29). If a readout neuron's identity was associated with the value of the input variable which generates the highest response, the input signal was reproduced with surprising accuracy, given the noise level of the encoding response. In the solution described, it was argued that through a dynamic, learning system the biological applicability was widened: The sum of the population's response does not have to be constant, which was argued to be a bold claim in an error-prone, grown neural network. Furthermore, through adaptation correct behavior can be ensured when the encoding or readout population size changes, or the activation function of the encoding neurons is altered (Sections 0.28 and 0.31).

In the current simulation a discrete time step was used, the STDP-learning curve had perfect rectangular shape and an ideal WTA mechanism was assumed. Although the argument that SEM can be implemented in a biologically realistic context holds, a simulation that reflects such a context would be a prime candidate for future work. Such a simulation could show how training duration and performance are altered when using imperfect neural elements or more plausible STDP-shapes. Since external variation of the learning and/or spiking rate also showed promising results in simulations that were not presented in this work, further experiments that make use of these features would be commendable (possibly in context of perceptual learning (Gilbert, 2001)). The association of a readout neuron to its maximum response center would also be of interest. The author would propose a mechanism with lateral connectivity in a neighborhood of readout neurons. When such a neuron fires and adapts to the current input, neurons in its spatial surrounding are also likely to fire. This would lead to neuron fields similar to self-organizing maps, which have been found in several animal cortices (Kohonen, 2002). However, since the current SEM theory does not make predictions when using the required k-WTA mechanism, additional research would be necessary.

Appendix

.1. Experiment 1 Configuration

```
<?xml version="1.0" ?>
<SpikingExperiment>
  <Global>
    <!-- some variables here -->
    <ParameterString id="test1" />
    <ParameterInt time_step="1000" write="1"/>
    <ParameterDouble dt="0.01" write="1"/>
    <ParameterDouble min_delay="0.00001" />
    <ParameterDouble max_delay="0.0001" />
    <ParameterInt construction_prng_seed="349871" />
    <ParameterInt simulation_prng_seed="345678" />
  </Global>

  <VarScheduleDouble id="etaschedule1">
    <ParameterString target_id="wta1" />
    <ParameterString target_var="eta" />
    <ParameterInt advance_step="100" write="1"/>
    <ParameterDouble add_factor="0" write="1"/>
    <ParameterDouble mult_factor="0.995" write="1"/>
    <ParameterDouble min="0.001" write="1"/>
    <ParameterDouble max="1" write="1"/>
    <ParameterDouble init="0.1" write="1"/>
    <ParameterBool run_at_start="true"/>
  </VarScheduleDouble>

  <SemCircuitElement type="SquarePulseResponse" id="
      squarepulseresponse1" >
    <ParameterDouble a = "1" write="1"/>
    <ParameterDouble tau = "0.05" write="1"/>
  </SemCircuitElement>

  <SemCircuitElement type="GaussNeuron" id="gaussneuron1">
    <ParameterDouble min_bias="-5" write="1"/>
    <ParameterDouble max_bias="0" write="1"/>
    <ParameterDouble init_bias="-0.1" />
    <ParameterDouble eta_mult_bias="1" write="1"/>
    <ParameterDouble init_cw="0" />
    <ParameterDouble eta_mult_cw="1" write="1"/>
    <ParameterDouble sigma_sq="20" write="1"/>
    <ParameterBool use_approx="true" write="1" />
  </SemCircuitElement>

  <SemCircuitElement type="GaussSynapse" id="gausssynapse1">
    <ParameterDouble w_init="0.5" />
    <ParameterDouble sigma_sq="20" write="1"/>
    <ParameterDouble min_mean="0" write="1"/>
    <ParameterDouble max_mean="20" write="1"/>
    <ParameterDouble eta_mult_theta="1" write="1"/>
  </SemCircuitElement>

  <SemCircuitElement type="WTALowCutoff" id="wta1">
    <ParameterDouble spiking_rate="50" write="1"/>
    <ParameterDouble eta="0.01" write="1"/>
    <ParameterDouble hard_cut_off="10" write="1"/>
    <ParameterDouble soft_cut_off="0" write="1"/>
  </SemCircuitElement>

  <Layers>
    <InputLayer id="inputlayer1">
      <ParameterInt w="128" />
      <ParameterInt h="128" />
    </InputLayer>

    <SemCircuit id="semelement1" spikeresponse="squarepulseresponse1"
        neuronmodel="gaussneuron1" synapsemodel="gausssynapse1"
        spikingmechanism="wta1">
      <ParameterInt input_w="128" />
      <ParameterInt input_h="128" />
      <ParameterInt output_w="8" />
      <ParameterInt output_h="8" />
    </SemCircuit>
  </Layers>

  <Connections>
    <SimpleConnection>
      <ParameterString src="inputlayer1" />
      <ParameterString dst="semelement1" />
    </SimpleConnection>
  </Connections>

</SpikingExperiment>
```

.2. Experiment 2 Configuration

```
<?xml version="1.0" ?>
<SpikingExperiment>
  <Global>
    <!-- some variables here -->
    <ParameterString id="test1" />
    <ParameterInt time_step="1000" write="1"/>
    <ParameterDouble dt="0.01" write="1"/>
    <ParameterDouble min_delay="0.00001" />
    <ParameterDouble max_delay="0.0001" />
    <ParameterInt construction_prng_seed="349871" />
    <ParameterInt simulation_prng_seed="345678" />
  </Global>

  <SemCircuitElement type="SquarePulseResponse" id="
      squarepulseresponse1" >
    <ParameterDouble a = "1" write="1"/>
    <ParameterDouble tau = "0.05" write="1"/>
  </SemCircuitElement>

  <SemCircuitElement type="SquarePulseResponse" id="
      squarepulseresponse2" >
    <ParameterDouble a = "1" write="1"/>
    <ParameterDouble tau = "0.5" write="1"/>
  </SemCircuitElement>

  <SemCircuitElement type="GaussNeuron" id="gaussneuron1">
    <ParameterDouble min_bias="-5" write="1"/>
    <ParameterDouble max_bias="0" write="1"/>
    <ParameterDouble init_bias="-0.1" />
    <ParameterDouble eta_mult_bias="1" write="1"/>
    <ParameterDouble init_cw="0" />
    <ParameterDouble eta_mult_cw="1" write="1"/>
    <ParameterDouble sigma_sq="20" write="1"/>
    <ParameterBool use_approx="true" write="1" />
  </SemCircuitElement>

  <SemCircuitElement type="PoissonNeuron" id="poissonneuron1">
    <ParameterDouble min_bias="-5" write="1"/>
    <ParameterDouble max_bias="0" write="1"/>
    <ParameterDouble init_bias="-0.1" />
    <ParameterDouble eta_mult_bias="0" write="1"/>
    <ParameterDouble init_cw="0" />
    <ParameterDouble mcorr="10" write="1"/>
    <ParameterDouble eta_mult_cw="1" write="1"/>
    <ParameterBool use_approx="true" write="1" />
  </SemCircuitElement>


  <SemCircuitElement type="GaussSynapse" id="gausssynapse1">
    <ParameterDouble w_init="0.5" />
    <ParameterDouble sigma_sq="20" write="1"/>
    <ParameterDouble min_mean="0" write="1"/>
    <ParameterDouble max_mean="20" write="1"/>
    <ParameterDouble eta_mult_theta="1" write="1"/>
  </SemCircuitElement>

  <SemCircuitElement type="ShiftedPoissonSynapse" id="poissonsynapse1
      ">
    <ParameterDouble w_init="10" />
    <ParameterDouble m="10" write="1"/>
    <ParameterDouble max_theta="12" write="1"/>
    <ParameterDouble min_real_theta_for_learning="-3" write="1"/>
    <ParameterDouble eta_mult_theta="1" write="1"/>
  </SemCircuitElement>

  <SemCircuitElement type="WTALowCutoff" id="wta1">
    <ParameterDouble spiking_rate="50" write="1"/>
    <ParameterDouble eta="0.01" write="1"/>
    <ParameterDouble hard_cut_off="10" write="1"/>
    <ParameterDouble soft_cut_off="0" write="1"/>
  </SemCircuitElement>

  <VarScheduleDouble id="etaschedule1">
    <ParameterString target_id="wta1" />
    <ParameterString target_var="eta" />
    <ParameterInt advance_step="100" write="1"/>
    <ParameterDouble add_factor="0" write="1"/>
    <ParameterDouble mult_factor="0.995" write="1"/>
    <ParameterDouble min="0.001" write="1"/>
    <ParameterDouble max="1" write="1"/>
    <ParameterDouble init="0.1" write="1"/>
    <ParameterBool run_at_start="true"/>
  </VarScheduleDouble>

  <SemCircuitElement type="WTASpikingMechanism" id="wta2">
    <ParameterDouble spiking_rate="10" write="1"/>
    <ParameterDouble eta="0.005" write="1"/>
  </SemCircuitElement>

  <VarScheduleDouble id="etaschedule2">
    <ParameterString target_id="wta2" />
    <ParameterString target_var="eta" />
    <ParameterInt advance_step="100" write="1"/>
    <ParameterDouble add_factor="0" write="1"/>
    <ParameterDouble mult_factor="0.985" write="1"/>
    <ParameterDouble min="0.0001" write="1"/>
    <ParameterDouble max="1" write="1"/>
    <ParameterDouble init="0.1" write="1"/>
    <ParameterBool run_at_start="true"/>
  </VarScheduleDouble>

  <Layers>
    <InputLayer id="inputlayer1">
      <ParameterInt w="64" />
      <ParameterInt h="64" />
    </InputLayer>

    <SemCircuit id="semelement1" spikeresponse="squarepulseresponse1"
        neuronmodel="gaussneuron1" synapsemodel="gausssynapse1"
        spikingmechanism="wta1">
      <ParameterInt input_w="64" />
      <ParameterInt input_h="64" />
      <ParameterInt output_w="16" />
      <ParameterInt output_h="16" />
    </SemCircuit>
    <SemCircuit id="semelement2" spikeresponse="squarepulseresponse2"
```

```
            neuronmodel="poissonneuron1" synapsemodel="poissonsynapse1"
            spikingmechanism="wta2">
          <ParameterInt input_w="16" />
          <ParameterInt input_h="16" />
          <ParameterInt output_w="2" />
          <ParameterInt output_h="2" />
        </SemCircuit>

    </Layers>

    <Connections>
      <SimpleConnection>
        <ParameterString src="inputlayer1" />
        <ParameterString dst="semelement1" />
      </SimpleConnection>
      <SimpleConnection>
        <ParameterString src="semelement1" />
        <ParameterString dst="semelement2" />
      </SimpleConnection>
    </Connections>

</SpikingExperiment>
```

.3. Experiment 3 Configuration

```
<?xml version="1.0" ?>
<SpikingExperiment>
      <Global>
      <!-- some variables here -->
      <ParameterString id="test1" />
      <ParameterInt time_step="1000" write="1"/>
      <ParameterDouble dt="0.01" write="1"/>
      <ParameterDouble min_delay="0.00001" />
      <ParameterDouble max_delay="0.0001" />
      <ParameterInt construction_prng_seed="349871" />
      <ParameterInt simulation_prng_seed="345678" />
    </Global>

    <SemCircuitElement type="SquarePulseResponse" id="
          squarepulseresponse1" >
      <ParameterDouble a = "1" write="1"/>
      <ParameterDouble tau = "0.05" write="1"/>
    </SemCircuitElement>
    <SemCircuitElement type="AlphaResponse" id="alpharesponse1" >
      <ParameterDouble tau = "0.03" write="1"/>
    </SemCircuitElement>

    <SemCircuitElement type="SquarePulseResponse" id="
          squarepulseresponse2" >
      <ParameterDouble a = "1" write="1"/>
      <ParameterDouble tau = "0.5" write="1"/>
    </SemCircuitElement>

```

```
  <SemCircuitElement type="GaussNeuron" id="gaussneuron1">
    <ParameterDouble min_bias="-5" write="1"/>
    <ParameterDouble max_bias="0" write="1"/>
    <ParameterDouble init_bias="-0.1" />
    <ParameterDouble eta_mult_bias="1" write="1"/>
    <ParameterDouble init_cw="0" />
    <ParameterDouble eta_mult_cw="1" write="1"/>
    <ParameterDouble sigma_sq="20" write="1"/>
    <ParameterBool use_approx="true" write="1" />
  </SemCircuitElement>

  <SemCircuitElement type="GaussNeuron" id="gaussneuron2">
    <ParameterDouble min_bias="-5" write="1"/>
    <ParameterDouble max_bias="0" write="1"/>
    <ParameterDouble init_bias="-0.1" />
    <ParameterDouble eta_mult_bias="1" write="1"/>
    <ParameterDouble init_cw="-100000" />
    <ParameterDouble eta_mult_cw="1" write="1"/>
    <ParameterDouble sigma_sq="1" write="1"/>
    <ParameterBool use_approx="true" write="1" />
  </SemCircuitElement>

  <SemCircuitElement type="PoissonNeuron" id="poissonneuron1">
    <ParameterDouble min_bias="-5" write="1"/>
    <ParameterDouble max_bias="0" write="1"/>
    <ParameterDouble init_bias="-0.1" />
    <ParameterDouble eta_mult_bias="0" write="1"/>
    <ParameterDouble init_cw="0" />
    <ParameterDouble mcorr="10" write="1"/>
    <ParameterDouble eta_mult_cw="1" write="1"/>
    <ParameterBool use_approx="true" write="1" />
  </SemCircuitElement>

  <SemCircuitElement type="GaussSynapse" id="gausssynapse1">
    <ParameterDouble w_init="0.5" />
    <ParameterDouble sigma_sq="20" write="1"/>
    <ParameterDouble min_mean="0" write="1"/>
    <ParameterDouble max_mean="20" write="1"/>
    <ParameterDouble eta_mult_theta="1" write="1"/>
  </SemCircuitElement>

  <SemCircuitElement type="GaussSynapse" id="gausssynapse2">
    <ParameterDouble w_init="10" />
    <ParameterDouble sigma_sq="1" write="1"/>
    <ParameterDouble min_mean="0" write="1"/>
    <ParameterDouble max_mean="20" write="1"/>
    <ParameterDouble eta_mult_theta="1" write="1"/>
  </SemCircuitElement>

  <SemCircuitElement type="ShiftedPoissonSynapse" id="poissonsynapse1
     ">
    <ParameterDouble w_init="10" />
    <ParameterDouble m="10" write="1"/>
    <ParameterDouble max_theta="12" write="1"/>
    <ParameterDouble min_real_theta_for_learning="−3" write="1"/>
    <ParameterDouble eta_mult_theta="1" write="1"/>
  </SemCircuitElement>

  <SemCircuitElement type="WTALowCutoff" id="wta1">
    <ParameterDouble spiking_rate="50" write="1"/>
    <ParameterDouble eta="0.01" write="1"/>
    <ParameterDouble hard_cut_off="10" write="1"/>
    <ParameterDouble soft_cut_off="0" write="1"/>
  </SemCircuitElement>

  <VarScheduleDouble id="etaschedule1">
    <ParameterString target_id="wta1" />
    <ParameterString target_var="eta" />
    <ParameterInt advance_step="100" write="1"/>
    <ParameterDouble add_factor="0" write="1"/>
    <ParameterDouble mult_factor="0.995" write="1"/>
    <ParameterDouble min="0.001" write="1"/>
    <ParameterDouble max="1" write="1"/>
    <ParameterDouble init="0.1" write="1"/>
    <ParameterBool run_at_start="true"/>
  </VarScheduleDouble>

  <SemCircuitElement type="WTASpikingMechanism" id="wta2">
    <ParameterDouble spiking_rate="10" write="1"/>
    <ParameterDouble eta="0.005" write="1"/>
  </SemCircuitElement>

  <VarScheduleDouble id="etaschedule2">
    <ParameterString target_id="wta2" />
    <ParameterString target_var="eta" />
    <ParameterInt advance_step="100" write="1"/>
    <ParameterDouble add_factor="0" write="1"/>
    <ParameterDouble mult_factor="0.985" write="1"/>
    <ParameterDouble min="0.0001" write="1"/>
    <ParameterDouble max="1" write="1"/>
    <ParameterDouble init="0.1" write="1"/>
    <ParameterBool run_at_start="true"/>
  </VarScheduleDouble>

  <Layers>
    <InputLayer id="tracker0">
      <ParameterInt w="48" />
      <ParameterInt h="48" />
     </InputLayer>
    <InputLayer id="tracker1">
      <ParameterInt w="48" />
      <ParameterInt h="48" />
     </InputLayer>
    <SemCircuit id="semelement1" spikeresponse="squarepulseresponse1"
        neuronmodel="gaussneuron1" synapsemodel="gausssynapse1"
        spikingmechanism="wta1">
      <ParameterInt input_w="48" />
      <ParameterInt input_h="96" />
      <ParameterInt output_w="12" />
      <ParameterInt output_h="24" />

      <ParameterInt num_inputs_per_patch_w="48" />
      <ParameterInt num_inputs_per_patch_h="48" />
      <ParameterInt num_neurons_w = "12" />
      <ParameterInt num_neurons_h = "12" />

      <ParameterInt step_x="48" />
      <ParameterInt step_y="48" />

    </SemCircuit>
    <SemCircuit id="semelement2" spikeresponse="squarepulseresponse2"
        neuronmodel="poissonneuron1" synapsemodel="poissonsynapse1"
        spikingmechanism="wta2">
      <ParameterInt input_w="12" />
      <ParameterInt input_h="24" />
      <ParameterInt output_w="4" />
      <ParameterInt output_h="2" />

      <ParameterInt num_inputs_per_patch_w="12" />
      <ParameterInt num_inputs_per_patch_h="12" />
      <ParameterInt num_neurons_w = "4" />
      <ParameterInt num_neurons_h = "1" />

      <ParameterInt step_x="12" />
      <ParameterInt step_y="12" />

    </SemCircuit>

  </Layers>

  <Connections>
    <SimpleConnection>
      <ParameterString src="tracker0" />
      <ParameterInt src_x = "0" />
      <ParameterInt src_y = "0" />
      <ParameterString dst="semelement1" />
      <ParameterInt dst_x = "0" />
      <ParameterInt dst_y = "0" />
      <ParameterInt w = "48" />
      <ParameterInt h = "48" />
    </SimpleConnection>
    <SimpleConnection>
      <ParameterString src="tracker1" />
      <ParameterInt src_x = "0" />
      <ParameterInt src_y = "0" />
      <ParameterString dst="semelement1" />
      <ParameterInt dst_x = "0" />
      <ParameterInt dst_y = "48" />
      <ParameterInt w = "48" />
      <ParameterInt h = "48" />
    </SimpleConnection>
    <SimpleConnection>
```

```
      <ParameterString src="semelement1" />
      <ParameterInt src_x = "0" />
      <ParameterInt src_y = "0" />
      <ParameterString dst="semelement2" />
      <ParameterInt dst_x = "0" />
      <ParameterInt dst_y = "0" />
      <ParameterInt w = "12" />
      <ParameterInt h = "24" />
    </SimpleConnection>
  </Connections>

</SpikingExperiment>
```

.4. Experiment 4 Configuration

```
<?xml version="1.0" ?>
<SpikingExperiment>
  <Global>
    <!-- some variables here -->
    <ParameterString id="test1" />
    <ParameterInt time_step="1000" write="1"/>
    <ParameterDouble dt="0.01" write="1"/>
    <ParameterDouble min_delay="0.00001" />
    <ParameterDouble max_delay="0.0001" />
    <ParameterInt construction_prng_seed="349871" />
    <ParameterInt simulation_prng_seed="345678" />
  </Global>

  <SemCircuitElement type="SquarePulseResponse" id="
      squarepulseresponse1" >
    <ParameterDouble a = "1" write="1"/>
    <ParameterDouble tau = "0.1" write="1"/>
  </SemCircuitElement>
  <SemCircuitElement type="AlphaResponse" id="alpharesponse1" >
    <ParameterDouble tau = "0.03" write="1"/>
  </SemCircuitElement>

  <SemCircuitElement type="SquarePulseResponse" id="
      squarepulseresponse2" >
    <ParameterDouble a = "1" write="1"/>
    <ParameterDouble tau = "0.5" write="1"/>
  </SemCircuitElement>

  <SemCircuitElement type="GaussNeuron" id="gaussneuron1">
    <ParameterDouble min_bias="-5" write="1"/>
    <ParameterDouble max_bias="0" write="1"/>
    <ParameterDouble init_bias="-0.1" />
    <ParameterDouble eta_mult_bias="1" write="1"/>
    <ParameterDouble init_cw="0" />
    <ParameterDouble eta_mult_cw="1" write="1"/>
    <ParameterDouble sigma_sq="1" write="1"/>
    <ParameterBool use_approx="true" write="1" />
  </SemCircuitElement>

  <SemCircuitElement type="GaussNeuron" id="gaussneuron2">
    <ParameterDouble min_bias="-5" write="1"/>
    <ParameterDouble max_bias="0" write="1"/>
    <ParameterDouble init_bias="-0.1" />
    <ParameterDouble eta_mult_bias="1" write="1"/>
    <ParameterDouble init_cw="-100000" />
    <ParameterDouble eta_mult_cw="1" write="1"/>
    <ParameterDouble sigma_sq="1" write="1"/>
    <ParameterBool use_approx="true" write="1" />
  </SemCircuitElement>

  <SemCircuitElement type="PoissonNeuron" id="poissonneuron1">
    <ParameterDouble min_bias="-5" write="1"/>
    <ParameterDouble max_bias="0" write="1"/>
    <ParameterDouble init_bias="-0.1" />
    <ParameterDouble eta_mult_bias="0" write="1"/>
    <ParameterDouble init_cw="0" />
    <ParameterDouble mcorr="10" write="1"/>
    <ParameterDouble eta_mult_cw="1" write="1"/>
    <ParameterBool use_approx="true" write="1" />
  </SemCircuitElement>
  <SemCircuitElement type="PoissonNeuron" id="poissonneuron2">
    <ParameterDouble min_bias="-5" write="1"/>
    <ParameterDouble max_bias="0" write="1"/>
    <ParameterDouble init_bias="-0.1" />
    <ParameterDouble eta_mult_bias="0" write="1"/>
    <ParameterDouble init_cw="0" />
    <ParameterDouble mcorr="10" write="1"/>
    <ParameterDouble eta_mult_cw="1" write="1"/>
    <ParameterBool use_approx="true" write="1" />
  </SemCircuitElement>

  <SemCircuitElement type="GaussVTSynapse" id="gausssynapse1">
    <ParameterDouble w_init="0.5" write="1"/>
    <ParameterDouble sigma_sq="100" write="1"/>
    <ParameterDouble min_mean="0" write="1"/>
    <ParameterDouble max_mean="20" write="1"/>
    <ParameterDouble eta_mult_theta="1" write="1"/>
    <ParameterDouble init_lr="0.5" write="1"/>
    <ParameterDouble min_lr="0.001" write="1"/>
    <ParameterBool debug_display="false" write="1"/>
  </SemCircuitElement>

  <SemCircuitElement type="GaussKSynapse" id="gaussksynapse1">
    <ParameterDouble w_init="10.5" write="1"/>
    <ParameterDouble sigma_sq="1" write="1"/>
    <ParameterDouble min_mean="0" write="1"/>
    <ParameterDouble max_mean="20" write="1"/>
    <ParameterDouble eta_mult_theta="1" write="1"/>
    <ParameterDouble init_var="10" write="1"/>
    <ParameterDouble input_var="2" write="1"/>
    <ParameterDouble add_var="0.1" write="1"/>
    <ParameterBool debug_display="false" write="1"/>
```

```
</SemCircuitElement>

<SemCircuitElement type="GaussSynapse" id="gausssynapse2">
  <ParameterDouble w_init="10" write="1"/>
  <ParameterDouble sigma_sq="1" write="1"/>
  <ParameterDouble min_mean="0" write="1"/>
  <ParameterDouble max_mean="20" write="1"/>
  <ParameterDouble eta_mult_theta="1" write="1"/>
</SemCircuitElement>

<SemCircuitElement type="ShiftedPoissonVTSynapse" id="
    poissonsynapse1">
  <ParameterDouble w_init="10" />
  <ParameterDouble m="10" write="1"/>
  <ParameterDouble max_theta="12" write="1"/>
  <ParameterDouble min_real_theta_for_learning="-3" write="1"/>
  <ParameterDouble eta_mult_theta="1" write="1"/>
  <ParameterDouble init_lr="0.5" write="1"/>
  <ParameterDouble min_lr="0.001" write="1"/>
  <ParameterBool debug_display="false" write="1"/>
</SemCircuitElement>
<SemCircuitElement type="ShiftedPoissonSynapse" id="poissonsynapse2
    ">
  <ParameterDouble w_init="10" />
  <ParameterDouble m="10" write="1"/>
  <ParameterDouble max_theta="12" write="1"/>
  <ParameterDouble min_real_theta_for_learning="-3" write="1"/>
  <ParameterDouble eta_mult_theta="1" write="1"/>
</SemCircuitElement>

<SemCircuitElement type="WTALowCutoff" id="wta1">
  <ParameterDouble spiking_rate="50" write="1"/>
  <ParameterDouble eta="0.01" write="1"/>
  <ParameterDouble hard_cut_off="100" write="1"/>
  <ParameterDouble soft_cut_off="0" write="1"/>
</SemCircuitElement>

<VarScheduleDouble id="etaschedule1">
  <ParameterString target_id="wta1" />
  <ParameterString target_var="eta" />
  <ParameterInt advance_step="100" write="1"/>
  <ParameterDouble add_factor="0" write="1"/>
  <ParameterDouble mult_factor="0.999" write="1"/>
  <ParameterDouble min="0.001" write="1"/>
  <ParameterDouble max="1" write="1"/>
  <ParameterDouble init="1" write="1"/>
  <ParameterBool run_at_start="true"/>
</VarScheduleDouble>

<SemCircuitElement type="WTASpikingMechanism" id="wta2">
  <ParameterDouble spiking_rate="10" write="1"/>
  <ParameterDouble eta="0.005" write="1"/>
</SemCircuitElement>

```

```
  <VarScheduleDouble id=”etaschedule2”>
    <ParameterString target_id=”wta2” />
    <ParameterString target_var=”eta” />
    <ParameterInt advance_step=”100” write=”1”/>
    <ParameterDouble add_factor=”0” write=”1”/>
    <ParameterDouble mult_factor=”0.985” write=”1”/>
    <ParameterDouble min=”0.0001” write=”1”/>
    <ParameterDouble max=”1” write=”1”/>
    <ParameterDouble init=”0.1” write=”1”/>
    <ParameterBool run_at_start=”true”/>
  </VarScheduleDouble>

  <Layers>
    <InputLayer id=”tracker0”>
      <ParameterInt w=”48” />
      <ParameterInt h=”48” />
    </InputLayer>
    <SemCircuit id=”semelement1” spikeresponse=”squarepulseresponse1”
        neuronmodel=”gaussneuron1” synapsemodel=”gaussksynapse1”
        spikingmechanism=”wta1”>
      <ParameterInt input_w=”48” />
      <ParameterInt input_h=”48” />
      <ParameterInt output_w=”20” />
      <ParameterInt output_h=”20” />
    </SemCircuit>
    <SemCircuit id=”semelement2” spikeresponse=”squarepulseresponse2”
        neuronmodel=”poissonneuron2” synapsemodel=”poissonsynapse2”
        spikingmechanism=”wta2”>
      <ParameterInt input_w=”20” />
      <ParameterInt input_h=”20” />
      <ParameterInt output_w=”2” />
      <ParameterInt output_h=”2” />
    </SemCircuit>

  </Layers>

  <Connections>
    <SimpleConnection>
      <ParameterString src=”tracker0” />
      <ParameterString dst=”semelement1” />
    </SimpleConnection>
    <SimpleConnection>
      <ParameterString src=”semelement1” />
      <ParameterString dst=”semelement2” />
    </SimpleConnection>
  </Connections>

</SpikingExperiment>
```

Bibliography

[1] BI, G. AND POO, M. 2001. Synaptic modification by correlated activity: Hebb's postulate revisited. *Annual review of neuroscience 24*, 139. (Cited on pages 13 and 23.)

[2] BISHOP, C. M. 2006. *Pattern Recognition and Machine Learning (Information Science and Statistics)*. Springer-Verlag New York, Inc., Secaucus, NJ, USA. (Cited on page 25.)

[3] BLASDEL, G. 1992. Orientation selectivity, preference, and continuity in monkey striate cortex. *Journal of Neuroscience 12,* 8, 3139. (Cited on page 81.)

[4] CHAN, V., LIU, S., AND VAN SCHAIK, A. 2007. AER EAR: A matched silicon cochlea pair with address event representation interface. *Circuits and Systems I: Regular Papers, IEEE Transactions on 54,* 1, 48–59. (Cited on page 99.)

[5] DAN, Y. AND POO, M. 2006. Spike timing-dependent plasticity: from synapse to perception. *Physiological reviews 86,* 3, 1033. (Cited on page 23.)

[6] DELBRÜCK, T. 2008. Frame-free dynamic digital vision. In *Proceedings of Intl. Symposium on Secure-Life Electronics, Advanced Electronics for Quality Life and Society*. University of Tokyo, Tokyo, Japan, 21–26. (Cited on pages 35, 37, and 58.)

[7] DOUGLAS, R. AND MARTIN, K. 2004. Neuronal circuits of the neocortex. *Neuroscience 27,* 1, 419. (Cited on page 24.)

[8] ET.AL., R. B. 2007. Simulation of networks of spiking neurons: A review of tools and strategies. *Computational Neuroscience*. doi:10.1007/s10827-007-0038-6. (Cited on page 37.)

[9] GEORGOPOULOS, A., SCHWARTZ, A., AND KETTNER, R. 1986. Neuronal population coding of movement direction. *Science 233,* 4771, 1416. (Cited on pages 24 and 99.)

[10] GERSTNER, W. 2000. Population dynamics of spiking neurons: fast transients, asynchronous states, and locking. *Neural Computation 12,* 1, 43–89. (Cited on page 82.)

[11] GERSTNER, W. AND KISTLER, W. M. 2002. *Spiking Neuron Models: Single Neurons, Populations, Plasticity*. Cambridge University Press. (Cited on pages 21 and 28.)

[12] GILBERT, C., SIGMAN, M., AND CRIST, R. 2001. The neural basis of perceptual learning. *Neuron 31,* 5, 681–697. (Cited on page 99.)

[13] HABENSCHUSS, S. Novel methods for probabilistic inference and learning in spiking neural networks. M.S. thesis, University of Technology Graz. (Cited on page 28.)

[14] HODGKIN, A. AND HUXLEY, A. 1952. A quantitative description of membrane current and its application to conduction and excitation in nerve. *The Journal of physiology 117,* 4, 500. (Cited on page 22.)

[15] IZHIKEVICH, E. 2004. Which model to use for cortical spiking neurons? *Neural Networks, IEEE Transactions on 15,* 5, 1063–1070. (Cited on page 22.)

[16] JAYNES, E. AND BRETTHORST, G. 2003. *Probability theory: the logic of science*. Cambridge Univ Pr. (Cited on page 25.)

[17] JAZAYERI, M. AND MOVSHON, J. A. 2006. Optimal representation of sensory information by neural populations. *Nat Neurosci 9,* 5 (May), 690–696. (Cited on pages 13, 71, 73, and 99.)

[18] JOLIVET, R., KOBAYASHI, R., RAUCH, A., NAUD, R., SHINOMOTO, S., AND GERSTNER, W. 2008. A benchmark test for a quantitative assessment of simple neuron models. *Journal of neuroscience methods 169,* 2, 417–424. (Cited on page 22.)

[19] KAY, S. 1993. *Fundamentals of statistical signal processing: estimation theory*. (Cited on page 26.)

[20] KHAN, M., LESTER, D., PLANA, L., RAST, A., JIN, X., PAINKRAS, E., AND FURBER, S. 2008. SpiNNaker: mapping neural networks onto a massively-parallel chip multiprocessor. In *Neural Networks, 2008. IJCNN 2008.(IEEE World Congress on Computational Intelligence). IEEE International Joint Conference on*. IEEE, 2849–2856. (Cited on page 99.)

[21] KOCH, C. AND SCHUTTER, E. 1999. *Biophysics of computation: information processing in single neurons*. Vol. 428. Oxford University Press New York. (Cited on page 22.)

[22] KOHONEN, T. 2002. The self-organizing map. *Proceedings of the IEEE 78,* 9, 1464–1480. (Cited on page 100.)

[23] LICHTSTEINER, P., POSCH, C., AND DELBRUCK, T. 2008. A 128 128 120 db 15 us latency asynchronous temporal contrast vision sensor. *IEEE Journal of solid-state circuits 43,* 2. (Cited on pages 13 and 35.)

[24] MAASS, W. 2000. On the computational power of winner-take-all. *Neural Computation 12,* 11, 2519–2535. (Cited on page 24.)

[25] NESSLER, B., PFEIFFER, M., AND MAASS, W. 2010. STDP enables spiking neurons to detect hidden causes of their inputs. In *Proc. of NIPS 2009: Advances in Neural Information Processing Systems*. Vol. 22. MIT Press, 1357–1365. (Cited on page 26.)

[26] PEARL, J. 1988. *Probabilistic reasoning in intelligent systems: networks of plausible inference*. Morgan Kaufmann. (Cited on page 25.)

[27] PECEVSKI, D., NATSCHLAEGER, T., AND SCHUCH, K. 2009. PCSIM: A parallel simulation environment for neural circuits fully integrated with Python. *Frontiers in Neuroinformatics 3*. doi:10.3389/neuro.11.011.2009. (Cited on pages 37 and 38.)

[28] POUGET, A., DAYAN, P., AND ZEMEL, R. 2000. Information processing with population codes. *Nature Reviews Neuroscience 1,* 2, 125–132. (Cited on page 24.)

[29] SEUNG, H. AND SOMPOLINSKY, H. 1993. Simple models for reading neuronal population codes. *Proceedings of the National Academy of Sciences of the United States of America 90,* 22, 10749. (Cited on page 71.)

[30] VOGELS, R. 1990. Population coding of stimulus orientation by striate cortical cells. *Biological cybernetics 64,* 1, 25–31. (Cited on pages 24 and 99.)

[31] WISKOTT, L. AND SEJNOWSKI, T. 2002. Slow feature analysis: Unsupervised learning of invariances. *Neural computation 14,* 4, 715–770. (Cited on page 66.)