Karlsruhe Institute of Technology

Spiking Convolutional Deep Belief Networks for Unsupervised High Level Feature Extraction and Pattern Reconstruction

Master Thesis of

David Zimmerer

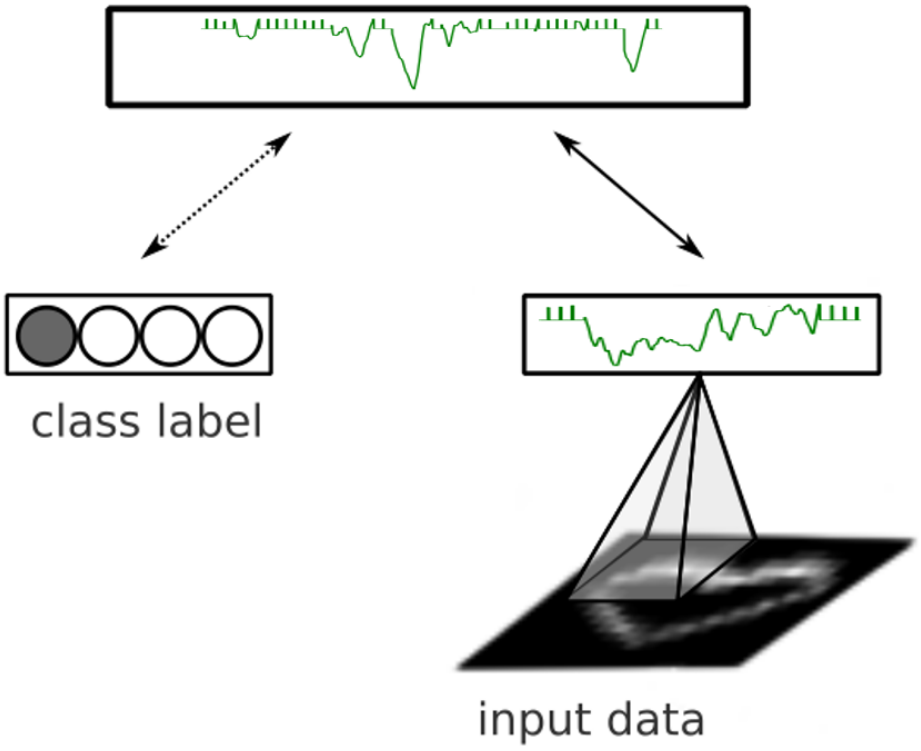
Department of Computer Science
Institute for Anthropomatics and Robotics
and
FZI Research Center for Information Technology

Reviewer:	Prof. Dr.–Ing. R. Dillmann
Second reviewer:	Prof. Dr.–Ing. J. M. Zöllner
Advisor:	M. Sc. Jacques Kaiser

Research Period: 06. July 2016 – 05. January 2017

Spiking Convolutional Deep Belief Networks for Unsupervised High Level Feature Extraction and Pattern Reconstruction

by
David Zimmerer



Master thesis
in January 2017

FZI

Master thesis, FZI
Department of Computer Science, 2017
Gutachter: Prof. Dr.–Ing. R. Dillmann, Prof. Dr.–Ing. J. M. Zöllner

Abteilung Technisch Kognitive Assistenzsysteme
FZI Research Center for Information Technology

Affirmation

Ich versichere wahrheitsgemäß, die Arbeit selbstständig angefertigt, alle benutzten Hilfsmittel vollständig und genau angegeben und alles kenntlich gemacht zu haben, was aus Arbeiten anderer unverändert oder mit Abänderungen entnommen wurde.

Karlsruhe, *David Zimmerer*
in January 2017

Abstract

Extracting high level features in data automatically can benefit various tasks such as classification and prediction. Deep belief networks allow to build feature extractors in an unsupervised manner. Utilizing a convolutional multilayered architecture can further improve the quality of the features on compositional data and allow a more abstract representation. Most architectures perform operations in discrete time steps and do not utilize a continuous information processing similar to the brain, which could lead to faster response times and a lower power consumption. In this thesis we propose different biologically inspired architectures to build and train a spiking deep belief network with a convolutional architecture. Two different training algorithms are presented. The first approach trains the network in discrete time steps and the resulting network is afterwards transformed into a spiking neural network. The second approach trains a spiking network directly with an adapted spike time dependent plasticity learning rule and weight synchronization. Both algorithms are evaluated on different discrete and event-based datasets. On the different datasets the algorithms are able to discriminate classes with high accuracy without any modification of the learning rules, thus indicating an adaptive feature extraction algorithm. By comparing both approaches it becomes apparent that by introducing lateral inhibitory connections the directly trained algorithm is able to extract more discriminative features.

Zusammenfassung

Die automatische Extraktion von "High-Level Features" unterstützt und erleichtert Aufgaben wie Klassifikation und Prädiktion. Mit Hilfe von Deep Belief Networks können solche Feature Extraktoren unüberwacht gelernt werden. Durch eine vielschichtige Convolutional Architektur kann die Feature Qualität auf kompositionellen Daten weiter verbessert werden und eine abstraktere Repräsentation ermöglicht werden. Die meisten Architekturen arbeiten, im Gegensatz zum Gehirn, in diskreten Zeitschritten und nutzen keine kontinuierliche Informationsverarbeitung, was zu schnelleren Verarbeitungszeiten und einer niedrigeren Leistungsaufnahme führen könnte. In dieser Arbeit werden verschiedene biologisch motivierte Architekturen vorgestellt, die es ermöglichen ein Convolutional Spiking Deep Belief Network aufzubauen und zu trainieren. Es werden zwei verschiedene Ansätze präsentiert. Der erste Ansatz trainiert ein Netzwerk in diskreten Zeitschritten und wandelt danach das trainierte Netzwerk in ein Spiking Netzwerk um. Der zweite Ansatz trainiert direkt ein Spiking Netzwerk mit einer angepassten Version der "Spike-Time dependent plasticity" Lern-Regel mit Parameter-Synchronization. Die Ansätze werden auf verschiedenen diskreten und Event-basierten Datensätzen evaluiert. Beide Ansätze erreichen auf den verschiedenen Datensätzen eine hohe Klassifikationsgenauigkeit ohne die zugrunde liegende Lern-Regel zu verändern, was auf einen adaptionsfähigen Feature-Extraktions Algorithmus schließen lässt. Der Vergleich der beiden Ansätze legt nahe, dass aufgrund von lateral-hemmenden Verbindungen der zweite Ansatz diskriminativere Features extrahieren kann.

Abbreviations

ANN	Artificial Neural Network
BM	Boltzmann Machine
CD	Contrastive Divergence
CNN	Convolutional Neural Network
COBA	Conductance-Based
CUBA	Current-Based
DBN	Deep Belief Network
DVS	Dynamic Vision Sensor
EBM	Energy-Based Model
eCD	Event-driven Contrastive Divergence
HCS	High Conductance State
LIF	Leaky Integrate-and-Fire
LTD	Long-Term Depression
LTP	Long-Term Potentiation
MCM	Markov Chain Monte Carlo
MLP	Multilayer Perceptron
MRF	Markov Random Field
NN	Neural Network
PGM	Probabilistic Graphical Model
pCD	Persistent Contrastive Divergence
PSP	Post Synaptic Potential
RBM	Restricted Boltzmann Machine
RV	Random Variable

SNN	Spiking Neural Network
STDP	Spike Time Dependent Plasticity
$\langle \cdot \rangle$	Expected Value of $\cdot$
$\sigma(\cdot)$	$sigmoid(\cdot)$

Contents

1. Introduction

1.1. Motivation

By winning the Imagenet Large Scale Visual Recognition Challenge in 2012, convolutional neural networks (CNNs) gained momentum [49]. Their powerful abstraction mechanism made them one of the most used algorithms in the fields of image and video classification and description and speech recognition [11, 46, 51, 81]. This can be primarily contributed to their ability to extract high level features on spatial and/or temporal conditioned data and utilize the compositional structure in information of the world.

Creating discriminative high level feature extractors allows the system to dynamically adapt to the input data and work on various kinds of data. In addition, the feature extractors do not need to have any semantic, human interpretable representation and can be more complex than manually built feature extractors. This also removes the labour-intensive and time consuming task of building feature extractors manually. Consequently, they recently got adapted to solve robotic problems like grasp planning, drone navigation and autonomous driving [21, 31, 57].

One precursor of CNNs are the deep belief networks (DBNs), which are built up of restricted Boltzmann machines (RBMs). DBNs have shown excellent performance on image classification tasks in the early 2000s [41, 55].

Compared to classical CNNs, DBNs allow recurrent connections and can be trained unsupervised. They have been described as "probably the most biologically plausible learning algorithm for deep architectures we currently know" [15]. DBNs can be used as a generative model as well, which means they can sample data according to a learned distribution, e.g., find the most probable completion for a partially erased image.

Adding convolution to DBNs increased the performance of DBNs on image classification tasks, caught up with the state of the art results and made the system more similar to the primate visual cortex than a standard DBN [55].

All these approaches use scalar values between neurons at discrete time slices to propagate information. While being well suited for GPUs, they are not very biologically plausible, since biological neuron interleave linear and non-linear operations, communicate by stochastic binary values and are not synchronized [15]. Spiking neural networks (SNNs) designed to simulate the communication between neurons with spikes work in continuous time by design and do not suffer from the aforementioned discrepancies of classical artificial neural networks.

1.2. Problem Statement

To the best of our knowledge, up to today, no system which utilizes the benefits of a convolutional architecture, spiking neural networks and deep belief networks altogether has been suggested. The main objective of the presented thesis is to realize such a spiking network which integrates convolution and can be easily trained utilizing an RBM learning algorithm in order to extract high level features. Two approaches are explored. The first approach trains convolutional RBMs on discretized input data to build up a DBN which is then converted to an SNN. The second approach learns from continuous, event-driven input-data and adapts the synaptic weights with a plasticity rule with shared weights to train spiking convolutional RBMs directly. Both approaches will learn to extract high level features which can be further used to classify an object.

1.3. Overview

This thesis describes the approach and implementation of spiking convolutional deep belief networks to extract high level features on continuous visual input. The thesis is structured as follows:

Chapter 2 introduces background information to present the state-of-the-art research used in Chapter 3. Chapter 4 describes the different approaches to build a spiking convolutional deep belief networks. In Chapter 5 the different implementation steps and architectures are described. Chapter 6 outlines and compares the performance of the networks and Chapter 7 will conclude the gathered insight of this thesis, state its limitations, and give suggestions for further improvements and research.

2. Background

In this chapter some foundations of spiking deep belief networks are presented, starting with probabilistic models, and explaining different neural network models afterwards.

2.1. Probabilistic Graphical Models

Probabilistic graphical models (PGMs) or structured probabilistic models can be used to describe, formalize, and model neural learning architectures [32, 69]. They can capture the basic probabilistic structure of data, which makes a significant number of data-related task feasible, such as [32]:

- Density estimation: Assigning a data sample x an estimate of its true probability density $p(x)$ (e.g., indicating unusual data).
- Denoising: Estimating the original data sample x given a perturbed or noisy input $\widetilde{x}$.
- Missing value imputation: Given a partial input sample of x, it returns the most probable completion of the missing values (e.g., inferring a label of data sample).
- Sampling: Generating data samples x drawn from the data distribution $p(x)$.

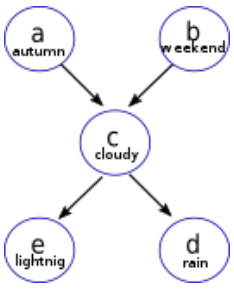
PGMs facilitate the process of modeling the probability distribution by using graphs to describe the probability distribution over the interactions of its random variables (RVs). A node in the graph represents an RV, and an edge a direct interaction between two RVs. This allows a less complex description of the probability density with basic factors over interacting RVs instead of modelling the probability density over all RVs.

How to build a PGM and set the parameters for a given problem is still an active research topic [30, 87]. There exist different approaches such as the SGS algorithm [87] or EM algorithms [30], but in the following we use Boltzmann machines and the contrastive divergence algorithm to build up and determine the parameters in a PGM (see Chapter 2.2.2).

PGMs can be divided into two basic categories, models with directed graphs and models with undirected graphs.

2.1.1. Bayesian Networks

Bayesian networks or belief networks are directed acyclic graphs, in which random variables are represented by nodes and their causal dependencies are represented by (directed) edges [26, 32]. It poses a way to depict a probability distribution factorized by repeatedly applying the Bayes rule.



(a)

p(c)	a	b
0.08	0	0
0.24	0	1
0.32	1	0
0.36	1	1

(b)

$$p(c) = \sigma(w_{ac}a + w_{bc}b)$$

(c)

Figure 2.1.: A sample Bayesian network with 5 binary nodes (a). In the network, the RV c is directly dependent on a, b and thus c is the child of its parents a, b. The RVs d and e are the children of c. Since there is a path from a to e, a is an ancestor of its descendent e. In (b) the conditional probability of $p(c|a,b)$ is given in a tabular form. Another variant of the conditional probability $p(c|a,b)$ is given in (c). The probability is defined by the parameters w_{ac}, w_{bc}, and due to the sigmoid activation function σ a network with such probability functions is called sigmoid belief network.

A connection from an RV x_i to an RV x_j represents a conditional probability distribution from x_j dependent on x_i, and x_i is called the parent of its child x_j:

$$p(x_j|x_i).$$

If a path from node x_n to x_k exists, then x_n is called an ancestor of x_k, and x_k a descendant of x_n.

In addition to the graph structure, which is often called the "qualitative" part of the models, the "quantitative" part has to be determined as well. The quantitative parameters are described by the local conditional probability distribution at each node given its parents. A sample network with the qualitative part as well as the quantitative part can be seen in Figure 2.1.

This is consistent with the Markov property, since each node is only depend on its direct parents:

$$p(x_i|x_{\setminus i}) = p(x_i|parents(x_i)),$$

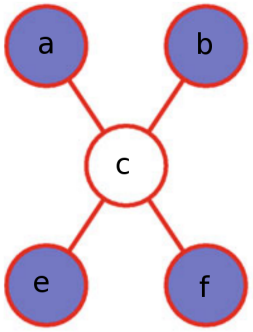
where $x_{\setminus i} = \mathbf{z} \setminus x_i$ and $\mathbf{z} = (x_1, ..., x_n)$ is the state of the model.

A Bayesian network contains a simple conditional independence assertion, i.e. each variable is, given its parents, independent of all non descendants:

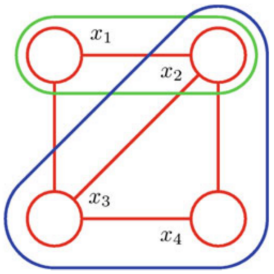
$$p(\mathbf{z}) = \prod_{x_i \in \mathbf{z}} p(x_i|x_{\setminus i}) = \prod_{x_i \in \mathbf{z}} p(x_i|parents(x_i)).$$

This reduction provides (i.a. from a computational perspective) an efficient way for learning i.e. parameter estimation and inference. Given some observed variables, the evidences, the network can be used to compute the posterior probabilities and infer the most likely state of the unobserved variables. This process of computing the posterior distributions given the evidences is called probabilistic interference.

Thus, a Bayesian network can be seen as a tool to apply and simplify the Bayes rule to complex problems.



(a) A Markov network with 5 nodes.



(b) Cliques in a Markov network

Figure 2.2.: (a) A Markov network with 5 nodes. The white node is dependent on all connected nodes (blue nodes). Given the blue nodes the white node is independent of any other node in the network. Given the white node all the blue nodes are independent of each other. (b) Two cliques in a Markov network. The blue clique is maximal, since no vertex can be added, which is fully connected to all others in the blue clique. The green one is not maximal, since the node x_3 could be added [16].

2.1.2. Markov Random Fields

In contrast to Bayesian networks, Markov random fields (MRFs) are undirected graphical models, in which random variables are also represented by nodes, but edges in this case indicate conditional dependencies [32, 62]. If two nodes x_i and x_j are connected by an edge, they are called each others neighbours. Given all of their neighbours, two unconnected nodes x_k and x_m are independent of each other. Thus, an MRF can be seen as a model of the joint probabilities of RVs (see Fig. 2.2a).

As a result the complexity and redundancies of the probability distribution can be reduced. Instead of defining one probability distribution over all RVs, the probability distribution can be factorized into fully connected partial graphs, so-called cliques:

$$p(\mathbf{z}) = \frac{1}{Z}\prod \phi(z_k),$$

where Z is the normalizing factor, so that $p(\mathbf{z})$ is a valid probability distribution with $\sum_{\mathbf{z}} p(\mathbf{z}) = 1$ ($\Rightarrow Z = \sum_z \prod \phi(z_k)$) and $\phi(z_k)$ is the partition function of the clique z_k.

The partition function $\phi(z_k)$ defines the potential of a clique, which indicates the likelihood of a state in the clique to be present. The higher the clique potential of a state the more likely is that clique-state. The clique potential for each state always has to be greater or equal to zero, but a general, computational convenient choice is to represent each state with a purely positive value.

Usually the maximal (sized) cliques are chosen since they contain all smaller sub-cliques and allow a finer factorization over those sub-cliques. But depending on the concrete problem, smaller clique sizes may be useful as well, e.g., Boltzmann machines usually are modelled with cliques of size two (see Fig. 2.2b).

2.1.3. Energy-Based Models

One convenient way to model the clique potentials is to use an energy function $E(z_k)$ [32]:

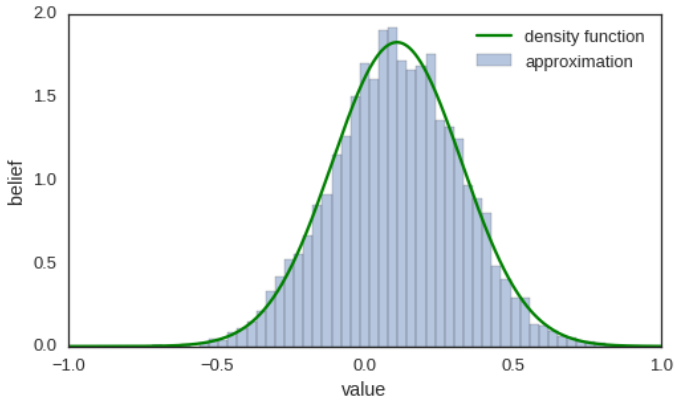


Figure 2.3.: Sampling a Gaussian distribution. The number of samples in an interval approximates the true density function. As more samples are drawn, the approximation of the density function will become more exact [7].

$$\phi(z_k) = exp(-E(z_k)).$$

Models with exponentials over energy functions are called energy-base models (EBMs).

This is useful since, due to $exp(z_n)exp(z_m) = exp(z_n + z_m)$, the energy of the whole graph can be decomposed into the sum of the energy of all cliques. Because the exponential function is always positive, the probability for each state is guaranteed to be greater than zero. This offers the freedom of choice of an arbitrary energy function $E(z_k)$, which can simplify optimization.

A probability distribution given by an EBM is also called Boltzmann distribution due to the similarity to statistical mechanics of gases in thermal equilibrium (and this is also where a Boltzmann machine get its name as presented in Chapter 2.2.2).

2.1.4. Sampling

Often, calculating the exact probability distribution or marginal distributions is a computational inefficient or untraceable problem [32, 69]. Sampling is a probabilistic solution to this problem, as it allows to approximate the target distribution nearly arbitrarily exact, and also turns calculating marginal probabilities into a by-product. An exemplary sampling of a Gaussian distribution is given in Figure 2.3.

Sampling can be described as the selection of a subset of individuals from a distribution to estimate properties of the complete distribution. To select a representative subset, the samples have to be unbiased of external influences such as the sampling procedure or problems in data collection. This can be particularly difficult since it may need infinite data, computational or temporal resources. In this case graphical models can be useful since they often facilitate the task of drawing samples from a distribution.

Ancestral Sampling For directed graphical models such a Bayesian networks there is a simple and efficient procedure called ancestral sampling, which can produce samples from the probability

distribution represented by the model [32]. The basic idea is to sort the variables x_n in the graph in a topological ordering, so that for all i and j, i is greater than j if x_i is a descendant of x_j. The variables can then be sampled in this order.

The topological sorting operation guarantees that the conditional distributions are valid and one can sample from them in order.

Markov Chain Monte Carlo If the probability distribution is represented by an undirected model, Markov Chain Monte Carlo (MCMC) methods can be used [32]. MCMC methods interpret the model as a Markov chain, and work best in an irreducible and aperiodic chain that is when no state in the undirected model has zero probability.

The basic idea is to begin in a state z with some arbitrary value. Then for some time z is repeatedly randomly updated using the, by the model given, transition distribution $T(z', z)$. Given enough update steps (possibly an infinite number), the network states will have reached a distribution which does not change any more (even though the individual states might do, the relative time spent in one state becomes constant). Such a distribution is referred to as stationary distribution or equilibrium distribution (due to its similarity to particle physics). Eventually, z becomes a fair sample of $p(z)$, which is equivalent to the stationary distribution of the Markov chain.

To get more than one sample, one can run more Markov chains in parallel, each initialized with a random starting state. Another method is to run only one Markov chain, run it for some time to allow the Markov chain to reach its equilibrium, and then take samples after different timesteps. Those approaches need the Markov chain to reach its equilibrium distribution, which is usually done, by letting it run for some burn-in time. But there are no guarantees that the Markov chain will settle in the given timespan. The second approach may lead to another problem since it can be hard to escape probable states, and when not run for an infinite timespan, more likely states can be over represented and less likely states under represented if they did not occur or over represented if they did occur.

Gibbs sampling Gibbs sampling is a commonly used MCMC algorithm [32]. The basic idea to perform the transition from one state to another, in accordance with $T(z', z)$, is to sample a single variable x_i only conditioned on its neighbours. Several variables can be sampled at the same time as long as they are conditionally independent given all of their neighbours.

2.2. Neural Networks

In this section we examine the foundations of natural and artificial neural networks (NNs). At first, we look at the mechanisms behind natural neural networks in the brain. Then different artificial models, starting with models working at discrete time steps and then models working in continuous time are described. To distinguish those, we refer to models working at discrete time steps as *artificial neural networks* (ANNs) and models working in continuous time as *spiking neural networks* (SNNs) throughout this thesis.

Figure 2.4.: A small section in the brain. The neurons *a* - *g* are connected to other neurons in a complex network [28].

2.2.1. Natural Neural Networks

Human Brain

The human brain is the main organ of the human central nervous system and with a weight of about 1.2 - 1.4 kg (2% of the total body weight) and a power consumption of around 20 watts (20% of the total human power consumption) it is assumed to be mainly responsible for what is commonly referred to as human intelligence [19, 28].

A human brain constantly performs tasks such as learning, memory, self-control, planning, reasoning, abstract thought, motor control, vision, and language. Different specialized regions in the brain which are involved in a task can be localized.

Even though different regions are involved in different tasks, the basic building blocks of the brain are astonishingly uniform. It is mainly composed of neurons, glial cells and blood vessels. While the neurons cells are the main computational unit, the other constituents are required for structural stabilization and energy support. A single neuron can only perform rather simple operations, but as the human brain contains 10^{12} neurons and each neuron is interconnected with 10^4 other neurons the operations can become more complex. The resulting complex neural network with roughly 10^{15} neural connections is the main component for human intelligence and enables complex tasks such as scene understanding, language processing and motion planing. A small section of a neural network recorded in a human brain is illustrated in Figure 2.4.

Neuron

Neurons are the main information processing unit in the brain [19, 28]. Information is primarily processed through chemical and electrical signals.

A neuron, which is a specialized kind of a (biological) cell, is separated from its surroundings by a cell membrane. Due to ion concentration differences between the interior and the exterior of the cell, there are potential differences at the cell membrane, which are referred to as the membrane potential. Ion channels, often voltage-gated, in the cell membrane allow positively and negatively

charged ions to flow from the inside to the outside and from the outside to the inside of the cell. This can lead to an increase (de-polarization) or decrease (hyper-polarization) of the membrane potential. The ion flow is primarily determined by the charge difference at the membrane (electro-chemical potential), the ion concentration differences (diffusion potential) and ion pumps actively pumping ions across the membrane. The membrane of a neuron at rest, with an equal influx and outflow of ions neutralizing each other, has usually an equilibrium potential of -65 mV.

Each neuron can be divided into three functional distinct parts: the dendrites, the soma and the synapses (see Fig. 2.5).

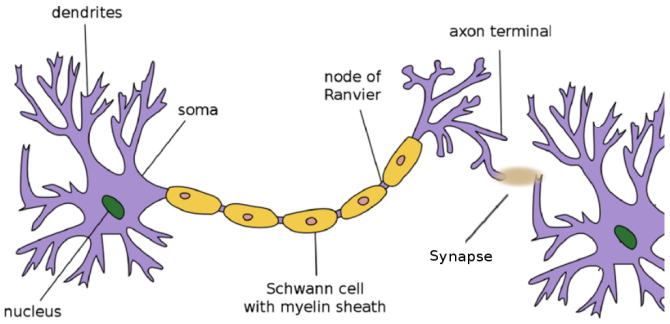


Figure 2.5.: A schematic view of a natural neuron. Other neurons are presynaptically connected via the dendrites. The spikes are then forwarded and accumulated in the soma and further propagated via the axon to the axon terminal and the out going synapses [6].

The dendrites are thin, complexly branching structures emerging from the cell body, also known as soma. They are the primary access for signals of preceding neurons through their synapses. These signals can polarize or depolarize the part of the dendrites and so inhibit or promote a spike.

The soma, which encompasses the nucleus, accumulates all polarizations from the dendrites and if the accumulation exceeds a "pseudo" threshold, usually around -55 mV, certain ion channels become more active, which allows an influx of positively charged ions and a spike emerges (there is no threshold in natural neurons, but rather the activation probability of some ion channels increases significantly as the membrane potential reaches some value. However, due to computational benefits some abstract neuron models explicitly choose to model a spike threshold as described in Chapter 2.2.3).

Starting from the soma the spike is propagated across the axon to the axon terminal where it is distributed to the dendrites of other subsequent (post-synaptic) neurons via the synapses, where they in turn can evoke (post-synaptic) potentials. The axon is often covered by a fatty substance, the myelin, to regulate and improve the conductivity.

Synapses can be divided into two different categories: Electrical, which communicate with other neurons with direct electrical connections and chemical, which use chemical compounds. Probably due to their more diverse form of signal exchange, almost all synapses found in the brain have a chemical nature [19].

Learning

The exact mechanisms behind learning in the brain are still mostly unknown [19, 28, 60]. Neural plasticity is often considered to be one of the methods behind learning. It describes the ability of the brain to change and reorganize the structure of brain, build new and alter connections.

Synaptic Plasticity specifies the changes to the synaptic strength, which is often associated with task learning and memory (in contrast to learning in brain, which is the general mechanism of altering the information processing in the brain, task learning describes the actual learning and remembering of a task).

Synaptic plasticity builds on the principle that temporally and locally correlated neural activity can lead to synaptic changes. It is often further divided into **short-term plasticity**, which acts on a time scale of milliseconds to minutes, and **long-term plasticity** lasting minutes or more.

"What fires together, wires together" A principle, which generalizes these principles, is the Hebbian principle or Hebbian rule. It is often commonly summarized as "Cells that fire together, wire together". Hebb originally stated it as follows: "When an axon of cell A is near enough to excite a 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."[35]. Thus, meaning that the more often A is active directly before B, the more likely A will have contributed to B's spike and the more causal B will become of A and A and B become associated. This can be mathematically expressed as

$$\Delta w_{ab} = \eta v_a v_b,$$

where Δw_{ab} describes the change of the synaptic weight between the pre-synaptic neuron A and the post-synaptic neuron B. v_a and v_b describe the activity of A and B respectively and η is some positive learning rate.

While this describes the classical Hebbian principle, different algorithms processing the neural activity of two connected neurons are often summarized and labelled as general Hebbian learning algorithms. Such algorithms can in general be formalized as

$$\Delta w_{ab} = F(v_a, v_b),$$

where F is a function describing the weight change conditioned on the neural activities [28]. This formulation allows a more complex processing of the neural activity and implementations e.g., the original formulation of the Hebbian principle as well as the anti-Hebbian rule.

2.2.2. Artificial Neural Networks

The first artificial neural network models performed computations at discrete time slices. While it simplifies the neuron model considerably, it makes them exceptionally easy to handle on most processors, which also work at a discrete tact rate.

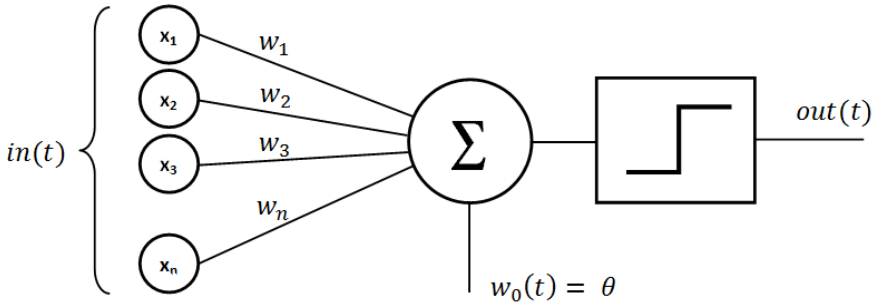


Figure 2.6.: Structure of a perceptron. The input $in(t)$ is set at the input variables x_i and multiplied with the corresponding synaptic weight w_i and accumulated. In addition, a threshold offset θ is added. Afterwards the step-function is applied i.e. the output $out(t)$ is 1 if the sum is positive and 0 if the sum is smaller 0 [5].

Perceptron

The perceptron, also called Rosenblatt Perceptron, was invented in the late 1950s by Frank Rosenblatt [71]. It was one of the first artificial neural networks, and can be seen as the foundation of most of the modern (deep) neural networks as well as linear discriminating classifiers.

Model The perceptron loosely models a neuron with a multi-dimensional input and a single output (compare Fig. 2.6).

Let $\mathbf{x} \in \mathbb{R}^n$ be the input of dimension n and $\mathbf{w} \in \mathbb{R}^n$ the n-dimensional vector describing the synaptic weights, then each x_i is multiplied by its synaptic weight w_i and then accumulated:

$$\sum_i x_i w_i = \mathbf{x}^\intercal \mathbf{w}.$$

If the sum exceeds a threshold $-b$, often referred to as the bias, the perceptron "fires" and the output y is set to 1 whereas otherwise it is set to 0:

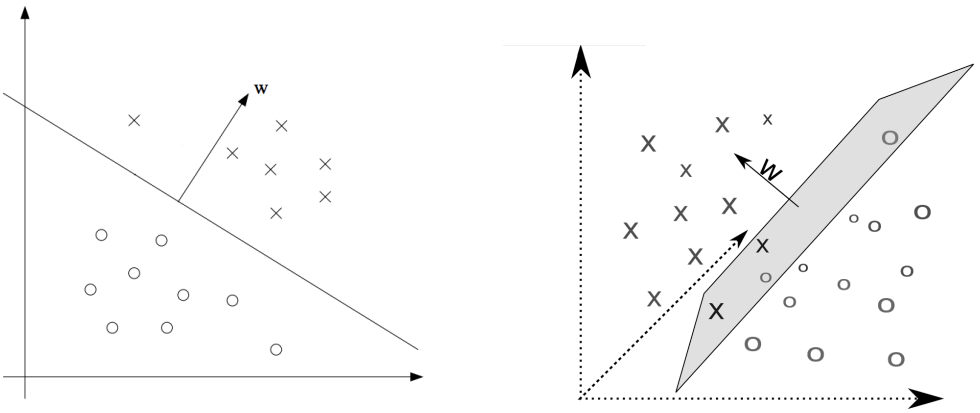
$$f = \begin{cases} 1, \text{ if } \mathbf{x}^\intercal \mathbf{w} + b > 0 \\ 0, \text{ if } \mathbf{x}^\intercal \mathbf{w} + b \leq 0. \end{cases}$$

One simplification is to add the bias b to the weight vector $\mathbf{w}' = (b, w_1, ..., w_n)$ and to extend the input dimension by a constant 1 i.e. $\mathbf{x}' = (1, x_1, ..., x_n)$. This allows us to handle the bias as a simple synaptic weight, and thus we cease to model it explicitly:

$$f = \begin{cases} 1, \text{ if } \mathbf{x}'^\intercal \mathbf{w}' > 0 \\ 0, \text{ if } \mathbf{x}'^\intercal \mathbf{w}' \leq 0. \end{cases}$$

Using the heaviside step function φ_{step}, the perceptron calculation rule can be further rewritten as

$$f = \varphi_{step}(\mathbf{x}'^\intercal \mathbf{w}').$$



(a) (b)

Figure 2.7.: The discrimination function of a perceptron. The discrimination function has the shape of a linear hyper plane in data space and is defined by the synaptic weight-vector $\mathbf{w}$. It divides the data space and thus the data samples into two subspaces, the positive space $\mathbf{x}^\intercal\mathbf{w} > 0$ and the negative space $\mathbf{o}^\intercal\mathbf{w} < 0$. In (a) the discrimination function is given in a two-dimensional data space and in (b) in a three-dimensional data space.

Decision Function This equation can be interpreted as a linear discrimination function, in which $\mathbf{w}$ defines a hyper plane in the data space, dividing it into two half spaces. Two exemplary linear discrimination functions in data space are given in Figure 2.7. This separation of the data space into distinct sub-spaces is often regarded to as classification. While a perceptron with a linear decision function only allows quite simple discrimination, more complex decision functions can be chosen e.g., multilayer perceptrons combine simple functions into more complex ones.

Perceptron Learning Be $\mathbf{X}$ a set of data-points and $\mathbf{Y}$ their corresponding labels and μ a learning rate. For a sample $x \in \mathbf{X}$ and $y \in \mathbf{Y}$ and $\tilde{y}$ the output of the perceptron, one update step can be described as

$$w = w + \Delta w,$$

with

$$\Delta w = \mu(\tilde{y} - y)x.$$

Thus the learning algorithm can be described as follows:

1. Initialize w randomly.
2. Select a data sample and calculate its output.
3. Calculate Δw and update the weights.
4. Perform steps 2.-3. until all data samples are correctly classified i.e. $\tilde{y} = y$ or a predetermined number of iterations has been completed.

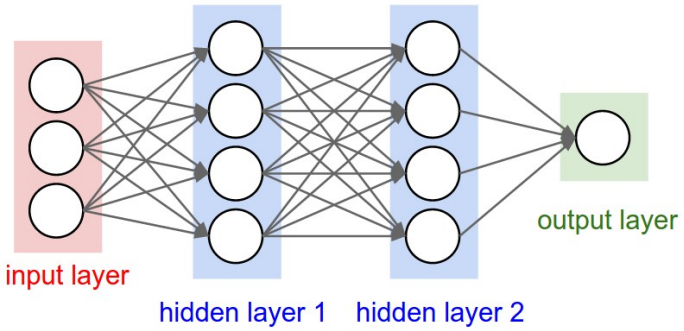


Figure 2.8.: A schematic multi layer perceptron with three layers [3].

Mutlilayer-Perceptron

While a perceptron models a single neuron, a multilayer perceptron (MLP), consisting of stacked perceptrons, can be seen as an extension of the model to neural networks and thus overcomes the perceptrons disadvantage to only discriminate linearly [32, 72].

Model architecture A MLP is built up of multiple consecutive layers (Fig. 2.8).

Each layer combines multiple perceptrons to map a multi-dimensional input $\mathbf{x} \in \mathbb{R}^n$ to a multi-dimensional output $\mathbf{y} \in \mathbb{R}^m$. The output $\mathbf{y}$ of the layer is composed of the m individual outputs y_i of the perceptrons in the layer on the same input.

A layer is defined by:

1. The input dimension n,
2. The output dimension m (which can be seen as the number of individual perceptrons in the layer),
3. The weight matrix $W \in \mathbb{R}^{n \times m}$ defining the weights between the synaptic connections,
4. The activation function $\varphi : \mathbb{R}^m \to \mathbb{R}^m$.

The output $\mathbf{y}$ of the layer is calculated as:

$$\mathbf{y} = \varphi(xW).$$

Each element $y_i \in \mathbf{y}$ can be interpreted as the output of a perceptron given the input $\mathbf{x}$ and the synaptic weights $w_i \in W = (w_1, \ldots, w_m)$.

By using the output of a previous layer l as input for a next layer $l+1$, several layers can be stacked up:

$$\mathbf{y}^{l+1} = \varphi(\mathbf{y}^l W^{l+1}),$$

where the superscript index represents the layer number.

Modelled like this the output is always forwarded to the next layer, i.e. there are no cycles in the information flow of the network. Such a network with only forward connections is called

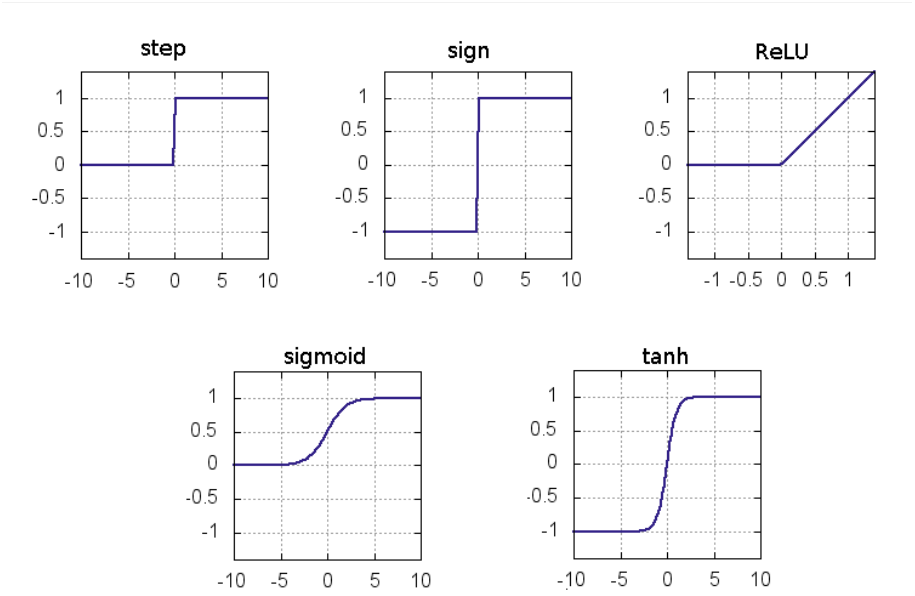


Figure 2.9.: The output of different activation functions plotted given the input.

feed-forward network (in contrast to recurrent networks such as Boltzmann machines, which are introduced in Chapter 2.2.2 and allow information to be forwarded in cycles).

Activation functions Basically, the activation function φ can be arbitrarily chosen. Different activation functions have been proposed, which also have proven to perform well on certain tasks:

- Step-function: $\varphi_{step}(x_i) = \begin{cases} 1, & \text{if } x_i > 0 \\ 0, & \text{if } x_i \leq 0 \end{cases}$,
- Sigmoid-function: $\sigma(x_i) = \frac{1}{1+e^{-x_i}}$,
- Softmax-function: $\varphi_{softmax}(x_i) = \frac{e^{x_i}}{\sum_k e^{x_k}}$,
- Sign-function: $\varphi_{sign}(x_i) = \begin{cases} 1, & \text{if } x_i > 0 \\ -1, & \text{if } x_i \leq 0 \end{cases}$,
- Tanh-function: $tanh(x_i) = \frac{e^{x_i}-e^{-x_i}}{e^{x_i}+e^{-x_i}}$,
- ReLU-function $\varphi_{ReLU}(x_i) = max(0, x_i)$.

A visualization of those functions is given in Figure 2.9.

Thus, an activation function describes the behaviour of a neuron on input data. This is very similar to the different neuron models for spiking neural networks (see Chapter 2.2.3).

Error functions In machine learning, in order to validate the quality of a model an error function or cost function is used. The error function provides a quantification of the performance of the model on a given task. Thus, the primary goal of a learning algorithm is to reduce the error on its

task. Hereby, it is important to note that the main error function is often primarily task dependent and rather independent of the learning algorithm used.

To compare the outputs $\tilde{\mathbf{y}}$ of the network with parameters θ to the correct data-labels $\mathbf{y}$ some of the most used error function or cost function $E(\mathbf{y}, \tilde{\mathbf{y}}|\theta)$ are:

- Mean squared error: $MSE = \frac{1}{N}\sum_{i=1}^{N}(\tilde{\mathbf{y}}_i - \mathbf{y}_i)^2$,
- Cross entropy: $CE = -\frac{1}{N}\sum_{i=1}^{N}[\mathbf{y}_i \log \tilde{\mathbf{y}}_i + (1-\mathbf{y}_i)\log(1-\tilde{\mathbf{y}}_i)]$.

Backpropagation In order to reduce the error E on a task, the parameters θ of the model can be modified. This modification of the parameters is often referred to as learning. Thus, the goal is to obtain parameters θ^* which form a (global) minimal point in the error function. Different optimization algorithms can be used to achieve this objective. However, the most common class of algorithms use the gradient of the error function with respect to the weights in order to determine the contributions of the weights to the error and thus, reduce the error and reach a minimal point. For gradient descent the gradient for the parameters are calculated and, by following the negative gradient direction, the weights are updated.

Since MLPs have a clearly defined structure, using the chain rule of calculus the gradient descent update rule can be simplified to an iterative procedure called backpropagation (or backprop for short) [32, 72].

We define the output y_j of neuron j as

$$y_j = \varphi(net_j) = \varphi(\sum_{k=1}^{n} w_{kj} y_k).$$

The partial derivative of the error function E with respect to a weight w_{ij} can be simplified by the repeated application of the chain rule:

$$\frac{\partial E}{\partial w_{ij}} = \frac{\partial E}{\partial y_j}\frac{\partial y_j}{\partial net_j}\frac{\partial net_j}{\partial w_{ij}},$$

where the single factors of the derivation can be resolved to:

$$\frac{\partial net_j}{\partial w_{ij}} = \frac{\partial}{\partial w_{ij}}(\sum_{k=1}^{n} w_{kj} y_k) = y_i$$

and

$$\frac{\partial y_j}{\partial net_j} = \frac{\partial \varphi(net_j)}{\partial net_j} = \varphi'(net_j).$$

Next, the first factor $\frac{\partial E}{\partial y_j}$ has to be determined. This can be further divided into two simple cases:

1. The neuron j is in the output layer:

$$\frac{\partial E}{\partial y_j} = \frac{\partial E(y_j)}{\partial y_j} = E'(y_j).$$

2. The neuron j is not in the output layer and L is the layer above neuron j:

$$\frac{\partial E}{\partial y_j} = \sum_{l \in L} \left(\frac{\partial E}{\partial net_l} \frac{\partial net_l}{\partial y_j} \right) = \sum_{l \in L} \left(\frac{\partial E}{\partial net_l} w_{jl} \right).$$

We define

$$\delta_j = \frac{\partial E}{\partial net_j} = \frac{\partial E}{\partial y_j} \frac{\partial y_j}{\partial net_j} = \begin{cases} E'(y_j)\varphi'(net_j), & \text{if } j \text{ is an output neuron,} \\ \left(\sum_{l \in L} \delta_l w_{jl}\right)\varphi'(net_j), & \text{if } j \text{ is not an output neuron.} \end{cases}$$

This all concludes to

$$\frac{\partial E}{\partial w_{ij}} = \begin{cases} E'(y_j)\varphi'(net_j)y_i, & \text{if } j \text{ is an output neuron,} \\ \left(\sum_{l \in L} \delta_l w_{jl}\right)\varphi'(net_j)y_i, & \text{if } j \text{ is not an output neuron.} \end{cases}$$

An update step for a weight w_{ij} can now be written as

$$w_{ij} = w_{ij} - \mu \Delta w_{ij} = w_{ij} - \mu \frac{\partial E}{\partial w_{ij}},$$

with a learning rate μ.

Convolutional Neural Networks

Convolutional neural networks (CNN) exploit spacial relations and the compositional structure of the input data to regularize the complexity of the neural network by putting further constraints on the architecture of those networks. This facilitates training and allows greater generalization on fewer training samples. It is implemented by having solely partially connected layers with shared connection weights instead of fully connected networks [32, 52].

Convolution As the name implies, CNNs perform a convolution operation [32]. The one dimensional convolution operation is defined as

$$c(t) = (a * b)(t) = \int_{-\infty}^{\infty} a(x)b(t-x)dx.$$

In this equation a is usually called the input and b the kernel of the convolution. The result c is often referred to as the feature map.

Since recorded data (e.g., images, speech) is often discretized, we extend the convolution to discrete data

$$c(t) = \sum_{x=-\infty}^{\infty} a(x)b(t-x).$$

Whereas convolution is originally defined over the one-dimensional temporal dimension, it can

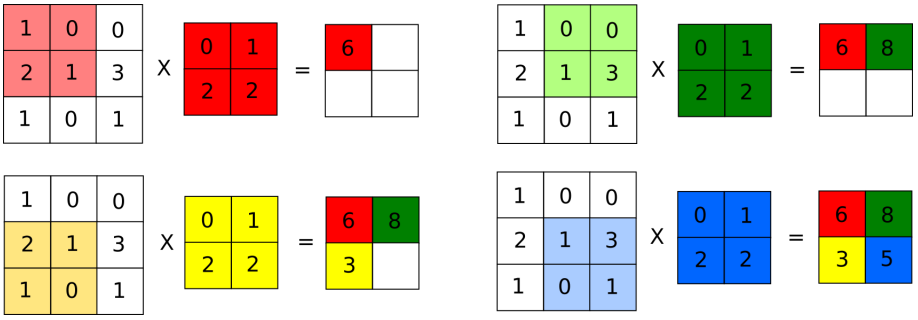


Figure 2.10.: A cross correlation of a 3×3 image matrix with a 2×2 kernel without stride and padding. The result is a 2×2 feature map.

also be applied to multiple arbitrary dimensions, e.g., two dimensional images I

$$C(i,j) = \sum_m \sum_n I(m,n)K(i-m, j-n).$$

In this case, the kernel matrix $K \in \mathbb{R}^{M \times N}$ as well as the feature map C also span multiple dimensions. A more intuitive way to rewrite this equation can be achieved by using the commutative nature of the convolution operation

$$C(i,j) = \sum_m \sum_n I(i-m, j-n)K(m,n).$$

Similar to the convolution is the cross correlation, which is basically a convolution without a flipped kernel

$$C(i,j) = \sum_m \sum_n I(i+m, j+n)K(m,n).$$

Due to this similarity, the terms convolution and cross correlation are often used ambiguously (see Fig. 2.10 for a sample cross correlation over a two dimensional input).

Convolution Layers In neural networks convolution is implemented in the so-called convolution layer, where the convolution operation with a learnable kernel matrix K is applied. An input given as a 3D tensor Y is composed of m 2D feature maps. Each feature map has the dimension $s \times t$. In the input layer, m is for example the number of color channels (three in case of an RGB image), and s is the width and t the height of the input image. A discrete convolution with a (M,P,Q) filter matrix K_j at position (x,y) is then defined as :

$$y_i^{jxy} = \sigma\Big(\sum_M \sum_{p=-\frac{P}{2}}^{\frac{P}{2}} \sum_{q=-\frac{Q}{2}}^{\frac{Q}{2}} w_{ij}^{mpq} y_{(i-1)}^{m(x+p)(y+q)}\Big).$$

where w_{ij}^{mpq} is the value at position (m,p,q) of the jth kernel matrix K_j in the ith layer and y_i^{jxy} is the entry in the jth 2D-feature map in the ith layer at position (x,y).
In a typical CNN the weights w_{ij}^{mpq} of all kernel matrices K_j in all layers i are the free parameters

that can be learned. Other so called hyper-parameters, which can be determined, are the number j of kernel matrices for each layer i, their stride size defining the shift of a filter at each step, and therefore the output dimension of the layer. Recent state-of-the-art systems have mostly reached the common consensus, of choosing a stride of 1, which we have adapted throughout this thesis [78].

Architecture The most common architectures for CNNs are built up from stacks of alternating convolutional and pooling layers. After those layers, fully connected layers are used as a classifier to assign labels to the extracted features [49, 78, 81]. A sample architecture is given in Figure 2.11.

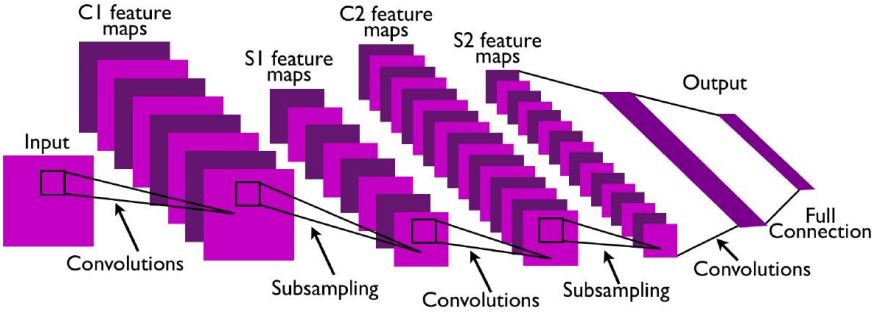


Figure 2.11.: Typical architecture of a convolutional neural network with two convolution-pooling stages [54].

Training The training is performed by using the backprop algorithm for CNNs:

$$\frac{\partial E}{\partial w_{ij}^{mpq}} = \sum_{m'}\sum_{q'}\sum_{p'} \frac{\partial E}{\partial y_i^{m'p'q'}} \frac{\partial y_i^{m'p'q'}}{\partial w_{ij}^{mpq}} = \sum_{m'}\sum_{q'}\sum_{p'} \delta_i^{m'p'q'} \frac{\partial y_i^{m'p'q'}}{\partial w_{ij}^{mpq}}.$$

By applying the chain rule this can be rewritten as

$$\frac{\partial E}{\partial w_{ij}^{mpq}} = \sum_{m'}\sum_{q'}\sum_{p'} \delta_i^{m'p'q'} \varphi(y_{i-1}^{m'-m,p'-p,q'-q}) = \delta_i^{mpq} * \varphi(y_{i-1}^{-m,-p,-q}).$$

Another more intuitive update rule is given by defining the contribution to the error of a single weight as

$$\frac{\partial E}{\partial y_i^{m'p'q'}} \frac{\partial y_i^{m'p'q'}}{\partial w_{ij}^{mpq}} = \Delta w_{ij}^{m'p'q'}.$$

Thus $\frac{\partial E}{\partial y_i^{m'p'q'}}$ can now be defined as

$$\frac{\partial E}{\partial w_{ij}^{mpq}} = \sum_{m'}\sum_{q'}\sum_{p'} \Delta w_{ij}^{m'p'q'},$$

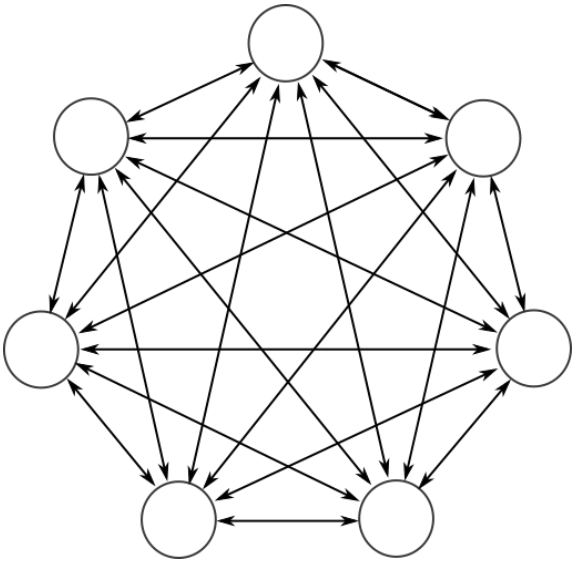


Figure 2.12.: A blueprint of a Hopfield nets with 7 binary units. The units are connected with symmetric undirected connections.

which results in applying the sum of each individual weight in a group of "tied" weights to all tied weights of the group.

Hopfield Nets

While CNNs have been wildly successful on image and speech recognition tasks, due to the pure forward nature of their connections, they still lack some biological plausibility since there are more feedback than feed-forward connections in the visual cortex [77]. Hopfield nets try to overcome some of those issues by using recurrent connections and binary units [32, 43] (see Fig. 2.12 for a sample network with 7 units).

Model Hopfield nets use binary units, meaning each unit can be either "firing" (having value 1) or "not firing" (having value -1).

A Hopfield net is fully connected with recurrent connections with symmetric weights but has no self connections:

$$w_{ii} = 0,$$
$$w_{ij} = w_{ji}.$$

The activation of an unit is similar to perceptron with a sign activation function and only depends on its adjacent units. It is calculated using the following rule:

$$s_i = \begin{cases} 1, & \text{if } \sum w_{ij} s_j - b_i > 0, \\ -1, & \text{if } \sum w_{ij} s_j - b_i \leq 0, \end{cases}$$

where s_i is the current state of unit i.

In contrast to feed-forward networks in Hopfield nets, there is no clearly defined bottom layer or order of the layers and thus, the updates can be performed in two different ways :

- Asynchronous: One unit is updated at a time. The units are either chosen randomly or in a predefined order
- Synchronous: All units are updated at the same time (based on the previous states of all units)

Similar to energy based models, each state $z = (s_1, ..., s_n)$ can be described by an energy. For Hopfield nets the energy of a state is defined as

$$E(z) = -\frac{1}{2}\sum w_{ij}s_i s_j + \sum b_i s_i.$$

By introducing recurrent connections, the network can be trained to store information. Indeed, with each (asynchronous) update step, the energy is thus guaranteed to stay the same or lower in value. Consequently, the network will converge to a local minimum of the energy function similar to a stable equilibrium state, where it can not escape from. Such a state can be called mode or attractor state. Hopfield nets can be trained as associative memory, where each stored pattern corresponds to such an attractor state.

The training rule for associative memory is a local Hebbian rule, i.e. for an update it only uses information of neurons on either side of a connection:

$$w_{ij} = \frac{1}{n}\sum_{patterns} s_i s_j.$$

Such Hopfield nets can consequently perform pattern completion by reaching a low energy state. However, they can also end up in a spurious state, an attractor state i.e. a local energy minimum, which was not presented as a training data.

Boltzmann Machines

Boltzmann machines try to improve Hopfield nets by replacing the deterministic update rule with a stochastic one. This allows a more stochastic exploration of the network states due to probabilistic escaping of minima states [12, 32].

Model Similar to a Hopfield net, the units s_i in a Boltzmann machine are binary as well and can have the value $s_i = 1$ or $s_i = 0$. A further similarity are bidirectional, symmetric weights without any self connections, and an equivalent energy function:

$$E(z) = -\frac{1}{2}\sum w_{ij}s_i s_j + \sum b_i s_i.$$

This allows the interpretation of a Boltzmann machine as an energy-based model with the probability distribution defined by the energy function E and its parameters $\mathbf{w}$.

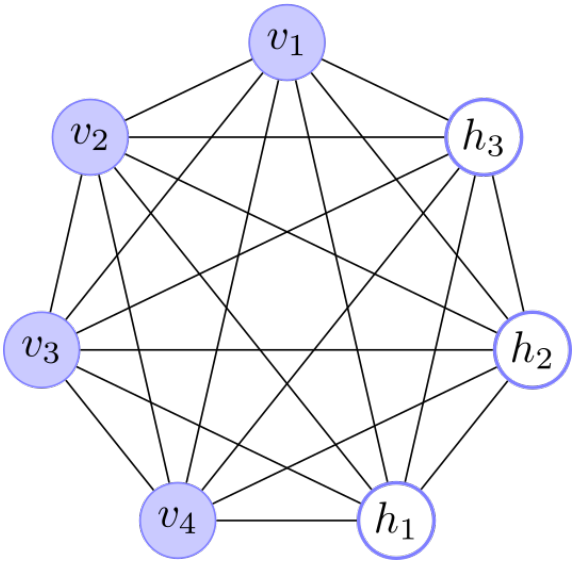


Figure 2.13.: A Boltzmann machine with 7 units. In contrast to a Hopfield nets, units are divided into visible and hidden units with stochastic activations [1].

In other words, a Boltzmann machine is an energy based model with the probability distribution defined by the energy. Thus, each state $\mathbf{z}$ can be assigned an energy which directly indicates the probability of the state $P(\mathbf{z}) \propto -E(\mathbf{z})$:

$$P(\mathbf{z}) = -\frac{1}{Z}\exp^{-E(\mathbf{z})}.$$

A unit is updated probabilistically given the states of its neighbour units:

$$p_{on}(s_i) = \sigma(\sum s_j w_{ij} + b_i),$$

where σ is the sigmoid-function.

An update step can be seen as a Gibbs sampling step from the distribution defined by E. Running a certain number of consecutive update steps probabilistically drives the network to a low energy or equilibrium state.

In contrast to Hopfield nets where each unit of the network is represented by an element a data sample, in Boltzmann machines latent variables are introduced to increase the capacity of the network. The units are thus divided into observable visible units and unobservable hidden units (compare Fig. 2.13). With hidden units a Boltzmann machine can universally approximate arbitrary probability mass functions over discrete variables.

Learning Rule The goal for training a Boltzmann machine is to get a probability distribution similar to the distribution, which generated the training data, the so-called data distribution. Thus, the objective for a Boltzmann machine is to assign a high probability to its training data (while assigning data not drawn from the data distribution a low probability) [12, 14, 38, 85]. By applying gradient descent, the weights of the Boltzmann machine are modified towards a configuration,

which assigns training samples $\mathbf{x}$ a high probability:

$$w_{ij} = w_{ij} - \mu \Delta w_{ij},$$

where

$$\Delta w_{ij} = \frac{\partial P(\mathbf{x})}{\partial w_{ij}}.$$

By using the logarithm of the probability function, $\frac{\partial P(\mathbf{x})}{\partial w_{ij}}$ can be rewritten as

$$\frac{\partial \log P(\mathbf{x})}{\partial w_{ij}} = \frac{\partial Z}{\partial w_{ij}} - \frac{1}{K}\sum_{i=1}^{K} \frac{\partial \log E(\mathbf{x}_i)}{\partial w_{ij}} = \frac{\partial Z}{\partial w_{ij}} - \left\langle \frac{\partial \log E(\mathbf{x})}{\partial w_{ij}} \right\rangle_{\text{data}}.$$

The derivation of the partition function Z by w_{ij} can be restated as

$$\frac{\partial Z}{\partial w_{ij}} = \int p(\mathbf{x}) \frac{\partial \log E(\mathbf{x})}{\partial w_{ij}} dx = \left\langle \frac{\partial \log E(\mathbf{x})}{\partial w_{ij}} \right\rangle_{\text{model}}.$$

Thus the update rule is now given as

$$\frac{\partial \log P(\mathbf{x})}{\partial w_{ij}} = \left\langle \frac{\partial \log E(\mathbf{x})}{\partial w_{ij}} \right\rangle_{\text{model}} - \left\langle \frac{\partial \log E(\mathbf{x})}{\partial w_{ij}} \right\rangle_{\text{data}}.$$

The derivation of $\log E$ after a weight w_{ij}

$$\frac{\partial \log E(\mathbf{x})}{\partial w_{ij}} = s_i s_j$$

gives the quite simple update rule called contrastive divergence (CD)

$$w_{ij} = w_{ij} + \mu(\langle s_i s_j \rangle_{\text{data}} - \langle s_i s_j \rangle_{\text{model}}),$$

where $\langle s_i s_j \rangle_{\text{data}}$ are the expected activations from the data distribution and $\langle s_i s_j \rangle_{\text{model}}$ are the expected activations from the model distribution. The most common way to get the data and model distribution is to perform consecutive Gibbs update steps with either training data or no data clamped to visible units until an equilibrium distribution is reached.

Here $\langle s_i s_j \rangle_{\text{data}}$ is referred to as the positive phase and $\langle s_i s_j \rangle_{\text{model}}$ as negative phase. A simple interpretation of the update rule is a shift of the weights away from the model distribution towards the data distribution.

RBMs

To perform such an update, samples from the model distribution are needed. To get those samples, the Boltzmann machine has to reach an equilibrium distribution. This can be computationally expensive and there is no guarantee to tell when the equilibrium distribution is reached. Thus, potentially infinite sampling steps have to be performed.

To evade the problem, a simple solution is to make the hidden units independent of each other

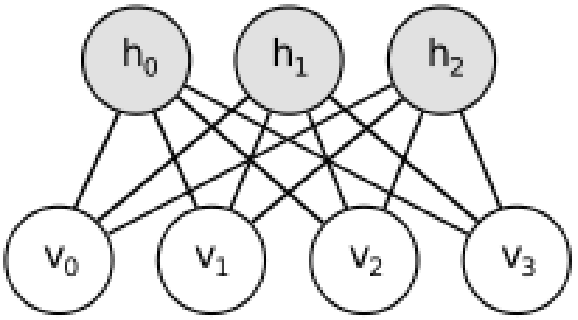


Figure 2.14.: A restricted Boltzmann machine is special kind of Boltzmann machine with no lateral connections in the hidden and visible layer. This facilitates sampling, since the visible are solely dependent on the hidden units and the hidden units solely on the visible units [9].

(and the visible units independent of each other), so that all hidden units can be sampled independently and in parallel to each other. Thus, a Boltzmann machine with two bipartite layers is called restricted Boltzmann machine (RBM) (see Fig. 2.14) [80].

Sampling the hidden units **h** given the visible units **v** is called a *upward pass* and sampling the visible units **v** given the hidden units **h** is called a *downward pass*.

In this case the probability of data sample **x** can be expressed using the free energy $F(\mathbf{v})$ over its visible activations **v** [27, 37]:

$$p(\mathbf{x}) = p(\mathbf{v}) = \sum_{\mathbf{h}} \exp^{-E(\mathbf{v},\mathbf{h})} = \exp^{-F(\mathbf{v})},$$

where the free energy can also be expressed as

$$F(\mathbf{v}) = -\sum x_i b_i - \sum \log(1 + \exp^{x_i}).$$

CD-k Training The simplified structure of an RBM enables faster weight updates. Hinton showed empirically that this training procedure can be further improved by running only a limited number of Gibbs sampling steps to approximate the model distribution [38].

The gradient calculated with this approximation can be seen as "good enough" to perform updates, which drive the underlying distribution towards the data distribution.

Thus, given a data sample **x** an CD-k update is computed as follows:

1. Perform one upward pass given $\mathbf{v}_0 = \mathbf{x}$ to get $\mathbf{h}_0$.
2. Perform k alternating downward and upward passes to get $\mathbf{v}_k$ and $\mathbf{h}_k$.
3. Update $w_{ij} = w_{ij} + ((s_i s_j)_0 - (s_i s_j)_k)$.

In Figure 2.15 the sampling steps are visualized by unrolling the RBM as a directed model.

Persistent CD (pCD), proposed by Tieleman, does not initialize the Boltzmann machine each time a new sample is drawn, but uses the activations of previous runs instead [83]. An intuition

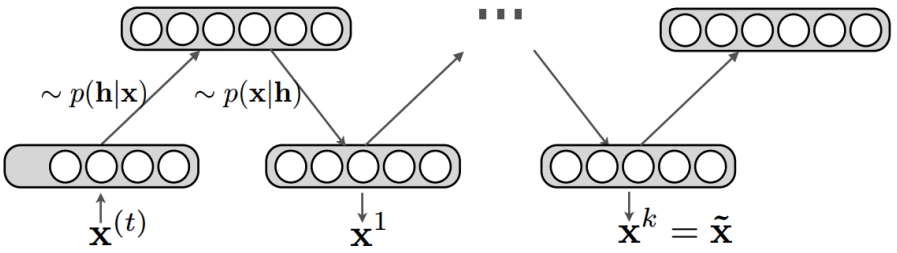


Figure 2.15.: A temporal unrolling of the contrastive divergence algorithm with k sampling steps. The hidden units and the visible units are alternately sampled conditioned on the current state of the other [8].

why it sometimes shows better performance could be that the previous activations are already closer to an energetic minimum, so that fewer steps are required to get a good estimate of the model distribution.

Deep Belief Networks

A deep belief network (DBN) is a generative graphical model, or alternatively a type of deep neural network, composed of multiple layers of latent variables (hidden units) with connections between consecutive layers, but with no connections within layers [32, 39, 41].

A deep belief network can be built up by stacking up RBMs. This does not only enable unsupervised training of a deep belief net, but also improves the conditional distribution of the bottom RBM (each time an RBM is added, it improves the prior over the hidden units with a better and more complex prior).

In a deep belief net the two top layers form an RBM with symmetric weights while the weight symmetry can be broken up between the lower layers to allow asymmetric weights.

Training Training of an DBN can be split up into greedily training RBMs in a cascade of stacked up RBMs (as illustrated in Fig. 2.16). The training procedure can be formalized as:

1. Train a new top RBM given the transformed data from the bottom layers.
2. After training one RBM, transform the training data by sampling the hidden layer activations of the top RBM.

After training, the DBN still has symmetric weights.

Fine-tuning After greedily training the DBN, the bidirectional connections can be split up and fine tuned to a certain task. In general there are two most used fine-tuning algorithms, the wake-sleep algorithm for data generation and the backprop algorithm for classification.

In the wake-sleep algorithm the symmetric weights are split up into bottom-up and top-down weights and are fine-tuned individually [40]:

1. Compute a stochastic forward pass, and for each layer adjust the top-down weights to better reconstruct the activations in the layer below.

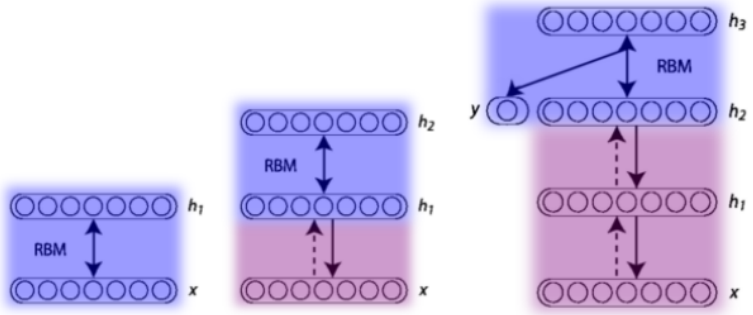


Figure 2.16.: Building up a deep belief network by training RBMs greedily and stacking them up on top of each other. At first, only one RBM is trained. On top of the first RBM the next RBM is trained. This procedure can be performed for an arbitrary number of repetitions. In the top layer "association" RBM the label information y can be in-cooperated as well [4].

2. Perform sampling steps in the top level RBM and adjust the weights with CD.

3. Compute a stochastic backward pass, and for each layer adjust the bottom-up weights to better reconstruct the activations in the layer above.

Another way to fine-tune the weights for classification, given labels and an error function, is to perform back propagation to further fine tune the bottom-up weights (while the top-down weights are usually discarded). One interpretation of this is to see a DBN as a pre-trained ANN.

2.2.3. Spiking Neural Networks

While all the previous presented models run at discrete time steps, the next models are designed to run contentiously, which makes them more similar to natural neurons and neural networks [59].

Neuron Models

Similar to the activation function in ANNs, in SNNs there are also different models describing how neurons process input. Many neuron models describe the development of a neuron's membrane potential over time. Therefore, the models commonly have a resting potential, which describes the membrane potential a rest without any external influences. Most models also have a way to model external input from incoming spikes and an internal leakage pulling the membrane potential back to its resting potential. If a "pseudo" threshold is exceeded usually a spike is emitted and the neuron is in a refractory phase for some time.

LIF The leaky integrate and fire (LIF) neuron is a neuron model, which phenomenological describes the membrane potential at the soma [10, 28, 69]. It is one of the simplest and consequently computationally most efficient, most important and popular spiking neuron models.

By discarding the different shapes of the action potentials and reducing it the uniform spike events, the information is condensed to the precise spike times. The model, described by linear equations, models the membrane potential integration due to spike input currents with a capacitor

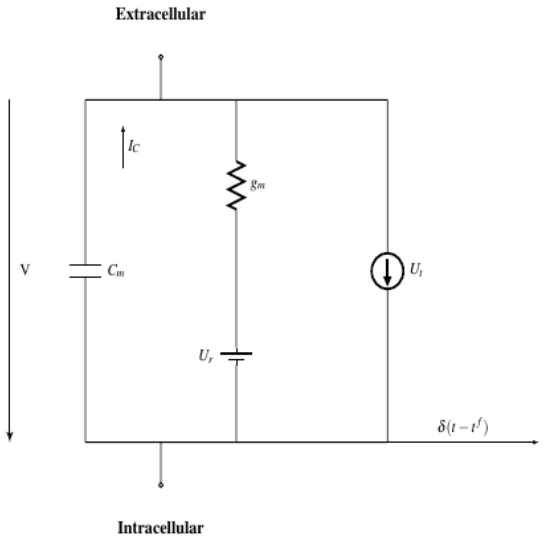


Figure 2.17.: A LIF neuron as an electrical circuit. The capacitor C_m corresponds to the potential at the membrane, g_l the leakage potential and E_l the resting potential. If the membrane potential is greater than U_t a spike δ is emitted [25].

and introduces a leaky current with a resistor. The model can be represented by a circuit of a single capacitor and a resistor with a battery (see Fig. 2.17):

$$C_m \frac{\partial u}{\partial t} = g_l(E_l - u(t)) + I^{syn} + I^{ext},$$

where C_m is the membrane capacitance, g_l is the leak conductance, E_l the resting potential and the input current $I = I^{syn} + I^{ext}$ is divided into a static external input I^{ext} and a synaptic input I^{syn}.

If the membrane potentiality exceeds a threshold θ a spike s is emitted and the membrane potential is instantaneously pulled back to its reset potential u_{reset} and kept at this potential for its refractory period t_{ref}.

$$u(t_s < t \leq t_s + t_{ref}) = u_{reset}.$$

The emitted spikes are modelled as a spike train ρ using only the precise spike times:

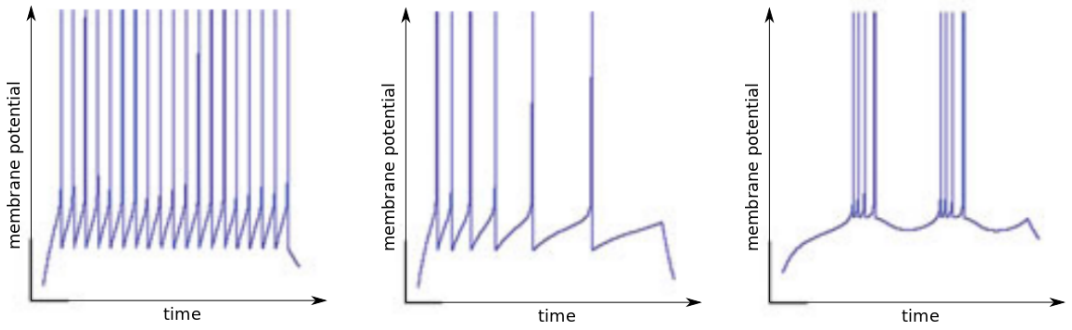
$$\rho(t) = \sum_{\text{spikes } s} \delta(t - t_s),$$

where δ is the Dirac delta function.

Due to its simplifications the LIF model can not capture some natural observed behaviour such as bursting or adaptation (see Fig. 2.18).

Hodgkin-Huxley The Hodgkin-Huxley model tries to improve some of the limitations of the LIF model by explicitly modelling different ion channels [28, 42].

The first model, as described by Hodgkin and Huxley, introduced three different ion channels, namely sodium, potassium and a leak current of CL^- ions, which they discovered in their experi-



(a) Constant tonic firing (b) Frequency adaptation (c) Delayed regular bursting

Figure 2.18.: Different neural firing behaviour observed in the Brain. In (a) a neuron shows constant tonic firing, while in (b) the neuron shows frequency adaptation and in (c) the neurons shows delayed regular bursting [28].

ments on the axon of a squid. Each channel is modelled as a resistor with a battery (see Fig. 2.19). The Hodgkin-Huxley model with three ion channels can be described by the following equations :

$$C_m \frac{\partial u}{\partial t} = g_{Na} m^3 h (E_{Na} - u(t)) + g_K n^4 (E_K - u(t)) + g_l (E_l - u(t)) + I,$$

where the variables h, m, n are described by :

$$\begin{aligned} \frac{\partial h}{\partial t} &= \alpha_h u(t)(1-h) - \beta_h u(t) * h, \\ \frac{\partial m}{\partial t} &= \alpha_m u(t)(1-m) - \beta_m u(t) * m, \\ \frac{\partial n}{\partial t} &= \alpha_n u(t)(1-n) - \beta_n u(t) * n, \end{aligned}$$

with the rate constants α, β for each ion channel.

This model can be extended and generalized to model more than three ion channels with their dynamics in order to better match the characteristics and biophysics of different neurons in the brain:

$$C_m \frac{\partial u}{\partial t} = \sum_{k \in K} g_k m^{p_k} h^{q_k} (E_k - u(t)) + I,$$

with an arbitrary number of ion channels K.

While this allows the model to predict and simulate various effects observed in the brain like frequency adaptation with high accuracy, the Hodgkin-Huxley model is more computationally expensive.

Activity in a network One way to introduce activity into a spiking neural network is to use a Poisson spike generator [36]. A Poisson spike generator produces stochastic firing according to a Poisson point process. The firing rate λ or rate function $\lambda(t)$ determines the dynamics of the homogeneous or inhomogeneous Poisson process respectively and thus of the spike times.

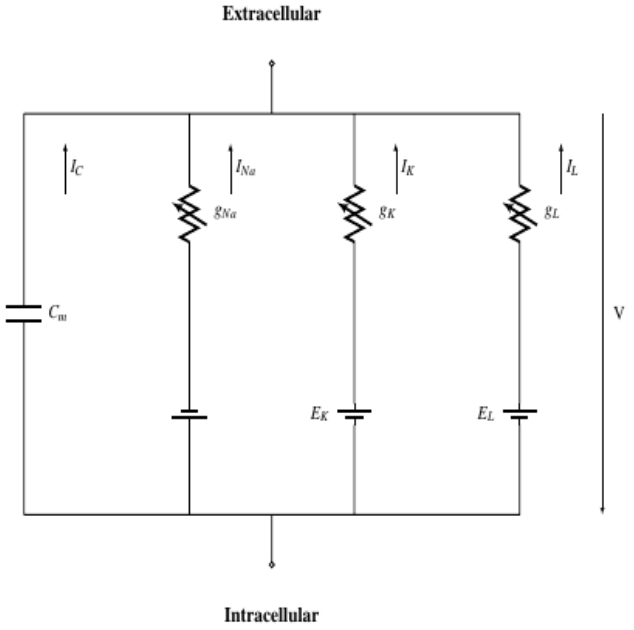


Figure 2.19.: A Hodgkin-Huxley neuron as an electrical circuit. The membrane potential corresponds to C_m, and the ion channels to g_{Na}, g_k, g_L with their reversal potentials E_{Na}, E, E_L [25].

The probability of an emitted spike in a time interval δt is given by:

$$P_{spike}(t,t+\delta t) = \lambda(t)\delta t,$$

where the occurrence of a spike is independent of previous spikes.

Since the spikes can be formalized as a Poisson process, the expected number of spikes for a time interval δt is given by :

$$\langle P_{spike}(t,t+\delta t)\rangle = \int_t^{t+\delta t} \lambda(t)dt.$$

Synapses

While synapses in ANNs are simply multiplying an incoming input with their weights, for SNNs there are different models which add additional dynamics in order to model naturally observed behaviour closer.

The behaviour of the synapses are described by modelling the synaptic input f^{syn}. A basic model of the influence of synapses can be described as follows [69]:

$$f^{syn}(t) = \sum_{\text{synapses } k} \sum_{\text{spikes } s} w_k \varepsilon(t-t_s),$$

where ε is a function describing the shape of the post synaptic potential (PSP) and w_k a synaptic weight. Different shapes are discussed in the paragraph *Kernel functions* (2.2.3).

Current-based synaptic interaction One synapse type models the input current of neuron I^{syn} directly as the synaptic input f^{syn}. This is a logical implementation for LIF neurons since they directly model the potential at the soma and do not consider most of the membrane dynamics in the dendrites. Thus, the synaptic input current is given as a linear summation of the post synaptic potentials with temporal effects:

$$I^{syn}(t) = \sum_{\text{synapses } k} \sum_{\text{spikes } s} w_k \varepsilon(t - t_s),$$

where ε is a kernel describing the explicit shape of a post synaptic spike.

Conductance-based synaptic interaction Conductance-based synapse models describe the synaptic dynamics more closely to its natural counterpart. They consider the conductance changes of incoming spikes, which push the conductance locally towards the reversal potential of the specific ion type.

In this case the synaptic input f^{syn}can be seen as a change in the synaptic membrane conductance g^{syn} :

$$g_x^{syn} = \sum_{\text{synapses } k} \sum_{\text{spikes } s} w_k \varepsilon(t - t_s) \ , x \in \{e, i\}.$$

Thus, the synaptic input current I^{syn} can now be described using the membrane conductance by applying Ohm's law:

$$I^{syn}(t) = g_e^{syn}(E_e^{rev} - u(t)) + g_i^{syn}(E_i^{rev} - u(t)),$$

where E_e^{rev} and E_i^{rev} is the excitatory and inhibitory reversal potential of the membrane, which can be e.g., chosen as E_{Na} and E_K respectively.

High conductance state One interesting property of conductance based neurons is that they can reach a so-called high conductance state (HCS) [69]. Such a state is defined by a high total membrane conductance

$$g^{tot} = g_l + \sum_{\text{synapses } k} g_k^{syn},$$

with

$$\sum_{\text{synapses } k} g_k^{syn} =: g^{syn} >> g_l.$$

Such a state can be reached by a lot of incoming synaptic firing. e.g., high frequency Poisson noise from other neurons.

The HCS is especially interesting because the dynamics of the membrane potential can be described by a Gaussian process called Ornstein–Uhlenbeck process where the mean is primarily determined by the effective synaptic input (with the noise) and the variance by the total membrane conductance (see Fig. 2.20). In addition, Petrovici showed that in such a state the time the neuron needs to get from its resting potential to a value given by the Ornstein–Uhlenbeck process can be basically neglected (see Fig. 2.21). This will show further application as a stochastically firing neuron model used for neural sampling in Chapter 3.2.

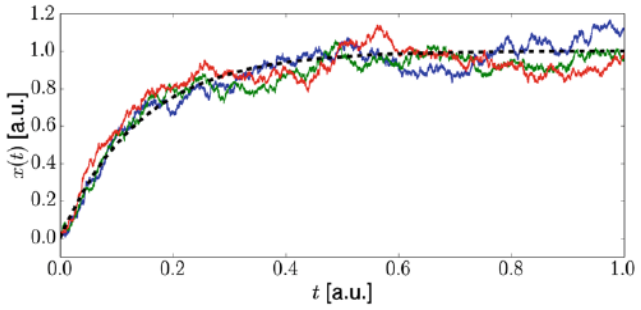


Figure 2.20.: Three samples of Ornstein–Uhlenbeck processes. This can be seen as the membrane potential of neurons in a high conductance state (in comparison to the black doted line can be seen as a neuron without noisy input) [69].

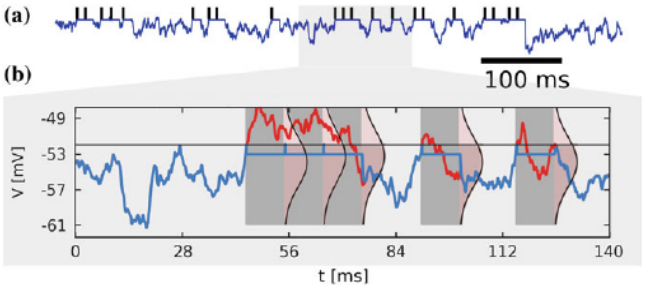


Figure 2.21.: A membrane potential trajectory of a neuron in a high conductance state. (a) The spike train and the membrane potential. (b) The blue curve represents the actual membrane potential with refractory periods after each spike, while the red curve represents a membrane potential without a firing threshold. It is apparent that the blue curve returns to the hypothetical red potential without a "return from rest" time [69].

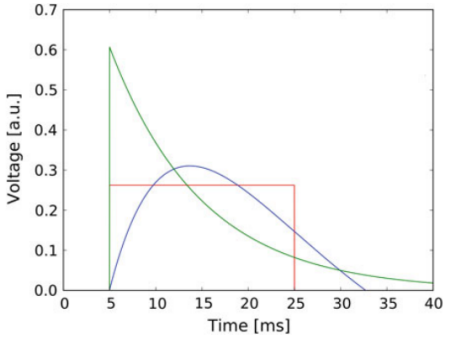


Figure 2.22.: Three different PSP kernels. The green one has an alpha-shape, the blue one is exponential shaped and the red one is rectangular [69].

Kernel functions Besides the synaptic models, the kernel function ε probably has the most important part in modelling the synaptic input current I^{syn}. The kernel describes the explicit shape of the post synaptic potential over time and thus the influence it has on the membrane potential. Several different kernel functions have been proposed, whereof some of the most important are:

- Rectangular-shaped:
$$\varepsilon(t) \propto \begin{cases} 1, & \text{if } t_s < t \leq t_s + t_{ref} \\ 0, & \text{otherwise .} \end{cases}$$
- Exponential-shaped: $\varepsilon(t) \propto t \exp(-\frac{t}{\tau_{syn}})$.
- Alpha-shaped: $\varepsilon(t) \propto \exp(-\frac{t}{\tau_{syn}})$.

These are also visualized in Figure 2.22.

Neural Coding

Spikes are the primary way of information transmission in the brain. In neural networks this information can be encoded in different ways. We present three encoding mechanisms, which were observed in the brain, rate coding, temporal coding and population coding [61].

In the *rate coding* scheme the information is purely contained in the firing rate of a neuron. In contrast to rate coding, in *temporal coding* the information is additionally transmitted via the different points in time at which the spikes are transmitted. Thus, the information is carried in the exact spike timings. *Population coding* encodes information as the joint activity of several neurons in a population.

In this thesis we use temporal or rate encoded input data, which is represented in the network as a population coded state over the joint activity of several neurons and is decoded in a rate-based manner.

Learning

Learning in the brain describes the generalized term how information in the brain is stored (in contrast to task learning, memory adaptation is also considered learning).

Most models of learning algorithms in spiking neural networks build on changes in the synaptic strength between neurons to store information [28]. To consider these changes as learning, they have to span over minutes to days or more, which is described by long-term plasticity, e.g., by LTP (long-term potentiation), LTD (long-term depression) or STDP (spike time dependent plasticity) (see Fig. 2.23).

The most commonly used models are inspired by the research and discoveries of Hebb and the previously discussed Hebbian principle (Chapter 2.2.1).

Spike time dependent plasticity Spike time dependent plasticity (STDP), inspired by research on single neurons with artificially induced current, describes such a Hebbian learning algorithm.

Experiments indicated an increase of the synaptic weight if a post-synaptic spike occurred in strong temporal vicinity after a pre-synaptic spike and a decrease of the synaptic weight if a post-synaptic spike occurred in strong temporal vicinity before a pre-synaptic spike. The further apart the spike times of the different neurons were, the weaker was the effect on the spike (see Fig. 2.23).

Given the spike times of the pre- and post-synaptic neurons t^{pre} and t^{post}, this leads to the following update rules [28, 79]

$$\Delta w_{ij} = \sum_{f=1}^{N} \sum_{n=1}^{N} W(t_n^{post} - t_f^{pre}),$$

with a common choice for the STDP function W

$$W(x) = \begin{cases} A_+ \exp(\frac{-|x|}{\tau_+}) & \text{for } x > 0, \\ -A_- \exp(\frac{-|x|}{\tau_-}) & \text{for } x < 0, \end{cases}$$

given the time constants τ_+ and τ_- and the weight depend parameters A_+ and A_-.

Long-term potentiation Long-term potentiation (LTP) can be interpreted as an STDP variant with only the positive part of the STDP function

$$W(x) = A_+ \exp(\frac{-|x|}{\tau_+}).$$

This leads to a purely positive increment of weights (see Fig. 2.23).

Long-term depression Long-term depression (LTD) can be seen as the negative counterpart to LTP, with an STDP function

$$W(x) = -A_- \exp(\frac{-|x|}{\tau_-}).$$

This leads to a purely negative decrement of weights (see Fig. 2.23).

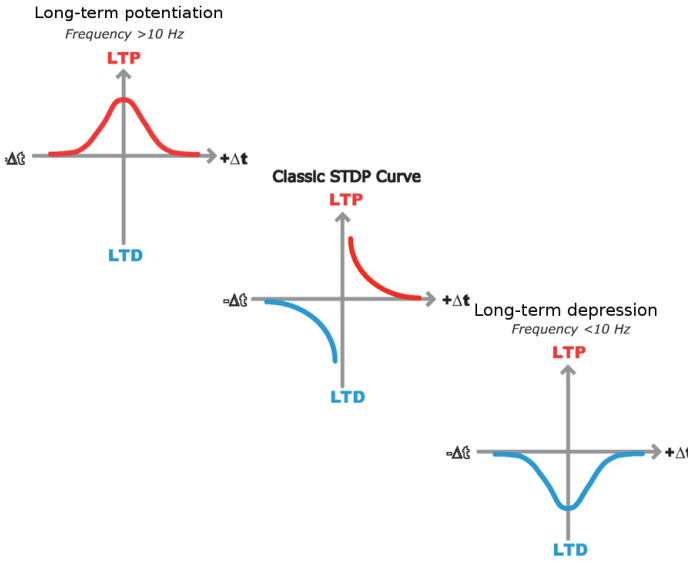


Figure 2.23.: Different STDP curves observed in the Brain. The first curve shows only long term potentiation. The middle one shows a variant of the classic STDP Curve and the last one shows purely long term depression [17].

3. Related Work

In this section we present recent work related to convolutional RBMs, which have already been proposed in discrete time. We further show recent approaches for sampling and implementing Boltzmann machines in spiking neural networks.

3.1. Convolutional RBM

The convolutional RBM (cRBM) was invented more or less at the same time by Desjardins et al., Lee et al. and Norouzi et al. in 2008/2009 [23, 55, 66]. In similarity to CNNs it can be seen as the adaptation of an energy-based model to compositional data. Describing images in terms of spatially local features needs fewer parameters, generalizes better and offers re-usability as identical local features can be extracted from different locations of an image. Modelled after CNNs and unlike a normal RBM, the visible and hidden layer in the cRBM are connected in a convolutional manner (as described in Chapter 2.2.2) instead of being fully connected.

Propagating information up can be seen as the convolution with a filter matrix W (see Fig. 3.1a):

$$P(\mathbf{h}|\mathbf{v}) = \sigma((W * \mathbf{v}) + \mathbf{b}_h).$$

The down propagation utilizes the flipped kernel $\tilde{W}$ (see Fig. 3.1b & 3.1c):

$$P(\mathbf{v}|\mathbf{h}) = \sigma((\tilde{W} * \mathbf{h}) + b_v).$$

Using the convolution operation the energy of the network can thus be rewritten as

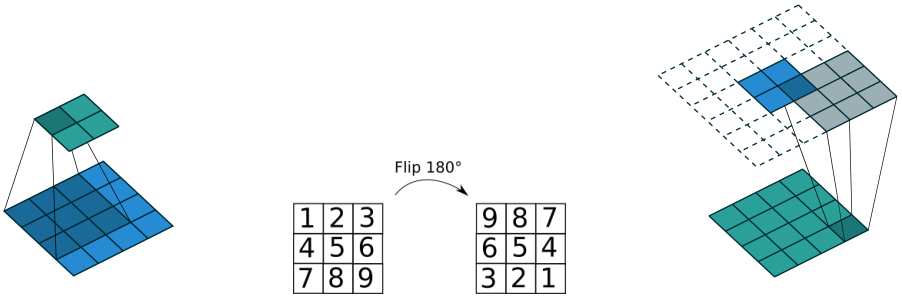
$$E(\mathbf{h}, \mathbf{v}) = \mathbf{h}^\intercal (W * \mathbf{v}) + \mathbf{h}^\intercal \mathbf{b}_h + \sum b v_i.$$

Similar to a normal RBM, a convolutional RBM is trained with the objective to maximize the probability of the training data. This can be achieved by using the CD algorithm with a few adaptations due to the tied weights (similar to backprop for CNNs in Chapter 2.2.2) [67]:

$$\frac{\partial E}{\partial w} = \sum_{w \in W_{\text{group}}} \Delta w,$$

where W_{group} is a group of weights tied to the same value.

In contrast to CNNs, due to their local learning rule, RBMs can not be explicitly trained to perform max pooling operations. Thus Lee et al. proposed a softmax based probabilistic max pooling to introduce local sparseness in the hidden layer activations, on top of which a pooling



(a) The upward pass. (b) The 180°flipped kernel. (c) The downward pass.

Figure 3.1.: A Gibbs sampling step in a convolutional RBM. At first the visible data is convolved with the kernel to get the hidden activations (a). Afterwards the kernel matrix is flipped 180°(b). In the downward pass the hidden activations with padding are convolved with the flipped kernel to get the new visible activations (c) [2].

layer can be stacked [55]:

$$P(h_{ij}^k|\mathbf{v}) = \frac{\exp(I(h_{ij}^k))}{1+\sum_{(i',j')\in B}\exp(I(h_{i'j'}^k))},$$

where $I(h_{ij}^k)$ is the activation of the hidden unit before applying the sigmoid function ($I(h_{ij}^k) = (W * \mathbf{v}) + \mathbf{b}_h$) and B is a partition block containing the unit h_{ij}^k. Such an architecture is visualized in Figure 3.2. This drives the activity in a block to be sparse and likely only one or none unit in a block will be active.

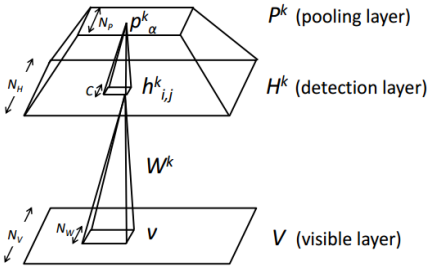


Figure 3.2.: The RBM layer architecture with probabilistic max pooling [55].

3.2. Sampling in SNNs

Indicated by stochastic neural transitions found in the brain in experimental studies, a new way of information encoding in the brain as representations of probability distributions and probabilistic interference has been suggested [34, 56, 86].

A first framework, which proposed how spiking neurons can perform MCMC sampling, was introduced by Buesing [18]. A simplification to discretize time into time slices can be introduced without interfering with the basic concept (see Buesing for a generalization to continuous time [18]).

A neural network can be considered as a network of RVs, where the state of a neuron is defined

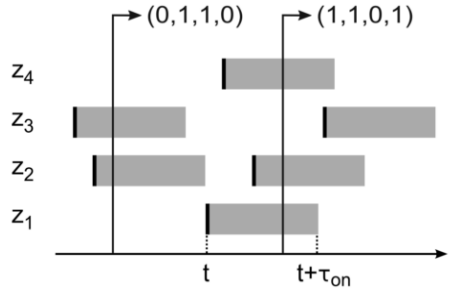


Figure 3.3.: A spiking neural network as probabilistic model. The state z_i of a neuron is set to 1 for a time period t_{on} after a spike. The complete state of the network $\mathbf{z}$ is given by the individual states (z_1, z_2, z_3, z_4) [69].

by its firing. Since the probability of two neurons spiking at the same time is basically zero, after a neuron has fired it is set to a firing state for a time period τ

$$z_k(t) = \begin{cases} 1, & \text{if k has fired in the time intervall } (t-\tau, t], \\ 0, & \text{if k has not fired in the time intervall } (t-\tau, t], \end{cases}$$

where z_k is the state of neuron k and set to 1 for the "firing" state and 0 for the "not firing" state (see Fig. 3.3). A common choice for τ is the refractory period of the neuron τ_{ref}. Consequently, one way to characterize the firing probability of a neuron is to take the ratio of the time a neuron has spent in state $z_i = 1$ compared to the total timespan T : $\frac{n_{\text{spikes}_i}\,\tau}{T}$.

Thus, for a given time step t the state of the network $\mathbf{z}(t) = (z_1(t), ..., z_n(t))$ is defined by the state of the individual neurons $z_i(t)$.

To get a process with Markovian properties $p(\mathbf{z}(t)|\mathbf{z}(0), ..., \mathbf{z}(t-1)) = p(\mathbf{z}(t)|\mathbf{z}(t-1))$, an auxiliary counter variable $\xi \in \mathbb{N}$ is introduced which discretizes the time a neuron has left to stay in the "firing" state into time slices

$$z_k(t) = 1 \iff \xi_k(t) \geq 1$$

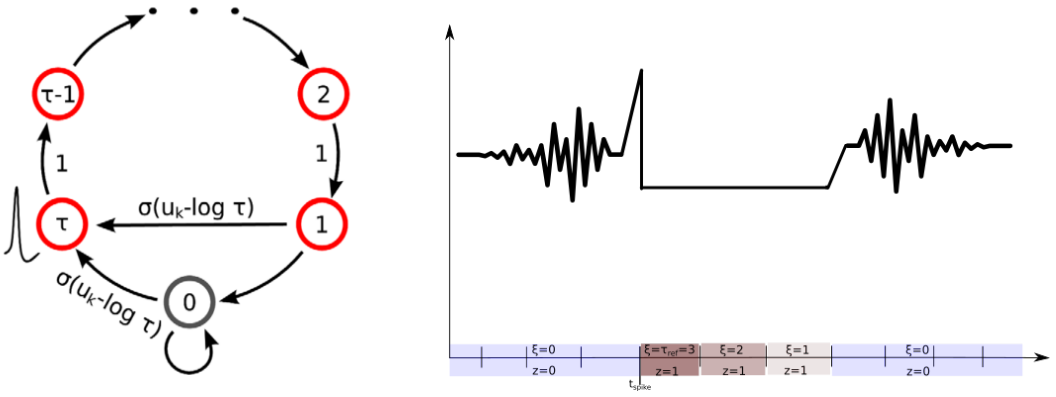
Thus, ξ can be seen as a counter, counting down from τ to 0 in each time step after a neuron has fired (see Fig. 3.4 for schematic overview and a sample implementation)

$$p(\xi_t|\xi_{t-1}) = \begin{cases} 1, & \text{for } \xi_{t-1} > 1 \text{ and } \xi_t = \xi_{t-1} - 1, \\ p(\text{"}firing\text{"}), & \text{for } \xi_{t-1} \leq 1 \text{ and } \xi_t = \tau, \\ p(\text{"}not\,firing\text{"}), & \text{for } \xi_{t-1} \leq 1 \text{ and } \xi_t = 0, \\ 0, & \text{otherwise.} \end{cases}$$

Buesing proposes an abstract stochastic neuron model which activates with a probability proportional to the input

$$P(\text{"}i\ fires\ at\ time\ t\text{"}) \approx \sigma(\sum_j w_{ij} z_j(t) + b_i).$$

With this model Buesing proved that spikes and the corresponding state updates in such networks

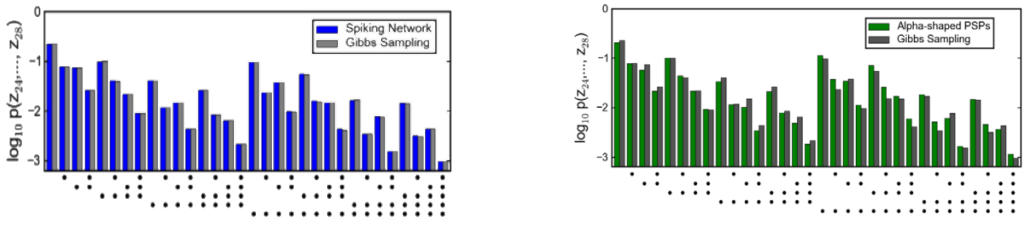


(a) (b)

Figure 3.4.: The artificial counter state ξ of a neuron in discrete time. In (a) the red state represent an active state $z_i = 1$. After a spike ξ is set to the refractory period τ. In each time step the refractory period counter ξ is reduced by 1 until the neuron is inactive and can spike again. In (b) an exemplary membrane potential with the corresponding states of a neuron is given [18].

can be seen as MCMC sampling. Experiments show, as $t \to \infty$, the network is in fact able to approximate a Boltzmann distribution. Replacing the rectangular PSP with a more biological plausible alpha shaped PSP deteriorates the performance a little, due to overshooting at the beginning and accumulation effects, but is still reasonable well (see Fig. 3.5).

Petrovici improved the model by replacing the abstract stochastic neuron model by conductance bases LIF neurons, a more common and biologically inspired neuron model [69]. He proved under high frequency (Poisson) noise, which leads to a high conductance state of the membrane potential, the neuron shows stochastic firing, determined by the input current and the noise frequency (as described in Chapter 2.2.3). This allows the neuron to show a firing behaviour which can be matched by a sigmoid function (see Fig. 3.6). By scaling the weights the LIF neurons can be used to perform neural sampling similar to Buesing.



(a) (b)

Figure 3.5.: Comparison of the state probabilities of neural sampling with Gibbs sampling in a five state Boltzmann machine. In (a) a probabilistic neuron model with a rectangular PSP is used and no discrepancies can be detected given a long enough sampling period. In (b) a more biologically plausible alpha shaped kernel is used, which leads to minor differences [18].

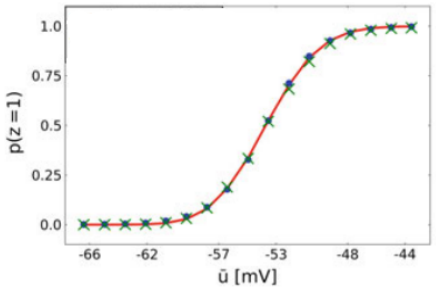


Figure 3.6.: Input output transfer function of a neuron in a high conductance state. The output rate is normalized by $\frac{1}{\tau_{ref}}$ to get an output probability. The transfer function is identical to shifted sigmoid function σ [69].

3.3. Artificial to Spiking Neural Network Conversion

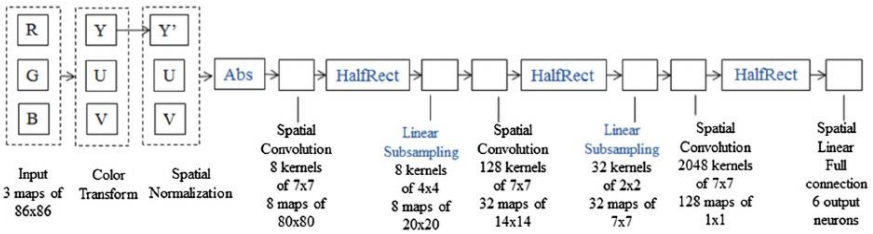
The results of Petrovici and Buesing allow a quite simple transformation of a Boltzmann machine to spiking neural networks of noisy conductance based LIF neurons with the synaptic weights scaled to match the impact on the membrane potential [69].

O'Connor uses a different approach, where instead of approximating sigmoid units by LIF neurons, he uses the Siegert neuron, a rate base approximation of LIF neurons with Poisson input, in order to implement units in the artificial RBM which activate similarly to LIF neurons [68]. Such a trained net can be directly transferred to an SNN. For a neuron its firing rate can be approximated by the Siegert transformation and can be normalized by the maximal firing rate to get an activation probability, which can be adapted by the units of the RBM. After the RBM with the Siegert activation function is trained, the weights can be simply adapted to an SNN with LIF neurons described by the Siegert approximation.

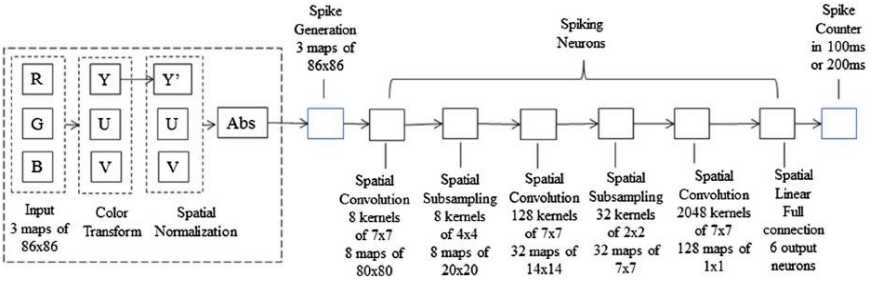
There also have been several approaches to transform a CNN to an SNN. Cao et al. and Diehl et al. propose certain constraints on the CNN architecture to show reasonable performance in the converted SNN [20, 24]:

- The CNN is only fed positive data since SNNs can't represent negative pre-synaptic spikes.
- The ReLU activation function is used in the CNN, which closely matches the input-output mapping of LIF neurons without a refractory period ($t_{ref} = 0$) and is always positive.
- The bias terms are eliminated.
- Instead of max pooling, average pooling is used, since it has a simple spiking counterpart (see [20]).

After the CNN is trained with the backpropagation algorithm, the weights are transferred to an SNN with an equivalent architecture using LIF neurons without a refractory period (see Fig. 3.7 for an adapted CNN and the equivalent SNN architecture). In addition, before applying the final



(a) A for conversion adapted CNN architecture.



(b) A to an SNN converted convolutional network.

Figure 3.7.: The proposed architectures to conversion between CNNs and SNNs [20].

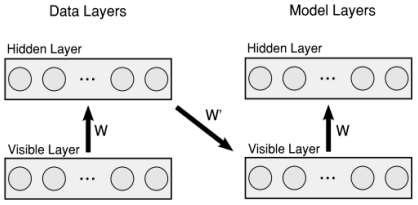


Figure 3.8.: An unrolled RBM with tied weights, which can be used to simulate the positive and negative phase and learn the weighs online [65].

weights, Diehl et al. use a model- and data-based weight normalization procedure to further fine-tune the synaptic weights [24].

3.4. Event-Driven CD

Different approaches to train a rate-based or spiking RBM have been proposed, of which the first one was probably by Teh et al.[82]. They use multiple identical binary stochastic input units for each element in a data sample to approximate a rate based input.

The approaches described next make use of the synaptic sampling described in the previous Chapter 3.2.

One approach is the evtCD by Neil et al., which works in continuous time with spiking networks and an STDP variant [24]. He simulates the positive and negative phase of the CD algorithm by unrolling the RBM with shared weights (as seen in Fig. 3.8 where the RBM is unrolled to simulate CD-1 updates). But this approach only allows a fixed number of CD steps and is, due to the weight synchronization, not very plausible.

A more sophisticated approach, called eCD, which also uses STDP was proposed by Neftci et al.[64]. They use bidirectional synapses between a visible and hidden layer of LIF neurons. They

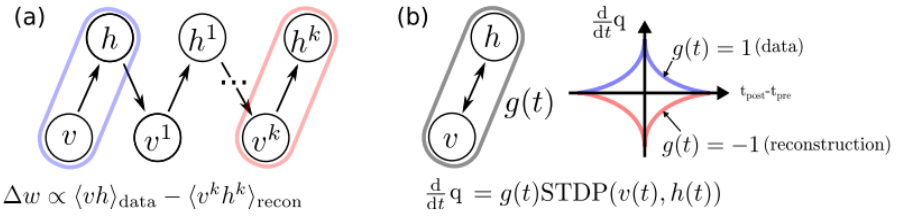


Figure 3.9.: Comparison between classical CD and event-based CD. In the classical k-CD (a) the weight update is determined by positive phase and negative phase, while in event-base CD (b) the weight update is determined by STDP and the sampling phase, determined by g [64].

implement an adapted symmetric STDP variant, which, at a given time, only allows LTP or LDP to model the positive or negative phase of the CD algorithm respectively (see Figure 3.9 for a comparison). The symmetric learning rule for two neurons, given their spike trains $v_i(t)$ and $h_j(t)$, can be expressed as:

$$\Delta w_{ij} = \mu g(t) STDP(v_i(t), h_j(t)),$$

with the learning rule μ, the global STDP status flag $g(t)$ determining the positive and negative phase of the CD algorithm and $STDP(v,h)$ parametrized to implement a symmetric weight change dependent on the neural activity:

$$\begin{aligned} STDP(v_i(t), h_j(t)) &= v_i(t) A_{h_j}(t) + h_j(t) A_{v_i}(t), \\ A_{h_j}(t) &= A \int_{-\infty}^{t} W(t-s) h_j(s) ds, \\ A_{v_i}(t) &= A \int_{-\infty}^{t} W(t-s) v_i(s) ds. \end{aligned}$$

In this STDP rule $A(t)$ can be seen as an activity trace indicating recent activity and $v_i(t)$, $h_j(t)$ as a control variable enabling weight changes given a spike at time t. The kernel function W should be symmetric, a common choice is $W(x) = exp(\frac{x}{\tau})$.

In their approach a training step can be divided into four phases (see Fig. 3.10):

1. The data signal is applied and the system is allowed to model the data distribution ($g(t) = 0$).
2. Positive STDP is used to get $v_i h_j$-data (with LTP) and is added to the weights (postive phase $g(t) = 1$).
3. The data signal is removed and the system is allowed to model the model distribution ($g(t) = 0$).
4. Negative STDP is used to get $v_i h_j$-model (with LTD) and is subtracted from the synaptic weights (negative phase $g(t) = -1$).

In similarity to RBMs, the weight change in the second phase can be summarized as $\mu\ \delta w_{pos}$, the weight change in the fourth phase as $\mu\ \delta w_{neg}$, which results in the CD update rule:

$$w = w + \mu\ \delta w_{pos} - \mu\ \delta w_{neg} = w + \mu(\delta w_{pos} - \delta w_{neg}).$$

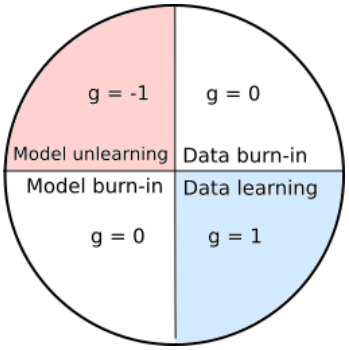


Figure 3.10.: Visualization of the phases of the event based contrastive divergence.

4. Approach

In this section after describing the general convolutional architecture in spiking neural networks in order to find the best performing spiking convolutional DBN, we discuss two different approaches to build these.

The first one is the more classical one, in which an in discrete time trained DBN is transferred to a spiking neural network. The second approach trains the RBMs event based with STDP directly as spiking neural networks.

In Chapter 6, we then evaluate and compare the performance of both approaches.

4.1. Convolutional Architecture in Spiking Neural Networks

We implement a convolutional layer with receptive fields and shared weights between two neuron populations. Each population has a similar three dimensional topology as in artificial CNNs with the number of neurons given by $n_{\text{neurons}} = \text{channels} \times \text{height} \times \text{width}$.

Groups of neurons in the bottom layer in close vicinity, a so-called receptive fields, are each synaptically connected to the same top layer neuron (see Fig. 4.1a). Different receptive fields have the same shape and size n and share the same synaptic weights (given the top layer neurons belong to the same feature map). Adjacent receptive fields can overlap by several neurons and the output neurons have the same topology as their input regions.

A receptive field covers a partial region of the input data and the synaptic weights from the bottom layer neurons to the top layer neuron can the be seen as a convolution over the partial input data. Since the weights of the receptive field can be shared, the neural activity in the top layer can be seen as a convolution over the input data with a filter of the size of a receptive field. In this case the top layer activations of receptive fields with shared weights can also be called a feature map (see Fig. 4.1b). By using more neurons with different sets of shared weights over the same receptive fields, more feature maps can be implemented.

(a) (b)

Figure 4.1.: Receptive fields over 4 neurons. In (a) the top layer neuron (red) is only connected to a some neurons in close vicinity to each other. In (b) four receptive fields are next to each other with a stride of two and shared weights.

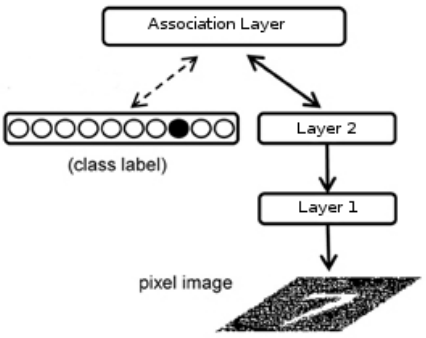


Figure 4.2.: A common structure of a deep belief network, built up of RBMs, used for classification. The first layers transform the input data/ pixel image and can extract features and the top layer RBM is used for association the data with the correct label [88].

4.2. Conversion

The conversion approach can be roughly divided into two steps:

1. Train artificial RBMs to build up a DBN.
2. Convert the DBN to a spiking neural network.

4.2.1. Convolutional DBNs

To train a convolutional DBN we proceed similar to Hinton et al. and Lee et al. [41, 55].

At first the convolutional RBMs are trained greedily with CD, as described in Chapter 3.1, for a certain number of iterations on data batches of batch-size b. After an RBM is trained, we convert the dataset X into a new dataset X' by sampling of the hidden layer of the RBM (one forward-pass step):

$$p(x') = \sigma(W * x).$$

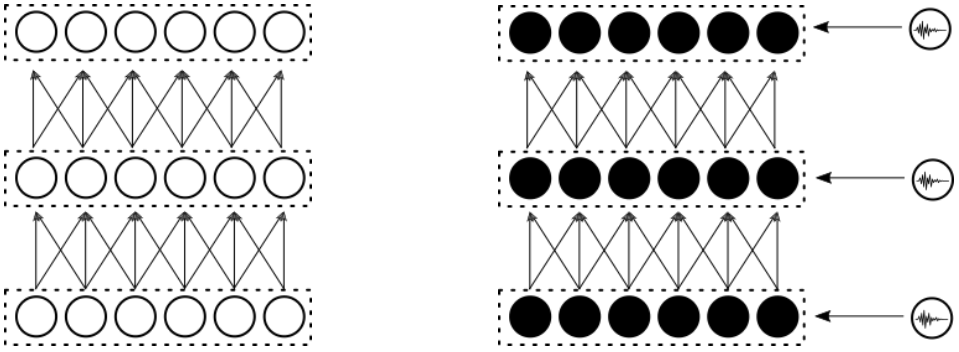
On this converted dataset the next convolutional RBM is trained greedily.

Evaluating feature qualities is still an active research topic. To get a measurement of our feature quality, we use labelled data and train a classifier on top of the extracted features. To stay in a biologically plausible domain, we train a fully connected RBM on the top level features combined with the label of the data samples in order to associate the features with the correct labels. This architecture is similar to the architecture proposed by Hinton [41] (see Fig. 4.2).

To evaluate the final performance, we input a data sample without a label into the DBN and after a forward-pass let the top layer sample a label prediction by performing Gibbs sampling steps, which can be seen as a reconstruction of partially presented data (as mentioned in Chapter 2.1).

4.2.2. Conversion

We present three different variations of the conversion.



(a) Structure of a spiking CNN with LIF neurons. (b) Structure of a spiking DBN with LIF neurons.

Figure 4.3.: Different converted structures of a CNN and a DBN. The CNN is converted to a spiking network by simply using simple LIF neurons (white) (a). In the DBN (b) the LIF neurons are put in a high conductance state (black) by inserting high frequency Poisson noise. The weights are scaled accordingly to fit the activations of the neurons.

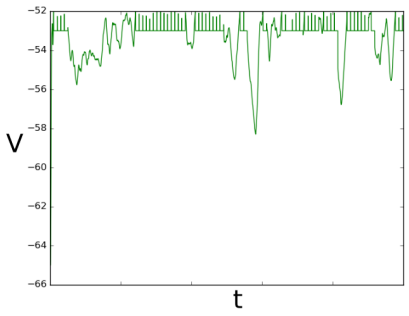
Conversion as CNN One way to convert the DBN to the spiking domain, is by interpreting it as a pre-trained CNN with purely forward connections (we do not perform any commonly used gradient descent fine tuning to get comparable results with only CD trained models, but fine tuning could further improve the performance). While this allows classification, it removes the generative capabilities of the network.

For the conversion, we proceed similar to Cao et al. and Diehl et al. [20, 24]. They use average pooling and ReLU functions to get a similar architecture as SNNs. In contrast, we don't use any pooling, since for RBMs there is no simple way to integrate average pooling (used by Cao et al. and Diehl et al. in their trained CNNs), and for spiking CNNs there is no simple way to integrate probabilistic max pooling, which is to our knowledge currently the only into the training integrated pooling thought of for RBMs. We also use the sigmoid function, since RBMs are commonly trained with sigmoid activations (but some approaches propose ReLU for RBMs as well [63]). Furthermore the input-rate output-rate transfer function of rate based LIF neurons with a refractory period matches the sigmoid function more closely.

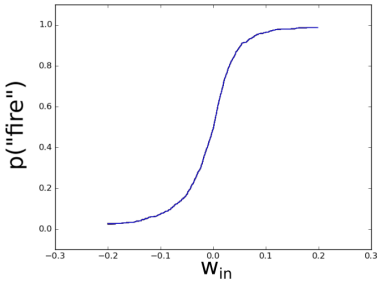
The DBN layers are each replaced by a LIF neuron population with an identical architecture (see Fig. 4.3a). The connections are replaced by (directed) synapses and the weights of the synapses scaled with a constant factor to get similar activations.

Conversion with conductance-based LIF neurons Another way to convert the DBN to the spiking domain is by interpreting it as a directed graphical model, a sigmoid belief network, and performing ancestral sampling. This approach is heavily based on the synaptic sampling theory, i.e. it uses spiking neurons to perform sampling. The sampling can be either performed with current-based or conductance-base LIF neurons as described in Chapter 3.2.

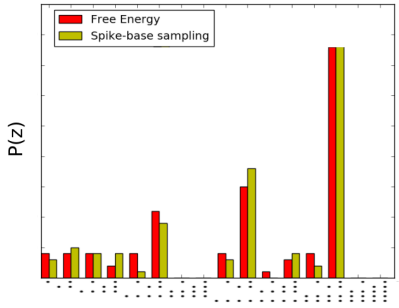
For the COBA neurons, we choose a biological plausible neuron model (see parameters in Table A.3). The high membrane conductance, an increased Gaussian distributed mean membrane potential and thus a firing probability of 0.5, is achieved by using high frequency Poisson generated inhibitory and excitatory spikes to get the neuron to a high conductance state (HCS) (see Fig. 4.4a).



(a) Firing of an HCS COBA LIF neuron.



(b) Input output transfer function of an HCS COBA LIF neuron.



(c) omparison between an HCS COBA LIF neuron sampling and the true distribution in an RBM.

Figure 4.4.: Properties of a conductance based LIF neuron put into a high conductance state with high frequency input noise. The neuron fires with a probability of 0.5 given no additional input (a). With input the input output transfer function approximates a sigmoid function (b). A network with such neurons is able to sample in a Boltzmann distribution similar to original distribution (c).

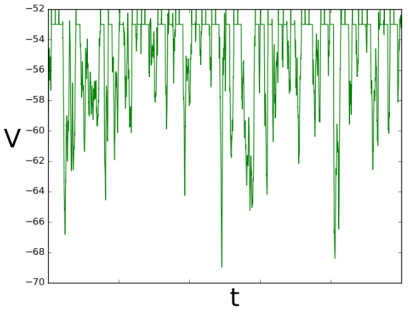
This neuron model has an input-rate output-rate transfer function which approximates a sigmoid function (see Fig. 4.4b).

The PSPs are chosen to have an alpha shape instead of a rectangular shape, which as described in Chapter 3.2 may introduce some discrepancies when performing sampling compared to Gibbs sampling, but is more biologically plausible (see Fig. 4.4c).

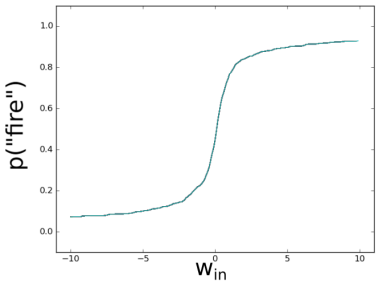
The DBN is converted by replacing each layer with a layer of conductance-based LIF neurons with Poisson noise, and the connections are transformed to synapses with the weights scaled to achieve an action function similar to the sigmoid function (see Fig. 4.3b).

Consequently, the DBN simply performs ancestral sampling with the data sample as evidences and the label as inferred state.

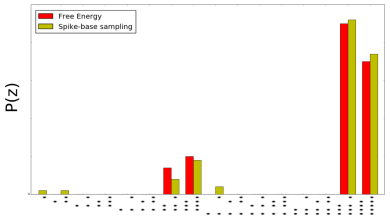
Conversion with current-based LIF neurons To reduce the computational expenses of the COBA models, they can be replaced by less computational complex CUBA models. The model parameters as chose to simulate an HCS state. Therefore, the membrane time constant is reduced and the membrane conductance is increased and a static input current is inserted. Adding high frequency Poisson noise results in a sigmoid shaped input-rate output-rate transfer function (see



(a) Firing of an HCS CUBA LIF neuron.



(b) Input output transfer function of an HCS CUBA LIF neuron.



(c) Comparison between an HCS CUBA LIF neuron sampling and the true distribution in an RBM.

Figure 4.5.: Properties of a current based LIF neuron put into a simulated high conductance state by high frequency input noise and an artificial high membrane conductance. The neuron also fires with a probability of 0.5 given no additional input (a) and with input the input output transfer function approximates a sigmoid function (b). A network with such neurons can also to sample in a Boltzmann distribution similar to original distribution (c).

Fig. 4.5).

The DBN conversion is similar to the COBA case, but instead of COBA LIF neurons the adapted CUBA LIF neurons are used (see Fig. 4.3b).

4.3. eCD

Another approach implements the training of the convolutional spiking DBN with an STDP based learning rule.

The main idea of the learning rule is adapted from Neftcis eCD and is applied to a network of LIF neurons with symmetric bidirectional synapses [64]. We modify the STDP learning rule described in Chapter 3.4 by extending the model with a learning rate. This rule can be reformulated as an iterative rule as follows:

- A visible unit v spikes:

$$A_v = A_v \exp(\frac{-\Delta t}{\tau}) + a_\delta,$$
$$A_h = A_h \exp(\frac{-\Delta t}{\tau}),$$
$$\delta w = \mu g(t) A_v,$$

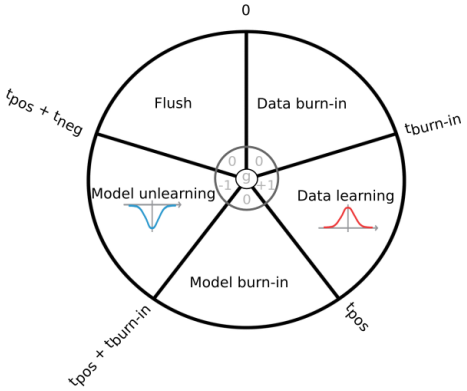


Figure 4.6.: The five phases for a data sample in the adapted eCD algorithm.

- A hidden unit h spikes:

$$A_h = A_h \exp(\frac{-\Delta t}{\tau}) + a_\delta,$$
$$A_v = A_v \exp(\frac{-\Delta t}{\tau}),$$
$$\delta w = \mu g(t) A_h,$$

where $g(t)$ is the STDP status flag, Δt is the time difference to the last previous spike, and a_{delta} represents the input of the incoming spike.

The original division into four training phases poses similarities to pCD since the activity of the hidden layer of the previous step is used as starting state for the next step. We extent the model by a 5th phase between two data samples, where the network is "flushed" thus enabling normal CD (see Fig. 4.6):

1. The data signal is applied and the system is allowed to model the data distribution ($g(t) = 0$).
2. Positive STDP is used to get $v_i h_j$-data and is added to the weights (postive phase $g(t) = 1$).
3. The data signal is removed and the system is allowed to model the model distribution ($g(t) = 0$).
4. Negative STDP is used to get $v_i h_j$-model and is subtracted from the synaptic weights (negative phase $g(t) = -1$).
5. The neural activity is "flushed" by inserting a strong negative current into the visible and hidden layer, no learning is performed ($g(t) = 0$).

4.3.1. Convolution in eCD

We implement convolutions with local receptive fields and shared weights between synapses through weight synchronization. Since each synaptic weight has its local STDP based update

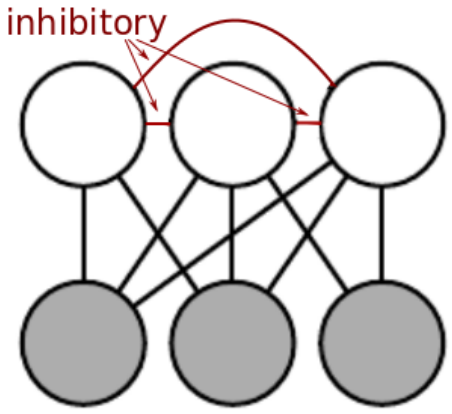


Figure 4.7.: A (restricted) Boltzmann machine with lateral inhibition in the top layer.

rule (eCD), we have to find a way to synchronize weights between synapses of the same feature map. To keep the shared weights the same, we perform an STDP weight synchronization step at discrete time steps, since updating all weights after a single update did not show any promising results, due to some self-reinforcement and resulting weight "explosion" (see A.4 for further insight). Thus the weight synchronization at a time step for shared weights w_i , w_j can be illustrated by the following rule:

$$w_i(t) = w_{shared}(t-1) + \delta w_i,$$
$$w_j(t) = w_{shared}(t-1) + \delta w_j,$$

which gives the new shared weight

$$w_{shared}(t) = \frac{1}{2}(w_i(t) + w_j(t)) = \frac{1}{2}(w_{shared}(t-1) + \delta w_i + w_{shared}(t-1) + \delta w_j) = w_{shared}(t-1) + \frac{1}{2}(\delta w_i + \delta w_j).$$

This results in an update rule similar to the convolutional RBM update rule as presented in Chapter 3.1. Thus we can simply take the mean of the weight changes and apply it to all the weights, which is equivalent to just taking the average of the new individual weights.

Lateral inhibition In addition, we introduce fixed negative connections between neurons in the hidden layers (which is more biologically plausible [48], see a simplified sample in Fig. 4.7). This removes one advantage of RBMs since the hidden units are no longer independent, which makes it harder to sample from the true distributions, but since the network continuously performs sampling steps, the approximation has shown to be sufficient, if the weights are not too strong and prevent changes of different modes (see Chapter 6.2.3 and Fig. A.1).

Connecting neurons to neurons on a similar position in other feature maps appears to make the features more discriminative and less correlated. This also poses some similarities to adding a negative structured bias to the hidden units, which has shown to result in better features [47, 50, 67].

An intuitive interpretation is that if one feature reacts to a certain input it will be highly active and prevent the others from being active as well and thus prevent them from learning the same features.

4.3.2. Spiking DBNs

To build up a spiking DBN, we train the convolutional BMs layer-wise and forward the input of the previous layer to the next layer.

Either replacing the symmetric bidirectional synapses with forward only synapses or keeping them bidirected did not show any significant differences (see Fig. A.2). To be exact the spiking DBN with bidirectional synapses in the bottom RBMs is a mixture between a deep belief network and a deep Boltzmann machine, since the single Boltzmann machines are still bidirectional connected and only the hidden layer activations are forwarded in a directed manner to the next Boltzmann machines (see Fig. 4.8).

Due to some top-down influences, when stacking a new BM on the trained BM directly, the hidden distribution gets distorted and unfit input for the new BM to be trained on. To solve this in our case, we use forward connections from the hidden layer of the bottom BM to the visible layer of a new two layered top BM. An approach to predict the influences of the top RBM in advance like Salakhutdinovs DBMs by doubling the weights or joint training may sound promising and could be an area for further research [73].

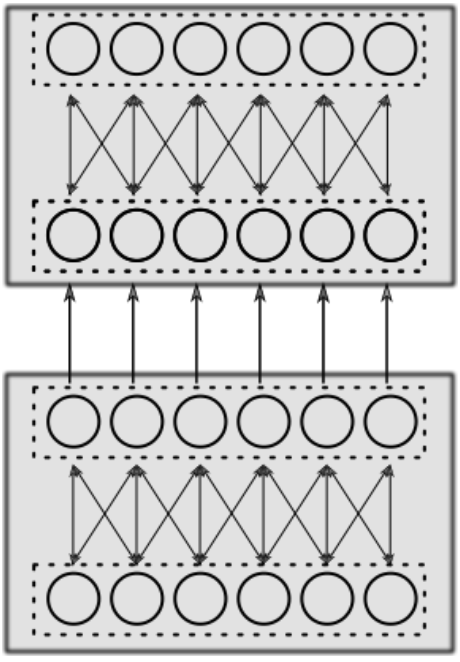


Figure 4.8.: The structure of a deep belief network trained with eCD. The spiking DBN consists of single Boltzmann machines stacked on top of each others. In contrast to artificial DBNs, the Boltzmann machines are still bidirection connected, and two Boltzmann machines are stacked up by forwarding the activations of the hidden layer in the bottom RBM to the visible layer in the top RBM.

5. Implementation

5.1. Artifical DBNs

The DBNs were implemented in the Theano-Framework [13] to utilize the computational power of the GPU. The implementation of the single RBMs is adapted from the official Theano RBM [9].

An upward pass in performed as follows: As an input we take a 4D tensor $\mathbf{V}$ of size *(batch size, input channels, input rows, input columns)* and for the upwards pass convolve it with a filter $\mathbf{W}$ of the shape *(number of filters, input channel, filter height, filter width)* with a stride of 1 and without any padding. This results in feature maps $\mathbf{H}_{pre} = \mathbf{W} * \mathbf{V}$ of the shape *(batch size, number of filters, input height - filter height + 1 , input width - filter width + 1)*. A constant c is added to the pre-activation feature maps as bias. On those feature maps a sigmoid function σ is applied and accordingly Bernoulli sampled to get the activation of the hidden units $\mathbf{H}$.

Algorithm 1 Upward pass

$H_{pre} \leftarrow W * V + c$
$H_{sigmoid} \leftarrow \sigma(H_{pre})$
$H \leftarrow H_{sigmoid} > Random()$

The downward pass on the activation of the hidden units is performed similar: The input is the activation of the hidden units $\mathbf{H}$ with zero padding and is convolved with the 180 degree flipped kernel $\tilde{\mathbf{W}}$ with the first and second dimensions swapped. Afterwards a sigmoid function σ is applied and accordingly sampled to get the new visible layer activations $\mathbf{V}'$.

Algorithm 2 Downward pass

$V_{pre} \leftarrow \tilde{W} * H + c$
$V_{sigmoid} \leftarrow \sigma(V_{pre})$
$V \leftarrow V_{sigmoid} > Random()$

One complete Gibbs sampling step consists of an upwards and a consecutive downward pass.

Algorithm 3 Gibbs step

$H_0 \leftarrow upward(V_0)$
$V_1 \leftarrow downward(H_0)$

Each RBM is trained with the CD-k update rule, therefore after performing one Gibbs sampling step to get a sample of the data distribution $\mathbf{V}_0, \mathbf{H}_0$ (positive phase), we perform $k-1$ additional Gibbs sampling steps to get a sample of an estimate of the model distribution $\mathbf{V}_k, \mathbf{H}_k$ (negative phase).

As error function we define the difference between the free energy of the positive phase and the negative phase:

$$\Delta \mathbf{W} = \frac{\partial F(V_k)}{\partial \mathbf{W}} - \frac{\partial F(V_0)}{\partial \mathbf{W}}.$$

We use Theanos auto-differentiation to determine the gradient and perform gradient decent with a learning rate μ and a weight decay of υ:

$$\mathbf{W}' = \mathbf{W} - \mu \Delta \mathbf{W} - \upsilon \mathbf{W}.$$

A common choice for μ is 0.1 and for υ is 0.003 throughout this thesis. We train the RBM for a given number of iterations over the dataset, with a batch size smaller than the dataset, thus performing stochastic gradient descent.

To build up the DBN, each RBM is trained greedily. After training one RBM the entire dataset is converted by performing one upwards pass in the trained RBM and using the activation of the hidden units as the new input data for the next RBM.

Algorithm 4 build DBN

for i in #*layers* **do**
  Train RBM_i on dataset ds_i
  convert dataset: $\text{ds}_{i+1} = upward_i(\text{ds}_i)$
end for

Up to this point no label information is used to train any RBM, thus the learned features are trained purely unsupervised. The last layer, the association layer, consists of a fully connected RBM trained on input data comprised of the converted data set concatenated with an one-hot encoding of the label.

5.2. Conversion

To simulate the converted DBNs, we use the PyNN framework with Nest as spiking network simulator [22, 29]. To simulate the CNN and DBN, we use current and conductance based LIF neurons, respectively, see Chapter 4.2.2 for more details. Each unit is injected with high frequency Possion distributed excitatory and inhibitory input spikes with the frequency $\lambda_{noise} = 5000$Hz and synaptic weights w_{n+} and w_{n-} respectively. A common choice in this thesis is $w_{n+} \approx -w_{n-} \approx 0.1$ for current-based LIF neurons and $w_{n+} \approx -w_{n-} \approx 0.005$ for conductance-based LIF neurons.

A unit in the DBN is represented by a neuron, a layer by a neuron population. For the connections static synapses with the scaled original weights were used.

The input is transformed to Poisson distributed spike trains, where the rate of the Poisson process λ_{data} is proportional to the original data value (e.g., the image intensities). The input is directly fed into the bottom layer neurons with a high fixed weight w_{in} to reliably generate equal spikes in the bottom layer.

For each data sample the network is simulated for a runtime of t. To get the classification results, the spike count of all single neurons in the label layer a_i is recorded and to get a label prediction

the index i of the most frequent spiking neuron is determined:

$$y_{pred} = argmax_i \, a_i,$$

with the size of the label layer being, due to the one-hot encoding of the label, equivalent to the number of different classes ($i \in \{0, ..., n_{classes}\}$).

5.3. eCD

The eCD learning was implemented in PyNN with Nest with shared synaptic weights and shared continuous STDP weight updates as well as in the Brian simulator with individual STDP weight updates and weight synchronization at discrete timesteps [22, 29, 33]. As shown in A.4 due to self-facilitation leading to a weight "explosion" when using convolutional shared weight updates, we chose Brian for most of our experiments.

As neuron type LIF neurons with high frequency input noise were chosen. Each input element $x_i \in \mathbf{x}$ of a data sample $\mathbf{x}$ gets transformed to a Poisson distributed spike train, with the rate λ proportional to the input value $\lambda \propto x_i$.

Each RBM consist of a visible and hidden layer. The visible layer consist of one neuron population with $n = |\mathbf{x}|$ neurons representing the input, the data population. If needed a second neuron population representing the label input $\mathbf{y}$, the label population, can be added to the visible units. The hidden layer is a neuron population of the size $k = number\ of\ filters \times (input\ height - filter\ height + 1) \times (input\ width - filter\ width + 1)$.

The data population is sparsely connected to the hidden layer with synchronized weights implementing the convolution, while the label population is fully connected to the hidden layer. In the hidden layer we connect a neuron to other neurons in a square around same position in different feature maps with an inhibitory synapse with a fixed negative weight.

The training of the spiking RBM is performed with the adapted eCD algorithm (see 4.3). The RBM is trained on one data sample for $t_{sample} = n \times t_{ref}$ cycles, which can, considering the different training phases, be further divided into $t_{sample} = t_{burn-in} + t_{learn} + t_{burn-in} + t_{learn} + t_{flush} = t_{pos} + t_{neg} + t_{flush}$ (with $t_{pos} = t_{neg} = t_{burn-in} + t_{learn}$), as visualized in Figure 4.6. As a result we receive the following training procedure:

- $t \in [0, t_{burn-in}]$: In the first phase ("Data burn-in phase"), the visible layer is induced with a strong negative current and the data input is given as spikes with a high synaptic weight, so that the visible layer only spikes in accordance to the input data and is unaffected from any spikes in the top layer. The STDP learning flag is set to $g = 0$, so no learning is allowed.

- $t \in (t_{burn-in},\ t_{burn-in} + t_{learn}]$: In the second phase ("Data distribution phase"), now the STDP learning flag is set to $g = 1$ so positive learning is allowed. This should drive the weights to represent the data distribution.

- $t \in (t_{pos},\ t_{pos} + t_{burn-in}]$: In the third phase ("Model burn-in phase"), we set the data input of the visible layer to zero and remove the induced negative current to let it reach the model

distribution. In this phase the learning is disabled, setting the STDP learning flag to $g = 0$.

- $t \in (t_{pos} + t_{burn-in},\ t_{pos} + t_{burn-in} + t_{learn}]$: In the fourth phase ("Model distribution phase"), the STDP learning flag is set to $g = -1$, enabling only negative (un-)learning. This will unlearn the model distribution.

- $t \in (t_{pos} + t_{neg},\ t_{pos} + t_{neg} + t_{flush}]$: In the optional fifth phase ("Flush phase"), we induce a strong inhibitory current to the visible and hidden layer to remove all activity and allow a fresh start in the first phase.

The weight synchronization is set at discrete time points, after n training samples. This is executed by taking the mean over all the weights, which are to be the shared. In addition a small weight decay is introduced:

$$W_{new} = \text{mean}(\mathbf{W}_{group}) - \upsilon * \text{mean}(\mathbf{W}_{group}),$$

where υ is the weight decay rate and $\mathbf{W}_{group} = (W_0, ...W_n)$ are all the updated weights of synapses belonging to a group of synapses with the same shared weights.
Each RBM is trained on m data samples.
After an RBM is trained, the next RBM is trained on top of the previous RBM. Therefore, a connection with a strong synaptic weight is established from the hidden layer of the previous RBM to the visible layer of the new RBM. The original input data is still fed to the bottom RBM while the activations of hidden layer in previous RBM act as training data for the top RBM.

At test time the network is run for a fixed timespan t_{test} with the test data fed to the bottom RBM. There is no external input forwarded to the label layer while the number of spikes in the label neurons are counted. The neuron with the most spikes represents the predicted label.

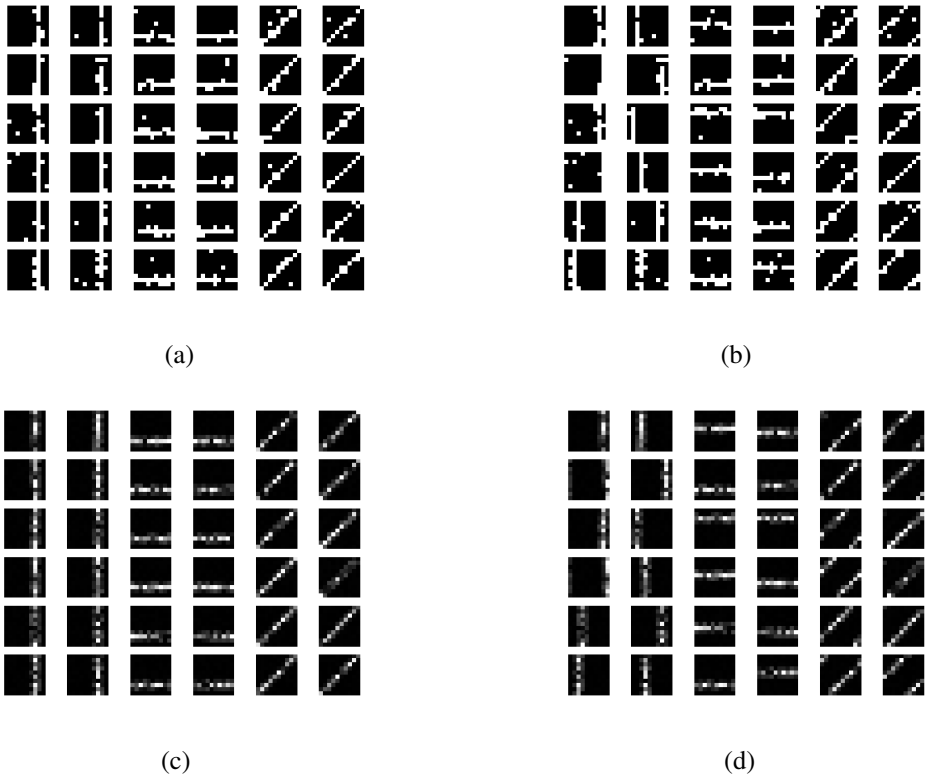
6. Experiments & Results

6.1. Datasets

We evaluate our models on three different datasets. The dataset size is primarily limited by the computational resources, such as memory and computation time.

6.1.1. Stripe Dataset

We generate a 10×10 pixel noisy stripe dataset with three different oriented stripes, horizontal, diagonal and vertical. This could represent an object similar to a pen in different orientations. In the easiest version of this dataset the stripes always occur on the same places with some random noise (see Fig. 6.1a). A more complex version of the datasets contains the stripes randomly distributed across the whole image (see Fig. 6.1b). This dataset can be either binary or continuous (see Fig. 6.1c & 6.1d).



(a) (b)

(c) (d)

Figure 6.1.: Samples from the 10×10 pixel stripe dataset. The stripes in (a) and (c) have the same position in the image, while the stripes in (b) and (d) can appear anywhere in the image. In (a) and (b) the images are binary , i.e a pixel value $p \in \{0, 1\}$, while in (c) and (d) the values are continuous i.e $p \in [0, 1]$.

6.1.2. MNIST

We also evaluate the models on the MNIST dataset [53]. The MNIST dataset consists of 60000 28x28 pixel gray images of handwritten numbers 0-9. Samples of the dataset are given in Figure 6.2.

Figure 6.2.: Samples from the 28×28 pixel MNIST dataset [53]. The pixel values p are scales to be in the interval $p \in [0, 1]$.

6.1.3. Poker-DVS

Another dataset used in this thesis is the Poker DVS dataset [76]. The dataset consists of 131 poker pip symbols extracted from 3 separate DVS recordings, while quickly browsing poker cards. From the 128×128 recorded image, a 32×32 pixel patch, containing the symbol is extracted. As a compromise between computational and classification performance, we down sample the patches to a size of 16×16 pixels and only consider spikes in the first $8ms$, due to the constraint learning time per sample. The resulting dataset is visualized in Figure 6.3 by integrating all events of a sample and then normalizing them by the maximal number of events per pixel.

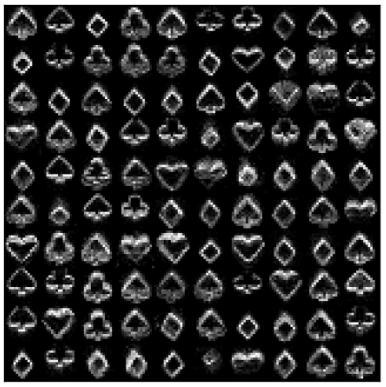


Figure 6.3.: Visualization of samples from the Poker-DVS dataset [76]. The images are generated by integrating all events over 8 ms. The actual training is performed on the events.

6.1.4. Ball-Can-Pen-DVS

For this thesis we recorded a dataset of three different object categories with a Dynamic Vision Sensor (DVS), the Ball-Can-Pen-DVS (BCP-DVS) dataset. Motivated by different types of grasps, balls, cans and pens were chosen. To generate the dataset, images of these objects were flashed for 200*ms* at different positions on an LED-display and recorded with a DVS. The recordings were further divided into 100*ms* samples, which results in 90 samples for each class. Similar to the Poker-DVS dataset, the 128×128 pixel samples were scaled to a size of 16×16 pixels.

The original recorded events and the scaled events are visualized in Figure 6.4 by integrating all events of a sample and then normalizing them by the maximal number of events per pixel.

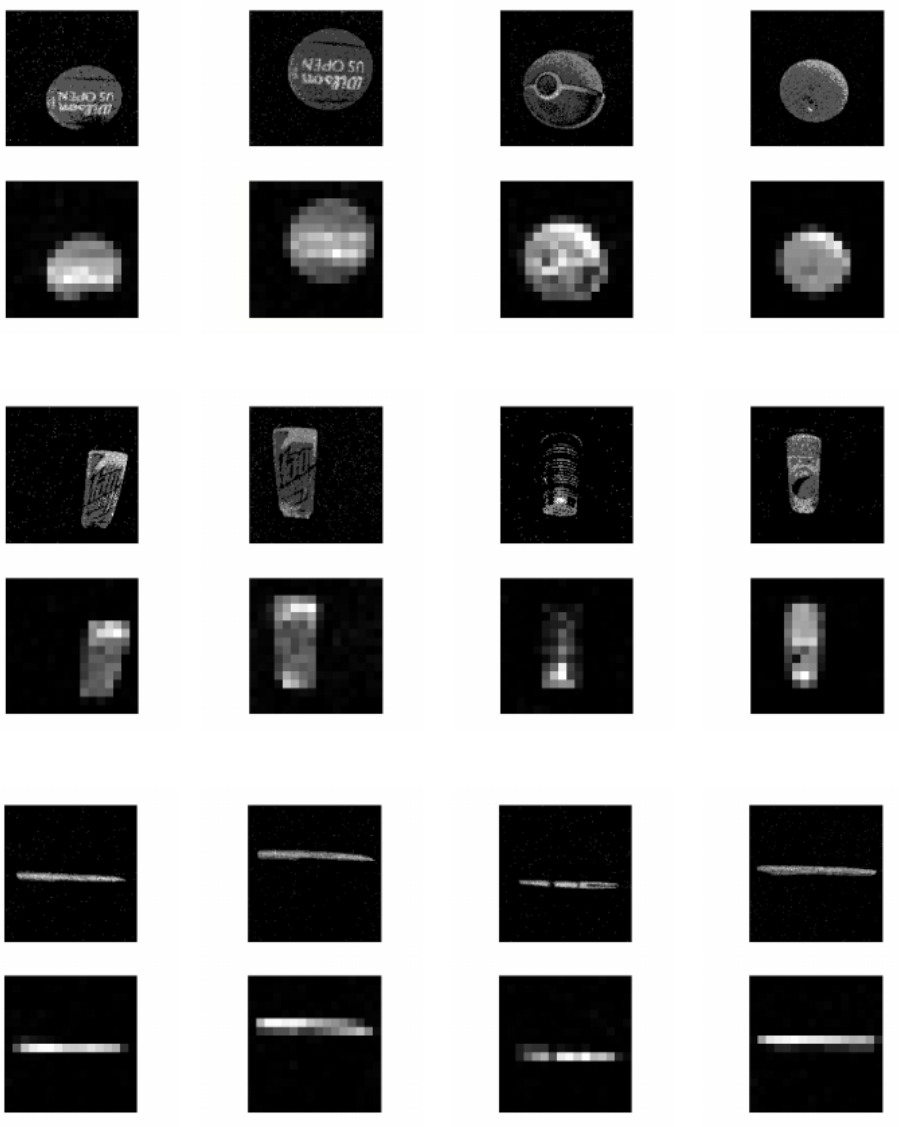


Figure 6.4.: Recorded and down sampled samples from the Ball-Can-Pen-DVS dataset visualized by integrating all events of a sample and then normalizing them by the maximal number of events per pixel. Each first row shows the original sized DVS samples with the corresponding scaled samples in the second row. In the top rows are samples of different Balls, in the middle rows are samples of Cans and in the bottom rows are samples of Pens presented.

6.2. Experiments

We focus our experiments primarily on the stripe dataset, due to time and computational constraints, which are explained first. We then look at and compare the performance of the converted models. Afterwards, to get better insight into the eCD algorithm, we evaluate our models on the stripe dataset and then look at the performance on other datasets for comparison and generalization. In the end we compare those two approaches.

6.2.1. Computational Constraints

Most of the experiments were executed on a quad-core processor with 3.2 GHz, 16 GB RAM and a NVidia GTX 650.

In the ANN case, CNNs can be optimized to utilize the processing units (e.g., the GPU) of most computers. Due to the shared weights, only one copy of each convolutional filter has to be stored. In addition, each feature map has to be calculated and stored as well. Calculating a features map can be reduced to a convolution operation over the previous feature maps with a kernel, which can be implemented quite computationally efficient. Thus the data needed for an artificial CNN is given by the size of the feature maps and the kernel matrices, the operations can be primarily reduced to convolutions.

In contrast to CNNs, spiking networks with a convolutional structure can not utilize all of the benefits. While the shared weights still reduce the number of learning steps needed compared to a network without shared weights, the structure is more complicated. Due to the time dependent nature of the synapses and neurons, each single neuron and synapse has to be explicitly modelled. While the number of neurons is equivalent to the number of elements in a feature map and thus comparable to the units in a classical CNN because all synapses have an internal state, synapses with shared weights cannot be reduced to a single synapse as in kernel matrices. In addition to the additional computations needed for the neuron dynamics, these synapse dynamics, which do not have to be modelled in classical CNNs, can drastically increase the computational complexity.

For example a CNN over $5 \times 5 = 25$ dimensional input data with a $4 \times 4 = 16$ kernel would result in a $2 \times 2 = 4$ feature map so the number of stored values is 45. For a spiking network, in addition to modelling the 25 dimensional input data and the 4 feature maps as time dynamic neurons, now $(2 \times 2) \times (4 \times 4) = 64$ synapses and their dynamics have to be modelled. This number increases quadratically as the filter size is reduced or the number of filters is increased.

Due to these constraints, the experiments primarily focus on smaller datasets and are not performed on current state-of-the-art image recognition datasets. Specialized neuromorphic hardware, e.g., SpiNNaker or Spikey as discussed in the "Future Work" Chapter 7.1.1, could speed up the computations and allow the evaluation of more complex datasets. In addition, most spiking neural network frameworks have, in contrast to most artificial neural network frameworks, which are optimized for machine learning and speed, their main focus on biological imitation and plausibility.

6.2.2. Conversion

At first we compare the three different converted DBNs as introduced in Chapter 4.2.2. Therefore, an artificial DBN was trained on the MINST dataset. The dataset was split into 50.000 training images and 10.000 images for testing. The DBN consists of three RBMs (similar to Fig. 4.2), where the first RBM is convolutional with 20 filters of size $1 \times 16 \times 16$. The second RBM has 15 filters of size $20 \times 10 \times 10$. The third RBM, which can also be seen as the association layer of the DBN, is fully connected and uses the output of the second RBM as well as the labels as input data. Each RBM is trained for 5 epochs with CD-2 over the whole dataset with a learning rate of 0.1 and a weight decay of 0.0003.

The resulting features of the first layer are visualized in Fig. 6.5.

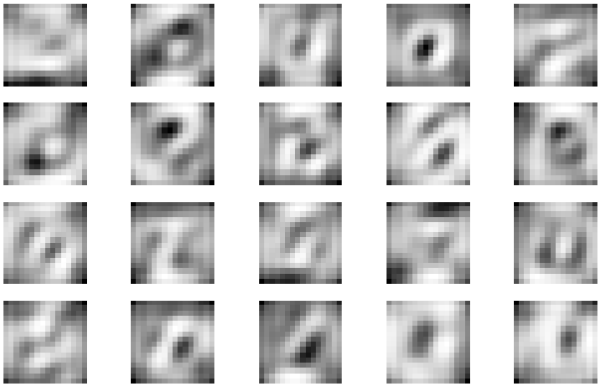


Figure 6.5.: Visualized 16×16 filters of the first layer convolutional RBM of a DBN trained on the 28×28 pixel MNIST dataset.

In the end the DBN reaches a classification performance of 82.45 %. This could be further improved by increasing the training time, fine-tuning the structure and parameters of the training algorithm of the DBN, but this was not the focus of this thesis. An intentionally simple DBN structure was chosen due to the computational effort necessary to simulate the DBN as a spiking neural network.

Conversion comparison

For the comparison of the conversion methods, we convert the RBM to the different spiking architectures, as described in Chapter 4.2.2. The input values of the images are converted to Poisson spike-trains with a maximal frequency of $\lambda = 100$Hz . Each input sample is simulated for 500ms, but we inspect how the simulated timespan affects the output later on. Since it can take approximately one minute real time to simulation 500ms of such a network, we evaluate all approaches on the same randomly drawn subset of the test set consisting of 100 samples.

Table 6.1.: Kullback-Leibler divergence between the activations in the feature maps.

Layer	Spiking CNN	COBA LIF DBN	CUBA LIF DBN
1	0.014	0.014	0.014
2	0.925	0.072	0.069
3	0.260	0.170	0.248
4	0.296	2.61	1.067
5	0.624	1.254	0.445

Table 6.2.: Classification performances of the converted spiking DBNs to a the artificial DBN on a subset of 100 samples.

	Classification Accuracy	Average runtime per sample
DBN	0.87	0.5 s
Spiking CNN	0.83	24.2 s
COBA LIF DBN	0.92	141.3 s
CUBA LIF DBN	0.93	30.6 s

A visual inspection of the activations can be seen in Figure 6.6. It is quite apparent that all approaches capture the basic activation characteristics. While the conversion as a DBN is able to simulate the activation probabilities more closely to the original DBN, the spiking CNN shows a more deterministic behaviour replicating only the essential activation structure. To quantify those differences we calculate the Kullback-Leibler divergence between the activation, which indicates the similarity of two distributions (the lower the value, the more similar are the distributions), as seen in Table 6.1.

The performance on the partial dataset can be seen in Table 6.2. It is apparent that all approaches show a similar performance. Interestingly the spiking DBN sometimes shows better performance than the RBM. Most misclassifications can from a human perspective be categorized as "reasonable" mistakes (as seen in Fig. 6.7) and hint, that further fine-tuning of the DBN could resolve some misclassifications.

Another important factor is the simulated runtime for each sample. It is quite apparent from Table 6.3 that a longer classification time benefits the performance.

In contrast to this approach in which the network is trained in discrete timesteps, next we look at the performance of networks trained in continuous time.

Table 6.3.: Classification performances of the converted spiking DBNs with different simulated runtimes.

	Spiking CNN		CUBA LIF DBN	
Simulated time	Classification Accuracy	Runtime	Classification Accuracy	Runtime
50 ms	0.69	7.8 s	0.81	8.1 s
100 ms	0.77	9.2 s	0.89	10.5 s
200 ms	0.76	13.1 s	0.89	14.6 s
300 ms	0.75	15.1 s	0.91	18.5 s
500 ms	0.83	24.2 s	0.93	30.6 s

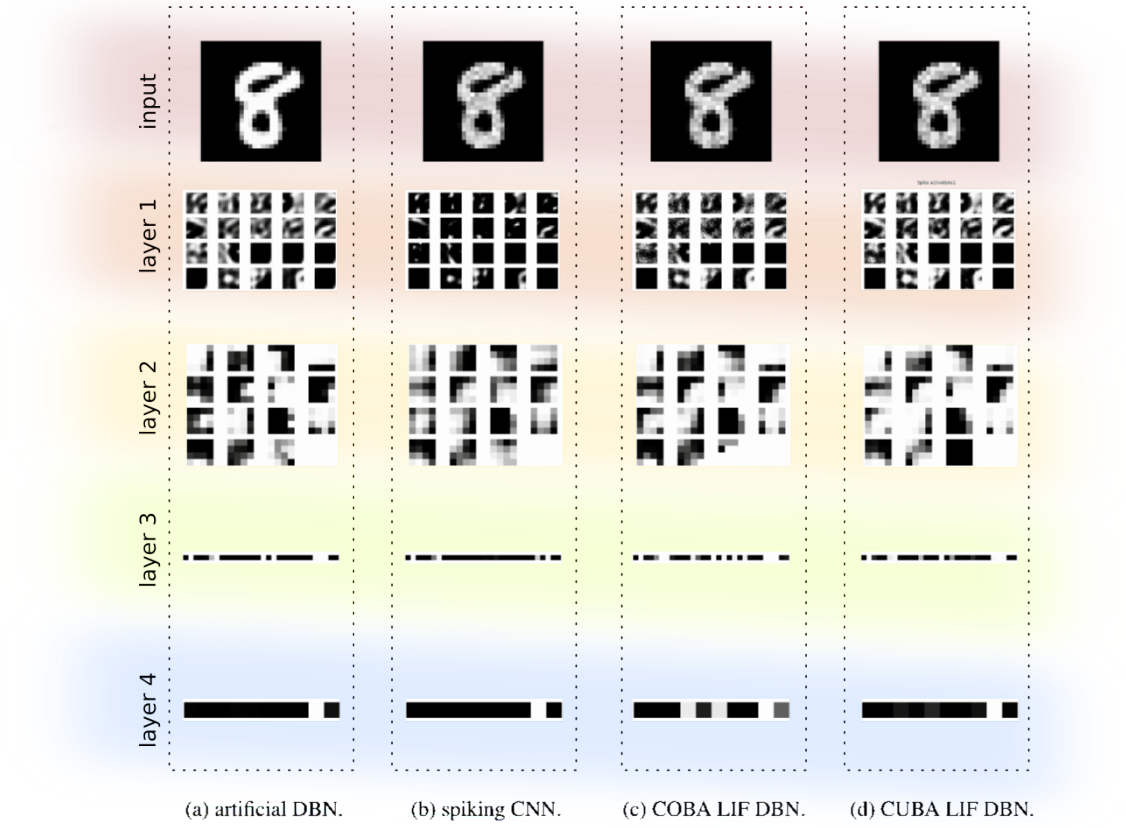


Figure 6.6.: Activations in the features maps in an artificial convolutional DBN and the converted spiking network architectures.

Figure 6.7.: Misclassifications of the converted spiking DBNs on MNIST. Often 9 and 4 are mixed up. The first three are classified as a 9, the fourth one is classified as a 4. The images all share a lot of features with the suggested class and the correct class was always the second most probable.

6.2.3. eCD

To get an intuitive insight into the eCD algorithm, we visualize the positive and negative phase for the stripe dataset. It is apparent that in the positive phase a sample from the data distribution is learned and in the negative phase the model distribution is unlearned (see Fig. 6.8). As the training progresses and the model distribution approximates the data distribution more closely the weight updates become smaller (see Fig. 6.9). The eCD-parameters chosen for most of our experiments are given in Table 6.4.

All this indicates some learning and we further inspect the capability of this algorithm to learn the data distribution.

Table 6.4.: eCD-parameters for most experiments.

$t_{burn-in}$	14 ms
t_{learn}	56 ms
t_{flush}	28 ms
Learn-rate	1.0
Weight-decay	0.001
Weight synchronization after n samples	1

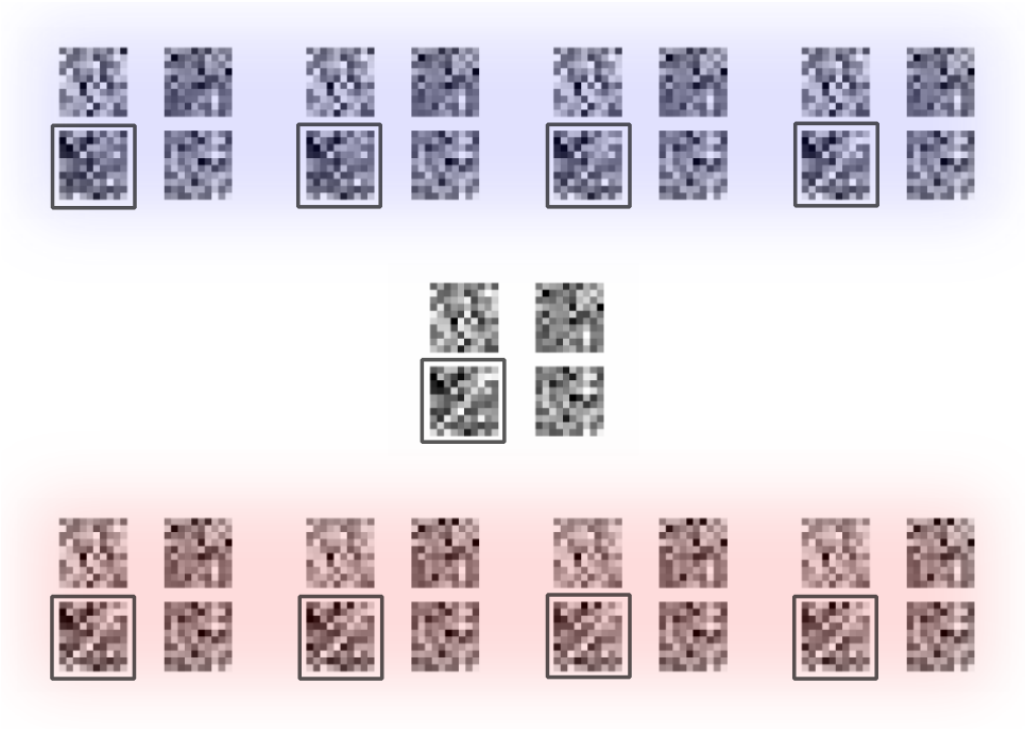


Figure 6.8.: Positive phase and negative phase of a diagonal stripe with 4 filters. In the first 4 images sets a stripe is learned, while during the last 4 images the model distribution is unlearned.

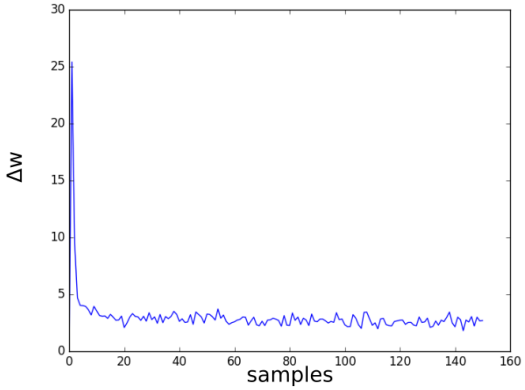


Figure 6.9.: Accumulated weight change for each training sample. As more samples are presented and the weights specialize, the total weight updates decrease.

Learning the data distribution

To evaluate how good the model distribution matches the data distribution, we compare the input sample to the reconstruction obtained after the removing the input and letting the system settle towards it's model distribution. This is visualized in Figure 6.10 and 6.11. In the beginning of the training the model distribution and thus the reconstruction is approximately random. As the training progresses, the reconstruction matches the original input image more closely and thus the model distribution shifts towards the data distribution.

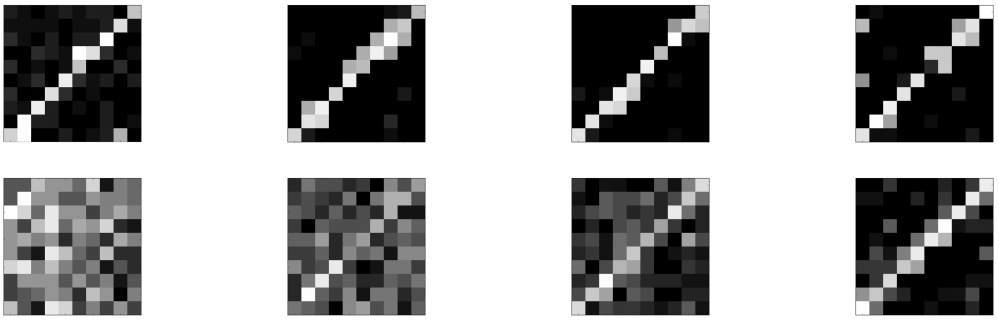
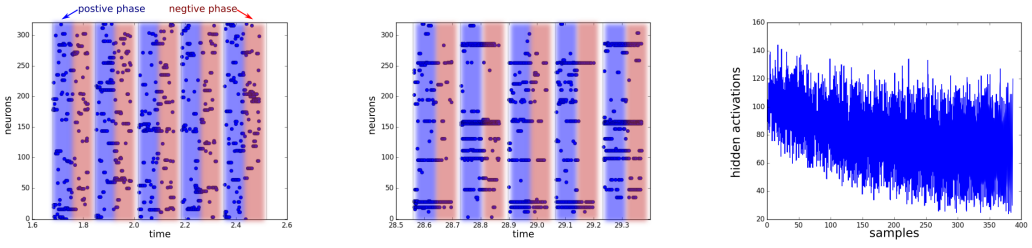


Figure 6.10.: Visualization of the spikes in the visible layer during the positive phase and negative phase of a diagonal stripe. In the first column, at the beginning of the training, in the negative phase the network is not able to reconstruct the data distribution. As the training progresses (in the second and third column), the reconstruction approximates the original data sample more closely and is in the last column nearly perfect.

In addition, the filters become more specialized, thus resulting in a sparse activation of the hidden layer as shown in Figure 6.11.



(a) Spikes at the beginning. (b) Spikes after some training. (c) Total number of spikes per sample over time.

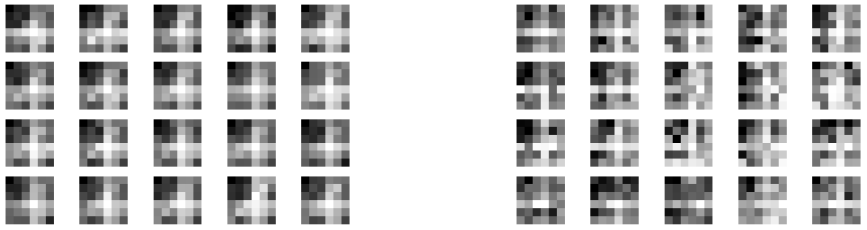
Figure 6.11.: Spikes in the hidden layer of a spiking convolutional RBM. At the start of the training (a) is appears mostly random, while after training the activations become more sparse ((b) and (c)) and certain neurons become specialized on certain input patterns (b).

As it turns out, to approximate the data distribution and to learn discriminative features, especially in the convolutional case, lateral connections can be used.

Lateral connections

An important addition to the convolution spiking RBM are the lateral inhibitory connections in the hidden layer. Each neuron in the hidden layer at position i, j in feature map k is connected

to other neurons in all other feature maps $K \setminus k$ at the approximately same position i', j' with $|i' - i| \leq 2, |j' - j| \leq 2$. This results in faster training and more discriminative features and better results (see Fig. 6.12 for the learned weights of both approaches).



(a) (b)

Figure 6.12.: 5×5 convolution filter matrices with and without lateral inhibitory connections in the top RBM layer on the stripe dataset. In (a) the weight are trained without lateral inhibitory connections and in (b) they are trained with lateral inhibitory connections. Whereas the filters without lateral connections (a) look more similar the filters with lateral connections (b) are more different and discriminative.

By using the lateral connection we can get some insight into the benefits of a convolutional architecture.

Convolution vs. no Convolution

To demonstrate the benefits of convolution even on a small dataset, we train two DBNs with the same number of free parameters, one convolutional, one without any convolution. The parameters for the first layer are chosen to be 20 and for the second "association" layer 250.

For the convolution DBN this results in a structure of 3 $7x7$ convolutional filters in the first layer and a fully connected RBM with 5 hidden units in the second layer. For the DBN without convolution, this results in two fully connected RBMs, where the first RBM has 2 hidden units and the second RBM has 120 hidden units.

Both DBNs are trained with 100 samples for the first layer and 500 samples with labels for the second layer, since it has more free parameters. The resulting weights are visualized in Fig. 6.13.

(a) Weights of the DBN with convolutions. (b) Weights of the DBN without convolutions.

Figure 6.13.: Weights of the first layers of the DBNs with and without convolutions with the same number of free parameters. In (a) are weights of the DBN with convolutions visualized and in (b) the weights of the DBN without convolutions.

This indicates that the DBN with convolution is able to capture the structure of the data better with the same number of free parameters. This is also apparent in the final classification results. While the DBN without convolutions reaches its top performance of 60% after 500 samples, the convolutional DBN reaches its top performance of 100% already after being presented with 300 samples. One disadvantage of convolutions is the increased runtime, which we inspect in the next section.

Simulation time

We also evaluate how the simulation time for one time step changes as the network complexity increases. A network on the 10×10 pixel stripe dataset is run for 1 learning step, which simulates a time of 168ms, and the runtime is measured. As we increase the number of hidden units, the runtime increases more or less linearly (see Fig. 6.14).

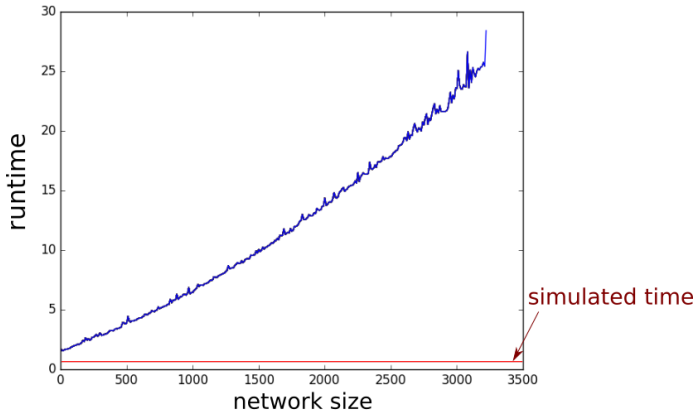


Figure 6.14.: The runtime of a learning step in dependence on the network size. While the simulated time of 168ms stays constant, the runtime increases linearly with the number of hidden units. If the blue line matched the red line, the simulation could be performed in realtime.

After inspecting some mechanisms and properties of the eCD algorithm, we next look at the performance of more complex DBNs trained with eCD on different datasets.

Performance on the stripe dataset

The DBN consists of two layers with the first layer consisting of a convolutional RBM with 20 filters of size 7×7 and the second layer being fully connected with 20 hidden units (see Fig. 6.15 for the abstract architecture). The first layer is trained with 1000 samples without labels. The weights of the first layer are then keep fixed and the second layer is trained over 500 samples with their corresponding labels. The development of the weights in the first layer can be seen in Figure 6.16. Another indicator of the training progress is given by the spikes as seen in Figure 6.17 & 6.18. The resulting spiking DBN show shows a performance of 100% classification accuracy on the test set.

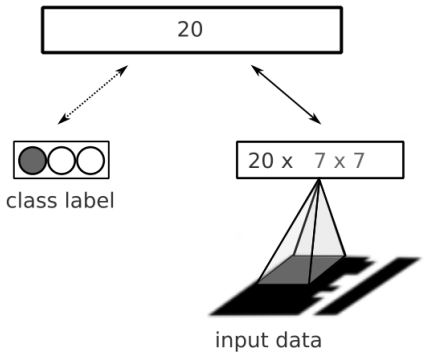
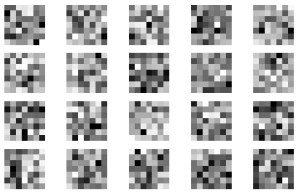
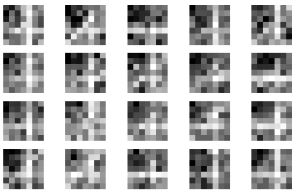


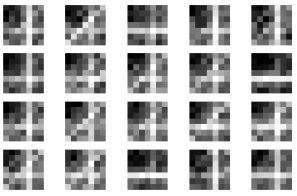
Figure 6.15.: Abstract architecture of the DBN for the stripe dataset.



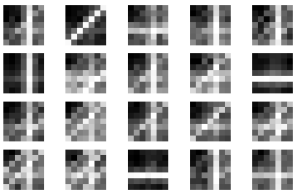
(a) Weights a the beginning of training.



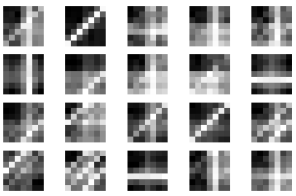
(b) Weights after 250 samples.



(c) Weights after 500 samples.

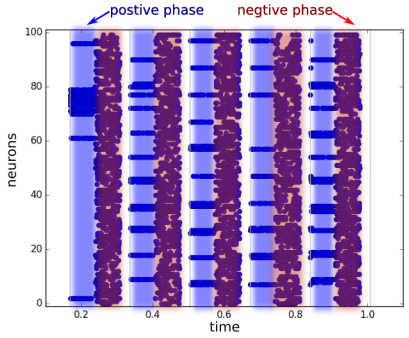


(d) Weights after 750 samples.

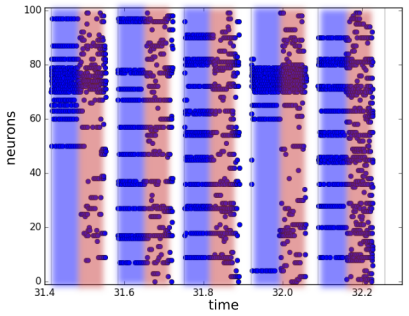


(e) Weights after 1000 samples.

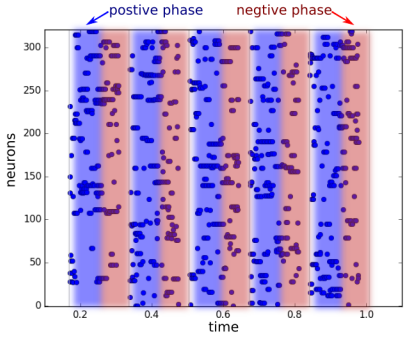
Figure 6.16.: The development of the 5×5 convolution filter matrices in the first layer of a DBN during training with the convolutional eCD algorithm on 1000 samples of the stripe dataset.



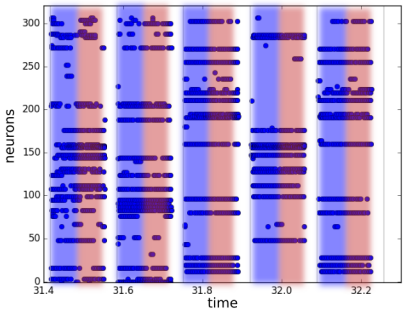
(a) Visible layer activations in the beginning.



(b) Visible layer activations after 1000 samples.

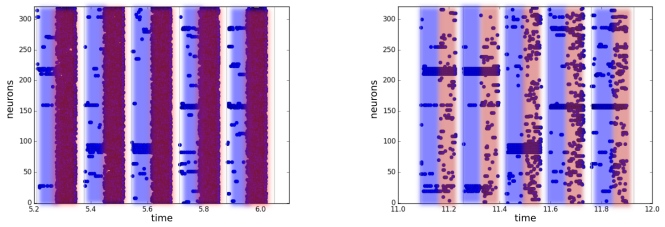


(c) Hidden layer activations in the beginning.



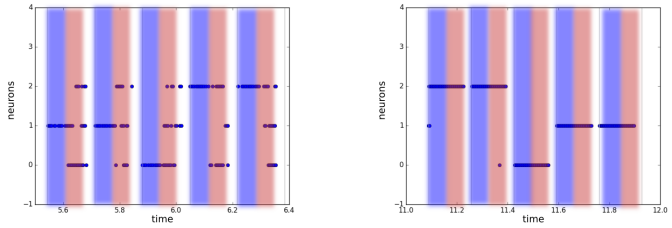
(d) Hidden layer activations after 1000 samples.

Figure 6.17.: Spikes in the layers of the first RBM during training. As the training progresses, the activations become more sparse. The hidden layer learns new representations for the data. The model distribution approximates the data distribution resulting in a nearly perfect reconstruction of the input data.



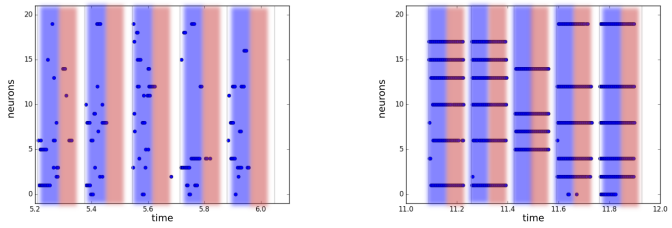
(a) Visible layer activations in the beginning.

(b) Visible layer activations after 500 samples.



(c) Label layer activations in the beginning.

(d) Label layer activations after 500 samples.



(e) Hidden layer activations in the beginning.

(f) Hidden layer activations after 500 samples.

Figure 6.18.: Spikes in the layers of the second RBM during training. As the training progresses, the activations become more sparse. The hidden layer learns new 20 dimensional representations for the data. The model distribution approximates the data distribution resulting in a nearly perfect reconstruction of the input data, especially of the label.

Performance on the Poker-DVS dataset

For the Poker-DVS dataset, a DBN with two layers, a convolutional and an association layer, is used. The convolutional RBM is defined by 10 filters of size 14×14. The association RBM layer has 10 hidden units (see Fig. 6.19 for the architecture). Each layer is trained with 200 samples drawn randomly from the training set. The resulting filters can be seen in Figure 6.21. To evaluate the extracted features, we show a reconstruction and completion of the input data (see Fig. 6.22 - 6.24) and determine the classification performance. The trained DBN reaches a peak performance of 94% on the training set (see Fig. 6.20).

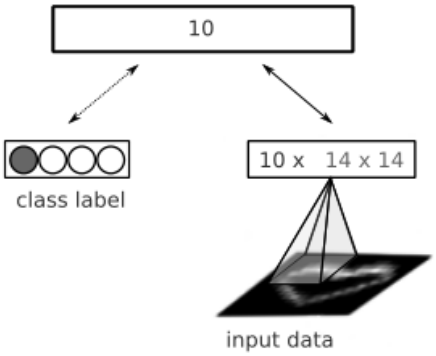
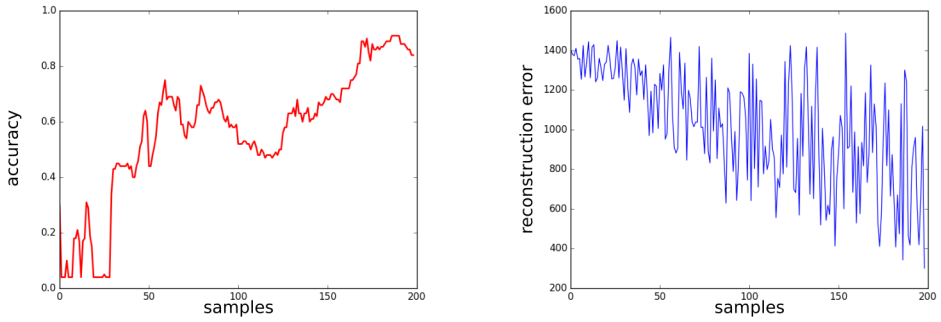


Figure 6.19.: Abstract architecture of the DBN for the poker dataset.



(a) Accuracy.

(b) Reconstruction error.

Figure 6.20.: The accuracy and reconstruction error of the DBN on the Poker-DVS dataset. The accuracy increases to a maximal value of 94% and the reconstruction error decreases indicating discriminative features.

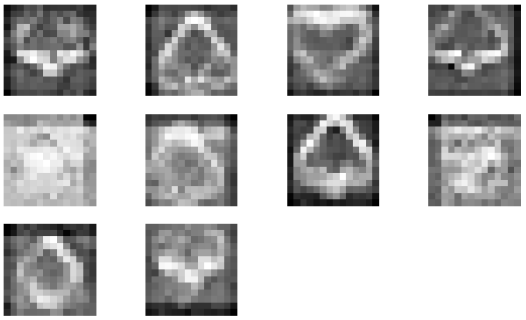


Figure 6.21.: Visualization of the convolution filters of the first layer RBM trained with the convolutional eCD algorithm on the Poker-DVS dataset.

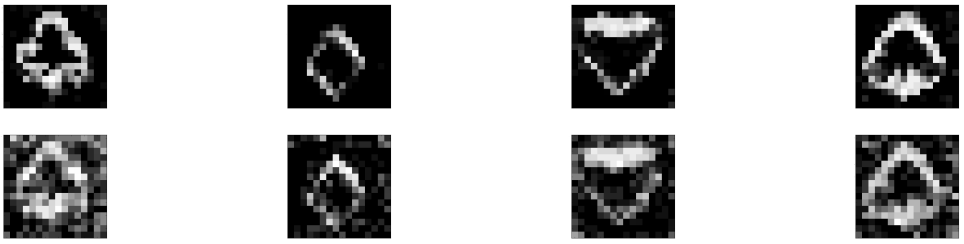


Figure 6.22.: Positive phase and negative phase in the first layer of the DBN of after training on the Poker-DVS dataset. The reconstruction of each class (bottom) is nearly perfect indicating, the DBN has learned the basic structure of the data.

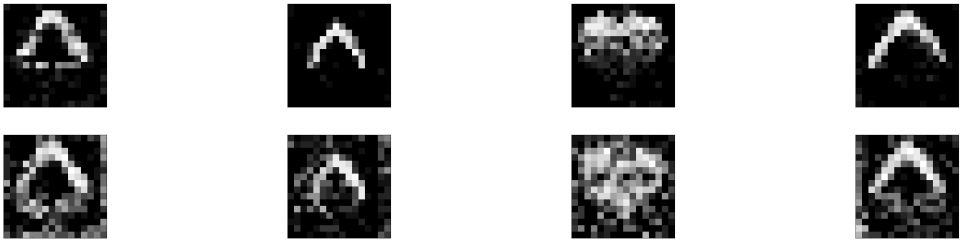


Figure 6.23.: Completion of partially presented input data of the Poker-DVS dataset. As input data, the bottom half of the image data was omitted (top). After training, the DBN was able to reconstruct and complete some parts of the input data (bottom).

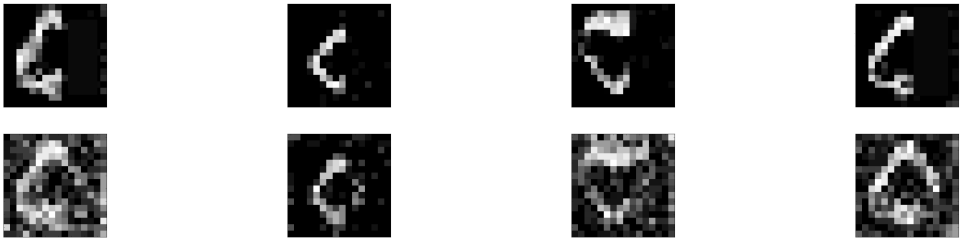
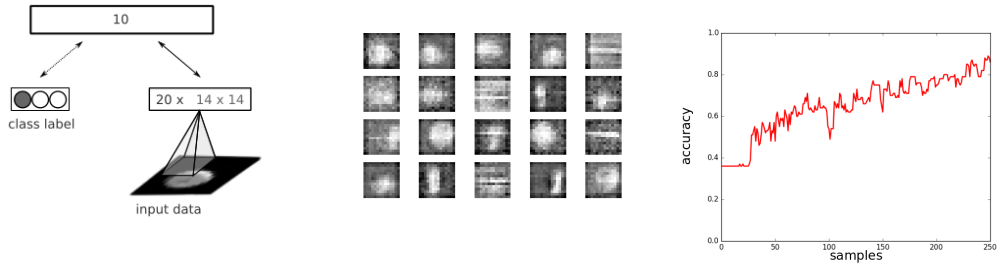


Figure 6.24.: Completion of partially presented input data of the Poker-DVS dataset. As input data, the right half of the image data was omitted (top). After training, the DBN was able to reconstruct and complete some parts of the input data (bottom).

Performance on the Ball-Can-Pen-DVS dataset

The DBN for the Ball-Can-Pen-DVS dataset is similar to the DBN used for the Poker-DVS dataset, but with 20 filters of size 14×14 in the first layer. The second "association" layer has 10 hidden units (see Fig. 6.25a). Each layer is trained once over all samples in the dataset, i.e. for 270 training steps. The resulting convolutional filters are visualized in Figure 6.25b.

The DBN reaches a peak performance of 90% on the training set (see Figure 6.25c).



(a) (b) (c)

Figure 6.25.: Performance on the Ball-Can-Pen-DVS dataset. (a) Abstract architecture of the DBN for the Ball-Can-Pen-DVS dataset. (b) Visualization of the convolution filters of the first layer RBM trained with the convolutional eCD algorithm on the Ball-Can-Pen-DVS dataset. (c) Accuracy of the DBN during training with over the whole Ball-Can-Pen-DVS dataset.

6.2.4. Conversion vs. eCD

At last we compare the converted DBN with a directly trained spiking DBN.

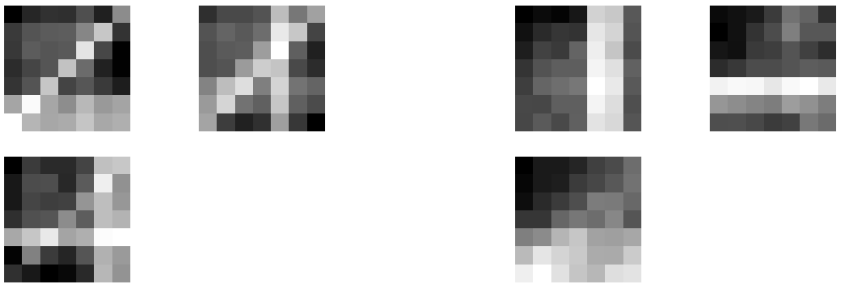
Direct comparison

We train an artificial DBN and a spiking DBN with the same architecture and number of samples and compare their performance.

We choose a simple structure with 3 convolutional filters of size 7×7 for the first layer RBM and for the second RBM we use a fully connected RBM with 5 hidden variables. The first layer of the DBN is trained unsupervised on 100 samples from the stripe dataset and the second layer is trained on 500 data samples with their labels from the stripe dataset. The artificial DBN is trained with a batch size of one to get a similar learning behaviour as the spiking DBN. The learning rates are chose to show similar weight alterations in each step and to show best classification performance after being presented with 500 data samples.

The resulting convolutional filters are visualized in Fig. 6.26.

It is apparent that the weights of the first layer in the spiking DBN are more discriminative compared to the artificial DBN. This also shows in the classification performance, where the converted artificial DBN only reaches 67% and the spiking DBN reaches 100%. This divide can be i.a. be attributed to the inhibitory lateral connections in the spiking DBN.



(a) Weights of the artificial DBN.

(b) Weights of the spiking DBN.

Figure 6.26.: Convolutional filters of the first layers of the artificial and spiking convolutional DBNs after 100 training samples. The weights of the spiking DBN appear to be more discriminative.

7. Conclusion and Outlook

In this thesis we demonstrated how to train and build a spiking network to generate discriminative higher level features, which natively suit event-based data. Although many systems for discrete data, such as images, have been proposed, event-based systems are still mostly neglected. Systems working on event-based data with highly distributed computational units can lower the power performance, the response-time and show more biologically plausible performance. Using a biologically inspired convolutional architecture has allowed systems to exploit the hierarchical structure in data and to facilitate training [49]. In this thesis two different approaches with a focus on biological resemblance to build a convolutional event-based system were shown. The first system trains a probabilistic graphical model, a deep belief network, on visual input by using the contrastive divergence algorithm to approximate some data distribution. This probabilistic model was transferred to different variations of spiking neural networks. We outlined the similarities between the deep belief network and the spiking neural network and evaluated and compared their performance on a classification task. We showed that the conversion to spiking network did not degrade the performance of the system. The second system directly trains a spiking neural network using a version of spike time dependent plasticity to simulate the contrastive divergence algorithm. By training consecutive parts of the network layerwise, a more complex system was built. Due to the computational complexity, we evaluated the performance on small visual datasets. We showed how lateral inhibitory connections allow more discriminative features and allow a classification accuracy of 100 % on a dataset. At last we compared those two approaches and showed how the event-based approach is able to generate more discriminative features and learns faster than the other approach.

Biological plausibility

Studies by Hubel and Wiesel suggest certain neurons respond so similar features at different position of in visible field [44]. This suggests similar synaptic weights, but due to contortions in the cornea most probably not exactly the same weights. One explanation for this could be shared weights similar to CNNs. Since the weight updates in the brain are primarily believed to be local, there is no known principle to keep weights between different neurons in the brain synchronized [75]. So while the trained structure with receptive fields and similar weights is quite plausible, the training procedure presented here is not. A more plausible way in the brain to get similar weights is due to the similarity of the inputs, e.g., if two receptive fields get quite similar input, their weights will with a high probability converge to the same target values. While this requires all receptive fields to be presented with the complete data, presenting each field with some part of the data but updating all fields with a combined update can be more computational effective.

This could be a principle CNNs utilize to get biological plausible result, while performing not completely biological plausible updates.

Another biological not completely plausible part of our presented system are the bidirectional synapses. This in turn could be easily translated to two directional synapses with some weight synchronization. While in this case local updates are sufficient to keep the weights similar (e.g., applying a similar learning rule to both weights) and research on discrete NNs has shown some automatic weight synchronization in autoencoders [84], there is no biological proof. The STDP flag determining either completely positive or negative and dividing the training into different phases does not appear to be plausible as well.

Even so this system has many constraints, it might could be counted among one of the more biological plausible deep learning architectures [15].

7.1. Future Work

While this system shows promising results on small dataset, there is still a lot of space for improvement on the performance and biological plausibility level.

7.1.1. Neuromorphic Hardware

One of the biggest constraints of the here proposed frameworks is the simulation speed of neural simulations. One way the training could be speed up is to perform the simulations on dedicated Neuromorphic hardware. There exists a few platforms which could be used, each having different advantages and disadvantages.

SpiNNakker is a massively-parallel computer architecture, which can use up to 1 million ARM processors to simulate up to 1 billion neurons [45]. In addition, it can be interfaced with PyNN. While implementing a custom STDP rule is quite feasible although having only integer weights, keeping weights synchronized either for symmetric connections or for convolution poses a bigger challenge whose complexity exceeds this thesis.

Spikey emulates spiking neural networks with conductance based leaky integrate-and-fire neurons and synapses implemented in analog microelectronics [70]. It can simulate 384 neurons and 256 incoming synapses in each neuron resulting in a total of 98304 synapses. It can also be interfaced with PyNN. This can allow approximately 10000-fold faster computations than biological real-time. But due to the weights, which are implemented directly in hardware and the noisy and less fault tolerant analog electronics this poses challenges for a "phased" custom STDP mechanism and weight-sharing.

While implementing a custom STDP rule on both neuromorphic devices could be quite challenging, applying already trained and converted spiking neuronal networks for a fast and energy efficient feature extraction should not pose any problems.

7.1.2. Biologically Plausible Learning

Another direction, which poses great potential for further research, is tackling the current restrictions on the biological plausibility.

One of the primary problems is the division in distinct learning phases. This is currently an active topic of research and there are different algorithms proposed, which loosen the constraints on explicit positive and negative learning phases. There are two approaches which appear in particular promising. One learns with a Hebbian learning variant only for a short period after a data sample is presented and the distribution converges towards the data distribution [75]. Another approach replaces a data sample by a sample of the model distribution, which has only been nudged towards a real data input, so the distribution in the positive phase only shifts towards, but does not have to reach the data distribution completely [74]. A combination of those might be interesting as well. These approaches show similarities to some STDP mechanism, but to the best of our knowledge have not been implemented in spiking networks.

These approaches might also help with the problem of cascaded learning of different layers. With the contrastive divergence algorithm a good estimate of the data and model distribution is needed and thus for arbitrary Boltzmann machines potentially infinite sampling has to be performed. This leads to only simple and shallow Boltzmann machine structures, which are used. Due to the breaking up of hard sampling phases, this allows a simple transition to arbitrary Boltzmann machines. Thus researching joint layer learning algorithms shows great promise as well.

Another constraint are the synchronized weights. On going research points to the idea that the synchronization might not be so important. Lillicrap et al. showed that a linear system was still able to be trained with using backpropagation, even if the feedback weights were chosen randomly and kept constant [58]. In addition, research on autoencoders showed another kind of weight synchronization, where even without tied weight autoencoders often end up with symmetric weights [84]. It would be interesting to see how such mechanisms perform in spiking neural networks and if they can be transferred to local learning with STDP.

Eventually, it would be interesting to see if a biological plausible system integrating the relaxed phases without shared weights could be realized and simulated in real time.

A. Appendix

A.1. Neuron Parameters

Table A.1.: Current-based LIF neuron parameters used for the converted CNN.

Resting potential	-65 mV
Membrane capacity	1.0 nF
Membrane time constant	20.0 ms
Refractory period	1.0 ms
Offset current	0.0 nA
Reset potential	-65.0 mV
Spike threshold	-50.0 mV

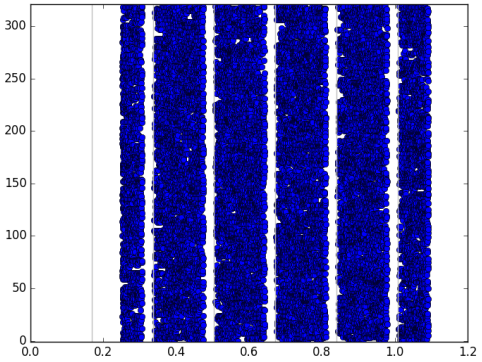
Table A.2.: Current-based LIF neuron parameters used for the converted DBN.

Resting potential	-65 mV
Membrane capacity	1.0 nF
Membrane time constant	1.0 ms
Refractory period	10.0 ms
Offset current	1.0 nA
Reset potential	-53.0 mV
Spike threshold	-52.0 mV

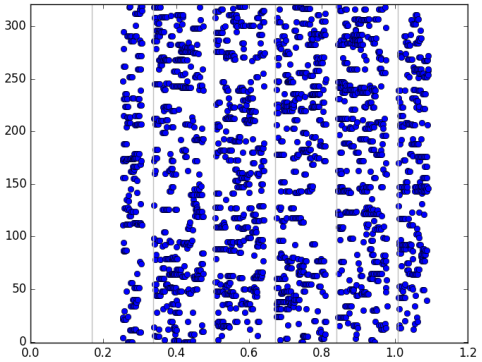
Table A.3.: Conductance-based LIF neuron parameters used for the converted DBN.

Resting potential	-65 mV
Membrane capacity	1.0 nF
Membrane time constant	20.0 ms
Refractory period	10.0 ms
Offset current	1.0 nA
Reset potential	-53.0 mV
Spike threshold	-52.0 mV
Inhibitory reversal potential	90.0 mV
Excitatory reversal potential	-0.0 mV

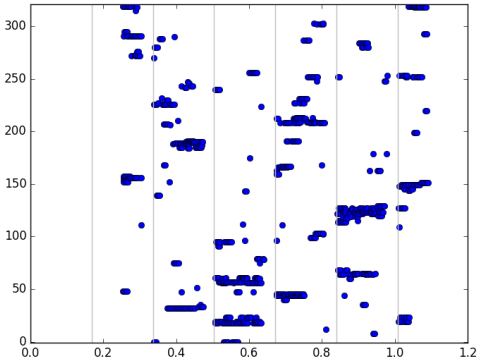
A.2. Lateral Inhibition



(a) Spikes without lateral inhibition.



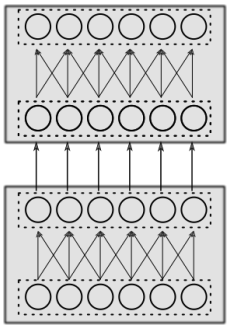
(b) Spikes with small lateral inhibition.

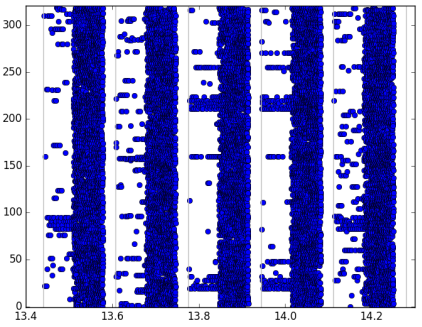


(c) Spikes with big lateral inhibition.

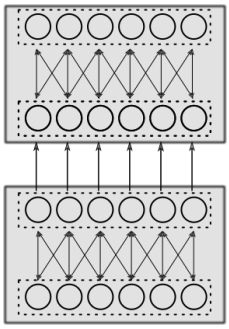
Figure A.1.: Spikes in a hidden layer of a spiking RBM with lateral inhibition. As the lateral inhibition increases, the activity becomes more sparse. In the case with big inhibitory weights, there are a few neurons dominating the activity allowing only a few different states of the network and thus poor mode switching in a training step. In a network with small inhibitory weights, there is sparse activity, but the network visits many different states in a simulation step and shows sufficient mode mixing.

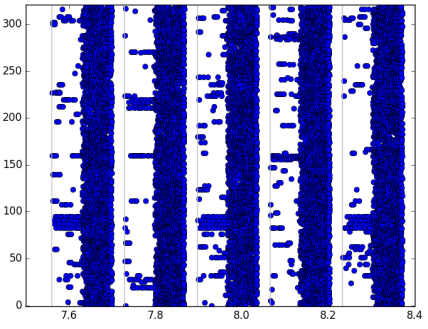
A.3. Spiking DBN Architectures



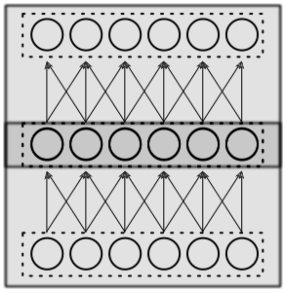


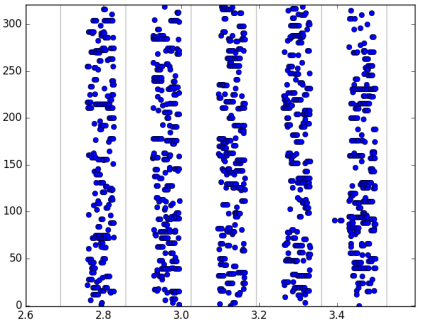
(a) DBN with distinct RBM layers.





(b) DBN with distinct RBM layers with bidirectional weights.





(c) DBN with RBM directly stacked to each other.

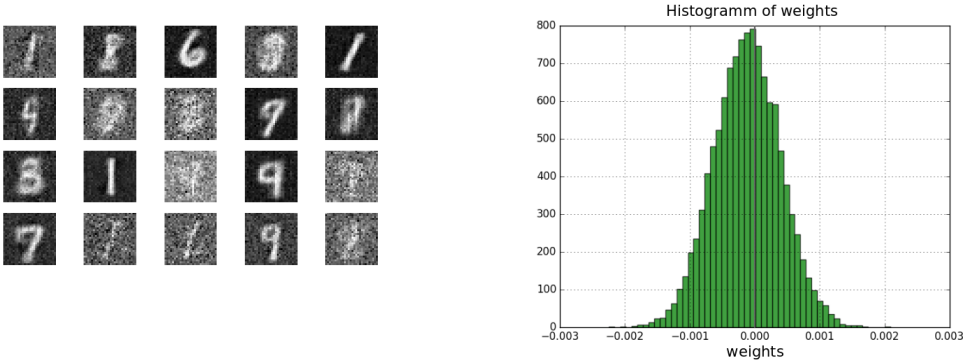
Figure A.2.: Activity in the visible layer of the top RBM in a DBNs with different architectures. The first two architectures model separate RBM layers which are stacked by one-on-one forward connections from the hidden layer of the bottom RBM to the visible layer of the top RBM, while in the last architecture the RBMs are directly stacked, i.e. the hidden layer of the bottom RBM is the visible layer of the top RBM. While in the top two architectures, there are hardly any differences and the input is modelled correctly, due to top down influences the output of the bottom RBM gets distorted by activity in the hidden layer of the top RBM.

A.4. Synchronous Weight Updates

For this thesis two different approaches were implemented.

The first approach, as presented in Chapter 4.3.1 and 6.2.3, synchronizes the weights at a discrete time step after a training sample is presented and processed. This poses some similarities to the normal contrastive divergence algorithm since, due to discrete nature of artificial neural networks, the weights are updated and synchronized after the positive and negative phase.

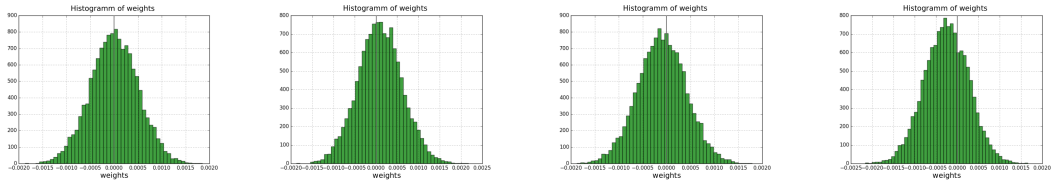
The second approach keeps the weights synchronized by applying the same STDP update to all synapses which share the same weight at the same time. Namely, if one synapses is updated by a local STDP rule, the update is shared with all tied synapses. For the non-convolutional case i.e. only the weights of forward and backward synapses between the same neurons are shared, this leads to discriminative results, as shown in Figure A.3.



(a) (b)

Figure A.3.: Weights with shared STDP weight updates and without any convolution. The weights are visualized in (a) as the filter matrices and in (b) as a histogram.

In our case, the case with shared STDP updates with convolutions did not show any promising results. Although the learning rate was scaled accordingly, due to some self-facilitation and weight explosion, the weights become mostly negative as illustrated in Figure A.4.



(a) (b) (c) (d)

Figure A.4.: Weights development for eCD with synchronous weight updates. The weight distribution in the beginning is shown in (a). After the positive phase (b) the weights increased a lot, which leads to a high decrement during the negative (c). During further training this trend continues until most weights are negative (d).

As a result, due to the mostly negative weights, this leads to a "dying-out" of the spike activity which is shown in Figure A.5

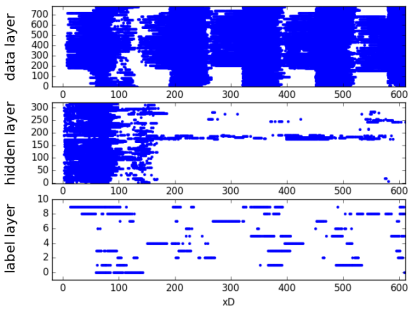


Figure A.5.: Spikes activity during convolutional eCD with synchronous weight updates. After a few training samples are presented the spiking activity decreases, due to the decrement of the synaptic weights.

Thus, no learning happens and the weights become not very discriminative (compare Figure A.6).

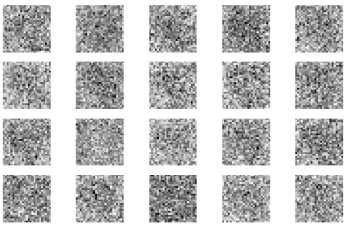


Figure A.6.: Learned weights for convolutional eCD with synchronous weight updates.

Increasing the learning rate to increase the weights leads to a few strong positive weights, while the majority of the weights still become negative leading to no good results either (Fig. A.7).

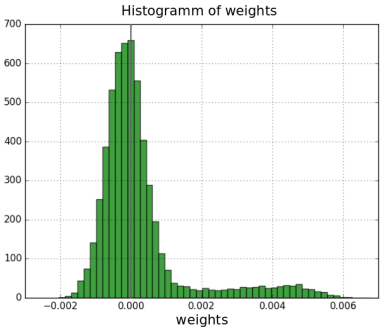


Figure A.7.: Weight histogram for convolutional eCD with synchronous weight updates and an increased learning rate during the positive phase. This leads to a division of the weights into a few very positive weights and many negative weights.

Further research on this topic could indicate whether a fine-tuned learning rate or a dynamic learning rate decay or adaptation could be a sound remedy to this problem.

B. List of Figures

C. List of Tables

D. Bibliography

[1] Boltzmann machines. http://gorayni.blogspot.de/2014/06/boltzmann-machines.html. Accessed: 2016-11-22.

[2] Convolution arithmetic. https://github.com/vdumoulin/conv_arithmetic. Accessed: 2016-11-22.

[3] Cs231n convolutional neural networks for visual recognition. http://cs231n.github.io/neural-networks-1/. Accessed: 2016-11-22.

[4] Deep belief networks - lisa wiki – pascallamblin. http://www.iro.umontreal.ca/~lisa/twiki/bin/view.cgi/Public/DeepBeliefNetworks. Accessed: 2016-11-22.

[5] Fgoml/perceptron at master github.com. https://github.com/cdipaolo/goml/tree/master/perceptron. Accessed: 2016-11-22.

[6] File:neuron hand-tuned.svg - wikipedia. http://en.wikipedia.org/wiki/File:Neuron_Hand-tuned.svg. Accessed: 2016-11-22.

[7] Mcmc sampling for dummies. http://twiecki.github.io/blog/2015/11/10/mcmc-sampling/. Accessed: 2016-11-22.

[8] Neural networks: Restricted boltzmann machine - contrastive divergence – hugo larochelle. http://info.usherbrooke.ca/hlarochelle/neural_networks/content.html. Accessed: 2016-11-22.

[9] Restricted boltzmann machines (rbm) - deeplearning 0.1 documentation. http://deeplearning.net/tutorial/rbm.html. Accessed: 2016-11-22.

[10] L. F. Abbott. Lapicque's introduction of the integrate-and-fire model neuron (1907). *Brain research bulletin*, 50(5):303–304, 1999.

[11] O. Abdel-Hamid, A.-R. Mohamed, H. Jiang, L. Deng, G. Penn, and D. Yu. Convolutional neural networks for speech recognition. *IEEE/ACM Transactions on audio, speech, and language processing*, 22(10):1533–1545, 2014.

[12] D. H. Ackley, G. E. Hinton, and T. J. Sejnowski. A learning algorithm for Boltzmann machines. *Cognitive science*, 9(1):147–169, 1985.

[13] R. Al-Rfou, G. Alain, A. Almahairi, C. Angermueller, D. Bahdanau, N. Ballas, F. Bastien, J. Bayer, A. Belikov, A. Belopolsky, Y. Bengio, A. Bergeron, J. Bergstra, V. Bisson,

J. Bleecher Snyder, N. Bouchard, N. Boulanger-Lewandowski, X. Bouthillier, A. de Brébisson, O. Breuleux, P.-L. Carrier, K. Cho, J. Chorowski, P. Christiano, T. Cooijmans, M.-A. Côté, M. Côté, A. Courville, Y. N. Dauphin, O. Delalleau, J. Demouth, G. Desjardins, S. Dieleman, L. Dinh, M. Ducoffe, V. Dumoulin, S. Ebrahimi Kahou, D. Erhan, Z. Fan, O. Firat, M. Germain, X. Glorot, I. Goodfellow, M. Graham, C. Gulcehre, P. Hamel, I. Harlouchet, J.-P. Heng, B. Hidasi, S. Honari, A. Jain, S. Jean, K. Jia, M. Korobov, V. Kulkarni, A. Lamb, P. Lamblin, E. Larsen, C. Laurent, S. Lee, S. Lefrancois, S. Lemieux, N. Léonard, Z. Lin, J. A. Livezey, C. Lorenz, J. Lowin, Q. Ma, P.-A. Manzagol, O. Mastropietro, R. T. McGibbon, R. Memisevic, B. van Merriënboer, V. Michalski, M. Mirza, A. Orlandi, C. Pal, R. Pascanu, M. Pezeshki, C. Raffel, D. Renshaw, M. Rocklin, A. Romero, M. Roth, P. Sadowski, J. Salvatier, F. Savard, J. Schlüter, J. Schulman, G. Schwartz, I. V. Serban, D. Serdyuk, S. Shabanian, É. Simon, S. Spieckermann, S. R. Subramanyam, J. Sygnowski, J. Tanguay, G. van Tulder, J. Turian, S. Urban, P. Vincent, F. Visin, H. de Vries, D. Warde-Farley, D. J. Webb, M. Willson, K. Xu, L. Xue, L. Yao, S. Zhang, and Y. Zhang. Theano: A Python framework for fast computation of mathematical expressions. *arXiv e-prints*, abs/1605.0, 2016.

[14] Y. Bengio and O. Delalleau. Justifying and generalizing contrastive divergence. *Neural Computation*, 21(6):1601–1621, 2009.

[15] Y. Bengio, D.-H. Lee, J. Bornschein, and Z. Lin. Towards biologically plausible deep learning. *arXiv preprint arXiv:1502.04156*, 2015.

[16] C. M. Bishop. *Pattern Recognition and Machine Learning*. Information science and statistics. Springer, 2013.

[17] K. A. Buchanan and J. R. Mellor. The activity requirements for spike timing-dependent plasticity in the hippocampus. *Frontiers in Synaptic Neuroscience*, 2(JUN):1–5, 2010.

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

[19] J. H. Byrne and N. Dafny. *Neuroscience Online: An Electronic Textbook for the Neurosciences*. Department of Neurobiology and Anatomy, The University of Texas Medical School at Houston (UTHealth), 1997.

[20] Y. Cao, Y. Chen, and D. Khosla. Spiking Deep Convolutional Neural Networks for Energy-Efficient Object Recognition. *International Journal of Computer Vision*, 113(1):54–66, nov 2015.

[21] C. Chen, A. Seff, A. Kornhauser, and J. Xiao. Deepdriving: Learning affordance for direct perception in autonomous driving. In *Proceedings of the IEEE International Conference on Computer Vision*, pages 2722–2730, 2015.

[22] A. Davison, D. Brüderle, J. Eppler, J. Kremkow, E. Muller, D. Pecevski, L. Perrinet, and P. Yger. PyNN: a common interface for neuronal network simulators. *Frontiers in Neuroinformatics*, 2:11, 2009.

[23] G. Desjardins and Y. Bengio. Empirical evaluation of convolutional RBMs for vision. *DIRO, Universit{é} de Montr{é}al*, pages 1–13, 2008.

[24] P. U. Diehl, D. Neil, J. Binas, M. Cook, S. C. Liu, and M. Pfeiffer. Fast-classifying, high-accuracy spiking deep networks through weight and threshold balancing. *Proceedings of the International Joint Conference on Neural Networks*, 2015-Septe, 2015.

[25] H. Donat. Towards Grasping with Spiking Neural Networks for an Anthropomorphic Robot Hand.

[26] F. Faltin and R. Kenett. Bayesian Networks. *Encyclopedia of Statistics in Quality & Reliability*, 1(1):4, 2007.

[27] A. Fischer and C. Igel. Training Restricted Boltzmann Machines: An Introduction. pages 25–39, 2014.

[28] W. Gerstner, W. M. Kistler, R. Naud, and L. Paninski. *Neuronal Dynamics: From Single Neurons to Networks and Models of Cognition*. Cambridge University Press, 2014.

[29] M.-O. Gewaltig and M. Diesmann. NEST (NEural Simulation Tool). *Scholarpedia*, 2(4):1430, 2007.

[30] Z. Ghahramani. Graphical models : parameter learning. *Handbook of Brain Theory and Neural Networks*, (2):486–490, 2002.

[31] A. Giusti, J. Guzzi, D. C. Cire\csan, F.-L. He, J. P. Rodr\'\iguez, F. Fontana, M. Faessler, C. Forster, J. Schmidhuber, G. Di Caro, and Others. A machine learning approach to visual perception of forest trails for mobile robots. *IEEE Robotics and Automation Letters*, 1(2):661–667, 2016.

[32] I. Goodfellow, Y. Bengio, and A. Courville. Deep Learning. 2016.

[33] D. Goodman and R. Brette. Brian: a simulator for spiking neural networks in Python. *Frontiers in Neuroinformatics*, 2:5, 2008.

[34] T. L. Griffiths, C. Kemp, and J. B. Tenenbaum. Bayesian models of cognition. 2008.

[35] D. Hebb. The organization of behavior, 1968.

[36] D. Heeger. Poisson Model of Spike Generation. *Handout*, pages 1–13, 2000.

[37] G. Hinton. A practical guide to training restricted Boltzmann machines. *Momentum*, 9(1):926, 2010.

[38] G. E. Hinton. Training products of experts by minimizing contrastive divergence. *Neural computation*, 14(8):1771–1800, 2002.

[39] G. E. Hinton. Deep belief networks. *Scholarpedia*, 4(5):5947, 2009.

[40] G. E. Hinton, P. Dayan, B. J. Frey, and R. M. Neal. The" wake-sleep" algorithm for unsupervised neural networks. *Science*, 268(5214):1158, 1995.

[41] G. E. Hinton, S. Osindero, and Y.-W. Teh. A fast learning algorithm for deep belief nets. *Neural computation*, 18(7):1527–1554, 2006.

[42] A. L. Hodgkin and A. F. Huxley. A quantitative description of membrane current and its application to conduction and excitation in nerve, aug 1952.

[43] J. J. Hopfield. Neural networks and physical systems with emergent collective computational abilities. *Proceedings of the national academy of sciences*, 79(8):2554–2558, 1982.

[44] D. H. Hubel and T. N. Wiesel. Receptive fields of single neurones in the cat's striate cortex. *The Journal of Physiology*, 148:574–591, 1959.

[45] X. Jin, S. B. Furber, and J. V. Woods. Efficient modelling of spiking neural networks on a scalable chip multiprocessor. In *2008 IEEE International Joint Conference on Neural Networks (IEEE World Congress on Computational Intelligence)*, pages 2812–2819. IEEE, 2008.

[46] A. Karpathy, G. Toderici, S. Shetty, T. Leung, R. Sukthankar, and L. Fei-Fei. Large-scale video classification with convolutional neural networks. In *Proceedings of the IEEE conference on Computer Vision and Pattern Recognition*, pages 1725–1732, 2014.

[47] S. R. Kheradpisheh, M. Ganjtabesh, S. J. Thorpe, and T. Masquelier. Stdp-based spiking deep neural networks for object recognition. *CoRR*, abs/1611.01421, 2016.

[48] P. D. King, J. Zylberberg, and M. R. DeWeese. Inhibitory interneurons decorrelate excitatory cells to drive sparse code formation in a spiking model of V1. *The Journal of neuroscience : the official journal of the Society for Neuroscience*, 33(13):5475–85, 2013.

[49] A. Krizhevsky, I. Sutskever, and G. E. Hinton. ImageNet Classification with Deep Convolutional Neural Networks. In F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems 25*, pages 1097–1105. Curran Associates, Inc., 2012.

[50] Y. LeCun. 5 Years From Now, Everyone Will Learn Their Features.

[51] Y. LeCun, Y. Bengio, and G. Hinton. Deep learning. *Nature*, 521(7553):436–444, may 2015.

[52] Y. LeCun, B. Boser, J. S. Denker, D. Henderson, R. E. Howard, W. Hubbard, and L. D. Jackel. Backpropagation applied to handwritten zip code recognition. *Neural computation*, 1(4):541–551, 1989.

[53] Y. LeCun and C. Cortes. MNIST handwritten digit database. 2010.

[54] Y. LeCun, K. Kavukcuoglu, C. Farabet, et al. Convolutional networks and applications in vision.

[55] H. Lee, R. Grosse, R. Ranganath, and A. Y. Ng. Convolutional deep belief networks for scalable unsupervised learning of hierarchical representations. In *Proceedings of the 26th annual international conference on machine learning*, pages 609–616. ACM, 2009.

[56] T. S. Lee and D. Mumford. Hierarchical Bayesian inference in the visual cortex. *JOSA A*, 20(7):1434–1448, 2003.

[57] S. Levine, P. Pastor, A. Krizhevsky, and D. Quillen. Learning Hand-Eye Coordination for Robotic Grasping with Deep Learning and Large-Scale Data Collection. *arXiv preprint arXiv:1603.02199*, 2016.

[58] T. P. Lillicrap, D. Cownden, D. B. Tweed, and C. J. Akerman. Random feedback weights support learning in deep neural networks. *arXiv:1411.0247 [cs, q-bio]*, pages 1–27, 2014.

[59] W. Maass. Networks of spiking neurons: the third generation of neural network models. *Neural networks*, 10(9):1659–1671, 1997.

[60] H. Markram, W. Gerstner, and P. J. Sjöström. Spike-Timing-Dependent Plasticity: A Comprehensive Overview, 2012.

[61] B. Meftah, O. Lézoray, S. Chaturvedi, A. A. Khurshid, and A. Benyettou. Image processing with spiking neuron networks. *Studies in Computational Intelligence*, 427:525–544, 2013.

[62] K. P. Murphy. *Machine Learning: A Probabilistic Perspective*. Adaptive computation and machine learning. MIT Press, 2012.

[63] V. Nair and G. E. Hinton. Rectified Linear Units Improve Restricted Boltzmann Machines. *Proceedings of the 27th International Conference on Machine Learning*, (3):807–814, 2010.

[64] E. Neftci, S. Das, B. Pedroni, K. Kreutz-delgado, G. Cauwenberghs, L. Jolla, L. Jolla, and L. Jolla. Event-Driven Contrastive Divergence for Spiking Neuromorphic Systems. 2013.

[65] D. Neil. Online Learning in Event-based Restricted Boltzmann Machines. (October), 2013.

[66] M. Norouzi, M. Ranjbar, and G. Mori. Stacks of convolutional restricted boltzmann machines for shift-invariant feature learning. In *Computer Vision and Pattern Recognition, 2009. CVPR 2009. IEEE Conference on*, pages 2735–2742. IEEE, 2009.

[67] Norouzi M. Convolutional restricted Boltzmann machines for feature learning. *School of Computing Science-Simon Fraser University*, pages 2735–2742, 2009.

[68] P. O'Connor and M. Pfeiffer. Real-time classification and sensor fusion with a spiking deep belief network. 7(October):1–13, 2013.

[69] M. A. Petrovici. *Form Versus Function: Theory and Models for Neuronal Substrates*. 2016.

[70] T. Pfeil, A. Grübl, S. Jeltsch, E. Müller, P. Müller, M. A. Petrovici, M. Schmuker, D. Brüderle, J. Schemmel, and K. Meier. Six networks on a universal neuromorphic computing substrate. *Frontiers in Neuroscience*, 7:11, 2013.

[71] F. Rosenblatt. The perceptron: a probabilistic model for information storage and organization in the brain. *Psychological review*, 65(6):386, 1958.

[72] D. E. Rumelhart, G. E. Hinton, and R. J. Williams. Learning internal representations by error propagation. Technical report, DTIC Document, 1985.

[73] R. Salakhutdinov and G. E. Hinton. Deep Boltzmann Machines.

[74] B. Scellier and Y. Bengio. Equilibrium Propagation: Bridging the Gap Between Energy-Based Models and Backpropagation Benjamin. 2016.

[75] B. Scellier and Y. Bengio. Towards a Biologically Plausible Backprop. *CoRR*, abs/1602.05179, 2016.

[76] T. Serrano-Gotarredona and B. Linares-Barranco. A 128 128 1.5 contrast sensitivity 0.9 FPN 3 mirco s latency 4 mW asynchronous frame-free dynamic vision sensor using transimpedance preamplifiers. *IEEE Journal of Solid-State Circuits*, 48(3):827–838, 2013.

[77] A. M. Sillito, J. Cudeiro, and H. E. Jones. Always returning: feedback and sensory processing in visual cortex and thalamus. *Trends in neurosciences*, 29(6):307–316, 2006.

[78] K. Simonyan and A. Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.

[79] P. J. Sjostrom and W. Gerstner. Spike-timing-dependent plasticity. *Scholarpedia*, 2(2):1362, 2010.

[80] P. Smolensky. Information processing in dynamical systems: Foundations of harmony theory. Technical report, DTIC Document, 1986.

[81] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich. Going deeper with convolutions. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 1–9, 2015.

[82] Y. Teh and G. E. Hinton. Rate-coded Restricted Boltzmann Machines for Face Recognition. *ACM Transactions on Graphics*, 24:1071, 2005.

[83] T. Tieleman. Training restricted Boltzmann machines using approximations to the likelihood gradient. In *Proceedings of the 25th international conference on Machine learning*, pages 1064–1071. ACM, 2008.

[84] P. Vincent, H. Larochelle, I. Lajoie, Y. Bengio, and P.-A. Manzagol. Stacked denoising autoencoders: Learning useful representations in a deep network with a local denoising criterion. *Journal of Machine Learning Research*, 11(Dec):3371–3408, 2010.

[85] O. Woodford. Notes on Contrastive Divergence. *Notes*, (Cd):1–3, 2002.

[86] T. Yang and M. N. Shadlen. Probabilistic reasoning by neurons. *Nature*, 447(7148):1075–1080, 2007.

[87] Y. Zhou. Structure Learning of Probabilistic Graphical Models : A Comprehensive Survey. *arXiv: 1111.6925*, 2007.

[88] M. Zorzi, A. Testolin, and I. Stoianov. Modeling language and cognition with deep unsupervised learning: a tutorial overview. *Frontiers in Psychology*, 4:515, 2013.