Karlsruhe Institute of Technology

Title of thesis
Second title line

Master Thesis of

Forename Surname

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

Reviewer:	Prof. Dr.–Ing. J. M. Zöllner
Second reviewer:	Prof. Dr.–Ing. R. Dillmann
Advisor:	Dipl.–Inform. Max Mustermann

Research Period: XX. Monat 20XX – XX. Monat 2016

Title of thesis - Second title line

by
Forename Surname

Master thesis
in Monat 2016

FZI

Master thesis, FZI
Department of Computer Science, 2016
Gutachter: Prof. Dr.–Ing. J. M. Zöllner, Prof. Dr.–Ing. R. Dillmann

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, *Forename Surname*
in Monat 2016

Contents

1. Introduction

1.1. Motivation

In 2012 by winning the Imagenet Large Scale Visual Recognition Challenge 2012 convolutional neural network gained a big rise in popularity. Now they are becoming popular for their powerful abstraction mechanism in the fields of image and video classification and description and speech recognition. This can be contributed to compositional structure in which the world can be perceived and to their ability to extract high level features on spatial and/or temporal conditioned data.

Generating discriminative high level features extractors allows the system to dynamically adapt to the input data and work on various kinds of data. In addition the features extractors do not need to have any semantic representation and can be more complex to the manually build feature extractors. This also removes the labor-intensive and time consuming task of manually building feature extractors. Consequently they recently got adapted to solve robotic problems like grasp planning, drone navigation and autonomous driving.

One precursor of those 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.

Compared to classical CNNs, DBNs allow recurrent connections and are trained in an unsupervised manner and do not need labeled data. They have been described as “probably the most biologically plausible learning algorithm for deep architectures we currently know”. DBNs can be used as 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 to state of the art results and made the system more similar to the primate visual cortex than a standard RBM.

All those approaches use scalar values between neurons at discrete time slices to propagate information. This proposes some difficulties and is not biologically plausible, since biological neuron interleave linear and nonlinear operations, they communicate by stochastic binary values and are not synchronized. Spiking neural networks (SNNs), designed to simulate the communication between neurons with action potentials/ spikes, work in continuous time by design and do not suffer from the aforementioned limitations.

1.2. Problem Statement

To our best knowledge, up to today, there exists no system which utilizes the benefits of all those approaches. The main objective of this thesis is to realize such a spiking network, which integrates convolution and can be easily trained utilizing the RBM learning algorithm, to extract high level features. Two approaches are described. The first approach trains convolutional RBMs on discretized input data, to build up a DBN which is then converted to a SNN. The next approach directly works on continuous (event-driven) input-data and realizes a STDP learning algorithm with shared weights to train spiking convolutinal RBMs directly. Both approaches will learn to extract high level features, which can be further used to classify an object or directly generate a grasp id.

1.3. The Human Brain Project

Heiko:

This thesis is under the scope of the research of the SP10 (sub-project 10) of the Human Brain Project. The Human Brain Project is an European Commission Future and Emerging Technologies Flagship and a large ten-year research project which aims to create a collaborative research infrastructure across national borders to progress the knowledge in neuroscience, computational neuroscience, medicine, scientific computation, and robotics. In the project over 120 institution from across Europe collaborate in 12 sub-projects.

The sub-project 10 of the Human Brain Project develops the Neurorobotics Platform which permits researchers to simulate robotic experiments with regard to neurorobotics. Apart from the platform, research is focused on the development of applications in robotics based on insights from neuroscience. One focus of the research group at the FZI Karlsruhe is the development of computational models for neurobiological inspired robotic grasping.

1.4. Overview

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

Chapter 2 introduces sine background information, which is used in chapter 3 to describe state-of-the-art research used in this thesis. Chapter 4 will describe the different approaches to build such a spiking convolutional deep belief networks. In chapter 5 the different implementation steps and the architecture will be described. Chapter 6 outlines and compares the performance of the networks. Chatper 7 will conclude the gathered insight of this thesis, state its limitations and give suggestions for further improvements and research.

2. Background

2.1. Probabilistic Graphical Models

Probabilistic graphical models (PGMs) or structured probabilistic models can be used to describe, formalize and model neural learning architectures.

Having captured the basic probabilistic structure of data makes a lot of data-related task feasible, such as:

- Density estimation: Assigning a data sample x an estimate of its true probability density $p(x)$ (e.g. indicating unusual data).
- Denoising: Given a perturbed or noisy input $\widetilde{x}$, estimating the original data sample x
- Missing value imputation: Given a partial input sample of x, it returns the most probable completion of the mission values to return x (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).

Each node in the graph represents a RV and each edge between two RVs a direct interaction.

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.

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

2.1.1. Bayesian network

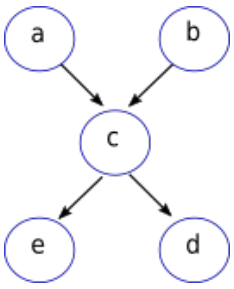
Bayesian networks or belief networks are directed acyclic graphs, in which random variables are represented by nodes and their causal dependencies are represented by (directed) edges/connections.

It poses a way to depict a probability distribution factorized by repeatedly applying the Bayes rule.

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

$p(x_j|p_i)$.

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



(a) A Bayesian with 5 RVs.

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

(b) Tabular conditional probability $p(c|a,b)$.

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

(c) Conditional probability $p(c|a,b)$ given by a function over a,b.

Figure 2.1.: A sample Bayesian network with 5 nodes. (a) In the network RV c is directly depended on a, b and thus c is the child of its parents a, b. The RVs d, e are the children of c. Since there is a path from a to e, a is an ancestor of its descended e. (b) The conditional probability of $p(c|a,b)$ 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.

As well as 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.

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

$p(x_i|x_{j\setminus i}) = p(x_i|parents(x_i))$, where $x_{j\setminus i} = \mathbf{z} \setminus x_i$.

A Bayesian network contains a simple conditional independence assertion, i.e. each variable is given its parent in-depended on all non descendants:

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

where $z = (x_1, ..., x_n)$ is the state of the model.

This reduction provides (i.a from a computational perspective) an efficient way for learning/ parameter estimation and inference.

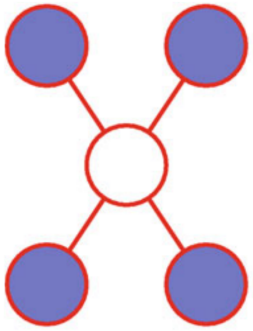
Given some observed variables, the evidences, the net can be used to compute the posterior probabilities and infer the most likely state of the hidden/ unobserved variables.

This process of computing the posterior distributions given the evidences is called probabilistic interference.

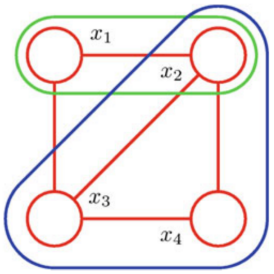
Thus a Bayesian network can be seen as a tool to simplify and apply the Bayes rule to complex problems.

2.1.2. Markov Random Field

In contrast to Bayesian networks, Markov random fields are undirected graphical models, in which random variables are represented by nodes and edges/ connections indicate conditional dependen-



(a) A Markov network with with 5 nodes.



(b) Cliques in a Markov network

Figure 2.2.: (a) A Markov network with 5 nodes. The white node is depended on all connected nodes (blue nodes). Given its the blue nodes the white node is independent on any other node in the network. (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.

cies.

If two nodes x_i and x_j are connected by a edge, they are called each others neighbours.

Given all of their neighbours, two nodes x_k and x_m are in-depended on each other.

Thus an MFR can be seen as a model of the joint probabilities of RVs.

This allows the probability distribution to be factorized into fully connected partial graphs, cliques:

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

where Z is the normalization factor, so that $\sum_z p(\mathbf{z}) = 1$ ($Z = \sum_z \prod \phi(z_k)$) and $\phi(z_k)$ is the partition function of the clique/ partial graph z_k.

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 modelled problem, smaller clique sizes may be useful as well, e.g. Boltzmann machine are modelled with cliques of size two.

The potential of each clique has always to be greater or equal to zero, but a common, computational beneficial choice is to represent each state with is positive value.

2.1.3. Energy-Based Models

One convenient way to model the potentials is by use an energy function $E(z_k)$:

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

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 complete graph can be decomposed into the sum of the energy all cliques.

Because the exponential function is always positive ($exp(z) > 0$), the probability for each state is guaranteed to be greater than zero.

This offers the freedom of choice of an arbitrary energy function v$E(z_k)$, which can simplify optimization.

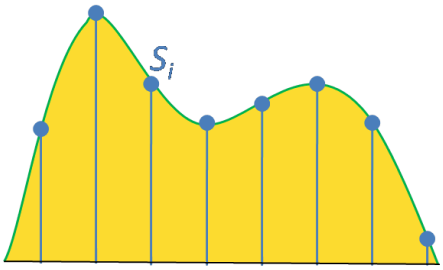


Figure 2.3.: Sampling at discrete points S_i in an simple distribution. The samples S_i approximate the true density function. As more samples are drawn, the approximation will represent the function more exact.

A probability distribution given by an EBM is also called Boltzmann distribution, due to the similarity to particle physics (and this is also where the Boltzmann machine get its name as we will see in chapter 2.x).

2.1.4. Sampling

Often calculating the exact probability distribution or marginal distributions is a computational not efficient or untraceable problem. Sampling is an 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.

Sampling can be described as the selection of a subset of individuals from a distribution to estimate properties of the complete distribution.

This is especially useful for graphical models, since they often facilitate the task of drawing samples from a distribution.

Ancestral Sampling For directed graphical models there is a simple and efficient procedure called ancestral sampling, which can produce samples from the joint distribution represented by the model. The basic idea is to sort the variables x_n in the graph into a topological ordering, so that for all i and j, j is greater than i if x_i is a parent 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. 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 a (infinite) time z is repeatedly randomly updated using the by the model given transition distribution $T(z',z)$. Eventually z becomes a fair sample from $p(z)$, 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 burn in/ mixing time, which allows the Markov chain to reach its equilibrium, and than 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 is no guaranty, that the Markov chain has settled in the given timespan. Another problem with the second approach is, that 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. The basic idea in Gibbs sampling to perform the transition from one state to another in accordance with $T(z',z)$ is, to select a single variable x_i and sample it 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 will examine the foundations of natural and artificial neural networks (NNs). At first we will look at the mechanism behind natural neural networks in the brain. After that different artificial models will be described, starting with models working at discrete time steps and then describing models, which work in continuous time. To distinguish those, throughout this thesis we will refer to models working at discrete time steps as *artificial neural networks* (ANNs) and models working in continuous time as *spiking neural networks* (SNNs).

2.2.1. Natural

Brain

The human brain is the main organ of the human central nervous system and with a weigh of about 1.2 - 1.4 kg (2% of the total body weight) and a power consumption around 20 Watts (20% of the total human power consumption), it is thought to be mainly responsible, for what's commonly called human intelligence.

It is to contribute for tasks as learning, memory, self-control, planing, reasoning, abstract thought, motor control, vision and language.

All these tasks can be attributed to different specialized regions in the brain.

Even so different regions perform different operations, the basic building blocks of the brain are astonishingly uniform.

It is mainly composed of neurons, glial cells and blood vessels.

While the neurons are the main computational unit, the other constituents are required for structural stabilization and energy support.

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

A single neuron can perform only quite simple operations but, the human brains contains 10^{12} neurons and each single neuron is interconnected with 10^4 other neurons.

The resulting complex neural network with roughly 10^{15} neural connections are the main component for human intelligence and enable complex task as scene understanding, language processing and motion planing.

Neuron

Neurons are the main information processing unit in the brain. Information is primarily processed through chemical and electrical signals.

A neuron, which is a specialized kind of (biological) cell, is separated for 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 refereed to as the membrane potential.

Ion channels, often voltage gated, in the cell membrane allow positive 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(? cell).

The ion flow is primarily determined by the charge difference at the membrane (electro static force ? / reversal potential), the ion concentration differences (diffusion force/nernst 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 a resting/equilibrium potential of -65 mV.

Each singe neuron can be divided into three functional distinct parts: the dendrites, the soma and the synapses.

The dendrites are thin, complexly branching structures emerging the the cell body/ 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 signal.

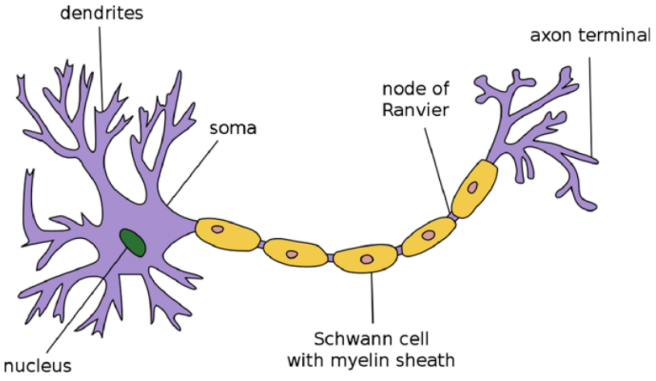


Figure 2.5.: A schematic view of a natural neuron. Other neurons are pre-synaptic connected via the dendrites. The signals are then forwarded and accumulated in the soma and from there on via the in myelin sheath cover axon to the axon terminal and the out going synapses.

The cell body or soma, which encompasses the nucleus, accumulates all signals/ polarizations from the dendrites and if the accumulation exceeds are "pseudo" threshold, usually around -55 mV, certain ion channels become more active, which allows an influx of positively charged ion and an action potential/spike emerges.

The spike is propagated across the axon and distributed to other subsequent (post-synaptic) neurons via the synapses, where they in turn 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 electrical connections/synapses and chemical which chemical compounds.

Probably due to their more diverse form of signal exchange, almost exclusively all synapses found in the brain have a chemical nature.

Learning

The exact algorithm behind learning in the brain is still mostly unknown. Neural plasticity is ofter considered to one of the primary mechanisms 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 between single neurons, which is often associated with human learning and memory (in contrast to learning in brain, which is the general mechanism of altering the information processing in the brain, human learning encompasses 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 in **short-term plasticity** which acts on a time scale one milliseconds to a minutes and **long-term plasticity**, lasting minutes or more.

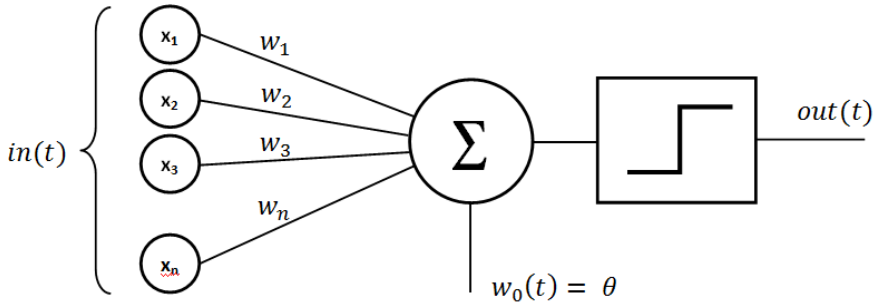


Figure 2.6.: Structure of a perceptron. The input $in(t)$ is set at the input variables x_i and the multiplied with the corresponding synaptic weight w_i and accumulated. In addition a threshold offset θ is added. On the sum the step-function is applied i.e. the output $out(t)$ is 1 if the sum is greater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

"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."

Meaning, 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/ A and B become associated.

This can be mathematically generalized as:

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

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 activitiy of A and B, respectively. F is a function describing the weight change conditioned on the neural activites.

2.2.2. Artificial neural networks

One of the first artificial neural networks were networks which work/compute at discrete time slices. While it simplifies the neuron models a lot, it makes them exceptionally easy to handle on most processors, which also work at a discrete tact rate.

Perceptron

The perceptron, also called Rosenblatt Perceptron, was invented in the late 1950s by Frank Rosenblatt. 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.

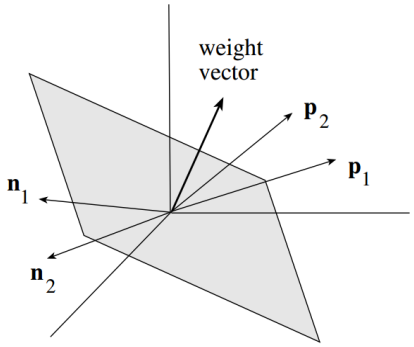


Figure 2.7.: The discrimination function of a perceptron. The discrimination function has the shape of a linear hyper plane in data space and it defined by the synaptic weight-vector $\mathbf{w}$. It divides the data space and thus the data samples into two subspaces, the positive space $\mathbf{x}^\intercal\mathbf{w}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

Be $\mathbf{x}$ the input of dimension n and $\mathbf{w}$ the n-dim vector describing the synaptic weights, then each x_i is multiplied by it's synaptic weight w_i and then accumulated/summed up. $\sum x_i w_i = \mathbf{x}^\intercal\mathbf{w}$.

If the sum exceed a threshold b, the perceptron "fires" and the output y is set to 1 and 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 append the bias b to the weight vector $\mathbf{w}' = (b, w_1, ..., w_n)$ and to extend the input dimension by a constant one $\mathbf{x}' = (1, x_1, ..., x_n)$. This allows us to handle the bias as a simple synaptic weight, and thus we will cease to model it explicitly and further on.

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

Using the heaviside step function θ, the perceptron calculation rule can be rewritten as

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

Decision Function This can be interpreted as a linear discriminating function, where $\mathbf{w}$ is a hyper plane in the data space diving it into two half spaces. 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. multi layers perceptrons combine simple ones function into more complex ones.

Perceptron Learning Be $\mathbf{X}$ a set of datapoints and $\mathbf{Y}$ their corresponding label 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:

$$w = w + \delta w$$

, where

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

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.

This can be performed until all data samples are correctly classified i.e. $\tilde{y} = y$ or a predetermined number of iterations have been completed.

Mutlilayer-Perceptron

While the perceptron models a single neuron, the multi-layer perceptron can be seen as an extension modelling neural networks and by doing so overcoming the perceptrons disadvantage to only discriminate linearly.

Model architecture A MLP consists of multiply consecutive layers.

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 $W \in \mathbb{R}^{nxm}$ 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 can be calculated as:

$$\mathbf{y} = \varphi(x^\intercal W)$$

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 $w_i \in W = (w_1, ..., w_n)$.

By using the output of a previous layer as input for a next layer, serveral layers can be stacked up:

$$\mathbf{y}^2 = \varphi((\mathbf{y}^1)^\intercal W^2),$$

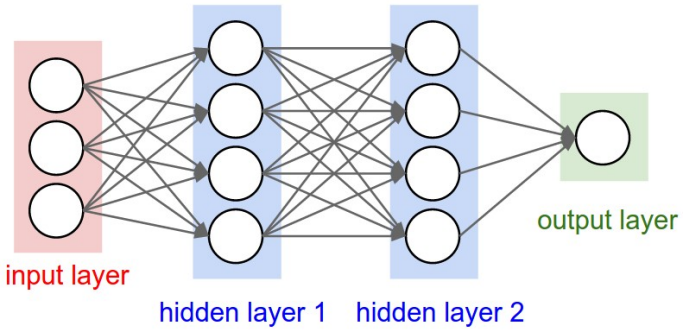


Figure 2.8.: A schematic multi layer perceptron with four layers.

where the superscript index represents the number of the layer.

Modeled like this, the output is always forwarded to the next layer, so there are no cycles in the information flow of the network. Such a network with only forward connections is called feedforward network (in contrast to recurrent networks such as Boltzmann machines, which allow information to be forwarded in cycles).

Activation functions The activation function φ can be basically arbitrarily chosen. There different activation functions, with different attributes, have been proposed, which also have proven to perform well on certain tasks:

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

Different activation functions describe the behaviour of the neurons on the input data. This is very similar to choosing different neuron models for spiking neural networks.

Error functions In machine learning, to validate the quality of the model an error function/cost function is used. The error function gives 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 depended and rather independent on the learning algorithm used.

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

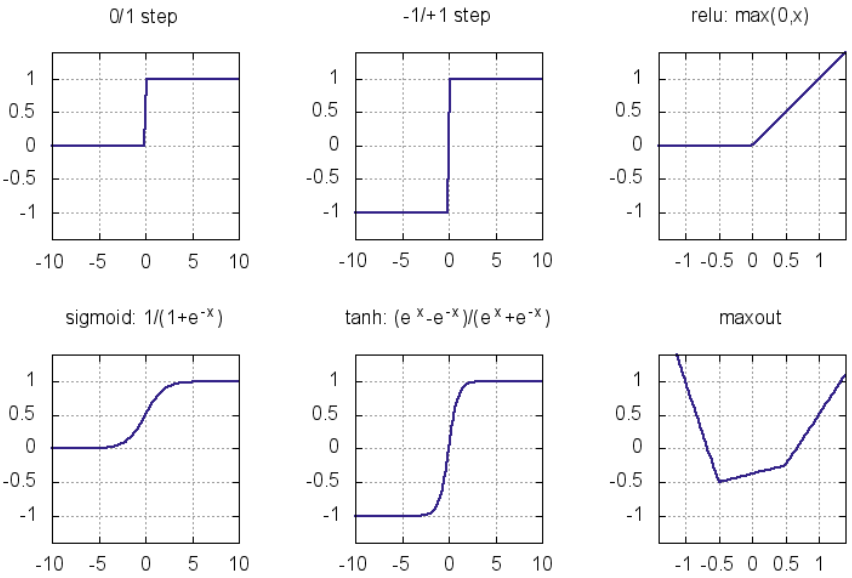


Figure 2.9.: The output of different activation functions plotted given the input.

- 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 changed. Thus the objective is to get parameters θ^* which form a (global) minimal point in the error function. Different optimization algorithms can be used to achieve this objective, but the most common class of algorithms use the gradient to determine the contributions of the weights to the error and thus reduce the error and reach a minimal point.

Gradient descent calculates the gradient for the parameters, and by following the negative gradient direction the weights are updated.

Since MLPs have a clearly defined structure, gradient descent update rule can be simplified, using the chain rule of calculus, to an iterative procedure, called backpropagation.

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 repetitive 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:

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

and

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

This just leaves the first factor $\frac{\partial E}{\partial y_j}$. It 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:

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

We define

$$\delta_l = \frac{\partial E}{\partial net_l} = \frac{\partial E}{\partial y_l}\frac{\partial y_l}{\partial net_l} = \begin{cases} E'(y_l)\varphi'(net_l), & \text{if } j \text{ is an output neuron} \\ (\sum_{k \in K} \delta_k w_{lk})\varphi'(net_l), & \text{if } j \text{ is not an output neuron.} \end{cases}$$

This all concludes to

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

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.

Convolutinal 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 constrains on the architecture of those nets, which makes them easier to train and allows greater generalization with fewer training samples. This is implemented by instead of having fully connected nets, having only partially connected layers with shared connection weights.

Convolution As the name suggests CNNs perform a convolution operation. The convolution operation c is defined as

$$c(t) = a * b = \int a(x) b(t-x) dx.$$

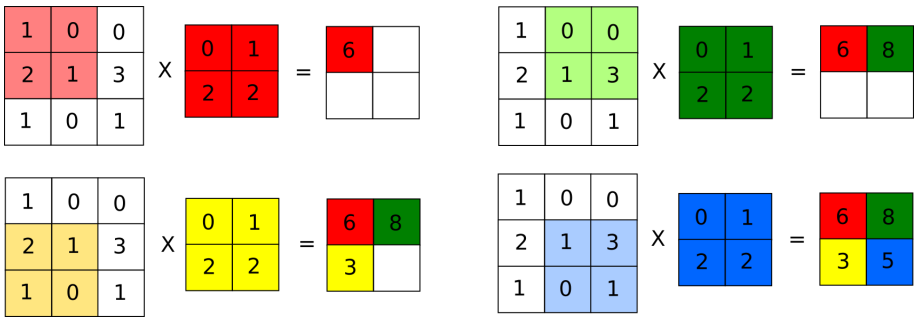


Figure 2.10.: Convolving or to be more exact a cross correlation of a 3×3 image matrix with a 2×2 kernel without stride and padding. The result is a 2×2 feature map.

In eq ... a is usually called the input and b the kernel of the convolution. The result $c(t)$ 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 only defined over the one-dimensional temporal dimension, it can be also 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 as well as the feature map $C(i,j)$ also spans 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.

Convolution Layers In neural networks convolution is applied in the so-called colvolution layer, where the convolution operation with a learnable kernel matrix K is applied. An input given as an 3D tensor Y be 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 (3 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

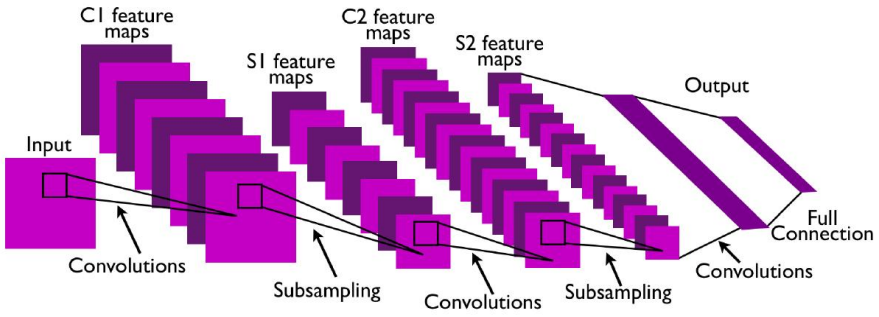


Figure 2.11.: Typical architecture of a convolutional neural network with two convolution-pooling stages.

matrix K_j at position (x,y) is then defined as :

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

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 positon (x,y).
In a typical layer the weights w_{ij}^{mpq} of all Kernel matrices K_j in all layers i are the free parameters that have to be learned. Another parameter which has to be determined is the number j of Kernel matrices at each layer i and therefore the output dimension of the layer.

Architecture The most common architectures for CNNs are build 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.

Training The training is performed applying the backprop algorithm to convolution:

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

By applying the chain rule this can be rewritten as

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

Another more intuitive update rule is given by defining the contribution to the error of an 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

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

which basically boils down to applying the sum of each individual weights of 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, they still lack some biological plausibility due to their pure forward nature of their connections. Hopfield nets try to overcome some of those issues by using recurrent connections and binary units.

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

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

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

The activation of a unit is similar to perceptron with a step activation function and depends only on its neighbour 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/ activity of neuron i.

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

- Asynchronous: One unit is updated at a time. The units are either chosen randomly or in a prefined 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, which for Hopfield nets the energy of a state is defined as

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

Properties By introducing recurrent connections, the network can be trained to store information. It's quite easy to see, with each (asynchronous) update step the energy is thus guaranteed to

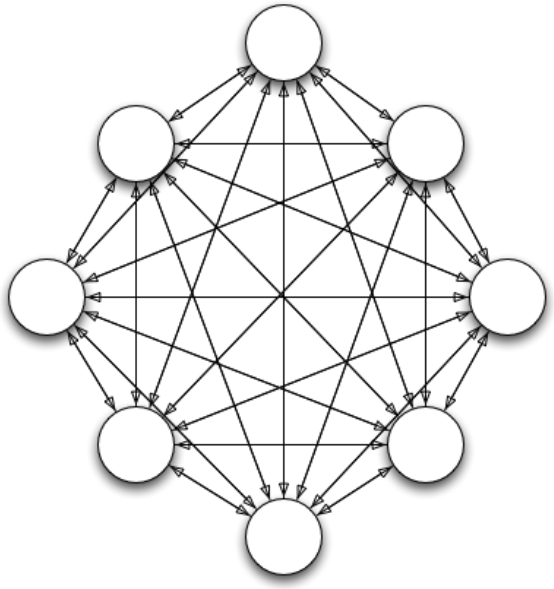


Figure 2.12.: A blueprint of a Hopfield nets with 8 binary units. The units are connected with symmetric undirected connections.

stay the same or lower in value. And thus 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 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 they only use information of neurons on either side of a connection:

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

Such Hopfields net can consequently perform pattern completion by reaching a low energy state, but it can also end in a spurious state, an attractor state/local energy minimum, which was bot 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.

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 weights $w_{ij} = w_{ji}$ without any self connections $w_i i = 0$, and an equivalent energy function:

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

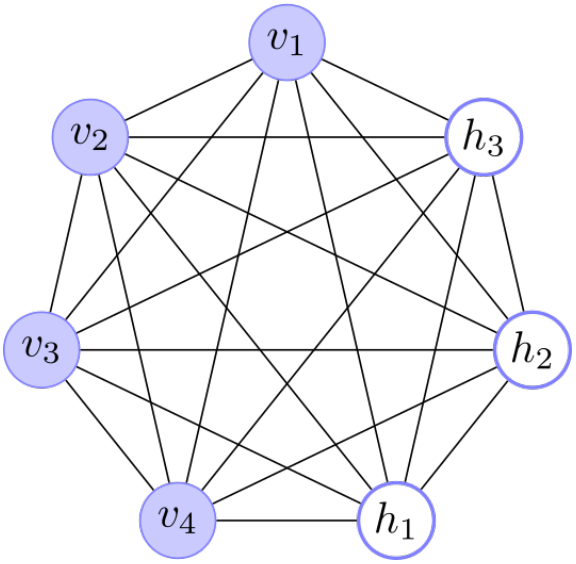


Figure 2.13.: A Boltzmann machine with 7 units. In contrast to a Hopfield nets, units are divided into visible and hidden/ unobserved units with stochastic activations.

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 $\mathbf{w}$.

A Boltzmann machine is an energy base model with the prob distribution defined by the Energy. Thus each state can be assigned a Energy which is directly indicates its probability.

A unit activates probabilistic 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 samping step from the distribution defined by E. Running a certain number of consecutive update steps probabilisticly drives the network to a low energy state. Given enough steps (possibly an infinite number) the network states will have reached a distribution, which does not change any more (even so the individual states might do, the relative number of time spend in on state becomes constant). Such a state is called equilibrium state (due to its similarity to particle physics).

In contrast to Hopfield nets where each unit of the network is represented by a 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/latent units. With hidden units a Boltzmann machine becomes a universal approximation probability mass functions over discrete variables.

Learning Rule The goal for training a Boltzmann machine is to get a probability distribution, which is similar to the distribution, which generated the training data, the data distribution. Thus the objective is to for the Boltzmann machine to assign a high probability to its training data (while assigning a low probability data not drawn from the data distribution). By using gradient descent

we will adapt the parameters/ weights of the Boltzmann machine in a directions which assigns trainings samples a high probability:

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

where

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

By using the logarithm of the by energy models defined probability function (@TODO ref) $\frac{\partial P(\mathbf{x})}{\partial w_{ij}}$ can be rewritten as

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

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

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

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 until an equilibrium distribution is reached, with either training data clamped or no data clamped to visible units.

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 the Boltzmann machine has to reach a equilibrium distribution. This can be quite computational expensive and there is no guarantee to tell when the equilibrium distribution is reached. Thus potentially infinite sampling steps have to be performed.

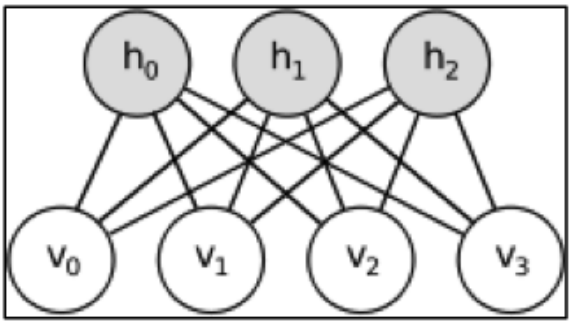


Figure 2.14.: A restricted Boltzmann machine is special kind of Boltzmann machine with no lateral connections in the hidden and visible layer. This eases sampling, since the visible are only dependent on the hidden units and the hidden units only on the visible units.

To evade the problem, a simple solution is to make the hidden units not depended on each other (and the visible not depended on each other), so all hidden units can be sampled independed and parallel of each other.

Thus a Boltzmann machine with two bipartite layers is called restricted Boltzmann machine (RBM).

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*.

CD-l 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.

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 downward and upward passes to get $\mathbf{v}_k$ and $\mathbf{h}_k$.
3. Update $w_{ij} = w_{ij} + ([s_i s_j]_0 - [s_i s_j]_k)$

Persistent CD , proposed by Tieleman, does not initialize the Boltzmann machine new each time a new sample is drawn, but uses the activations of previous runs instead. An explanation why sometimes shows better performance could be that the previous activations are already closer to an energetic minimum, so that fewer step are required to get an 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 layer.

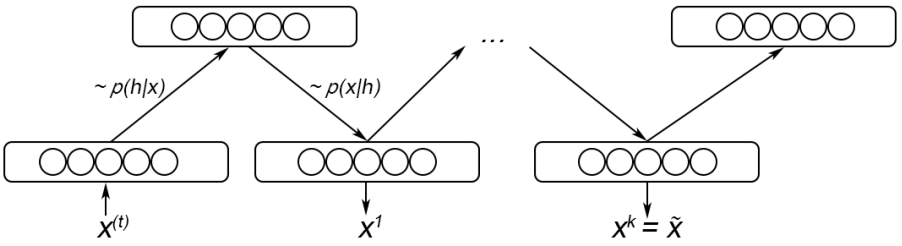


Figure 2.15.: A temporal unrolling of the contrastive divergence algorithm with k sampling step. The hidden units and the visible units are alternatingly sampled conditioned on the current state of the other.

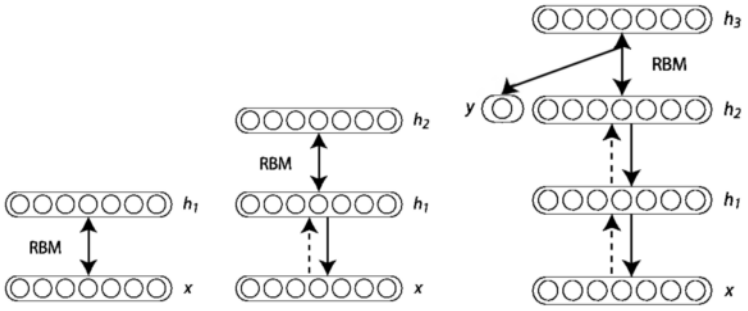


Figure 2.16.: Building up a deep belief network, by training RBMs greedily and stacking them up on top of each other. In the top layer "association" RBM the label information y can be incooperated as well.

A deep belief network can be build up by stacking up RBMs.

This does not only able to unsupervised train a deep belief net, but also improve the conditional distribution of the bottom RBM (Each time we improve the prior over the hidden units with a better and more complex prior).

In a deep belief net the two top layers have similar to a RBM symmetric weights while the symmetry is broken up between the lower layers to allow unsymmetrical weights (Hinton Example).

Training Training of a DBN can be split up into greedily training simple RBMs on a cascade of stacked up RBMs. 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 of the top RBM

Fine-tuning After greedily training the DBN, the bidirectional connections can be split up and fine tuned to a certain task.

One method is the the wake-sleep algorithm:

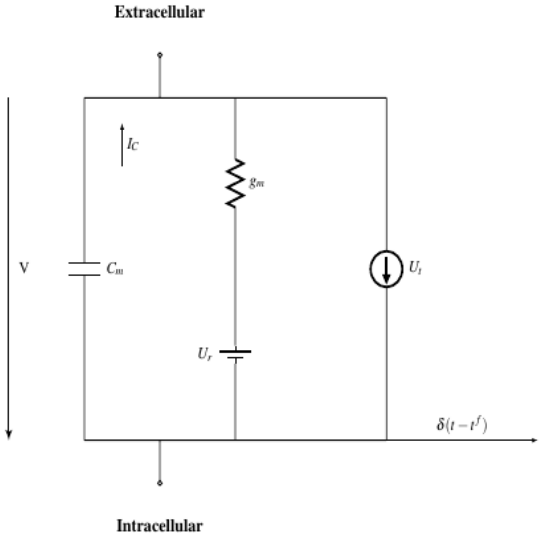


Figure 2.17.: A figure.

1. Do a stochastic bottom-up pass, and for each layer adjust the top-down weights, to be better at reconstructing the activation in the layer below.
2. Perform sampling steps in the top level RBM and adjust the weights with CD.
3. Do a stochastic top-down pass, and for each layer adjust the bottom-up weights, to be better at reconstructing the activation in the layer above.

Another and more common way to fine-tune the weights if labels and an error function are given, is to perform back propagation to further fine tune the bottom up weights (while the top down weights are usually removed). One interpretation of this is to see a DBN as a pretrained CNN.

2.2.3. Spiking neural networks

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

Neuron Models

Similar to the activation function in ANNs, in SNNs there are also different models describing have neurons process input.

LIF The leaky integrate and fire (LIF) neuron which phenomenological describes the membrane potential at the soma. It is one of the simplest and thus computationally most efficient, most important and popular spiking neuron model.

By discarding the different forms/shapes of the action potentials and reducing it the uniform spike events, the information in 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 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:

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

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 ^{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 it's reset potential u_{reset} and clamped to that for its refractory period t_ref.

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

The emitted spikes are modeled 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 function.

Due to its simplifications the LIF model can not cape some natural observed behaviour such as long-lasting refractoriness or adaptation.

Hodgkin-Huxley The Hodgkin-Huxley model tries to improve some of the limitations of the LIF model, by explicitly allowing to model different ion channels.

The first model described by Hodgkin-Huxley introduced three different ion channels, namely sodium, potassium and a leak current of CL- ions, which they discovered in their experiments on axon of a squid. Each channel is described be a resistor with a battery. 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/generalized to cope more three ion channels with their dynamics,

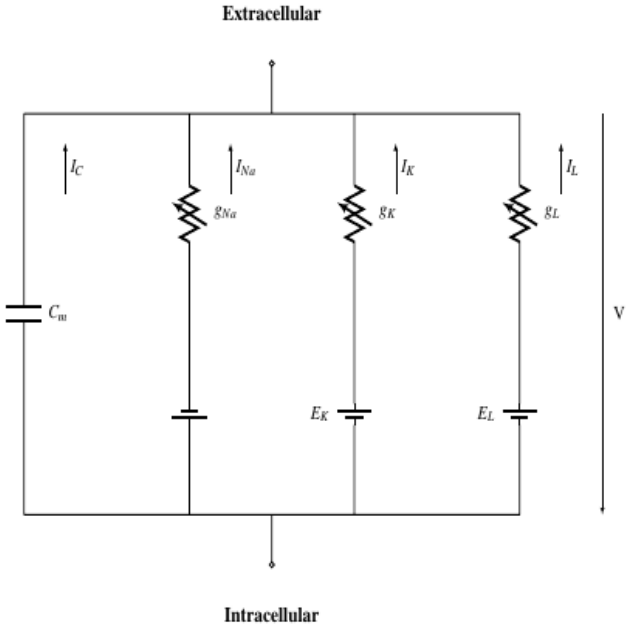


Figure 2.18.: A figure.

to better match the characteristics/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 to 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.

Poisson A Poisson neuron, 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.

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

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

where the occurrence of a spike is independed on previous spikes.

Since the spikes can be formalized as a Poisson process, the expected number of spikes for an 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 is ANNs simply multiply an incoming signal with its weight, for SNNs there are different models which add additional dynamics to closer model naturally observed behaviour.

The behavior of the synapses are described by modeling the synaptic input f^{syn}. A basic model of how the influence of synapses can be described as follows:

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

where ε is a function describing the spike handling dynamics of a synapse.

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 extension for LIF neurons since they directly model the potential at the soma and don't 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 by considering 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 properties of conductance based neurons is, that they can reach a so-called high conductance state (HCS). Such a state is defined by a high total membrane conductance

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

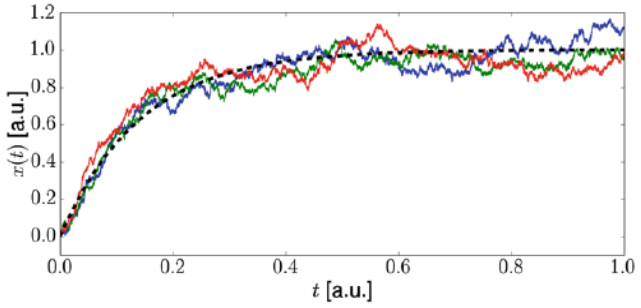


Figure 2.19.: A figure.

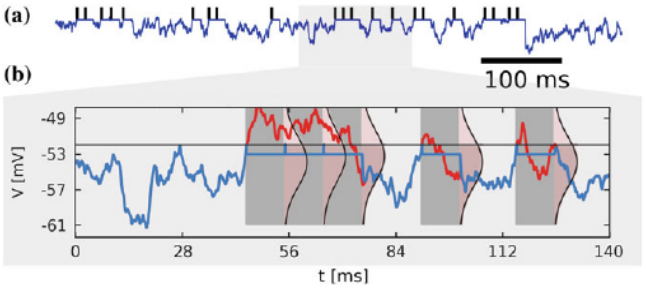


Figure 2.20.: A figure.

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, with a mean primarily determined by the effective synaptic input (with the noise) and the variance by the total membrane conductance. 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 Ornstein–Uhlenbeck process can basically be neglected.

Kernel function Next to the synaptic models, the kernel function ε has probabily 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. There have been several different kernel 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: $\varepsilon(t) \propto t \exp(-\frac{t}{\tau_s yn})$
- Alpha-shaped: $\varepsilon(t) \propto \exp(-\frac{t}{\tau_s yn})$

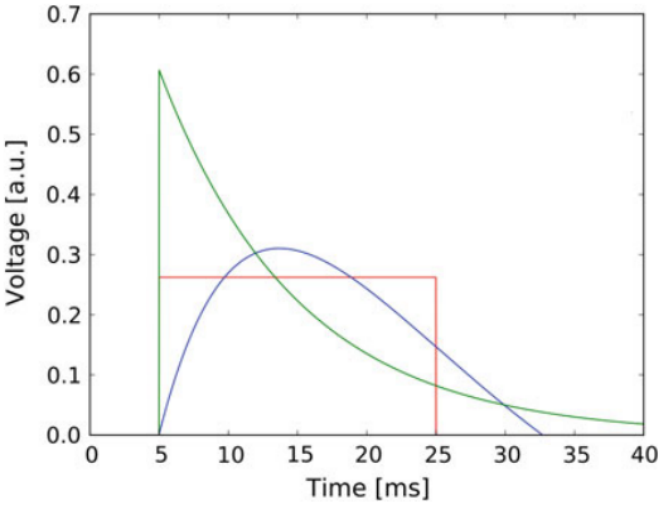


Figure 2.21.: A figure.

Neural Coding

Spikes are the primary way of information transmission in the brain. In neural networks this information can be encoded in different ways. Three encoding mechanisms, which were observed in the brain, encompass rate coding, temporal coding and population coding.

In the *rate coding* scheme the information is purely contained in the spike frequency/ 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.

Learning

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

Most models of learning algorithms in spiking neural networks build on changes in the synaptic weights/ strength between to neurons to store information. 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).

The models most commonly used are inspired on the research and discoveries of Hebb and the previously discussed Hebb principle.

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

Experiments indicated a 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-

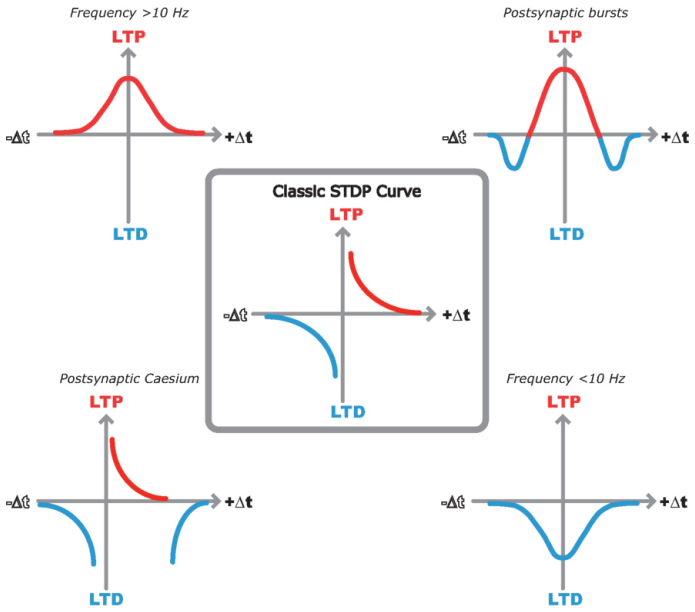


Figure 2.22.: A figure.

synaptic spike occurred in strong temporal vicinity before a pre-synaptic spike. If the further apart the spike times of the different neurons were, the weaker was the effect on the spike.

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

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

with a common choice for the STDP function W

$$W(x) = A_+ \exp(\frac{-x}{\tau_+}) \qquad \text{for } x > 0,$$
$$W(x) = -A_- \exp(\frac{x}{\tau_-}) \qquad \text{for } x < 0,$$

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

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

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

This leads to a purely positive increase of weights.

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

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

3. Related Work

3.1. Convolutional RBM

The convolutional RBM was invented more or less at the same time by Bengio and Lee. In similarity to CNN it can be seen as the advancement of energy based model adapting 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. Modeled after CNNs, cRBM have shared weights and are not fully connected.

For propagating information up can be seen as the convolution/cross correlation with a filter matrix W :

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

The down propagation uses the flipped kernel $\tilde{W}$:

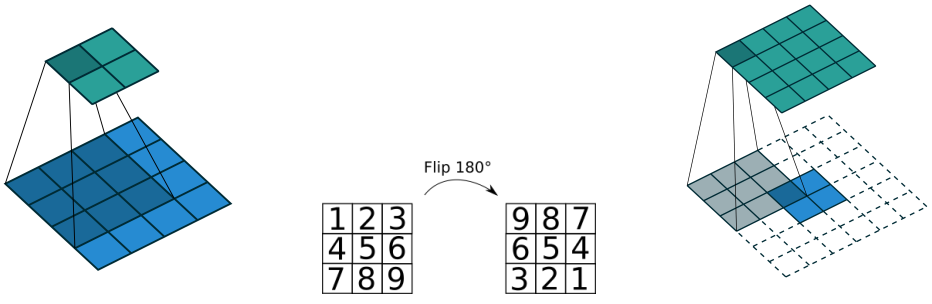
$$P(\mathbf{v}|\mathbf{h}) = \sigma((\tilde{W} * \mathbf{h}) + b_v).$$

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

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

Learning is similar to normal a RBM, a convolutional RBM is trained with the objective to maximize the probability of the training data. This can be achieved using the CD algorithm with a few adaptations due to the tied weights (see backprop for CNNs, $\frac{\partial E}{\partial w} = \sum \Delta w$).

In contrast to CNNs, due to their local learning rule, RBMs can not be explicitly trained to perform max pooling operations. Thus Lee proposed an softmax based probabilistic max pooling



(a) A subfigure (b) A subfigure (c) A subfigure

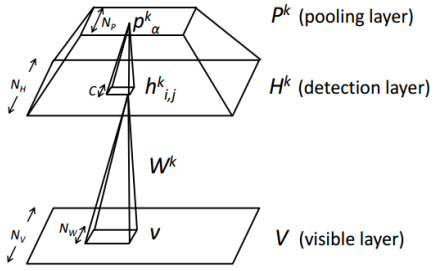


Figure 3.2.: A figure.

to introduce local sparseness in the hidden activations, on which a pooling layer can be stacked:

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

where $I(h^k_{ij})$ is the activation of the hidden unit before applying the sigmoid function ($(W * \mathbf{v}) + \mathbf{b}_h$) and B is a partition block containing unit h^k_{ij}.

3.2. Sampling in SNNs

Indicated by stochastic neural tranisitions found in the brain in experimental studdies, a new way of information encoding in the brain as representations of probability distributions and probabilistic interference has been suggested.

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

The neuron network can considered a network of RV, with the state of a neuron is defined 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 the firing state for a time period τ

$$z_k(t) = 1 \iff k \text{ has fired in the time intervall } (t-\tau, t]$$

where z_k is the state of neuron k and set to 1 for the "firing" state and 0 for the "not firing" state. 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 : $\frac{\#\text{spikes}_i\,\tau}{T}$.

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 $p(z_t|(z_0, ..., z_{t-1})) = p(z_t|z_{t-1})$, a auxiliary counter variable ξ is introduced which discretizes the time a neuron has left to stay in the "firing" 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

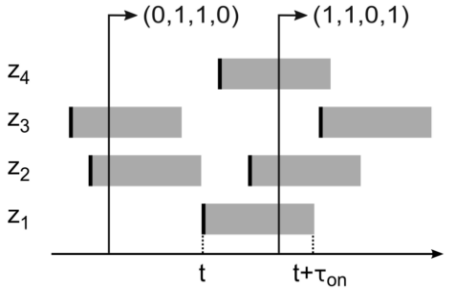


Figure 3.3.: A figure.

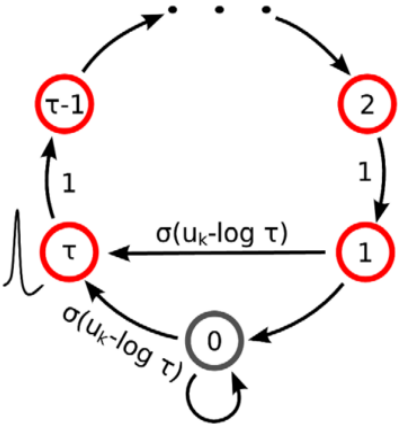


Figure 3.4.: A figure.

fired

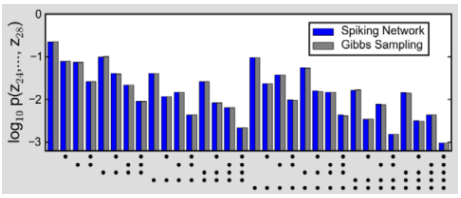
$$p(\xi_t|\xi_{t-1}) = \begin{cases} 1, & \text{for } \xi_{t-1} > 1 \text{ and } \xi_t = \xi_{t-1} - 1, \\ p(\text{"}firing\text{"}), & \text{for } \xi_{t-1} \leq 1 \text{ and } \xi_t = \tau, \\ p(\text{"}not\,firing\text{"}), & \text{for } \xi_{t-1} \leq 1 \text{ and } \xi_t = 0, \\ 0, & \text{otherwise.}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

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

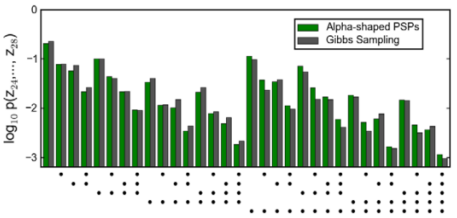
$$P(\text{"}i\ fires\ at\ time\ t\text{"}) \approx \sigma(\sum_j w_{ij} z_j(t) + b_i).$$

With this model Buesing proved that spikes and the corresponding state updates in a such networks 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 biolgical plausible alpha shaped PSP deteriorates the performance, due to overshooting at the beginning and accumulation effects, a little but is still reasonable well.

Petrovici improved the model by replacing the stochastic neuron model by conductance bases LIF neurons, a more common and biologically inspired neuron model. 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. This



(a) A subfigure



(b) A subfigure

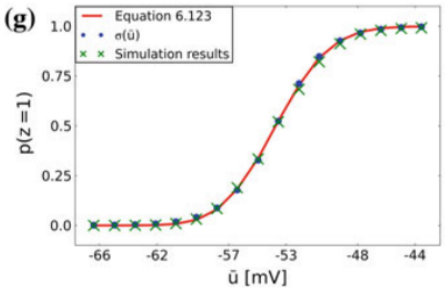


Figure 3.6.: A figure.

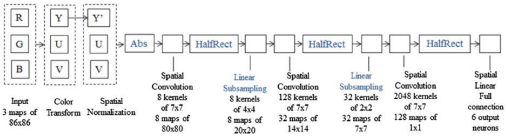
allows the neuron to show a firing behavior which can be matched by a logistic function. By normalizing the weights, the LIF neurons can so perform neural sampling similar to Buesing.

3.3. Artificial to spiking neural network conversion

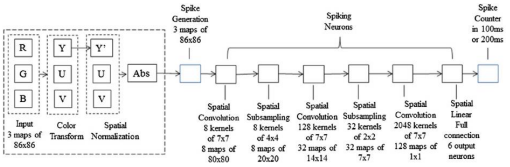
The results of Petrovici allow a quite simple transformation of a Boltzmann machine to spiking neural networks of noisy conductance based LIF neurons with the synaptic weights scaled to match the impact on the membrane potential.

Connor uses a different approach, where he instead of approximating sigmoid units by LIF neurons, uses the Siegert neuron, a rate base approximations of LIF neurons with Poisson input, to implement units in the RBM which activate similary to LIF neurons. Such a trained net can be directly transfered to a SNN. For a neuron with input rates ρ and weights w the output rate can be approximated by the Siegert transformation as

$$\rho_{out} = \left(t_{ref} + \frac{\tau}{\Gamma} \sqrt{\frac{\pi}{2}} \int_{V_{reset}+k\gamma\Gamma}^{V_{thres}+k\gamma\Gamma} \exp\left[\frac{(u-\Upsilon)^2}{2\Gamma^2}\right] \left[1+\mathrm{erf}\left(\frac{u-\Upsilon}{\Gamma\sqrt{2}}\right)\right] du \right)^{-1},$$



(a) A subfigure



(b) A subfigure

with the auxiliary variables

$$\mu_Q = \tau \sum w\rho,$$
$$\sigma_Q^2 = \frac{\tau}{2} \sum w^2 \rho,$$
$$\Upsilon = V_{rest} + \mu_Q,$$
$$\Gamma = \sigma_Q,'$$
$$k = \sqrt{\frac{\tau_{syn}}{\tau}},$$
$$\gamma = |\xi(\frac{1}{2})|,$$

where ξ is the Riemann zeta function.

ρ_{out} can be normalized by the maximal firing rate $\frac{1}{\tau_{ref}}$ 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 weight can be simply adapted to a SNN with LIF neurons described by the Siegert approximation.

There also have been several approaches to transform a CNN to a SNN. Cao et al and Diehl et al propose certain constrains on the CNN architecture to show reasonable performance in the converted SNN:

- 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} .
- The bias term are eliminated.
- Instead of max pooling, average pooling is used, since it has a simple spiking counterpart.

After the CNN is trained with the back propagation algorithm, the weights are transferred to a SNN with an equivalent architecture using LIF neurons without a refractory period. In addition Diehl et al use some model- and data-based weight normalization procedure to further fine-tune the synaptic weights.

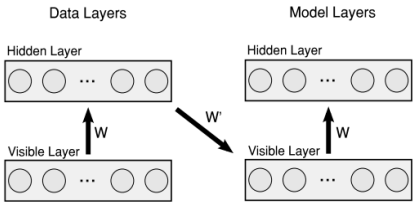


Figure 3.8.: A figure.

3.4. eCD and Sampling Machines

Different approaches have been proposed to train a rate based RBM, where the first was probably by Hinton. They use multiple binary stochastic input units of the same input to simulate a rate based input.

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

One approach is the evtCD by Daniel Neil which works in continuous time with spiking networks and a STDP variant. He simulates the positive and negative phase of the CD by simply unrolling the RBM (with shared weights). But this approach only allows a certain number of CD steps and is due to the weight synchronization not very plausible.

A more sophisticated approach which also uses STDP was proposed by Neftci. He uses bidirectional synapses, between a visible and hidden layer of LIF neurons. They use 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. The symmetric learning rule for two neurons, given their spike train $v_i(t)$ 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 STDP status flag $g(t)$ determining the positive and negative phase of the CD algorithm and the STDP function $STDP(v,h)$ determining the weight change depended in 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 behaviour 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 the training time can be divided into four phases:

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 stdp) and is added to the weights (postive phase $g(t) = 1$)

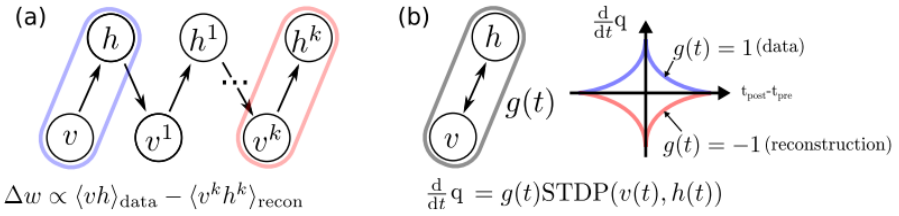


Figure 3.9.: A figure.

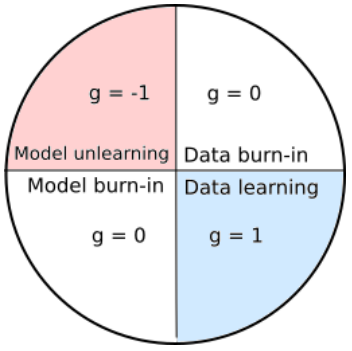


Figure 3.10.: A figure.

3. The data signal is remove 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 stdp) and is substracted 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:

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

4. Approach

In this section after describing the general convolutional architecture in spiking neural network, we discuss two different approaches to build convolutional deep belief networks.

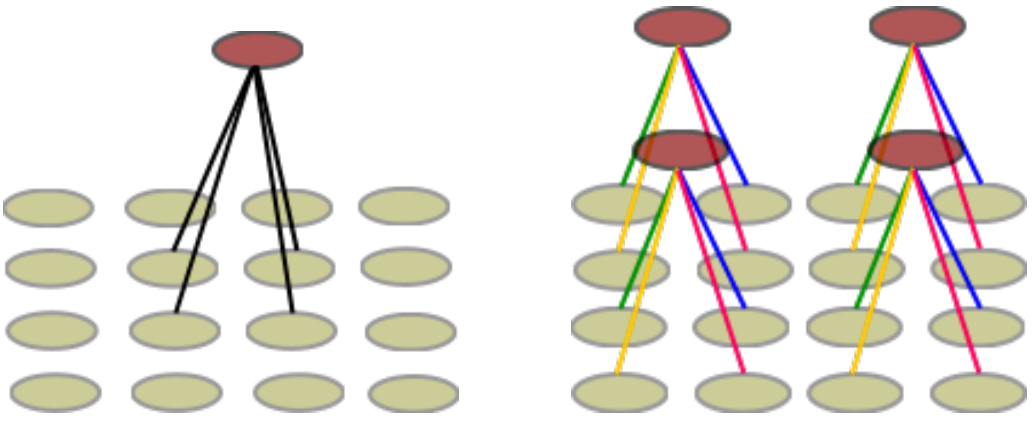
The first one is the most classical one, where a offline/ in discrete time trained DBN is transferred to spiking neural networks. The second approach trains the RBMs event based with STDP directly as spiking neural networks.

4.1. Convolutional architecture in spiking neural network

We implement a convolutional layer with receptive fields and shared weights between two neuron populations. Each population has a similar 3 dimensional topology as in "normal" CNNs with the number of neurons given by #neurons = channels $\times$ height $\times$ width.

Groups of neurons in the bottom layer in close vicinity, a so-called receptive fields, are each connected with synapses to a top layer neuron. These receptive fields have the same size n and shape and shared the same synaptic weights (given the top layer neurons belong to the same feature map). The receptive fields are overlapping by $n - s$ neurons, where s is the stride, and the output neurons have the same topology as their input regions. Recent state-of-the-art systems have mostly reached the common consensus, of choosing a stride of $s = 1$, which we have adapted throughout this work.

The receptive fields cover partial regions of the input data, where 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 weight of 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 the receptive fields. In this case the top layer activation of receptive fields with the same weights can also be called a feature map.



(a) A subfigure (b) A subfigure

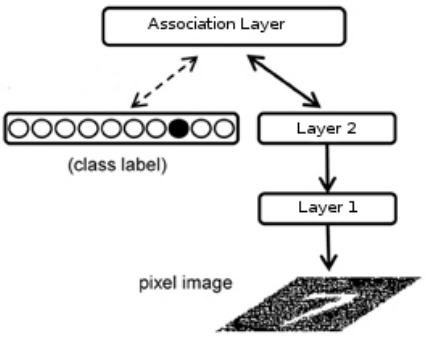


Figure 4.2.: A figure.

By using more neurons with different set of shared weights over the same receptive fields, more feature maps can be implemented.

4.2. Conversion

The conversion approach can be roughly described in two steps:

1. Train the RBMs to build up an DBN.

2. Convert the DBN to the spiking neural network.

4.2.1. Conv DBNs

To train a convolutional DBNs we proceed similar to Hinton and Lee.

At first the convolutional RBMs are greedily trained with CD, as described in section 2.x, on images batches of batch-size b for a certain number of iterations over the whole dataset, in this context referred to as epochs. After a RBM is trained, we convert the dataset by sampling of the hidden layer of the RBM (one sampling/forward-pass step), into a new dataset:

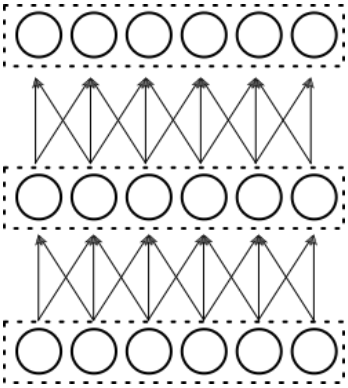
$$x' = \sigma(W * x) > rnd,$$

where rnd is a random number between 0 and 1.

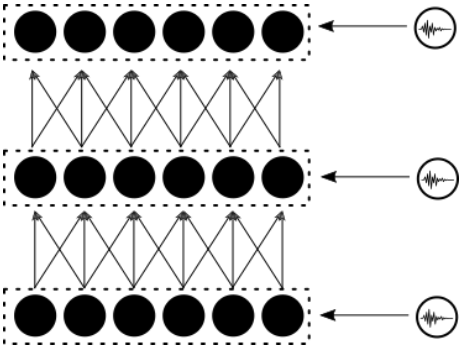
On this converted dataset the next convolutional RBM is trained similar to the previous one.

Evaluating feature qualities is still an active top of research. To get a measurement of our feature quality we use labeled data and train a classifier on top of the extracted features. To stay in a biological plausible domain, we train a fully connected RBM on the top level features as well as the label of the data samples, to associate the features with the correct labels. Out approach is similar to the approach of Hinton.

To evaluate the final performance, we input a data sample without a label into the DBN and let the top layer sample a label prediction by performing Gibbs sampling steps.



(a) A subfigure



(b) A subfigure

4.2.2. Conversion

For the conversion we have three different variations

Conversion as CNN One way to convert the DBN to the spiking domain, is by interpreting it as a pretrained 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).

For the conversion, we proceed similar to Cao and Diehl. They use avg pooling and ReLU functions, to get a similar architecture as SNNs. In contrast, we don't use any pooling, since for RBM there is no simple way to integrate avg pooling (used by Cao and Diehl in their trained CNNs), and for spiking CNNs there is no simple way to integrate probabilistic max pooling, which is currently probably the only training integrated pooling thought of for RBMs. We also use the sigmoid function, since RBMs are commonly trained with sigmoid activations (even so some approaches proposed ReLU for RBMs as well (see Hinton)) and the input-rate output-rate transfer function of rate based LIF neurons with a refractory period matches the sigmoid function more closely.

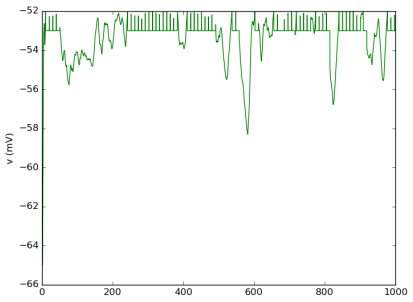
The DBN layers are replaced by a LIF neuron population layers with an identical architecture. The connections are replaced by (directed) synapses, with the weights of the DBN synapses scaled with a constant factor, to get similar activations.

Conversion with conductance-based LIF Another way to convert the DBN to the spiking domain, is by interpreting it as a directed graphical model, a sigmoid belief network, and perform ancestral sampling.

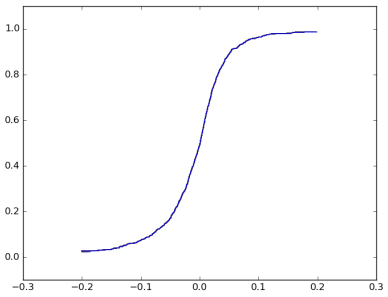
This approach is heavily based on the synaptic sampling theory, i.e. it uses spiking neurons to perform sampling.

The sampling can be either performed LIF neurons as described in 3.2.

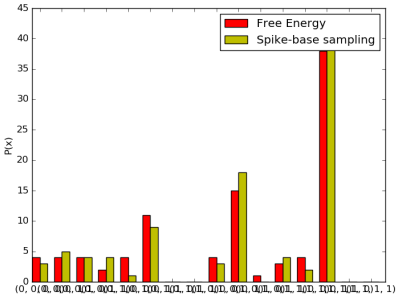
For the COBA neurons, we choose a biological plausible neuron model (see parameters in table). The high conductance and increased mean potential (gaussian distributed) and thus a firing probabilty of .5, is achieved by using high frequency Poisson generated (inhibitory and exhibitory)



(a) A subfigure



(b) A subfigure



(c) A subfigure

spikes to bring the neuron to a high conductance state (HCS).

This neuron model has an input-current/spikes to firing frequency mapping/ transfer function which approixmates a sigmoid function.

The PSP are chosen to have an alpha shape in stead of rectangular PSP, which as described in 3.2 may introduce some discrepancies when performing sampling in comparison to Gibbs sampling.

The DBN is converted by simply converting each layer to a layer of COBA LIFs with poisson noise, and the connections are transformed to synapses with the weights scaled to achieve a action function similar to the sigmoid function.

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

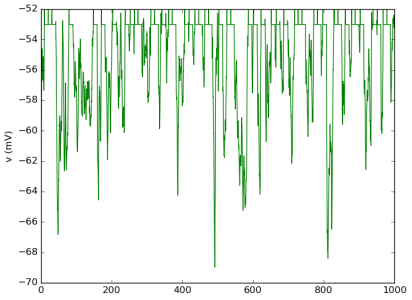
Conversion with current-based LIF To reduce the computational expenses of the COBA models, they can be replaced by less computational complex CUBA models.

Their model parameters as chose to simulate a HCS state.

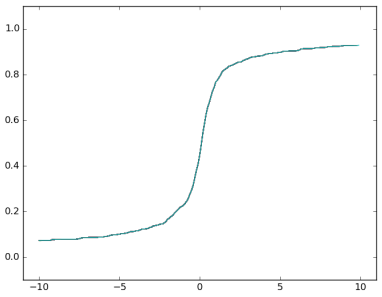
Therefore the membrane time constant as well as the membrane conductance are reduced and a static input current is inserted.

By adding high frequency poisson noise, a sigmoid shaped input-rate output-rate transfer function is achieved.

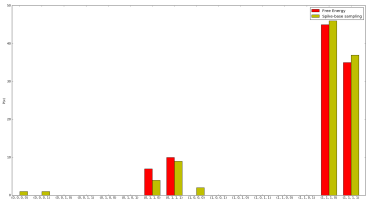
The DBN conversion is similar to the COBA case, where the COBA neurons are now replaced by the adapted CUBA LIFs.



(a) A subfigure



(b) A subfigure



(c) A subfigure

4.3. eCD

Another approach tries to learn the convolutional spiking DBN with an STDP based learning rule.

The main idea of the learning rule is adapted from Neftci. We also use a LIF neuron model. A similar STDP rule to the one described in 3.2 is used, but we extended the model with a learning rate. This rule can be reformulated as an iterative rule as follows:

- The visible unit v spikes:

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

- The visible unit h spikes:

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

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 a Dirac-shaped spike train.

The original division into four training phases poses similarites 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

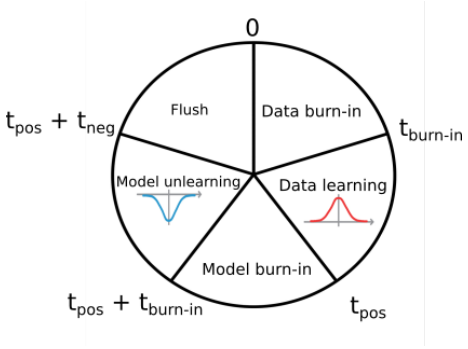


Figure 4.6.: A figure.

a 5th phase between the two data samples, where the model is "flushed" thus enabling normal CD:

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 stdp) and is added to the weights (postive phase $g(t) = 1$)

3. The data signal is remove 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 stdp) and is substracted 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, to learning is performed ($g(t) = 0$).

STDPCURVE

4.3.1. Convolution

We implement convolution with local receptive field and sharing weights between neuron through weight synchronization. Since each weight has their local STDP based update rule (eCD), we have to find a way to synchronize/share weights between all neurons in a layer. To keep the weights in neuron in a layer the same, we perform a weight synchronization step at discrete time steps, since updating all weights after a single update did not show any promising results. Thus the synchronization at a time step for weight shared weights w_i , w_j can be described by the following rule:

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

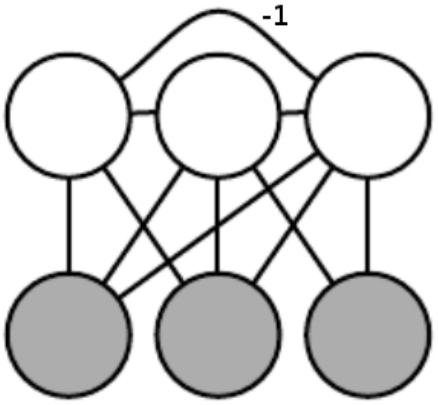


Figure 4.7.: A figure.

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 a update rule similar to the convolutional RBM update rule 3.x. 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 neuron in the top/ hidden layers (biological plausible –> more sparse King). This removes one advantage of RBMs since the hidden layers are no longer independed, which makes it harder to sample from the true distributions, but since the network continuously performs sampling steps, the approximation is is sufficient, if the weights are not to strong and prevent a change to a different mode.

By connecting neurons to neurons on a similar positions in other feature maps, appears to make the features more discriminative and less correlated. Also this poses some similarities to adding a negative structured bias to the hidden units, which has shown to result in better features (see Norouzi M).

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.

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.

To be exact the spiking DBN is a mixture between a DBN and a DBM, since the single BMs are still bidirectional connected and only the hidden layer is forwarded in a directed manner. Converting a RBM to a directed BN did not show any good results, since the activation of the hidden unit turned out to be important for generating a good estimate of the data distribution.

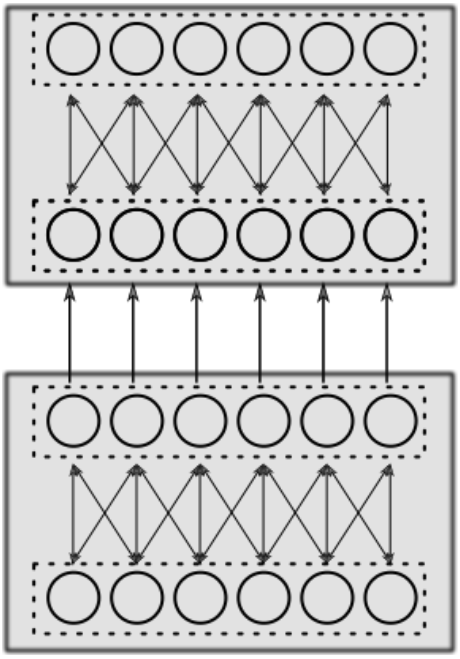


Figure 4.8.: A figure.

IMAGE

Due to some top-down influences, when stacking a new BM on the trained BM, the hidden distribution gets distorted and unfit input for the new BM to be trained on. To solve this in our case we simply use forward connections from the hidden activations to the input layer of a new two layered BM. An approach similar to Salakhutdinovs DBMs with doubling the weights and composing the models in the end may sound promising and could be tried as well.

5. Implementation

5.1. Analog DBNs

The DBNs were implemented in the Theano-Framework, to utilize the computational power of the GPU. The implementation of the singe RBMs is adapted from the official Theano RBM (REF).

An upward pass in performed as follows: As an input we take a 4D tensor $\mathbf{V}$ in the form of *(batch size, input channels, input height, input width)*, and for the upwards pass convolve it with a filters $\mathbf{W}$ of the shape *(number of filters, input channel, filter height, filter width)* with a stride of 1. 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 unit $\mathbf{H}$, and convolve it the the 180 degree flipped kernel $\tilde{\mathbf{W}}$, where the first and second dimensions are 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 on upwards and a consecutive downward pass.

Algorithm 3 Gibbs step

$V_0 \leftarrow upward(H_0)$
$H_1 \leftarrow downward(V_0)$

The free energy of the model is calculated according to equation (@TODO ref).

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

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

We train the RBM for a given number of epochs, with a batch size smaller than the dataset, thus performing stochastic gradient descent.

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

PSEUDOcode

Up to this time no label information is used to train any RBM, thus the learned features up to this point are trained purely unsupervised. The last layer, the classification layer, consists of a fully connected RBM trained on the input data of the converted data set concatenated with a 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. To simulate the CNN and DBN we use current and conductance based LIF neurons, respectively. Each unit is injected with high frequency Possion distributed excitatory and inhibitory input spikes with the frequency λ_{noise} and the synaptic weights w_{n+} and w_{n-} respectively. The parameters for the models can be seen in Table x(@TODO table).

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 a 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 spikes count of all single neurons in the label layer a_i is recorded and to get a label prediction the index of the most frequent spiking neuron is determined:

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

5.3. eCD

The eCD learning was implemented in PyNN with Nest with simulated STDP weight updated at discrete time steps outside the simulation, as well as Brian simulator with event-based online STDP weight updates, but due to the simulation speed 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 Poisson distributed spikes 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 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 an neuron population of the size $k = \textit{number of filters} \times (\textit{input height} - \textit{filter height} + 1) \times (\textit{input width} - \textit{filter width} + 1)$.

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 a 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 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 fed 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.

IMAGE

The weight synchronization is set at discrete time points, after n training samples. This is performed 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 a RBM is trained, the next RBM will be trained on top of the previous RBM. Therefore the a connection with a strong synaptic weight is established from the hidden layer of the previous RBM to 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 is fed to the bottom RBM . There is no external input forwarded the label layer while the number of spikes in the label are counted. The neuron with the most spikes represents the predicted label.

6. Experiments&Results

6.1. Datasets

We evalute 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. 1x4 Dataset

The simplest dataset we use a 1 dimensional binary dataset of size 4, where either the first or the last element is set to 1.

6.1.2. Strip Dataset

We generate a 10x10 pixel noisy stripe dataset, with three different oriented stripes, horizontal, diagonal, vertical.

This could represent a similar object to a pen recorded with a event-based camera, and result in an grasp id.

In the easiest version of this dataset, the stripes always occur on the same places, with some random noise.

An more complex version of the datasets contains the stripes randomly distributed across the whole image.

The dataset can be either binary or continuous.

6.1.3. MNIST

We also evaluate the models on the MNIST dataset.

The MNIST dataset consists of 60000 28x28 pixel gray images of the numbers 0-9.

We evaluate our models on the normal MNIST dataset, as well as a to dvs events converted version of MNIST.

6.2. Experiments

We orient our experiments primarily on the strip dataset, due to time and computational constrains.

Figure 6.1.: A figure.

Figure 6.2.: A figure.

After evaluating our models on the strip dataset, we look on the performance on other datasets for comparison and generalization.

6.2.1. Computational Constrains

6.2.2. Conversion comparison

6.2.3. Convolution vs no Convolution

6.2.4. Lateral connections

6.2.5. Hidden sparsity/ learning the data distribution

6.2.6. Same number of samples

7. Conclusion and Outlook

7.1. Biological plausibility

Studies by Hubel and Wiesel suggest, certain neurons respond so similar features at different position of in visible field.

This suggest similar synaptic weights.

One explanation of this could be shared weights similar to CNNs.

Since the weight updates in the brain are primarily believed to be local, there is no know principle to keep weights between different neurons in the brain synchronized.

So while the trained structure, with receptive fields and similar weights is quite plausible, the training procedure 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 repcetive fields get quite similar input, their weights will probably converge to the same target values.

While this requires all receptive fields to be presented with the whole 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 CNN utilize to get biological plausible result, while performing 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 (Bengio), where is no biological proof.

The STDP flag determining either completely positive or negative does not appear to be plausible as well.

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

A. Some appendix

A.1. (if needed)

Liste der noch zu erledigenden Punkte

B. List of Figures

C. List of Tables

D. Bibliography

[1] G. Berndes, M. Hoohwijk, and R. van den Broek. The contribution of biomass in the future global energy supply: a review of 17 studies. *Biomass and Bioenergy 25*, pages 1–28, 2003.

[2] W. Feng, H. J. van der Kooi, and J. de S. Arons. Phase equilibrium for biomass conversion processes in subcritical and supercritical water. *Chem. Eng. J. 98*, pages 105–113, 2003.

[3] X. H. Hao, L. Guo, X. M. Mao, and H. J. Chen. Hydrogen production from glucose as a model compound of biomass gasified in supercritical water. *Int. J. Hydrogen Energy*, pages 53–64, 2003.

[4] R. A. H. III. The age of energy gases. *Int. J. Hydrogen Energy 27*, pages 1–9, 2002.

[5] I.-G. Lee, M.-S. Kim, and S.-K. Ihm. Gasification of glucose in supercritical water. *Ind. Eng. Chem. Res. 41*, pages 1182–1185, 2002.

[6] Y. Matsumura and T. Minowa. Biomass gasification in near- and super-critical water: Status and prospects. *Biomass and Bioenergy 29*, pages 269–292, 2005.

[7] M. Osada, T. Sato, M. Watanabe, T. Adschiri, and K. Arai. Low temperature catalytic gasification of lignin and cellulose with a ruthenium catalyst in supercritical water. *Energy & Fuels 18*, pages 327–333, 2004.

[8] I. T. and Y. Matsumura. Gasification of cellulose, xylan and lignin mixtures in supercritical water. *Ind. Eng. Chem. Res. 40*, pages 5469–5474, 2001.

[9] S. T, S. Kurosawa, R. L. Smith, T. Adschiri, and K. Arai. Water gas shift reaction kinetics under noncatalytic conditions in supercritical water. *J. Sup. Fluids 29*, pages 113–119, 2004.

[10] M. O. T. Sato, M. Watanabe, M. Shirai, and K. Arai. Gasification of alkylphenols with supported noble metal catalyst in supercritical water. *Ind. Eng. Chem. Res. 42*, pages 4277–4282, 2003.

[11] J. D. Taylor, C. M. Herdman, B. C. Wu, K. Wally, and S. F. Rice. Hydrogen production in a compact supercritical water reformer. *Int. J. of Hyd. Ene. 28-11*, pages 1171–1178, 2003.

[12] T. Yoshida, Y. Oshima, and Y. Matsumura. Gasification of biomass model compounds and real biomass in supercritical water. *Biomass&Bioenergy 26*, pages 71–78, 2004.

[13] Y. Yoshida, K. Dowaki, Y. Matsumura, R. Matsuhashi, D. Li, H. Ishitani, and H. Komiyama. *Comprehensive comparison of efficiency and CO2 emissions between biomass energy conversion technologies-position of supercritical water gasification in biomass technologies, Biomass and Bioenergy*. 2003.

[14] D. Yu, M. Aihara, and M. J. Antal. Hydrogen production by steam reforming glucose in supercritical water. *Energy & Fuels, 7*, pages 574–577, 1993.