Universiteit Leiden
Opleiding Informatica

Theory and Applications of Deep Learning Networks

Name: Leon Helwerda

Date: 02/07/2014

1st supervisor: Walter Kosters
2nd supervisor: Jeannette de Graaf

BACHELOR THESIS

Leiden Institute of Advanced Computer Science (LIACS)
Leiden University
Niels Bohrweg 1
2333 CA Leiden
The Netherlands

Theory and Applications of Deep Learning Networks

Leon Helwerda

July 2, 2014

Abstract

We analyze the capabilities of various deep learning algorithms which employ multiple-layer networks in order to model abstract structures from data sets and recognize patterns. This makes it possible to perform various classification and normalization tasks. We investigate the use of a Deep Belief Network (DBN) and the Restricted Boltzmann Machine (RBM) as components for a deep learning network, as well as the alternative training algorithms that can be used to adjust the representation of the learned data. We introduce the underlying concepts and study the convergence properties. This allows us to derive another simplified version of a learning rule for the Contrastive Divergence algorithm used in pretraining the RBM. We also present a number of applications of Deep Learning by experiments on data sets. This includes handwriting detection and games.

1 Introduction

The field of artificial intelligence has been undergoing a transition towards the development of algorithms that are inspired by the data representations and connections within the human brain [ARK10]. In this bachelor thesis, we investigate the practical uses of emerging technologies such as deep learning networks.

We commonly refer to a deep architecture as a network composed of multiple levels, such as three or more, that contain nodes that are able to perform operations on their inputs in a non-linear way. It has been shown that there is no universally perfect depth of these networks in order to perform well on any problem. The addition of more layers or new techniques to train the nodes in those layers can improve the performance on one problem, but it might have unfavourable effects on other groups of problems.

This bachelor thesis was created in association with the LIACS Insitute of Leiden University, under supervision of Walter Kosters and Leannette de Graaf. In this thesis, we delve into the topic of deep learning to investigate the underlying theory, the decisions that can be made in the process of selecting a learning algorithm, and the applications of the technique.

1.1 Desires and problems

Although learning algorithms are improving time by time, there are still a few desired capabilities that these architectures have as of yet not completely fulfilled. We would like to reduce the number of training instances in the data set, such that the learning algorithm trains more quickly.

However, we want to do so without creating a large impact on the error rate, so that we still receive acceptable predictions for the test data.

This is especially important for results that have complex functions underlying them. A function that is highly-varying cannot have a piecewise approximation without a large number of data points. The architecture would need a large number of nodes if it is not structured correctly. If the number of variations is larger than the training set size, then the network can easily make the pitfall of only predicting local minima rather than generate correct predictions overall.

On the other hand, if we have generated a large set of training data, the algorithm should still be able to train quickly on it. We want to have an almost-linear complexity for the supervised training, otherwise the waiting time becomes too large. Another wish is that the programmer should not need to waste a lot of time to model the problem that needs to be solved. The algorithm needs to learn from the abstractions by itself.

There are also some other open questions in the field of machine learning. It has not been widely researched whether new learning architectures perform better on conventional problems, when we compare them to algorithms that were specifically created for those problems. Furthermore, there are problems that might take humans years to learn, while a deep learning algorithm might be able to train quickly. It is not yet known to which extent this is possible. We would also like to compare deep network architectures against existing algorithms that can also be trained in a supervised manner, such as Support Vector Machines [BL07, HL06].

The kinds of networks used in deep learning take vector data as input, but it is unknown how that would work for problems that are represented by other structures, such as trees or graphs. One could imagine that a game can be represented by an arbitrary number of moves, which would be difficult to input in a network with a fixed number of input nodes. We could also represent a game by the state it is currently in, but that may not be enough to allow the deep algorithm to play it [Ben09].

1.2 Applications

The applications of deep learning are now often geared toward image processing and other sensory inputs, such as face recognition and object detection in photos, or pattern recognition and automatic subtitling in audio streams. This trend can be seen in the models of networks that are proposed to solve these problems, because they often use components and sampling algorithms that specifically work well on binary data such as pixels. However, it might be possible that the proposed networks also perform well on more generic data.

Some of the underlying techniques of the deep networks have been based on definitions from physical systems, but also from abstract mathematical theories. In general, the application of concepts from various fields into machine learning brought great progress in the performance of algorithms such as deep learning, but it has also increased the complexity of the system.

1.3 Overview

After this short introduction of the importance of deep learning, we examine a number of varieties in the structure of deep networks. We investigate the functionality of Convolutional Neural Networks in Section 2 and Deep Belief Networks in Section 3. We further investigate the techniques that can be used in these networks, such as Restricted Boltzmann Machines in Section 4.

We take a closer look at a specific algorithm used for Restricted Boltzmann Machines in Section 4.2, and theoretically describe its convergence properties. We provide a simplified derivation of its learning rules in Section 4.2.2.

In Section 5, we propose a number of experiments, comparing the properties and parameters of the deep learning algorithms.

2 Convolutional Neural Network

The disadvantage of deep neural networks is that the commonly used learning algorithms become very slow. Furthermore, any attempt to correctly train the parameters of the nodes requires a large data set when the algorithm is too slow to update them into the correct direction. The *Convolutional Neural Network* (CNN) attempts to mitigate this problem. Inspired by the structure of the visual system of animals, this kind of network is organized in a way that allows modifying or preprocessing the input data so that the feature set contains only the important details. In the subsampling layers, the data is first weighted by a trainable filter and bias, and then passed on to a convolution grid, which uses data from neighboring neurons in order to average the value for the given input. Afterwards, the value is weighted again and passed through a sigmoid function, before being fed to a classical neural network [ARK10].

Because the data is averaged over its neighboring pixels, the Convolutional Neural Network is very applicable to the field of image processing. This is because it attempts to reduce noise from false positives in small areas of the image while detecting large areas better. This is useful in, e.g., face detection algorithms.

However, there are some downsides to this approach. Some parts of the CNN cannot be trained as flexibly as the well-known *BackPropagation* training method. Also, it is a discriminative method, meaning that we can only calculate $p(\text{Label} \mid \text{Observation})$, the probability that a certain classification label is correct for the given observation input. The reverse probability, which defines whether an observation occurs for a given label, cannot be deduced. A CNN is therefore more difficult to research, because one cannot determine whether there is an actual relation between an observation and the classification.

The uses of Convolutional Neural Nets have been somewhat limited to images [ARK10], although there are also successful attempts to use them on Go games [SN08]. The fact that a CNN employs filtering on the data set can reduce the necessary number of inputs for the eventual neural network, and it can also make it possible to train the CNN on a small data set, even when the number of possible states is enormous. However, it remains a challenge to select the most important features from a data set.

3 Deep Belief Network

As a further expansion of deep architectures, the *Deep Belief Network* (DBN) was proposed in [HOT06]. This kind of network has layers that are based on *Restricted Boltzmann Machines* (RBM), which are visibly similar to old-fashioned two-layer neural networks, and are described in Section 4. Each RBM layer is restricted in the sense that it only has one hidden and one visible layer. It is very easy to train them in order to find the weights for the connections between the layers.

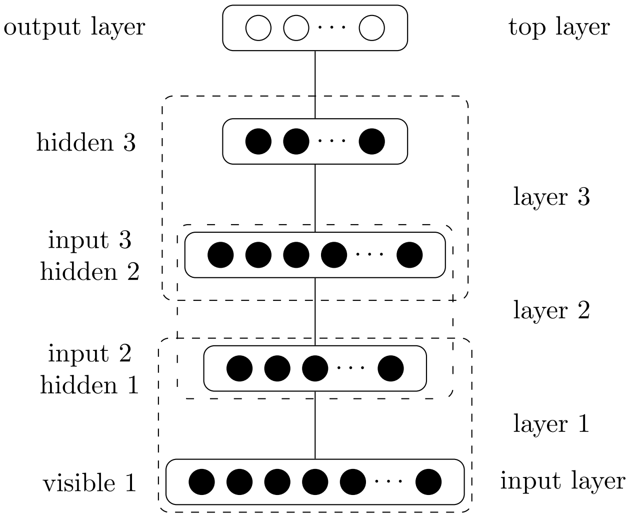


Figure 1: The generic structure of a Deep Belief Network. Each layer of nodes is connected to the next layer by weighted connections, except for the top layer.

The DBN is not a discriminative algorithm, since it is also possible to perform unsupervised training on the weights. This even holds for connections that are directed the other way around. This process works by calculating the activations of the hidden layers and then greedily reconstructing the corresponding input in a probabilistic manner [BLPL07]. This method is possible due to a principle called *contrastive divergence* [Hin02]. We use unsupervised training in a preprocessing step in order to initialize the parameters. This allows the network to contain a representation that can already approach the optimal solution.

After the unsupervised pre-training step, the RBM layers can be applied on the actual data set as if it were a normal *Multilayer Perceptron* (MLP) [RHW88]. For example, the DBN can link the output of the lower levels with an associative memory that contains samples. A more simple DBN would have some other top level layer, which can even have sigmoidal nodes like the other layers.

The top layer provides the actual prediction. Thus one could see this layer as a new visible layer that receives additional input. The RBM levels therefore mostly play an indicative role for the final prediction. An advantage of this structure is that it is still possible to perform *BackPropagation* on the network for fine-tuning.

3.1 Model

A Deep Belief Network consists of several kinds of layers that are placed on top of each other, as seen in Figure 1. Primarily, the DBN consists of RBM layers, which are explained in more detail in Section 4. These kinds of layers have one input layer and one hidden layer. The values of the hidden nodes of the RBM at layer i become the input for the RBM at layer $i + 1$. At the same time, each intermediate output of stack of RBM layers is associated with a normal sigmoidal layer, so that the DBN can be handled as if it is a Multilayer Perceptron, i.e., a normal neural network. This makes it possible to pretrain the network using the RBM stack, while using the same network as a normal feed-forward network for the finetuning and predictions.

At the top of the DBN, the results of the MLP need to be combined into an output, which can be done using a simple logistic regression layer or a more complex associative memory performing the final prediction.

As we can see, it is possible to modify the structure of the DBN in several ways: the types of the layers in the two-folded network can be changed, and the sizes of those layers are also parameters to the meta-network. The pretraining and training steps can be performed by different algorithms that work with the layer types.

The parameters to build a Deep Belief Network are as follows:

- m, the dimension of the input vector that the DBN accepts.
- $\boldsymbol{S}$, a list containing the number of nodes for each hidden layer in an RBM layer. This is a list of integers, so that each RBM layer i has its hidden layer size s_i registered in it, and the length of the list is the number of RBM layers in the network. Each RBM also has a visible input layer, and the sizes of these are determined by the number of outputs (and thus number of nodes in the hidden layer) of the previous layer s_{i-1}, or by m for the first layer. There needs to be at least one RBM layer; otherwise we would only have the top layer which does not entail a deep architecture.
- n, the dimension of the output of the top layer. For classification problems, one output might be enough, but for image processing problems such as the well-known handwriting detection, there is a need for more outputs to more accurately distinguish between cases, rather than collapsing them together.

Now, for every number of the list of hidden layer's sizes $\boldsymbol{S}$, we construct *two* types of layers, namely one layer of the MLP network, and an RBM layer, which has a visible and a hidden layer. Both types of layers share some parameters, namely their inputs (determined as described above), and the number of outputs (or hidden nodes in the RBM). The number of input nodes at layer $i + 1$ is ensured to be equal to the number of hidden nodes at RBM layer i. The RBM and MLP layers also share their weights with each other, so that the weights of the MLP are updated when the RBM layers are pretrained.

Next, we create the top layer of the DBN, which could be for example a logistic regression layer or a plain sigmoidal layer. It receives the outputs of the last MLP layer and its output consists of n values. The logistic layers use a specific error-correction cost function, which is computed using the negative log-likelihood function [BW88].

3.2 Samples and training

The network's input and output need to be modeled in a specific way. The input could be a raster of image pixels, which is received as a flattened version of a matrix. The output is also a one-dimensional vector that provides the prediction label.

Each RBM has a pretraining function based on the *Contrastive Divergence* function, as explained in Section 4. We pass the input data to the cost function and *Contrastive Divergence* update function of the RBM. Pretraining is done on small partitions of the training set, so the training functions should receive the batch index, and optionally a learning rate ε which can be altered during training. The training function simply passes the input data to the Contrastive Divergence update function of the RBM.

We can do the same for the finetuning, validation, and testing steps, calculating the error function from the sigmoidal network layers. We apply the finetuning on the full training set, and calculation of error scores on the test sets. In order to run a learning algorithm such as BackPropagation or logistic regression on all of the nodes during finetuning, we distribute the error correction updates across the weights of the nodes using the parameters of the MLP layers.

3.3 Additional techniques

Some of the deep network algorithms can make use of *Autoencoders*, also known as Autoassociators, to train their nodes more efficiently. An Autoassociator can encode the input while still making it possible to reconstruct the original data later on. This representation makes it possible to train the data in a separate one-layer network. More importantly, however, the Autoencoders have propagation functions that allow them to connect between two other layers. These Autoencoders are particularly helpful in Deep Belief Networks, and in fact the Restricted Boltzmann Machines that we define in Section 4 are somewhat similar to this technique.

There are also other deep learning architectures that are modeled after parts of the human brain, such as the Hierarchical Temporal Memory network. The layers follow a hierarchical structure so that higher levels correspond to larger regions of the input, in contrast to a DBN, where the compacting structure network has layers that specialize for specific features of the input. These kinds of networks are again specialized towards image inputs.

As we have seen in the CNN network, a lot of attention in the learning is attempting to reduce the problem instances to the most important data. Several other methods exist in order to filter and combine the input data. The reasons for this is that it becomes exponentially harder to learn when the dimensions of the instances increase. This problem, called the curse of dimensionality, effectively expresses the need for larger data sets when the instances have a large dimensionality, which needs to be prevented. Another suggested solution to this dire problem is to combine the concepts of RBM layers inside DBNs with the principle of Autoencoders, thus forming Stacked Autoencoders that are able to be trained in a similar way [HS06].

4 Restricted Boltzmann Machines

An RBM layer is represented by a visible input layer and a hidden layer. The bipartite network graph is connected by edges with weights. The nodes of each layer can be supplemented by visible biases and hidden biases, respectively; in this case there are as many bias nodes as there are normal nodes.

While RBM layers appear to be simple neural network components as shown in Figure 2, the method by which they can be pretrained is very intricate. The idea is that while there are dense connections between the nodes of the visible layer to the hidden layer, the nodes on the same layer are independent from each other. This structure makes it possible to obtain simultaneous samples by propagation, and we can start from reconstructions of other samples, in order to create chains of samples.

The Contrastive Divergence gradient approximates the actual log-likelihood [GG84], which cannot be determined in this network. The Contrastive Divergence update rules simply make use of the propagation functions of the network, which we will see in Section 4.1 and on.

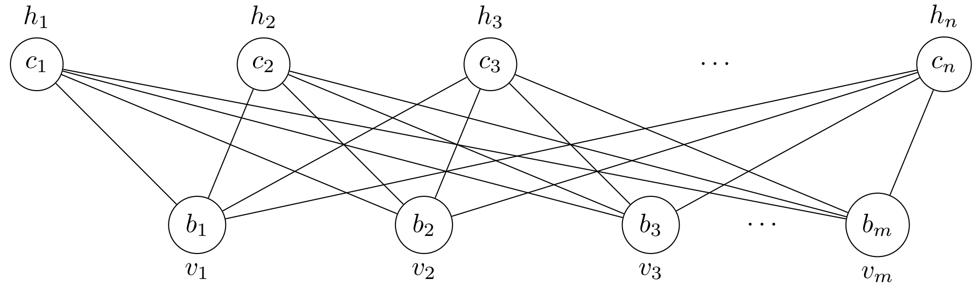


Figure 2: The graph of an RBM network, with n hidden nodes $h_1, \ldots, h_n$ and m visible nodes $v_1, \ldots, v_m$, as well as bias nodes $c_1, \ldots, c_n$ when going to the hidden layer and $b_1, \ldots, b_m$ when performing a propagation to the visible layer.

The definition of a Restricted Boltzmann Machine consists of the following parameters:

- m, the number of nodes in the visible layer.
- n, the number of nodes in the hidden layer.
- $\boldsymbol{W}$, a matrix of weights for the connections between the visible and hidden layer.
- $\boldsymbol{b}$ and $\boldsymbol{c}$, a list of biases for the visible nodes and hidden nodes, respectively.

If the RBM is used standalone, it initializes the weights to a uniformly distributed sample in a restricted domain, and the biases are initialized to zeroes or randomized values. However, when it is used in the DBN, the RBM layer is connected to a shared weight vector from the sigmoidal layer. The weights can be reused for the DBN network's weights. The biases can also be shared with the MLP section of the DBN, in particular the hidden biases.

4.1 Propagation properties

The hidden nodes and the visible nodes are activated by the unit values of the opposite layer, the biases, and the weights between the layers. Therefore, we need to be able to propagate the activations of the nodes upwards as well as downwards, as if it were a bidirectional symbolic weighted graph [Hin10]. In this section, we examine the propagation functions using the graph structure of the Restricted Boltzmann Machine.

The propagation functions, which we define in the context of the Contrastive Divergence algorithm in Section 4.2, are used during learning as well as in the practical use of the model. This is because they determine how the input values are converted to values on the next layer. Therefore, the propagation functions are an integral part of the RBM.

Restricted Boltzmann Machines are a special form of energy-based Boltzmann machine models. They define conditional probabilities for the activation of the nodes given a certain configuration, through the use of energy functions. For the restricted version of the Boltzmann machines, the generic equations can be rewritten to simpler versions. This is because the hidden nodes are conditionally independent from each other, and the visible nodes are as well [TWOH03].

The probability that the hidden nodes are activated, given the visible node values, is equal to

$$p(\boldsymbol{h} \mid \boldsymbol{v}) = \prod_{i}^{n} p(h_i \mid \boldsymbol{v})$$

and the probability for the visible nodes is

$$p(\boldsymbol{v} \mid \boldsymbol{h}) = \prod_{j}^{m} p(v_j \mid \boldsymbol{h}),$$

where $\boldsymbol{v}$ are the inputs of visible nodes and $\boldsymbol{h}$ hidden units [FI14].

The resulting functions that we define for the propagation of each node are shown in Equation 4.1 for hidden nodes and Equation 4.2 for visible nodes. These functions calculate the probability of the activation of a single hidden or visible node. The functions need to be given the state of the visible and hidden units as input, respectively. These symmetrical functions work by taking the dot product of the sample node values with their corresponding weights, adding the bias for the other layer, and calculating the sigmoid function. This makes it possible to sample the nodes in the other layer, by comparing the activations with a uniformly distributed chance.

4.1.1 Real-valued nodes

Propagation works with the conditional probability functions in binary-valued Restricted Boltzmann Machines, where each node $v_j, h_i \in \{0, 1\}$. If it is desirable, we can decide to replace the nodes with real-valued variables $v_j, h_i \in [0, 1]$. We can pretend that the expected activation $p(v_j = 1 \mid \boldsymbol{h})$ and $p(h_i = 1 \mid \boldsymbol{v})$ is that node's state.

Since it is not very straightforward to find the results of the probability terms, we can choose to simply set the node's value to that of the probability. This has the advantage that the network can accept data sets with real-valued inputs. The network loses some of its stochastic behavior and becomes more deterministic, but this is sometimes helpful for research.

4.2 Contrastive Divergence

The RBM can employ supervised or unsupervised learning through the use of these propagation functions, known as *Gibbs sampling.* The Contrastive Divergence algorithm repeats the transitions between the visible and hidden units over and over again. That should cause the probabilities to converge to the underlying distribution of the data. This means that those transitions can alter the state of the visible and hidden nodes so that it might become slightly different.

Based on the difference between the input and the reconstructed output from the propagations, we can train the weights in a certain way. We can alter them so that the reconstruction of the visible nodes will not change when it is given certain inputs. This causes the network to contain a representation of the most important samples from the data set.

Each step of Gibbs sampling can therefore be used to determine the difference between the input and generated output, and this error margin can be used to adapt the network so that it can perform better next time. Note that this technique will remain an approximation unless it is done infinitely long. However, even one step of Gibbs sampling appears to work well. This is a special case of the Contrastive Divergence (CD-k) algorithm, where $k = 1$ is the number of steps.

The Contrastive Divergence algorithm for binary units is outlined as follows [FI14]:

1. Given a training example $\boldsymbol{v}$ with its classification label, initialize $\boldsymbol{v}^0$ to this sample.
2. For every t from 0 to $k-1$, do the following steps:
   a. Create a sample for each of the hidden nodes h_i^t using its conditional probability

   $$p\left(h_i^t \mid \boldsymbol{v}^t\right) = \operatorname{sig}\left(\sum_{j=1}^{m} w_{ij} v_j^t + c_i\right), \tag{4.1}$$

   where w_{ij} is the weight between visible node j and hidden node i, v_j^t is the input value at node j during this step and c_i is the hidden node's bias. The node h_i is enabled if its probability or *activation* is greater that a uniformly distributed random number.

   b. Create a new sample for each of the visible nodes v_j^{t+1} using

   $$p\left(v_j^{t+1} \mid \boldsymbol{h}^t\right) = \operatorname{sig}\left(\sum_{i=1}^{n} w_{ij} h_i^t + b_j\right). \tag{4.2}$$

3. Calculate the final activation of the hidden nodes $p(h_i = 1 \mid \boldsymbol{v}^k)$ at step k; creating this sample is not necessary during learning.
4. Using the probabilities and activations found above, determine the updated values for $\Delta w_{ij} := \Delta w_{ij} + p(h_i = 1 \mid \boldsymbol{v}^0) \cdot v_j^0 - p(h_i = 1 \mid \boldsymbol{v}^k) \cdot v_j^k$, as well as for $\Delta b_j := \Delta b_j + v_j^0 - v_j^k$ and $\Delta c_i := \Delta c_i + p(h_i = 1 \mid \boldsymbol{v}^0) - p(h_i = 1 \mid \boldsymbol{v}^k)$.

The update values Δw_{ij}, Δb_j and Δc_i are initially zero. The Contrastive Divergence algorithm can be applied to all the examples in the training set S, in a random order, or on subsets of the training set. After each batch or epoch, the weights w_{ij} and biases b_j, c_i are updated using a learning rate ε and the mean update values as $\varepsilon \cdot \frac{\Delta}{|S|}$, and then the update values are reset. Due to this, smaller batch sizes allow the weights to update faster, so that it takes less epochs to initialize toward the model's distribution and lead toward better performance, when compared to putting a full training set in one batch per epoch.

4.2.1 Variants, convergence and backpropagation

An alternative for the basic Contrastive Divergence algorithm is *Persistent Contrastive Divergence* (PCD) [Tie08]. In this case, the chain of Gibbs sampling steps is again initialized with a training example, but when another sample $\boldsymbol{v}$ is given, the chain is not reset. Instead, the activations of the visible nodes from the previous sample are reused, and only the update rules of the Δ compare against the newer sample. This should allow the chain to respond when the input changes, and thus make a longer chain in general.

The Contrastive Divergence algorithm and its variations have been studied as to whether the resulting samples, which do not follow the gradient of the log-likelihood function [CPH05], actually converge to the underlying model distribution. We would like the generated samples to be as representative and error-free as possible. However, the algorithm starts with a sample from the data set and we can only perform a finite k iterations in the algorithm, so the sample at $\boldsymbol{v}^k$ will have some bias toward the specific input.

Despite these setbacks and estimations, the bias is generally small, and could be eliminated by a more precise algorithm that would be computationally intensive to run on its own.

It has been shown that the general use of Contrastive Divergence provides slightly improving results when k increases [BD09]. As k approaches infinity, it will exhibit fixed points [ST10]. Under certain conditions it has been proven that the learning samples indeed converge to a precise unbiased model [Yui05].

Contrastive Divergence is often compared and combined with the BackPropagation algorithm that is often used in feed-forward networks like Multilayer Perceptrons [RHW88]. Indeed, the variant known as Contrastive BackPropagation is a simpler version of CD that updates weights in such networks in a similar way [MH05]. As shown in [HOWT06], Contrastive BackPropagation updates the weights in the network using a probabilistic forward pass like in CD, then uses BackPropagation for the backward step. This removes some of the probabilistic nature of the chain that the algorithm generates, which might make it more biased. However, deterministic algorithms like these are easier to analyze.

4.2.2 Simplification

It can be shown that Contrastive Divergence is related to the error propagation gradient used in the BackPropagation algorithm if we simplify the algorithm somewhat. We use real-valued nodes using the chance itself as the value instead of a probabilistic distribution and restrict ourselves to a single step of the CD algorithm. Thus we use the propagation functions

$$h_i^k = g\left(\sum_{j=1}^{m} w_{ij} v_j^k + c_i\right) \qquad (i = 1, \dots, n)$$

and

$$v_j^{k+1} = g\left(\sum_{i=1}^{n} w_{ij} h_i^k + b_j\right) \qquad (j = 1, \dots, m)$$

based on Equations 4.1 and 4.2. In particular, we are interested in the cases where $g = \mathrm{sig}$, which has $\mathrm{sig}(x) = \frac{1}{1+e^{-x}}$ and $\mathrm{sig}' = \mathrm{sig} \cdot (1 - \mathrm{sig})$, as well as for the case where $g = \mathrm{id}$. Note that $\mathrm{id}(x) = x$ and $\mathrm{id}' = 1$.

We can define the error function

$$\mathrm{Error} = \|v^0 - v^1\|_2 = \sqrt{\sum_{j=1}^{m} \left(v_j^0 - v_j^1\right)^2}$$

and an energy function

$$E = \frac{1}{2}\mathrm{Error}^2 = \frac{1}{2}\sum_{j=1}^{m} \left(v_j^0 - v_j^1\right)^2,$$

which are derived from our update values of the visible units, for which we want to minimize the difference or error between propagation steps [RN10].

Now, for a fixed visible bias b_j, we determine the gradient, giving the trend in which the error E decreases the most:

$$-\frac{\partial E}{\partial b_j} = \left(v_j^0 - v_j^1\right) \cdot \frac{\partial \mathrm{Error}}{\partial b_j} = \left(v_j^0 - v_j^1\right) \cdot g'(\mathrm{in\_v}_j^0) \tag{4.3}$$

where $\text{in\_v}_j^0 = \sum_{i=1}^n w_{ij} h_i^k + b_j$ (thus $v_j^{k+1} = g(\text{in\_v}_j^k)$). This function depends on the values of the hidden layer, which is constant for this partial derivative, since they will not contain any more b_j's.

For the hidden biases c_i, we have

$$-\frac{\partial E}{\partial c_i} = \sum_{j=1}^{m} \left(v_j^0 - v_j^1\right) \cdot g'(\text{in\_v}_j^0) \cdot w_{ij} \cdot g'(\text{in\_h}_i^0) \tag{4.4}$$

where $\text{in\_h}_i^k = \sum_{j=1}^m w_{ij} v_j^k + c_i$ (so $h_i^k = g(\text{in\_h}_i^k)$). In the case of $g = \text{id}$, Equation 4.3 results in

$$-\frac{\partial E}{\partial b_j} = v_j^0 - v_j^1,$$

which can also be seen in the case that we take $g = \text{sig}$. The second term $g'(\text{in\_v}_j^0)$ remains constant, since the sigmoidal functions will not contain any term with a b_j and is thus eliminated.

Furthermore, for $g = \text{id}$, Equation 4.4 results in

$$-\frac{\partial E}{\partial c_i} = \sum_{j=1}^{m} \left(v_j^0 - v_j^1\right) \cdot w_{ij} = \text{in\_h}_i^0 - \text{in\_h}_i^1 = h_i^0 - h_i^1.$$

Now the only calculation that remains is the error gradient for the weights:

$$\begin{aligned}
-\frac{\partial E}{\partial w_{ij}} &= \left(v_j^0 - v_j^1\right) \cdot g'(\text{in\_v}_j^0) \cdot \frac{\partial \text{in\_v}_j^1}{\partial w_{ij}} + \sum_{\ell=1}^{m} \left(v_\ell^0 - v_\ell^1\right) \cdot g'(\text{in\_v}_\ell^0) \cdot \frac{\partial \text{in\_v}_l^1}{\partial w_{i\ell}} \\
&= \left(v_j^0 - v_j^1\right) \cdot g'(\text{in\_v}_j^0) \cdot h_i^0 + \sum_{\ell=1}^{m} \left(v_\ell^0 - v_\ell^1\right) \cdot g'(\text{in\_v}_\ell^0) \cdot w_{i\ell} \cdot g'(\text{in\_h}_i^0) \cdot \frac{\partial \text{in\_h}_i^0}{\partial w_{i\ell}} \\
&= \left(v_j^0 - v_j^1\right) \cdot g'(\text{in\_v}_j^0) \cdot h_i^0 + \sum_{\ell=1}^{m} \left(v_\ell^0 - v_\ell^1\right) \cdot g'(\text{in\_v}_\ell^0) \cdot w_{i\ell} \cdot g'(\text{in\_h}_i^0) \cdot v_j^0 \qquad (4.5)
\end{aligned}$$

In the case $g = \text{id}$ we have

$$\begin{aligned}
-\frac{\partial E}{\partial w_{ij}} &= \left(v_j^0 - v_j^1\right) \cdot h_i^0 + \sum_{\ell=1}^{m} \left(v_\ell^0 - v_\ell^1\right) \cdot w_{i\ell} \cdot v_j^0 \\
&= \left(v_j^0 - v_j^1\right) \cdot h_i^0 + \left(h_i^0 - h_i^1\right) \cdot v_j^0. \qquad (4.6)
\end{aligned}$$

Now note that this result will converge to

$$\left(v_j^0 - v_j^1\right) \cdot h_i^0 + \left(h_i^0 - h_i^1\right) \cdot v_j^0 \longrightarrow h_i^0 \cdot v_j^0 - h_i^1 \cdot v_j^1$$

if $||v^0 - v^1||_2 \to 0$, or in other words, our Error term converges to zero.

This is an interesting observation, since we were able to find the learning rules of the Contrastive Divergence algorithm for the visible and hidden biases, but the update rule for the weights is more intricate.

The resulting error gradient terms thus closely correspond to the update rules in the algorithm, and the Error function we chose is visibly similar to the update rule for BackPropagation. We could also incorporate probabilities in the update rules and perform this derivation for the general Contrastive Divergence algorithm as long as an E is chosen with which we can take the derivative of the error term. Often E is taken as $\log(p(\boldsymbol{v}))$ to generate the log-likelihood gradient.

4.2.3 Example

We can further show that the formula found above, even though it has different convergence properties than the update formula used in CD, is correct by means of an example. Let $m = 2$ and $n = 2$. Thus the error gradient E is

$$E = \frac{1}{2} \sum_{j=1}^{m} \left(v_j^0 - v_j^1\right)^2 = \frac{1}{2}\left(v_1^0 - v_1^1\right)^2 + \frac{1}{2}\left(v_2^0 - v_2^1\right)^2.$$

In the case of $g = \mathrm{id}$, this is equal to

$$\begin{aligned}
E &= \frac{1}{2}\left(v_1^0 - v_1^1\right)^2 + \frac{1}{2}\left(v_2^0 - v_2^1\right)^2 \\
&= \frac{1}{2}\left(v_1^0 - w_{11}h_1^0 - w_{21}h_2^0 - b_1\right)^2 + \frac{1}{2}\left(v_2^0 - w_{12}h_1^0 - w_{22}h_2^0 - b_2\right)^2 \\
&= \frac{1}{2}\left(v_1^0 - w_{11}(w_{11}v_1^0 + w_{12}v_2^0 + c_1) - w_{21}(w_{21}v_1^0 + w_{22}v_2^0 + c_2) - b_1\right)^2 \\
&\quad + \frac{1}{2}\left(v_2^0 - w_{12}(w_{11}v_1^0 + w_{12}v_2^0 + c_1) - w_{22}(w_{21}v_1^0 + w_{22}v_2^0 + c_2) - b_2\right)^2 \\
&= \frac{1}{2}\left(v_1^0 - w_{11}^2 v_1^0 - w_{11}w_{12}v_2^0 - w_{11}c_1 - w_{21}^2 v_1^0 - w_{21}w_{22}v_2^0 - w_{21}c_2 - b_1\right)^2 \\
&\quad + \frac{1}{2}\left(v_2^0 - w_{12}w_{11}v_1^0 - w_{12}^2 v_2^0 - w_{12}c_1 - w_{22}w_{21}v_1^0 - w_{22}^2 v_2^0 - w_{22}c_2 - b_2\right)^2
\end{aligned}$$

where every $w_{rs}^2 = w_{rs} \cdot w_{rs}$ is a square of the weight between the hidden node r and visible node s; the exponent 2 is *not* a step index (remember that we only investigate the case of a single step). So

$$v_1^1 = w_{11}^2 v_1^0 + w_{11}w_{12}v_2^0 + w_{11}c_1 + w_{21}^2 v_1^0 + w_{21}w_{22}v_2^0 + w_{21}c_2 + b_1$$

and

$$v_2^1 = w_{12}w_{11}v_1^0 + w_{12}^2 v_2^0 + w_{12}c_1 + w_{22}w_{21}v_1^0 + w_{22}^2 v_2^0 + w_{22}c_2 + b_2.$$

Now we determine the error gradient for every combination of $i = 1, 2$ and $j = 1, 2$.

$$\begin{aligned}
-\frac{\partial E}{\partial w_{11}} &= (v_1^0 - v_1^1) \cdot \frac{\partial v_1^1}{\partial w_{11}} + (v_2^0 - v_2^1) \cdot \frac{\partial v_2^1}{\partial w_{11}} \\
&= (v_1^0 - v_1^1) \cdot (2w_{11}v_1^0 + w_{12}v_2^0 + c_1) + (v_2^0 - v_2^1) \cdot w_{12}v_1^0 \\
&= (v_1^0 - v_1^1) \cdot (w_{11}v_1^0 + h_1^0) + (v_2^0 - v_2^1) \cdot w_{12}v_1^0 \\
&= v_1^0 w_{11}v_1^0 + v_1^0 h_1^0 - v_1^1 w_{11}v_1^0 - v_1^1 h_1^0 + v_2^0 w_{12}v_1^0 - v_2^1 w_{12}v_1^0.
\end{aligned}$$

Now we can combine the terms again, taking care of the signs:

$$-\frac{\partial E}{\partial w_{11}} = v_1^0(w_{11}v_1^0 + w_{12}v_2^0 - w_{11}v_1^1 - w_{12}v_2^1) + h_1^0(v_1^0 - v_1^1).$$

Given the equalities $w_{11}v_1^0 + w_{12}v_2^0 = h_1^0 - c_1$ and $-w_{11}v_1^1 - w_{12}v_2^1 = -h_1^1 + c_1$ and these opposing c_1's cancel each other out, this results in

$$-\frac{\partial E}{\partial w_{11}} = v_1^0(h_1^0 - h_1^1) + h_1^0(v_1^0 - v_1^1)$$

This corresponds with the $v_j^0(h_i^0 - h_i^1) + h_i^0(v_j^0 - v_j^1)$ that we found in Equation 4.6. Similarly,

$$\begin{aligned}
-\frac{\partial E}{\partial w_{12}} &= (v_1^0 - v_1^1) \cdot \frac{\partial v_1^1}{\partial w_{12}} + (v_2^0 - v_2^1) \cdot \frac{\partial v_2^1}{\partial w_{12}} \\
&= (v_1^0 - v_1^1) \cdot w_{11} v_2^0 + (v_2^0 - v_2^1) \cdot (w_{11} v_1^0 + 2 w_{12} v_2^0 + c_1) \\
&= (v_1^0 - v_1^1) \cdot w_{11} v_2^0 + (v_2^0 - v_2^1) \cdot (w_{12} v_2^0 + h_1^0) \\
&= v_1^0 w_{11} v_2^0 - v_1^1 w_{11} v_2^0 + v_2^0 w_{12} v_2^0 + v_2^0 h_1^0 - v_2^1 w_{12} v_2^0 - v_2^1 h_1^0 \\
&= v_2^0 (v_1^0 w_{11} + v_2^0 w_{12} - v_1^1 w_{11} - v_2^1 w_{12}) + h_1^0 (v_2^0 - v_2^1) \\
&= v_2^0 (h_1^0 - h_2^0) + h_1^0 (v_2^0 - v_2^1).
\end{aligned}$$

The same can be done for w_{21} and w_{22}; the eventual indices match up with the general expression given in Equation 4.6.

4.3 Learning properties

After the RBM has learned from the input data, it contains a black-box representation of the data distribution. It has therefore determined the important features from the examples that distinguish one classification from the other. Through this, the RBM alone can already predict the target value of unlabeled test examples at a decent error rate. Even when it is learned without labels, it can provide normalized samples of the data at an uniform distribution of the data set. If we use supervised learning with labels, then we can additionally generate normalized samples for a specific classification. This can be done by providing a nulled sample with only the target set to a given label. The RBM will then very likely provide a corresponding sample for that label. If it was not sufficiently trained, however, it might generate noisy data, or accidentally return a sample for another classification.

In supervised training, both the sample values and the target are passed to the RBM as input nodes. In this case, the learning algorithm converges the network from its initial random state to specialist hidden nodes that detect patterns in the input. Also, the output of the RBM when used in a classification problem is actually generated at the visible nodes again. It is the result of a chain of Gibbs sampling, which may be followed as long as one wishes. This can provide accurate predictions because it should not use the target area of the inputs during the classification problem, by setting them to zero.

If the RBM is trained without labels, it extracts features from the input that can be used to represent the data in a more compact manner. This makes the Restricted Boltzmann Machine particularly useful as a component in a larger, deep neural network, such as Deep Belief Networks. The state of the network still exhibits the properties seen in supervised learning, but it cannot be used for classification on its own. It is vital to use unsupervised training when working with the RBM as a component, whereas training with the labels given only works if the RBM is used as a standalone network.

When multiple layers of RBM components are used in a Deep Belief Network, it is important to completely perform the unsupervised pretraining on the first RBM layer until it is settled. Otherwise, training the next layer uses hidden activations of the previous layer that have not yet converged. The next layer would then not receive a representative distribution of the input. Naturally, that means that this layer cannot be pretrained properly as well.

One use of RBMs is to generate textures or fill in missing pieces in images [KW12]. This is done in a process called *inpainting*, where the trained RBM receives a piece of an unlabelled image. Given this input, the RBM is able to determine the likely values for surrounding pixels, so that it creates an admissible picture.

5 Experiments

In this thesis, we have highlighted various theoretical properties of deep learning architectures. The question now is whether the standalone Restricted Boltzmann Machine and the Deep Belief Network can actually represent complex classification problems. These experiments apply the networks to learn from various data sets. We investigate whether the simplified version of the Contrastive Divergence update rule has similar properties as the learning rules in [FI14]. We also study the effects of various parameter settings and make sure that our setup can adapt to different network configurations.

5.1 Program

We have analyzed various available packages that are able to train Restricted Boltzmann Machines and Deep Belief Networks. The Python implementations available for pylearn2 [GWFL$^+$13] using SciPy and Theano [BBB$^+$10], such as [Tea13], are interesting because of the possibility to perform certain calculations on GPUs and the opportunity to use precompiled executables. These features might bring performance improvements for function evaluations and scanning large matrices, and mitigates some of the downsides of evaluating an interpreted language.

However, the question is whether a training program could also be written in a more lower-level language like C. This choice would allow for more direct memory management, so that certain operations, e.g., copying large arrays, are very fast. The code could also run on server-like lab instances that do not employ GPUs for speedups, and multi-threading is not a major concern. As such, the task is to design a new program that can train a deep network, and not to reuse one and adapt it to our needs. We use this new program to perform experiments with many varying parameters.

A major advantage of C-based languages is *const-correctness* [SA04], which allows us to require that functions used for validation and reporting cannot change anything in the network's state.

The program needs to be able to accept various settings so that they can be compared in experiments, by performing training algorithms based on those settings and reporting about them. Specifically, we need to be able to choose between a DBN or RBM network and whether they contain binary nodes or real-valued nodes that use the activations as values. Of course we also need to choose the number of nodes per layer, although the input and output nodes are defined by the format of the chosen data set. When we train an RBM layer or standalone network, we want to be able to choose between normal Contrastive Divergence or the persistent-chain version PCD, or use the learning rule from our simplified formula shown in Section 4.2.2. Other parameters for the algorithm, such as the number of Gibbs steps, as well as the batch size of the number of samples to run before updating weights, can also be given. Monitoring is available via image samples and graphical output of the filter weight for every layer. Additionally, we generate a report of the network when validated against a test set.

5.2 Setup

We will perform experiments on three different data sets. The first data set is the MNIST database of handwritten digits, which is often used to compare various classification algorithms, including deep networks [LCB]. The MNIST data set is a graphical data set of small images. There are 60000 examples in the training set and 10000 test samples, and each image has a label corresponding to the digit it represents. Each classification has about the same number of examples so that the data set is uniformly distributed. The data set is also very useful for generating normalized images of each classification label in a network like a Restricted Boltzmann Machine.

We also use two instances of data sets that are related to complex games. Often these data sets have ordinal data, and a network with binary or with real-valued nodes should also be able to train on them. For some data sets, we need to convert the ordinal tags or textual classifications so that the samples can be read by our network. We use the Poker data set from UCI [CO] to test the performance of the networks on games. The training set has 25010 examples while the test set has one million instances, and the distribution over the classifications is very unbalanced so that common hands such as a worthless hand or only one pair occur much more frequently than a rare hand, e.g., a Royal flush: the former classifications both encompass almost 50% of the data set while the Royal flush has only eight instances in total.

We use the program defined in Section 5.1 to process these data sets and train the deep networks. Through the following experiments, we compare the influence of input properties, parameters, network structures, and learning algorithms:

- We analyze how well a Restricted Boltzmann Machine can recognize a large number of target classifications. We use different sets of target labels and filter the inputs of the training and test sets so that the network can run on a subset of the data. We also determine how the performance on these subsets develops when the number of hidden nodes in the RBM is changed.
- We further investigate how the choice of the number of hidden nodes affects the learning of the RBM on the data set. We take runs of the pretraining phase and also validate its performance after each epoch. We pass the data set using the normal parameters for batch sizes, and other influences remain constant. Thus we can plot the number of hidden nodes and the number of epochs against the mean error that the network has on the test set.
- We compare the various algorithms and parameters for RBM networks. We seek whether the simplified version of our learning rule has any benefit over the normal Contrastive Divergence. We also test the Persistent version of CD, and vary the k parameter for the number of Gibbs steps. We also use these pretraining methods for RBM layers when they are used as components in a DBN network. In this case, we keep the structure of the DBN fixed.
- Finally, we run complete DBN networks on our data sets. These experiments attempt to select a decent structure for the given representation of the data set. This means that we can vary the number of nodes on each layer. We can also decide how many epochs we run for both the pretraining and the training phase. This can be somewhat simplified by some monitoring functionality such as *early-stopping*, so that we can end a training phase before the mean error becomes too large.

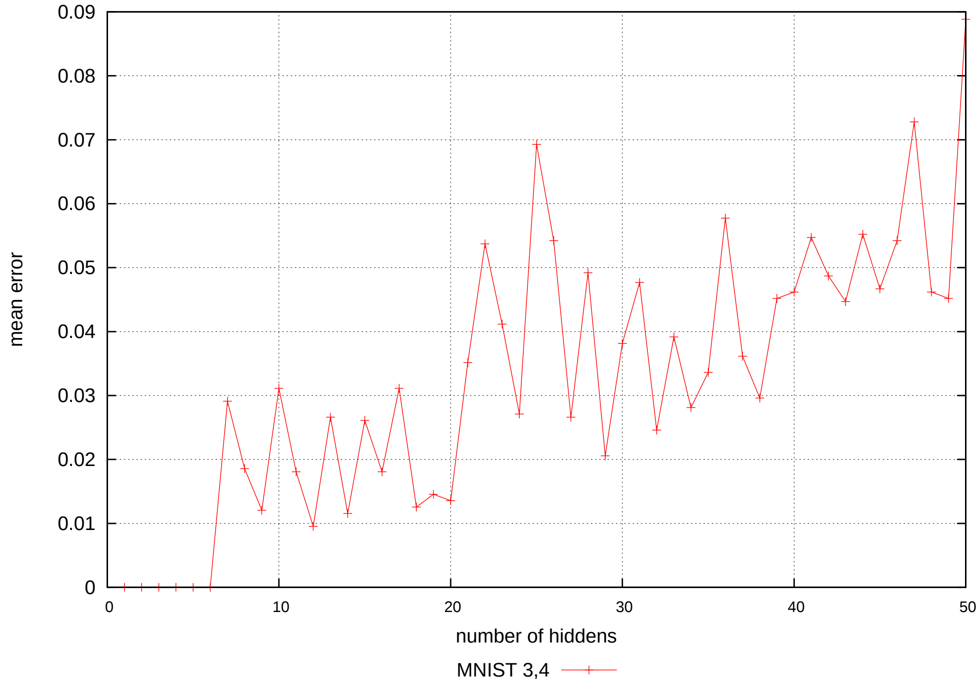
5.3 Results

We first observe what happens when the number of hidden nodes of the RBM is altered. A low number of hidden nodes provides a good starting point, because there is lower chance of inconsistent results. We use 15 epochs to train the network for every configuration of hidden nodes, and furthermore the batch size remains 20, with one Gibbs step per sample trained.

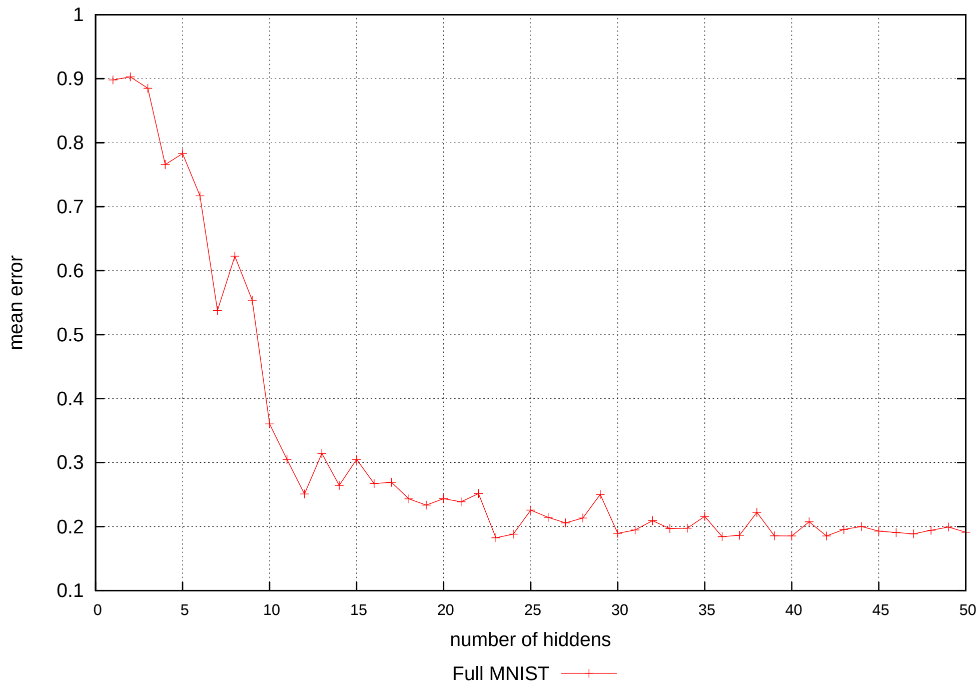
We define the mean error as the number of incorrect classifications divided by the number of testing samples. In the case of ordinal-numbered classifications, the number of errors includes both incorrect classification bits in the label vectors, as well as a missing bit for the correct label, so in some cases the error can even be twice the number of incorrect labels as a whole. We think that this is not a large problem, since we want to reduce both of these erroneous classifications. However, this might mean that the result seems worse than expected.

5.3.1 Classification in MNIST

In Figure 3a we see the outcome of limiting the MNIST data set to only use samples with classification label 3 or 4. We limit ourselves to 15 epochs of pretraining on the data set, and we vary the number of hidden nodes. As a result, the standalone RBM is able to train the model distribution precisely when it contains few hidden nodes. One of the nodes will likely become a feature detector for the most relevant parts of a digit 3, while the other can detect the usual shape of a 4. However, when we add more nodes, the network becomes more erratic, and although the error remains below 0.1, it is not helpful to use more than 2 hidden nodes in this case.



(a) MNIST data set filtered on digits 3 and 4



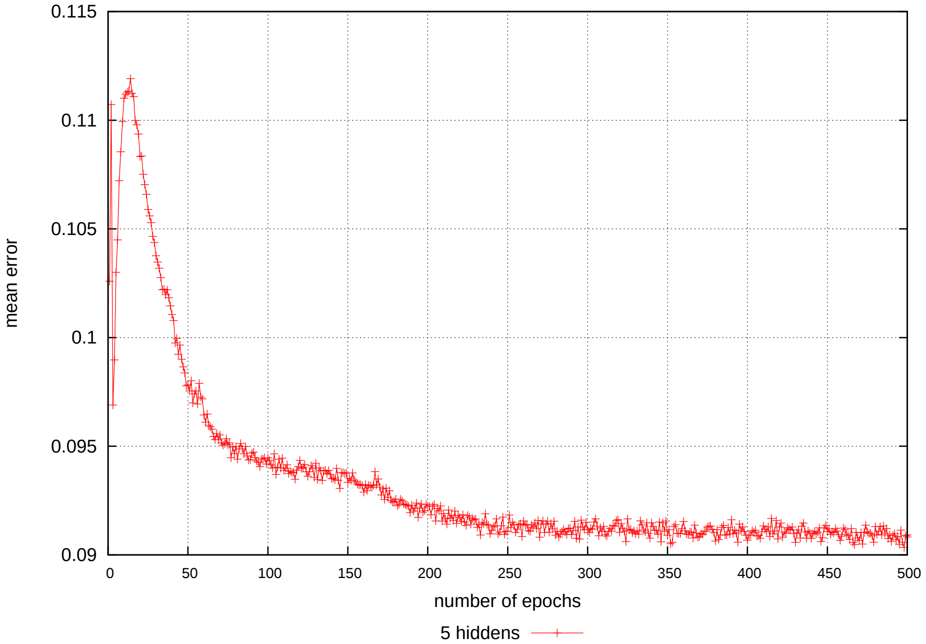
(b) Full MNIST data set

Figure 3: Comparison of RBM training on different targets from the MNIST data set. For the case where we have only two possible classifications, a small number of hidden nodes can perfectly represent the distribution. When there are more labels, we need an increasing number of nodes.

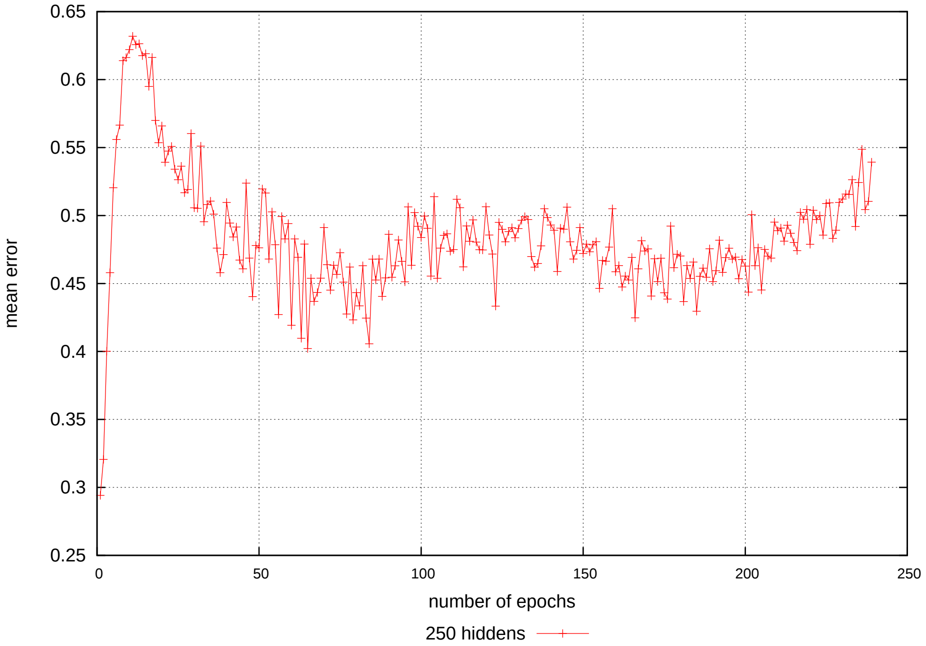
On the other hand, if we train on the full data set, including all digits 0, 1, 2, 3, 4, 5, 6, 7, 8, and 9, the number of hidden nodes plays are role in the mean error, as we can see in Figure 3b. The mean error starts off very high, with only a 10th of the test set being given a correct classification. This may be caused by the way we count errors in the classification's binary nodes. It is also what the Contrastive Divergence algorithm uses in its learning update rules, so it is a fair measure. With more hidden nodes, the RBM is able to decrease the mean error a lot. Using more than 50 hidden nodes is a worthwhile idea.

Now, it could also be the case that we need to train for a longer time, or with different training settings. We therefore investigate the relation between the number of epochs and the number of hidden nodes. In Figure 4a, we see the result of training runs of a Restricted Boltzmann Machine with 5 hidden nodes, for up to 500 epochs, with similar settings as in the previous experiment. The results have been averaged over 200 runs. Although there is some variance between each epoch, the mean error remains between 0.09 and 0.1 after some initial outliers.

The same experiment has been done with an almost unchanged RBM, but now with 250 hidden nodes. The result in Figure 4b shows a peculiarity in the first few epochs. The error is at around 0.325 after the first epoch, but it then grows toward 0.6. When we run another 50 epochs, the error decreases, but after that, it is unable to find any improvement. It can therefore be noted that a large number of epochs is not always a practical idea, but the number of hidden nodes might need some finetuning as well.



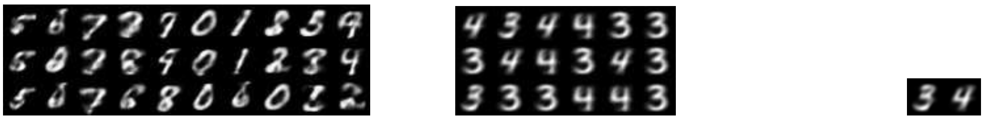
(a) RBM with 5 hidden nodes



(b) RBM with 250 hidden nodes

Figure 4: Comparison of RBM training on the MNIST data set with a varying number of hidden nodes and epochs.

The experiments focus on classification problems, because we can easily compare algorithms and parameters in this way. However, we can also investigate the use of Restricted Boltzmann Machines for generating images after unsupervised training. In Figure 5a, we see an example of a distribution that the network can generate. Figure 5b shows a similar result for the RBM that was used in the experiment with the data set filtered to only contain samples with classifications 3 and 4. Finally, Figure 5c shows two images of samples that are generated when set a classification bit in an empty image, so that the RBM generates a normalized sample with that classification.



(a) A distribution of images generated by the RBM on the MNIST data set after 100 epochs and with 20 hidden nodes.

(b) A distribution of images generated by the RBM on the data set filtered on 3 and 4 after 15 epochs and with 2 hidden nodes.

(c) Samples that are generated by the RBM on the MNIST data set filtered on 3 and 4, when given classification bits.

Figure 5: Comparison of examples of generated images on the MNIST data set.

5.3.2 Application to Poker

We want to study the performance of the Contrastive Divergence algorithm in an RBM when used on a more complex data set, such as a situation within a game. The Poker data set contains a large number of hands with five playing cards from which one can deduce a classification for the category the hand falls in [CO]. The Poker data set is introduced in Section 5.2.

In Figure 6, we compare the use of the normal learning rules of Contrastive Divergence against the alternative learning rule which we found in Section 4.2.2. The network trains with 25 hidden nodes, for 100 epochs and with default settings. The results were averaged over 100 runs. It is interesting to see that while the simplified CD algorithm is more erratic than the standard version, but its performance is similar.

We can further investigate how the Contrastive Divergence algorithm performs in an RBM by comparing it to other algorithms and networks. We can also compare it to the use of BackPropagation by building a simple DBN and performing a pretraining and training phase. In Figure 7a, we see the results of BackPropagation after a long pretraining phase using Contrastive Divergence on a Deep Belief Network with two intermediate layers of 100 and 1000 nodes. The error on the test set has decreased significantly, and the training phase is able to decrease it a little more. However, after a number of epochs, the error starts to rise slightly again, and it continues to rise in the epochs after that. The use of early-stopping can prevent the downturn by resetting the network to the state where it performed the best.

On the other hand, if we use the alternative CD learning rule which we found in Section 4.2.2, the pretraining phase ends up to perform about two times as worse as the normal rule, as seen in Figure 7b. The DBN network's BackPropagation training algorithm is unable to improve on this margin, and behaves erratically.

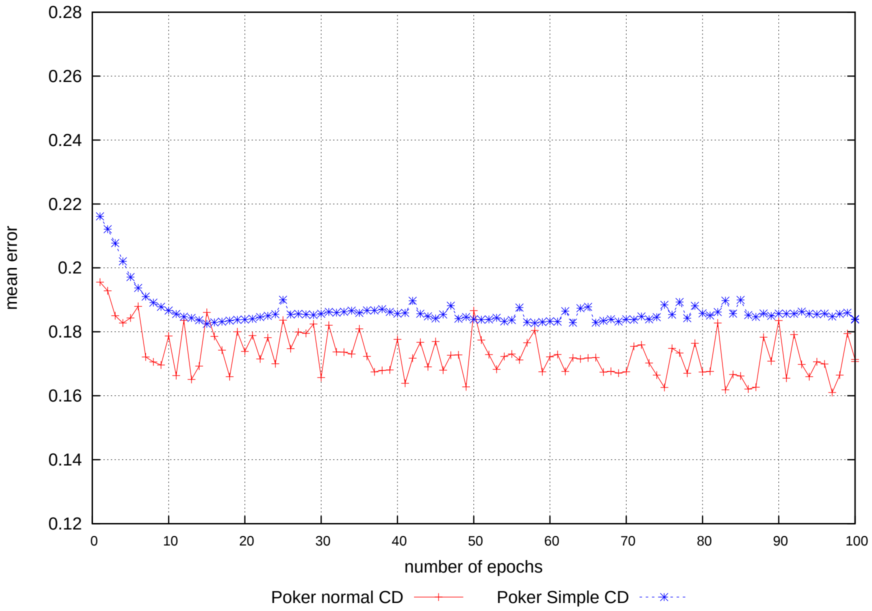


Figure 6: Comparison of RBM training on the Poker data set with a different learning rule.

6 Conclusion

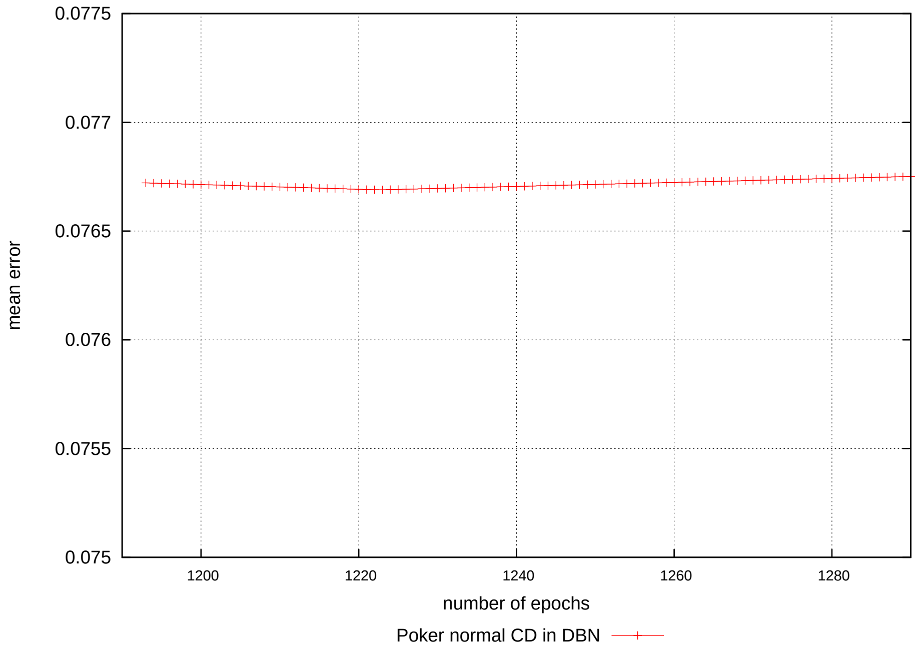
We have seen that Restricted Boltzmann Machines are already very able to model an underlying distribution of a data set so that it is able to provide accurate predictions for new samples. Even when we use it on its own, it is able to be trained supervised so that it can learn to generate classifications for inputs. It can also be trained unsupervised, but both learning methods allow the RBM to generate normalized samples of its model distribution.

The RBM can also be employed as components of a larger Deep Belief Network. We can pretrain the RBM components consecutively, and afterwards we can adjust the weights through the use of BackPropagation in a normal Multilayer Perceptron deep network.

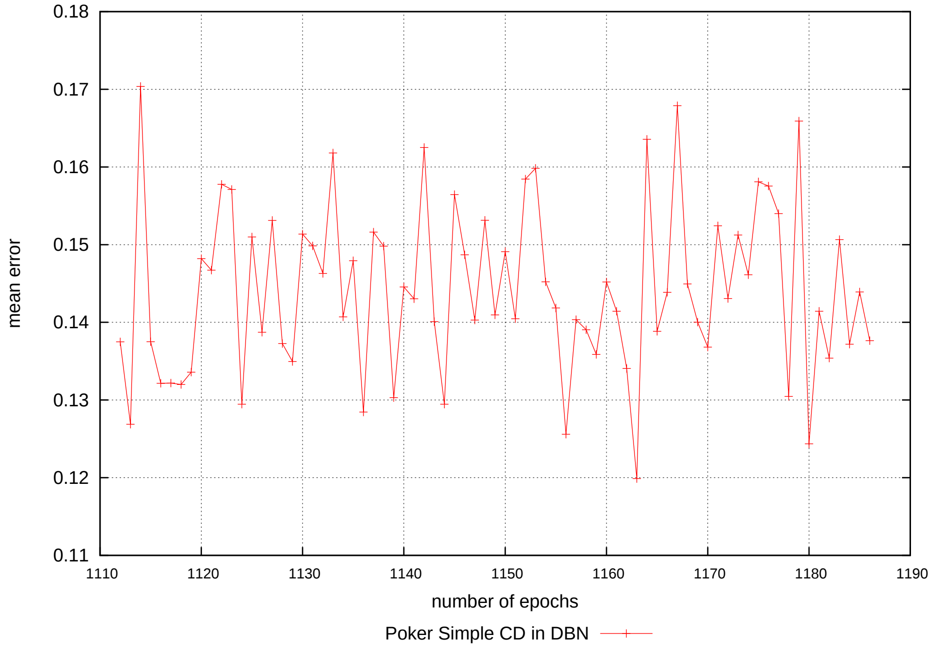
6.1 Further research

In this bachelor thesis, we investigated how the various deep learning network structures and algorithms work, and how they compare against each other. This also includes an exploration on whether they are applicable in fields such as game solving, graphics generation and predictions.

While the deep networks are able to train very well for those target purposes, there is still a problem of representation. Many data sets for games are structured in a different way, and often they are not compatible with the vector-based inputs of a neural network. They need to be preprocessed and parsed before they are able to be used in this way. It would be an interesting study to find out whether the deep networks can be adapted so that they can work with unprocessed data from these game databases. Specifically, it would be an interesting study to find out whether a Deep Belief Network can improve on playing Go by computers [SN08].



(a) DBN with normal CD



(b) DBN with Simple CD

Figure 7: Comparison of DBN trained on the Poker data set with different methods of Contrastive Divergence.

References

[ARK10] Itamar Arel, Derek C. Rose, and Thomas P. Karnowski. Deep machine learning: A new frontier in artificial intelligence research. *Computational Intelligence Magazine*, 5(4):13–18, 2010.

[BBB+10] James Bergstra, Olivier Breuleux, Frédéric Bastien, Pascal Lamblin, Razvan Pascanu, Guillaume Desjardins, Joseph Turian, David Warde-Farley, and Yoshua Bengio. Theano: A CPU and GPU math expression compiler. In *Proceedings of the Python for Scientific Computing Conference (SciPy)*, pages 3–10, 2010.

[BD09] Yoshua Bengio and Olivier Delalleau. Justifying and generalizing contrastive divergence. *Neural Computation*, 21(6):1601–1621, 2009.

[Ben09] Yoshua Bengio. Learning deep architectures for AI. *Foundations and Trends in Machine Learning*, 2(1):1–127, 2009.

[BL07] Yoshua Bengio and Yann LeCun. Scaling learning algorithms towards AI. In *Large-Scale Kernel Machines*. MIT Press, 2007.

[BLPL07] Yoshua Bengio, Pascal Lamblin, Dan Popovici, and Hugo Larochelle. Greedy layer-wise training of deep networks. In *Advances in Neural Information Processing Systems*, volume 19, pages 153–160. MIT Press, 2007.

[BW88] Eric B. Baum and Frank Wilczek. Supervised learning of probability distributions by neural networks. In *Advances in Neural Information Processing Systems*, volume 1, pages 52–61. MIT Press, 1988.

[CO] Roert Cattral and Franz Oppacher. Poker hand data set. *UCI Machine Learning Repository*. `https://archive.ics.uci.edu/ml/datasets/Poker+Hand`.

[CPH05] Miguel A. Carreira-Perpinan and Geoffrey E. Hinton. On contrastive divergence learning. In *Proceedings of the Tenth International Workshop on Artificial Intelligence and Statistics*, pages 33–40. Society for Artificial Intelligence and Statistics, 2005.

[FI14] Asja Fischer and Christian Igel. Training restricted Boltzmann machines: An introduction. *Pattern Recognition*, 47(1):25–39, 2014.

[GG84] Stuart Geman and Donald Geman. Stochastic relaxation, Gibbs distributions, and the Bayesian restoration of images. Number 6, pages 721–741. IEEE, 1984.

[GWFL+13] Ian J. Goodfellow, David Warde-Farley, Pascal Lamblin, Vincent Dumoulin, Mehdi Mirza, Razvan Pascanu, James Bergstra, Frédéric Bastien, and Yoshua Bengio. Pylearn2: A machine learning research library. *arXiv:1308.4214*, 2013.

[Hin02] Geoffrey E. Hinton. Training products of experts by minimizing contrastive divergence. *Neural Computation*, 14(8):1771–1800, 2002.

[Hin10] Geoffrey E. Hinton. A practical guide to training restricted Boltzmann machines. *Momentum*, 9(1):926, 2010.

[HL06] Fu Jie Huang and Yann LeCun. Large-scale learning with SVM and convolutional for generic object categorization. In *Computer Society Conference on Computer Vision and Pattern Recognition, IEEE*, volume 1, pages 284–291. IEEE, 2006.

[HOT06] Geoffrey E. Hinton, Simon Osindero, and Yee-Whye Teh. A fast learning algorithm for deep belief nets. *Neural Computation*, 18(7):1527–1554, 2006.

[HOWT06] Geoffrey Hinton, Simon Osindero, Max Welling, and Yee-Whye Teh. Unsupervised discovery of nonlinear structure using contrastive backpropagation. *Cognitive Science*, 30(4):725–731, 2006.

[HS06] Geoffrey E. Hinton and Ruslan R. Salakhutdinov. Reducing the dimensionality of data with neural networks. *Science*, 313(5786):504–507, 2006.

[KW12] Jyri J. Kivinen and Christopher Williams. Multiple texture Boltzmann machines. In *Proceedings of the International Conference on Artificial Intelligence and Statistics*, pages 638–646, 2012.

[LCB] Yann LeCun, Corinna Cortes, and Cristopher Burges. The MNIST database of handwiritten digits. `http://yann.lecun.com/exdb/mnist/`.

[MH05] Andriy Mnih and Geoffrey Hinton. Learning nonlinear constraints with contrastive backpropagation. In *Proceedings of the International Joint Conference on Neural Networks*, volume 2, pages 1302–1307. IEEE, 2005.

[RHW88] David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. Learning representations by back-propagating errors. In *Neurocomputing: Foundations of Research*, pages 696–699. MIT Press, Cambridge, MA, USA, 1988.

[RN10] Stuart J. Russell and Peter Norvig. *Artificial intelligence: A Modern Approach*, volume 3. Prentice Hall, 2010.

[SA04] Herb Sutter and Andrei Alexandrescu. *C++ Coding Standards: 101 Rules, Guidelines, and Best Practices*. C++ in Depth Series. Addison-Wesley, 2004.

[SN08] Ilya Sutskever and Vinod Nair. Mimicking Go experts with Convolutional Neural Networks. In *Artificial Neural Networks — ICANN 2008*, volume 5164 of *Lecture Notes in Computer Science*, pages 101–110. Springer, 2008.

[ST10] Ilya Sutskever and Tijmen Tieleman. On the convergence properties of contrastive divergence. In *Proceedings of the International Conference on Artificial Intelligence and Statistics*, pages 789–795, 2010.

[Tea13] Theano Development Team. DeepLearning documentation. 2008–2013. `http://deeplearning.net/tutorial/contents.html`.

[Tie08] Tijmen Tieleman. Training restricted Boltzmann machines using approximations to the likelihood gradient. In *Proceedings of the 25th International Conference on Machine Learning*, pages 1064–1071. ACM, 2008.

[TWOH03] Yee-Whye Teh, Max Welling, Simon Osindero, and Geoffrey E. Hinton. Energy-based models for sparse overcomplete representations. *The Journal of Machine Learning Research*, 4:1235–1260, 2003.

[Yui05] Alan L. Yuille. The convergence of contrastive divergences. In *Advances in Neural Information Processing Systems*, volume 17, pages 1593–1600. MIT Press, 2005.