DEGREE PROJECT, IN MATHEMATICAL STATISTICS , SECOND LEVEL
STOCKHOLM, *SWEDEN 2015*

Deep Learning for Multivariate Financial Time Series

GILBERTO BATRES-ESTRADA

KTH ROYAL INSTITUTE OF TECHNOLOGY

SCI SCHOOL OF ENGINEERING SCIENCES

Deep learning for multivariate financial time series

G I L B E R T O B A T R E S - E S T R A D A

Degree Project in Mathematical Statistics (30 ECTS credits)
Degree Programme in Engineering Physics (270 credits)
Royal Institute of Technology year 2015
Supervisor at Söderberg & Partners: Peng Zhou
Supervisor at KTH: Jonas Hallgren and Filip Lindskog
Examiner: Filip Lindskog

TRITA-MAT-E 2015:40
ISRN-KTH/MAT/E--15/40--SE

Royal Institute of Technology
School of Engineering Sciences

KTH SCI
SE-100 44 Stockholm, Sweden

URL: www.kth.se/sci

Abstract

Deep learning is a framework for training and modelling neural networks which recently have surpassed all conventional methods in many learning tasks, prominently image and voice recognition.

This thesis uses deep learning algorithms to forecast financial data. The deep learning framework is used to train a neural network. The deep neural network is a DBN coupled to a MLP. It is used to choose stocks to form portfolios. The portfolios have better returns than the median of the stocks forming the list. The stocks forming the S&P 500 are included in the study. The results obtained from the deep neural network are compared to benchmarks from a logistic regression network, a multilayer perceptron and a naive benchmark. The results obtained from the deep neural network are better and more stable than the benchmarks. The findings support that deep learning methods will find their way in finance due to their reliability and good performance.

Keywords: Back-Propagation Algorithm, Neural networks, Deep Belief Networks, Multilayer Perceptron, Deep Learning, Contrastive Divergence, Greedy Layer-wise Pre-training.

Acknowledgements

I would like to thank Söderberg & Partners, my supervisor Peng Zhou at Söderberg & Partners, my supervisor Jonas Hallgren and examiner Filip Lindskog at KTH Royal Institute of Technology for their support and guidance during the course of this interesting project.

Stockholm, May 2015

Gilberto Batres-Estrada

Contents

Chapter 1

Introduction

Deep learning is gaining a lot of popularity in the machine learning community and especially big technological companies such as Google inc, Microsoft and Facebook are investing in this area of research. Deep learning is a set of learning algorithms designed to train so called artificial neural networks an area of research in machine learning and artificial intelligence (AI). Neural networks are hard to train if they become too complex, e.g., networks with many layers, see the Chapter on neural networks. Deep learning is a framework facilitating training of deep neural networks with many hidden layers. This was not possible before its invention.

The main task of this master thesis is to use methods of deep learning, to compose portfolios. It is done by picking stocks according to a function learned by a deep neural network. This function will take values in the discrete set $\{0, 1\}$ representing a class label. The prediction task will be performed as a classification task, assigning a class or label to a stock depending on the past history of the stock, see Chapter 4.

We begin this work by presenting the background in which the formulation of the task to be solved is presented in detail, we then continue by presenting a short survey of the literature studied in order to understand the area of deep learning. We have tried to distinguish between the theory of neural networks' architecture and the theory on how to train them. Theory and architecture is given in Chapter 2, and the training of neural networks is presented in Chapter 3. In Chapter 4 the financial model is presented along with the assumptions made. In Chapter 5 we describe the experiments done and show the results from those experiments. The thesis concludes with Chapter 6 comprising a discussion of results and reflections about model choice and new paths to be taken in this area of research.

1.1 Background

Our artificial intelligent system will be constructed around neural networks. Neural networks, and in particular shallow networks, have been studied over the years to predict movements in financial markets and there are plenty of articles on the subject, see for instance (Kuo et al., 2014). In that paper there is a long reference list on the subject. We will use a type of neural network, called DBN. It is a type of stochastic learning machine which we connect to a multilayer perceptron (MLP).

The theory describing these networks is presented in the Chapter on neural networks. The task of predicting how financial markets evolve with time is an important subject of research and also a complex task. The reason is that stock prices move in a random way. Many factors could be attributed to this stochastic behaviour, most of them are complex and difficult to predict, but one that is certainly part of the explanation is human behaviour and psychology.

We rely on past experience and try to model the future empirically with the help of price history from the financial markets. In particular the historical returns are considered to be a good representation of future returns (Hult, Lindskog at al., 2012). Appropriate transformations of historical samples will produce samples of the random returns that determine the future portfolio values. If we consider returns to be identically distributed random variables then we can assume that the mechanisms that produced the returns in the past is the same mechanism behind returns produced in the future (Hult, Lindskog at al., 2012). We gather data from the financial markets and present it to our learning algorithm. The assumptions made are presented in the Chapter on financial model, Chapter 4. We chose to study the S&P 500.

1.2 Literature Survey

This work is based, besides computer experiments, on literature studies of both standard books in machine learning as well as papers on the subject of deep learning. An introduction to artificial neural networks can be found in The Elements of Statistical learning of Hastie, Tibshirani and Friedman, (Hastie and Friedman, 2009). Simon Haykin goes more in depth into the theory of neural networks in his book Neural Networks and Learning Machines where he also introduces some theory on deep learning (Haykin, 2009). Research on deep learning is focused mostly on tasks in artificial intelligence intending to make machines perform better on tasks such as vision recognition, speech recognition, etc. In many papers e.g. (Hinton et al., 2006),

(Larochelle et al., 2009) and (Erhan et al., 2010), tests are made on a set of handwritten digits, called MNIST and is a standard for comparison of results among researchers to see if their algorithms perform better than other's.

A classic paper is (Hinton et al., 2006), on training DBN. They show that their test error is only 1.25%. The second best result in the same article to which they compare their results has a test error of 1.40% achieved with a support vector machine. The test error measures how well the model generalises or how good its prediction power is to new unseen data. This terminology is explained more in depth in later Chapters. For training the networks used in this thesis the suggestions given in the the technical report (Hinton, 2010) on how to train RBMs where helpful, because RBMs are the building blocks of DBNs. The theory of DBN is presented for example in (Bengio, 2009), (Haykin, 2009) and (Salakhutdinov, 2009).

Relevant work on deep learning applied to finance was found in (Takeuchi et al., 2013) where they make a similar study as the one presented here but in that report they used an auto encoder, a different type of network with training and test data from the S&P 500, the Nasdaq Composite and the AMEX lists. Their test error was around 46.2%. In (Zhu and Yin, 2014) deep learning methods were used to construct a stock decision support system based on DBN with training and test data from the S&P 500. They draw the conclusion that their system outperforms the buy and hold strategy. Other approaches to deep learning on tasks such as regression on chaotic time series are presented in (Kuremoto et al., 2014) and (Kuremoto et al., 2014), where in the first paper only a DBN is used and in the second a DBN-MLP is used and both papers show good results.

The next Chapter introduces the theory of artificial neural networks. It begins with single layer neural networks and continues by presenting the theory of multilayer neural networks and concludes with the theory of DBN.

Chapter 2

Neural Networks

This Chapter begins with the theory of artificial neurons, the building blocks of neural networks. Then we present the theory step by step by introducing single layer networks and continue with multilayer neural networks. The Chapter finishes with deep neural networks.

The research on artificial neural networks was inspired by the research and models of how the brain works in humans and other mammals (Haykin, 2009). Researchers think of the human brain as a highly complex, nonlinear and parallel computer or information processing system capable of performing highly complex tasks. It is a fact that the brain is composed of cells called neurons. These neurons are responsible for performing complex computations as pattern recognition, perception or control, to name a few, faster than the fastest digital computers available today.

For the human brain it can take milliseconds to recognise a familiar face. The human brain learns new tasks and how to solve problems through experience and adapts to new circumstances. Because of this adaptation the brain is considered to be plastic, which means that neurons learn to make new connections. Plasticity appears to be essential for the functioning of neurons as information processing units in the human brain. This also appears to be the case for the neural networks made up of artificial neurons (Haykin, 2009).

A neural network is thought of as a machine designed to model how the brain performs a particular task. A neural network is built up by a network of computing units, known as neurons. These computing units are represented as nodes in the network and they are connected with each other through weights. For formal definition of neural networks we refer to the appendix.

2.1 Single Layer Neural Network

We begin this section by introducing the theory of artificial neurons and activation functions, the building blocks of neural networks.

2.1.1 Artificial Neurons

The computing units that are part of a neural network are called artificial neurons or for short just neurons. The block diagram of Figure 2.1 shows a model of an artificial neuron. The neural model is composed of the following building blocks:

1. A set of synapses or connection links, each characterized by a weight or strength. A signal x_j at the input of synapse j connected to neuron k is multiplied by the synaptic weigh w_k.

2. An adder for summing the input signals, weighted by the respective synaptic strengths of the neuron. The operations here constitute a linear combiner.

3. An activation function, $\varphi(\cdot)$, for limiting the amplitude of the output of a neuron.

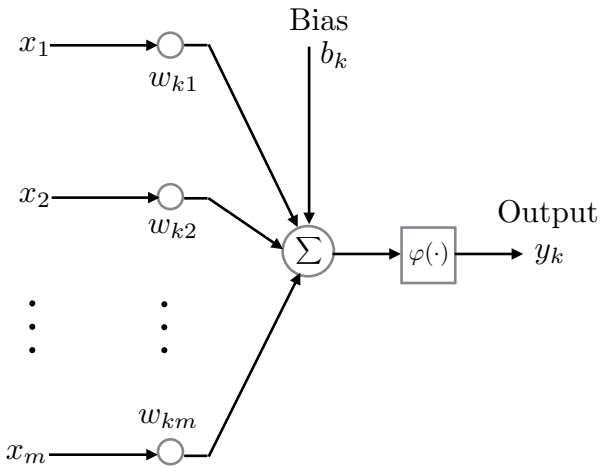

Figure 2.1: *Nonlinear model of a neuron.*

In the neural model presented in Figure 2.1 we can see a bias, b, applied to the network. The effect of the bias is to decrease or increase the net input of the activation function depending on whether it is negative or positive.

More about the activation function is presented in the next section. A mathematical representation of the neural network in Figure 2.1 is given by the equations:

$$\begin{aligned} u_k &= \sum_{j=1}^{m} w_{kj} x_j, \\ y_k &= \varphi(u_k + b_k), \end{aligned} \tag{2.1}$$

where $x_1, x_2, \ldots, x_m$ are the input signals and $w_1, w_2, \ldots, w_m$ are the respective synaptic weights of neuron k. The output of the neuron is y_k and u_k represents the linear combiner output due to the input signals. The bias is denoted b_k and the activation function by φ. The bias b_k is an affine transformation to the output u_k of the linear combiner. We now define the *induced local field* or *activation potential* as:

$$v_k = u_k + b_k. \tag{2.2}$$

Putting all the components of the neural network into a more compact form we end up with the following equations:

$$v_k = \sum_{j=0}^{m} w_{kj} x_j, \tag{2.3}$$

and

$$y_k = \varphi(v_k), \tag{2.4}$$

where we now have added a new synapse with input $x_0 = 1$ and weight $w_{k0} = b_k$ accounting for the bias. In Figure 2.2 we have added the bias as a fixed input signal, $x_0 = 1$, of weight $w = b = 1$ showing how (2.3) can be interpreted.

2.1.2 Activation Function

The activation function, $\varphi(v)$, defines the output of a neuron in terms of the induced local field v. In this section we present some activation functions, the threshold, the sigmoid function and the linear rectifier.

1. **Threshold Function**. The threshold function is depicted in Figure 3.5 and it is defined as:

$$\varphi(v) = \begin{cases} 1 & \text{if} : v \geq 0 \\ 0 & \text{if} : v < 0 \end{cases}$$

The threshold function is commonly referred to as Heaviside function. The output of a neuron k using a threshold function as activation function is

$$y_k = \begin{cases} 1 & \text{if} : v_k \geq 0 \\ 0 & \text{if} : v_k < 0 \end{cases}$$

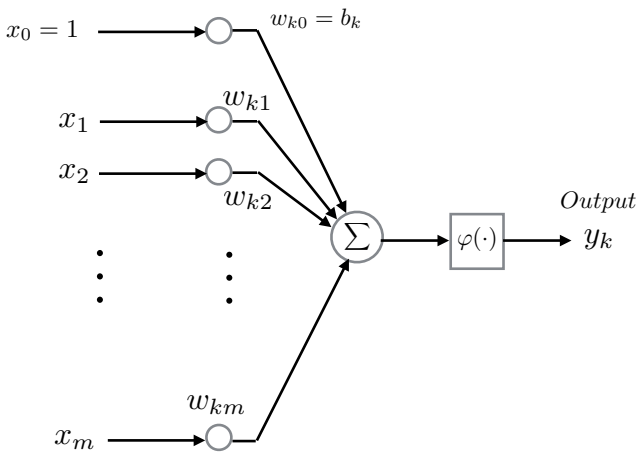

Figure 2.2: *Neural network with bias as input,* $x_0 = +1$*, and weight* $w_{k0} = b_k$*.*

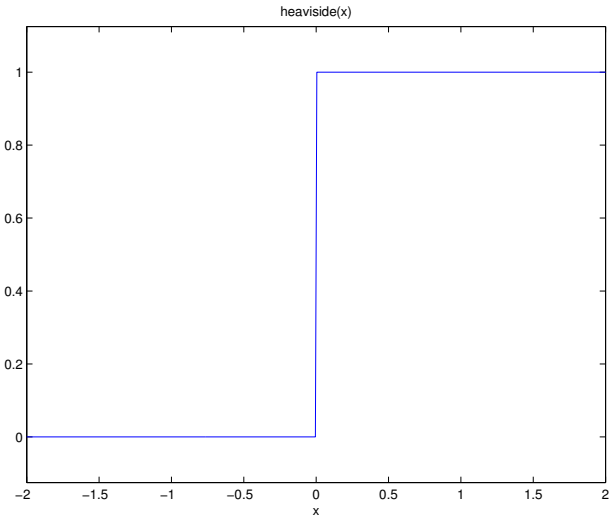

Figure 2.3: *Threshold function.*

where v_k is the induced local field of the neuron, that is

$$v_k = \sum_{j=1}^{m} w_{kj} x_j + b_k. \tag{2.5}$$

This model is called the McCulloch-Pitts model. The output of the neuron takes the value 1 if the induced local field is non-negative and 0 otherwise.

2. **Sigmoid Function**. The sigmoid function is by far the most common form of activation function used in modelling neural networks. Most of the articles cited in this work make use of the sigmoid function. It is defined as a strictly increasing function that exhibits a balance between a linear and nonlinear behaviour. An example of the sigmoid function is the logistic function whose graph is shown in Figure 2.4 for different values of a and its mathematical formula is:

$$\varphi(v) = \frac{1}{1 + \exp(-av)}. \tag{2.6}$$

The parameter a is the slope parameter. One of the most interesting properties of the sigmoid function is that it is differentiable, a property very useful when training neural networks.

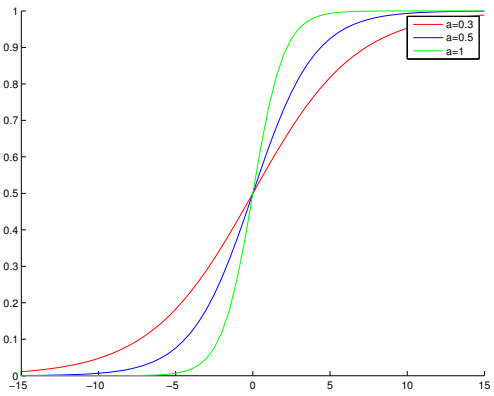

Figure 2.4: *Sigmoid function for* $a = \{0.3, 0.5, 1.0\}$.

3. **Rectified Linear Units**. A rectified linear unit is a more interesting model of real neurons, (Nair and Hinton, 2010). It is constructed by making a large number of copies from the sigmoid. This is done under the assumption that all copies have the same learned weights and biases.

The biases have different fixed offsets which are $-0.5, -1.5, -2.5, \ldots$. Then the sum of the probabilities is given by

$$\sum_{i=1}^{N} \varphi(v - i + 0.5) \approx \log(1 + e^{v}), \tag{2.7}$$

where $v = \mathbf{w}^T\mathbf{x} + b$. A drawback of this model is that the sigmoid function has to be used many times to get the probabilities required for sampling an integer value correctly. A fast approximation is

$$\varphi(v) = \max\Big(0, v + N\big(0, \varphi(v)\big)\Big), \tag{2.8}$$

where $N(0, \sigma)$ is a Gaussian noise with zero mean and variance σ.

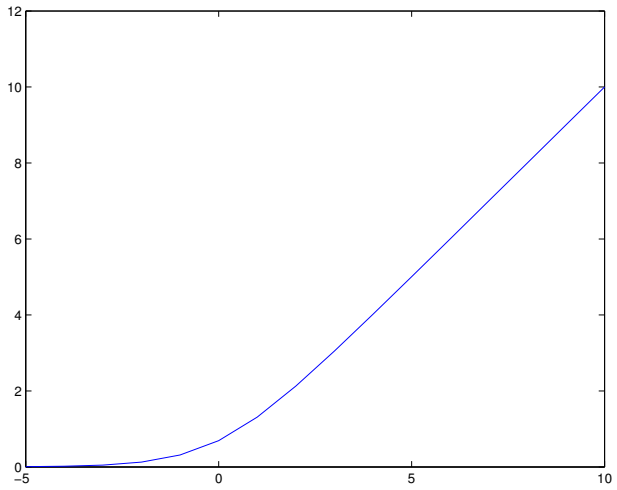

Figure 2.5: *Rectified linear units. The figure shows* $\log(1 + e^{x})$.

4. **Odd Activation Functions**. It is worth noting in these examples of activation functions, that they range from 0 to +1. In some situations it is preferable to have an activation function whose range is from −1 to +1 in which case the activation function is an odd function of the local field. The threshold function ranging from −1 to +1 which is known as the signum function is:

$$\varphi(v) = \begin{cases} 1 & \text{if} : v \geq 0 \\ 0 & \text{if} : v = 0 \\ -1 & \text{if} : v < 0 \end{cases}$$

and the corresponding form of the sigmoid is the hyperbolic tangent function defined by

$$\varphi(v) = \tanh(v). \tag{2.9}$$

So far we have been talking about deterministic neural models, where the input-output behaviour is accurately defined for all inputs. Sometimes a stochastic model is needed. For the McCulloch-Pitts model we could for instance give the activation function a probabilistic interpretation by assigning probabilities for certain states. We can allow this network to attain two states $+1$ or -1 with probabilities $p(v)$ and $1 - p(v)$ respectively, then this is expressed as

$$x = \begin{cases} +1 & \text{with probability} : p(v) \\ -1 & \text{with probability} : 1 - p(v) \end{cases}$$

where x is the state of the system and $p(v)$ is given as

$$p(v) = \frac{1}{1 + \exp(-v/T)}, \tag{2.10}$$

where T is a parameter used to control the noise level thus controlling the uncertainty in changing the state. In this model we can think of the network making a probabilistic decision to react by changing states from off to on.

2.1.3 Single-Layer Feedforward Networks

In the single-layer network there is an input layer of source nodes which direct data onto an output layer of neurons. This is the definition of a feedforward network where data goes from the input to the output layer but no the other way around. Figure 2.6 shows a single-layer feedforward network.

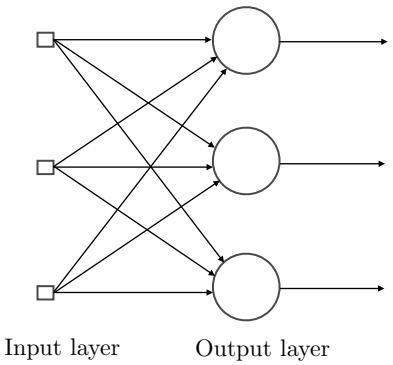

Figure 2.6: *Single-Layer Feedforward Network.*

2.1.4 The Perceptron

The Perceptron was the first model of a neural network. Rosenblatt, a psychologist, published his paper (Rosenblatt, 1957) on the perceptron. This paper was the starting point in the area of supervised learning. Engineers, physicists and mathematicians got interested in neural networks and began to use those theories in their respective fields in the 1960s and 1970s, (Haykin, 2009). McCulloch and Pitts (1943) introduced the idea of neural networks as computing machines, Hebb (1949) postulated the first rule for self-organised learning. Rosenblatt's perceptron is built around a nonlinear neuron, that of the McCulloch-Pitts model, Figure 2.1.

2.1.5 The Perceptron As a Classifier

The perceptron consists of a linear combiner followed by a hard limiter, $\varphi(\cdot)$, performing the signum function. The summing node in the neural network computes a linear combination of the inputs applied to its synapses, and it incorporates a bias. The resulting sum is applied to the hard limiter. The neuron then produces an output equal to $+1$ if the hard limiter input is positive, and -1 if it is negative.

In the neuron model of Figure 2.1 we have m input signals, $x_1, x_2, \ldots, x_m$, and m weights, $w_1, w_2, \ldots, w_m$. Calling the externally applied bias b we see that the hard limiter input or induced local field of the neuron is

$$v = \sum_{i=1}^{m} w_i x_i + b. \tag{2.11}$$

A perceptron can correctly classify the set of externally applied input signals $x_1, x_2, \ldots, x_m$ into one of two classes $\mathscr{C}_1$ or $\mathscr{C}_2$. The decision for the classification is based on the resulting value of y corresponding the to input signal $x_1, x_2, \ldots, x_m$ where $y = 1$ belongs to $\mathscr{C}_1$ and $y = -1$ belongs to $\mathscr{C}_2$.

To gather insight in the behaviour of a pattern classifier, it is customary to plot a map of the decision regions in a m-dimensional signal space spanned by the input variables $x_1, x_2, \ldots, x_m$. The simplest case of a perceptron network consists of two decision regions separated by a *hyperplane* which is defined by

$$\sum_{i=1}^{m} w_i x_i + b = 0. \tag{2.12}$$

This is illustrated in Figure 2.7 for the case of two variables x_1, x_2 for which the decision line is a straight line. A point (x_1, x_2) lying in the region above the boundary line is classified as belonging to class one, $\mathscr{C}_1$, and a point (x_1, x_2) lying below the boundary to class two, $\mathscr{C}_2$. We can see from the

figure that the effect of the bias is merely to shift the boundary region away from the origin. The synaptic weights of the perceptron are adapted by

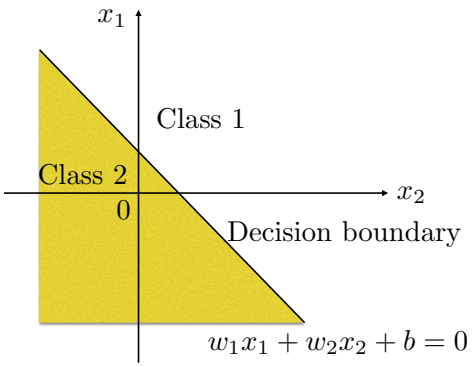

Figure 2.7: *Hyperplane as a decision boundary for a 2-dimensional, 2-class pattern-classification problem.*

iteration. This is done by using an error-correction rule, which is called the perceptron convergence algorithm. Adopting a more general setting, we could define the bias as a synaptic weight driven by a fixed input equal to 1. This situation is similar to the one depicted in Figure 2.1. We define the $(m+1)$-by-1 input vector

$$\mathbf{x}(n) = [1, x_1(n), x_2(n), \ldots, x_m(n)]^T, \tag{2.13}$$

where n is the time step and T denotes transposition. In the same manner we define the $(m+1)$-by-1 weight vector as

$$\mathbf{w}(n) = [b, w_1(n), w_2(n), \ldots, w_m(n)]^T. \tag{2.14}$$

Then the linear output combiner can be written as

$$\begin{aligned} v(n) &= \sum_{i=0}^{m} w_i(n)x_i(n) \\ &= \mathbf{w}^T(n)\mathbf{x}(n), \end{aligned} \tag{2.15}$$

where in the first line $w_0(n) = b$, is the bias. Fixing n and plotting $\mathbf{w}^T\mathbf{x} = 0$ in a m-dimensional space with coordinates $x_1, x_2, \ldots, x_m$, the boundary surface is a hyperplane defining a decision surface between two classes.

For the perceptron to function properly, the two classes $\mathscr{C}_1$ and $\mathscr{C}_2$ must be linearly separable. This means that the two patterns to be classified must

be sufficiently separated from each other to ensure that the decision surface consists of a hyperplane. An example of linearly separable patterns is shown in Figure 2.8 in the two-dimensional case, while an example of non separable patterns is shown in Figure 2.9. The perceptron's computation ability is

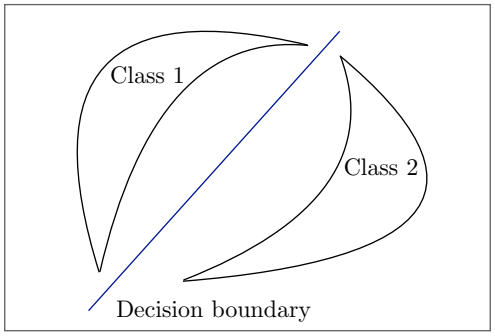

Figure 2.8: *Figure showing separable patterns. Note that the two regions can be separated by a linear function.*

not enough to handle the non separable case. Assume therefore that we have two linearly separable classes. Define $\mathscr{H}_1$ as the subspace of training vectors $\mathbf{x}_1(1), \mathbf{x}_1(n), \ldots$ that belong to class $\mathscr{C}_1$ and $\mathscr{H}_2$ as the subspace of training vectors $\mathbf{x}_2(1), \mathbf{x}_2(n), \ldots$ that belong to $\mathscr{C}_2$. Then $\mathscr{H} = \mathscr{H}_1 \cup \mathscr{H}_2$ is the complete space. With these definitions we can state the classification problem as

$$\begin{aligned} \mathbf{w}^T\mathbf{x} > 0 \quad & \text{for every input vector } \mathbf{x} \text{ belonging to class } \mathscr{C}_1, \\ \mathbf{w}^T\mathbf{x} \leq 0 \quad & \text{for every input vector } \mathbf{x} \text{ belonging to class } \mathscr{C}_2. \end{aligned} \tag{2.16}$$

In the second line we see that $\mathbf{w}^T\mathbf{x} = 0$ belongs to class $\mathscr{C}_2$. The classification problem for the perceptron consists of finding a weight vector $\mathbf{w}$ satisfying the relation 2.16. This can be formulated as follows

1. If the nth member of the training set $\mathbf{x}(n)$ is correctly classified by the weight vector $\mathbf{w}(n)$ then no correction is made to the weight of the perceptron according to

$$\begin{aligned} \mathbf{w}(n+1) = \mathbf{w}(n) \quad & \text{if } \mathbf{w}^T\mathbf{x} > 0 \text{ belongs to class } \mathscr{C}_1, \\ \mathbf{w}(n+1) = \mathbf{w}(n) \quad & \text{if } \mathbf{w}^T\mathbf{x} \leq 0 \text{ belongs to class } \mathscr{C}_2. \end{aligned}$$

2. Otherwise, the weight vector is updated according to the rule

$$\begin{aligned} \mathbf{w}(n+1) = \mathbf{w}(n) - \eta(n)\mathbf{w}(n) \quad & \text{if } \mathbf{w}^T\mathbf{x} > 0 \text{ belongs to class } \mathscr{C}_1, \\ \mathbf{w}(n+1) = \mathbf{w}(n) - \eta(n)\mathbf{w}(n) \quad & \text{if } \mathbf{w}^T\mathbf{x} \leq 0 \text{ belongs to class } \mathscr{C}_2, \end{aligned}$$

where the *learning-rate* parameter $\eta(n)$ controls the adjustment applied to the weight vector at time-step n.

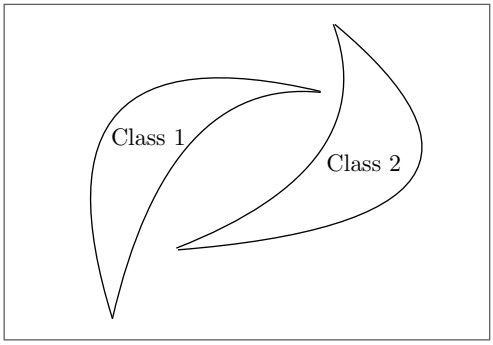

Figure 2.9: *Figure showing non separable patterns. Here it is not longer possible to separated both regions by a linear function.*

2.2 Multilayer Neural Networks

Multilayer networks are different from single-layer feedforward networks in one respect, (Haykin, 2009). The multilayer network has one or more hidden layers, whose computation nodes are called hidden neurons or hidden units. The term hidden is used because those layers are not seen from either the input or the output layers. The task of these hidden units is to be part in the analysis of data flowing between the input and output layers. By adding one or more hidden layers the network can be capable of extracting higher order statistics from its input.

The input signal is passed through the first hidden layer for computation. The resulting signal is then an input signal to the next hidden layer. This procedure continues if there are many hidden layers until the signal reaches the output layer in which case it is considered to be the total response of the network. Figure 2.10 shows an example of a multilayer feedforward network with 3 input units, 3 hidden units and 1 output unit. Because of this structure the networks is referred to as a $3-3-1$ network.

2.2.1 The Multilayer Perceptron

The Multilayer Perceptron (MLP) consist of neurons whose activation functions are differentiable (Haykin, 2009). The network consists of one or more hidden layers and have a high degree of connectivity which is determined by

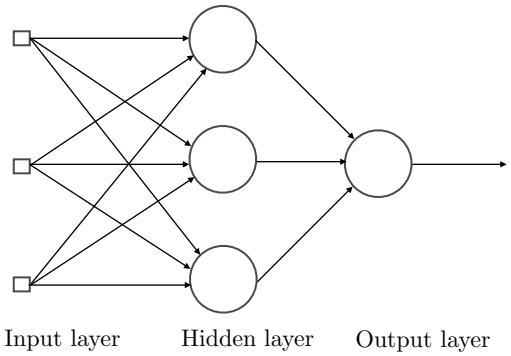

Figure 2.10: *Fully connected feedforward network with one hidden layer and one output layer.*

synaptic weights. Much of the difficulty in analysing these networks resides in the nonlinearity and high connectivity of its neurons. Added to these difficulties is the problem of learning when the network has hidden units.

When working with multilayer perceptrons we will use the term function signal to mean input signal. The function signal is the output of a neuron in a preceding layer in the direction from input to output. These signals are then passed to other neurons which receive them as input. The error signal comes at the output units and is propagated backward layer by layer. Each hidden or output neuron of the multilayer perceptron perform the following computations

1. Computation of the function signal appearing at the output of each neuron,

2. Estimation of the gradient vector, which is used in the backward pass through the network.

The hidden neurons act as *feature detectors*, as the learning process progresses the hidden neurons discover silent features in the training data. The hidden units perform a nonlinear transformation on the input data into a new space, called the *feature space*. In the feature space, the classes of interest in a pattern classification problem, may be more easily classified than it would be in the original input space.

2.2.2 Function Approximation with MLP

The multilayer perceptron can perform nonlinear input-output mapping and is therefore suitable for the approximation of a function $f(x_1, \ldots, x_{m_0})$. If

m_0 is the number of input nodes and $M = m_L$ the number of neurons in the output layer then the input-output relationship of the network defines a mapping from an m_0-dimensional Euclidean input space to an M-dimensional Euclidean output space. The universal approximation theorem gives us the minimum number of hidden layers required in the approximation of a function $f(x_1, \ldots, x_{m_0})$. The theorem is as follows, (Haykin, 2009):

Theorem 2.1 *Let $\varphi(\cdot)$ be a non-constant, bounded and monotone-increasing continuous function. Let I_{m_0} be the m_0-dimensional unit hypercube $[0,1]^{m_0}$. The space of continuous functions on I_{m_0} is denoted by $C(I_{m_0})$. Then, given any function $f \in C(I_{m_0})$ and $\epsilon > 0$, then there exists an integer m_1 and sets of real constants α_i, b_i and w_{ij}, where $i = 1, \ldots, m_1$ and $j = 1, \ldots, m_0$ such that we may define*

$$F(x_1, \ldots, x_{m_0}) = \sum_{i=1}^{m_1} \alpha_i \varphi\Big(\sum_{j=1}^{m_0} w_{ij} x_j + b_i \Big), \tag{2.17}$$

as an approximate realization of the function $f(\cdot)$, that is

$$|F(x_1, \ldots, x_{m_0}) - f(x_1, \ldots, x_{m_0})| < \epsilon,$$

for all $x_1, x_2, \ldots, x_{m_0}$ that lie in the input space.

Any sigmoid function meets the requirements of the theorem and can be used in the approximation procedure. The output of a multilayer perceptron such as that given by (2.17) has the following features

1. The network has m_0 input nodes and a single hidden layer with m_1 neurons. The inputs in this equation are given by $x_1, \ldots, x_{m_0}$.
2. Hidden neuron i has weights $w_{i_1}, \ldots, w_{m_0}$ and bias b_i.
3. The output is a linear combination of outputs of hidden neurons, with $\alpha_1, \ldots, \alpha_{m_1}$ as synaptic weights of the output layer.

The universal approximation theorem allows us to use neural networks, under the assumptions of the theorem, to approximate functions. Therefore neural networks are suitable for regression tasks as well as classification tasks.

2.2.3 Regression and Classification

Neural networks are often used to solve classification as well as regression problems. As the universal approximation theorem tells us how to use neural networks for approximation of functions we see that it is suitable for regression tasks. We also saw how the perceptron can be used for classification. On a more general footing we can say that the only difference between those

tasks is how we represent the output from the network and how the cost function is defined. For example if the output from the network is $f_k(X)$ then we can formulate the classification or regression solution as

$$\begin{aligned} Z_m &= \varphi(\alpha_{0m} + \alpha_m^T X), \quad m = 1, 2, \ldots, M \\ T_k &= \beta_{0k} + \beta_k^T Z, \quad Z = (Z_1, Z_2, \ldots, Z_M), \\ f_k(X) &= g_k(X), \quad k = 1, 2, \ldots, K, \quad T = (T_1, T_2, \ldots, T_K), \end{aligned}$$

where Z_m are the m hidden units in the network, $\varphi(\cdot)$ is the activation function, often of sigmoid type, and $f_k(X)$ is the response of the network due to the input X. For the regression problem we have the relation $g_k(T) = T_k$ and for the classification problem we have the *softmax* function

$$g_k(T) = \frac{e^{T_k}}{\sum_{l=1}^{K} e^{T_l}}. \tag{2.18}$$

In the learning process, there is a parameter estimation step, where one has to estimate the synaptic weights. The estimation is done by back-propagation and its objective is to minimize a cost function. The procedure is presented in the Chapter on training neural networks. For the regression problem we have a cost function that looks like

$$\mathcal{E}(\theta) = \sum_{k=1}^{K} \sum_{i=1}^{N} \Big(y_{ik} - f_k(x_i; \theta) \Big)^2, \tag{2.19}$$

where y_{ik} is the desired response, $\theta = \{\alpha_{ml}, \beta_{km}\}$ is the set of parameters of the model and $f_k(x_i; \theta)$ the actual response of the network. For the classification problem the cost function, sometimes called the cross entropy, is

$$\mathcal{E}(\theta) = -\sum_{k=1}^{K} \sum_{i=1}^{N} y_{ik} \log f_k(x_i; \theta). \tag{2.20}$$

2.2.4 Deep Architectures

Deep neural networks are built around a deep architecture in which there are more than one layer, often with many hidden layers. Here we present the theory for deep or multilayer neural networks.

We refer to the depth of a neural network as the number of levels or layers it comprises. Each layer performs non-linear operations on the input data in order to learn the function under study. Classical networks are considered to have shallow architectures often composed of $1, 2$ or 3 layers. For a long time, researchers tried to train neural networks consisting of deep architectures but often reached a boundary where only networks with a depth at

most 3 layers (with two hidden layers), were possible to train (Bengio, 2009). In (Hinton, 2006), it is shown how to train a DBN with the help of a greedy learning algorithm that trains a layer at a time. The principle used in deep learning is to train the intermediate levels using unsupervised learning, which is performed locally at each level. More on the theory on how to train deep neural networks will be covered in the Chapter on training neural networks.

Deep neural networks are trained in an unsupervised manner but in many applications they are also used to initialise feedforward neural networks. Once a good representation has been found at each level, it can be used to initialise and train a deep neural network by supervised gradient-based optimization.

In machine learning we speak of labeled and unlabelled data, where the first refers to the case where we have a target or known result to which our network output can be compered. In the second case we do not have such a target. With deep learning algorithms it is possible to train deep neural networks without labels or target values. Deep learning algorithms are supposed to handle the following issues, (Bengio, 2009):

- Ability to learn complex, highly varying functions, with a number of variations much greater than the number of training examples.
- To learn with little human input the low-level, intermediate and high-level abstractions in the input data.
- Computation time should scale well with the number of examples and be close to linear.
- To learn from mostly unlabelled data and work in the semi-supervised setting, where examples may have incorrect labels.
- Strong unsupervised learning which captures most of the statistical structure in the observed data.

Deep architectures are needed for the representation of complicated functions, a task which can be difficult to handle for shallow architectures. The expression of a function is considered to be *compact* if it has few degrees of freedom that need to be tuned by learning. Functions that can be compactly represented by an architecture of depth k may require an exponential number of computational elements to be represented by a depth $k-1$ architecture. The number of computing units consume a lot of resources and is dependent on the number of training examples available. This dependence leads to both computational and statistical performance problems resulting in poor generalization. As we shall see this can be remedied by a deep architecture. The depth of a network makes is possible to use less computing neurons in

the network. The depth is defined as the number of layers from the input node to the output node.

An interesting result is that it is not the absolute number of layers that is important but instead the number of layers relative to how many are required to represent efficiently the target function.

The next theorem states how many computing units that are needed for representing a function when the network has depth k, (Håstad & Goldmann, 1991), (Bengio, 2009).

Theorem 2.2 *A monotone weighted threshold network of depth $k-1$ computing a function $f_k \in \mathcal{F}_{k,N}$ has size at least 2^{cN} for some constant $c > 0$ and $N > N_0$.*

Here $\mathcal{F}_{k,N}$ is the class of functions where each function has N^{2k-2} inputs, defined by a k depth tree. The interpretation of the theorem is that there is no right depth for a network but instead the data under study should help us to decide the depth of the network. The depth of the network is coupled to what degree a function under study varies. In general, highly varying functions require deep architectures to represent them in a compact manner. Shallow architectures would require a big number of computing units if the network has an inappropriate architecture.

For a deep neural network, we denote the output of a neuron at layer k by $\mathbf{h}^k$, and its input vector by $\mathbf{h}^{k-1}$ coming from the preceding layer, then we have

$$\mathbf{h}^k = \varphi(\mathbf{b}^k + W^k\mathbf{h}^{k-1}), \tag{2.21}$$

where $\mathbf{b}^k$ is a vector of offsets or biases, W^k is a matrix of weights and $\varphi(\cdot)$ is the activation function, which is applied element-wise. At the input layer, the input vector, $\mathbf{x} = \mathbf{h}^0$, is the raw data to be analysed by the network. The output vector $\mathbf{h}^\ell$ in the output layer is used to make predictions. For the classification task we have the output, called the softmax

$$\mathbf{h}_i^\ell = \frac{\exp(\mathbf{b}_i^\ell + W_i^\ell\mathbf{h}^{\ell-1})}{\sum_j \exp(\mathbf{b}_j^\ell + W_j^\ell\mathbf{h}^{\ell-1})}, \tag{2.22}$$

where W_i^ℓ is the ith row of W^ℓ, $\mathbf{h}_i^\ell$ is positive and $\sum_i \mathbf{h}_i^\ell = 1$. For the regression task the output is

$$\mathbf{h}_i^\ell = \alpha_{0k} + \alpha_k\varphi(\mathbf{b}_i^\ell + W_i^\ell\mathbf{h}^{\ell-1}), \tag{2.23}$$

a definition we have already seen in past sections with α_{0k} as the bias applied to the last output layer and α_k representing a set of weights between the last and next to last layers. The outputs are used together with the target function or labels y into a loss or cost function $\mathcal{E}(\mathbf{h}^\ell, y)$ which is convex

in $\mathbf{b}^\ell + W^\ell \mathbf{h}^{\ell-1}$. We are already familiar with the cost function for both regression and classification tasks

$$\begin{aligned} \mathcal{E}(\mathbf{h}^\ell, y) &= -\log \mathbf{h}^\ell_y, \quad \text{Classification,} \\ \mathcal{E}(\mathbf{h}^\ell, y) &= ||y - \mathbf{h}^\ell_y||^2, \quad \text{Regression.} \end{aligned} \tag{2.24}$$

where $\mathbf{h}^\ell_y$ is the network output and y is the target or desired response. Note that we change notation for the output of each layer from y to h as this is the standard notation in the literature.

Next we introduce the theory on deep neural networks, their building blocks and workings. We will focus on the deep belief network which is the model to be used to solve our classification task.

2.3 Deep Belief Networks

Deep Belief Networks (DBN) are constructed by stacking Restricted Boltzmann Machines (RBM), (Hinton et al., 2006), (Erhan et al., 2010). By successively training and adding more layers we can construct a deep neural network. Once this stack of RBMs is trained, it can be used to initialise a multilayer neural network for classification, (Erhan et al., 2010). To understand how this learning machine works we need to cover the theory of Boltzmann machines and in particular that of Restricted Boltzmann Machines.

2.3.1 Boltzmann Machines

The approach taken in the theory of the Boltzmann machine is that of statistical physics and for an account of those methods we refer to the appendix.

The Boltzmann machine is a stochastic binary learning machine composed of stochastic neurons and symmetric weights. The stochastic neurons can take on two states, $+1$ for the on-state and -1 for the off-state. But $+1$ for the on-state and 0 for the off-state are also common in the literature and applications.

The neurons of the Boltzmann machine are divided in two groups or layers, a visible and a hidden layer respectively. This can be expressed as $\mathbf{v} \in \{0,1\}^V$ and $\mathbf{h} \in \{0,1\}^H$ with the indexes V and H representing the visible and hidden layers respectively. The visible neurons are connected to each other and are at the interface with the environment. It is into those visible units that we send our input signal. Under training the visible neurons are *clamped* onto specific states determined by the environment. The hidden neurons operate freely and extract features from the input signal by capturing higher order statistical correlations in the clamping vectors.

In Figure 2.11 we see a Boltzmann machine, where visible neurons are connected to each other as well as to neurons in the hidden layer. The same type of connections apply to the hidden layer neurons. Notice that connections between both layers are represented by double arrows which in this picture are represented by lines without arrow heads. This means simply that the connections are symmetrical, which is also the case for connections between neurons in the same layers.

The network learns the underlying probability distribution of the data by processing the sent clamped patterns into the visible neurons. The network can then complete patterns only if parts of the information are available assuming that the network has been properly trained.

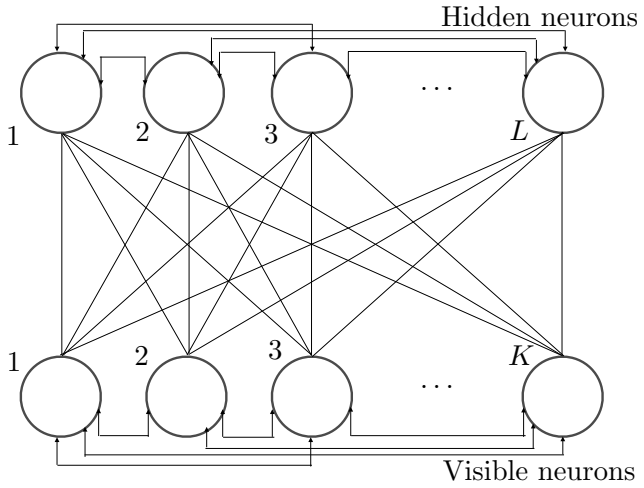

Figure 2.11: *Boltzmann machine, with K visible neurons and L hidden neurons.*

The Boltzmann machine assumes the following

1. The input vector persists long enough for the network to reach thermal equilibrium.

2. There is no structure in the sequential order in which the input vectors are clamped into the visible neurons.

A certain weight configuration of the Boltzmann machine is said to be a perfect model of the input data if it leads to the same probability distribution of the states of the visible units when they are running freely as when these units are clamped by the input vectors. To achieve such a perfect model requires an exponentially large number of hidden units as compared to the number of visible units. But this machine can achieve good performance if there is a regular structure in the data, in particular this is true when the network uses the hidden units to capture these regularities. Under such circumstances the machine can reconstruct the underlying distribution with a manageable number of hidden units.

The Boltzmann machine is said to have two phases of operation:

1. **Positive Phase**. In this phase the machine operates in its clamped condition, that is to say that it is under the influence of the training sample $\mathscr{T}$.

2. **Negative Phase**. In this phase the machine is allowed to run freely, that means that there is no environmental input.

2.3.2 Restricted Boltzmann Machines

In the Boltzmann machine there are connections among input neurons and among hidden neurons besides the connections between input and hidden layers, see Figure 2.11. In the RBM there are only connections between input and hidden layers and no connections among units in the same layer, see figure Figure 2.12. The RBM machine is a generative model and later we will see that it is used as building blocks in DBN. RBMs have also been used as building blocks in deep auto-encoders. The RBM consists of a visible and a hidden layer of binary units connected by symmetrical weights. In the RBM the hidden units are seen as feature detectors. The network assigns a probability to each pair of visible and hidden neuron-vectors according to the distribution

$$P(\mathbf{v}, \mathbf{h}; \theta) = \frac{1}{Z(\theta)} e^{-E(\mathbf{v},\mathbf{h};\theta)}, \tag{2.25}$$

where the partition function is given by $Z(\theta) = \sum_{\mathbf{v}} \sum_{\mathbf{h}} \exp(-E(\mathbf{v}, \mathbf{h}; \theta))$. The energy of the system is

$$\begin{aligned} E(\mathbf{v}, \mathbf{h}) &= -\mathbf{a}^T\mathbf{h} - \mathbf{b}^T\mathbf{v} - \mathbf{v}^T\mathbf{w}\mathbf{h} \\ &= -\sum_i b_i v_i - \sum_j a_j h_j - \sum_{i,j} w_{ij} v_i h_j, \end{aligned} \tag{2.26}$$

where a_i and b_j are the bias of the input variables v_i and hidden variables h_j respectively, and w_{ij} are the weights of the pairwise interactions between units i and j, in the visible and hidden layers. The marginal probability distribution P of data vector $\mathbf{v}$ is given by, (Salakhutdinov, 2009),

$$\begin{aligned} P(\mathbf{v}; \theta) &= \sum_{\mathbf{h}} \frac{e^{-E(\mathbf{v},\mathbf{h};\theta)}}{\sum_{\mathbf{v},\mathbf{h}} e^{-E(\mathbf{v},\mathbf{h};\theta)}} \\ &= \frac{1}{Z(\theta)} \sum_{\mathbf{h}} e^{-E(\mathbf{v},\mathbf{h};\theta)} \\ &= \frac{1}{Z(\theta)} \sum_{\mathbf{h}} \exp\left(\mathbf{v}^T w \mathbf{h} + \mathbf{b}^T\mathbf{v} + \mathbf{a}^T\mathbf{h}\right) \\ &= \frac{1}{Z(\theta)} \exp(\mathbf{b}^T\mathbf{v}) \prod_{j=1}^{F} \sum_{h_j \in \{0,1\}} \exp\left(a_j h_j + \sum_{i=1}^{D} w_{ij} v_i h_j\right) \\ &= \frac{1}{Z(\theta)} \exp(\mathbf{b}^T\mathbf{v}) \prod_{j=1}^{F} \left(1 + \exp\left(a_j + \sum_{i=1}^{D} w_{ij} v_i\right)\right), \end{aligned} \tag{2.27}$$

in this equation, $\mathbf{v}$, is the input vector and $\mathbf{h}$ is a vector of hidden units. This is the marginal distribution function over visible units. A system $(\mathbf{v}, \mathbf{h})$ with low energy is given a high probability and one with high energy is given low

probability. With the help of the energy function we can define the following probabilities

$$P(\mathbf{v}|\mathbf{h};\theta) = \prod_i P(v_i|\mathbf{h}) \quad \text{and} \quad P(v_i = 1|\mathbf{h}) = \varphi\Big(b_j + \sum_j h_j w_{ij}\Big), \tag{2.28}$$

$$P(\mathbf{h}|\mathbf{v};\theta) = \prod_j P(h_j|\mathbf{v}) \quad \text{and} \quad P(h_j = 1|\mathbf{v}) = \varphi\Big(a_j + \sum_i v_i w_{ij}\Big), \tag{2.29}$$

where φ is the sigmoid function $\varphi(x) = 1/(1+\exp(-x))$. The energy function is defined over binary vectors and is not suitable for continuous data, but by modifying the energy function to define the Gaussian - Bernoulli RBM by including a quadratic term on the visible units we can also use RBM for continuous data

$$E(\mathbf{v},\mathbf{h};\theta) = \sum_i \frac{(v_i - b_i)^2}{2\sigma_i^2} - \sum_j a_j h_j - \sum_{i,j} w_{ij}\frac{v_i}{\sigma_i} h_j. \tag{2.30}$$

The vector $\theta = \{W, \mathbf{a}, \mathbf{b}, \sigma^2\}$ and σ_i represents the variance of the input variable v_i. The vector $\mathbf{a}$ is the bias of the visible units and $\mathbf{b}$ is the bias of the hidden units. The marginal distribution over the visible vector $\mathbf{v}$ is given by, (Salakhutdinov, 2009),

$$P(\mathbf{v};\theta) = \sum_{\mathbf{h}} \frac{\exp(-E(\mathbf{v},\mathbf{h};\theta))}{\int_{\mathbf{v}'}\sum_{\mathbf{h}}\exp(-E(\mathbf{v},\mathbf{h};\theta))d\mathbf{v}'}, \tag{2.31}$$

$P(\mathbf{v}|\mathbf{h})$ becomes a multivariate Gaussian with mean $a_i + \sigma_i \sum_j w_{ij} h_j$ and diagonal covariance matrix. The conditional distributions for the visible and hidden units become, (Salakhutdinov, 2009),

$$\begin{aligned} P(v_i = x|\mathbf{h}) &= \frac{1}{\sigma_i\sqrt{2\pi}} \exp\Big(- \frac{\big(x - b_i - \sigma_i \sum_j w_{ij} h_j\big)^2}{2\sigma_i^2}\Big), \\ P(h_j = 1|\mathbf{v}) &= \varphi\Big(b_j + \sum_i \frac{v_i}{\sigma_i} w_{ij}\Big), \end{aligned} \tag{2.32}$$

where $\varphi(x) = 1/(1 + \exp(-x))$ is the sigmoid activation function. (Hinton et al., 2006) and (Salakhutdinov, 2009), mention that the Binomial-Bernoulli RBM also works for continuous data if the data is normalised to the interval [0,1]. This has been tested in experiments and seems to work well.

2.3.3 Deep Belief Networks

In the deep belief network the top two layers are modelled as an undirected bipartite associative memory, that is, an RBM. The lower layers constitute

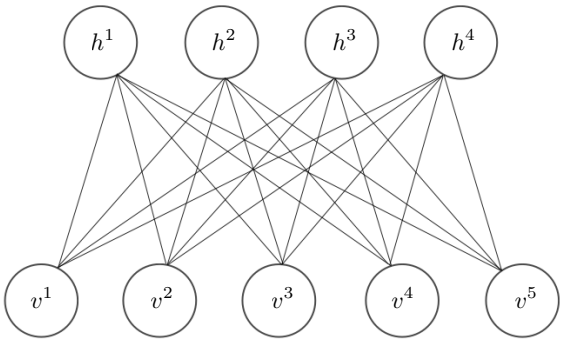

Figure 2.12: *Picture of a Restricted Boltzmann machine.*

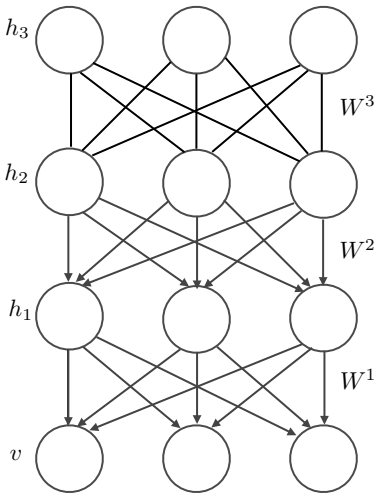

Figure 2.13: *A Deep Belief Network.*

a directed graphical model, a so called sigmoid belief network, see the appendix. The difference between sigmoid belief networks and DBN is in the parametrisation of the hidden layers, (Bengio, 2009),

$$P(\mathbf{v}, \mathbf{h}^1, \ldots, \mathbf{h}^l) = P(\mathbf{h}^{l-1}, \mathbf{h}^l)\Big(\prod_{k=0}^{l-2} P(\mathbf{h}^k|\mathbf{h}^{k+1})\Big), \tag{2.33}$$

where $\mathbf{v}$ is the vector of visible units, $P(\mathbf{h}^{k-1}|\mathbf{h}^k)$ is the conditional probability of visible units given the hidden ones in an RBM at level k. The joint distribution in the top level, $P(\mathbf{h}^{l-1}, \mathbf{h})$, is an RBM, see Figure 2.13. Another way to depict a DBN with a simpler model is shown in Figure 2.14 and it explains why a DBN is a generative model. We can see there are two kinds of arrows, dotted arrows and solid arrows.

The dotted arrows represent learning features of features while the solid ones mean that the DBN is a generative model. The dotted arrows represent the learning process and are not part of the model. The solid arrows demonstrate how the generation of data flows in the network. The generative model does not include the upward arrows in the lower layers.

A DBN is made by stacking RBMs on top of each other. The visible layer of each RBM in the stack is set to the hidden layer of the previous RBM. When learning a model for a set of data, we want to find a model $Q(\mathbf{h}^l|\mathbf{h}^{l-1})$ for the true posterior $P(\mathbf{h}^l|\mathbf{h}^{l-1})$. The posteriors Q are all approximations, except for the top level $Q(\mathbf{h}^l|\mathbf{h}^{l-1})$ posterior which is equal to the true posterior, $P(\mathbf{h}^l|\mathbf{h}^{l-1})$, (Bengio, 2009), where the top level RBM allows us to make exact inference.

In the next Chapter we present the theory for training neural networks. We introduce the back-propagation algorithm, which is used in training both shallow as well as deep neural networks. We also present some heuristics which are useful in training neural networks, then we go trough the theory for training deep neural networks.

2.3.4 Model for Financial Application

The model chosen to work with in this thesis is a DBN which was coupled to a MLP, where the later network performed the classification task. In the initialisation phase of the implementation of the DBN, the RBMs shared weights and biases with the MLP. This simply means that the RBMs are initialised with the same set of parameters as those for the MLP.

In the initialisation of the DBN-MLP we use the same weight matrices and the same bias vectors in both the DBN-module and the MLP-module.

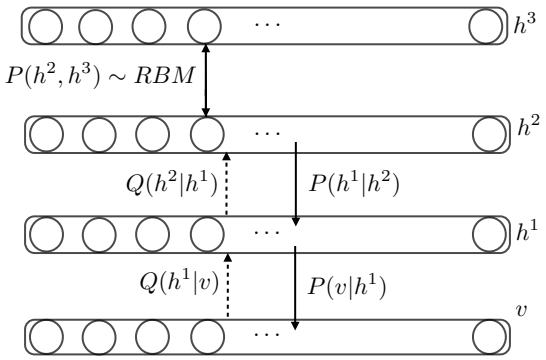

Figure 2.14: *General representation of a DBN, this model extracts multiple levels of representation of the input. The top two layers h^2 and h^3 form a RBM. The lower layers form a directed graphical model, a sigmoid belief network.*

When training begins these matrices and vectors will be adjusted according to the learning rule. As training progresses, the weight matrices and bias vectors will change in both the DBN and the MLP and they will not be the same anymore.

When training the whole network, the parameters are adjusted according to the theory presented in the Chapter on training neural networks. As explained later, see Chapter on financial model, we decided to have three hidden layers in the DBN and a hidden layer in the MLP. A sketch is shown in the figure below. Later we will see that the input vector consists of 33 features or variables and that the output layer consists of the softmax function.

Our neural network will model an unknown function from the input space $\mathbf{x}$ to the output space $Y = \{0, 1\}$:

$$F_\theta : \mathbf{x} \to Y, \tag{2.34}$$

where θ is the parameters of the neural network. We will show that our final model is give by

$$Y_{t+1} = F_\theta(\mathbf{x}), \tag{2.35}$$

which eventually leads to the estimator

$$\hat{Y}_{t+1} = \mathbb{E}[Y_{t+1}]. \tag{2.36}$$

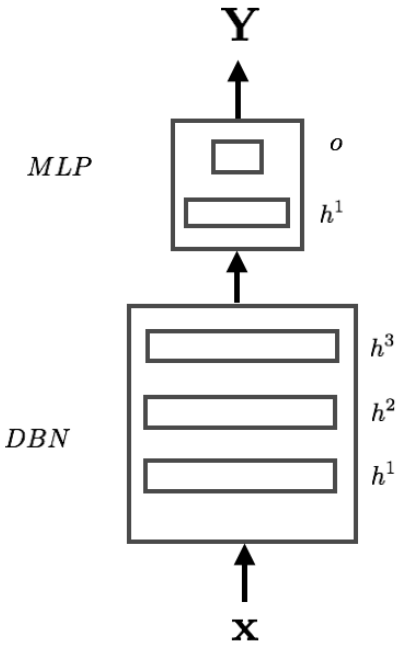

Figure 2.15: *Sketch of DBN-MLP consisting of a 3-layered DBN and a MLP with 1 hidden layer. Here h^i represents the hidden layers for $i = \{1, 2, 3\}$ and o is the output layer consisting of the softmax function. The input vector has 33 features or variables and Y takes one of the values in the set $Y = \{0, 1\}$.*

Chapter 3

Training Neural Networks

This Chapter presents the theory for training both shallow as well as deep neural networks. The first algorithm to be presented is the so called back propagation algorithm. This algorithm is used to train multilayer perceptrons, but it is also used in the training of deep neural networks, where it is used in the last step of training, for fine-tuning the parameters. Then we proceed by presenting the theory for training RBM and DBN. Contrastive Divergence (CD) is the approximating learning rule for training RBMs and is part of the layer-wise pre-taining for DBNs.

3.1 Back-Propagation Algorithm

The most common method for updating the weights in neural network theory is the so called steepest descent method, which is introduced next.

3.1.1 Steepest Descent

Steepest descent updates the weights in the direction opposite to the gradient vector $\nabla\mathcal{E}(\mathbf{w})$. Here $\mathcal{E} = \frac{1}{2}e_j^2(n)$, whith $e_j(n) = d_j(n) - h_j(n)$ representing the error between the network output $h_j(n)$ and the desired response $d_j(n)$. The steepest descent algorithm takes the form

$$\mathbf{w}(n+1) = \mathbf{w}(n) - \eta\nabla\mathcal{E}(\mathbf{w}), \tag{3.1}$$

where η is the *stepsize* or *learning-rate*. From time step n to time step $n+1$ the correction becomes

$$\begin{aligned}\Delta\mathbf{w}(n) &= \mathbf{w}(n+1) - \mathbf{w}(n)\\ &= -\eta\nabla\mathcal{E}(\mathbf{w}).\end{aligned} \tag{3.2}$$

The above equations can be used to make an approximation of $\mathcal{E}(\mathbf{w}(n+1))$ using a first order Taylor series expansion:

$$\mathcal{E}(\mathbf{w}(n+1)) \approx \mathcal{E}(\mathbf{w}(n)) + \nabla\mathcal{E}^T(n)\Delta\mathbf{w}(n). \tag{3.3}$$

In (Haykin, 2009) it is shown that this rule fulfils the condition of iterative-descent which states the following:

Proposition: *Starting with* $\mathbf{w}(0)$ *generate a sequence of weight vectors* $\mathbf{w}(1), \mathbf{w}(2), \ldots$ *such that the cost function is reduced at each iteration*

$$\mathcal{E}(\mathbf{w}(n+1)) < \mathcal{E}(\mathbf{w}(n)), \tag{3.4}$$

where $\mathbf{w}(n)$ *is the old value of the weight vector and* $\mathbf{w}(n+1)$ *is its updated value.*

Consider neuron j which receives the input signal $h_1(n), h_2(n), \ldots, h_m(n)$ and in response of this input it produces the output $v_j(n)$

$$v_j(n) = \sum_{i=0}^{m} w_{ji}(n) h_i(n). \tag{3.5}$$

In this model we think of $h_0 = 1$ corresponding to the bias with weight $w_{j0} = b_j$. The neuron output is passed through the activation function yielding the stimulus from that neuron as

$$h_j(n) = \varphi_j(v_j(n)). \tag{3.6}$$

Taking the gradient and recalling the chain rule for differentiation gives

$$\begin{aligned} \frac{\partial \mathcal{E}(n)}{\partial w_{ji}(n)} &= \frac{\partial \mathcal{E}(n)}{\partial e_j(n)} \frac{\partial e_j(n)}{\partial h_j(n)} \frac{\partial h_j(n)}{\partial v_j(n)} \frac{\partial v_j(n)}{\partial w_{ji}(n)} \\ &= -e_j(n) \varphi_j'(v_j(n)) h_i(n), \end{aligned} \tag{3.7}$$

where we have made used of the derivatives of the error signal $e_j(n) = d_j(n) - h_j(n)$, the error energy $\mathcal{E}(n) = \frac{1}{2} e_j^2(n)$, the function signal $h_j(n)$ from neuron j and the local field $v_j(n)$.

3.1.2 The Delta Rule

The delta rule is a correction $\Delta w_{ji}(n)$ applied to $w_{ji}(n)$ and it is given by

$$\Delta w_{ji}(n) = -\eta \frac{\partial \mathcal{E}(n)}{\partial w_{ji}(n)}, \tag{3.8}$$

where as usual η is the learning parameter. The minus sign accounts for gradient descent in weight space. The Delta rule can be expressed with the help of the error energy as

$$\Delta w_{ji}(n) = \eta \delta_j(n) h_i(n), \tag{3.9}$$

where $\delta_j(n) = e_j(n) \varphi_j'(v_j(n))$, is the *local gradient*. One interesting feature of the delta rule is that it gives the weight adjustment with the help of the error signal at the output of neuron j. Therefore we have to consider two different cases depending on if the neuron j producing the output signal is in the output layer or in a hidden layer.

Case 1 Output Layer

When neuron j is in the output layer we can compute the local gradient using (3.9) with the help of the error signal $e_j(n) = d_j(n) - h_j(n)$.

Case 2 Hidden Layer

When neuron j is part of a hidden layer, we can write the local gradient as

$$\begin{aligned}\delta_j(n) &= -\frac{\partial \mathcal{E}(n)}{\partial h_j(n)} \frac{\partial h_j(n)}{\partial v_j(n)} \\ &= -\frac{\partial \mathcal{E}(n)}{\partial h_j(n)} \varphi_j'(v_j(n)).\end{aligned} \tag{3.10}$$

If neuron k is an output neuron then the cost function is $\mathcal{E}(n) = \frac{1}{2}\sum_{k\in C} e_k(n)^2$, see (Haykin, 2009) page 162. Putting this in the gradient of the cost function gives

$$\frac{\partial \mathcal{E}(n)}{\partial h_j(n)} = \sum_k e_k \frac{\partial e_k(n)}{\partial h_j(n)}. \tag{3.11}$$

The error is

$$\begin{aligned}e_k(n) &= d_k(n) - h_k(n) \\ &= d_k(n) - \varphi_k(v_k(n)),\end{aligned} \tag{3.12}$$

which gives

$$\frac{\partial e_k(n)}{\partial v_k(n)} = -\varphi_k'(v_k(n)). \tag{3.13}$$

Finally the gradient of the cost function becomes

$$\begin{aligned}\frac{\partial \mathcal{E}(n)}{\partial h_j(n)} &= -\sum_k e_k(n)\varphi_k'(v_k(n))w_{kj}(n) \\ &= -\sum_k \delta_k(n) w_{kj}(n),\end{aligned} \tag{3.14}$$

where $v_k(n) = \sum_k w_{kj}(n)h_j(n)$ and $\frac{\partial v_k(n)}{\partial h_j(n)} = w_{kj}(n)$. The calculations give us the local gradient for hidden neuron j as

$$\delta_j(n) = \varphi_j'(v_j(n)) \sum_k \delta_k(n) w_{kj}(n). \tag{3.15}$$

Summary

Here we present a summary of the updating rule for the parameters of the network

1. Update the parameters according to the delta rule

$$\Delta w_{ji}(n) = \eta \delta_j(n) h_i(n). \tag{3.16}$$

2. If neuron j is an output node use the result of case 1 above for output layers.

3. If neuron j is a hidden node use the result of case 2 above for hidden layers, in which case we need the δs for the neurons from the next hidden or output layer.

The back propagation algorithm is used to update the weights, and it works by passing the data in a forward phase and a backward phase. Next we explain what forward and backward phases mean.

3.1.3 Forward and Backward Phase

Forward Phase

The back-propagation algorithm is applied in two phases. In the first phase or forward phase, the input data passes through the synaptic weights from one layer to the next, till the data finally comes out in the output neurons. The function signal out from the network is expressed as

$$h_j(n) = \varphi\Big(\sum_{i=0}^{m} w_{ij}(n) h_i(n) \Big), \tag{3.17}$$

where φ is the activation function. The total number of inputs is m, excluding bias, applied to neuron j and $w_{ji}(n)$ is the synaptic weight connecting neuron i to neuron j. The input signal of neuron j is $h_i(n)$. Recall that $h_i(n)$ is at the same time the output of neuron i. If neuron j is in the first hidden layer then

$$h_i(n) = x_i(n), \tag{3.18}$$

where $x_i(n)$ is the ith element of the input data to the network. If instead neuron j is in the output layer then

$$h_j(n) = o_j(n), \tag{3.19}$$

where $o_j(n)$ is the jth element of the output vector. The output is compared to the desired response $d_j(n)$ giving us the error $e_j(n)$.

Backward Phase

In the backward phase we start at the output nodes and go through all the layers in the network and recursively compute the gradient, δ, for each neuron in every layer. In this way we update the synaptic weights according

to the Delta rule. In the output layer, δ is simply the error multiplied by the first derivative of its activation function. We use (3.9) to compute the changes of the weights of all the connections leading to the output layer. When we obtain δ for the output layer we can then continue by computing the δs for the layer just before the output layer with the help of (3.15). This recursive computation is continued layer by layer by propagating the changes to all the synaptic weights in the network.

The presentation of each training example in the input pattern is fixed or "clamped" through out the round-trip process of the forward phase followed by the backward phase. Later we will see that there are two ways to pass data through the learning algorithm under training. Those methods are called on-line learning mode and batch learning mode. There is yet a third method, called mini batch training which is a hybrid of the first two mentioned methods and it is the method chosen for training in this master thesis.

3.1.4 Computation of δ for Known Activation Functions

Here we compute δ for the *sigmoid* or logistic function and for the *hyperbolic tangent* function given by $a \cdot \tanh(b \cdot v_j(n))$. Their derivatives are

$$\begin{aligned} \varphi_j'(v_j(n)) &= \frac{a \exp(-a v_j(n))}{\Big(1 + \exp\big(-a v_j(n)\big)\Big)^2}, \\ \varphi_j'(v_j(n)) &= a \cdot b \cosh^{-2}(b v_j(n)), \end{aligned} \tag{3.20}$$

respectively. We note that $h_j(n) = \varphi_j(v_j(n))$ which can be used in the formulas for the derivatives. Now we show two formulas for δ according to if the neurons reside in the output layer or in a hidden layer.

1. **The neurons reside in the output**. Using that $h_j(n) = o_j(n)$ is the function signal at the output of neuron j and $d_j(n)$ is the desired response

$$\delta_j(n) = \begin{cases} a\Big[d_j(n) - o_j(n)\Big] o_j(n) \Big[1 - o_j(n)\Big], & \text{sigmoid} \\ \Big(b/a\Big)\Big[d_j(n) - o_j(n)\Big]\Big[a - o_j(n)\Big]\Big[a + o_j(n)\Big], & \text{h. tangent,} \end{cases}$$

   where the abbreviation h. tangent refers to the hyperbolic tangent.

2. **The neurons reside in a hidden layer**. In the case of a hidden

layer we have the expressions

$$\delta_j(n) = \begin{cases} a \cdot h_j(n)\Big[1 - h_j(n)\Big] \sum_k \delta_k(n) w_{kj}(n), & \text{sigmoid} \\ \Big(b/a\Big)\Big[a - h_j(n)\Big]\Big[a + h_j(n)\Big] \sum_k \delta_k(n) w_{kj}(n), & \text{h. tangent,} \end{cases}$$

where the abbreviation h. tangent refers to the hyperbolic tangent.

3.1.5 Choosing Learning Rate

Choosing a small value for the learning rate makes the interactions in weight space smooth, but at the cost of longer learning rate. Choosing a large learning rate parameter makes the adjustment too large which makes the network unstable. To speed up calculations and at the same time not jeopardise the stability of the network, a momentum term is incorporated and the delta rule becomes

$$\Delta w_{ji}(n) = \alpha \Delta w_{ji}(n-1) + \eta \delta_j(n) h_j(n), \tag{3.21}$$

here α is a positive constant called the momentum constant. When written in this form the delta rule is called the generalised delta rule. It can be cast to a time series by introducing the index t which goes from 0 to the current time n. Solving the equation for $\Delta w_{ji}(n)$ we end up with

$$\Delta w_{ji}(n) = \eta \sum_{t=0}^{n} \alpha^{n-t} \delta_j(t) h_j(t). \tag{3.22}$$

We can use earlier equations to express this last equation with the help of the error energy function

$$\Delta w_{ji}(n) = -\eta \sum_{t=0}^{n} \alpha^{n-t} \frac{\partial \mathcal{E}(t)}{\partial w_{ji}(t)}. \tag{3.23}$$

3.1.6 Stopping Criteria

Let the weight vector $\mathbf{w}^*$ be a minimum, local or global. It can be found by taking the gradient of the error surface with respect to $\mathbf{w}$ where it must be zero for $\mathbf{w} = \mathbf{w}^*$. A convergence criterium for the back-propagation algorithm is, (Kramer and Sangiovanni-Vincentelli, 1989):

The back-propagation algorithm is considered to have converged when the Euclidean norm of the gradient vector reaches a sufficiently small gradient threshold.

On the negative side is the fact that we have to compute the gradient, in addition the learning time may be long. An improvement to the stated criterium is to use that $\mathcal{E}_{av}(\mathbf{w})$ is stationary at $\mathbf{w} = \mathbf{w}^*$ and the following criterium follows, (Haykin, 2009):

The back-propagation-algorithm is considered to have converged when the absolute rate of change in the average squared error per epoch is sufficiently small.

The rate of change is considered small when it lies between 0 and 1 percent per epoch. Better still is to test the generalization performance of the algorithm. The back-propagation algorithm is summarised in Algorithm 1.

Early-Stopping

We divide the training data in two sets, the estimation data and the validation data. As usual we train our network with the estimation data and the validation set is used to test generalisation capacity. Training is stopped periodically e.g. after every 5 epochs, and the network is tested on the validation subset after each period of training. We proceed as follows:

1. After a period of training (- every five epochs, for example -) the synaptic weights and bias of the MLP are all fixed, and the network is operated in its forward mode. The validation error is measured for each example in the validation subset.

2. When validation is completed, training is resumed for another period and the process is repeated.

This method of training will be used in our experiments. In Figure 3.1 we see an illustration of early stopping based on cross validation.

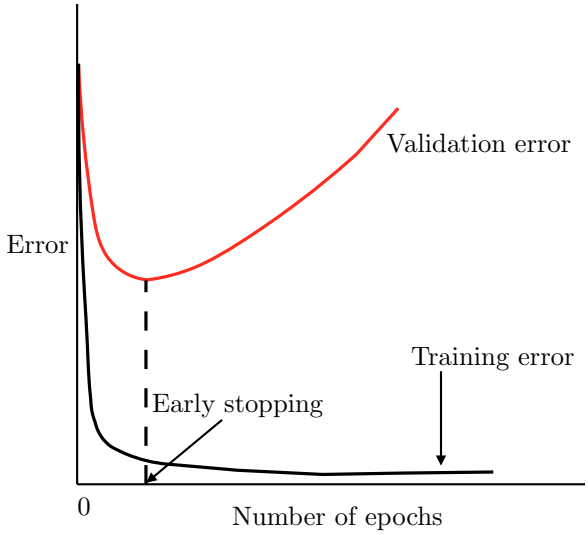

Figure 3.1: *Early stopping based on cross-validation.*

In Algorithm 1 we show the pseudo code for the back propagation algorithm. Let us define the output for all the neurons at layer i by

$$\mathbf{h}^i = \varphi(\mathbf{w}^i \mathbf{h}^{i-1}),$$

where $\varphi(\cdot)$ is the sigmoid function, $\mathbf{h}^{(i-1)}$ is the output of all the neurons from the previous layer $i-1$ and $\mathbf{w}^i$ is the synaptic weight matrix between the neurons in layer $i-1$ and layer i. For the first hidden layer, $i=1$ and we set

$$\mathbf{h}^0 = \mathbf{x}_t.$$

The output from the network is

$$\mathbf{h}^\ell = \mathbf{o}(\mathbf{x}_t).$$

The softmax output $\mathbf{h}^\ell$ can be used as an estimator of $P(Y=i|x)$ where Y is the class associated with the input x. When we solve classification problems we use the conditional log-likelihood given by

$$\mathcal{E}(\mathbf{h}^\ell, y) = -\log \mathbf{h}^\ell_y.$$

Let $|h^i|$ be the size of layer i, $\mathbf{b}$ the bias vector, K the number of outputs from the network, n_{in} the number of units in layer $(i-1)$ and n_{out} the number of units in layer i. Finally let $\eta > 0$ denote the learning parameter, then we can summarise the algorithm for a network with ℓ layers as in the following table.

Algorithm 1: *Back-Propagation Algorithm*

Initialisation:
weights $\mathbf{w}^i \sim U(-4 \cdot \sqrt{a}, 4 \cdot \sqrt{a})$, $a = 6/(n_{in} + n_{out})$
bias $\mathbf{b} = 0$
while Stopping criterion is not met
  pick input example $(\mathbf{x}_t, y_t)$ from training set
  Forward propagation:
  $\mathbf{h}(\mathbf{x}_t) \leftarrow \mathbf{x}_t$
  for $i \in \{1, \ldots, \ell\}$
    $\mathbf{a}^i(\mathbf{x}_t) = \mathbf{b}^i + \mathbf{w}^i \mathbf{h}^{i-1}(\mathbf{x}_t)$
    $\mathbf{h}^i(\mathbf{x}_t) = \varphi(\mathbf{a}^i(\mathbf{x}_t))$
  end for
  $\mathbf{a}^{\ell+1}(\mathbf{x}_t) = \mathbf{b}^{\ell+1} + \mathbf{w}^{\ell+1}\mathbf{h}^{\ell}(\mathbf{x}_t)$
  $\mathbf{o}(\mathbf{x}_t) = \mathbf{h}^{\ell+1}(\mathbf{x}_t) = \text{softmax}(\mathbf{a}^{\ell+1}(\mathbf{x}_t))$

  Backward gradient propagation and parameter update:
  $\frac{\partial \log o_{y_t}(\mathbf{x}_t)}{\partial a_j^{\ell+1}(\mathbf{x}_t)} \leftarrow \mathbf{I}(y_t = j) - o_j(\mathbf{x}_t)$ **for** $j \in \{1, \ldots, K\}$
  $\mathbf{b}^{\ell+1} \leftarrow \mathbf{b}^{\ell+1} + \eta \frac{\partial \log o_{y_t}(\mathbf{x}_t)}{\partial \mathbf{a}^{\ell+1}(\mathbf{x}_t)}$
  $\mathbf{w}^{\ell+1} \leftarrow \mathbf{w}^{\ell+1} + \eta \frac{\partial \log o_{y_t}(\mathbf{x}_t)}{\partial \mathbf{a}^{\ell+1}(\mathbf{x}_t)} \mathbf{h}^{\ell}(\mathbf{x}_t)^T$

  for $i \in \{1, \ldots, \ell\}$, in decreasing order
    $\frac{\partial \log o_{y_t}(\mathbf{x}_t)}{\partial \mathbf{h}_j^i(\mathbf{x}_t)} \leftarrow (\mathbf{w}^{i+1})^T \frac{\partial \log o_{y_t}(\mathbf{x}_t)}{\partial \mathbf{a}^{i+1}(\mathbf{x}_t)}$ **for** $j \in \{1, \ldots, |\mathbf{h}^i|\}$
    $\frac{\partial \log o_{y_t}(\mathbf{x}_t)}{\partial a_j^i(\mathbf{x}_t)} \leftarrow \frac{\partial \log o_{y_t}(\mathbf{x}_t)}{\partial h_j^i(\mathbf{x}_t)} h_j^i \left(1 - h_j^i(\mathbf{x}_t)\right)$
    $\mathbf{b}^i \leftarrow \mathbf{b}^i + \eta \frac{\partial \log o_{y_t}(\mathbf{x}_t)}{\partial \mathbf{a}^i}$
    $\mathbf{w}^i \leftarrow \mathbf{w}^i + \eta \frac{\partial \log o_{y_t}(\mathbf{x}_t)}{\partial \mathbf{a}^i} \mathbf{h}^{i-1}(\mathbf{x}_t)^T$
  end for
end while

3.1.7 Heuristics For The Back-Propagation Algorithm

There are some tested design choices which make the back-propagation algorithm improve its performance. Here is a list of proven methods, (Haykin, 2009).

1. **Update choice**. The stochastic or sequential mode of the algorithm means that learning is performed by presenting the training set example by example. This makes the algorithm run faster than when the batch mode is used.

2. **Maximise information content**. Here the choice is to pick examples of data that make the training error differ, in that way we search in the whole weight space. One way to achieve this is by presenting the training data in a random order from one epoch to the next.

3. **Activation function**. Using a sigmoid activation function that is odd $\varphi(-v) = -\varphi(v)$ speeds up the learning process. A good choice is the hyperbolic tangent function

$$\varphi(v) = a \tanh(bv).$$

with values (LeCun, 1993) $a = 1.7159$ and $b = 2/3$.

4. **Target values**. The target values should be within the range of the sigmoid activation function. For example the maximum value of the target should be offset by some amount ϵ away from the limiting value of the activation function depending on if the limiting value is positive or negative. For example if the target or desired value from neuron j is d_j then for the limiting value $+a$

$$d_j = a - \epsilon,$$

and for the limiting value $-a$

$$d_j = -a + \epsilon.$$

Otherwise the back-propagation algorithm tends to drive the free parameters to infinity making the learning process slow down and drive the hidden neurons into saturation.

5. **Normalizing inputs**. Data should be normalized so that its mean value, averaged over the entire training sample should be close to zero. The correlation in the data should be taken away and its covariance should be the same. For a deeper discussion concerning this process see (Haykin, 2009).

6. **Initialisation** If the initial values are large then the neurons are driven to saturation. If this happens the local gradients in the back-propagation algorithm assume small values which causes the learning process to slow down. If on the other hand, the weights are assigned small values, the back-propagation algorithm may operate in a very flat area around the origin of the error surface. This is true for the sigmoid functions such as the hyperbolic tangent. The origin is a saddle point, which means that the curvature across the saddle point is negative and the curvature along the saddle point is positive. It is in this area that we have a stationary point. We should therefore avoid both too high values and too small values as initial values. Haykin shows that we should initialise the weights according to drawing random values from a uniform distribution with mean zero and variance equal to the reciprocal of the number of synaptic connections of a neuron.

7. **Learning from hints**. Our objective is to train a model which can represent a map from the input space to the output space, which is represented by $f(\cdot)$. If we have previous knowledge of $f(\cdot)$ we should use this knowledge in the training of the network. This could be e.g. known symmetries of the mapping $f(\cdot)$ or other invariance properties which could accelerate the learning process.

8. **Learning rates**. All neurons should learn at the same rate, but the last layers in the network have larger local gradients than the layers in the front end of the network. For this reason the learning rate should be given a small value at last layers than in the front layers. It is also experimentally known that neurons with many inputs should have a smaller learning rate, for example (LeCun 1993) suggests that the learning rate for a neuron should be inversely proportional to the square of the number of connections to that neuron.

3.2 Batch and On-Line Learning

The back-propagation algorithm can be used together with on-line learning or batch-learning. In this thesis we will use a hybrid of those methods using mini batches of size much smaller than the number of training examples. For that reason this method is sometimes called mini-batch stochastic gradient descent. This procedure has the benefit of being enough stochastic to avoid being trapped in bad local minima in the error space spanned by the weight vector. For the multilayer perceptron, let $\mathscr{T} = \{\mathbf{x}(n), \mathbf{d}(n)\}_{n=1}^{N}$ be the training samples used in a supervised manner. Denote by $h_j(n)$ the function signal at the output of neuron j, in the output layer, corresponding to the stimulus applied to the input layer. Then the error signal is

$$e_j(n) = d_j(n) - h_j(n), \tag{3.24}$$

where $d_j(n)$ is the jth element of the desired response vector $\mathbf{d}(n)$. We define the instantaneous *error energy* of neuron j by

$$\mathcal{E}_j(n) = \frac{1}{2} e_j^2(n). \tag{3.25}$$

Now summing over all output neurons in the output layer

$$\begin{aligned} \mathcal{E}(n) &= \sum_{j \in C} \mathcal{E}_j(n) \\ &= \frac{1}{2} \sum_{j \in C} e_j^2(n), \end{aligned} \tag{3.26}$$

where C is the set of all the neurons in the output layer. Taking averages over all N samples we arrive at the *error energy averaged over all training*

samples

$$\begin{aligned}\mathcal{E}_{av}(N) &= \sum_{n=1}^{N} \mathcal{E}(n) \\ &= \frac{1}{2N} \sum_{n=1}^{N} \sum_{j \in C} e_j^2(n).\end{aligned} \tag{3.27}$$

3.2.1 Batch Learning

An implementation of the perceptron algorithm using batching is presented in the section below. Using batching, means that all the weights of the multi layer perceptron are adjusted after all the N examples in the training set $\mathscr{T}$ that constitute an *epoch* have been presented. Here we make use of the average error function, $\mathcal{E}_{av}$ just presented and adjust the weights in an epoch-by-epoch basis. The examples in the training set are randomly shuffled. Using gradient descent with the batch method offers two benefits.

1. *Accurate estimation* of the gradient vector and convergence to a local minimum.

2. *Parallelisation* of the learning process

On the negative side is the requirement of storage. This method suits the solution of nonlinear regression problems.

3.2.2 The Use of Batches

Here we generalize the perceptron algorithm by presenting a more general cost function which can be used to formulate a batch version of the algorithm. In this setting the perceptron cost function is defined as

$$J(\mathbf{w}) = \sum_{\mathbf{x} \in \mathscr{H}} (-\mathbf{w}^T \mathbf{x}), \tag{3.28}$$

here $\mathscr{H}$ stands for the set of misclassified $\mathbf{x}$. Taking the derivatives on the cost function yields

$$\nabla J(\mathbf{w}) = \sum_{\mathbf{x} \in \mathscr{H}} (-\mathbf{x}), \tag{3.29}$$

were the gradient operator is given by

$$\nabla = \left[\frac{\partial}{\partial w_1}, \frac{\partial}{\partial w_2}, \ldots, \frac{\partial}{\partial w_m} \right]^T.$$

In the method of *steepest descent* the adjustment to the weight vector $\mathbf{w}$ is applied in a direction opposite to the gradient vector $\nabla J(\mathbf{w})$ arriving at

$$\begin{aligned}\mathbf{w}(n+1) &= \mathbf{w}(n) - \eta(n)\nabla J(\mathbf{w}) \\ &= \mathbf{w}(n) + \eta(n) \sum_{\mathbf{x}\in\mathscr{H}} \mathbf{x}. \end{aligned} \tag{3.30}$$

This formula embodies a general representation for the perceptron algorithm and has the single-sample version of the convergence algorithm as a special case.

3.2.3 On-Line Learning

On-line learning means that adjustment to the weights of the multilayer perceptron is done on an *example-by-example basis*. The cost function used in this case is the error energy function $\mathcal{E}(n)$. The training is done by arranging an epoch of N training samples and present those in the order, $[\mathbf{x}(1), \mathbf{d}(1)], [\mathbf{x}(2), \mathbf{d}(2)], \ldots, [\mathbf{x}(N), \mathbf{d}(N)]$, adjusting the weights with the method of gradient descent. We continue until we reach the N-sample pair. On-line learning behaves in a stochastic manner and for this reason it avoids to be trapped in local minima. On the positive side of this method we can mention that it requires less storage resources, it is well suited for large scale and difficult pattern classification problems and it is simple to implement. On the negative side we can mention that it does not work well on parallalization.

3.2.4 Generalization

The concept of generalization refers to the predictive power of a model to new data. As mentioned before we divide the data in three sets. A training set for training the network, a test set for testing the predictive power of the model and a validation set for tuning hyper parameters. After the training we expose our model to test data and hope to get a good prediction from the model. If the model has good generalization it should predict in a good manner if the data comes from the same distribution.

A model generalises well if the input-output mapping computed by the network is correct for data never used in training or creation of the network. The learning process can be considered as a curve fitting problem where the network is a nonlinear input-output mapping and the network model can therefore be considered as an interpolation problem of the input data. A problem in building network models is that of overfitting or overtraining, where the network memorises the data. In this case the network might learn features such as noise in the training data that might not exist in the underlying distribution of the data. When the network is overtrained

it will not generalize well to similar input-output patterns. Recall Occam's razor criterium which says that it is better to choose the simplest model that describes our data in a smooth manner. Because the number of training examples has such an impact in the generalization properties of neural networks, researchers have come up with some criteria on how to choose the number of examples.

Three key properties have been identified to contribute to the generalization, these are:

1. the size of training data
2. the architecture of the neural network
3. the physical complexity of the problem to be studied.

For good generalization it is enough that the number of training examples, N, satisfies the condition,

$$N = O\left(\frac{W}{\epsilon}\right), \tag{3.31}$$

where W is the number of weights and biases and ϵ is the fraction of classification errors or a threshold for the cost in regression problems on test data. $O(\cdot)$ refers to the ordo notation giving the order of magnitude for the problem.

3.2.5 Example: Regression with Neural Networks

In this section we perform a regression with the back-propagation algorithm on a toy example given by the function

$$y(x) = e^{-\frac{1}{2}\cdot x^2} \cdot (4 \cdot \sin^2 6x + 3\cos^2 x \cdot \sin^2 4x + 1). \tag{3.32}$$

This function will be the target for our network and the cost function is given by

$$\mathcal{E}(\theta) = \frac{1}{2}\sum_{i=1}^{N}\Big(y(x_i) - f(x_i;\theta)\Big)^2, \tag{3.33}$$

where $f(x_i;\theta)$ is the output of our network. As this example is a toy example not so much time will be spent in the analysis but we show some results and compare results from a MATLAB standard implementation and one implementation from scratch. Two modes of learning were tested, the on-line mode and the batch mode of learning. First we show how MATLAB's implementation of back-propagation performs in Figure 3.2 where the network has a hidden layer with 8 neurons. In this demonstration we performed back-propagation by steepest descent and the goal was to see how well the model fit to data. When learning a model it is always possible for the learning algorithm to learn well how to fit to the training data. The difficulty in

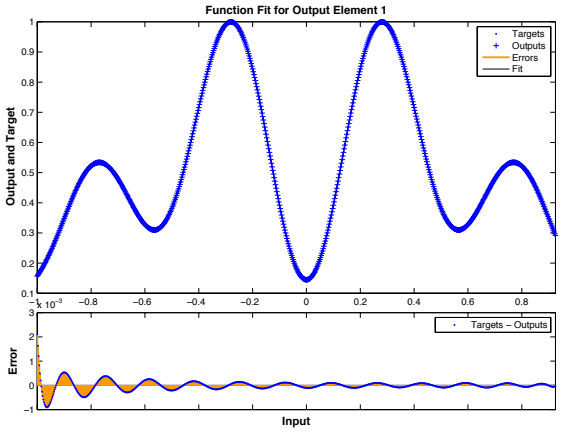

Figure 3.2: *Figure showing MATLAB implementation for our neural network. There is a perfect match in the regression. The model is composed of a hidden layer with 8 neurons and an output layer with one neuron.*

learning is how well the model is at generalisation. This was not proven for this demonstration but we held out some of the data and showed it to the algorithm. The graphs can be found in the appendix for different situations. No validation error or test error were measured.

The implementation from scratch is shown in Figure 3.3 and shows a perfect fit, as did the MATLAB implementation. The first implementation for the neural network is composed of two layers, one hidden layer and one output layer. The number of neurons in the hidden layer are chosen such that the cost function is minimised. For the first example we see that it is enough with 8 neurons. Both implementations are based on on-line learning. The batch implementation seems to be not so good because the algorithm is updated after all training examples have been processed. In the on-line implementation the training set is presented to the update rule in the back-propagation in an example by example basis. In this case the approximation to the gradient is near the real gradient and it seems not to get stuck in valleys in the error surface.

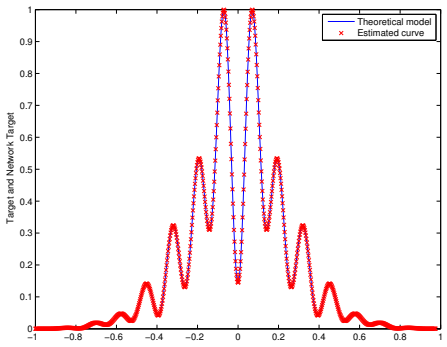

Figure 3.3: *Neural network with 8 hidden neurons, own implementation. Different normalisation than in the MATLAB implementation, but showing perfect fit. Implementation in on-line mode of learning.*

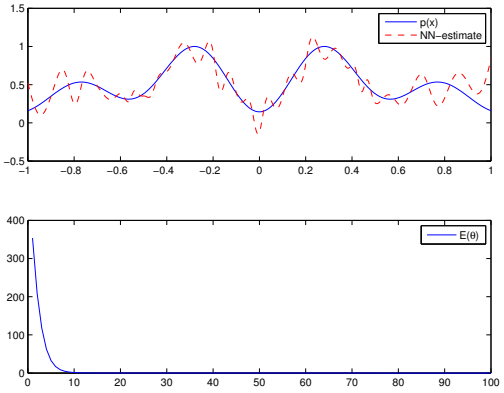

Figure 3.4: *Regression with back-propagation for a network with 170 neurons in the hidden layer. Implemented in the batch-mode of learning.*

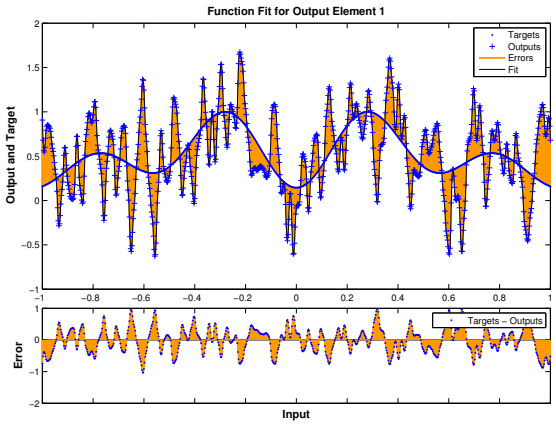

Figure 3.5: *Regression with MATLAB's own function.*

In the second example we tested more neurons and we saw that the model is not so good at generalisation either in MATLAB's implementation or our own, see figures in the appendix. More neurons show that the model is overfitted and probably not so good at generalisation. In both examples 170 weights were used in the network. The network is a 2-layer network with one hidden layer containing 170 neurons and one output layer with only 1 neuron.

3.3 Training Restricted Boltzmann Machines

The training of RBM is a maximum likelihood learning of Markov random fields, (Carreira-Perpiñán and Hinton), and as such is an intractable one. The reason for this, is the exponential number of terms in the estimates of the averages involved in the learning rule. Monte Carlo methods take very long time to reach an unbiased estimate, but (Hinton, 2002) has shown that if the Markov chain runs for a few steps we can still get a good estimate. This new method of estimation of the gradient is called Contrastive Divergence (CD). Carreira-Perpiñán and Hinton present the theory behind CD in their paper (Carreira-Perpiñán and Hinton).

Unsupervised learning means that we want to model an unknown distribution Q given some sample data without the need of labels. An RBM is a Markov Random Field (MRF), (Fischer and Igel, 2014), and unsupervised learning in this setting is the same as estimating the parameters, θ, of the model. The model to learn is given by $P(x|\theta)$, where x is the data and θ is the parameter set we want to estimate. Let $S = \{x_1, x_2, \ldots, x_l\}$ be the

data vector, then we can estimate the parameters by maximum-likelihood. From elementary statistics we know that maximisation of the likelihood is equivalent to maximisating the log-likelihood

$$\log \mathcal{L}(\theta|S) = \log \prod_{i=1}^{l} P(x_i|\theta) = \sum_{i=1}^{l} \log P(x_i|\theta). \tag{3.34}$$

Maximisation of the log-likelihood is equivalent to minimisation of the distance between the unknown distribution Q underlying S and the true distribution P in terms of the Kullback-Leibler divergence (KL divergence), (Fischer and Igel, 2014),

$$\begin{aligned} \mathrm{KL}(Q||P) &= \sum_{x\in\Omega} Q(x) \log\left(\frac{Q(x)}{P(x)}\right) \\ &= \sum_{x\in\Omega} Q(x) \log Q(x) - \sum_{x\in\Omega} Q(x) \log P(x). \end{aligned} \tag{3.35}$$

Learning in the RBM framework is done with gradient descent or ascent, the difference being only a matter of which sign, plus or minus, is used in the update rule. Consider the update rule:

$$\begin{aligned} \theta^{(t+1)} &= \theta^{(t)} + \eta \frac{\partial}{\partial \theta^{(t)}} \Big(\log \mathcal{L}(\theta^{(t)}|S) \Big) - \lambda \theta^{(t)} + \mu \Delta\theta^{(t-1)} \\ &= \theta^{(t)} + \Delta\theta^{(t)}, \end{aligned} \tag{3.36}$$

where η is the learning parameter, λ is the parameter corresponding to weight decay, (a form of regularisation, L_2) and μ is the parameter corresponding to the momentum term. Momentum can be used to stabilise the update rule against oscillations and speed up the learning process.

For $p(\mathbf{v}) = \sum_{\mathbf{h}} P(\mathbf{v},\mathbf{h}) = \frac{1}{Z}\sum_{\mathbf{h}} e^{-E(\mathbf{v},\mathbf{h})}$ we obtain the gradient of the log-likelihood for a single training as, (Fischer and Igel, 2014),

$$\begin{aligned} \frac{\partial \log \mathcal{L}(\theta|\mathbf{v})}{\partial\theta} &= \frac{\partial}{\partial\theta}\Big(\log \sum_{\mathbf{h}} e^{-E(\mathbf{v},\mathbf{h})} \Big) - \frac{\partial}{\partial\theta}\Big(\log \sum_{\mathbf{v},\mathbf{h}} e^{-E(\mathbf{v},\mathbf{h})} \Big) \\ &= -\sum_{\mathbf{h}} P(\mathbf{h}|\mathbf{v}) \frac{\partial E(\mathbf{v},\mathbf{h})}{\partial\theta} + \sum_{\mathbf{v},\mathbf{h}} P(\mathbf{v},\mathbf{h}) \frac{\partial E(\mathbf{v},\mathbf{h})}{\partial\theta} \\ &= -\left\langle \frac{\partial E(\mathbf{v},\mathbf{h})}{\partial\theta} \right\rangle_d + \left\langle \frac{\partial E(\mathbf{v},\mathbf{h})}{\partial\theta} \right\rangle_m, \end{aligned} \tag{3.37}$$

where angle brackets stand for expectation and their indexes d and m refers to whether those are taken with respect to the data distribution or model dis-

tribution respectively. In our case this can be written as, (Salakhutdinov, 2009),

$$\begin{aligned}
\frac{\partial \log P(\mathbf{v};\theta)}{\partial W} &= \mathbb{E}_d[\mathbf{v}\mathbf{h}^T] - \mathbb{E}_m[\mathbf{v}\mathbf{h}^T] = \langle \mathbf{v}\mathbf{h}^T \rangle_d - \langle \mathbf{v}\mathbf{h}^T \rangle_m, \\
\frac{\partial \log P(\mathbf{v};\theta)}{\partial \mathbf{a}} &= \mathbb{E}_d[\mathbf{h}] - \mathbb{E}_m[\mathbf{h}] = \langle \mathbf{h} \rangle_d - \langle \mathbf{h} \rangle_m, \\
\frac{\partial \log P(\mathbf{v};\theta)}{\partial \mathbf{b}} &= \mathbb{E}_d[\mathbf{v}] - \mathbb{E}_m[\mathbf{v}] = \langle \mathbf{v} \rangle_d - \langle \mathbf{v} \rangle_m,
\end{aligned} \tag{3.38}$$

where $\mathbb{E}_d[\cdot]$ is the expectation with respect to the data distribution $P_d(\mathbf{h}, \mathbf{v}; \theta) = P(\mathbf{h}|\mathbf{v}; \theta)P_d(\mathbf{v})$, $P_d(\mathbf{v}) = \frac{1}{N}\sum_n \delta(\mathbf{v} - \mathbf{v}_n)$ being the empirical distribution and $P_m = \frac{1}{Z(\theta)} e^{-E(\mathbf{x};\theta)}$ is the distribution of the model, in this case the Gibbs distribution, sometimes also denoted as $P_m = P_\infty(\mathbf{x}; \theta)$. In the literature we sometimes see the index ∞ instead of m, but to simplify matters we adopt to use the m index and we point out when we do otherwise.

3.3.1 Contrastive Divergence

The expectation that is difficult to compute is the one under the model distribution which involves the partition function as a normalisation constant. This normalisation constant has an exponential number of terms and is therefore ineffective to compute exactly. The approximation used is to estimate the average by a sample from $P(\mathbf{x}; \theta)$ with a Markov chain that converges to $P(\mathbf{x}; \theta)$ and running the chain to equilibrium. But in the end this approach is very time consuming and to remedy this problem (Hinton, 2002), showed that Contrastive Divergence (CD) is a better method for computing the log-likelihood. It is argued that CD follows the gradient of a different function. If we call our Kullback-Leibler divergence, (Carreira-Perpiñán and Hinton),

$$\mathrm{KL}(P_d||P_m) = \sum_{\mathbf{x}} P_d(\mathbf{x}) \log \left(\frac{P_d(\mathbf{x})}{P_m(\mathbf{x};\theta)} \right), \tag{3.39}$$

CD will approximate

$$\mathrm{CD}_n = \mathrm{KL}(P_d||P_m) - \mathrm{KL}(P_n||P_m), \tag{3.40}$$

where the learning process of the Markov chain starts at the data distribution P_d and is run for n full steps of alternating Gibbs sampling. We can reformulate CD as

$$\mathrm{CD} = \eta \Big(\mathbb{E}_d[\mathbf{v}\mathbf{h}^T] - \mathbb{E}_G^K[\mathbf{v}\mathbf{h}^T] \Big) = \eta \Big(\langle \mathbf{v}\mathbf{h}^T \rangle_d - \langle \mathbf{v}\mathbf{h}^T \rangle_G^K \Big), \tag{3.41}$$

where η is the learning rate and $\langle \cdot \rangle_G$ is the distribution sampled from a Gibbs chain initialised at the data, for K full steps. Contrastive Divergence is a training algorithm used to speed up training of RBMs. This method uses two tricks to speed up the sampling process:

- We want $P(v) \approx P_d(v)$, then we can initialise the Markov chain with a training example.
- Contrastive divergence does not wait for the chain to converge, and in practice $K = 1$ is enough for most common tasks and yields good results.

Because there are no direct connections between hidden units in an RBM, it is simple to sample from $\mathbb{E}_d[v_i h_i]$. Given a randomly selected training example, $\mathbf{v}$, the binary state, h_j, of each hidden unit, j, equals 1 with probability

$$P(h_j = 1|\mathbf{v}) = \varphi\left(b_j + \sum_i v_i w_{ij}\right), \tag{3.42}$$

where $\varphi(\cdot)$ is the logistic sigmoid, and $v_i h_j$ is an unbiased sample. In the same manner, because there are no connections among the visible units, we can get an unbiased sample of the state of a visible unit given a hidden vector

$$P(v_i = 1|\mathbf{h}) = \varphi\left(a_i + \sum_j h_j w_{ij}\right), \tag{3.43}$$

and we obtain an unbiased sample of $\mathbb{E}_m[v_i h_j]$. But this time the sampling is much more difficult to obtain. Sampling can be done by starting at any random state of the visible units and performing alternating Gibbs sampling for a very long time. Hinton, (Hinton, 2002), proposes a faster method by setting the states of the visible units to a training vector, then the states of the hidden units are all computed in parallel by using (3.42). Once we have sampled the hidden states we can proceed to construct a "reconstruction" by setting each v_i to 1 with probability given by (3.43). The weights can then be updated by

$$\Delta W = \eta\left(\mathbb{E}_d[\mathbf{v}\mathbf{h}^T] - \mathbb{E}_{recon}[\mathbf{v}\mathbf{h}^T]\right), \tag{3.44}$$

where η is the learning rate and in the second expectation we have the index for the reconstructed states. For the biases we can simplify further by using a version of the same learning rule which uses the states of individual units instead of pairwise products. The learning rule is much more closely approximating the CD. It is known that the more steps in the alternating Gibbs sampling the better models the RBM learns before collecting statistics for the second term in the learning rule. The following figure depicts the alternating Gibbs sampling, in which we measure the state of the chain after the first update and then at the end of the chain. After many steps the visible and hidden vectors are sampled from the stationary distribution defined by the current parameters of the model. The weights are then updated by providing this signal to the learning rule.

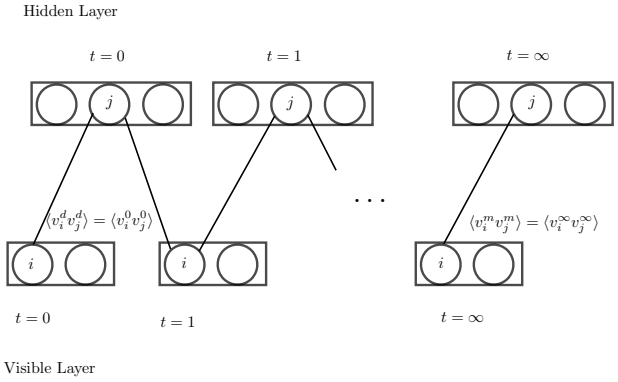

Figure 3.6: *Markov chain in which alternating Gibbs sampling is done. In one step of Gibbs sampling all the hidden units are updated simultaneously by receiving the inputs from the visible layer, then all the visible units are all updated simultaneously given the hidden units. The visible units are started at the data vector,* $\langle v_i^d h_j^d \rangle = \langle v_i^0 h_j^0 \rangle$*. After many steps the visible and hidden vectors are sampled from the stationary distribution defined by the current parameters of the model.*

Next, in Algorithm 2 we present the contrastive divergence update rule for RBM. The notation $a \sim P(\cdot)$ will mean that a random sample has been taken from $P(\cdot)$. We will also use the following conditional distributions when taking random samples in the algorithm. These are the distributions for a Gauss-Bernoulli RBM, where the visible units are changed from binomial to normal distributed to compensate for continuous data. The vector $\hat{\mathbf{h}}$ refers to the posterior with a hat to emphasise that this is a deterministic representation of $\mathbf{x}$.

We show once again the conditional distributions for the Gauss-Bernoulli RBM, which is the model we used to build our DBN. This model is suitable for continuous data, but the regular Binomial-Bernoulli RBM can also be used if the data is normalised to $[0, 1]$. As we noticed under training the cost function evolved better when using the Gauss-Bernoulli RBM as building blocks. The conditional distributions used in the contrastive divergence update rule for the RBM are given by

$$P(v_i = x|\mathbf{h}) = \frac{1}{\sigma_i\sqrt{2\pi}} \exp\left(- \frac{\left(x - a_i - \sigma_i \sum_j w_{ij} h_j\right)^2}{2\sigma_i^2} \right), \tag{3.45}$$

$$P(h_j = 1|\mathbf{v}) = \varphi\left(b_j + \sum_i \frac{v_i}{\sigma_i} w_{ij} \right). \tag{3.46}$$

Next we present the pseudo code for contrastive divergence. The weights are initialised from a uniform distribution, $\mathbf{a}$ is the bias of the visible layer, $\mathbf{b}$ is the bias of the hidden layer.

Algorithm 2: *Pseudo code for Restricted Boltzmann Machine Update (Contrastive Divergence)*

Training input $\mathbf{x}$, learning rate ϵ.
Initialisation:
weights $\mathbf{w}^i \sim U(-4 \cdot \sqrt{c}, 4 \cdot \sqrt{c})$, $c = 6/(n_{in} + n_{out})$
bias $\mathbf{b}^i = 0$
$\mathbf{w} \leftarrow \mathbf{w}^i$
$\mathbf{b} \leftarrow \mathbf{b}^i$
$\mathbf{a} \leftarrow \mathbf{b}^{i-1}$
Positive phase:
$\mathbf{v}^0 \leftarrow \mathbf{x}$
$\hat{\mathbf{h}}^1 \leftarrow \varphi(\mathbf{b} + \mathbf{w}\mathbf{v}^0)$
Negative phase, conditional distributions according to (eq 3.45 and eq 3.46):
$\mathbf{h}^0 \sim P(\mathbf{h}|\mathbf{v}^0)$
$\mathbf{v}^1 \sim P(\mathbf{v}|\mathbf{h}^0)$
Update:
$$\mathbf{w}^i \leftarrow \mathbf{w}^i + \epsilon\left(\hat{\mathbf{h}^0}(\mathbf{v}^0)^T - \hat{\mathbf{h}^1}(\mathbf{v}^1)^T\right)$$
$$\mathbf{b}^i \leftarrow \mathbf{b}^i + \epsilon\left(\hat{\mathbf{h}^0} - \hat{\mathbf{h}^1}\right)$$
$$\mathbf{b}^{i-1} \leftarrow \mathbf{b}^i + \epsilon(\mathbf{v}^0 - \mathbf{v}^1)$$

3.4 Training Deep Belief Networks

DBNs are composed of stacked Restricted Boltzmann machines. Training of a DBN is done by training the first RBM from the input instances and then the other RBMs in the DBN are trained sequentially. Patterns generated by the top RBM can be propagated back to the input layer using only the conditional probabilities as in a belief network. This arrangement is what defines a DBN.

When bottom-up training is performed, the top level RBM learns from the hidden layer below. When the top-down generation is performed the top-level RBM is the initiator of generative modelling. Recalling the picture of a DBN in Figure 2.14, data is generated as follows

1. An equilibrium sample is taken from the top-level RBM by performing Gibbs sampling for many time steps as shown in Figure 3.6 until equilibrium is reached.

2. A single top-down pass starting at the visible units of the top level RBM is used to stochastically pick the states of all the other hidden layers of the network, see Figure 3.7.

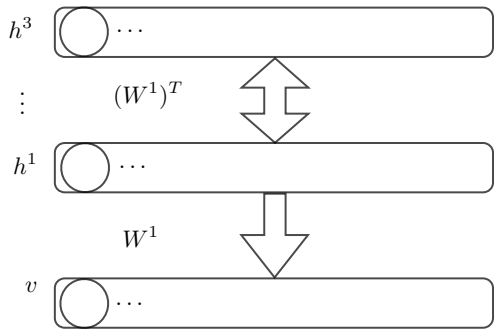

Figure 3.7: *A two-hidden layer Deep Belief Network with tied weights* $W^2 = (W^1)^T$. *The joint distribution defined by this DBN is identical to the joint distribution of an RBM,* $P(\mathbf{v}, \mathbf{h}; W^1)$.

A Greedy learning algorithm was designed to train DBN (Hinton & Osindero). Before its design it was considered rather hard to work with deep neural networks. To make the analysis simple we consider a DBN with only two hidden layers $\{\mathbf{h}^1, \mathbf{h}^2\}$ and follow (Salakhutdinov, 2009) closely. We will let the number of hidden units in the second hidden layer equal the number of visible layers. The top two layers, going from the visible layer and up in

the figure, represent an RBM and the lower layer is a directed sigmoid belief network. The joint distribution over $(\mathbf{v}, \mathbf{h}^1, \mathbf{h}^2)$ is

$$P(\mathbf{v}, \mathbf{h}^1, \mathbf{h}^2; \theta) = P(\mathbf{v}|\mathbf{h}^1; W^1)P(\mathbf{h}^1, \mathbf{h}^2; W^2), \tag{3.47}$$

where $\theta = \{W^1, W^2\}$ are the parameters of the model, $P(\mathbf{v}|\mathbf{h}^1; W^1)$ is the directed sigmoid belief network and $P(\mathbf{h}^1, \mathbf{h}^2; W^2)$ is the joint distribution over the second layer. These distributions have the following form

$$\begin{aligned} P(\mathbf{v}|\mathbf{h}^1; W^1) &= \prod_i p(v_i|\mathbf{h}^1; W^1), \\ P(v_i = 1|\mathbf{h}^1; W^1) &= g\Big(\sum_j W^1_{ij} h^1_j\Big), \\ P(\mathbf{h}^1, \mathbf{h}^2; W^2) &= \frac{1}{Z(W^2)} \exp\Big((\mathbf{h}^1)^T W^2 \mathbf{h}^2\Big). \end{aligned} \tag{3.48}$$

The greedy algorithm proceeds as follows. In our two hidden layer DBN we have so called tied parameters, meaning that the weights are defined as $W^2 = (W^1)^T$. Then the DBN's joint distribution is given by $P(\mathbf{v}, \mathbf{h}^1; \theta) = \sum_{\mathbf{h}^2} P(\mathbf{v}, \mathbf{h}^1, \mathbf{h}^2; \theta)$ which is identical to the RBM's joint distribution $P(\mathbf{v}, \mathbf{h}^1; W^1)$. Using our previous equations and $W^2 = (W^1)^T$, our DBN's joint distribution is given by, (Salakhutdinov, 2009),

$$\begin{aligned} P(\mathbf{v}, \mathbf{h}^1; \theta) &= P(\mathbf{v}|\mathbf{h}^1; W^1) \cdot \sum_{\mathbf{h}^2} P(\mathbf{h}^1, \mathbf{h}^2; W^2) \\ &= \prod_i p(v_i|\mathbf{h}^1; W^1) \cdot \frac{1}{Z(W^2)} \prod_i \Big(1 + \exp\Big(\sum_j W^2_{ji} h^1_j\Big)\Big) \\ &= \prod_i \frac{\exp\Big(v_i \sum_j W^1_{ij} h^1_j\Big)}{1 + \exp\Big(\sum_j W^1_{ij} h^1_j\Big)} \cdot \frac{1}{Z(W^2)} \prod_i \Big(1 + \exp\Big(\sum_j W^2_{ji} h^1_j\Big)\Big) \\ &= \frac{1}{Z(W^1)} \prod_i \Big(\exp\Big(v_i \sum_j W^1_{ij} h^1_j\Big)\Big) \\ &= \frac{1}{Z(W^1)} \exp\Big(\sum_{ij} W^1_{ij} v_i h^1_j\Big), \end{aligned}$$

were in the second to last line we have used that $W^2_{ji} = W^1_{ij}$ and $Z(W^1) = Z(W^2)$. This last joint distribution is identical to the RBM's distribution. For that reason the greedy learning algorithm is used with a stack of RBMs.

The steps of the algorithm are as follows. First the bottom RMB with parameters W^1 is trained. The second layer weights are initialised with

$W^2 = (W^1)^T$. This is to ensure that the two hidden DBN is at least as good as our RBM. Then improvement of the DBN is achieved by untying and redefining W^2. For any approximating distribution $Q(\mathbf{h}^1|\mathbf{v})$ the log-likelihood of the two hidden layer DBN model has the following *variational lower bound*, where the states $\mathbf{h}^2$ are summed out, (Salakhutdinov, 2009),

$$\begin{aligned}\log P(\mathbf{v};\theta) &\geq \sum_{\mathbf{h}^1} Q(\mathbf{h}^1|\mathbf{v}) \log P(\mathbf{v},\mathbf{h}^1;\theta) + H(Q(\mathbf{h}^1|\mathbf{v})) \\ &= \sum_{\mathbf{h}^1} Q(\mathbf{h}^1|\mathbf{v}) \log P(\mathbf{h}^1;W^2) + \log P(\mathbf{h}^1;W^1) + H(Q(\mathbf{h}^1|\mathbf{v})),\end{aligned} \tag{3.49}$$

here $H(\cdot)$ is the entropy functional. We let $Q(\mathbf{h}^1|\mathbf{v}) = P(\mathbf{h}^1|\mathbf{v};W^1)$, which is defined by the bottom RBM. The learning algorithm attempts to learn a better model for $P(\mathbf{h}^1;W^2)$ by freezing the parameter vector W^1 and maximising the *variational lower bound* with respect to W^2. The estimation procedure with frozen W^1 amounts to maximising

$$\sum_{\mathbf{h}^1} Q(\mathbf{h}^1|\mathbf{v}) \log P(\mathbf{h}^1;W^2), \tag{3.50}$$

this is maximum likelihood training of the second layer RBM with vectors $\mathbf{h}^1$ drawn from $Q(\mathbf{h}^1|\mathbf{v})$ as data. If the second layer of the RBM, $P(\mathbf{h}^1;W^2)$, is presented with N input vectors, the layer will learn a better model of the aggregated posterior over $\mathbf{h}^1$. This posterior is a mixture of factorial posteriors $\frac{1}{N}\sum_n P(\mathbf{h}^1|\mathbf{v}_n;W^1)$. We can continue in this manner training subsequent layers, where now the next hidden layer works as input to the next hidden layer. For the next level this means that we initialise $W^3 = (W^2)^T$ and we continue in this manner if more layers are added. Initialisation of the parameters before the greedy layer wise pre-training starts is as follows.

The weights are initialised from a uniform distribution with a range in the same regime as the range of the activation function, in our case the sigmoid activation and consequently, (Bengio, 2012),

$$W \sim U(-a, a), \tag{3.51}$$

where $a = 4\sqrt{\frac{6}{n_{in}+n_{out}}}$, n_{in} is the number of units in layer $(i-1)$ and n_{out} is the number of units in layer i. The bias for the visible and hidden layers are initialised to zero.

We conclude this Chapter by summarising the algorithms used for training DBNs. The algorithms can be found in (Larochelle et al., 2009). We use the symbol $\hat{\mathbf{h}}(\mathbf{x})$ to represent the posterior of $\mathbf{x}$ and to emphasise that it is deterministic. Let $\mathbf{h}^i$ be the output of layer i, $\mathbf{o}$ the output of the whole network, K the number of output from the network, ℓ the total number of layers and $|h^i|$ the size of layer i. As usual we let $\mathbf{w}$ denote the weight matrix between two layers and $\mathbf{b}$ the bias of each layer in the network while $\varphi(\cdot)$ denotes the sigmoid activation function.

Algorithm 3: Pseudo code for greedy layer-wise pre-training

Training set $\mathcal{D} = \{\mathbf{x}_t\}_{t=1}^T$, pre-training learning rate ϵ_p.Initialisation:
weights $\mathbf{w}_{jk}^i \sim U(-4 \cdot \sqrt{a}, 4 \cdot \sqrt{a})$, $a = 6/(n_{in} + n_{out})$
bias $\mathbf{b} = 0$
Pre-training:
for $i \in \{1, \ldots, \ell\}$
  while Error does not converge:
  Pick input example $\mathbf{x}_t$ from training set
  $\hat{\mathbf{h}}^0(\mathbf{x}_t) \leftarrow \mathbf{x}_t$
  for $j \in \{1, \ldots, i-1\}$
    $\mathbf{a}^j(\mathbf{x}_t) = \mathbf{b}^j + \mathbf{w}^j \hat{\mathbf{h}}^{j-1}(\mathbf{x}_t)$
    $\hat{\mathbf{h}}^j(\mathbf{x}_t) = \varphi(\mathbf{a}^j(\mathbf{x}_t))$
  end for
  With $\hat{\mathbf{h}}^{i-1}(\mathbf{x}_t)$ as input:
  Update $\mathbf{w}^i$, $\mathbf{b}^i, \mathbf{b}^{i-1}$ with contrastive divergence (CD).
  end while
end for

We have divided the training in a set of three algorithms. First we present the pseudo code for pre-training which is applied to the DBN module. The second algorithm, back-propagation is applied to the entire network DBN-MLP. This is step is for fine-tuning the parameters of the network. The last algorithm is a call to this algorithms.

Algorithm 4: Pseudo code for fine-tuning

$\mathcal{D} = \{\mathbf{x}_t, y_t\}_{t=1}^{T}$, fine-tuning learning rate, ϵ_f.
while Error does not converge:
  Pick input example $(\mathbf{x}_t, y_t)$ from training set
  Forward propagation:
  $\hat{\mathbf{h}}(\mathbf{x}_t) \leftarrow \mathbf{x}_t$
  for $i \in \{1, \ldots, \ell\}$
    $\mathbf{a}^i(\mathbf{x}_t) = \mathbf{b}^i + \mathbf{w}^i \hat{\mathbf{h}}^{i-1}(\mathbf{x}_t)$
    $\hat{\mathbf{h}}^i(\mathbf{x}_t) = \varphi(\mathbf{a}^i(\mathbf{x}_t))$
  end for
  $\mathbf{a}^{\ell+1}(\mathbf{x}_t) = \mathbf{b}^{\ell+1} + \mathbf{w}^{\ell+1} \hat{\mathbf{h}}^{\ell}(\mathbf{x}_t)$
  $\mathbf{o}(\mathbf{x}_t) = \hat{\mathbf{h}}^{\ell+1}(\mathbf{x}_t) = \text{softmax}(\mathbf{a}^{\ell+1}(\mathbf{x}_t))$
  Backward gradient propagation and parameter update:
  $\frac{\partial \log o_{y_t}(\mathbf{x}_t)}{\partial a_j^{\ell+1}(\mathbf{x}_t)} \leftarrow \mathbf{I}(y_t = j) - o_j(\mathbf{x}_t)$ **for** $j \in \{1, \ldots, K\}$
  $\mathbf{b}^{\ell+1} \leftarrow \mathbf{b}^{\ell+1} + \epsilon_f \frac{\partial \log o_{y_t}(\mathbf{x}_t)}{\partial \mathbf{a}^{\ell+1}(\mathbf{x}_t)}$
  $\mathbf{w}^{\ell+1} \leftarrow +\epsilon_f \frac{\partial \log o_{y_t}(\mathbf{x}_t)}{\partial \mathbf{a}^{\ell+1}(\mathbf{x}_t)} \hat{\mathbf{h}}^{\ell}(\mathbf{x}_t)^T$
  for $i \in \{1, \ldots, \ell\}$, in decreasing order
    $\frac{\partial \log o_{y_t}(\mathbf{x}_t)}{\partial \hat{\mathbf{h}}_j^i(\mathbf{x}_t)} \leftarrow (\mathbf{w}^{i+1})^T \frac{\partial \log o_{y_t}(\mathbf{x}_t)}{\partial \mathbf{a}^{i+1}}$
    $\frac{\partial \log o_{y_t}(\mathbf{x}_t)}{\partial a_j^i(\mathbf{x}_t)} \leftarrow \frac{\partial \log o_{y_t}(\mathbf{x}_t)}{\partial h_j^i(\mathbf{x}_t)} \hat{h}_j^i \left(1 - \hat{h}_j^i(\mathbf{x}_t)\right)$ **for** $j \in \{1, \ldots, |\hat{\mathbf{h}}^i|\}$
    $\mathbf{b}^i \leftarrow \mathbf{b}^i + \epsilon_f \frac{\partial \log o_{y_t}(\mathbf{x}_t)}{\partial \mathbf{a}^i}$
    $\mathbf{w}^i \leftarrow \mathbf{w}^i + \epsilon_f \frac{\partial \log o_{y_t}(\mathbf{x}_t)}{\partial \mathbf{a}^i} \hat{\mathbf{h}}^{i-1}(\mathbf{x}_t)^T$
  end for
end while

The pseudo code for the whole network is a call to the pre-training phase and then a call to the fine-tuning phase.

Algorithm 5: Pseudo code for training DBN

Pre-training:
Estimate parameters of RBMs:
$\{\mathbf{w}, \mathbf{b}\} \leftarrow \text{pre-train}(\mathbf{x}_t)$
Fine-tune the whole network, DBN-MLP:
$\{\mathbf{w}, \mathbf{b}\} \leftarrow \text{fine-tuning}(\mathbf{x}_t, \mathbf{y}_t)$

3.4.1 Implementation

The implementation is done in Theano, a compiler for mathematical expressions in Python. This library is optimised to native machine language and one of its strengths is that it provides symbolic differentiation. Theano translates expressions to C++ when programming for the CPU or to CUDA when programming for the GPU and compiles them to dynamically loaded Python modules. According to (Bergstra at el., 2010), common learning algorithms implemented with Theano are 1.6× to 7.5× faster than competitive alternatives, e.g. implemented with C/C++, Numpy/Scipy and MATLAB when tested on the same computer architecture.

Chapter 4

Financial Model

We suppose that stock prices are samples from the same distribution. In that way we can use all the stocks' log-returns indiscriminately. We make the following assumption:

There are correlations between most stocks in the long term and therefore most stocks on average move in the same direction.

With this assertion we assume that on average the stock markets move in the same direction in the long term. In the short term it may happen that there are not any correlations at all but eventually there will be correlations. We believe that there are correlations between most stocks pushing them in the same direction, either up or down.

4.1 The Model

The model chosen in this master thesis is a DBN coupled to a MLP. The DBN is at the interface with the environment and receives the input signal. The input signal passes through each layer until it reaches the hidden layer in the MLP. From the MLP's hidden layer the signal reaches the output layer which is a softmax layer with two output neurons for classification to the two classes 0 or 1, to be explained shortly. The MLP has one hidden layer containing as many neurons as the last layer of the DBN. The number of layers and computing units are chosen so as to minimize the validation error and test errors. Our system's task is to pick stocks from the S&P 500 and we verify the results by comparing the training error to the validation and test errors.

4.1.1 Input Data and Financial Model

Our study is similar to the one presented in (Takeuchi et al., 2013). We prepared our data according to their assumptions, but we use log-returns and solve our task with a DBN instead of an auto encoder. We believe that the DBN should also yield good results as it has been used to model time series models (Kuremoto et al., 2014) and stock decision systems (Zhu and Yin, 2014) with reported good results in those papers. The log-returns for both the monthly returns and the daily returns are defined as follows

$$r^i(t) = \log\left(\frac{S_t^i}{S_{t-1}^i}\right),$$

where S_t^i is the price of stock i at time t.

The input to our model is a matrix with 33 variables or features. These features are the $t-2$ trough $t-13$ monthly log-returns and the 20 daily log-returns for each stock at month t. Stocks with closing prices below \$5 at the last day of trading for each month are discarded from the data, but this does not mean those stocks will not contribute at all to the data.

It will happen that when preparing our feature vector for each stock per month it will have returns from the past which might be calculated using closing prices below \$5, this will introduce some noise to the data but our hope is to diminish the amount of noise through this procedure. An indicator variable $I(\cdot)$ is used to mark in which month the stocks are picked. It takes the value 1 if the month is January and 0 otherwise to account for the turn of the year effect, (Jaggedish and Titman, 1993). The stocks are normalized according to their Z-scores and in doing so we used the mean for all the stocks in each month in the same list. Similar computations where made for the standard deviation.

If we let the monthly log-returns at time-step t be $r_m^i(t)$ and $r_d^i(t)$ be the 20-daily returns at time-step t, then our model or hypothesis is given by the following equations

$$\begin{aligned} Y_{t+1}^i &= F_\theta\Big(r_d^i(t), \ldots, r_d^i(t-19), r_m^i(t-2), \ldots, r_m^i(t-13), I(t=\mathrm{J})\Big), \\ \hat{Y}_{t+1} &= \mathbb{E}[Y_{t+1}^i]. \end{aligned} \tag{4.1}$$

Here F_θ is the function which our DBN-MLP is modelling and θ is the parameter set of this model, Y_{t+1} takes the value 0 or 1 corresponding to two different classes. Those classes are defined according to if the monthly log-return of stock i, is below or above the median of the next month's log-returns for all the stocks in the list as described below.

The labels are assigned in the following manner. First we compute the log-returns for all stocks at month t. Then we compute the median for all those log-returns by computing

$$\text{Median}\left[r(t)^{(\text{All stocks})}\right] = \text{Median}\left[\log\left(\frac{S_t^{(\text{All stocks})}}{S_{t-1}^{(\text{All stocks})}}\right)\right].$$

Next we compare the log-return at time t, $\log(S_t^i/S_{t-1}^i)$, for each stock i with the median and assign labels according to (Takeuchi et al., 2013) i.e. we have the following expressions:

$$\log\left(\frac{S_t^i}{S_{t-1}^i}\right) \leq \text{Median}\left[r(t)^{(\text{All stocks})}\right], \quad \text{assign label 0}$$
$$\log\left(\frac{S_t^i}{S_{t-1}^i}\right) > \text{Median}\left[r(t)^{(\text{All stocks})}\right], \quad \text{assign label 1.}$$

To make predictions at time $t+1$, those labels will not help much as they are. Those labels, which are computed at time t, are assigned to the individual stocks' log-returns at time step $t-1$.

By shifting the labels from time step t to time step $t-1$ we can make predictions one time step later. We end up with labels computed at time t and assign those labels to our log-returns $\log(S_{t-1}^i/S_{t-2}^i)$ for each stock S^i at time $t-1$. Here comes the most important part. This procedure ensures that we compare the individual stocks' log-returns with its log-returns in the future. The future log-returns are compared to the median of all stocks. So in a sense we are comparing our log-returns at time t with a future median at time $t+1$ that is unknown to us but that we want our DBN-MLP to learn.

This is a regression in the class space where we fit a curve to the boundary separating the classes $\{0, 1\}$. This boundary is represented by all the stocks' median. Because we do not have any labels with the data gathered we use this procedure to create those ourselves.

The input vector at time $t-1$ is constructed by putting together the 20 daily log-returns for month $t-1$, and monthly log-returns for months $t-2, \ldots, t-13$ as well as the indicator function to account for if the month is January. To this input vector there will be a corresponding label computed at time t but that we shift to time $t-1$ as was just described. We then interpret a 1 as a signal to buy the stock and a 0 as a signal not to buy or sell if we own the stock.

More formally we can express our model in the following way. Let our labels be given by $y = \{0, 1\}$ and our input vector be given by the log-returns

$$\mathbf{x} = \left\{r_d^i(t), \ldots, r_d^i(t-19), r_m^i(t-2), \ldots, r_m^i(t-13), I(t = \text{J})\right\},$$

containing 33 elements, that is $\mathbf{x} \in \mathbb{R}^{33}$. If we have k training points, our training data can be denoted by $(\mathbf{x}^k, y^k)$, and our neural network will learn the model F_θ

$$F_\theta : \mathbf{x} \to y, \tag{4.2}$$

mapping the input vector space, $\mathbf{x}$ to the output space $y = \{0, 1\}$.

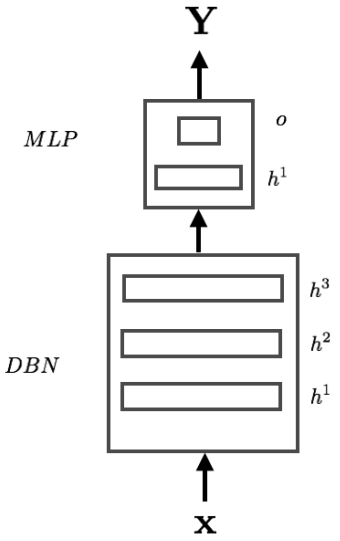

Figure 4.1: *Sketch of DBN-MLP consisting of a 3-layered DBN and a MLP with 1 hidden layer.* h^i *for* $i = \{1, 2, 3\}$ *represents the hidden layers,* o *is the output layer consisting of the softmax function. The input vector has 33 features or variables and* Y*, takes one of the values in the set* $Y = \{0, 1\}$.

The mapping of the input space to the output space through the network is shown in the sketch of Figure 4.1. The two classes or labels used for the classification task are 0 for stocks with monthly log-returns lying under the median or being equal to the median and 1 for monthly log-returns above the median at time t.

Chapter 5

Experiments and Results

Here we make an account on how the experiments were conducted and present the results that we obtained. Data from the S&P 500 was gathered for the period 1 January 1985 to 31 December 2006. The training data is further partitioned in a training set, a validation set and a test set and consists of 134 350 examples. This gives an input matrix of dimension 134 350 $\times$ 33 and a vector of 134 350 labels.

The training data was partitioned as follows:

- 70% of the data for training
- 15% of the data for validation
- 15% of the data for testing

The validation set is used for cross-validating the model to choose hyper parameters. Those parameters are the learning rate for pre-training and fine-tuning, the number of hidden layers, the number of neurons, the weight decay L_2 and the L_1 regularisation. These last two parameters are useful in training because they help in the optimisation so that the weights don't get saturated during training. We work with the Gauss-Bernoulli RBM as building block in the DBN as we noticed that the cost function had better convergence when making this change. We used weight decay and L_1 regularisation.

One important thing to have in mind when using Gauss-Bernoulli RBMs is that learning rates have to be many orders smaller than in the Binomial-Bernoulli RBMs to prevent the cost function from blowing up, (Hinton, 2010), which we experienced on tests.

We tested Persistent Contrastive Divergence PCD_1 instead of the standard method of Contrastive Divergence with one step of Gibbs block sampling (CD_1). Persistent contrastive divergence PCD_1, keeps track of the states of a number of persistent chains or "fantasy particles" instead of initialising each alternating Gibbs Markov chain at a data vector, see (Hinton, 2010)

for details. Hinton suggests to use PCD if the aim is to build the best density model for the data. Our tests did not show any difference in the validation and test errors when comparing PCD_1 to CD_1. The pre-training algorithm performs better with CD_1 than with CD_k for $k > 1$ steps in the Gibbs sampling procedure.

In (Larochelle et al., 2009) it is pointed out that in many situations a DBN with 3 hidden layers can perform better than one with 4 hidden layers. This might not apply to all circumstances and we have not made an extensive study of this subject. We performed some tests with deeper networks which yielded the same errors as with a network with 3 hidden layers. We therefore choose to fix the number of layers in the DBN to three hidden layers and instead try to optimise with respect to the number of neurons in each layer. The pre-training learning parameter affects the convergence of the cost function while the leaning parameter used in fine-tuning affects learning. How well our model learns is measured by how low validation error and test error we achieve.

Another observation made by (Larochelle et al., 2009) is that the best performing network was the one with the same number of neurons in all hidden layers. They point out that this might be a consequence of using the same unsupervised learning parameters for each layer, which we also did in our experiments. We tried different numbers of neurons and achieved as good results as keeping the same number of neurons.

5.1 Experiments

When conducting the experiments we reasoned in the following way. Because it was computationally expensive to use search algorithms to perform parameter optimisation we followed some guidelines and did not automatise the search for parameters. The guidelines found in (Hinton, 2010) on training RBMs are helpful, and also those found in (Bengio, 2012). The time consuming task of searching for the best parameters is due to the following facts.

We need to optimise with respect to learning parameters for the greedy layer-wise part and for the fine tuning part of the training algorithms. Other parameters which need to be found are the number of layers, the number of neurons, the weight decay parameter L_2, the regularisation parameter L_1, the number of epochs in the layer-wise pre-training and in the fine-tuning part. To this we add the amount of data used in training. Using an algorithm for searching the best parameters is the best way to do training, but it is computationally expensive. It requires computer clusters or at least programming in the computer Graphics Processing Unit (GPU).

To choose parameters we always test against the validation error and apply early stopping to prevent overfitting. We choose the parameters that give us the lowest validation error. When it concerns the number of epochs in the greedy layer-wise part we found that 100 epochs was enough, in some experiments it seems that even less epochs give better results with respect to the validation error and test error. To make training manageable we choose to keep 100 epochs. For the fine-tuning part we began with 3000 epochs and went down to 1000 epochs. We found that 1000 epochs gave better result with respect to the validation error and the test error.

We tested if a different number of computing units in the hidden layers affected the results and found that some configurations gave worse results or at least as good results as having the same amount of units in all the hidden layers. We choose to use the same number of units in all the layers in the DBN, see for example (Larochelle et al., 2009). We decided at the creation of the network to have only 1 hidden layer in the MLP-module. After all these decisions were made we had only to keep track of the following parameters:

- Learning parameter for the layer-wise pre-training.
- Learning parameter for fine-tuning with back-propagation.
- Number of computing units in all the hidden layers in the DBN (the same for all layers).
- The weight decay parameter.
- L_1 regularisation parameter.

5.2 Benchmarks

To compare how well our model learns, we make experiments with a logistic network and a MLP and use the results from those models as benchmark. The logistic network performs logistic regression. The MLP has one hidden layer and one softmax output layer with 2 output neurons. We also test our results to a naive benchmark given by $Y_{t+1} = Y_t$.

The logistic net has only one learning parameter which we can adjust, besides the number of epochs. In the MLP we can tune more parameters, for example the number of hidden units, the number of epochs, the learning parameter and the L_1 and L_2 regularisation terms in the the cost function. These regularisation terms are added to the cost function as penalty constraints to help our model to learn better. L_2 is also known as weight decay which is the term we used when training the DBN.

The network architectures are unchanged under all experiments and are as follows:

- The DBN-MLP consists of three hidden layers in the DBN module, 1 hidden layer in the MLP-module.
- The Logistic regression consists of one neuron.
- The MLP consists of one hidden layer.

Recall that in the DBN-MLP we use the same number of computing units in each layer. When the number of neurons is presented in our results, it should be interpreted as the number of neurons in all layers.

To simplify the information in the tables we introduce the following notation.

- The pre-training learning rate for DBN-MLP is given by ϵ_p.
- The fine-tuning learning parameter for DBN-MLP is given by ϵ_f.
- The learning parameter for logistic regression and MLP is given by ϵ.
- The validation error is given by E_V.
- The test error is given by E_T.

5.3 Results

In Table 5.1 the results from the naive benchmark are presented. The results consist of the validation and test error. The test error is a measure of the predictive power of the model. In Table 5.2 we present the results from the logistic regression.

E_V (%)	E_T (%)
50.03	50.93

Table 5.1: *Naive benchmark. The validation error is E_V and the test error is E_T.*

E_V (%)	E_T (%)
49.96	50.74

Table 5.2: *Logistic Regression. The validation error is E_V and the test error is E_T.*

The results obtained from the multilayer perceptron are shown in Table 5.3. Here there are two more parameters, L_1 and L_2 regularisation. We found that $L_1 = L_2 = 0.0001$ gave the best results with respect to the validation error. We found also that the learning rate with value 0.1 gave the best results. For that reason we keep those values constant throughout the experiments. The best result was achieved for a network with 20 neurons.

Neurons	ϵ	E_V (%)	E_T (%)
20	0.1	49.66	50.84

Table 5.3: *Multilayer Perceptron. The learning parameter is ϵ, E_V is the validation error and E_T is the test error.*

Neurons	ϵ_p	ϵ_f	E_V (%)	E_T (%)
100	10^{-10}	10^{-2}	47.39	48.67
100	10^{-10}	10^{-3}	47.96	48.64
100	10^{-11}	10^{-3}	47.86	48.29
300	10^{-11}	10^{-4}	48.16	47.17
400	10^{-11}	10^{-3}	46.52	47.11
500	10^{-11}	10^{-4}	47.24	47.56
500	10^{-11}	10^{-3}	46.93	47.50

Table 5.4: *Deep Belief Network (DBN-MLP). The pre-training learning rate is ϵ_p, ϵ_f is the fine-tuning learning parameter, E_V is the validation error and E_T is the test error.*

Table 5.4 shows the results from the experiments made with the DBN-MLP. Recall that the DBN has three hidden layers and that each layer has the same number of computing units. This restriction imposes the same number of hidden units to the MLP as those neurons receive its input from the DBN. Tests showed that 0.001 and 0.01 were good values for L_1 regularisation and weight decay, L_2, respectively.

From the results we can see that the DBN-MLP is a robust model. We can also see the fact that the predictive power of our network is satisfactory but it can be improved.

5.3.1 Summary of Results

To make Table 5.5, with the final results, easier to read we introduce some notation. Let DBN stand for the DBN-MLP model and let LReg stand for the logistic regression.

The model for DBN-MLP with 400 neurons the lowest validation error. Our results show that often the validation error is lower than the test error. From Table 5.5 we see that DBN-MLP performs better than all the bench-

Model	Neurons	ϵ_p	ϵ_f	E_V (%)	E_T (%)
Naive	-	-	-	50.03	50.93
LReg	-	-	-	49.96	50.74
MLP	20	-	0.1	49.66	50.84
DBN	400	10^{-11}	10^{-3}	46.52	47.11

Table 5.5: *Result from DBN-MLP and benchmarks.*

marks. Although the results are satisfactory, there is room for improvement. Things in our list which need improvement are:

- Automatically searching for the best parameters.
- The assumptions made in our mathematical model seem to be too general to capture the true underlying model.
- Alternatively we could use our model to study stocks by sector.
- Try to make predictions in a short term.

Deep learning is a good candidate to solve problems in finance. We can see that the test error is a good measure of how well the model learns the underlying distribution.

Chapter 6

Discussion

The main issue of this thesis has been to implement a DBN coupled to a MLP. As benchmark we used results obtained from a logistic regression network and a multilayer perceptron. Similar tests were performed on all networks. We used the DBN-MLP to pick stocks and predict market movements. The main assumption in solving this problem has been that stocks are correlated with each other in the long term. In that way we can use stock prices to compute log-returns and consider them as samples from the same distribution.

The DBN is built by a stack of RBMs. These RBMs are stochastic networks whose probabilities were presented in the Chapter on the theory of DBN. The MLP is a well known neural network and the theory surrounding it was presented in the Chapter on neural networks. DBN-MLP is a deep neural network, which is trained with the methods of deep learning. The name deep neural network refers to networks with more than two hidden layers. Deep learning is summarised by a set of training algorithms which is a two step procedure. The first step consists of applying greedy layer-wise training on the DBN and the second step is to apply back-propagation training on the whole network to fine-tune the parameters of the DBN-MLP.

We tested different approaches like running our algorithm with many steps of Gibbs sampling compared to using only one step. The main approach in deep learning is to use only one step as this yields good results, which we also experienced. We tested both Contrastive Divergence (CD) and Persistent Contrastive Divergence (PCD), explained in earlier Chapters. We chose to use CD as this method was computationally more efficient.

This work consisted of a large amount of tests to investigate the impact of all the parameters in the model. We list the parameters involved:

- Pre-training learning rate.
- Fine-tuning learning rate.
- Number of epochs or iterations for Pre-training
- Number of epochs for fine-tuning.
- Number of hidden layers in DBN.
- Number of neurons in each layer.
- Number of steps of CD.
- L_1 regularisation.
- Weight decay, also known as L_2 regularisation.

We also tested different initialisation values for the weights as there are different guidelines. The visible layer in the RBM was tested with binomially distributed units as well as with normally distributed. The results presented in this work are produced with RBMs with normally distributed visible units (Gauss-Bernoulli RBMs). To choose a certain parameter different configurations were tested to assure that the impact of the parameter was the same on all configurations used for the results presented in this work.

Data from S&P 500 was gathered for the period 1 January 1985 to 31 December 2006. The data was partitioned as: 70% of the data was used for training, 15% was used for validation and 15% was used for testing. It is considered a good training procedure to keep track of both the validation error and the test error. These quantities are considered to be measures of the predictive power of DBN-MLP when faced to new unseen data coming from the same distribution as the training data.

The results obtained from the S&P 500 were satisfactory. The validation error was 46.52% and the test error was 47.11%. The test error is considered as a measure for the predictive power of our model to unseen data.

Our work has given satisfactory results and it has raised more interesting questions and given some hints on how we can continue the research of deep learning in finance. Is it possible to get better results if we study stocks belonging to the same market sectors? We believe that better results can be obtained if we study markets by sector, as there can be stronger correlations between stocks in the same sector than in our more general assumption.

There is room for improvement, for instance we could build a model which uses more recent data and less monthly log-returns. We could also use intraday data and other variables such as highest price or lowest price for each trading day. The financial assumptions made in this work can be changed and the mathematical model can be improved. We could use market indicators that correlate better with stock markets, for example we could use currencies or other asset classes. Even the use of certain qualitative data could improve the performance e.g. figures from financial statements etc.

The optimisation procedure for finding the best parameters of the model can be automatised by the use of optimisation algorithms. This path was not used here due to the computation time needed. Often such tasks need computer clusters or at least that we program the algorithms in the computer's Graphics Processing Unit (GPU). The parameters were instead found by choosing from a certain set of parameters that seem reasonable and that often appear in deep learning artificial intelligence tasks. We followed recommendations made in the literature, see for example the guidelines given by (Hinton, 2010) or (Larochelle et al., 2009).

Another approach would be to test other types of neural networks coupled to our DBN. We could also solve a regression problem instead of a classification problem. The DBN-MLP seems to be more stable and reliable than the simpler networks we used as benchmarks and therefore a good candidate to continue research of the stock markets.

Appendices

Appendix A

Appendix

A.1 Statistical Physics

The distribution of a many-particle system in state i, with probability p_i has the following properties

$$\begin{aligned} p_i &\geq 0, \quad \forall i \\ \sum_i p_i &= 1. \end{aligned} \tag{A.1}$$

If the system is in state i its energy is E_i and if the system is in thermal equilibrium the probability of being in state i is

$$p_i = \frac{1}{Z} \exp\left(- \frac{E_i}{k_B T} \right), \tag{A.2}$$

where T is the absolute temperature, $k_B = 1.38 \times 10^{-23}$ joules/kelvin is Boltzmann's constant and Z is a constant independent of the states, defined as

$$Z = \sum_i \exp\left(- \frac{E_i}{k_B T} \right). \tag{A.3}$$

p_i is known as the canonical distribution, Gibbs distribution or Boltzmann distribution. From the Boltzmann distribution we can note the following

1. States of low energy have a higher probability of occurrence than states of high energy.

2. As the temperature T is reduced, the probability is concentrated on a smaller subset of low-energy.

In machine learning we make use of this distribution in the construction of the Boltzmann machine but we let $k_B = 1$. The free energy of a physical system is defined as

$$F = -T \log Z, \tag{A.4}$$

and the average energy of the system is

$$\langle E \rangle = \sum_i p_i E_i, \tag{A.5}$$

where $\langle \cdot \rangle$ is the mean taken on the ensemble of particles. It can be shown that the mean energy can also be expressed as

$$\langle E \rangle = F - T \sum_i p_i \log p_i = F + TH, \tag{A.6}$$

where the entropy of the system is defined by $H = -\sum_i p_i \log p_i$.

A.1.1 Logistic Belief Networks

The logistic belief network is a model with binary stochastic units. The probability of turning on unit i is given by the logistic function of the states of its immediate ancestors j and weights w_{ij} on the directed connections from the ancestors

$$p(s_i = 1) = \frac{1}{1 + \exp\left(-b_i - \sum_j s_j w_{ij}\right)}. \tag{A.7}$$

In the above equation b_i is the bias of unit i.

A.1.2 Gibbs Sampling

Gibbs sampling of the joint distribution of N random variables $S = (S_1, S_2, \ldots, S_N)$ is achieved by a sequence of sampling steps of the form

$$S_i \sim p(S_i | S_{-i}), \tag{A.8}$$

where S_{-i} is the set of random variables excluding S_i. In our setting of the restricted Boltzmann machine our S will contain visible and hidden units. In the RBM the visible and hidden units are conditional independent and therefor we can perform block Gibbs sampling and is of the form

$$\begin{aligned} h^{(n+1)} &\sim \varphi(W^T v^{(n)} + c), \\ v^{(n+1)} &\sim \varphi(W h^{(n+1)} + b), \end{aligned} \tag{A.9}$$

where $h^{(n)}$ is the set of all hidden units at the nth step of the Markov chain. This sample procedure is very costly and its convergence requires $t \to \infty$ to be certain that $(v^{(t)}, h^{(t)}) \sim p(v, h)$.

Formally Gibbs sampling is a method for simulating stochastic variables and as mentioned before it is used in the training of the RBM. The Gibbs sampler generates a Markov chain with the Gibbs distribution as its stationary distribution. The transition probabilities are non-stationary. We have

then a K-dimensional random vector $\mathbf{X}$ with components $X_1, X_2, \ldots X_K$. Suppose that we know the conditional distribution of X_k given values of all the other components of $\mathbf{X}$ for $k = 1, 2, \ldots, K$. Now we want to obtain a numerical estimate of the marginal density of the random variable X_k for each k. The Gibbs sampler proceeds by generating a value for the conditional distribution for each component of the random vector $\mathbf{X}$, given the values of all the other components of $\mathbf{X}$. Starting from a configuration $\{x_1(0), x_2(0), \ldots, x_K(0)\}$, we make the following drawing on the first iteration of Gibbs sampling

$$
\begin{aligned}
&\text{draw} \quad x_1(1) \sim f_1(X_1|x_2(0), x_3(0), \ldots, x_k(0)),\\
&\text{draw} \quad x_2(1) \sim f_1(X_2|x_1(1), x_3(0), \ldots, x_k(0)),\\
&\vdots\\
&\text{draw} \quad x_k(1) \sim f_1(X_k|x_1(1), \ldots, x_{k-1}(1), x_{k+1}(0), \ldots, x_{k+1}(0), \ldots, x_K(0)),\\
&\vdots\\
&\text{draw} \quad x_K(1) \sim f_1(X_K|x_1(1), x_2(1), \ldots, x_{K-1}(1)),
\end{aligned}
$$

where $f(X_k|X_{-k})$ is the conditional distribution of X_k given the other components X_{-k}. This procedure can be repeated a number of iterations in the sampling scheme. In the sampling we will visit every component in the random vector $\mathbf{X}$ which will give us a total of K new variates on each iteration. We notice that a new value for X_{k-1} is immediately used when a new value of X_k is drawn for $k = 2, 3, \ldots, K$.

After n iterations of the Gibbs sampler we generate K variates

$$X_1(n), X_2(n), \ldots, X_K(n).$$

The Gibbs sampler has f as its stationary distribution. Under mild conditions the ergodic theorem states that for any function g of random variables $X_1, X_2, \ldots, X_K$ whose expectation exist the following holds

$$\lim_{n\to\infty} \frac{1}{n} \sum_{i=1}^{n} g(X_1(i), X_2(i), \ldots, X_K(i)) \to \mathbb{E}[g(X_1, X_2, \ldots, X_K)], \quad \text{(A.10)}$$

with probability 1.

A.1.3 Back-Propagation: Regression

In Figure A.1 we can see how well MATLAB's implementation of the back-propagation is in generalising when presented to new data. In the next figure, Figure A.2, we can see that the model gets worse at generalisation when presented to new data. That is because the model overfits the data with the number of neurons, in this case 170 neurons in the hidden layer. In

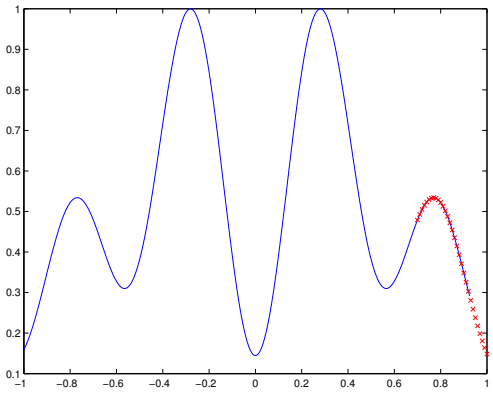

Figure A.1: *MATLAB's implementation of back-propagation. Network with one hidden layer containing 8 neurons.*

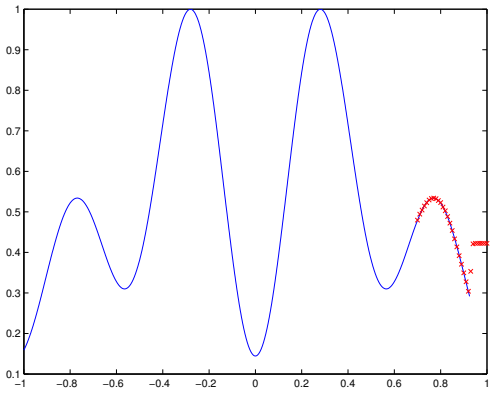

Figure A.2: *MATLAB's implementation of back-propagation for a network with 170 neurons.*

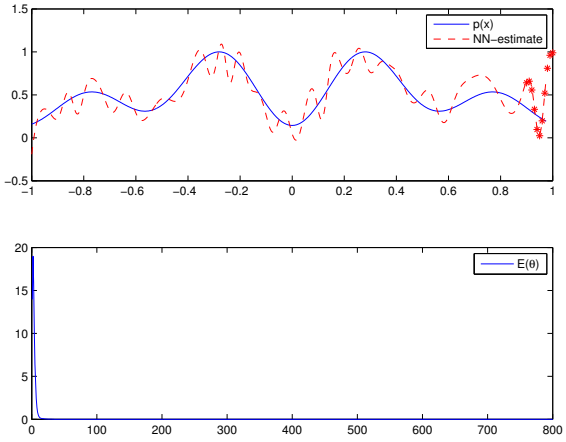

Figure A.3: *Network with 170 neurons in the hidden layer. Implemented in that batch-mode of learning.*

the last figure, Figure A.3, we can see that the generalisation power of the network has deteriorated and it is not so good at new data. This model is implemented in the batch-mode of learning.

A.2 Miscellaneous

Definition of Neural network according to, (Haykin, 2009):

Definition A.1 *A neural network is a massively parallel distributed processor made up of simple processing units that has the attribute of storing experiential knowledge and making it available for use. It resembles the brain in two respects:*

1. *Knowledge is acquired by the network from its environment through a learning process.*

2. *Interneuron connection strengths, known as syntactic weights, are used to store the acquired knowledge.*

In the process of learning, a neural network use what is called a learning algorithm. The purpose of the learning algorithm is to modify the synaptic weights of the network to attain a desired design objective. Knowledge can be defined as in the following definition, (Haykin, 2009):

Definition A.2 *Knowledge refers to stored information or models used by a person or a machine to interpret, predict, and appropriately respond to the outside world.*

The computing power of neural networks derives from its parallel distributed structure and its ability to learn or generalize. Generalization is defined as the production of reasonable outputs for inputs not encountered before during training. These two properties of neural networks makes it possible to find good approximate solutions to complex or large scale problems. Neural networks offer the following properties:

1. **Nonlinearity**. A neural network made up of nonlinear neurons is itself nonlinear.

2. **Input-Output Mapping**. As an example we can mention supervised learning or learning with a teacher. This refers to the situation where we modify the synaptic weighs of the network by applying labeled training examples or task examples. An example consists of an input signal which we send through the system and compare it to a desired response. We proceed to send input signals picked at random and modify the weights to minimize the difference between the desired response and the actual response of the network produced by the input signal. The minimization is done in accordance with a statistical criterion. The training of the network is repeated for many examples until the network reaches a steady state. If we repeat the training by applying the input signals in a different order then the network eventually will learn. For the learning process the neural network constructs an *input-output mapping* for the presented problem.

3. **Adaptivity**. Neural networks have a built-in capability to adapt their synaptic weights to changes in the surrounding environment. This adaptability make neural networks capable of performing pattern classification, signal processing, and control tasks. There is a trade off where the networks can be made robust by making them more adaptable to non stationary data. But making the system too adaptable may worsen the response if the time constants are too small. In that case the system responds to spurious disturbances causing a degradation in the system performance.

4. **Evidential Response**. In pattern classification, a neural network is designed to select a particular pattern with a degree of confidence in the decision made.

5. **Contextual Information**. Knowledge is represented by the very structure and activation state of a neural network. Every neuron in the network is potentially affected by the other neurons in the network. For this reason contextual information is processed effectively by neural networks.

6. **Fault Tolerance**. A neural network is fault tolerant which means that the network is capable of robust computation even though there is some damage in the links connecting the neurons in the network.

7. **Uniformity of Analysis and Design**. In the area of neural networks we can use theories and algorithms from different technical, scientific and engineering applications.

One major task of neural networks is to learn a model of the world. Knowledge of the world can be categorised as

1. Prior information, which simply means known facts of the state of the world.

2. Observation, in this case we gather measurements from the environment.

Bibliography

[Bastien et al., 2012] Bastien, F; Lamblin, P; Pascanu, R; Bergstra, J; Goodfellow, I; Bergeron, A; Bouchard, N; Warde-Farley, D and Bengio, Y. *Theano: New Features and Speed Improvements.* NIPS 2012 Deep Learning Workshop.

[Bengio, 2009] Bengio, Yoshua. *Learning Deep Architectures For Artificial Intelligence.* Foundations and Trends in Machine Learning, Vol. 2 No. 1 (2009) 1-127. ¸2009 Y. Bengio DOI: 10.1561/2200000006.

[Bengio et al., 2006] Bengio, Yoshua; Lamblin, Pascal; Popovici, Dan; Larochelle, Hugo. *Greedy Layer-Wise Training of Deep Networks.* Technical Report 1282, Département d'Informatique et Recherche Opérationelle. August 21, 2006.

[Bengio, 2012] Bengio, Yoshua. *Practical Recommendations for Gradient-Based Training of Deep Architectures.* Version 2, Sept. 16th, 2012.

[Bergstra at el., 2010] Bergstra, J; Breuleux, O; Bastien, F; Lamblin, P; Pascanu, R; Desjardins, G; Turian, D; Warde-Farley, D and Bengio, Y. *Theano: A CPU and GPU Math Expression Compiler.* Proceedings of The Python for Scientific Computing Conference. (SciPy) 2010. June 30-July 3, Austin, Tx.

[Carreira-Perpiñán and Hinton] Carreira-Perpiñán, Miguel Á; Hinton, Geoffrey E. *On Contrastive Divergence Learning.* www.gatsby.ucl.ac.uk/aistats/fullpapers/217.pdf

[Erhan et al., 2010] Erhan, Dumitru; Bengio, Yoshua; Courville, Aaron; Manzagol, Pierre-Antoine; Vincent, Pascal; Bengio, Samy. *Why Does Unsupervised Pre-training Help Deep Learning?* Journal of Machine Learning Research 11 (2010) 625-660

[Fischer and Igel, 2014] Fischer, Asja; Igel, Christian. *Forecast Chaotic Time Series Data by DBNs.* Training Restricted Boltzmann Machines: An Introduction. Pattern Recognition 47: 25-39, 2014.

[Glorot and Bengio] Glorot, Xavier; Bengio, Yoshua. *Understanding the Difficulty of Training Deep Feedforward Neural Networks.* 13th International Conference on Artificial Intelligence and Statistics, Italy. Volume 9 of JMLR: W&CP9.

[Hastie and Friedman, 2009] Hastie, Trevor; Tibshirani, Robert; Friedman, Jerome. *The Elements of Statistical Learning, Data Mining, Inference and Prediction.* Springer Series in Statistics. 2009.

[Haykin, 2009] Haykin, Simon. *Neural Networks and Learning Machines.* Pearson International Edition, 3rd edition, 2009.

[Hinton, 2002] Geoffrey, Hinton. *Training Products of Experts by Minimising Contrastive Divergence.* Neural Computation 14, 1771-1800 (2002) ¸2002 Massachusetts Institute of Technology.

[Hinton, 2006] Geoffrey, Hinton. *To Recognise Shapes, First Learn to Generate Images.* UTML TR 2006-004, Department of Computer Science, University of Toronto. October 26, 2006.

[Hinton et al., 2006] Hinton, Geoffrey E.; Osindero, Simon; The Yee-Whye. *A Fast Learning Algorithm for Deep Belief Nets.* Neural Computation, 18(7): 1527-1554. 2006.

[Hinton, 2007] Hinton, Geoffrey E. *Learning Multiple Layers of Representation.* Trends in Cognitive Sciences Vol.11 N0. 10. www.sciencedirects.com

[Hinton, 2010] Geoffrey, Hinton. *A Practical Guide to Training Restricted Boltzmann Machines.* UTML Technical Report 2010-003, Department of Computer Science, University of Toronto. August 2, 2010.

[Hornik et al., 1989] Hornik, Kur; Stinchcombe, Maxwell; White, Halber. *Multilayer Feedforward Networks Are Universal Approximators.* Neural Networks, vol. 2, pp. 359-366, 1989.

[Hult, Lindskog at al., 2012] Hult, Henrik; Lindskog, Filip; Hammarlind, Ola; Rehn, Carl Johan. *Risk and Portfolio Analysis. Principles and Methods.* Springer Series in Operations Research and Financial Engineering. ISBN 978-1-4614-4102-1. Springer 2012.

[Jaggedish and Titman, 1993] Jaggedish, Narasimhan; Titman, Sheridan. *Returns to Buying Winners and Selling Losers: Implications for Stock Market Efficiency* Journal of Finance, Vol. 48, No. 1 (Mar., 1993) pp. 65-91. Blackwell Publishing for the American Finance Association. http://www.jstor.org/stable/2328882?origin=JSTOR-pdf

[Kuo et al., 2014] Kuo, R J; Chen, C H; Hwang, Y C. *An Intelligent Stock Trading Decision Support System Through Integration of Generic Algorithm Based Fuzzy Neural Network and Artificial Neural Network.* fussy Sets and Systems 118 (2001) 21-45.0165-0114/01/$-see front matter ¸2001 Elsevier Science B.V. ELSEVIER. PIL: S0165-0114(98)00399-6.

[Kuremoto et al., 2014] Kuremoto, Takashi; Kimura, Shinsuke; Kobayashi, Kunikazu; Obayashi, Masanao. *Time Series Forecasting Using a Deep Belief Network with Restricted Boltzmann Machines.* Neurocomputing 137 (2014) 47-56, ELSEVIER. Available online 22 January 2014.

[Kuremoto et al., 2014] Kuremoto, Takashi; Obayashi, Masanao; Kobayashi, Kunikazu; Hirata, Takaomi; Mabu, Shingo. *Forecast Chaotic Time Series Data by DBNs.* The 2014 7th International Congress on Image and Signal Processing, Page(s): 1130 - 1135 DOI: 10.1109/CISP.2014.7003950, Date of Conference: 14-16 Oct. 2014.

[Larochelle et al., 2009] Larochelle, Hugo; Bengio, Yoshua; Louradour, Jérôme; Lamblin, Pascal. *Exploring Strategies for Training Deep Neural Networks.* Journal of Machine Learning Research 1 (2009) 1-40.

[LISA Lab, 2014] LISA Lab, University of Montreal. *Theano: Theano Documentation.* Release 0.6, November 12, 2014.

[LISA Lab, 2014] *Deep Learning Tutorial.* LISA Lab, University of Montreal. 2014.

[Mitchell, 1997] Mitchell, Tom M. *Machine Learning.* McGraw-Hill International Series, 1997.

[Nair and Hinton, 2010] *Rectified Linear Units Improve Restricted Boltzmann Machines.* Proceedings of the 27*th* International Conference on Machine Learning, Haifa, Israel, 2010. ¸2010 but the author(s)/owner(s).

[Rojas, 1996] Rojas, R. *Neural Networks.* Springer-Verlag, Berlin 1996.

[Rosenblatt, 1957] Rosenblatt, Frank. *The Perceptron, A Perceiving and Recognising Automaton.* Project Para Report no. 85-460-1, Cornell Aeronautical Laboratory (CAL), Jan. 1957.

[Rudin, 1976] Rudin, Walter. *Principles of Mathematical Analysis.* McGraw-Hill International Series, 1976.

[Salakhutdinov, 2009] Salakhutdinov, Ruslan. *PhD Thesis: Learning Deep Generative Models.* Department of Computer Science, University of Toronto. 2009.

[Salakhutdinov and Hinton, 2012] Salakhutdinov, Ruslan; Hinton, Geoffrey E. *An Efficient Learning Procedure for Deep Boltzmann Machines.* Neural Computation, 24, 1967-2006 (2012). 2012.

[Takeuchi et al., 2013] Takeuchi, Lawrence; Lee, Yu-Ying. *Applying Deep Learning to Enhance Momentum Trading Strategies in Stocks.* http://cs229.stanford.edu/proj2013/TakeuchiLee- ApplyingDeepLearningToEnhanceMomentum TradingStrategiesInStocks.pdf December 12, 2013.

[Zhu and Yin, 2014] Zhu, Chengzhang; Yin, Jianping; Li, Qian. *A stock Decision Support System Based on DBNs.* Journal of Computational Information Systems 10: 2 (2014) 883-893. Available at http://www-jofcis.com. Binary Information Press, DOI: 10.12733/jcis9653. January 15, 2014.

TRITA -MAT-E 2015:40
ISRN -KTH/MAT/E--15/40-SE