OULUN YLIOPISTO
UNIVERSITY of OULU

OULU BUSINESS SCHOOL

Daniel Stafford

MACHINE LEARNING IN OPTION PRICING

Master's Thesis

Department of Finance

Nov 2018

UNIVERSITY OF OULU | ABSTRACT OF THE MASTER'S THESIS
Oulu Business School

Unit Department of Finance			
Author Stafford, Daniel		Supervisor Conlin A., Postdoctoral researcher	
Title Machine learning in Option Pricing			
Subject Finance	Type of the degree Master's Thesis	Time of publication November 2018	Number of pages 94

Abstract

This paper gives an overview of the research that has been conducted regarding neural networks in option pricing. The paper also analyzes whether a deep neural network model has advantages over the Black-Scholes option pricing model in terms of pricing and hedging European-style call options on the S&P 500 stock index with data ranging from 1996 to 2014.

While the previous literature has focused on shallow MLP-styled neural networks, this paper applies a deeper network structure of convolutional neural networks to the problem of pricing and hedging options. Convolutional neural networks are previously known for their success in image classification.

The first chapters of this paper focus on both introducing neural networks for a reader, who is not familiar with the topic a priori, as well as giving an overview of the relevant previous literature regarding neural networks in option pricing. The latter chapters present the empirical methodology and the empirical results.

The empirical study of this thesis focuses on comparing an option pricing model learned by a convolutional neural network to the Black-Scholes option pricing model. The comparison of the two pricing models is two-fold: the first part of the comparison focuses on pricing performance. Both models will be tested under a test set of data, computing error measures between the price predictions of each model against the true price of an option contract. The second part of the comparison focuses on hedging performance: both models will be used in a dynamic delta-hedging strategy to hedge an option position using the data that is available in the test set. The models are compared to each other using discounted mean absolute tracking error as a measure-of-fit. Bootstrapped confidence intervals are provided for all relevant performance measures.

The empirical results are in line with the previous literature in terms of pricing performance and show that a convolutional neural network is superior to the Black-Scholes option pricing model in all error measures. The pricing results also show that a convolutional neural network is better than neural networks in previous studies with superior performance in pricing accuracy also when the data is partitioned by moneyness and maturity. The empirical results are not in line with the previous literature in terms of hedging results and show that a convolutional neural network is inferior to the Black-Scholes option pricing model in terms of discounted mean absolute tracking error.

The main findings show that combining a neural network with a traditional parametric pricing formula gives the best possible outcome in terms of pricing and hedging options.

Keywords
Options, hedging, neural networks, bootstrapping

Additional information

CONTENTS

FIGURES

TABLES

1 INTRODUCTION

The use of deep learning in finance in general has begun its second era since the mid 2000s. Deep learning was previously used in finance in the form of neural network applications by academic researchers in the late 1980s and 1990s but during that time, the success of these studies remained fairly poor mainly due to the lack of computing power needed to train truly successful neural networks. During those days, many of the more recent academic findings related to deep learning were still to be found and the machine learning methods that were used then had to be either hard-coded to hardware itself or shortly afterwards programmed completely on some of the first object-oriented programming languages, which were perhaps not that familiar for the academic researchers who had received their PhD's in economics rather than in computer science and had perhaps never been that familiar to heavy programming.

In the more recent days, machine learning and more specifically deep learning has become commoditized with open-source tools such as Tensorflow, Caffe and Theano that allow anyone, anywhere to implement their deep learning solution to any given problem that can be somehow specified in terms of deep learning, as Culkin and Das (2017) describe it. The deep learning tools of today are also supported by several programming languages such as Python and R that are accessible to the common public and are easy to learn even with academic aspirations elsewhere than in computing and computer science. The major advancements in computing power itself enables the implementation of deep(er) learning solutions over enormous datasets that were previously unthinkable.

With recent advancements of deep learning in the fields of medicine, computer vision and image recognition, speech and text recognition and in other technological fields such as in cloud-computing, one can only imagine what possibilities there must be for deep learning in the field of finance, where data is accessible at best in real time and in gigantic quantities.

This thesis aims to explore how deep learning of today can be exploited in the finance space with an empirical study in one of the most studied fields in finance, derivatives pricing. Since Black and Scholes published their famous formula for pricing options in 1973, also accompanied by Merton (1973) (hence the reference BSM), by modeling the price of an underlying asset S_t as a continuous time diffusion

$$dS_t = \mu S_t d_t + \sigma S_t dW_t \tag{1}$$

with a constant drift μ, constant volatility σ and W_t denoting a Wiener process (see e.g. Shreve 1997, pp. 142-152) and by solving the famous fundamental partial differential equation (PDE)

$$0 = \frac{dC}{dS} r S_t + \frac{dC}{dt} + \frac{1}{2}\frac{d^2C}{dS^2} S_t^2 \sigma^2 - rC_t \tag{2}$$

where r denotes the constant risk-free rate and C_t denotes the price of a call option, a new era of derivatives with ever more complex payoff functions had begun. Despite the questionable assumptions that were introduced to the model in the form of parameters such as constant volatility and interest rates, the model has proven to be valuable and is used with its numerous variations created in the years after by practitioners and academia to price options and more importantly, to back out implied volatilities from the market.

Hutchinson et al. (1994) were among the first in academia to question the use of parametric models to price options and proposed a non-parametric approach instead, using neural networks. They made remarkable findings, presenting that even a fairly shallow neural network such as the one hidden layer feedforward network they used at the time performed extremely well to learn the market's pricing function and predict option prices more accurately than the BSM. In their paper, Hutchinson et al. also tested the hedging performance of both models with the non-parametric neural network giving better results than the BSM for delta-hedging a call option position, at least to some degree in the discrete time.

Pursuing the non-parametric approach taken by Hutchinson et al. (1994) and other academic researchers such as Anders et al. (1996), Garcia and Gencay (2000), Yao et al. (2000), Amilon (2003), Gencay and Salih (2003) and Bennell and Sutcliffe

(2004), this thesis aims to explore the capabilities of deep learning and more specifically, deep neural networks in derivatives pricing with an empirical study to compare a deep neural network pricing model to the BSM with data gathered from the real markets. Following Hutchinson et al. (1994), the comparison is two-fold: a deep neural network model will be trained to learn the market's option pricing function with a train data set, after which the overall pricing performance will be compared on a test data set to that of the BSM using error estimates such as the Mean Squared Error (MSE) and Root Mean Squared Error (RMSE). The second comparison between the models will be made with a hedging experiment: both models will be used to delta-hedge an option position with the market data available in the test set, in discrete time hedging. The comparison will be made using the discounted mean absolute tracking error of each model for the hedging period.

Diverging from the previous non-parametric approaches to price derivatives, a different, deeper neural network structure will be introduced in this thesis, namely a convolutional neural network (CNN). This approach is taken based on the successes that have been witnessed in image recognition and machine vision in the previous years (e.g. Krizhevsky et al. 2012, Zeiler & Fergus 2013, Simonyan & Zisserman 2014). Apart from image recognition, CNNs have been previously used for example in speech recognition and they are well known for their abilities to recognize patterns from complex data sets.

CNNs seem like a natural choice of network structure to derivatives pricing as the problem can be approached as a pattern recognition based optimization problem of network parameters such that the input data observed in the market best fits the option prices also observed in the market. The key difference compared to the simple multilayer perceptron used in the previous papers on neural networks and option pricing is that the amount of fitted parameters between layers is extremely large and can be described in nearly ten figures, compared to the number of parameters in the previous papers described in thousands. Taking this approach, the desire is that more subtle non-linearities can be captured, thus improving the pricing accuracy and hedging performance even further.

The CNN used in this thesis was inspired by AlexNet, a CNN introduced by Krizhevsky et al. (2012), who won the 2012 ImageNet Large-Scale Visual Recognition Challenge (ILSVRC) with AlexNet. The network implemented in this thesis is not an exact replica, as it has less input feature dimensionality and only one channel instead of two, also with some additional changes in the number of neurons in the hidden layers and some differences in the output layer to better fit the learning problem of derivatives pricing.

The main research question of this thesis is formulated as:

- Does a market-data trained convolutional neural network model price call options significantly better than the BSM model?

Additional research questions are imposed:

- Does the delta-hedging performance of a convolutional neural network model significantly differ from that of the BSM model?
- Are there differences in the pricing accuracy for different types of options, i.e. for at-the-money (ATM), in-the-money (ITM) and out-of-the-money (OTM) options or for different types of time-to-maturities?
- Can the convolutional neural network model capture time-varying volatility properties compared to the BSM, which assumes a constant flat volatility?

The rest of the thesis is structured as follows: chapter two gives an overview to neural networks and convolutional neural networks in general. This is designed such that a reader with no previous experience can get familiarized with the topic. Chapter three discusses neural networks more specifically in the derivatives pricing space with an overview to the relevant previous literature. Chapter four presents the empirical methodology, while chapter five presents the empirical findings. Chapter six concludes.

2 REVIEW OF NEURAL NETWORKS

This chapter defines and gives an overview of neural networks and describes some of the relevant overviewing literature regarding neural networks. The aim is to give the reader a basic understanding of how neural networks work and what does the so-called learning of a neural network mean. First, the basic structures of a neural network will be introduced. Second, an overview will be given of the multilayer perceptron, which is the type of neural network used in previous studies related to option pricing. Third, an overview of convolutional neural networks will be given, which is the type of neural network used in this thesis for option pricing purposes.

2.1 Artificial Neural Networks

Artificial Neural Networks (ANNs) consist of one or more layers of simple connected processors called neurons (or nodes), designed to mimic the architecture of how the human (or any mammal) brain works (Kartalopoulos 1995, Schmidhuber 2015). According to Kartalopoulos (1995), these processors that are further referred to as neurons get activated through a sensor perceiving the environment or through weighted connections from a previously activated neuron. The neurons that are activated through a sensor perceiving the environment are called input neurons and they are usually the initial neurons that receive data from outside the network.

Formally, the topology of a neural network can be described at any given moment as a finite set of neurons $N = \{u_1, u_2, \dots, u_n\}$ and a finite set of $F \subseteq N \, x \, N$ directed connections, called activation (transfer) functions, between the neurons. These connections are associated with a "firing" of the neuron, i.e. whether the neuron activates or not (Kartalopoulos 1995, Schmidhuber 2015).

2.1.1 The Perceptron

The first neural network architectures were published in academia by McCulloch and Pitts (1943) and by Hebb (1949). McCulloch and Pitts (1943) introduced several mathematical theorems on calculus of a neuron firing in a biological neural system. Hebb (1949) proposed a theory on how biological neurons in the brain adapt during a learning process. The theory aimed to explain the problem related to Hebbian learning, i.e. how simultaneous neural activations lead to increases in synaptic strength (Hebb 1949), which worked as a biological base for unsupervised learning.

The neural network architecture published by McCulloch and Pitts (1943) was specifically targeted as a computational model of the neural network in a mammal brain (Kartalopoulos 1995, Aggarwal 2014). In their study, McCulloch and Pitts introduced the first known artificial neuron, called the *Linear Threshold Unit* (LTU), which employs a threshold equivalent to the *heaviside step function.*

The LTU can be thought of as a function that takes in an arbitrary numerical vector input, calculates a weighted sum of the input and applies the heaviside step function (1) to this sum with a binary output of either 0 or 1. The step function is only applied if the weighted sum exceeds some threshold value b, referred to as the *threshold bias.* When the threshold is not exceeded, the LTU remains unchanged. In neural network terms, the heaviside step function is known as a transfer function or activation function, which associates the unit with activation. These functions are further referred to as activation functions. Given the inputs of the weighted sum minus the bias, the LTU then outputs the result of its activation function (Kartalopoulos 1995). Figure 1 shows the visual representation of the step function defined in Equation 3.

$$g(x) = \begin{cases} 1, if\ x \geq 0, \\ 0, if\ x < 0 \end{cases}. \tag{3}$$

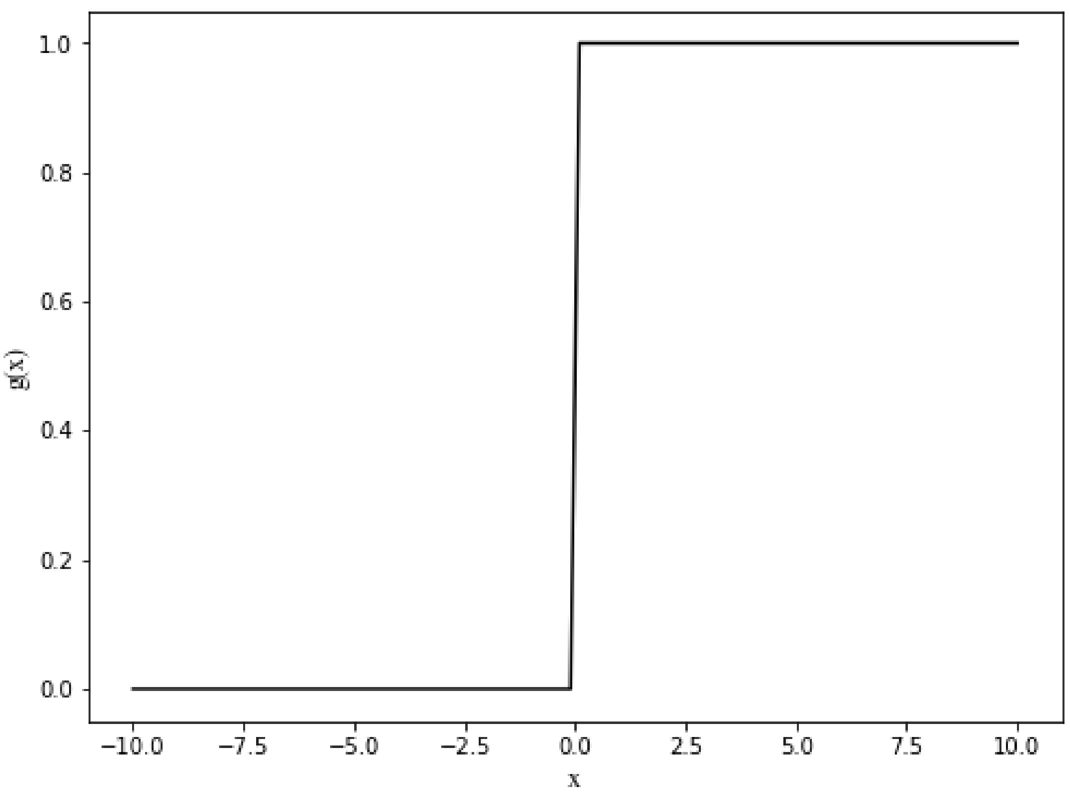

Figure 1. Heaviside step function

In addition to the heaviside, a more flexible (threshold) activation was also introduced by McCulloch and Pitts in the same paper from 1943, namely the *sign* step function (4), with similar boolean output properties. The sign function properties are shown visually in Figure 2.

$$g(x) = \begin{cases} -1, if\ x < 0 \\ 0, if\ x = 0 \\ 1, if\ x > 0 \end{cases} \tag{4}$$

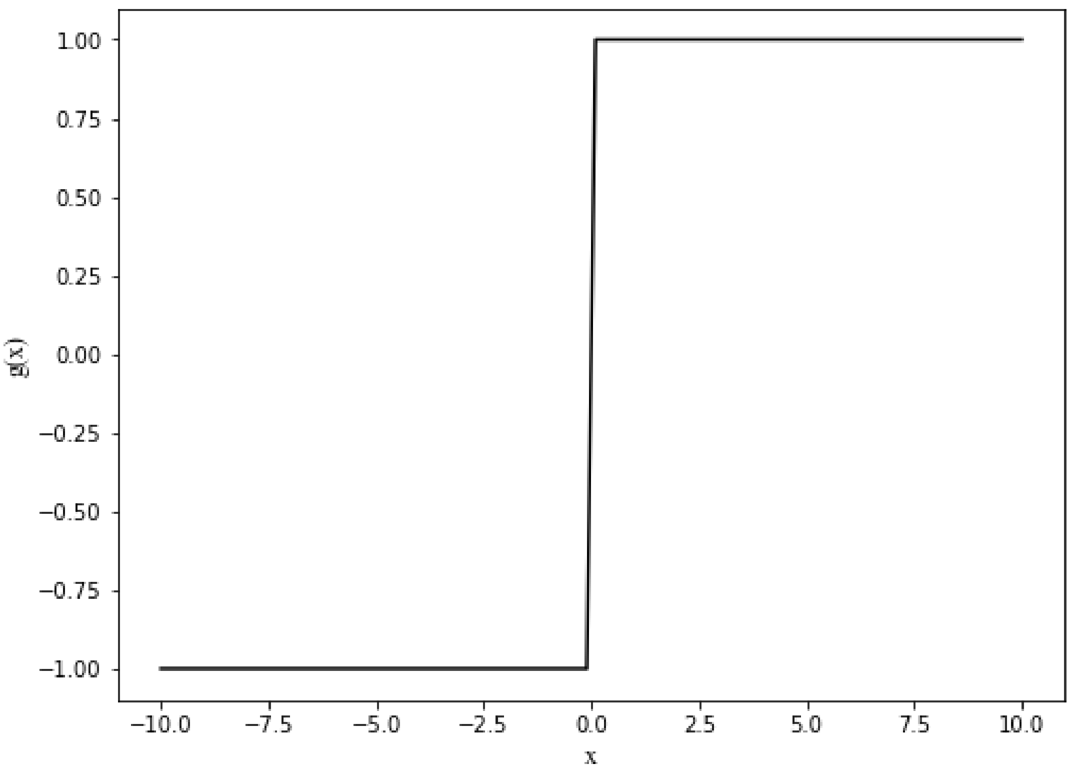

Figure 2. Sign step function

Although known and credited for its biological contributions, McCulloch and Pitts' LTU did not yet have the capabilities or the architecture of a modern-like artificial neural network that could be employed in machines with adaptive capabilities (Schmidhuber 2015). The simplest modern-type ANN model that could perform such tasks was presented first in 1958 by Rosenblatt and is called the *perceptron*, which comprises of McCulloch and Pitts' LTUs.

Rosenblatt's perceptron has a true network-like structure, composing of the input vector, a layer of parallel LTUs and an output vector with boolean output values, dependent on the sort of step function used as activation (Rosenblatt 1958). The perceptron was originally designed as a machine instead of a program, with capabilities to perform linear classification of input data (Bishop 2006). In the early perceptron of Rosenblatt, each of the LTU's are connected to all of the elements in the input vector. Figure 3 demonstrates the LTU, while Figure 4 demonstrates the perceptron. In both figures, the threshold bias is initiated with the numerical value 1.

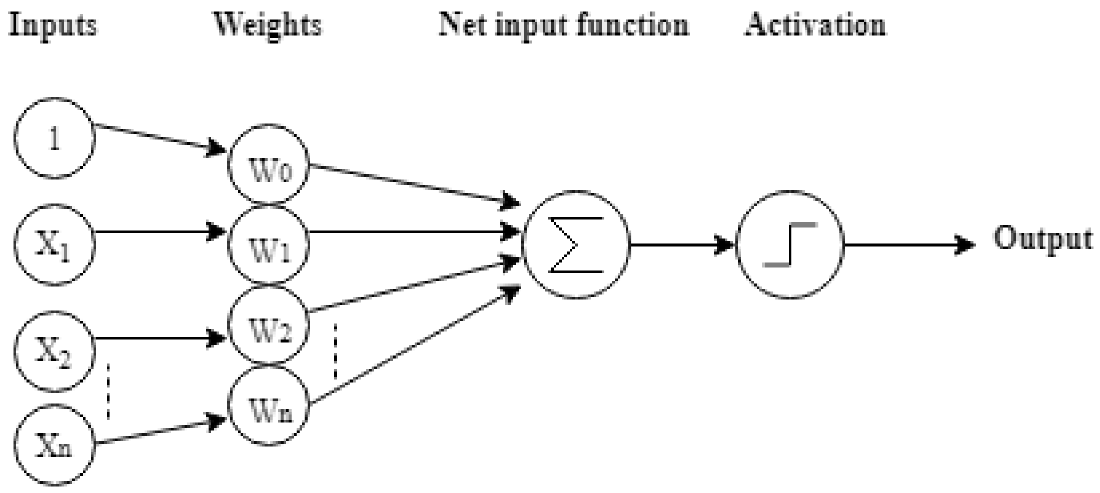

Figure 3. Linear Threshold Unit (LTU)

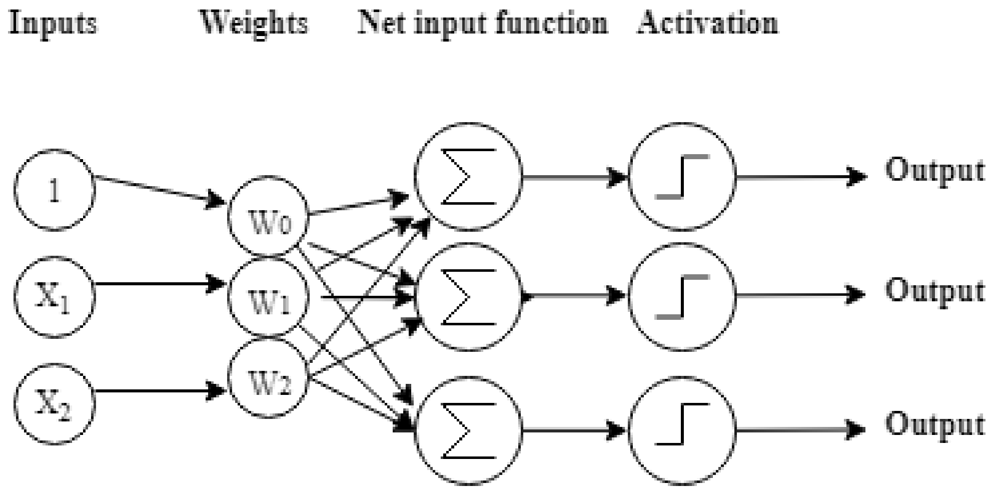

Figure 4. The Perceptron (Rosenblatt 1958)

Rosenblatt's architecture of the neural network launched what could later be called as the first golden era of machine learning and artificial intelligence (Schmidhuber 2015). Although accredited as the first modern ANN architecture and capable of performing tasks such as linear classification, implemented first in the IBM 704 software and then written on a custom hardware called Mark 1 Perceptron, the perceptron still imposed restrictions such as the restriction to binary output values that researchers in the upcoming decades proceeded to solve. The work of those decades lead to the development of a more advanced model, called the *Multilayer Perceptron* (MLP), known today as the modern neural network (Kartalopoulos 1995, LeCun et al. 2015, Schmidhuber 2015). The MLP will be described next in more detail.

2.1.2 The Multilayer Perceptron

The multilayer perceptron, also known as a *vanilla* neural network or the *feedforward network* is a class of artificial neural networks that consist of at least three layers of neurons. These layers are the input layer, which is generally similar to the input layer of the perceptron described above, a hidden layer, which is a layer of neurons between the input and output layers and the output layer, which again is generally similar to that of the perceptron (Kartalopoulos 1995, Schmidhuber 2015). There can be arbitrarily many hidden layers of arbitrary number of neurons between the input and output layers in the MLP (LeCun et al. 2015).

The multilayer perceptron is based on the works by Ivakhnenko (1970) and Fukushima (1980), who developed methods to stack the perceptron into multiple layers, thereby forming the first neural networks with more than two levels. Ivakhnenko proposed a network training procedure called the *Group Method of Data Handling* (GMDH), which is an inductive self-organizing family of algorithms that for a given training set incrementally grows the layers of perceptron involved in a network, training them by regression (Ivakhnenko 1970, Schmidhuber 2015). According to Schmidhuber (2015), Ivakhnenko’s GMDH was the first truly deep neural network structure in the sense that it had the capability of structuring networks with more than two hidden layers embedded in them.

Fukushima (1980) proposed a neural network called the *Neocognitron*, which is a multilayered neural network that used pre-wiring of network weights and was capable of performing tasks such as handwritten character recognition and pattern recognition (LeCun et al. 2015). According to LeCun et al. (2015) and Schmidhuber (2015), Fukushima’s Neocognitron was also the first paper to inspire the development of modern *convolutional neural networks* (CNNs) that are used today in tasks such as image, speech and text recognition and classification.

The basics of the MLP are similar to that of the perceptron: the input layer consists of a numerical input vector and some bias term b, from which a weighted sum is calculated minus the bias term such that the input to the first hidden layer of the network is the weighted sum of the inputs plus the threshold bias (LeCun et al. 2015, Schmidhuber 2015). Similar to the perceptron, in each layer of the network, there is again an activation function, which gets activated if the weighted sum of the inputs from the previous layer exceeds the threshold bias. The output of the activation function in each neuron is passed on to the next layer of neurons, either another hidden layer or the output layer, where a weighted sum is again calculated from the outputs of the subsequent layer and passed on to the activation function if this weighted sum exceeds another threshold bias of that layer. This procedure continues all the way until the output layer, where the final output of the network is given. This output may be a scalar value or an output vector, depending on the structure of the network (Kartalopoulos 1995, LeCun et al. 2015, Schmidhuber 2015). Figure 5 demonstrates an MLP with two hidden layers.

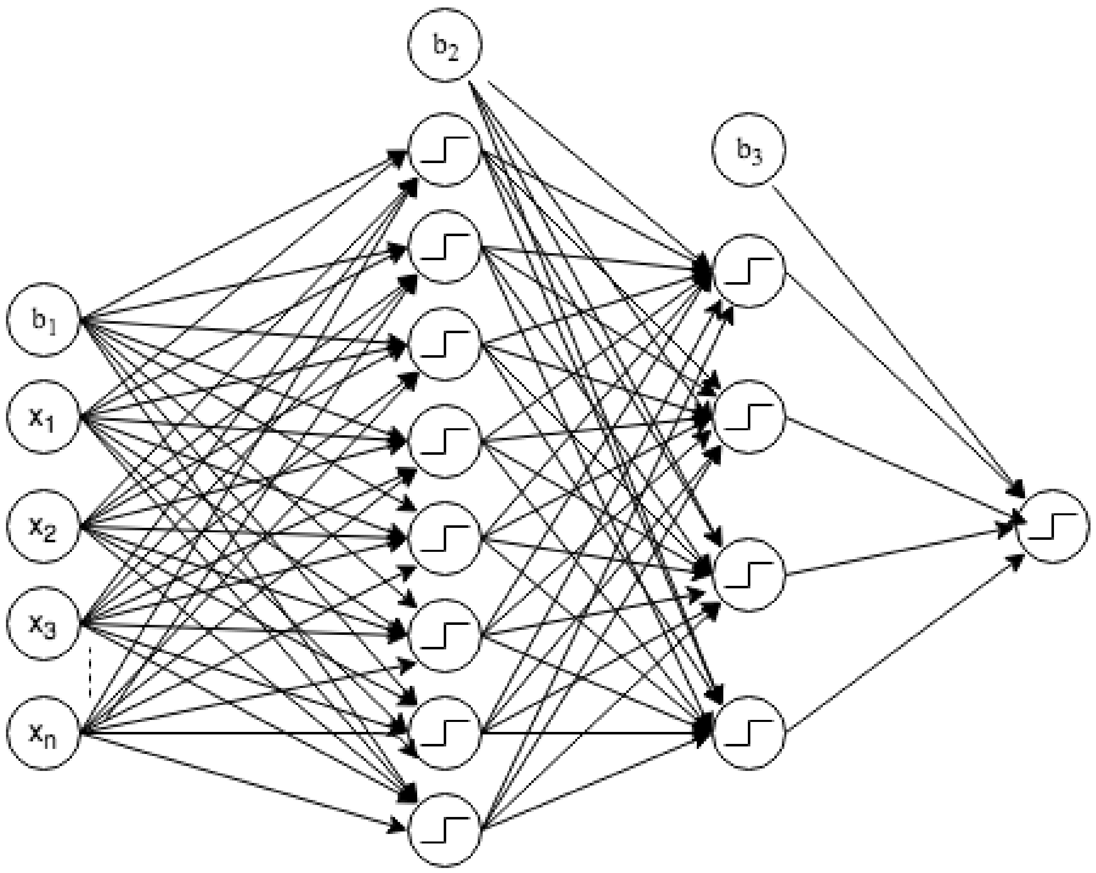

Figure 5. Multilayer Perceptron with two hidden layers

In contrast to the perceptron, outputs from the multilayer perceptron can also be other than binary values, e.g. the output can be a scalar value or a vector of real numbers. This is because the neurons in the MLP are different from the LTU neurons in Rosenblatt's (1958) perceptron: it is crucial in the MLP that the activation functions used can also produce non-linear activations instead of the simple linear step function-wise activations present in the perceptron (LeCun et al. 2015, Schmidhuber 2015). These non-linear activations include for example the sigmoid function, the hyperbolic tangent function, the exponential function and the rectified linear unit (ReLU) function, which has since first introduced by Hahnloser et al. (2000) gained also other versions such as the Leaky ReLU and the Noisy ReLU. In recent years, the rectifier has gained a lot of popularity as one of the main activations used in todays neural network applications amongst practitioners and academia (LeCun et al. 2015).

The idea of non-linear activations in the hidden layers of an MLP is that they can distort the input space away from a linear world and enable the MLP to output real-numbered outputs in each layer of neurons, enabling far more complex function approximations of the network, compared to the binary activations used in the perceptron (Kartalopoulos 1995, LeCun et al. 2015). The invention of non-linear activations made it possible to capture non-linear relationships between the input set and the correct output, enabling the MLP to face regression-like tasks and non-linear classification problems such as image recognition (LeCun et al. 2015). Using only the step-wise functions from the perceptron, similar results would not be possible, as it comes from mathematics that a linear combination of any amount of such functions would just be another linear function (Lay & Strang 2006: p. 125-126).

The sigmoid function is defined in Equation 5, where e is Euler's number (e refers to Euler's number also in later sections, unless stated otherwise). Figure 6 shows the sigmoid visualized in a two-dimensional space.

$$g(x) = \frac{1}{1+e^{-x}} \tag{5}$$

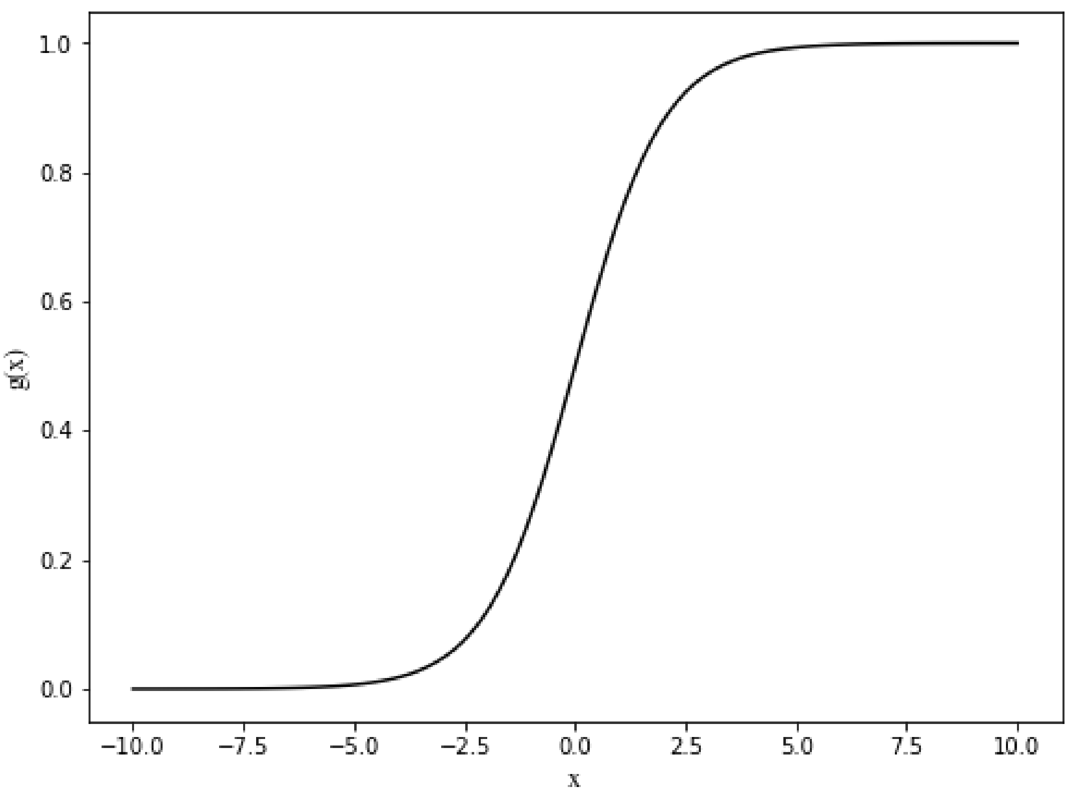

Figure 6. Sigmoid function

The benefit of the sigmoid function in an MLP compared to a step function or a linear function is that it is non-linear in nature, allowing a neural network to learn non-linear relations from a dataset. The sigmoid suffers from vanishing gradients in areas where, $g(x) \to 1$ or $g(x) \to 0$. This tends to slow down the gradient-descent based learning process of a neural network (Hochreiter 1998). This learning process will be described in full in the following sections of this chapter.

The hyperbolic tangent (also referred to as the tanh function) is defined in Equation 6 and visualized in a two-dimensional space in Figure 7. The hyperbolic tangent, as shown in Equation 6 is a scaled version of the sigmoid.

$$g(x) = \frac{2}{1+e^{-2x}} - 1 \ = 2\, sigmoid(x) - 1 \qquad (6)$$

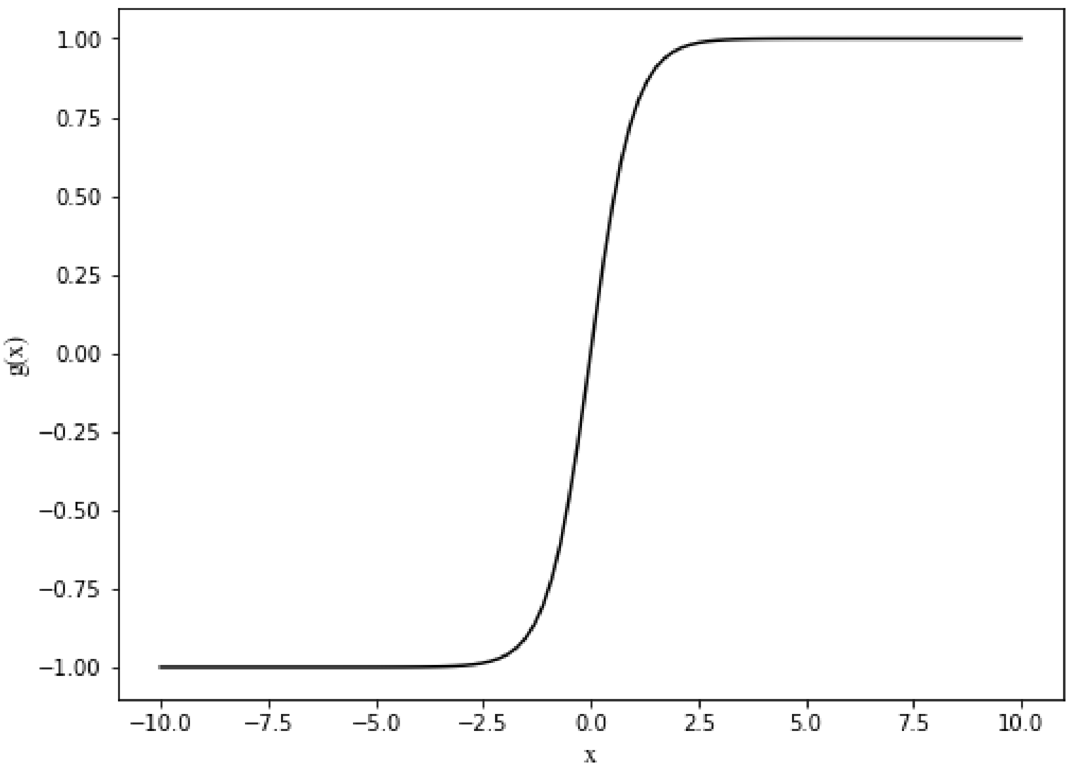

Figure 7. Hyperbolic tangent function

The hyperbolic tangent is very similar to the sigmoid in its nature but unlike the sigmoid, it is bound to a range of $g(x) \in [-1,1]$. Similarly to the sigmoid, the hyperbolic tangent is non-linear in nature. Compared to the sigmoid, the gradient of tanh is stronger with steeper derivatives, but similarly to the sigmoid, tanh also has a problem of vanishing gradients present in areas where $g(x) \to 1$ or $g(x) \to -1$ (Weisstein 2002).

The exponential function is defined in Equation 7 and visualized in a two-dimensional space in Figure 8.

$$g(x) = e^{x} \tag{7}$$

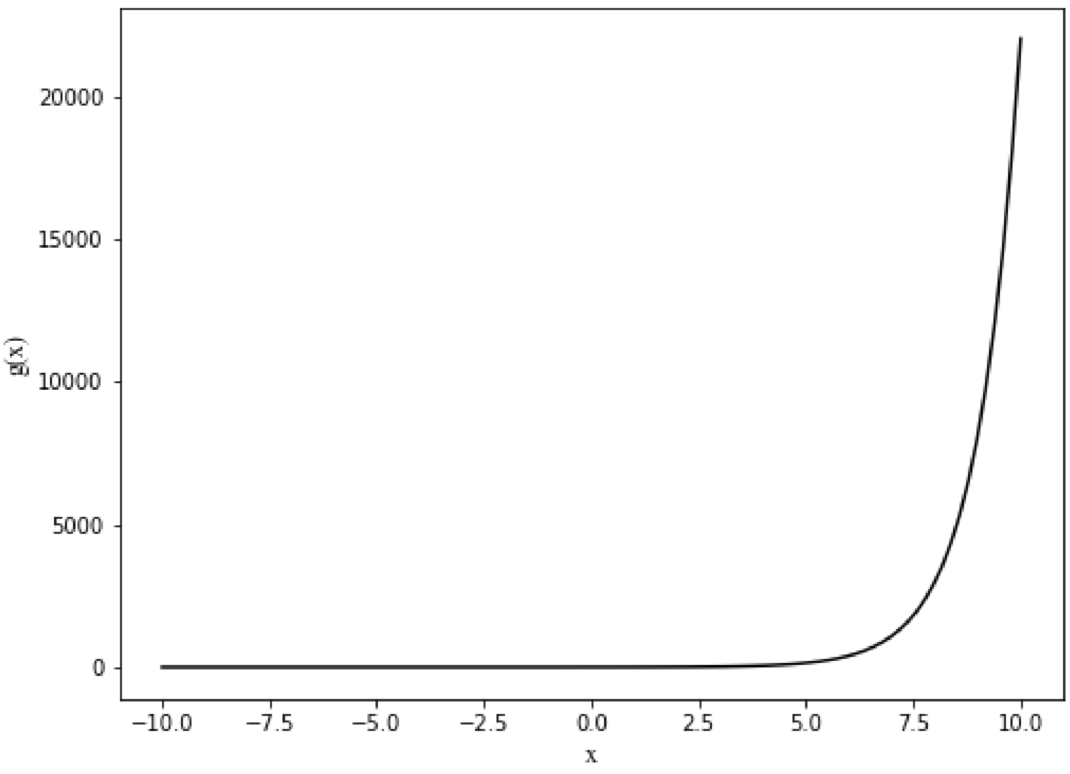

Figure 8. Exponential function

The exponential function, similarly to the other non-linear functions described above has the property that it is non-linear in nature. Compared to the sigmoid or the hyperbolic tangent, the exponential function is especially viable in tasks where the output from the neuron is desired to be restricted to positive values of the real number space. It also has smooth gradients for most of positive input. The down-side of the exponential function is that, as the function does not have an upper bound and $g(x) \in [0, \infty]$, the neurons that are activated by an exponential function can "blow up".

The basic ReLU formula is defined in Equation 8 and visualized in a two-dimensional space in Figure 9. There also exists a smooth approximation of the ReLU known as the Softplus function, formulated in Equation 9.

$$g(x) = x^{+} = \max(0, x) \tag{8}$$

$$g(x) = \log(1 + e^{x}) \tag{9}$$

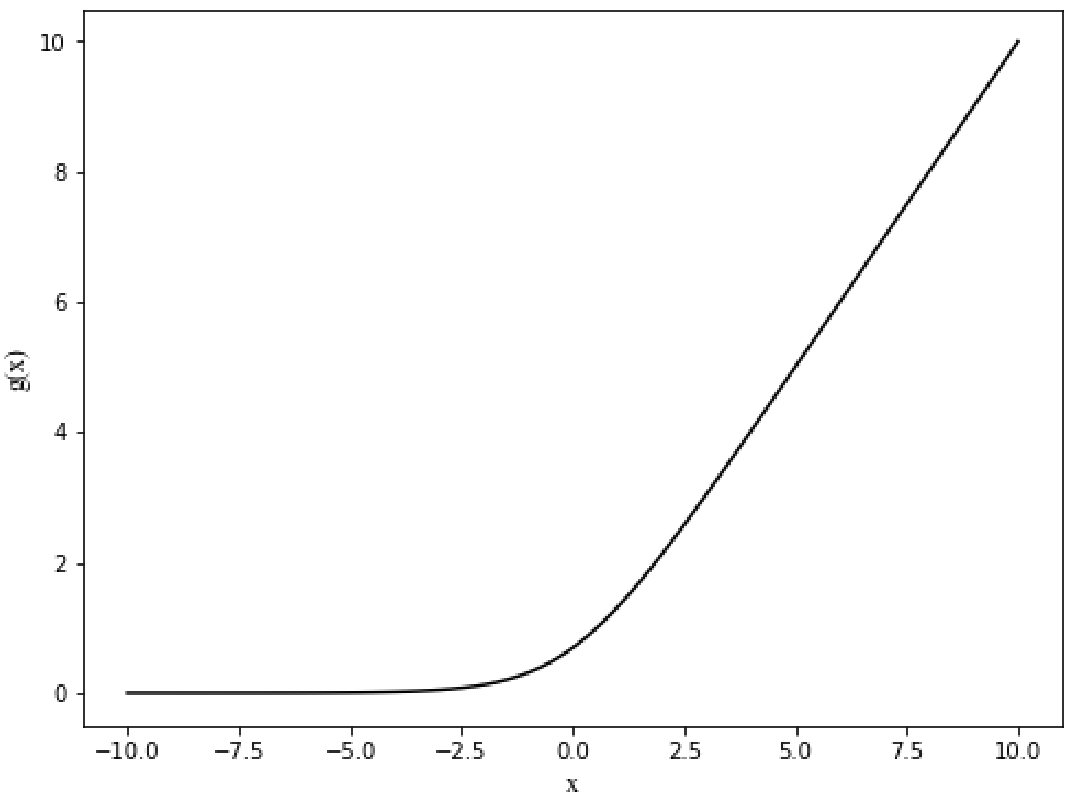

Figure 9. Rectified Linear Unit function

The ReLU represents the "positive part" of the real-valued non-linearity, as Jarret et al. (2009) refer to it, which ranges such that $g(x) \in [0, \infty]$. While the benefit of the ReLU is that it tends to have smooth gradients for positive input, the down-side is that ranging to the positive infinity, a neuron activated by the ReLU can "blow up" (Nair & Hinton 2010). The ReLU function is also associated with a *dead-neuron problem*, implying that with negative values of input, the gradient of the function can go to zero, leaving parts of the network passive in the gradient-descent training process (Li 2017). Academics have proposed alternatives such as the Leaky ReLU and the Noisy ReLU to fix the problem of dead neurons (See e.g. Nair & Hinton 2010, Maas et al. 2014).

The reason that rectifiers have become such popular to use in the recent years, according to Nair and Hinton (2010), is due to the benefit that a rectifier can be constructed as many copies of Restricted Boltzmann Machines (RBMs) (See e.g. Smolensky 1986, Hinton & Salakhutnidov 2006) that can be stacked together, requiring the probability of a neuron firing only to be calculated once. This property enables faster and more stable weight-updating of the network weights, eventually decreasing training time (Jarret et al. 2009, Nair & Hinton 2010, LeCun et al. 2015).

The cruciality of having activation functions with non-linear properties in the MLP is embedded to how the network "learns" to solve a task it is given (LeCun et al. 2015). As the goal of an MLP (or any other supervised regression-like learning task in general) is to find a function that best maps the set of inputs (the input vector and the bias term) to the correct output values, there must exist an error function (also referred in neural network language to as the cost function or loss function) that is used to represent the mapping error of the function that the neural network uses to do the mapping.

The learning task of the MLP can then be set to minimize the error function's outputs, which can be achieved by altering the weights that are used between the neural connections with each layer so that the error function's outputs are minimized. This optimization problem includes the calculation of the cost function's gradient using an algorithm called backpropagation, often originated to Rumelhart et al. (1986), based on a general technique known in mathematics as automatic differentiation. The backpropagation algorithm and the process of learning will be described in the following chapter.

2.2 The backpropagation algorithm

The backpropagation algorithm is often incorporated with the study by Rumelhart et al. (1986), who proposed a simple derivation of the algorithm, eventually an application of the chain rule and an algorithm called gradient descent, which is an algorithm formulated to find to local (global) optima of a function based on its' gradient. Alternative derivations of the backpropagation algorithm have been proposed (See e.g. Werbos 1974, LeCun 1985, LeCun 1986).

The idea of the backpropagation algorithm is to compute a gradient of the neural network's cost function by working the gradient backwards through each layer of the neural network, with respect to the output of each layer (or the input of the subsequent layer). The backpropagation is applied repeatedly to each layer of the network, starting from the output layer, where a network's output is produced and then computing the partial derivatives through each layer all the way back to the input layer, where the external input is given to the network. Once these partial derivatives have been computed, it is straightforward to compute the gradients in each layer with respect to the weights of that layer (LeCun 1988, Kartalopoulos 1995, LeCun et al. 2015). According to LeCun et al. (2015) the backpropagation has been found to work on all types of multilayer architectures, as long as the neurons in each of the layers are relatively smooth functions of their inputs and of their internal weights.

By using backpropagation to calculate the gradients of each of the layers in the network, the training of the MLP can be constructed as an iterative process where the parameters of the network (the weights) are updated during iterations. The backpropagation consists of two parts: a *forward propagation* is the first part. In the forward propagation part, the inputs are passed through the network with the current weight values of the network (usually initiated at the beginning randomly, e.g. drawing from a standard normal Gaussian) (LeCun 1988, LeCun et al. 2015). After the first forward propagation, the network outputs some real-number scalar or a vector of real-numbered values. The output from the network is compared to the true values of that output, similar to a regression-like task, and an output error (often denoted with E or ε) is calculated based on the comparison. This error is denoted as the *loss* of the network and is dependent on the formulation of the loss function.

The second part of the backpropagation algorithm is the *back propagation* part. Based on this above calculated error, the gradient of the loss is computed as a backward process starting from the output layer and working the gradients backward until the input layer. As stated above, once the gradients have been calculated with respect to each layer, it is straightforward to compute the gradients with respect to the weights of each layer (LeCun et al. 2015). The idea of calculating the gradients with respect to the weights is that the amount of loss each weight contributed to the total loss can be deduced. The network weights are then updated based on their overall contribution to the loss of the network with an objective that on each iteration, the loss becomes smaller. Figure 10 demonstrates this process in a simplified network architecture. The following subchapters will more formally define the forward propagation and the back propagation parts of the backpropagation algorithm.

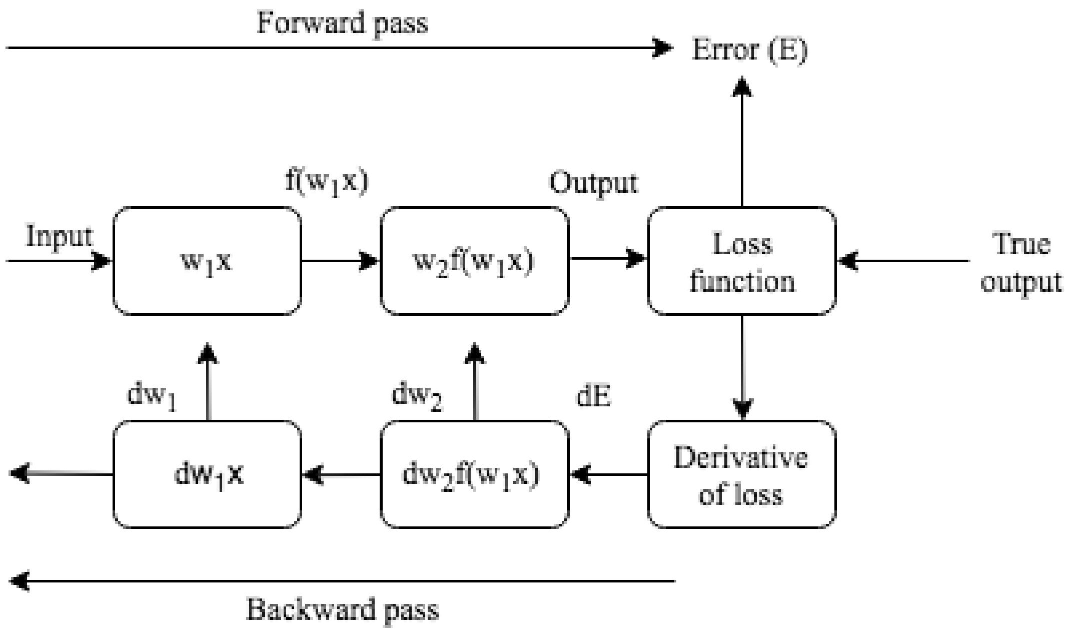

Figure 10. Backpropagation algorithm

2.2.1 The forward propagation

In this section, the first three steps shown in Figure 10 are defined more formally. The forward propagation of the backpropagation algorithm can be formulated as follows. Starting with the input vector, $x \epsilon \mathbb{R}^d$, the input bias term $b^{(1)}$, (usually initialized to some arbitrary real value such as 0 or 1) and the weights of the input layer in the weight matrix $W^{(1)} \epsilon \mathbb{R}^{m^{(1)} x\, d}$, where $m^{(1)}$ denotes the size of the first hidden layer $h^{(1)}$, the input elements of the input vector along with the bias term are passed to each neuron in the first hidden layer as a weighted sum (minus the bias term), where the input is activated by an activation function $f^{(1)}(\cdot)$. The process continues to the second layer, where the input vector is now the output from the subsequent layer $h^{(1)}$, the bias is the bias term of the second layer $b^{(2)}$, and the weight matrix is denoted as $W^{(2)} \epsilon \mathbb{R}^{m^{(2)} x\, m^{(1)}}$, where $m^{(2)}$ denotes the size of the second hidden layer $h^{(2)}$. The process can be expressed in matrix notation as

$$\begin{aligned} h^{(1)} &= f^{(1)}\big(b^{(1)} + W^{(1)}x\big) \\ h^{(2)} &= f^{(2)}\big(b^{(2)} + W^{(2)}h^{(1)}\big) \\ &\vdots \\ h^{(n)} &= f^{(n)}(b^{(n)} + W^{(n)}h^{(n-1)}) \\ \hat{y} &= \phi\big(b^{(n+1)} + w^T h^{(n)}\big) \end{aligned}$$

where n is the total number of hidden layers, $w^T \epsilon\ \mathbb{R}^{m^{(n)}}$ is the transposed vector of the output weights, ϕ is the activation function of the output layer and $\hat{y}$ is the output of the network. The network output can then be written as a convenient function

$$\hat{y} = g\big(x; W^{(1)}; W^{(2)}, \dots, W^{(n)}, w, b^{(1)}, b^{(2)}, \dots, b^{(n)}, b^{(n+1)}\big) = g(x;\theta) \quad (10)$$

The output given by Equation 10 is the output that will be compared to the true value of the output, denoted by y. Based on the comparison, an output error is calculated according to some loss function L between the true value and the predicted value of $\hat{y}$ and y. Rumelhart et al. (1986) use the *Squared Error* (SE) to compute the loss L, shown in Equation 11, but other loss functions, such as *absolute error* functions are also popular (See e.g. LeCun et al. 2015).

$$L(x, y, \theta) = (g(x;\theta) - y)^2 = (\hat{y} - y)^2 \quad (11)$$

where θ denotes all the parameters that are used to compute $\hat{y}$ and y denotes the desired true values. In this example of the backpropagation algorithm, Squared Error will be used.

2.2.2 The backward propagation

This section describes the *backward pass* part of the backpropagation algorithm, shown by the latter three steps in Figure 10. Using the Squared Error loss function, defined in Equation 11, the total loss of the network for that training set can be computed as

$$E(x, y, \theta) = \frac{1}{N}\sum_{i=1}^{N}(y_i - \hat{y}_i)^2 = \frac{1}{N}\sum_{i=1}^{N} L(x_i, y_i; \theta) \quad (12)$$

where N denotes the size of the training set, y_i is the i^{th} true value and $\hat{y}_i$ is the i^{th} network output from the input data. As described abstractly in the previous sections of this chapter, the objective is to minimize the total loss, now given by Equation 12. This can be achieved by computing the gradient of the loss function E with respect to each weight (parameter) in the network, i.e. $\partial E/\partial W$, where W now denotes a matrix of all network weights (Rumelhart et al. 1986). This gradient is simply the sum of the partial derivatives for each of the cases from i to N. The derivation of each partial derivative is shown for example in Rumelhart et al. (1986) and in LeCun (1986) and it requires defining the activation functions used in the network, which is why it is not in the interest of this thesis to show the entire derivation here.

After computing the gradient of the loss function E with respect to each weight of the network, it can be used in the weight updating process of the network with an algorithm called the (vanilla) gradient descent, formulized in Equation 13. There exist numerous variations of gradient descent algorithms (Rumelhart et al. 1986, LeCun 1986, LeCun 1988, LeCun et al. 2015), but according to LeCun et al. (2015), stochastic gradient descent is most widely used due to its ability to speed up the convergence process of the optimization, meaning that a stochastic algorithm, where noisy estimates of the gradients are used instead of accurate analytical solutions, speeds up the learning process of the network.

$$\theta^{(k+1)} = \theta^{(k)} - \delta \nabla E\left(x, y, \theta^{(k)}\right) \tag{13}$$

In Equation 13, k denotes the k^{th} iteration of the algorithm and δ denotes a *hyperparameter* called the learning rate, which is an external parameter, set by the researcher at the beginning of the training process to influence to the learning process of the network. As shown in Equation 13, the gradient of the loss function with respect to the network parameters is computed again at each iteration of the algorithm. The algorithm stops either (1) after a certain number of iterations k has been reached, which is again a hyperparameter that must be set by the researcher, or (2) when the algorithm converges, i.e. when $\delta \nabla E\left(x, y, \theta^{(k)}\right)$ becomes so close to zero that $\theta^{(k+1)} \approx \theta^{(k)}$. Figure 11 demonstrates the gradient descent process.

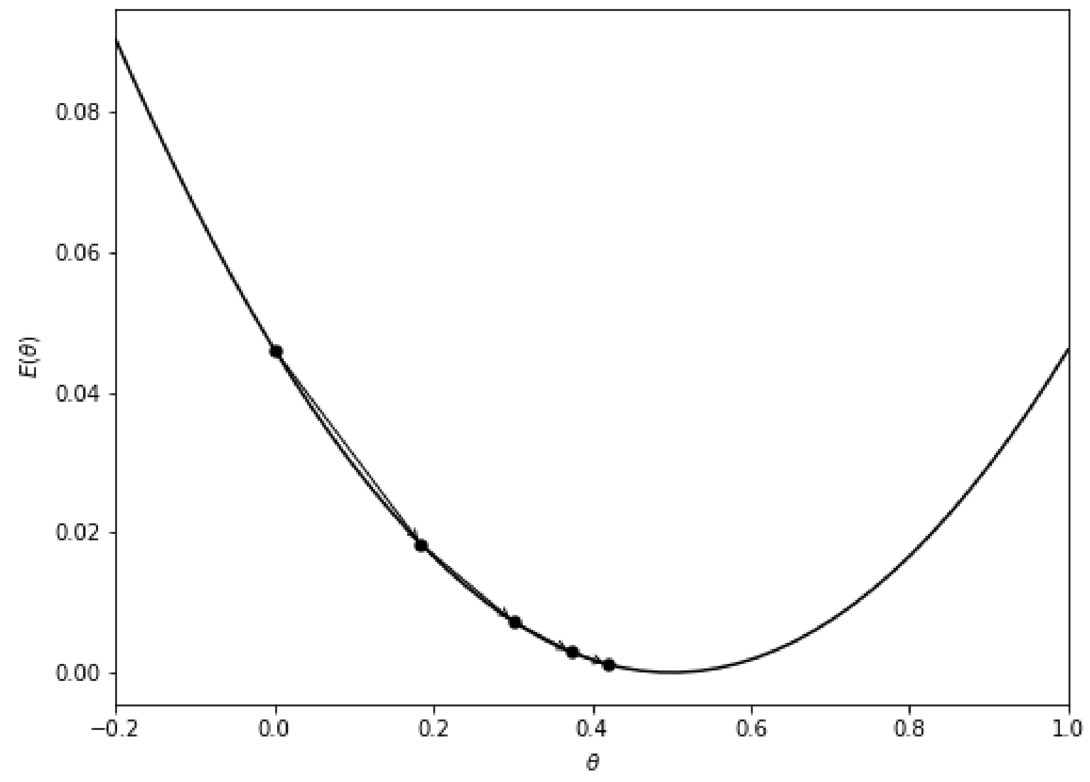

Figure 11. Gradient Descent algorithm

As shown in Figure 11, when the gradient descent converges, the cost function of the network is approximately minimized and the weights that were used in that iteration of the backpropagation algorithm are one set of optimal weights to map that specific input vector x to those specific outputs y. In the late 1990s, prior to recent findings about the backpropagation and gradient descent in large network structures, the academic opinion and that of the machine learning community was that neural networks and backpropagation are infeasible in practice, as it was expected that the gradient found using backpropagation would simply be a poor local minima somewhere in the loss function's surface. According to LeCun et al. (2015), recent research has shown that with large enough networks, the issue of running into very poor local minima is likely to be avoided, regardless of the initial conditions of the system, due to a combinatorially large number of saddle points (points where gradient is zero) in the loss surface with similar values of the loss function. Thus, it does not matter much to which of these saddle points the gradient descent algorithm gets stuck at, as the result is likely to be similar in most of the other points on the loss surface (Dauphin et al. 2014, Choromanska et al. 2014). These findings imply that there exist a large number of network weight combinations that end up in equally favorable saddle points on the loss surface.

2.2.3 Gradient-based optimization techniques

The backpropagation algorithm was defined above using the vanilla gradient descent algorithm to optimize network weights, defined in Equation 13. This subchapter introduces alternative gradient-based optimization methods for the network weights that can be used as an alternative to the gradient descent described above. Gradient-based methods with analytical solutions are not the only option in network weight optimization and are perhaps not always the best approach with respect to finding the global optima, but due to the analytical solution of those techniques they tend to be computationally more efficient than other methods such as the metaheuristic approaches of evolutionary algorithms (See e.g. Vikhar 2016), focusing on finding the optimal weights numerically from arbitrarily chosen starting points, or simulated annealing (See e.g. Kirkpatrick et al. 1983, Černý 1985), which is an approach to approximate the global optima of a function using a simulation approach such as the Metropolis-Hastings algorithm (Metropolis et al. 1953). Thus, this thesis focuses on the gradient-based optimization techniques.

The traditional, also referred to as the vanilla gradient descent algorithm, is a batch gradient descent method where the gradient of the cost function is computed with respect to the network weights (parameters) θ for the entire training dataset, as described in Equation 13. According to Ruder (2016), this can be computationally demanding as it is memory-wise heavy to compute the gradient for the entire dataset, making batch gradient infeasible in situations where the number of weights to optimize is large. Figure 12 demonstrates the smooth convergence process of the batch gradient descent algorithm.

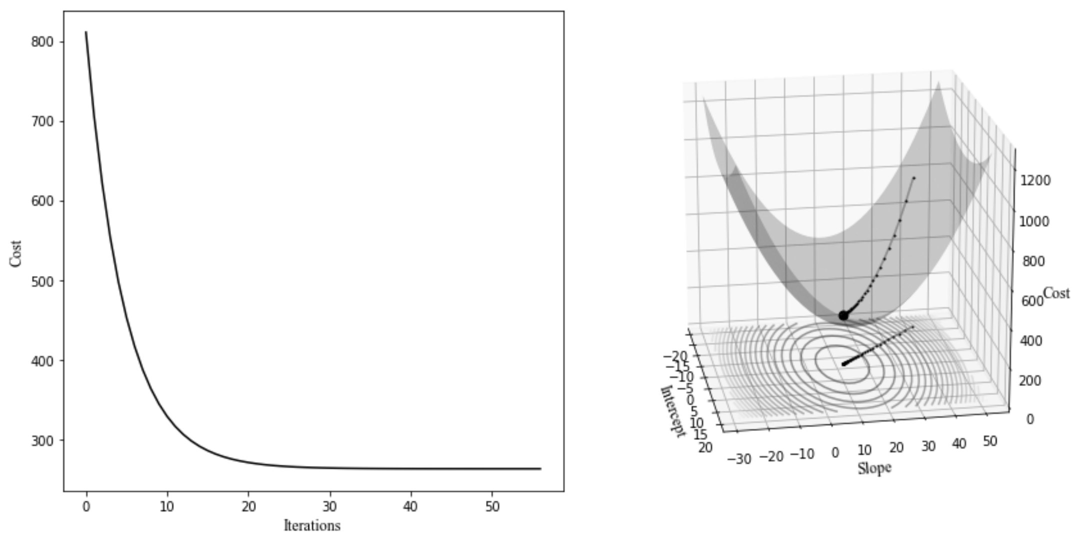

Figure 12. Batch gradient descent convergence

The first alternative to the batch gradient descent is the stochastic gradient descent (SGD), which updates the network weights at each step of the of the backward pass, i.e. each time a partial derivative is computed, the network weights are updated (Ruder 2016). The SGD is defined in Equation 14.

$$\theta^{(k+1)} = \theta^{(k)} - \delta \nabla E\left(x^{(i)}, y^{(i)}, \theta^{(k)}\right) \tag{14}$$

Compared to the batch gradient technique, SGD tends to be faster to compute as the weights are updated at each computation step, whereas the batch gradient approach recomputes the gradient for similar examples of x and y redundantly before the weight update, i.e. the batch gradient method might do many gradient computations that are not exactly needed to eventually reach the local or global optima of the loss function (Ruder 2016). Updating the weights at each example of x and y, the SGD tends to have a higher variance, causing the loss function result to fluctuate more heavily. According to LeCun et al. (2015) and Ruder (2016), these fluctuations sometimes may enable the SGD to also find better local optima than the batch gradient approach as it may jump on the loss surface from one nearby saddle point area to another. Figure 13 demonstrates the SGD convergence process. The fluctuation of the loss function may slow down convergence to the exact optima but it has been shown that by slowly decaying the rate of δ in the optimization process, a similarly good convergence can be achieved compared to the batch gradient method (Ruder 2016).

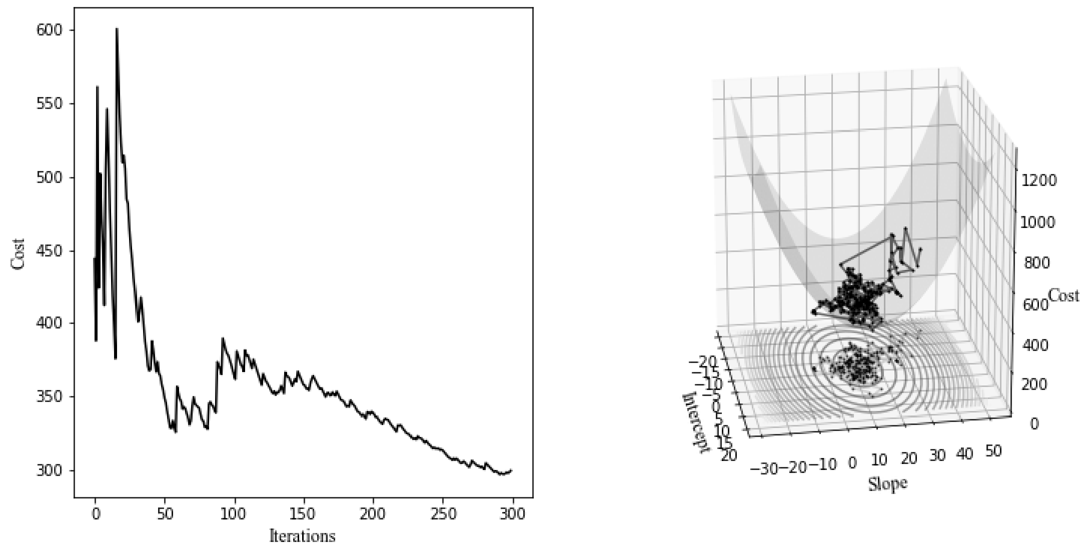

Figure 13. Stochastic gradient descent convergence

Another way of optimizing the network weights is with a mini-batch gradient descent method. This method is a combination of the batch gradient and the stochastic gradient descent approaches, taking a mini-batch of training examples, usually at the size of tens or hundreds of examples and computing the gradient of the loss function based on that sample. At the end of each mini-batch, the network weights are updated and another mini-batch set of data is taken (Ruder 2016). The mini-batch gradient descent algorithm is defined in Equation 15.

$$\theta^{(k+1)} = \theta^{(k)} - \delta \nabla E\left(x^{(i:i+n)}, y^{(i:i+n)}, \theta^{(k)}\right) \tag{15}$$

In Equation 15, n refers to the mini-batch size that is used to compute the gradient. According to Ruder (2016), the mini-batch approach reduces the variance of the weight updates, leading to a more stable convergence to the optima, but it also keeps the benefits of the SGD with faster computation as there are less redundant computations that need to be made, as well as the possibility to jump on the loss surface from one nearby optima area to another. Figure 14 demonstrates the convergence of the mini-batch gradient descent.

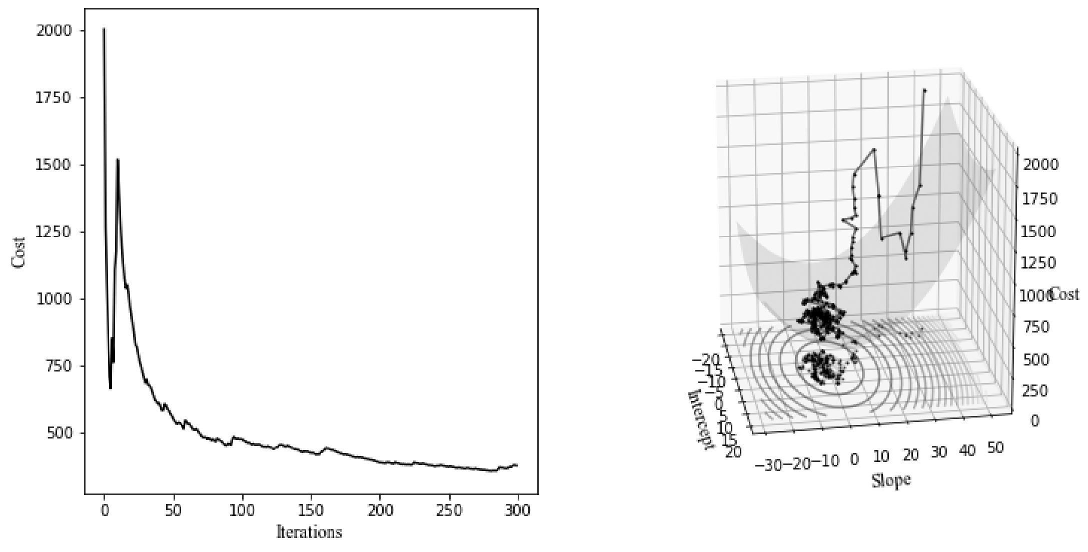

Figure 14. Mini-batch gradient descent convergence

According to Ruder (2016), the gradient descent approaches defined above also impose some challenges in practical machine learning applications such as neural networks, especially when the loss function constitutes a highly non-convex loss surface. The key challenge is to choose an appropriate learning rate δ, which is above all the task of the researcher, as this may significantly influence the convergence of the gradient descent to a proper minimum on the loss function surface. Another challenge according to Ruder (2016) may sometimes be to avoid getting trapped in suboptimal local minima, a point also demonstrated by LeCun et al. (2015).

These challenges are confronted by practitioners and by researchers in academia by using adapted gradient descent algorithms that are designed to tackle the exact challenges mentioned above. Several such algorithms, such as the Momentum algorithm (Qian 1999), the Nesterov Accelerated Gradient (NAG) (Nesterov 2004), Adagrad (Duchi et al. 2011), RMSProp (Hinton et al. 2012) and Adam (Kingma & Lei Ba 2014) have been proposed, all imposing some improvement that either tackles the choice of the learning rate or the avoidance of suboptimal local minima. These algorithms are a part of algorithms known as the first-order optimization algorithms that are based only on computing the gradient of the loss function. Another set of similar algorithms are known as the second-order optimization algorithms that also compute the Hessian of the loss function and often require numerical approaches as the Hessian may not always be computable analytically. It is beyond the scope of this thesis to describe these practitioners' use-case algorithms in depth, as they require the reader to be familiar with numerical methods and stochastic optimization. For an introduction, see e.g. Bubeck (2014).

This chapter described how a neural network such as the MLP learns by updating its network weights using backpropagation with a gradient descent algorithm. Today, one of the most popularly used neural networks is the convolutional neural network (also referred to as the CNN, ConvNet), already described earlier as being one of the logical followers of Fukushima's Neocognitron (See e.g. Fukushima 1980). While the Neocognitron used pre-wiring of network weights, the CNNs of today, such as the AlexNet (Krizhevsky et al. 2012) and the ZF Net (Zeiler & Fergus 2013) usually use similar supervised learning methods that were described in this chapter such as the backpropagation algorithm and stochastic gradient descent. These convolutional neural networks will be described more thoroughly in the next chapter.

2.3 Convolutional Neural Networks

Since inception at the early 1990's, convolutional neural networks have exhibited great performance at different tasks such as image and pattern recognition, signal processing, text- speech- and video recognition and classification tasks such as animal classification from images (Zeiler & Fergus 2013, Simonyan & Zisserman 2014). CNNs became clearly the most popular neural network structure after Krizhevsky et al. (2012) demonstrated great success in ImageNet 2012 classification benchmark with their CNN called AlexNet, which achieved an error rate of some 16 %, while the 2nd best error result was around 26 %.

The success of AlexNet in 2012 inspired aspiring research around CNNs, resulting in new record-low error rates in the ImageNet classification benchmark in the following years. Zeiler and Fergus (2013) utilized smaller receptive window size and smaller strides in their convolutional layers to win the title of best-performing submission in the ImageNet classification competition of 2013, while Simonyan and Zisserman (2014) explored deeper network structures with small convolutional layers stacked up to nearly 20 layers deep, resulting in even lower error rates than in the previous years on the ImageNet classification benchmark.

Before the ImageNet classification challenge of 2012, CNNs were considered by the machine learning community as unpractical, as the model structures are relatively complex compared to a feedforward neural network such as the MLP described in the previous chapter (Zeiler & Fergus 2013, LeCun et al. 2015) with training time measured in weeks, instead of days or a few hours. The recent advancements in availability of large pre-classified data sets for images, speech and other sorts of signals now usually used to train CNNs, as well as the advancements in GPU-based network training implementations and better model regularization practices, CNNs have become exceedingly practical and outstandingly well-performing choices of network for learning tasks of many sorts (Krizhevsky et al. 2012, Zeiler & Fergus 2013, LeCun et al. 2015, Simonyan & Zisserman 2014). It now takes somewhere between minutes to hours to train a CNN with millions of weight parameters and extremely deep architectures.

Compared to the MLP introduced in the previous chapter, convolutional neural networks differ from the more traditional feedforward structure in the sense that convolutional neural networks are structured to process data incoming from multiple arrays (vectors) instead of a simple input vector (LeCun et al. 2015). The overall structure of a neural network therefore differs from the MLP in a radical way. A simple example of such an input data is a basic color image, which composes of three two-dimensional arrays containing pixel intensities in the three color channels known as RGB channels (Red, Green, Blue channels).

According to LeCun et al. (2015), there are four key elements that convolutional neural networks take advantage when processing natural signals, compared to a normal feedforward network structure: CNNs use local connections between neurons, shared weights between neurons in the same unit, pooling layers (which will be described more thoroughly below) and deep structures, i.e. they tend to be 10-20 layers deep.

As already stated, the architecture of a convolutional neural network differs from the MLP described in the previous chapter quite radically. Rather than the feedforward network with weighted connections between each layer and activations in each layer, a CNN is structured as a series of stages, where each stage forms a different sort of a function in the network (LeCun et al. 2015). Typically, the first few stages are composed of convolutional layers and pooling layers. The role of the convolutional layer is to detect local conjuctions of features from the previous layer, while the pooling layer's role is to merge semantically similar features found by the convolutional layer to a combined set of features (Krizhevsky et al. 2012, Zeiler & Fergus 2013).

The convolutional layers are able to detect local conjuctions of features by a specific structure of those layers. The neurons in a convolutional layer are organized in feature maps, where each neuron is connected to local patches in the feature maps of the previous layer through a set of weights called in the CNN world as a filter bank. In each layer, there can be an arbitrary amount of feature maps that the neurons are connected to. Similarly to the MLP, an incoming input to each feature map is passed through a non-linearity (activation function) as a local weighted sum of the filter bank and the inputs (LeCun et al. 2015). Different feature maps in each layer use different filter banks but the neurons connected to mutual feature maps share the same filter banks with each other. Figure 15 demonstrates the convolutional layer in of a CNN.

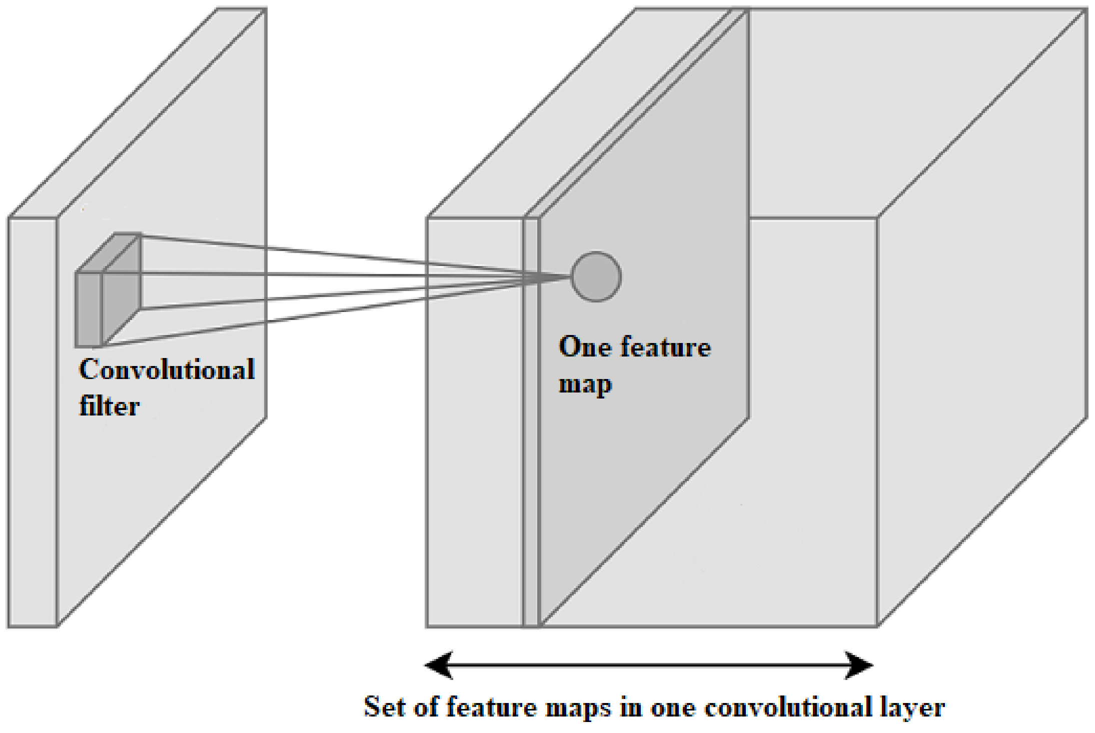

Figure 15. Convolutional layer

According to LeCun et al. (2015), there are two particular reasons for such a structure where neurons share a filter bank together instead of having individual weight connections with the previous layer: first, in an array dataset such as a set of images, local groups of values (i.e. the pixel intensities) are highly correlated, meaning in practice that nearby pixels are more likely to be of approximately the same color, thereby forming distinctive local patterns that can be easily detected. The second reason for such an architecture is that typically, for such cases where local groups of values are highly correlated, the local statistics of that pattern are shift invariant, i.e. they can appear in other parts of the input data as well (Krizhevsky et al. 2012, Zeiler & Fergus 2013, LeCun et al. 2015). Then, sharing weights with each other, a neuron located in some other place but connected to the same filter bank can detect a pattern already previously detected in some other neuron (Krizhevsky et al. 2012, LeCun et al. 2015).

The pooling layer after the convolution layer is designed to deal with these shift invariant patterns, for which the relative positions can vary across the data, by simplifying the position of each feature. The pooling layer does this coarse-graining process by computing the maximum of a local patch of neurons in a feature map or in a set of feature maps (LeCun et al. 2015). Neighbouring neurons in the pooling layer take input from other patches of neurons in a feature map that are shifted by one or more rows or columns in the data, reducing the dimensionality of the representation and creating an invariance to small shifts and distortions, thereby simplifying the overall view of patterns across the data, even when the elements (e.g. signals in the data) in the previous layer vary in position and appearance (Krizhevsky et al. 2012). Figure 16 demonstrates the pooling layer of a CNN.

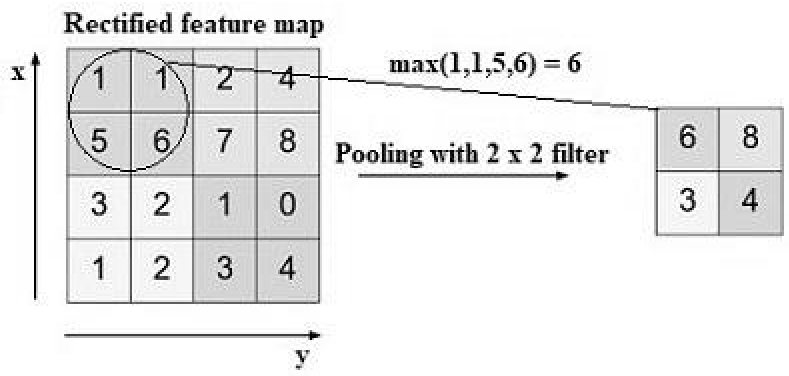

Figure 16. Pooling layer

A typical structure of a convolutional neural network has a few layers of convolutions accompanied by pooling layers in the beginning of the network. After these layers there are typically more convolutional layers and at the end of the network, there are fully connected layers similar to those of the MLP, structuring the found patterns to an output through activations (Krizhevsky et al. 2012, LeCun et al. 2015). Figure 17 demonstrates the architecture of a convolutional neural network. Similarly to the MLP, the weight optimization of the network can be achieved by backpropagation using a gradient-based search method such as a gradient descent algorithm (Krizhevsky et al. 2012, LeCun et al. 2015).

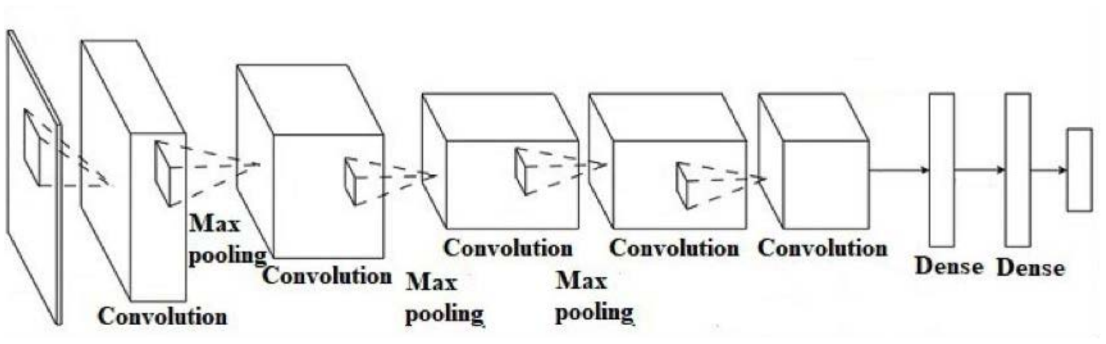

Figure 17. Convolutional Neural Network

As the amount of weights to be fitted tends to be very large in CNNs, the networks are easily bound to overfit the input data with the output data, meaning that the network only learns such features from the input training data that match the output training data extremely well, dismissing features that would be perhaps relevant but are not necessarily present in that particular set of output data (Krizhevsky et al. 2012, LeCun et al. 2015). Krizhevsky et al. (2012), Zeiler and Fergus (2013) and Simonyan and Zisserman (2014) proposed two ways to avoid overfitting: firstly, they found that creating augmented data from the dataset was crucial to reduce the overfitting of the network on training data, therefore improving their results on the test dataset.

Secondly, they use a *dropout* layer, first introduced by Hinton et al. (2012), between some of the fully-connected layers, which sets the output from each neuron in the subsequent layer to 0 with some arbitrary probability p, e.g. $p = 0.5$. This way some of the neurons are dropped out from the neural network's results and do not contribute to the output in the forward pass or in calculating the gradient of the loss function in the backward pass, though these dropped out neurons continue to share their weights within the feature maps of that layer. According to Krizhevsky et al. (2012) and Simonyan and Zisserman (2014), by doing a such a dropout of some neuron outputs, the neural network is forced to sample a different network architecture each type input data is presented, reducing the co-adaptation of neurons as they cannot rely on the presence of similar other neurons, therefore improving the network's ability to learn more robust features from the input data.

As a response to convolutional neural networks Zeiler and Fergus (2013) proposed a multilayered Deconvolutional Network (deconvnet) to project feature activations from a convolutional neural network back to the input space. They found that the features extracted from an input dataset during the training of a convolutional neural network are far from random, also visualizing the learned features in each of the feature maps they presented in their study. According to Zeiler and Fergus (2013), their deconvnet revealed that datasets such as the ImageNet images they used for training their network showed many intuitively desirable properties for the model to learn, such as compositionality, increasing invariance and class discrimination.

The findings of Zeiler and Fergus (2013) using their deconvnet provide a significant motivation for exploring how deep neural networks in general and especially a convolutional neural network behave and learn using a financial dataset instead of images, as their findings imply that similar properties could perhaps be found on a financial task such as option pricing as well. In fact, already in 1994, Hutchinson et al. predicated that neural networks can learn hidden structures such as volatility curves from a dataset without specifically giving them as input to the network. The next chapter introduces the reader to previous literature on deep learning in option pricing, as well as provides more reasoning to use non-parametric models such as neural networks to price options instead of parametric ones.

3 MACHINE LEARNING IN OPTION PRICING

This chapter gives an overview of the relevant literature on neural networks that have been researched by academia in pricing options. Reasoning based on previous literature on why to compare neural networks to parametric models in option pricing is given.

3.1 Overview of machine learning in option pricing

Probably the most cited paper and one of the very first in academia to explore the use of neural networks in option pricing was the one by Hutchinson et al. (1994), who explored three different non-parametric approaches to price and delta-hedge call options, comparing their non-parametric results to those of the traditional BSM model. In their paper, they study and compare the performance of an MLP (introduced above), a radial basis function (RBF, see e.g. Powell 1987) and a projection pursuit approach (PP, see e.g. Huber 1985, Jones & Sibson 1987) in learning the standard BSM model with an additional delta-hedging experiment, where they put their models to a test to experiment whether the models have truly learned the dynamics of the underlying with respect to the option price. Following their initiate, a number of papers with a similar topic were published in the following years.

In their groundbreaking paper, Hutchinson et al. (1994) proposed a non-parametric data-driven approach to learn the dynamics of the underlying asset's price process from the data directly, therefore allowing the existing market data to determine both the dynamics of the underlying S_t and its relation to the prices of the derivative securities C_t with minimal assumptions on both, S_t and the pricing function used to match the price of the underlying to that of the derivative security. In their paper, they state that any economic variable could be used as input variable, if it could possibly influence the price of the derivative security.

The approach Hutchinson et al. (1994) took in their paper was to use as input variables the underlying asset price, the strike price of each contract and the time to maturity of each contract, which were mapped to the call option price of each contract through the previously mentioned non-parametric models, i.e. the MLP, the RBF and the PP. In their study, they rely on the *universal approximation theorem* (Cybenko 1989, Hornik 1989, Hornik 1991), which states that neural networks with at least one hidden layer, such as the MLP, can approximate any arbitrarily chosen continuous function arbitrarily well. According to Hutchinson et al. (1994), translating the universal approximation theorem to the financial space implies that any derivative pricing function, such as the BSM or the market's pricing function, could be approximated arbitrarily well using a neural network with at least one hidden layer of neurons.

The assumption of Hutchinson et al. (1994) that a derivative's pricing function is continuous is a fair assumption to be made, but it has been shown e.g. by Merton (1976) and Cox and Ross (1976) that the stochastic processes of the underlying asset can also exhibit jumps that are not bound to be continuous but rather arrive in random time intervals. These jumps are often modeled using a Poisson process and the existence of jumps in an option price series (and in the time series of the underlying asset) can possibly affect the pricing result of the neural network model negatively.

Following the theorem of Merton (1990, pp. 202-203), which states that the underlying asset's return r_t is independent of the level of the underlying asset's price S_t and therefore the pricing function $f(\cdot)$ used to determine the price of an option C_t, given the input variables, is homogenous of degree one in both, the underlying asset's price S_t and the strike price K, Hutchinson et al. (1994) normalize their data by dividing their inputs with the strike prices K. This results in only two input variables to the neural network, i.e. the underlying asset's price divided by the strike price (moneyness) S_t/K and the time-to-maturity $T-t$. The data normalization used by Hutchinson et al. (1994) was also used in later studies on neural networks and option pricing (e.g. Anders et al. 1996, Yao et al. 2000, Amilon 2003, Gencay & Salih 2003, Bennell & Sutcliffe 2004). Equation 16 demonstrates this normalization scheme.

$$\hat{C}_t = f(S_t, K, T-t)$$

$$\widehat{C_t/K} = f(S_t/K, 1, T-t) \tag{16}$$

When first introduced by Hutchinson et al. (1994), their MLP model was state-of-the-art of that time with one hidden layer of four neurons and the two input variables. In their study, they performed pricing performance experiments as well as delta-hedging experiments on the neural network that was used to learn the BSM model. They expected the neural network models to perform better in delta-hedging experiments than the BSM due to the fact that the network was trained directly on discrete data, while the BSM expects continuous time hedging, which is bound to be ineligible in practice due to the fact that time, and therefore any financial time series, can only be measured discretely, resulting in some *tracking error* of the hedging portfolio that reflects the error between a continuous-time hedge and a discrete-time hedge.

The main findings on their simulated data set that Hutchinson et al. (1994) made was that their neural network models were able to learn the BSM on an arbitrarily good level of accuracy only after six months of daily training data. They also found the RBF and the MLP to outperform the BSM in 36 % (RBF) and 26 % (MLP) of their delta-hedging experiment test paths. They showed that for other than at-the-money and near-the-money options (NTM), the neural network models tended to outperform the BSM in a delta-hedging experiment reasonably well. Although left unexplained by Hutchinson et al. (1994), the result for the neural network to outperform especially well for out-of-the-money (OTM) options might be related to the volatility structure: deep OTM options tend to be more volatile (Fortune, 1996) and hence the BSM may misprice them, as it assumes a constant volatility structure.

In their study, Hutchinson et al. (1994) found that on the simulated, BSM created data set the BSM is superior to all of the neural network approaches in delta-hedging experiments. For their market data test set on the other hand, Hutchinson et al. (1994) found that the MLP outperforms the BSM model in their hedging experiment roughly 65 % of the time. Their results were also statistically significant on a paired t-test that was performed on the hedging errors, implying statistical significance of the neural network’s ability to outperform the traditional BSM in delta-hedging. However, the statistical significance of their results are ought to be approached with caution, as their price series may have been overlapping which would result in statistically dependent data points (Amilon 2003).

Several papers on how to use neural networks in option pricing have been published after the original paper by Hutchinson et al. (1994). Each of these papers, such as Anders et al. (1996), Garcia and Gencay (2000), Yao et al. (2000), Amilon (2003), Gencay and Salih (2003) and Bennell and Sutcliffe (2004) provide additional insights and improve the methods that were developed in the original paper, but all of them still rely on fairly shallow feed-forward networks that were already applied by Hutchinson et al. (1994). These additional insights and improvements include for example introducing statistical inference techniques in network choice (Anders et al. 1996), study of neural networks' ability to learn volatility structure (Yao et al. 2000, Gencay & Salih 2003), applying statistical inference to models' results (Anders et al. 1996, Garcia & Gencay 2000, Amilon 2003, Gencay & Salih 2003), neural network regularization techniques (Gencay & Salih 2003) and testing the neural network against different types of Black-Scholes models (Bennell & Sutcfliffe 2004).

In contrast to Hutchinson et al. (1994), Anders et al. (1996) focus solely on MLP networks, arguing that compared to RBF networks, MLPs can not only approximate any arbitrary function with an arbitrary degree of accuracy, but also approximate that function's derivatives with a similar degree of accuracy. This approximation capability is a requirement for analyzing the hedging parameters of a pricing function (Hutchinson et al. 1994, Anders et al. 1996). The derivative approximation property is in fact present also in RBF networks (See e.g. Girosi & Poggio 1990) but according to Anders et al. (1996), the more important reason to solely focus on MLP networks is because they can then directly apply statistical inference techniques in choosing the neural network model that they use in out-sample testing. In comparison, Hutchinson et al. (1994) used cross-validation techniques that are eventually based on trial and error, while according to Anders et al. (1996), statistical inference can decrease model selection time rapidly.

The statistical inference techniques Anders et al. (1996) rely on are the work by White (1989), who showed that if neural network parameters are identified, they can be consistently estimated using Maximum Likelihood Estimates (MLEs) and moreover, that the parameter estimates of a neural network are asymptotically normally distributed and therefore subordinate to standard asymptotic tests such as the Wald-test (Greene 2012, pp. 155-161) or the LM-test (Engle 1984, pp. 796-801).

Anders et al. (1996) use two neural networks in their study to learn the market's pricing function using empirical data from DAX, a German stock index. These networks are a vanilla MLP and a combination of an MLP and the BSM model. They assume to gain benefits from combining the neural network with the BSM, letting the network only focus on parts where there are differences between BSM predicted prices and the actual market prices. Their network architecture is a one hidden layer network with three hidden neurons and a linear activation prior to the output layer. This architecture is chosen based on the statistical inference techniques that they introduced, comparing the statistical relevance of each additional hidden neuron. Compared to Hutchinson et al. (1994), Anders et al. (1996) also introduce more input variables to the network, namely the estimated volatility and risk-free rate that they estimate from available market data.

In terms of pricing error, Anders et al. (1996) found that both of their neural network models dominated the BSM model with significantly smaller pricing errors in the test sample, measured in terms of four different error functions, and a significantly more explanatory model, measured in terms of R^2. In terms of hedging parameters, Anders et al. (1996) found that their neural network models were bound to defective especially the delta term (Equation 30) when the options were deep in-the-money. This result was suspected to be due to less trading volumes for those options, resulting in less observations and an uneven fit of the network parameters to be only precise for shorter-term near the money options (Anders et al. 1996).

Similarly to Anders et al. (1996), Yao et al. (2000) focus on MLP networks in their aim to predict the price of Nikkei 225 stock index options (Nikkei 225 is a large Japanese stock index for which there also exists exchange traded futures contracts). Similarly to Hutchinson et al. (1994), they also use only one hidden layer in their network and test the amount of optimal hidden neurons using a validation set between the training set and the test set. Compared to previous studies, Yao et al. (2000) partition their data already prior to training to different moneyness types, leaving them with seven different network architectures. They use similar inputs as Hutchinson et al. (1994) with asset prices and option prices normalized using the strike prices, together with the time-to-maturity. Yao et al. (2000) argue that using either historical or implied volatility as an input restricts the neural network's ability to learn the volatility from the data implicitly. They also argue that using similar inputs to the neural network as are used to the BSM could perhaps cause unintended assumptions, as the neural network model is then forced to learn similar sort of price dynamics to that of the BSM. Yao et al. (2000) also claim to have experimented using a time-varying and a constant interest rate as an input but with poor results, leaving interest rate out from the inputs prior to model training phase.

In their study, Yao et al. (2000) test multiple networks using different data ordering, for example by ordering the data randomly, as well as ordering the data using the time-to-maturity. They find that placing the data on dated ordering provides the best results in terms of minimized pricing error, claiming that the results are due to the implicitly computed volatility structure found by the neural network, which is disturbed if the data is not sequentially ordered by its date. Although not statistically validated, these results do support the implicit computation of volatility structure when the neural network a priori does not know that structure, also suggested by Hutchinson et al. (1994) and Anders et al. (1996).

For each of the neural networks studied, Yao et al. (2000) find that their neural network models perform particularly well in pricing for ITM and OTM options, also finding that there is not much difference between the pricing error of the neural network models and the BSM for ATM or NTM options. They claim that their models work particularly well when volatility is higher than the market's expectations, although not statistically tested for in the study. Unlike in previous literature, Yao et al. (2000) do not perform a hedging analysis, which has been emphasized to be of equal practical importance when comparing option pricing formulae (Hutchinson et al. 1994, Anders et al. 1996, Amilon 2003).

Amilon (2003) introduce several improvements to the original paper by Hutchinson et al. (1994). In his work, similarly to Anders et al. (1996) and Yao et al. (2000), Amilon (2003) focuses on MLP networks. Unlike in previous literature, he uses more input variables and determines the model output to be rather the bid and ask prices of the call options on OMX index (Swedish stock index) instead of a middle point determined scalar valued price, allowing the network to intrinsically also learn the spread of the bid and ask prices. The inputs are determined as the underlying asset's price, time-to-maturity in trading days, strike price, time-to-maturity in calendar days, risk free rate and additional, model specific inputs such as lagged asset price values and slowly varying volatility estimates estimated from the underlying asset's returns. According to Amilon (2003), including lagged values of the underlying asset's price should enforce the network to learn a volatility structure for the underlying asset's price. The models are compared to two BSM models with different volatility properties, namely one based on historical volatility and the other on implicitly estimated volatility.

Similarly to previous literature, Amilon (2003) also uses an MLP network with one hidden layer but a ranging number of hidden neurons from which the best model is chosen for separate test sets based on a validation data set, summing up to 20 different networks that are compared for each test set. The testing is divided into pricing and hedging tests. In the latter, Amilon (2003) explores the network's ability to outperform BSM in a dynamic delta-hedging strategy. In the study, statistical inference measures are also introduced for measuring the statistical significance of the pricing and hedging results, based on a block bootstrap technique by Efron (1979) and Küncsh (1989).

Amilon (2003) found in his study that the MLP models used in the test set had a tendency to outperform the BSM models in terms of both pricing and hedging. By comparing the models on different moneyness types he found that especially for OTM options, the MLPs provided far better pricing results. In terms of hedging, the MLPs provided significantly smaller tracking errors than the BSM models.

Gencay and Salih (2003), building on the work by Garcia and Gencay (2000), focus on MLP networks and introduce network selection tools to choose the optimal network architecture from an arbitrarily large set of networks that could possibly be used in the task of option pricing. In contrast to prior work where the model selection has been solely based on lower error rates in the validation set (Hutchinson et al.

1994, Yao et al. 2000, Amilon 2003) or in statistical inference techniques (Anders et al. 1996), Gencay and Salih (2003) introduce a Bayesian Regularization scheme to their study, where the optimal weights are chosen such that it maximizes a Bayesian posterior probability representing the researcher's knowledge of weights prior to any data is collected (For a detailed introduction see e.g. MacKay 1992, Foresee & Hagan 1997). By maximizing the Bayesian posterior probability, which is equivalent to minimizing a regularized cost function (MacKay 1992), they solve the optimal number of parameters (weights) that should be present in their network.

Gencay and Salih (2003) also introduce other regularization methods to improve the learning ability of their network with a method called bootstrap aggregating (bagging) and a method to avoid overfitting in training called early stopping. These methods are beyond the scope of this thesis but it is relevant to note that prior to their study, these issues were handled more or less heuristically in the literature regarding neural networks in option pricing. For detailed introductions on bootstrap aggregating, see Breiman (1996). For early stopping methods, see Girosi and Poggio (1995).

Similarly to previous literature, Gencay and Salih (2003) also use an MLP network with one hidden layer and a varying number of hidden neurons from one to ten neurons in each model, using the above mentioned regularization techniques to choose the optimal model in the validation set prior to each test period. They use options data on the S&P 500 stock index with five years in total sample, from which they divide the first two quarters to training, the third quarter to validation and the final quarter to testing. They claim that by dividing the data in such a way, it makes the years more comparable to each other, as the testing is always done in the latter part of each year. The inputs that they feed to the network are the exact replicas of the inputs that are given to the BSM model, claiming that using exact inputs in both the neural network models and the BSM make them more comparable. In contrast to Amilon (2003) with block bootstrapping, Gencay and Salih (2003) use the Diebold-Mariano (1995) statistic to analyze the statistical significance of their results. According to Amilon (2003), using the Diebold Mariano statistic must be approached with caution, as the option data is often not pure time series, resulting in overlapping sequences of data that are then not necessarily independent of each other.

Gencay and Salih (2003) find that their neural network models, especially the ones regularized with Bayesian Regularization, provide significantly smaller pricing errors in comparison to the BSM model, though the statistical significance of their Diebold-Mariano statistic is questionable for some of the results, with fairly large associated p-values. Gencay and Salih (2003) show that especially for deep OTM options, the BSM overestimates the prices substantially, while the neural network models on the other hand correct their pricing accordingly for the deep OTM options, resulting in small pricing errors for these options. Gencay and Salih (2003) claim that the result is due to a relaxed functional form at which the BSM may be constraining the data unnecessarily.

In their study, Gencay and Salih (2003) also investigate the effect of volatility to the pricing error and find that while the BSM tends to have fairly high pricing errors during periods of extremely low or extremely high volatility, especially for deep OTM options, the neural network models learned to price the options also in periods of extreme volatilities better, with mean pricing error around zero for all volatility levels. This gives some empirical proof for the hypothesis of Yao et al. (2000), that neural network models tend to provide better results than the BSM for high volatility assets. A similar finding was made for time-to-maturities, where the BSM tended to overprice deep OTM options for short time-to-maturities and underprice NTM options for long time-to-maturities, while the neural network models' pricing errors were located around zero for all maturity levels.

Bennell and Sutcliffe (2004) use an MLP network to price European FTSE 100 call options (FTSE 100 is a British stock index) and compare their pricing results to those of the BSM, also using Merton's (1973) correction for dividends in the BSM model. The prior literature to Bennell and Sutcliffe (2004) use mainly the uncorrected version of the BSM model, i.e. the B&S model (Black and Scholes 1973), while not correcting for dividends even when the data is directly associated with an equity index (e.g. Anders et al. 1996, Gencay & Salih 2003). They use similar inputs to the neural network as they use for the BSM, accompanied with the Merton dividend adjustment, open interest and daily volume of each option contract, estimating the risk-free rates and volatilities from available market data. In accordance to prior literature, Bennell and Sutcliffe (2004) also normalize their input data using the strike prices. They use a similar data partition scheme to Yao et al. (2000), dividing the data based on each observation’s moneyness and training three different MLP networks for each moneyness type (OTM, NTM, ITM). Similarly to previous literature, they only use one hidden layer with a varying number of hidden neurons and choose the best network structure based on validation set results.

By comparing the pricing results of inputting the underlying asset’s price and strike prices separately and on the other hand by using the normalization scheme, Bennell and Sutcliffe (2004) found that using the normalization scheme based on the homogeneity assumption of option prices, their network performance improved significantly. They also found that for OTM options, the MLP models were significantly better than the BSM model with far less pricing deviations than the BSM model, but that the BSM model outperformed the MLP models for ITM options in all performance measures. They concluded that especially for ITM options with longer maturities, there is not enough data to train the network on, as most of the trading occurs in short-term NTM options. By using a restricted training set from which they restricted deep ITM options, they found the MLP to be superior to the BSM in all performance measures.

Table 1. Comparison of previous literature's network architectures

Authors	Year	Network architecture	No. hidden layers	No. neurons	No. inputs	Empirical data
Hutchinson et al.	1994	MLP, RBF, PP	1	4	2	S&P 500 futures ix.
Anders et al.	1996	MLP	1	3	4	DAX 30 stock ix.
Yao et al.	2000	MLP	1	2-6	3-4	Nikkei 225 stock ix.
Amilon	2003	MLP	1	10-14	9	OMXS 30 stock ix.
Gencay and Salih	2003	MLP	1	1-10	5	S&P 500 stock ix.
Bennell and Sutcliffe	2004	MLP	1	3-5	3-7	FTSE 100 stock ix.

Table 1 compresses the relevant network architectures from the literature reviewed above. Based on the previous literature, it has been found that clearly, at least for European style OTM options, MLPs tend to generally perform better than the BSM model. For NTM and OTM options, the relationship does not seem so clear, as the results found in previous literature have been in conflict with each other: some results imply that the neural network model is superior on all moneyness types and time-to-maturities, while some imply that especially in cases of ITM options, the results may not be as clear. A distinct similarity in all of the previous literature can be found: all of them use a shallow neural network structures with only one hidden layer and fairly low amount of neurons in that layer, varying from three to ten neurons. This may have restricted the learning process of the networks to only learn the most obvious relationships in the underlying data, as it has been shown that learning elegant underlying features demands deeper (more layers) and wider (more neurons in each layer) network structures (Krizhevsky et al. 2012, Zeiler & Fergus 2013, LeCun et al. 2015).

Prior to moving to the empirical section of this thesis, additional reasoning is given for comparing a non-parametric approach of the neural network to parametric models such as the BSM in option pricing.

3.2 Machine learning vs. parametric models

In their original paper, Hutchinson et al. (1994) argued that the reason for the failure of parametric approaches to price derivatives (e.g. the BSM model) is due to the fact that a parametric pricing formula, no matter whether derived analytically or numerically, eventually depends on the particular parametric form of the underlying asset's price dynamics S_t, while the misspecification of the stochastic process for S_t will lead to systematic failures of the pricing formula in the form of pricing and hedging errors, when the pricing formula is used to price derivative securities linked to S_t. Therefore, any model used to price derivative securities is tied to the ability to capture the dynamics of the underlying asset's price process.

According to Hutchinson et al. (1994) Garcia and Gencay (2000), Yao et al. (2000), Gencay and Salih (2003) and Amilon (2003), the problem with parametric models is that the user of such model is required to specify the dynamics of the underlying asset's price process before using the model, which then necessitates some set of assumptions regarding those prices.

A traditional example of such assumption misspecification is the BSM model, which assumes that the underlying S_t follows a Geometric Brownian Motion (GBM) with a constant volatility and constant interest rates, also assuming continuous trading with no taxes applied to the underlying asset and that the market is frictionless. A number of biases have been documented for the BSM model's outputted prices when it has been studied for equity index options (See e.g. Bates 1996): The model's prices exhibit a volatility smile, which results from a known bias called the moneyness bias, implying that the model's prices are lower than the actual prices for deep OTM and deep ITM options. In addition, a term-structure bias and a put-call skew bias have been documented. The former implies that the BSM model's prices vary with option maturity while the latter implies that the BSM model's prices depend on the relative volume of the puts and calls traded. Also, it has been shown that volatilities or interest rates are not time-invariant processes (See e.g. Engle 1982, Bollerslev 1986).

Other parametric pricing formulae have been introduced in literature that try to take into account the assumptions that were misspecified in BSM, such as the non-normal asset return distributions, stochastic volatilities and stochastic interest rates (See e.g. Merton 1976, Cox & Ross 1976, Rubinstein 1983, Hull & White 1987, Duan 1995). However, all of these models still assume that the underlying asset's price of the derivative security follows some specific stochastic process, which drives the underlying asset's price dynamics (Anders et al. 1996, Yao et al. 2000, Gencay & Salih 2003).

Assuming that the stochastic process of an underlying's asset price stay constant over time might be a wrongly placed assumption in the first place, as has been hypothesized by Bennell and Sutcliffe (2004), who suggests that price processes across assets may vary and by Lo (2004), who argues that markets work in adaptive phases, where the dynamics of the market (and therefore, the underlying asset's prices in that market) might shift due to external forces such as market participants' behavior, forcing the market to adapt to new dynamics as an evolutionary process. The adaptive markets hypothesis introduced by Lo (2004) implies that the stochastic process driving the underlying asset's price dynamics might change over time, which could also be one of the reasons that lead to failures of the parametric pricing formulae that assume specific stochastic structures to take place in the underlying asset's price.

Non-parametric models such as neural networks on the other hand use the market's opinion, i.e. the market data to determine the current implicit stochastic process driving the underlying asset's price dynamics and the relationship of that price to that of a derivative security (Hutchinson et al. 1994, Anders et al. 1996, Yao et al. 2000, Gencay & Salih 2003, Sutcliffe 2004). The pricing formula is thus learned directly from the market and reflects the market's pricing formula of that moment, which can be adapted to changing market dynamics using the latest market data available.

According to the Universal Approximation Theorem (Cybenko 1989, Hornik et al. 1990, Hornik 1991) introduced in the previous chapter, the multilayer feed forward architecture of a neural network in particular enables the neural network to act as a universal function approximator. More specifically, Cybenko (1989), Hornik et al. (1990) and Hornik (1991) showed that neural networks can approximate any arbitrary continuous function arbitrarily well, i.e. with any desired accuracy, thus implying that neural networks have more general and flexible functional forms than any other statistical method can effectively process (Yu et al. 2007). Therefore, neural networks are especially great in such tasks where a highly non-linear pricing formula is to be derived directly from the market, which can exhibit adaptive structures over time, as long as enough parameters (weights) can be introduced to the mapping process of the neural network (Hornik et al. 1989, Hornik 1991).

Zeiler and Fergus (2013) also showed that deep models such as convolutional neural networks can implicitly compute correspondence between specific objects in images, e.g. that faces of a mammal have a specific spatial configuration of the eyes and nose. This implies that deep models can to some extent compute any sort of correspondence between two different observations in a given dataset implicitly. Their findings are remarkable and bring evidence to the assumptions by Hutchinson et al. (1994), Yao et al. (2000) and Gencay and Salih (2003) that deep models can implicitly compute other sorts of configurations from a different type of dataset, such as volatility structures from a model trained to price options even when the underlying asset's price dynamics are unknown.

With the knowledge gained from previous literature on neural networks and option pricing, and on the other hand with the recent advancements in deep learning models such as the convolutional neural networks, it can be hypothesized that such function approximator as the convolutional neural network should provide at least equal, if not better results in pricing options of any kind. As was shown in the previous subchapter, research has found that already with shallow network structures, neural networks have outperformed the traditional parametric option pricing models on a distinctive level. It is a reasonable assumption that with today's computation power and commoditized, open-source machine learning tools such as Tensorflow, Caffe, Torch and Theano, when training a network up to 20 layers deep with millions of parameters takes only a few hours on a singular GPU, one can achieve at least as good results as have been achieved in the previous studies.

4 EMPIRICAL METHODOLOGY

4.1 The data

The data that is used in the empirical analysis are daily closing quotes of the S&P 500 stock market index and daily closing quotes of the S&P 500 stock index options, collected from OptionMetrics. The options are European style call options and the data is daily data, starting from Jan 1996 and ending to Aug 2014. The S&P 500 stock market index is based on the market capitalizations of 500 large companies listed on the NYSE or NASDAQ stock markets. The index components and their weightings are determined by S&P Dow Jones Indices.

The full dataset consists of 2 175 217 observations with a total of 6 812 calendar days between the first and the last data point and 22 017 different option contracts with strike prices varying between 50 - 3 000. In addition to the stock index and index options data, a time-varying estimate of a risk-free rate is approximated using the continuously compounded return of the 90-day US treasury bill. The rates data is collected from FRED.

Following the approach taken by Anders et al. (1996), the data is filtered using the following exclusion criteria to remove non-representative observations.

1. The call option is traded below 10 basis points (C_t is the call price at time t)

$$C_t \leq 0.1 \tag{17}$$

2. The time-to-maturity ($T - t$) of the option contract is above two years or less than 15 days (252 is used as the number of trading days in one year)

$$T - t \geq 2 \text{ or } T - t \leq 15/252 \tag{18}$$

3. The option is at extreme moneyness states, i.e. the option is either deep OTM or deep ITM (S_t is the spot price of the underlying, K is the strike price of the option contract and S_t/K is the moneyness of the option)

$$S_t/K < 0.5 \text{ or } S_t/K > 1.55 \tag{19}$$

4. The lower bound condition for a call option price is violated (r_t is the risk free rate at time t)

$$C_t \geq S_t - Ke^{-r_t(T-t)} \tag{20}$$

According to Anders et al. (1996), options tend to be traded at integer values, leading to relatively high deviations between observed and theoretical option prices when the option values are very low. Thus, the first criterion excludes options with very low prices. The second criterion excludes options with very low or very high time to maturity, as these options have only a small, or on the other hand relatively high time-value, with which the integer pricing behaviour leads to large deviations between theoretical and real option prices. The third criterion excludes options that are either very deep OTM or very deep ITM, as these options are mostly traded at their intrinsic value and have very low trading volumes, making them non-representative presentations of the reality (Anders et al. 1996). The fourth criterion excludes options, whose prices are not consistent with the no-arbitrage condition, which is according to Hull (2009, pp. 219-220) binding for all European-style options, independent of the pricing model.

After removal of non-representative observations, the dataset consists of total 1 119 771 observations. Figure 18 presents the call prices, normalized with their strikes, as a function of their moneyness and time-to-maturity.

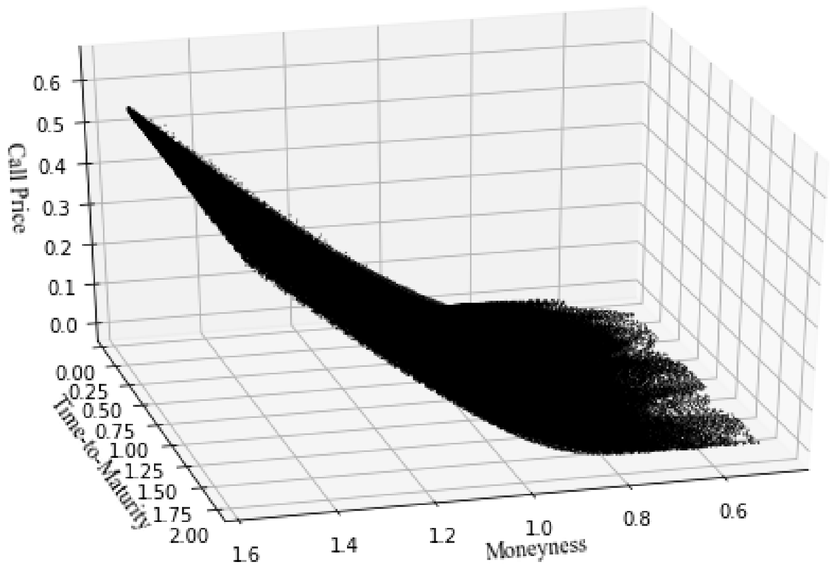

Figure 18. Normalized call prices as a function of moneyness and time-to-maturity

According to Hutchinson et al. (1994), Anders et al. (1996), Gencay and Salih (2003) and Bennell and Sutcliffe (2004), short-term NTM options are the mostly traded options in the market, while longer term options that are either OTM or ITM rarely exhibit equal amounts of trading volume. Figure 18 shows that there exists a decent amount of observations for all maturities and moneyness levels. Figure 19 presents the distribution of the moneyness.

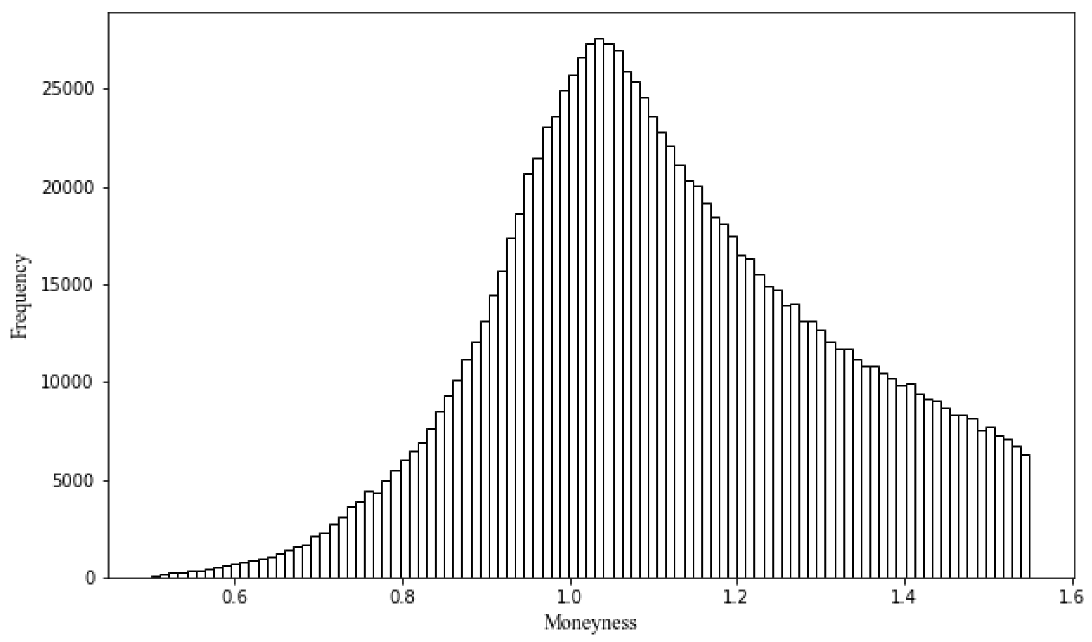

Figure 19. Distribution of moneyness as a histogram of spot prices divided by strike prices

Figure 19 shows that there clearly have been some options that were extremely ITM, as the histogram cuts them out following the exclusion criteria for moneyness levels that are over 1.55. The development of the spot prices across the dataset's time period is also an interesting feature to observe, as it tells the researcher whether the time series is trending and how should the dataset be divided such that the train set and test set are equal representations of the dataset. Figure 20 displays the spot price development of the underlying asset's price over the dataset time period.

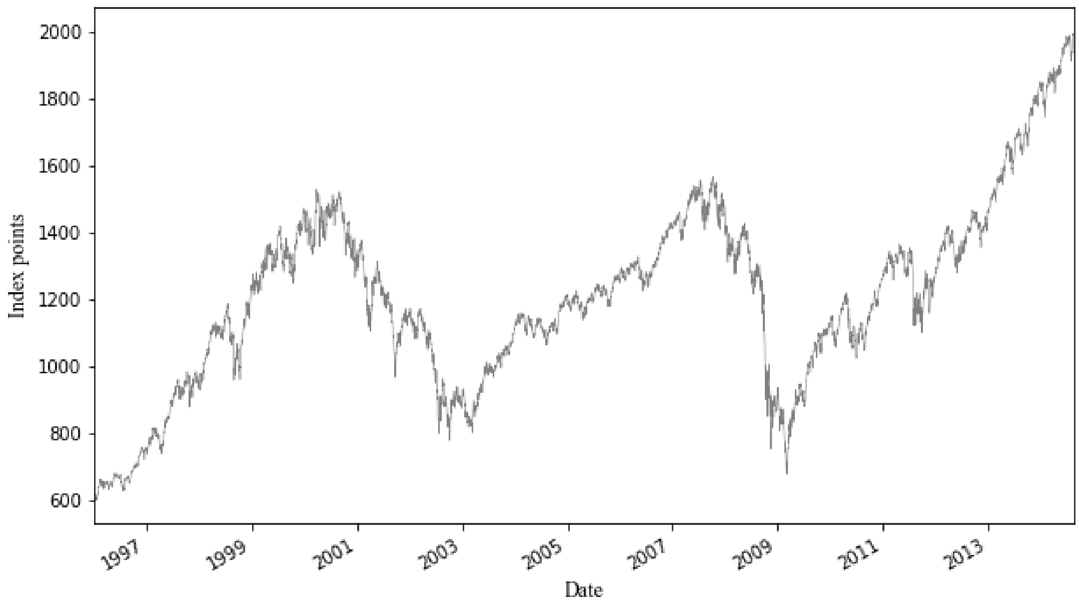

Figure 20. S&P 500 Index spot price between 1996-2014

From Figure 20 one can observe that there exists periods, when the index is rising and on the other hand periods, where the index has dropped quite a lot, such as in the tech bubble (1999-2003) and during the financial crisis (2007-2009). The dataset with non-representative observations excluded is further divided into a training set and a test set. The training set will be used to train the neural network, while the test set will be used to compare the out-sample pricing and hedging performance of the neural network against the BSM model. The train and test set distributions are 75% and 25% and from the train set, 10% will be used for validation purposes when training the network. That leaves a total number of 755 845 observations for training purposes, 83 982 observations for validation purposes and 279 942 observations for testing purposes. The training period ranges from 1996 to midst of 2011 with a total of 12 671 option contracts and strike prices varying between 400 and 2 000, while the test period ranges from midst 2011 to 2014 with 5 072 option contracts and strike prices varying between 710 and 2 500.

4.2 Research environment

The research experiment will be conducted using Python (version 3.6), which is a versatile open-source programming language that is used for example in software development, data analysis, machine learning and finance (Bowles 2015, Hilpisch 2015). All data is processed using an interactive Python Notebook called Jupyter Notebook and stored in a HDF5 data library.

The neural network used in the analysis is implemented with *Keras,* which is an interface to Tensorflow, an open-source software library for numerical computation using data flow graphs (Abadi et al. 2016). It was originally created by a team of researchers and engineers at Google Brain. The computations are performed on GeForce GTX 1070 8GB GPU, on which it takes about three to four hours to train the network.

4.3 Network architecture

Diverging from prior studies of neural networks in option pricing, the neural network used in this research experiment is a convolutional neural network instead of a normal multilayer perceptron. The network follows a one-channelled adapted version of the AlexNet, originally introduced by Krizhevsky et al. (2012).

The network consists of the input layer, four convolutional layers accompanied with pooling layers and three fully connected layers prior to the output layer. The rectifier is used in each layer as the activation function apart from the output layer. The output layer consists of one neuron, activated by an exponential function. The exponential function is chosen because it is bound to be positive for all positive values of x and because it was found to provide good learning results in terms of approximating the Black-Scholes' pricing function using a deep MLP network architecture (Culkin & Das 2017). The numbers of filters in the convolutional layers are 64, 128, 192 and 256, while the numbers of neurons in the fully connected layers prior to the output layer are 9216, 4096 and 1028.

The network is trained with mini-batch gradient descent on a batch size of 64 and validated at each iteration with 10 % of the input data. The specific gradient descent algorithm used is the Adam algorithm (Kingma & Lei Ba 2014) with a learning rate of $\delta = 0.001$ and betas of $\beta_1 = 0.9$ and $\beta_2 = 0.999$. The loss function of the network is determined as the Mean Squared Error (MSE), introduced in chapter two and demonstrated in Equation 12. Figure 21 demonstrates the network architecture used in this thesis.

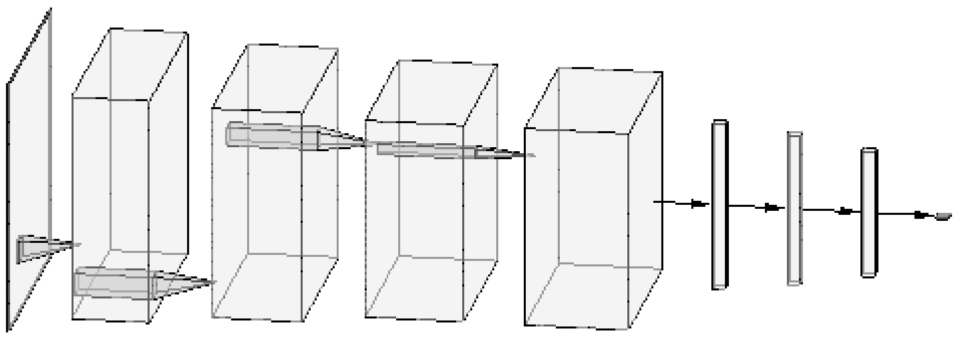

Figure 21. Neural network architecture

Two state-of-the-art deep learning methods are applied to the network to reduce overfitting and accelerate the learning process. In each hidden layer, a scheme called Batch Normalization (BN) (Ioffe & Szegedy 2015) is applied to reduce the covariate shift between each hidden layer's output. The BN applies a transformation to the activation of the previous layer at each mini batch, i.e. it normalizes the outputs from the previous layer to keep the mean of the output close to zero and the standard deviation close to one, normalizing the distribution of the data between each layer. Ioffe and Szegedy (2015) showed that this reduces the amount of iterations needed to train the network, as well as allows the user to be less careful in the initial hyperparameter setting. In addition, each of the fully connected layers is accompanied with a dropout scheme (Hinton et al. 2012), introduced in chapter two, which is applied to force the network to sample different network architectures, which should reduce overfitting and improve the learning ability of complex features found in the data.

The inputs that are used for the network are the underlying asset's spot price, the strike price, the time-to-maturity, the risk-free rate estimated from the 90-day US treasury bill and four volatility estimates estimated from the historical price series of the underlying asset, with 30-day, 60-day, 90-day and 120-day moving averages.

Thus, the input matrix then comprises of the underlying asset's price series S_t, the strike prices K, the time-to-maturities $T - t$, the risk-free rate r_t and the volatility estimates σ_{30}, σ_{60}, σ_{90} and σ_{120}. The network is used to approximate the market's option pricing function, which can be specified as some arbitrary $f(\cdot)$ such that

$$\hat{C}_t = \boldsymbol{f}(S_t, K, T - t, r_t, \sigma_{30,60,90,120}) \tag{21}$$

Where $\hat{C}_t$ is the call price estimated by the neural network. According to Amilon (2003), a normalization of the input data to the same order of magnitude is often an effective way of decreasing a neural network's complexity. Gencay and Salih (2003) and Bennell and Sutcliffe (2004) showed that normalizing the input price series and the output option price series with strikes resulted in smaller error rates between the network's predictions and the true values of the option price. Thus, following the approach originally taken by Hutchinson et al. (1994) and assuming the facet derived from the Black-Scholes function that call prices are linearly homogenous in S/K, i.e. $C(S/K) = K * C(S/K, 1)$ also applies to other call price functions, the underlying asset's prices and the call prices are normalized with the strikes before passing them on to the neural network such that

$$\hat{C}_t = \boldsymbol{f}(S_{t,}, K, T - t, r_t, \sigma_{30,60,90,120})$$

$$\Rightarrow \widehat{C_t/K} = \boldsymbol{f}(S_t/K, 1, T - t, r_t, \sigma_{30,60,90,120}) \tag{22}$$

The feature set fed to the neural network is then the input varibles $(S_t/K, 1, T - t, r_t, \sigma_{30,60,90,120})$ which are mapped through the network's pricing function $f(\cdot)$ to the normalized option prices $\widehat{C_t/K}$.

4.4 Performance metrics

4.4.1 Pricing performance

The pricing performance of the neural network will be compared to the basic Black-Scholes option pricing model. For simplicity, Merton's (1973) adjustment for dividends will not be employed in this comparison. The famous Black-Scholes model price for a call option at time t can then be defined as

$$C_t = S_t N(d_1) - K e^{-r(T-t)} N(d_2) \tag{23}$$

$$d_1 = \frac{\ln\left(\frac{S_t}{K}\right) + (r + \frac{\sigma^2}{2})(T-t)}{\sigma\sqrt{T-t}} \tag{24}$$

$$d_2 = d_1 - \sigma\sqrt{T-t} \tag{25}$$

where $N(\cdot)$ is the cumulative distribution function of the standard normal distribution, S_t is the spot price of the underlying asset at time t, $T-t$ is the time-to-maturity of the option, σ is the volatility of the underlying asset's price and r is the risk-free rate. The volatility estimates are approximated similarly to Hutchinson et al. (1994), using the rolling standard deviation with a varying window size of 30, 60, 90 and 120 days, demonstrated in Equation 26, where s denotes the standard deviation of the continuously compounded daily returns of the S&P 500 stock market index from the past $t = (30{,}60{,}90{,}120)$ days and $\hat{\sigma}$ is the t day volatility estimate, quoted in annual terms with 252 used as the number of trading days per year.

$$\widehat{\sigma_t} = s_t\sqrt{252} \tag{26}$$

The BSM model's pricing errors will be computed on each of the possible running volatility estimates for the training set and the BSM model with the lowest root mean squared pricing error (Equation 28) will proceed as the best BSM model to the test set, where it will be compared to the neural network model. The risk-free rate is estimated with the 90-day US treasury bond yield. Figure 22 presents the estimated time-varying volatilities for each of the running volatility estimates. Figure 23 presents the estimated risk-free rate that is used as input to the Black-Scholes pricing formula.

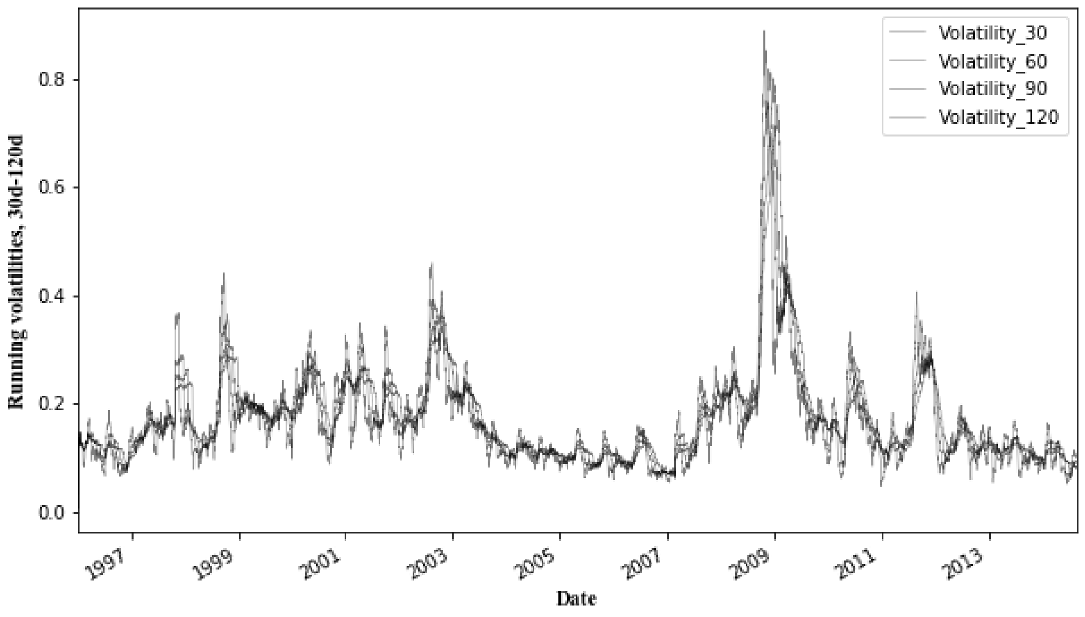

Figure 22. 30-120 day running volatility estimates for S&P 500

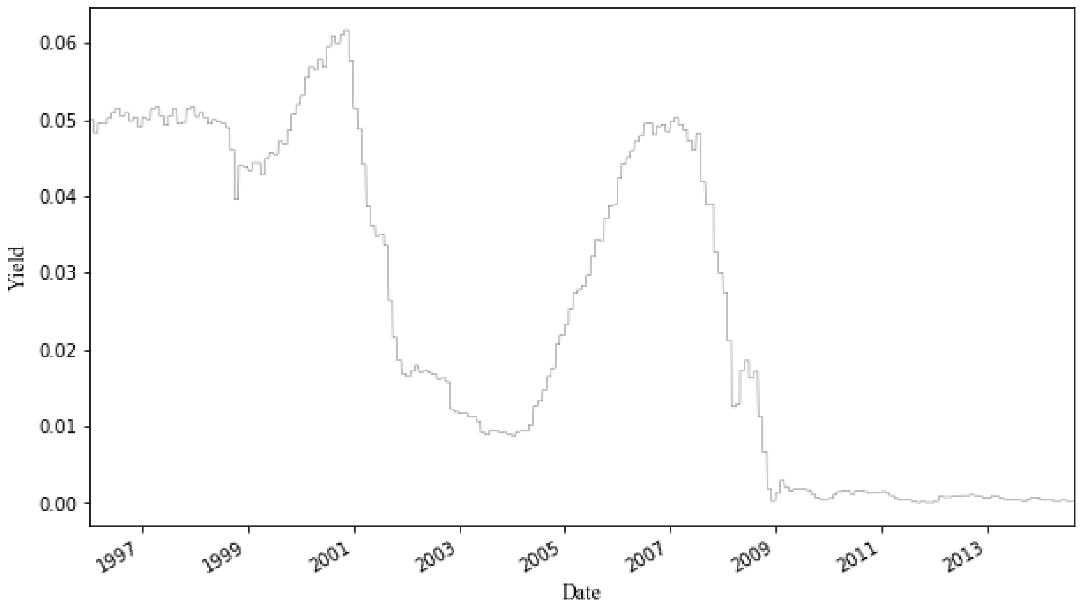

Figure 23. US 3-month treasury bond yield

The pricing performance will be evaluated with three different error functions, following the work of Anders et al. (1996), Bennell and Sutcliffe (2004) and Culkin and Das (2017): The Mean Squared Error (MSE), the Root Mean Squared Error (RMSE) and Mean Absolute Error (MAE). These error estimates are defined in Equations 27-29. For simplicity, here $C_i = C_{i,t}/K$.

$$MSE = \frac{1}{N}\sum_{i}^{N}(\widehat{C_\iota} - C_i)^2 \tag{27}$$

$$RMSE = \sqrt{\frac{1}{N}\sum_{i}^{N}(\widehat{C_\iota} - C_i)^2} \tag{28}$$

$$MAE = \frac{1}{N}\sum_{i}^{N} | \widehat{C_\iota} - C_i| \tag{29}$$

where N is the size of the test set, $\widehat{C_\iota}$ is the neural network's i_{th} call price estimate and C_i is the i_{th} real call price.

Statistical inference is applied to the error estimates with similar approach to Amilon (2003), who proposed an overlapping block-sampling (bootstrapping) scheme based on the works of Efron (1979) and Künsch (1989). According to Amilon (2003), using the Diebold and Mariano (1995) statistic to test the covariance stationarity of the errors is biased, as the reordering of the errors changes the autocovariance functions and thus the test statistic. Furthermore, it was shown by Fitzenberg (1998) that the block-sampling method is valid under weaker assumptions than the covariance stationarity.

Following the approach of Amilon (2003), the method goes as follows: draw a day from the test sample with probability $1/D$, where D is equal to the length of days in the test sample. Construct a block of neural network errors of size L by joining the network's errors of that day and of $L - 1$ consecutive days. Repeat the procedure until the bootstrapped series is about the same number of blocks as the original series and compute the statistics of interest. Repeat the process for $n = 1000$ times and extract the desired confidence intervals from the resampled distribution. This statistical inference technique will be applied to test the statistical significance of both, the pricing error estimates as well as the delta-hedging performance measures, described in the next subchapter. Before describing the delta-hedging experiment, qualitative measurements used to measure the pricing performance will be described.

Yao et al. (2000) and Gencay and Salih (2003) gathered general findings that have been made in research about the BSM model's ability to price options correctly. These findings can be presented in four main conclusions:

1. The BSM is a good model when pricing ATM and NTM options, especially when the time-to-maturity is above two months

2. The pricing accuracy of the BSM for deep ITM and OTM option decreases significantly and the model's prices exhibit particularly large deviations from the actual prices on these types of options

3. Short-term options (with time-to-maturity less than one month) are often mispriced by the BSM

4. Options on assets with extremely low or extremely high volatility ($\sigma \leq 0.05, \sigma \geq 0.8$) are often mispriced by the BSM

In addition to the quantitative pricing performance measures and statistical inference, the neural network model will be also qualitatively tested against these known biases of the BSM model by comparing the pricing ability of the neural network model on different moneyness types as well as on different time-to-maturities and historical volatilities. The performance will be mainly measured visually, by plotting the pricing results as a function against the known biases.

4.4.2 Hedging performance

Hutchinson et al. (1994) argue that even though of great significance, approximating prices with a good accuracy is not yet a sufficient test of how well a neural network model has learned the dynamics of an underlying asset's price and the relationship between the underlying's asset price to that of a derivative security. Similarly, Amilon (2003) argues that market's option prices reflect the expectation of the future distribution of the underlying asset and, if these expectations are not filled on average, market prices are wrong and any measures-of-fit based on pricing accuracy are bound to be off. Thus, according to Hutchinson et al. (1994) and Amilon (2003), performing a hedging analysis, where the option price is replicated with a portfolio of arbitrary other assets, is crucial to determine whether the neural network model is of practical use, since the very existence of an arbitrage-based pricing function is predicated on the ability to replicate an option position using a dynamic hedging strategy.

The idea of a dynamic hedging strategy is to set up a hedging portfolio that offsets the risk of an option position. This hedging portfolio works correctly, if the combined value of the option position and the hedging portfolio is zero at the expiration of the option. In the Black-Scholes world, it is assumed that hedging in continuous time is possible. In reality, hedging continuously is impossible as time can only be measured discretely. Thus, the combined value of the hedging portfolio and the option position is unlikely to be zero at expiration and there exists some difference, known as the *tracking error* between the two positions. The tracking error may be positive or negative, reflecting either a profit or a loss that is associated with the taken hedging position. According to Hutchinson et al. (1994) and Amilon (2003), this tracking error can be used as a performance measure to test different option pricing models' performance against each other, representing a measure-of-fit of the model. The model with a tracking error nearest to zero is a more accurate model to be used in practical hedging situations.

The dynamic hedging strategy that will be employed in this thesis, following the approaches of Hutchinson et al. (1994) and Amilon (2003), is the *delta-hedging strategy*, where the aim is to neutralize the *delta*, i.e. the sensitivity of the option price to changes that occur in the underlying asset's price. Formally, the delta Δ (30) is defined as the partial derivative of the option price with respect to the underlying asset price.

$$\Delta = \frac{\partial c_t}{\partial s_t} \tag{30}$$

It should be noted that as the option price used to approximate the delta is model dependent, so are the partial derivatives computed from it. Thus, the delta is different when computed for the BSM model and for the neural network's pricing model. As was described in chapter three, approximating first-order derivatives from a function that a neural network has learned should be possible to an arbitrary degree of accuracy (Hornik 1991).

Formally, the delta-hedging experiment can be structured as follows. Denote V_t as the total value of the replicating (hedging) portfolio at time t and let V_t be defined as

$$\mathrm{V}_t = \mathrm{V}_t(S) + \mathrm{V}_t(B) + \mathrm{V}_t(C) \tag{31}$$

where $\mathrm{V}_t(S)$ is the value of the underlying asset position, $\mathrm{V}_t(B)$ is the value of a bond position used to finance the position in the underlying asset and $\mathrm{V}_t(C)$ is the value of the option position held in the portfolio at time t. At initialization, $\mathrm{t} = 0$, the composition of the portfolio is assumed to be

$$\mathrm{V}_0(S) = \mathrm{S}_0 \mathit{\Delta}_{NN,0} \tag{32}$$

$$\mathrm{V}_0(C) = -C_{BSM,0} \tag{33}$$

$$\mathrm{V}_0(B) = -(\mathrm{V}_0(S) + \mathrm{V}_0(C)) \tag{34}$$

$$\mathit{\Delta}_{NN,0} = \frac{\partial \mathrm{C}_{NN,0}}{\partial S_0} \tag{35}$$

where $C_{BSM,0}$ is the call option price given by the BSM and $\mathit{\Delta}_{NN,0}$ is the delta of the call option, computed from the neural network's pricing formula with $\mathrm{C}_{NN,0}$ denoting the call price computed by the neural network. The portfolio positions in Equations 32-35 represent going short one call option, going long the underlying asset for $\mathit{\Delta}_{NN,0}$ number of shares at price S_0 and borrowing the amount $\mathrm{V}_0(S) + \mathrm{V}_0(C)$ to finance the rest of the long position in the underlying asset that is not financed with the sale of the call option. Since the long position is entirely financed with riskless borrowing and the sale of a call option, the initial value of the replicating portfolio is then identically zero.

$$\mathrm{V}_0 = \mathrm{V}_0(S) + \mathrm{V}_0(B) + \mathrm{V}_0(C) = 0 \tag{36}$$

After initialization at $t = 0$ and prior to expiration of the call option at $t = T$, the stock and the bond positions are updated in the replicating portfolio at discrete and regular time intervals of length τ to satisfy the following relations:

$$\mathrm{V}_t(S) = \mathrm{S}_t \mathit{\Delta}_{NN,t} \, , \mathit{\Delta}_{NN,t} = \frac{\partial \mathrm{C}_{NN,t}}{\partial S_t} \tag{37}$$

$$\mathrm{V}_t(B) = e^{r\tau}\mathrm{V}_{t-\tau}(B) - \mathrm{S}_t(\mathit{\Delta}_{NN,t} - \mathit{\Delta}_{NN,t-\tau}) \tag{38}$$

where r is the risk-free rate and τ is defined to be one day in this experiment. The tracking error of the replicating portfolio at expiration $t = T$ is then simply the value of the replicating portfolio at expiration, V_T (39). Following Hutchinson et al. (1994) the present value of the expected absolute tracking error is used as the final performance measure in the hedging analysis, defined in Equation 40.

$$\mathrm{V}_T = \mathrm{V}_T(S) + \mathrm{V}_T(B) + \mathrm{V}_T(C) \tag{39}$$

$$\epsilon = e^{-rT}\boldsymbol{E}[|\mathrm{V}_T|] \tag{40}$$

A similar analysis can be performed to the BSM model by replacing the $\mathit{\Delta}$ of the neural network to the $\mathit{\Delta}$ of the BSM model, i.e.

$$\mathit{\Delta}_{BSM,0} = \frac{\partial \mathrm{C}_{BSM,0}}{\partial S_0} \tag{41}$$

In this thesis, the hedging performance will be compared with a delta-hedge analysis for the BSM model and the neural network model. The comparison will be performed on the test set of the data for each of the option contracts that have over 50 days of observations in the test set. Statistical inference will be applied in the form of overlapping block sampling, introduced in the pricing performance section above.

5 EMPIRICAL RESULTS

This chapter presents the results of the empirical study performed in this thesis. The chapter is divided into two subchapters: the first subchapter presents the pricing performance results, while the second subchapter presents the hedging performance results.

5.1 Pricing performance results

Table 2 presents the pricing performance results, which are further divided to three separate tables: the first table contains the results of the entire test set, while the second and the third tables contain results partitioned by moneyness and time-to-maturity. The results are shown in columns MSE, RMSE and MAE for the neural network model and the BSM model separately, with 95% bootstrapped confidence intervals computed in parentheses below the values. In Table 2 b), ITM refers to $S/K > 1.02$, OTM to $S/K < 0.98$ and NTM to $S/K \geq 0.98, S/K \leq 1.02$. In Table 2 c), short time-to-maturity refers to $T - t < 1/12$, long time-to-maturity to $T - t > 1/2$ and medium time-to-maturity to $T - t \geq 1/12$ and $T - t \leq 1/2$.

Table 2 a). Pricing performance results - whole test set

Model	No. Observations	MSE	RMSE	MAE
Neural network		0.000014	0.003690	0.002610
CI	279942	(0.000014, 0.000014)	(0.003681, 0.003700)	(0.002606, 0.002614)
Black-Scholes		0.000167	0.012913	0.007236
CI		(0.000165, 0.000168)	(0.012851, 0.012977)	(0.007203, 0.007268)

Table 2 b). Pricing performance results - partitioned by moneyness

Model	Moneyness	No. Observations	MSE	RMSE	MAE
Neural network			0.000015	0.003857	0.002758
CI	ITM	186595	(0.000015, 0.000015)	(0.003845, 0.003869)	(0.002752, 0.002763)
Black-Scholes			0.000129	0.011363	0.006389
CI			(0.000127, 0.000131)	(0.011291, 0.011436)	(0.006364, 0.006417)
Neural network			0.000009	0.003059	0.002086
CI	OTM	69523	(0.000009, 0.000010)	(0.003031, 0.003089)	(0.002070, 0.002101)
Black-Scholes			0.000264	0.016244	0.009307
CI			(0.000260, 0.000268)	(0.016125, 0.016361)	(0.009227, 0.009382)
Neural network			0.000016	0.004025	0.002982
CI	NTM	23824	(0.000016, 0.000016)	(0.003985, 0.004062)	(0.002959, 0.003002)
Black-Scholes			0.000178	0.013336	0.007825
CI			(0.000170, 0.000186)	(0.013046, 0.013626)	(0.007759, 0.007890)

Table 2 c). Pricing performance results - partitioned by time-to-maturity

Model	Maturity	No. Observations	MSE	RMSE	MAE
Neural network	Short	15871	0.000007	0.002611	0.001949
CI			(0.000007, 0.000007)	(0.002580, 0.002647)	(0.001935, 0.001963)
Black-Scholes			0.000008	0.002883	0.001915
CI			(0.000008, 0.000009)	(0.002839, 0.002927)	(0.001894, 0.001938)
Neural network	Medium	160780	0.000008	0.002822	0.002090
CI			(0.000008, 0.000008)	(0.002813, 0.002832)	(0.002085, 0.002095)
Black-Scholes			0.000036	0.006039	0.003898
CI			(0.000036, 0.000037)	(0.006002, 0.006074)	(0.003881, 0.003913)
Neural network	Long	103291	0.000023	0.004844	0.003521
CI			(0.000023, 0.000024)	(0.004823, 0.004865)	(0.003511, 0.003533)
Black-Scholes			0.000394	0.019846	0.013249
CI			(0.000389, 0.000398)	(0.019731, 0.019949)	(0.013164, 0.013330)

Table 2 a) shows that the neural network model is superior to the BSM model in terms of pricing for all error measures. The results are statistically significant on 1% level. This result is consistent with previous literature, where the MLP was also found to be superior to the BSM model when the dataset was not partitioned by any measure (Hutchinson et al. 1994, Anders et al. 1996, Gencay & Salih 2003, Bennell & Sutcliffe 2004). The most intuitive error measure is the RMSE, which gives the pricing error as a percentage of the strike price (Hutchinson et al. 1994). Thus, it can be computed that on average, the neural network model prices options nearly 10 percentage points more correctly than the BSM model on the entire test set. The results for the entire test set can be seen in Figures 24 and 25, which show the models' predicted price as a function of the actual price.

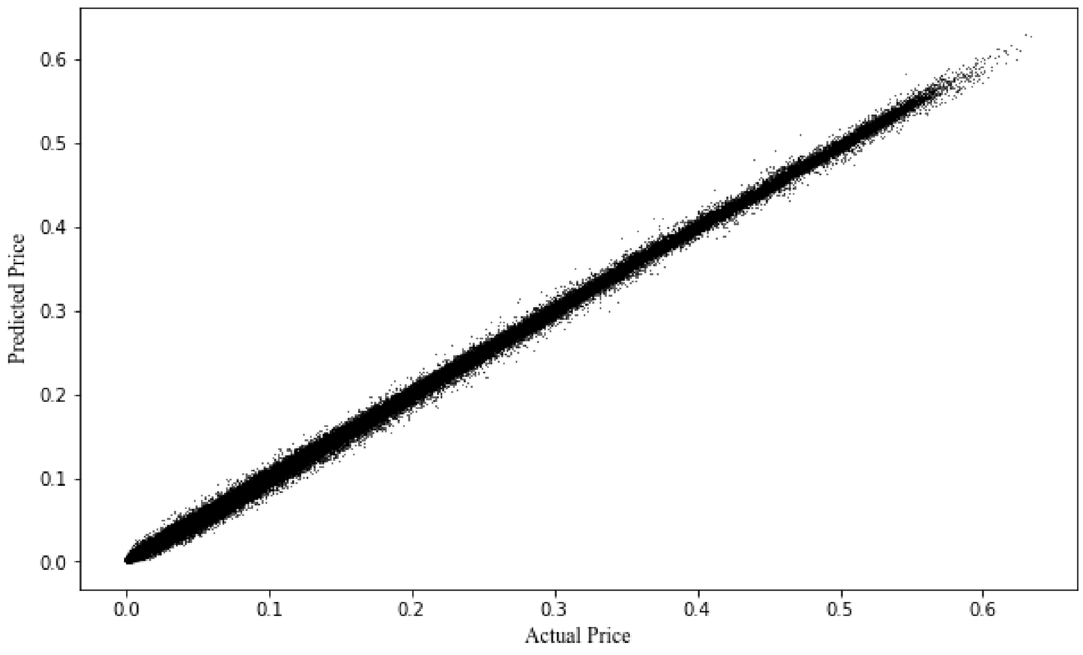

Figure 24. Neural network predicted price vs. actual price

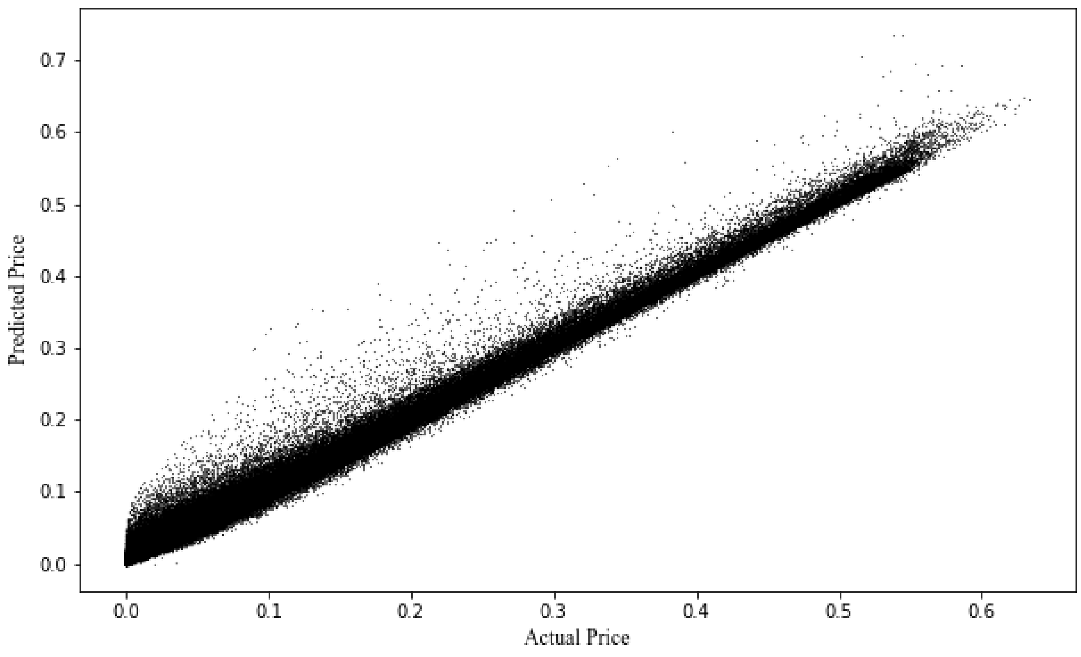

Figure 25. BSM predicted price vs. actual price

From the two figures, it can be seen that there is clearly less deviation from the straight line for the neural network model, while the BSM model's predictions tend to deviate especially when the actual price is small. It can also be observed that for the BSM, the line is skewed downwards and the predictions do not entire match those of the actual price. This can also be observed by looking at the histograms of the pricing error, shown in Figures 26 and 27 for the two models.

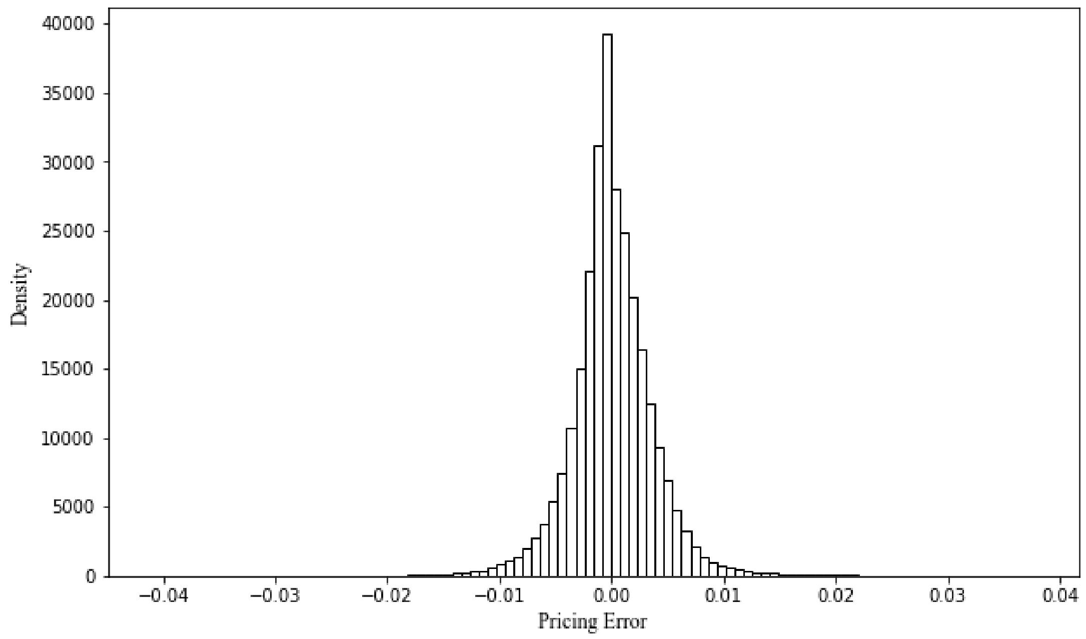

Figure 26. Neural network distribution of pricing error

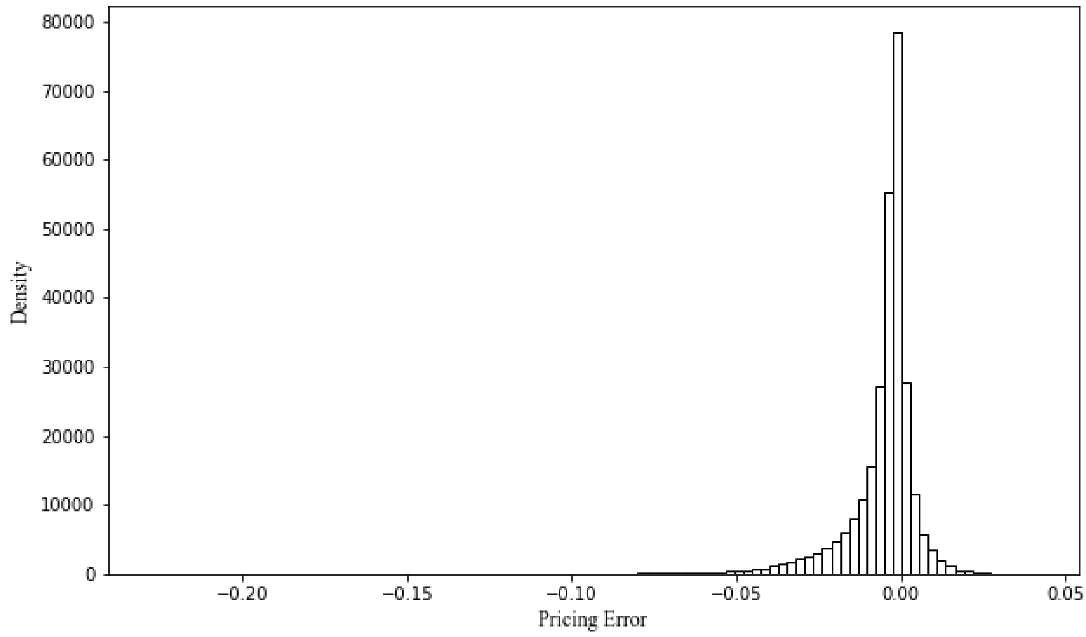

Figure 27. BSM distribution of pricing error

The distribution of the pricing errors for the neural network is clearly centered on zero, with a fairly equal amount of errors to both directions from zero. The pricing errors of the BSM on the other hand are clearly skewed to the negative side of the distribution with most of the observations somewhere between -0.01 and -0.02. This result is in line with previous literature, where it has been found that the BSM has more tendency to underprice than to overprice options (Gencay & Salih 2003).

According to Yao et al. (2000) and Gencay and Salih (2003), the BSM model tends to misprice especially deeply ITM and OTM options, underpricing options that are OTM and overpricing options that are ITM. Table 2 b) shows that when the results are partitioned by moneyness, the BSM model does in fact exhibit the greatest mispricing for OTM options. The result is thus in harmony with previous findings on the mispricing of the BSM. Figures 28 and 29 present the pricing error as a function of monenyness, showing the clear underpricing of OTM options and on the other hand the clear overpricing of ITM options for the BSM model. From Figure 28 it can also be observed that the neural network model does not show similar misbehaviour, with pricing errors centered near zero for nearly all moneyness levels. It seems that for the neural network model, there is a deviation for the extremely ITM options ($S/K > 1.4$), where the model starts to slightly overprice those options. An interesting observation is also that for this test set, the BSM underprices extreme ITM options, while it was found in previous studies (Yao et al. 2000, Gencay & Salih 2003), that ITM options are usually overpriced by the BSM.

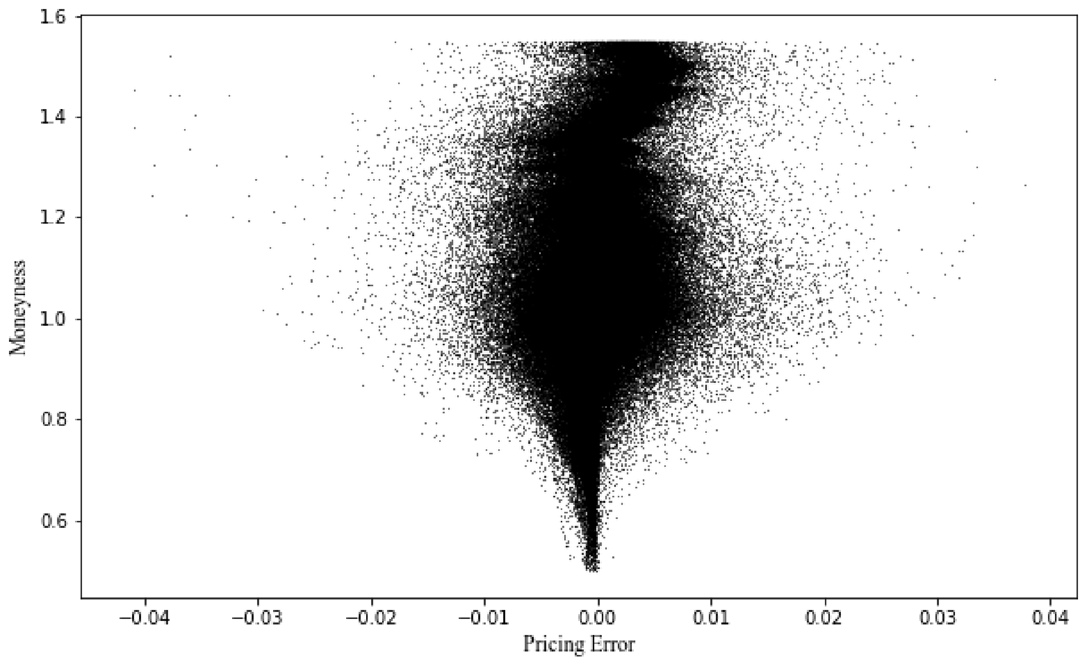

Figure 28. Neural network pricing error as a function of moneyness

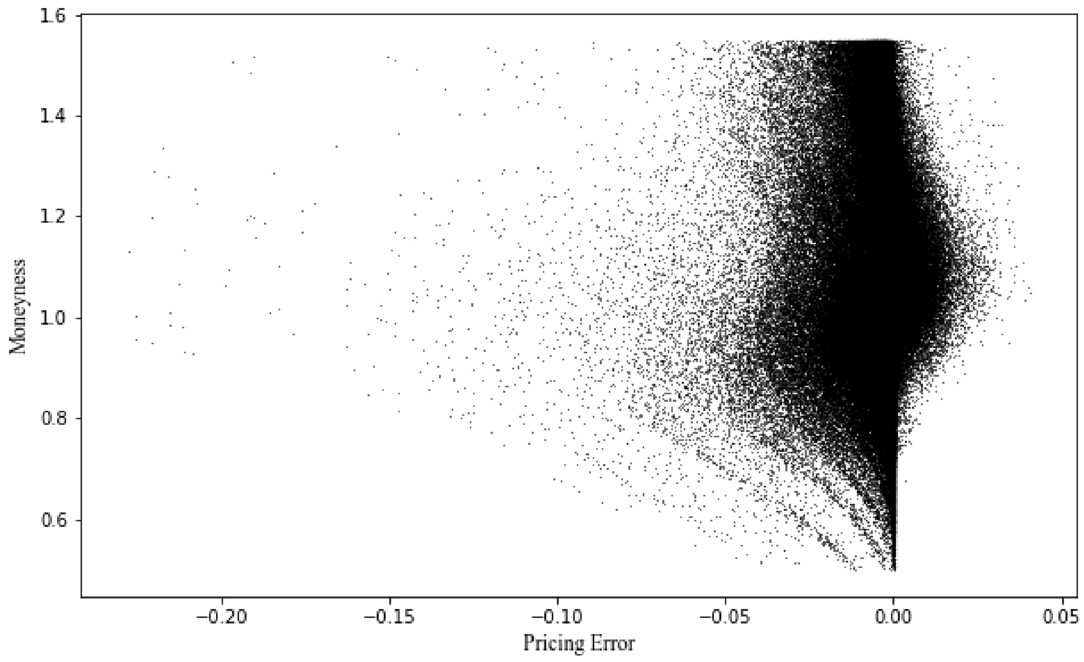

Figure 29. BSM pricing error as a function of moneyness

When the dataset is partitioned by moneyness, the neural network model remains to be superior to the BSM model for all moneyness types. Especially for OTM options, the neural network results in far better pricing with approximately 13 percentage points less pricing error than the BSM model. This result is consistent with previous studies, where the MLP model was found to be better in pricing OTM options than the BSM model (Gencay & Salih 2003, Bennell & Sutcliffe 2004). Unlike in previous studies, the neural network model also seems to have significantly smaller pricing errors for NTM and ITM options, for which Yao et al. (2000) found that there was not much difference in the degree of mispricing between the two models and Bennell and Sutcliffe (2004) found the BSM model to outperform the neural network model.

In Table 2 c), where the data is partitioned by maturity, the neural network model outperforms the BSM model clearly for medium- and long-term options. The neural network also outperforms the BSM for short-term options when measured in terms of RMSE, but only by a small amount. On the other hand, when the error is measured with MAE, the BSM seems to be slightly better than the neural network in terms of pricing. Thus, for short-term options, there does not seem to be a significant difference in the degree of mispricing of the two models. Overall, it seems that there is not much difference in the pricing error of the neural network between short-, medium- and long-term maturities.

The results when the data is partitioned by maturity are different from the previous literature in two significant ways: a surprising feature is that the neural network model does not provide significant performance difference for short-term options, as the neural network model was found to be superior to the BSM model for such options in previous literature (Gencay & Salih 2003, Bennell & Sutcliffe 2004). This may be due to the data pre-processing phase, where options with less than 15 days to maturity were removed from the dataset (Equation 18). These options tend to have relatively high trading volumes and thus it could be that by not following the exclusion criteria introduced by Anders et al. (1996) so strictly, the neural network model could have yielded better results in terms of short term options pricing with more observations to learn from.

On the other hand, the results show that for longer-term options, the neural network clearly outperforms the BSM model with approximately 15 percentage point smaller pricing errors for long-term options. This result is controversial to some of the previous studies, where Yao et al. (2000) and Bennell and Sutcliffe (2004) found that for longer-term options, the BSM model tended to work better than the neural network model. The result may be caused by two separate factors: the first explanation is that there are simply more data points for long-term options in this dataset than there have been in previous studies. The second possible explanation is that due to the shared feature maps of a convolutional network that are not present in MLP's, the neural network model is capable of utilizing the information gained from other maturities and extrapolating this knowledge for longer maturities. Figures 30 and 31 present the pricing errors as a function of time-to-maturity for both of the models.

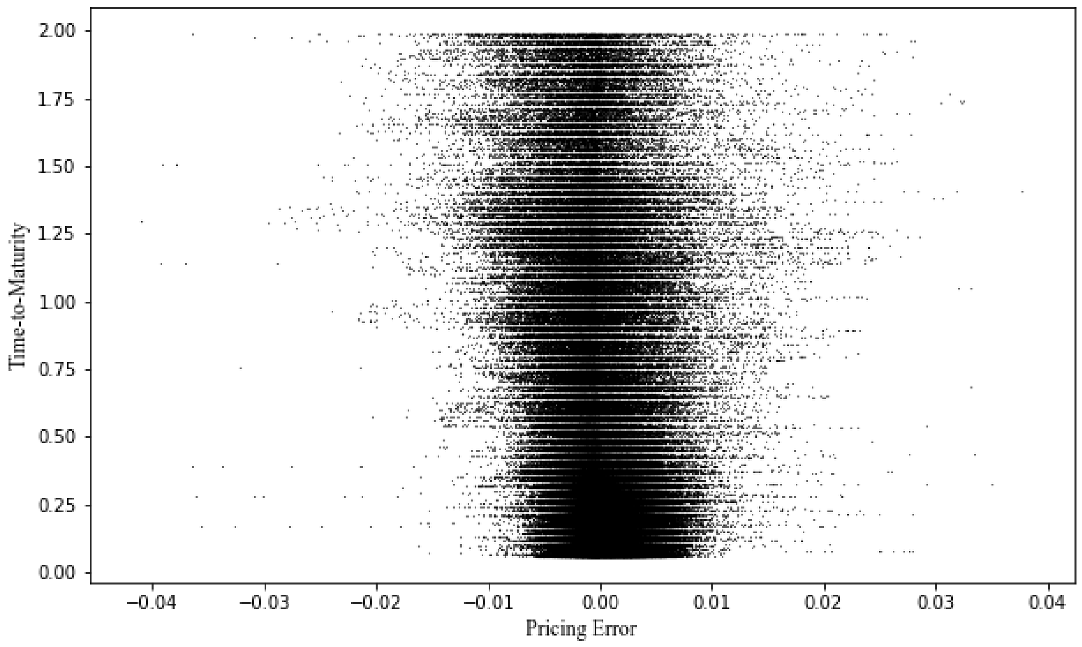

Figure 30. Neural network pricing error as a function of time-to-maturity

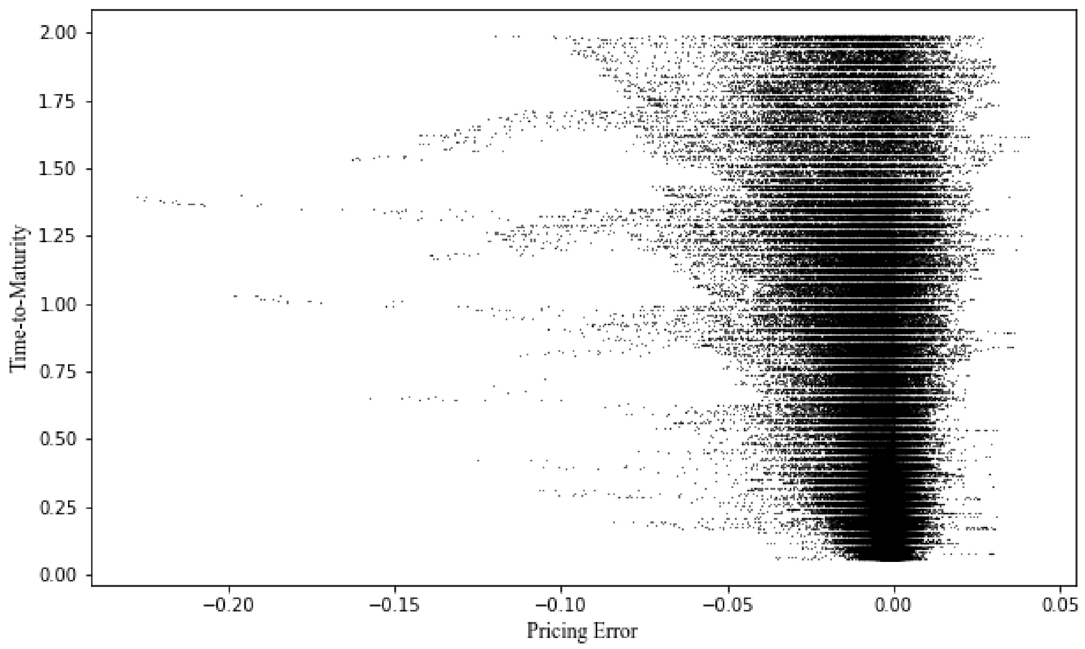

Figure 31. BSM pricing error as a function of time-to-maturity

From the above figures, it can be observed that as stated above, there is not much difference in the pricing error of the neural network for different time-to-maturities. The errors are fairly centered on zero and only slightly deviate from the center for the longest maturities, where the pricing errors start to increase to both directions from zero. The BSM model on the other hand exhibits an increasing pricing error with time-to-maturity, clearly underpricing options as the maturity increases. This result is similar to previous studies, such as the one by Gencay and Salih (2003).

To explore the influence of volatility in pricing performance, Figures 32 and 33 present the pricing errors as a function of the 90-day estimated historical volatility for both of the models. From the figures below, it can again be observed that for the neural network, the pricing errors are fairly centered on zero and do not exhibit large deviations to either direction. Only in the presence of extremely high volatility, the model slightly overprices options. The BSM model on the other hand seems to again systematically underprice options, with an increasing pricing error as the volatility increases. This result is a known bias of the BSM model (Yao et al. 2000, Gencay & Salih 2003). Based on these results, the neural network can clearly learn the influence of different levels of volatility on the price of an option. Thus, the neural network model does seem learn some volatility structure for the underlying, as the pricing errors do not deviate as heavily as they do for the BSM with increasing volatility.

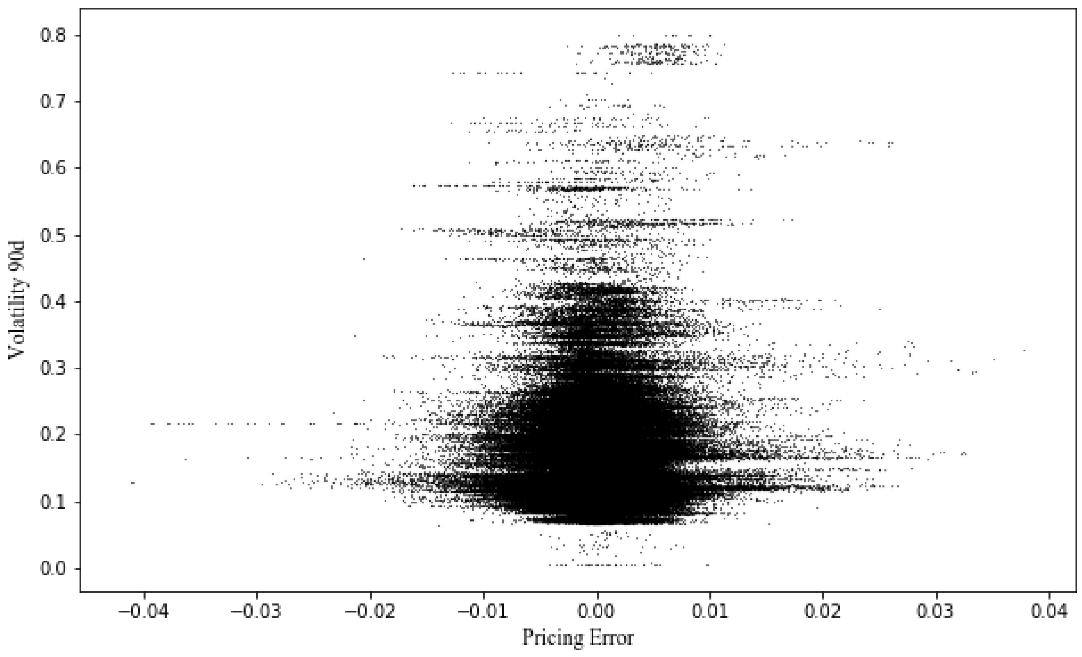

Figure 32. Neural network pricing error as a function of volatility

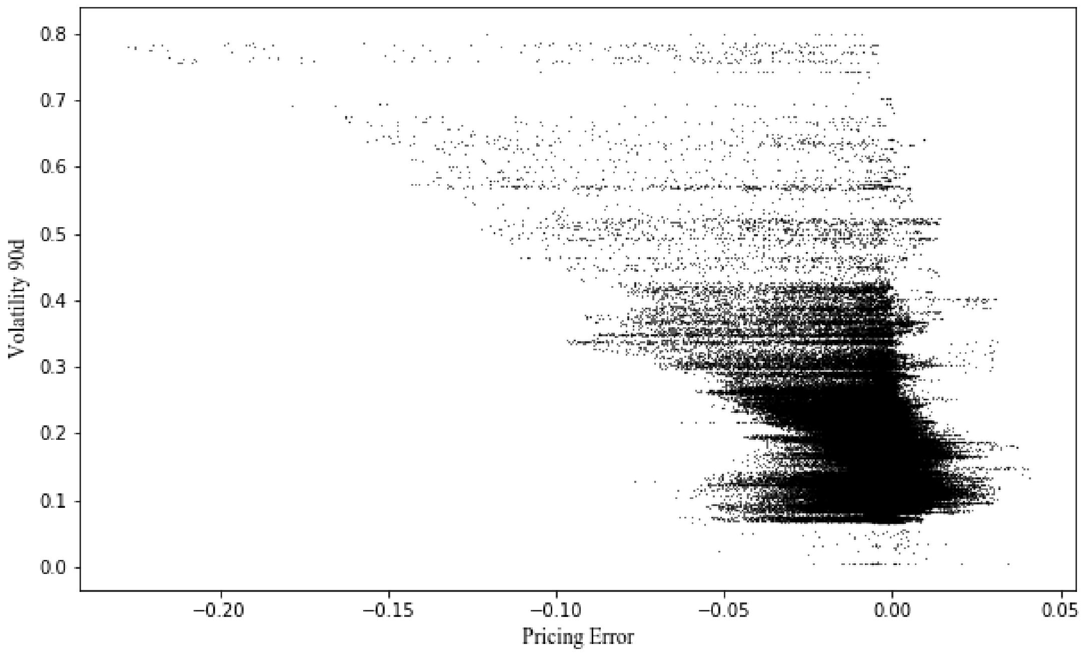

Figure 33. BSM pricing error as a function of volatility

To summarize the pricing results, it can be concluded that the neural network model is superior to the BSM model in pricing performance on all measures, except for short-term options where there was no significant difference between the pricing performances of the two models. This lack of superiority on short-term options may be caused either by the fact that both models exhibited fairly correct pricing for short-term options or due to a significantly smaller amount of observations of short-term options for the neural network to learn from.

The BSM model seems to be bound to underprice options more often than overprice them, resulting in a negatively skewed error distribution in terms of pricing errors, while the neural network's pricing errors were less skewed and tended to be near zero for all moneyness levels, maturities and volatilities with much smaller deviations from zero than those of the BSM model. The average pricing error as a percentage of the strike price (RMSE) was at most around 15 percentage points smaller for the neural network model, a result for the long-term options when the pricing results were partitioned by time-to-maturity. The result was also visualized and it was shown that especially for time-to-maturities and volatilities, the pricing errors of the BSM model tended to increase as the time-to-maturity and volatility increased. For the neural network model, different levels of volatility and time-to-maturity seemed to be insignificant in terms of pricing error, implying that the model has learned some sort of structure of both maturities and volatilities with respect to the option price, enabling it to price options of all sorts more accurately than the BSM model.

5.2 Hedging performance results

Table 3 presents the hedging performance results, which are further divided to three separate tables: the first table contains the results of the entire test set, while the second and the third tables contain results partitioned by moneyness and time-to-maturity. The hedging performance results are given as discounted mean absolute tracking error and standard deviation of the absolute tracking error in columns Mean and Standard Deviation for the neural network model and the BSM model separately, with 95% bootstrapped confidence intervals computed in parentheses below the values.

From the total test set, only options with over 50 days of observations were chosen to ensure a fair period of hedging. This results in a total of 1518 options for which the hedging performance analysis is performed. In Table 3 b), ITM refers to $S/K > 1.02$, OTM to $S/K < 0.98$ and NTM to $S/K \geq 0.98, S/K \leq 1.02$. In Table 3 c), short time-to-maturity refers to $T - t < 1/12$, long time-to-maturity to $T - t > 1/2$ and medium time-to-maturity to $T - t \geq 1/12$ and $T - t \leq 1/2$.

Table 3 a). Hedging performance results - whole test set

Model	No. Observations	Mean	Standard Deviation
Neural network	1518	12078.46	21234.42
CI		(5909.17, 17995.35)	(15489.27, 29877.08)
Black-Scholes		4457.06	7421.34
CI		(2431.09, 6154.86)	(6085.34, 9356.56)

Table 3 b). Hedging performance results - partitioned by moneyness

Model	Moneyness	No. Observations	Mean	Standard Deviation
Neural network	ITM	896	9792.61	17907.19
CI			(3799.25, 15432.04)	(13059.43, 27175.98)
Black-Scholes			4877.34	7899.54
CI			(2197.34, 7425.80)	(6378.77, 10682.15)
Neural network	OTM	505	15852.22	21142.46
CI			(8797.55, 22165.26)	(17019.44, 28362.00)
Black-Scholes			5518.74	7551.51
CI			(3437.26, 7489.28)	(6355.56, 9183.43)
Neural network	NTM	117	12874.03	22789.77
CI			(5969.11, 20251.68)	(16689.93, 32809.63)
Black-Scholes			4081.55	7092.58
CI			(2097.39, 6186.10)	(5488.93, 9962.40)

Table 3 c). Hedging performance results - partitioned by time-to-maturity

Model	Maturity	No. Observations	Mean	Standard Deviation
Neural network	Short	15	14139.94	33535.75
CI			(-4020.73, 26287.89)	(15480.07, 64202.57)
Black-Scholes			4973.01	6556.75
CI			(1680.33, 8106.17)	(4807.17, 10396.71)
Neural network	Medium	388	11742.33	20822.90
CI			(5976.08, 16997.15)	(15695.22, 28523.86)
Black-Scholes			4320.35	7357.65
CI			(2474.00, 6116.71)	(6021.43, 9604.98)
Neural network	Long	1115	12964.71	21755.84
CI			(6296.84, 19486.46)	(15916.75, 31878.77)
Black-Scholes			4829.95	7618.73
CI			(2791.37, 6774.29)	(6008.98, 9955.76)

Table 3 a) presents the hedging performance results for the entire test set, from which it can be observed that that the discounted absolute mean tracking error is significantly smaller for the BSM than for the neural network and that the standard deviation of the absolute tracking error is especially large for the neural network model. This result is in clear conflict with the pricing results presented in the previous subchapter, as it would be expected that with better pricing performance the hedging performance would also be substantially better. The results are statistically significant on at least 5% significance level.

A similar result is observed when the data is partitioned by moneyness, as shown in Table 3 b) and also when the data is partitioned by maturity, shown in Table 3 c). It seems that for this dataset, the BSM is superior to the neural network model in terms of hedging performance measured in the discounted absolute mean tracking error for all moneyness and maturity types with a significantly smaller tracking error. The neural network model is closest to the BSM model for ITM options, where the difference between the two models' tracking errors is approximately 5 000 index points. For NTM and OTM options, the tracking error of the neural network is nearly 10 000 index points larger than for the BSM model. Similar differences are observed when the data is partitioned by maturity, with BSM clearly being the more superior model for all maturities.

The hedging performance results are not consistent with the previous literature, where Hutchinson et al. (1994) and Amilon (2003) found the neural network to be superior to the BSM model measured in terms of the discounted absolute mean tracking error (Hutchinson et al. 1994) and in terms of discounted mean tracking error (Amilon 2003). However, Anders et al. (1996), who did not do a hedging experiment but did explore the hedging parameters visually in their study, found that their neural network was not capable of fully learning the partial derivatives of the options as they are defined in theory (See e.g. Cox & Rubinstein 1985). According to Anders et al. (1996), their neural network model overestimated the delta-parameter for deep ITM options and did not follow the boundaries of a monotonically increasing function, as delta is described to be in theory. To analyse further why the neural network fails to be superior in the hedging experiment even though the network is superior in terms of pricing, the delta surfaces of both the BSM and the neural network are plotted in Figures 34 and 35.

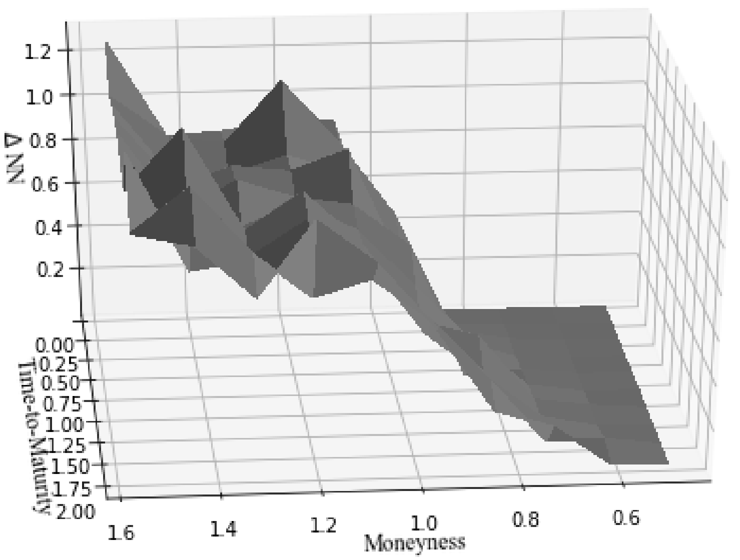

Figure 34. Neural network delta surface as a function of moneyness and time-to-maturity

Clearly, the delta-surface of the neural network does not exactly follow the theoretical boundaries of a monotonically increasing function, as there exists a period after $S/K > 1.2$, where the deltas in fact become smaller prior to increasing again after $S/K > 1.4$. In addition, the bounding condition that $\Delta \in [0,1]$ for a call option is also violated by the neural network. This however should not entirely explain the poor performance of the neural network in terms of the hedging, as the delta approximated by the neural network does seem like a fair estimate in other parts of the figure.

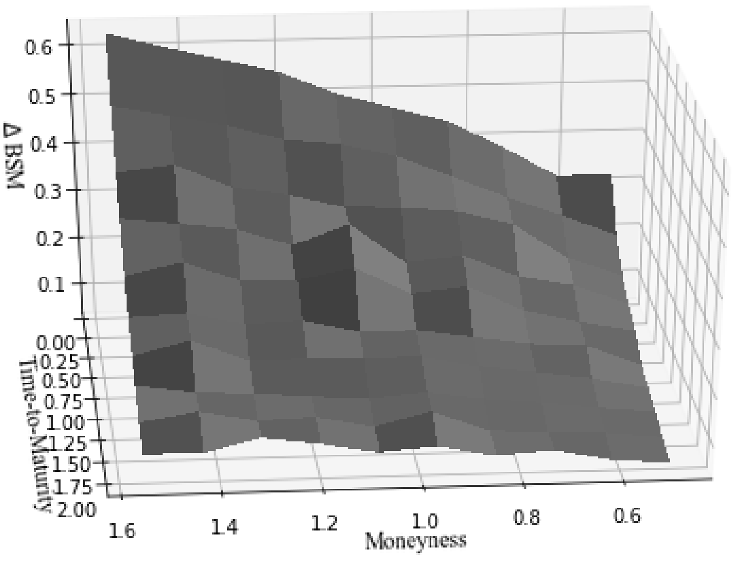

Figure 35. BSM delta surface as a function of moneyness and time-to-maturity

Unlike the delta of the neural network model, the delta computed from the BSM model is clearly now also influenced by the time-to-maturity, with smaller values of delta for higher maturities and vice versa. This is known as delta decay, where the time-value of an option increases for ITM options and decreases for OTM options (See e.g. Cox & Rubinstein 1985, pp. 46-47). Because a majority of the options in the hedging experiment were ITM options, the true delta should involve an increasing time-value, which clearly is present for the BSM model but not for the neural network model. This could be a possible cause for the weaker performance of the neural network in the hedging performance analysis.

Another possible explanation is that there is a problem with the data, or at least a problem with how the data is ordered in the hedging experiment. As the data was divided in a sequential ordering according to date, with data from 1996 to 2011 in the train set and from 2011 to 2014 in the test set, it could be that the dynamics of the underlying asset are so different for the train set (two market crashes and much higher volatility overall) compared to the test set (no major market crashes, lower overall volatility) that the neural network model failed to approximate the delta-term correctly, resulting in much poorer neural network performance in delta-hedging compared to the exceptionally good performance in pricing. Possible fixes to these issues would be to try a different type of ordering for the data, as well as trying different inputs that would imply the neural network model of the time-dependency of the delta-term.

To summarize the hedging results, the neural network seems to be inferior to the BSM model in terms of delta-hedging an option position for the entire dataset as well as when the data is partitioned either by moneyness or maturity. The results for the entire test set are statistically significant on a p-value of at least 5%. This implies that at least in this dataset and with this experiment methodology, the BSM remains to be the better model for hedging purposes.

6 CONCLUSIONS

This paper gives an overview of the research that has been conducted regarding neural networks in option pricing. The paper also analyzes whether a deep neural network model has advantages over the Black-Scholes option pricing model in terms of pricing and hedging European-style call options on the S&P 500 stock index. While the previous literature has focused on shallow MLP-styled neural networks, this paper applies a deeper network structure of convolutional neural networks to the problem of pricing and hedging options. Convolutional neural networks are previously known for their success in image and speech recognition and classification.

The pricing performance in this paper was measured in terms of pricing error with several error functions, such as the Root Mean Squared Error and the Mean Absolute Error. The hedging performance was measured in terms of discounted absolute mean tracking error, following the approaches that were taken in previous studies. In total, there were over 1 million observations in this study that were divided to a train set (75% of observations) and to a test set (25% of observations). The convolutional neural network was trained on the train set and the performance measures were evaluated on the test set of the observations, from which the results were then compared to the results given by the famous Black-Scholes option pricing model.

In terms of pricing performance, the results of this thesis implicate that a convolutional neural network model is superior to the BSM model in pricing European-style call options with nearly 10 percentage points smaller pricing errors for the entire test set. These results are in line with previous studies. However, the pricing results also showed that the convolutional neural network has its perks over a shallow MLP network with superior performance in pricing accuracy also when the data was partitioned by moneyness and maturity. In terms of moneyness, the convolutional neural network model was superior to the BSM model for all moneyness types, while in terms of maturity, the convolutional neural network was superior to the BSM model for mid- and long-term options, remaining equally good in pricing accuracy for short-term options. These results were better than what had been achieved in previous studies with shallow-style MLP networks. The pricing results of the neural network for short-term options may possibly have been weakened by too strict exclusion criteria for the dataset, which were implemented following the approach by Anders et al. (1996).

The two models’ pricing accuracy was also compared with respect to historically estimated volatility. In this comparison, the convolutional neural network model exhibited superiority over the BSM model. This result gives an implication that the neural network model has learned a volatility structure of the underlying asset, which it utilizes when pricing options over different volatility levels, as the average pricing error with different volatility levels remained around zero. In comparison, the BSM model had a tendency to underprice options with an increasing rate of underpricing for increasing volatility.

In terms of hedging performance, the convolutional neural network exhibited inferiority over the BSM model for the entire test set, as well as when the data was partitioned by moneyness or maturity. This result is in conflict with prior literature, where the hedging performance was found to be as good as or better than that of the BSM model. The inferiority is suspected to be a result of incomplete learning of the hedging parameters that were required in the hedging experiment, which further may be caused by very different sort of underlying asset dynamics for the train set and the test set, as the data was ordered in a sequential order by date. It is suggested for further studies to order the data in a non-date-sequential order and experiment input parameters that imply a time-dependency of the delta-term to examine whether these changes would result in a better hedging performance of the neural network.

Overall, based on the pricing and hedging results, it seems that the neural network model is superior to the BSM model in terms of pricing for all types of European-styled call options. On the other hand, the BSM model seems to be superior to the neural network model in terms of dynamic hedging strategies for European-style call options. While both models have their perks, based on the research conducted in this thesis, it can be argued that combining the two models gives the best possible outcome in terms of pricing and hedging options.

Further research is suggested especially towards option position hedging with neural networks and towards approximating the hedging parameters of an option position with a neural network, as these are topics that have not yet been studied as much as pricing. Experimenting with different input variables that could for example improve the approximation of the hedging parameters or imply the network of an underlying volatility structure is encouraged, as well as experimenting with different types of options data or by not following such strict exclusion criteria as was followed in this thesis. Additionally, research in the future might include different types of neural networks such as Recurrent Neural Networks (RNNs) into the comparison. The neural network models should also be compared to other types of parametric formulae, such as the ones by Merton (1976) and Heston (1993). Similarly, neural networks could be researched in areas where there are no analytical solutions to compute the price of an option (e.g. Asian, Cliquet, Bermudan, Rainbow and other exotic options).

REFERENCES

A., C., M., H., M., M., G., B. A. & Y., L. (2014). The loss surfaces of multilayer networks. ArXiv e-prints.

Abadi, M., Barham, P., Chen, J., Chen, Z., Davis, A., Dean, J., et al. (2016). TensorFlow: A system for large-scale machine learning. Osdi.

Aggarwal, C. (2014). Data classification: Algorithms and applications. CRC Press, 209.

Amilon, H. (2003). A neural network versus Black–Scholes: A comparison of pricing and hedging performances. Journal of Forecasting 22(4), 317-335.

Anders, U., Korn, O. & Schmitt, C. (1996). Improving the pricing of options: A neural network approach. Journal of Forecasting 17(5), 369-388.

Bates, D. S. (1996). Jumps and stochastic volatility: Exchange rate processes implicit in deutsche mark options. The Review of Financial Studies 9(1), 69-107.

Bennell, J. & Sutcliffe, C. (2004). Black–Scholes versus artificial neural networks in pricing FTSE 100 options. Intelligent Systems in Accounting, Finance and Management 12(4), 243-260.

Bishop, C. M. (2006). Pattern recognition and machine learning. Springer, 192-193.

Black, F. & Scholes, M. (1973). The pricing of options and corporate liabilities. Journal of Political Economy 81(3), 637-654.

Bollerslev, T. (1986). Generalized autoregressive conditional heteroskedasticity.

Bowles, M. (2015). Machine learning in python: Essential techniques for predictive analysis. Wiley.

Breiman, L. (1996). Bagging predictors. Machine Learning 24(2), 123-140.

Cox, J. C. & Rubinstein, M. (1985). Options markets. Prentice-Hall, Inc.

Cox, J. C. & Ross, S. A. (1976). The valuation of options for alternative stochastic processes.

Culkin, R. & Das, S. (2017). Machine learning in finance: The case of deep learning for option pricing. Santa Clara University.

Cybenko, G. (1989). Approximation by superpositions of a sigmoidal function. Mathematics of Control, Signals and Systems 2(4), 303-314.

D.P., K. & J., B. (2014). Adam: A method for stochastic optimization. ArXiv e-prints.

Diebold, F. X. & Mariano, R. S. (1995). Comparing predictive accuracy. Journal of Business & Economic Statistics 13(3), 253-263.

Duan, J. (1995). The garch option pricing model. Mathematical Finance 5(1), 13-32.

Duchi, J., Hazan, E. & Singer, Y. (2011). Adaptive subgradient methods for online learning and stochastic optimization. Journal of Machine Learning Research 12(Jul), 2121-2159.

Efron, B. (1979). Bootstrap methods: Another look at the jackknife. The Annals of Statistics 7(1), 1-26.

Engle, R. F. (1982). Autoregressive conditional heteroscedasticity with estimates of the variance of united kingdom inflation. Econometrica 50(4), 987-1007.

Engle, R. F. (1984). Chapter 13 wald, likelihood ratio, and lagrange multiplier tests in econometrics. Elsevier, 775.

Fitzenberger, B. (1998). The moving blocks bootstrap and robust inference for linear least squares and quantile regressions.

Foresee, F. D., & Hagan, M. T. (1997). Gauss-newton approximation to bayesian learning. Proceedings of International Conference on Neural Networks (ICNN'97), , 3 1930-1935 o.3.

Fortune, P. (1996). Anomalies in option pricing: The black-scholes model revisited. 17-40.

Fukushima, K. (1980). Neocognitron: A self-organizing neural network model for a mechanism of pattern recognition unaffected by shift in position. Biological cybernetics 36(4), 193-202.

G.E., H., N., S., A., K., I., S. & R.R., S. (2012). Improving neural networks by preventing co-adaptation of feature detectors. ArXiv e-prints.

Garcia, R. & Gençay, R. (2000). Pricing and hedging derivative securities with neural networks and a homogeneity hint. Journal of Econometrics 94(1), 93-115.

Gençay, R. & Salih, A. (2003). Degree of mispricing with the black-scholes model and nonparametric cures. Annals of Economics and Finance 73-101.

Girosi, F., Jones, M. & Poggio, T. (1995). Regularization theory and neural networks architectures. Neural computation 7(2), 219-269.

Greene, W. H. (2012). Econometric analysis, 7th edition. Pearson, 155-161.

Hahnloser, R. H. R., Sarpenshkar, R., Mahowald, M. A., Douglas, R. J. & Seung, H. S. (2000). Digital selection and analogue amplification coexist in a cortex-inspired silicon circuit. Nature 405(6789), 947.

Hebb, D. (1949). The organization of behavior: a neuropsychological theory. John Wiley & Sons, Inc.

Heston, S. L. (1993). A closed-form solution for options with stochastic volatility with applications to bond and currency options. The Review of Financial Studies 6(2), 327-343.

Hilpisch, Y. (2015). Python for finance. O'Reilly Media, Inc.

Hinton, G. E. & Salakhutdinov, R. R. (2006). Reducing the dimensionality of data with neural networks. Science 313(5786), 504-507.

Hinton, G., Srivastava, N. & Swersky, K. Overview of mini-batch gradient descent. Lecture slides, Coursera.

Hochreiter, S. (1998). The vanishing gradient problem during learning recurrent neural nets and problem solutions. Int.J.Uncertain.Fuzziness Knowl.-Based Syst. 6(2), 107-116.

Hornik, K. (1991). Approximation capabilities of multilayer feedforward networks.

Hornik, K., Stinchcombe, M. & White, H. (1989). Multilayer feedforward networks are universal approximators.

Huber, P. J. (1985). Projection pursuit. Ann.Statist. 13(2), 435-475.

Hull, J. C. Options, futures, and other derivatives. (8th edition). Pearson, 219-220.

HULL, J. & WHITE, A. (1987). The pricing of options on assets with stochastic volatilities. The Journal of Finance 42(2), 281-300.

Hutchinson, J. M., Lo, A. W. & Poggio, T. (1994). A nonparametric approach to pricing and hedging derivative securities via learning networks. The Journal of Finance 49(3), 851-889.

Ivakhnenko, A. G. (1970). Heuristic self-organization in problems of engineering cybernetics.

Jarrett, K., Kavukcuoglu, K., Ranzato, M., & LeCun, Y. (2009). What is the best multi-stage architecture for object recognition? 2009 IEEE 12th International Conference on Computer Vision, 2146-2153.

Jones, M. C. & Sibson, R. (1987). What is projection pursuit? Journal of the Royal Statistical Society. Series A (General) 150(1), 1-37.

K., S. & A., Z. (2014). Very deep convolutional networks for large-scale image recognition. ArXiv e-prints.

Kartalopoulos, S. V. (1995). Understanding neural networks and fuzzy logic: Basic concepts and applications. (Sep 1995 edition) Wiley-IEEE Press.

Kirkpatrick, S., Gelatt, C. D. & Vecchi, M. P. (1983). Optimization by simulated annealing. Science 220 (4598), 671-680.

Krizhevsky, A., Sutskever, I. & Hinton, G. E. (2017). ImageNet classification with deep convolutional neural networks. Communications of the ACM 60(6), 84-90.

Kunsch, H. R. (1989). The jackknife and the bootstrap for general stationary observations. The Annals of Statistics 17(3), 1217-1241.

Lay, D. C. & Strang, G. (eds.). (2006). Linear algebra and its applications: Update by David C lay (2006-05-04). Pearson Education.

Le Cun, Y. (1986). Learning process in an asymmetric threshold network. Disordered Systems and Biological Organization, 233-240.

Le Cun, Y. (1988). A theoretical framework for back-propagation.

Lecun, Y. (1985). Une procedure d'apprentissage pour reseau a seuil asymmetrique (A learning scheme for asymmetric threshold networks). Proceedings of Cognitiva 85, Paris, France, 599-604.

LeCun, Y., Bengio, Y. & Hinton, G. (2015). Deep learning. Nature 521 436.

Li, F. Convolutional neural networks for visual recognition. Lecture slides, Stanford university.

Lo, A. W. (2004). The adaptive markets hypothesis. The Journal of Portfolio Management 30(5), 15-29.

M.D, Z. & R., F. (2013). Visualizing and understanding convolutional networks. ArXiv e-prints.

Maas, A. L., Hannun, A. Y. & Ng, A. Y. Rectifier nonlinearities improve neural network acoustic models.

MacKay, D. J. C. (1992). Bayesian interpolation. Neural computation 4(3), 415-447.

McCulloch, W. S. & Pitts, W. (1943). A logical calculus of the ideas immanent in nervous activity. The Bulletin of mathematical biophysics 5(4), 115-133.

Merton, R. C. (1973). Theory of rational option pricing. The Bell Journal of Economics and Management Science 4(1), 141-183.

Merton, R. C. (1976). Option pricing when underlying stock returns are discontinuous.

Merton, R. C. (ed.). (1990). Continuous-time finance. Blackwell. 202-203.

N., M., A.W., R., M.N., R., A.H., T. & E., T. (1953). Equation of state calculations by fast computing machines. Jcp 21 1087-1092.

Nair, V., & Hinton, G. E. (2010). Rectified linear units improve restricted boltzmann machines. Icml.

Nesterov, Y. (2004). Introductory lectures on convex optimization. Springer.

Poggio, T. & Girosi, F. (1990). Networks for approximation and learning. Proceedings of the IEEE 78(9), 1481-1497.

Powell, M. J. D. (1987). Algorithms for approximation. In: Mason, J. C. & Cox, M. G. (eds.). . New York, NY, USA: Clarendon Press, 143-167.

Qian, N. (1999). On the momentum term in gradient descent learning algorithms.

Rosenblatt, F. (1958). The perceptron: A probabilistic model for information storage and organization in the brain. Psychological review 65 6 386-408.

Rubinstein, M. (1983). Displaced diffusion option pricing. The Journal of Finance 38(1), 213-217.

Rumelhart, D. E., Hinton, G. E. & Williams, R. J. (1986). Learning representations by back-propagating errors. Nature 323(6088), 533-536.

S., B. (2014). Convex optimization: Algorithms and complexity. ArXiv e-prints.

S., I. & C., S. (2015). Batch normalization: Accelerating deep network training by reducing internal covariate shift. ArXiv e-prints.

S., R. (2016). An overview of gradient descent optimization algorithms. ArXiv e-prints.

Schmidhuber, J. (2015). Deep learning in neural networks: An overview.

Smolensky, P. (1986). Information processing in dynamical systems: Foundations of harmony theory. Information processing in dynamical systems: foundations of harmony theory 194-281.

Černý, V. (1985). Thermodynamical approach to the traveling salesman problem: An efficient simulation algorithm. Journal of Optimization Theory and Applications 45(1), 41-51.

Vikhar, P. A. (2016). Evolutionary algorithms: A critical review and its future prospects. 2016 International Conference on Global Trends in Signal Processing, Information Computing and Communication (ICGTSPICC), 261-265.

Weisstein, E. W. (2002). Sigmoid function. From MathWorld--A wolfram web resource. http://mathworld.wolfram.com/SigmoidFunction.html.

Werbos, P. J. (1974). Beyond regression: New tools for prediction and analysis in the behavioral sciences. Harvard University.

White, H. (1989). Some asymptotic results for learning in single hidden-layer feedforward network models. Journal of the American Statistical Association 84(408), 1003-1013.

Y., D., R., P., C., G., K., C., S., G. & Y., B. (2014). Identifying and attacking the saddle point problem in high-dimensional non-convex optimization. ArXiv e-prints.

Yao, J., Li, Y. & Tan, C. L. (2000). Option price forecasting using neural networks.

Yu, L., HUANG, W., Lai, K. K., NAKAMORI, Y. & Wang, S. (2007). Neural networks in finance and economics forecasting. 06 113-140.