Project Thesis

Julie Johanne Uv

Institutt for matematiske fag — NTNU 2018

Contents

Contents

Chapter 1

Introduction

Chapter 2

Financial theory

2.1 Effiecient Market Hypopthesis

The financial market is a common term for markets that trade different financial instruments such as stocks, bonds, currencies and different financial derivatives such as futures, forwards, options and swaps. The Efficient Market Hypothesis is stated in several ways, but essentially amounts to the properties that

- the current market price reflects history up until the present, but does not hold any further information about the future
- that the market moves immediately with the arrival of new information.

From the former it follows that stock prices has the Markov Property, i.e. the future price depends only upon the present price.

$$P(X_{t+1} = x_{t+1}|X_t = x_t, \ldots, X_0 = x_0) = P(X_{t+1} = x_{t+1}|X_t = x_t) \tag{2.1}$$

As information essentially moves randomly, it follows from the latter that it is reasonable to model the prices randomly as well. If the market really is efficient, then assets must always be traded at their fair value(reference) and one can never outperform the market. If assets were over- or undervalued, the prices would immediately adjust. The Efficient Market Hypothesis does not mean that one can never profit from the market, but that one must do so at a risk.

2.2 Arbitrage

Arbitrage is a risk-free instantaneous profit which can be made by a exploiting price differences in the same or similar financial instruments or assets by simultaneous purchase and sale. Arbitrage opportunities will occur as the market is not perfectly efficient, but prices will adjust quite fast(?). Hence it is reasonable to assume market efficiency and thus no arbitrage in financial pricing models.

2.3 Assets

A financial asset is an asset of which value comes from a so called contractual agreement - an agreement that cash flows will be paid to the purchaser at times specified by the contract. Examples of financial assets are cash, equities, indices and commodities. As argued, a financial asset (traded on the market) may be modeled randomly and it is common to value it's price movement by a stochastic differential equation of the following form.

$$dS(t) = u(S,t)dt + w(S,t)dW(t) \tag{2.2}$$

Here $S(t)$ is the value of the asset price at time t, u and w are arbitrary functions of asset price and time. dW is a normally distributed random variable (appendix on r.v?) with the following properties

$$E[dW(t)] = 0 \tag{2.3}$$
$$Var[dW(t)] = dt. \tag{2.4}$$

$dW(t)$ follows what is called a Wiener process(or a Brownian motion)(appendix?) and represents the randomness of the asset (without it we would just have a deterministic differential equation). (u is the drift term and w is the diffusion term.)

Stocks

By issuing stocks to investors, a company can raise capital. If the company increases it's revenue, the value of the stocks increases. In addition the company can decide to pay out a part of the revenue as dividend per share to its shareholders.

When one values stocks, it is more reasonable to look at the relative change in price rather then the actual level of the price, i.e. $dS(t)/S(t)$ rather then $S(t)$. The contributions to the relative change, also known as the return, consists of a deterministic term μdt - a drift term containing the average growth of the asset price, μ, and a non-deterministic term σdW containing the volatility measuring the standard deviation of the returns, σ.

$$\frac{dS(t)}{S(t)} = \mu dt + +\sigma dW^o(t). \tag{2.5}$$

$dW^o(t)$ is a standard Brownian motion under the "real-world" measurement P_o (What this entails we will get back to in **??**. (Empirically, this also seems reasonable as price tend to increase expnentially.)

μ and σ can both be functions of S and t. The most basic model is taking μ and σ to be constants. A solution to (**??**) is (Ito's Lemma appendix?)

$$S(t) = S(0)e^{(\mu - \frac{1}{2}\sigma^2)t + \sigma W(t)}. \tag{2.6}$$

2.4 Derivatives

Derivatives are instruments where the value is derived from or rely upon an underlying asset (or a group of assets) such as financial assets or interest rates. (It is a contract between two or more parties.) Examples of derivatives are options, futures, forwards and swaps. This thesis will (focus on) valuing options.

And option is a contract between two parties that gives the holder of the option the right but not the obligation to buy or sell the underlying asset at a specified time and at a specified price called the exercise (or strike) price, in the future. As stated the purchaser of the option is the holder of the option, while the seller is the writer. An option in which the holder has the right to *buy* the underlying is called a *call* option, and an option in which the holder has the right to *sell* the underlying is called a *put* option. There are (also) several types of call and put options. American options can be exercised at any time up until expiry, while the one which will be examined(?) further in this thesis is the European option which can only be exercised at expiry. The payoff function for a If C is the price of a call option, E is exercise price and S is the underlying stock price, the European call option at expiry T is

$$C(S,T) = max(S(T) - E, 0). \tag{2.7}$$

Similarly, if P is the price of a put option, the payoff function for a European put option is

$$P(S,T) = max(E - S(T), 0). \tag{2.8}$$

The value of an option is dependent on the underlying asset price and the time to expiry (for obvious reasons) which both changes during the life of the option. In addition, the value of the option will be affected by the exercise price and the interest rate. The higher the exercise price of a call option, the higher the value of the option. The interest rate affects the price through time value as the payoff will be received in the future. Lastly, the volatility will have an effect on the option value (measures the fluctuation, annualized standard deviation of the returns). (How?)

2.5 Risk-Neutral Pricing

Risk can be interpreted(?) as the variance of the return - how much the price deviates from the expected price(?). There are two types of risk, namely specific and non-specific risk. The former is risk associated with the specific asset/company and can be minimized by diversification(building a portfolio with negatively correlated assets). The latter is risk associated with factors influencing the whole market and can be reduced by hedging(taking opposite positions in similar derivatives which gives less return and less non-specific risk). Looking at the risk-neutral case means that investor's risk preferences are irrelevant as all (non-specific) risk can be hedged away. There is no profit to be made above the risk free return. This relies on the existence of a self-financing strategy - in a

complete market, an underlying asset can be perfectly replicated, hence the price (of the derivative) is just the lowest initial investment needed to replicate the asset. (?) (Practically this means that we replace the rate of growth by the risk free rate for all random walks which means that even though investors may disagree on the rate of growth, they will value options the same.)
The money market account, $\beta(t)$, may be described as an asset such as (**??**) with drift equal to the risk-free rate and volatility equal to zero.

$$\frac{d\beta(t)}{\beta(t)} = rdt$$

where r is the risk-free rate, which has solution $\beta(t) = e^{rt}$ if $\beta(0) = 1$. $\beta(t)$ is attainable(appendix) and $\beta(t)/Z(t)$ is a positive martingale(appendix) for some stochastic discount factor(appendix?), $Z(t)$, of the market (why? positive since β and Z are positive and initial value is 1). Any such martingale defines a change in probability measure. I. e. for a fixed interval $[0, T]$, $\beta(t)/Z(t)$ defines the new probability measure P_beta such that

$$\left(\frac{dP_\beta}{dP_o}\right)_t = \frac{\beta(t)}{Z(t)}, \qquad 0 \leq t \leq T. \tag{2.9}$$

P_β is the risk-neutral measure and is equivalent (martingale) measure(appendix) to P_o, the "real-world" measure. In practice, this means we replace the standard Browninan motion $dW^o(t)$ under P_o by a Brownian motion $dW(t)$ under P_β such that

$$dW(t) = dW^o(t) + \nu(t)dt$$

with ν satisfying $\mu = r + \sigma\nu$.

2.6 Volatility

Volatility is the standard deviations of the logarithmic returns, so it might be viewed as a measurement of how random the asset price is(?). There are several ways of modelling the volatility.

- As a stochastic differential equation. This is a very reasonable model as the volatility seems to have a random component just as our asset price model. Then one would have two stochastic differential equations

$$dS(t) = \mu S(t)dt + \sqrt{\nu(t)}S(t)dW_1(t) \tag{2.10}$$
$$d\nu(t) = \alpha(\nu, t)dt + \beta(\nu, t)dW_2(t) \tag{2.11}$$

  where $dW_1(t)$ and $dW_2(t)$ are two Wiener processes(which may be correlated?).

- Historical volatility uses historical values to calculate some parameter estimation for the volatility based on statistics. (This may be the maximum likelihood e.g.). (In this paper $\hat{\sigma}$ has been used.) A question that arises for the use of the historical volatility is how much historical data one should include in the calculations.

- Implied volatility uses the quoted prices for options today and exploiting the idea that the market "knows" the volatility. When one has today's option price, asset price, life span of option, risk free rate and strike price one can solve the Black Scholes formula (which will be discussed in chapter 4, source) backwards to find the implied volatility.

Chapter 3

Monte Carlo Simulation

Monte Carlo simulation is a way of randomly simulating a stochastic process such as that of eqref 2.5. More abouot this. (Principles of Derivative Pricing?)
Monte Carlo simulation is useful(other word) when

- if the price dynamics are such that numerically solving the PDE (describing it) is difficult or does not exist

- the price depend on the paths of the underlying asset.

- the number of underlying assets are large(maybe even just larger than three)

For a general stochastic differential equation as that given by (**??**), the Euler discretization scheme ([**?**] p. 340) for a time grid $0 = t_0 < t_1 < \ldots < t_m$, where $\hat{X}$ a time-discretized approximation to X, $\hat{X}(0) = X(0)$ and for $i = 0, \ldots, m-1$, is given by

$$\hat{X}(t_{i+1}) = \hat{X}(t_i) + u(\hat{X}(t_i))[t_{i+1} - t_i] + w(\hat{X}(t_i))\sqrt{t_{i+1} - t_i}Z_{i+1} \tag{3.1}$$

Z_i independent one-dimensional standard normal random numbers(/vectors). For the random walk of stock returns given in (**??**) this is then

$$\hat{X}(t_{i+1}) = \hat{X}(t_i) + \mu\hat{X}(t_i)[t_{i+1} - t_i] + \sigma\hat{X}(t_i)\sqrt{t_{i+1} - t_i}Z_{i+1}. \tag{3.2}$$

From (**??**) one can obtain that

$$\ln\left(\frac{S(t)}{S(0)}\right) \sim N((\mu - \frac{1}{2}\sigma^2)t, \sqrt{t}). \tag{3.3}$$

Using this the parameters can be estimated using the historical logarithmic returns by the following.

$$r_i = \log\left(\frac{S_{i+1}}{S_i}\right) \tag{3.4}$$

$$m = \frac{1}{ndt}\sum_{i=0}^{n-1} r_i \tag{3.5}$$

$$\hat{\sigma} = \frac{1}{(n-1)dt}\sum_{i=0}^{n-1}(r_i - m)^2 \tag{3.6}$$

$$\hat{\mu} = m + \frac{1}{2}\hat{\sigma}^2 \tag{3.7}$$

In figure something the random walk of (**??**) something was implemented for a company stock and N paths were simulated. The figure also shows the true stock price.

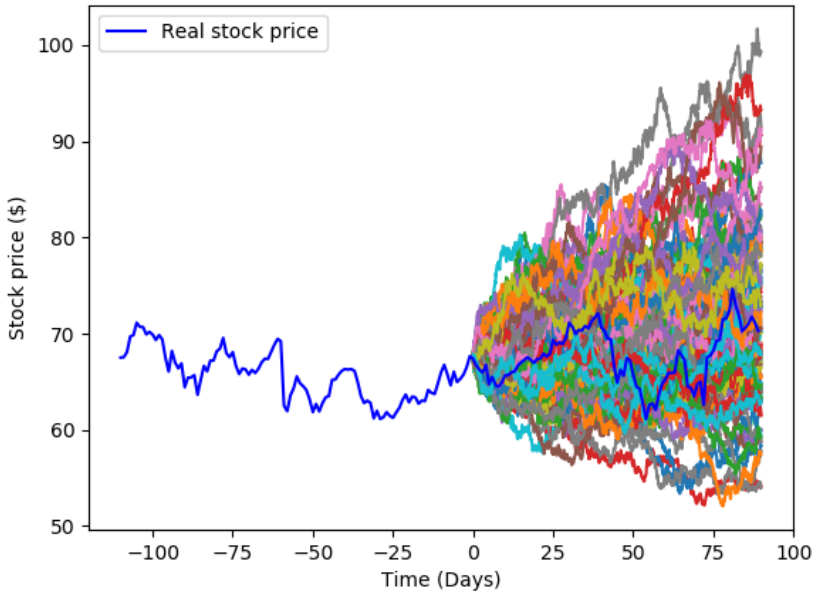


Figure 3.1: Simulation of 100 random walks over 90 days with time step = 0.1.

The Euler scheme described in (**??**) has strong convergence of order 1/2. I.e.

$$E\left[||X(0) - \hat{X}(0)||^2\right] \leq \kappa\sqrt{h}$$

and

$$||u(x,s) - u(x,t)|| + ||w(x,s) - w(x,t)|| \leq \kappa(1 + ||x||)\sqrt{|t-s|}$$

for some constant κ. (proven in Kloeden and Platen, p. 342-344?)
Valuing eqref(stock price SDE) under the risk-neutral measure P_β we would obtain the same solution as that of the Black-Scholes formula derived in chapter **??**.

Chapter 4

The Black-Scholes Model

4.1 Assumptions

(Black and Scholes(1973) reference) The Black-Scholes formula is a mathematical formula used to calculate a theoretical value for the option price at any point up until expiry using the input parameters underlying asset price, strike price, time until expiry, volatility of underlying and current risk free rate. (However, it can also be used for solving for expected volatility, interest rates, expected dividends.) When deriving the Black-Scholes formula, the following assumptions are made:

- The underlying asset follows a lognormal random walk. (This means)/I.e. the logarithmic returns are normally distributed.
- The risk free rate, r and volatility σ are known (and constant/functions of time).(This is not a very realistic assumption as r and σ seem to be stochastic reference. There are methods that also model them as stochastic differential equations.)
- There are no transaction costs for trading the underlying assets. (Also not a realistic assumption. Also exists models that take this into account.)
- Delta hedging is possible - underlying can be traded continuously and one can buy and sell any number(not necessarily an integer) of the underlying
- There are no arbitrage opportunities.
- There are no dividends paid out by underlying asset during the life of the option. (This can be adjusted for if one knows when and how much dividend will be paid out in advance.)

4.2 The Black-Scholes Partial Differential Equation

It is reasonable to assume the option value which will be denoted by V is a function of time, t, (even more specific, time to expiry, $T-t$) and the underlying asset S. Assuming

that S follows a lognormal random walk (eqref), then by Ito's Lemma(see Appendix)

$$dV = \frac{\partial V}{\partial S}dS + \frac{\partial V}{\partial t}dt + \frac{1}{2}\sigma^2 S^2 \frac{\partial^2 V}{\partial S^2}dt. \tag{4.1}$$

(The random walk of V) Now, consider the following portfolio, Π, consisting of such an option and an arbitrary quantity $-\Delta$ of the underlying asset.

$$\Pi = V(S,t) - \Delta S \tag{4.2}$$

(Δ is also called the delta.) Next we want to consider a infinitesimal change of the portfolio for some time interval $(t, t+dt)$. By (**??**) and eqref(lognormal random walk) and eqref(portfolio)

$$d\Pi = \Big(\mu S\frac{\partial V}{\partial S} + \frac{1}{2}\sigma^2 S^2 \frac{\partial^2 V}{\partial S^2} + \frac{\partial V}{\partial t} - \mu\Delta S\Big)dt + \sigma S\Big(\frac{\partial V}{\partial S} - \Delta\Big)dW. \tag{4.3}$$

By choosing $\Delta = \frac{\partial V}{\partial S}$ we eliminate the random term and thereby the risk(?). This is called delta hedging and is a dynamic hedging strategy. Our remaining expression for $d\Pi$ is

$$d\Pi = \Big(\frac{\partial V}{\partial t} + \frac{1}{2}\sigma^2 S^2 \frac{\partial^2 V}{\partial S^2}\Big)dt \tag{4.4}$$

and thus completely deterministic. As mentioned, this is a risk less portfolio and the change in value of the portfolio should therefore correspond to the amount we would get on return from a risk-free account (by the assumption of no arbitrage). I.e.

$$d\Pi = r\Pi dt \tag{4.5}$$

$$\Big(\frac{\partial V}{\partial t} + \frac{1}{2}\sigma^2 S^2 + \frac{\partial^2 V}{\partial S^2}\Big)dt = r\Big(V - \Delta S\Big)dt \tag{4.6}$$

From the latter we obtain the Black-Scholes partial differential equation

$$\frac{\partial V}{\partial t} + \frac{1}{2}\sigma^2 S^2 \frac{\partial^2 V}{\partial S^2}\ rS\frac{\partial V}{\partial S} - rV = 0. \tag{4.7}$$

for $S > 0$ and $0 \leq t \leq T$. (T being time until expiry.)

4.3 The Black Scholes Formula for European Options

For European options, by applying appropriate boundary and final conditions we can obtain a closed form solution for the option price. Using the put-call parity, one can obtain the solution for a put option from an existing solution of a call option and the other way around. Hence, consider first a European call option.
If, at expiry, $E > S(T)$ it would be reasonable to go through with the option and one would make a profit of $E - S(T)$. If $E \leq S$, the option's value would be zero. Hence, the final condition should be

$$V(S,T) = \max(E - S(T), 0), \tag{4.8}$$

Table 4.1: Some caption

Parameter	Value
σ	0.2
r	0.02
E	30
T	90 days

also called the payoff function.
Now, consider the case when $S = 0$. Then (**??**) would reduce to

$$\frac{\partial V}{\partial t} - rV = 0. \tag{4.9}$$

Solving the reduced equation using the final condition and realizing that the put option is worthless as $S \to \infty$, we obtain the boundary conditions

$$V(0,t) = Ee^{-r(T-t)} \tag{4.10}$$
$$V(S,t) = 0 \text{ as S } \to \infty \tag{4.11}$$

This leads to the following closed form solution for European put options.

$$V(S,t) = Ee^{-r(T-t)}N(-d_2) - SN(-d_1) \tag{4.12}$$

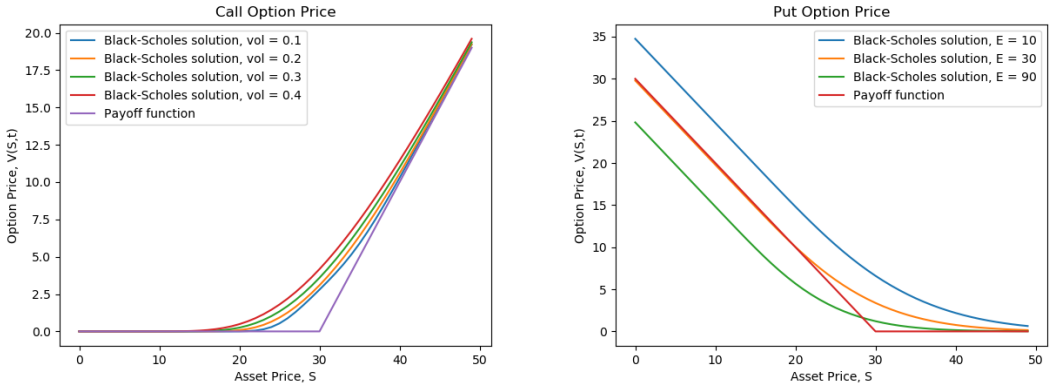
$N(\cdot)$ is the cumulative normal distribution and

$$d_1 = \frac{\log(S/E) + (r + \frac{1}{2}\sigma^2)(T-t)}{\sigma\sqrt{T-t}} \tag{4.13}$$
$$d_2 = \frac{\log(S/E) + (r - \frac{1}{2}\sigma^2)(T-t)}{\sigma\sqrt{T-t}} \tag{4.14}$$

Next, we can as mentioned use the put-call parity, combining underlying asset and an identical put and a call option in a portfolio in such a way that we always expect to get a return E, and then solve for the call option to obtain the closed form solution for a European call option.

$$V(S,t) = SN(d_1) - Ee^{-r(T-t)}N(d_2) \tag{4.15}$$

In fig.. the values of **??** were set for the parameters in eqref BS and solved as a function of the underlying asset price S ($\in [0, 50]$) in addition to the payoff function eqref.



(a) Call option. (b) Flower two.

Figure 4.1: Put Option.

Chapter 5

Artificial Neural Networks

Artificial Neural Networks are loosely inspired by the biological neural network in the brain an how it processes information, thereby the name. McCulloch and Pitts (1943) [?] were the first who tried to represent the brain of a mammal through an artificial neural network represented by basic brain cells they called neurons.

Rosenblatt (1958) [?] a bit later introduced the perceptron built up of linear threshold units(LTUs). An LTU sums over weighted inputs and applies the step function which outputs 1 if the sum is larger than some threshold and 0 if it is below. An LTU is illustrated in figure ??. The perceptron is then constricted by an input layer, a layer of LTUs connected to all the inputs and produces an output vector containing only binary values. However, the perceptron was restricted. Later, the neocognitron was introduced by Fukushima (1980) [?], a hierarchical multilayer neural network - what motivated for further work on multilayer perceptrons.

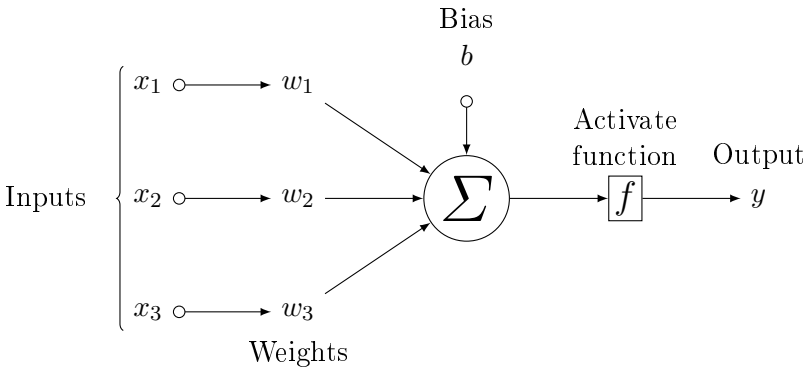


Figure 5.1: An LTU with input of dimension three.

5.1 Multilayer Perceptron

Multilayer perceptrons (MLPs) are networks consisting of multiple layers with neurons. The first layer is an input layer, the last layer is an output layer and the layer between are called hidden layers. The network is deep if it consists of two or more hidden layers.

MLPs are a type of deep feedforward networks. Feedforward neural networks are the most essential deep learning models. It is called feedforward because it is a directed acyclic graph built up of chain structures (where the lengths of the chain structure determines the depth of the network) and the input $\mathbf{x}$ floats *forward* through the network to finally produce an output $\mathbf{y}$. The goal of the network is to approximate some function $\mathbf{y} = f^*(\mathbf{x})$ by defining a mapping $\mathbf{y} = f(\mathbf{x}; \theta)$ and learning the parameters θ that best fit f^*.

It is constructed similar to a perceptron, consisting of an input layer with an input vector and a bias term usually initialized to zero. The input layer is connected to the first hidden layer such that every unit in the hidden layer is connected to every input unit. The difference from the perceptron is in the nonlinear *activation function*. Where the perceptron would use a binary step function, the MLP units can use a wide range of activation functions producing a real-valued output which it passes on to units in the next layer. The process continues like this until it reaches the output layer.

5.1.1 Activation function

The reason for using activation functions is the nonlinear element, namely the possibility to solve nonlinear functions. If all the activation functions were linear, one simply has multiple linear regression. An activation function also gives a restriction on the outputs of the neurons. This can be useful if one wants to squeeze the output in to a range. In addition it increases training stability(source?). Different layers can have different activation functions. The choice of activation function is closely related to the choice of cost function is discussed in section **??**. Common choices for activation functions follow.

Sigmoid

The sigmoid function restricts the output between 0 and 1.

$$f(z) = \frac{1}{1 + e^{-z}}$$

It has a defined gradient for the whole domain and might be especially useful where the output is a probability.

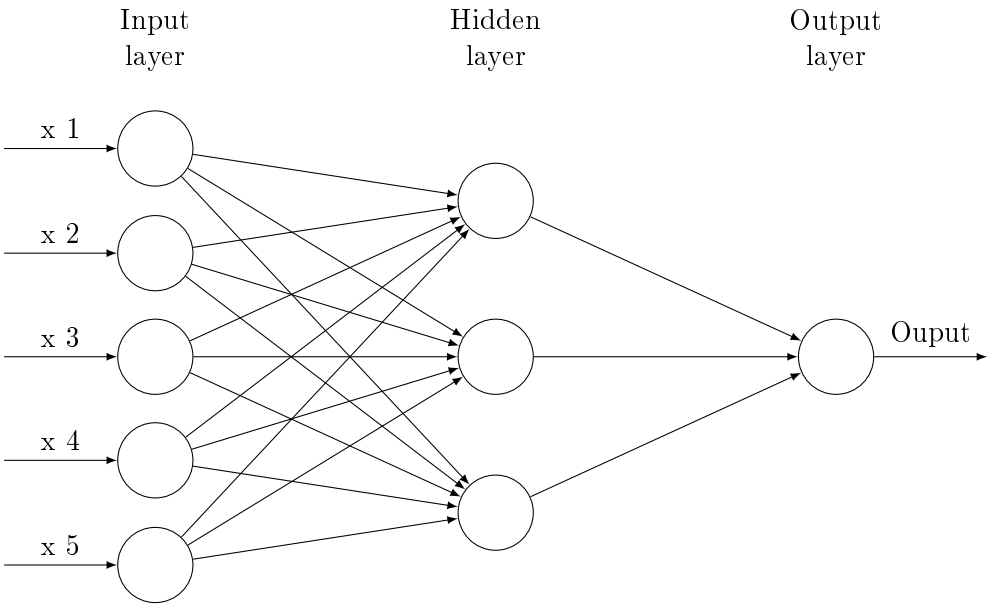


Figure 5.2: Illustration of a MLP with input dimension of five, a single dimension output and one hidden layer with three neurons.

Hyperbolic Tangent

The hyperbolic tangent activation function is quite similar to sigmoid, but restricts the output between -1 and 1.

$$f(z) = \frac{1 + e^{-2z}}{1 - e^{-2z}}$$

Rectified Linear Units (ReLU)

ReLU restricts the output to have a positive value, (but it's variations may produce a small positive value for negative values of the input as well.)

$$f(z) = max(0, z),$$

Exponential Linear Unit (ELU)

ELU is quite similar to ReLU, but outputs small values also for negative input values.

$$f(z) = \begin{cases} \alpha(e^x - 1) & \text{if } x < 0 \\ x & \text{if } x \geq 0 \end{cases}$$

(Is said to converge cost function faster to produce more accurate results(than ReLU?).)

Softplus

Softplus only outputs positive values and is very similar to ReLU, but has a softer transition around $z = 0$.

$$f(z) = \ln(1 + e^z)$$

Softsign

Is quite similar to the hyperbolic tangent, but converges polynomially rather than exponentially.

$$f(z) = \frac{z}{1 + |z|} \tag{5.1}$$

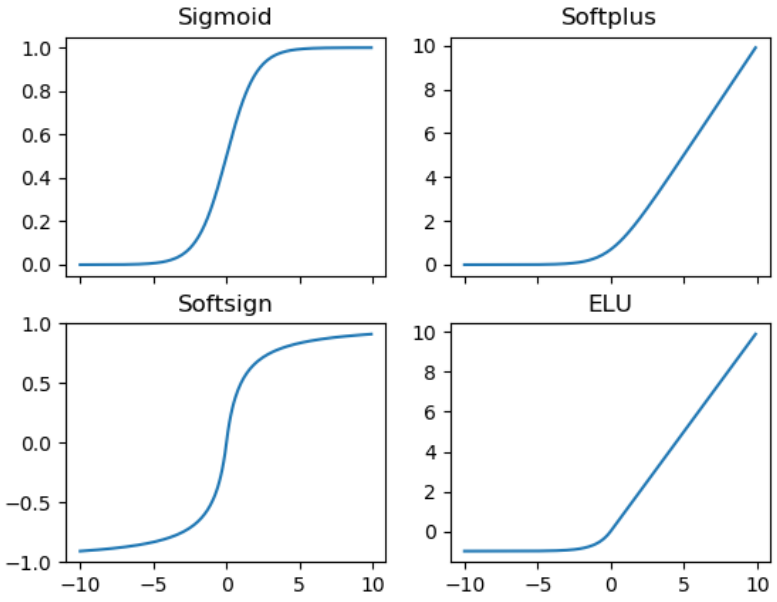


Figure 5.3: Plots of the sigmoid, softplus, softsign and ELU activation functions.

In the 1980s, the sigmoid activation function was very popular as it performed well on small neural networks. One avoided the use of rectified linear units until as late as the early 2000s because of its undefined derivative. But Jarrett et al. (2009) [?] observed that the use of rectiefied nonlinearity was the most important factor of improvement among several other factors that they examined. (The use of rectified linear units also connects the artificial neural networks closer to their biological inspiration as biological neurons operate quite similar, most of them being inactive(sparse activation) and some having output the same order as their input.) For this reason, ReLU and its variations(Leaky ReLU, PReLU) has had increased popularity as of recently.

5.1.2 Cost function

The cost function is a performance measure for the neural network and what is actually optimized with respect to the parameters θ. Important properties of the cost function is to have large and predictable enough gradients (for them to serve as a good guide). However, if activation functions saturates for some values, the cost function tends to saturate as well. Hence, why the combination of activation functions and cost functions should be chosen with care. (Happens for instance with sigmoid for very large positive or negative values.) Another problem that occurs in deep neural networks is the cost functions one usually evaluates become non-convex, i.e. they are harder to optimize as we might not find a global minimum, but merely just a very low value. We therefore use iterative and stochastic gradient-based optimizers. They do not have any global convergence guarantee and may be sensitive to initial values. The weights are (usually) initialized to small random values and the bias to small positive values or even zero. Common choices for regression tasks follow(?).

Mean Absolute Error, MAE

$$J(\theta) = \frac{1}{n}\sum_{i=1}^{n}|y_i - f(x_i;\theta)| \tag{5.2}$$

Mean Squared Error, MSE

$$J(\theta) = \frac{1}{n}\sum_{i=1}^{n}(y_i - f(x_i;\theta))^2 \tag{5.3}$$

Root Mean Squared Error, RMSE

$$J(\theta) = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(y_i - f(x_i;\theta))^2} \tag{5.4}$$

5.1.3 Backpropagation

In order to minimize the cost function, the gradient of the model has to be calculated. This is done using the backpropagation algorithm. The backpropagation algorithm was popularized and first used on multilayer neural networks by Rumelhart et al. (1986) [**?**] (Previous to this, the algortihm has been described and used by others(Linnainmaa, 1970,1976)..?)

The backpropagation algorithm consists of two parts, the forward pass and the backward pass.

Forward Pass

In the forward pass, the training samples are propagated forward through the network, calculating the output of each neuron in a layer using (**??**) and passing forward to the next layers to produce the output.

Let $\mathbf{x} \in \mathbb{R}^d$, $y \in \mathbb{R}$, $b^{(l)} \in \mathbb{R}$ for $l = 1, \ldots, n$ and w_l be the activation function of layer l. In addition, let $W^{(1)} \in \mathbb{R}^{m_1 \times d}$, $W^{(l)} \in \mathbb{R}^{m_l \times m_{l-1}}$ for $l = 2, \ldots n$ be the matrices such that element W^l_{ij} is the weight from node j in layer $(l-1)$ to node i in layer l. The the forward pass is

$$\begin{aligned}
\mathbf{h}^{(1)} &= w_1(b^{(1)} + W^{(1)}\mathbf{x}) \\
\mathbf{h}^{(2)} &= w_2(b^{(2)} + W^{(2)}\mathbf{h}^{(1)}) \\
&\vdots \\
\mathbf{h}^{(n)} &= w_n(b^{(n)} + W^{(n)}\mathbf{h}^{(n-1)}) \\
y &= w^{(out)}(b^{(out)} + \mathbf{w}^{\mathrm{T}}\mathbf{h}^{(n)})
\end{aligned}$$

The algorithm is presented in Algorithm **??**.

Backward Pass

The goal of the backpropagation algorithm is to calculate the gradient of the cost function with respect to any weight or bias in the network,

$$\frac{\partial J}{\partial W^{(l)}_{ij}}, \quad \frac{\partial J}{\partial b^{(l)}} \tag{5.5}$$

It does so by starting with the output layer. Let $\boldsymbol{\delta}^{(n)}$ be such that

$$\boldsymbol{\delta}^{(n)} = \nabla_h J \circ w'(\boldsymbol{z}^{(n)})$$

where

$$\mathbf{z}^{(l)} = W^{(l)}\mathbf{h}^{(l-1)} + b^l \tag{5.6}$$

and ∇_h is the partial derivative with respect to each node in the output layer, $\partial/\partial h^n_j$. ($\delta^{(l)}$ thus has the same dimension as output layer.)

Next, $\boldsymbol{\delta}^{(l)}$ is calculated for layer l given $\boldsymbol{\delta}^{(l+1)}$ for layer $(l+1)$,

$$\boldsymbol{\delta}^{(l)} = \left((W^{(l)})^T \boldsymbol{\delta}^{(l+1)}\right) \circ w'(\boldsymbol{z}^{(l)}).$$

Finally, we obtain the equations for the gradients,

$$\frac{\partial J}{\partial W_{ij}^{(l)}} = h_j^{(l-1)} \delta_i^{(l)} \tag{5.7}$$

$$\frac{\partial J}{\partial b^{(l)}} = \boldsymbol{\delta}^{(l)} \tag{5.8}$$

The backward pass starts by calculating $\boldsymbol{\delta}^{(n)}$ for the output layer and iterates backward through the layers using eqref and eqref. Finally, the gradient is obtained by equation eqref and eqref.

Algorithm 1: Forward Pass

for $k = 1$ *to* m_1 **do**
  $z_k^{(1)} = \sum_{j=1}^{d} W_{jk}^{(1)} x_{ij}$
  $h_k^{(1)} = w_1(z_k^{(1)})$
end
for $l = 2$ *to* n **do**
  for $k = 1$ *to* m_l **do**
    $z_k^{(l)} = \sum_{j=1}^{m_{l-1}} W_{jk}^{(l)} h_j^{(l-1)}$
    $h_k^{(l)} = w_l(z_k^{(1)})$
  end
end

Algorithm 2: Backward Pass

for $j = 1$ *to* m_n **do**
  $\delta_j^{(n)} = \frac{\partial J}{\partial h_j^{(n)}} w'(z_j^n)$
  $\frac{\partial J_i}{\partial W_{jk}^{(n)}} = \delta_j^{(n)} h_k^{(n-1)}$
end
for $l = n - 1$ *to* 1 **do**
  for $j = 1$ *to* m_l **do**
    $\delta_j^{(l)} = w'(z_j^{(l)}) \sum_{k=1}^{m_{l+1}} \delta_k^{(l+1)} W_{jk}^{(l+1)}$
    $\frac{\partial J_i}{\partial W_{jk}^{(l)}} = \delta_j^{(l)} h_k^{(l-1)}$
  end
end

5.1.4 Optimization

The parameters in the MLP are updated using some version of gradient descent(Appendix?). There are many different popular optimizers that are based on(?) gradient descent. In

this section, the Adam algorithm will be presented by first introducing Stochastic Gradient Descent.

Stochastic Gradient Descent

Stochastic gradient descent is a type of gradient descent, but instead of computing the exact gradient using the whole training set in each iteration, it divides the training set into non-overlapping, (roughly) equal subset called batches and computes an approximate gradient on the batches which it iterates over. This way, it is possible to obtain an unbiased estimator of the gradient as we average over a batch of identically independently distributed (data-generated) distributions. The stochastic gradient descent relies upon a hyperparameter - a parameter that is not decided by the algorithm, but has to be chosen in another way. That is the learning rate with which each update is made. Typically, this parameter needs to decrease in each iteration because the gradient estimate includes some noise that does not vanish at the minimum(the true gradient becomes zero at minimum). The algorithm is presented in Algorithm **??**.

Algorithm 3: Stochastic Gradient Descent

Data: Set of learning rates for each iteration, ϵ_k
initialize weights $\boldsymbol{\theta}$
$k \leftarrow 1$
while *stopping criterion not met* **do**
  sample minibatch of size m, input $\{\mathbf{x}_1, \ldots, \mathbf{x}_m\}$ with corresponding output $\{y_1, \ldots, y_m\}$
  compute gradient: $g \leftarrow \frac{1}{m} \nabla_{\boldsymbol{\theta}} \sum_{i=1}^{m} J(f(\mathbf{x}_i; \boldsymbol{\theta}), y_i)$
  update $\boldsymbol{\theta}$: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \epsilon_k g$
end

Sufficient guarantee of convergence is

$$\sum_{k=1}^{\infty} \epsilon_k = \infty \quad \text{and} \quad \sum_{k=1}^{\infty} \epsilon_k^2 < \infty$$

with common choice being to decrease ϵ_k linearly until some iteration, τ,

$$\epsilon_k = (1 - \alpha)\epsilon_0 + \alpha \epsilon_\tau, \qquad \alpha = \frac{k}{\tau}. \tag{5.9}$$

where after τ, ϵ_k is left constant. This requires to decide the parameters ϵ_0, τ and ϵ_τ. (Usually $\epsilon_\tau \approx 0.01\epsilon_0$). ($\epsilon_0$ too large $\implies$ large oscillations, increasing cost function. ϵ_0 too large $\implies$ slow learning, stuck at high cost value. Typically, optimal ϵ_0 is higher than the initial learning rate that gives the best performance after first 100 iterations). Computation time per iteration does not grow with size of training set.

Stochastic Gradient Descent with Momentum

The stochastic gradient descent is a very basic optimizer for neural networks, however learning can be slow. A refinement is the stochastic gradient descent with momentum. (Accelerate learning for high curvature, small consistent or noisy gradients.) It moves in the direction of a accumulated average of past gradients and contains a hyperparameter, α, that determines how fast the previous gradients decay. The larger α compared to ϵ, the more previous gradients determines the direction. Larger step size if many successive gradients have the same direction.

The momentum can be viewed/interpreted in the physical sense. If the position of a particle at any time is $\boldsymbol{\theta}(t)$, then the net force is

$$\mathbf{f}(t) = \frac{\partial^2 \boldsymbol{\theta}(t)}{\partial t^2}$$

which, by introducing $\mathbf{v}(t)$, can be decomposed to

$$\mathbf{f}(t) = \frac{\partial \mathbf{v}(t)}{\partial t} \tag{5.10}$$

$$\mathbf{v}(t) = \frac{\partial \boldsymbol{\theta}(t)}{\partial t} \tag{5.11}$$

where $\mathbf{v}(t)$ can be interpreted as the velocity. (**??**) and (**??**) can be solved numerically, the easiest solver being Euler. The two forces are proportional to the gradient of the cost function such that it always moves in direction of the negative direction of the gradient, and a force that can be thought of as a viscous drag that makes the particle to a local minimum, proportional to $-\mathbf{v}(t)$. Stochastic gradient descent with momentum is presented in Algorithm **??**.

Algorithm 4: Stochastic Gradient Descent with Momentum

Data: learning rate, ϵ
Data: momentum parameter, α
initialize weights $\boldsymbol{\theta}$
initialize velocity $\boldsymbol{v}$
while *stopping criterion not met* **do**
  sample minibatch of size m, input $\{\mathbf{x}_1, \ldots, \mathbf{x}_m\}$ with corresponding output $\{y_1, \ldots, y_m\}$
  compute gradient: $g \leftarrow \frac{1}{m} \nabla_{\boldsymbol{\theta}} \sum_{i=1}^{m} J(f(\mathbf{x}_i; \boldsymbol{\theta}), y_i)$
  compute velocity update: $\boldsymbol{v} \leftarrow \alpha \boldsymbol{v} - \epsilon g$
  update $\boldsymbol{\theta}$: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \boldsymbol{v}$
end

Adam

Even though Stochastic Gradient Descent with momentum performs better than just Stochastic Gradient Descent, there is still a hyperparameter that needs to be decided.

Adam, short for adaptive moments, is an algorithm with an adaptive learning rate. It first computes a first- and second-order moment for which it secondly performs a bias correction to account for the initialization made at 0. The algorithm is presented in Algorithm **??**.

Algorithm 5: Adam

Data: step size, ϵ (default: 0.001)
Data: exponential decay rates for momentum, ρ_1, $\rho_2 \in [0, 1)$ (default: 0.9, 0.999)
Data: small constant δ for numerical stabilization (default: 10^{-8})
initialize weights $\boldsymbol{\theta}$
initialize first and second momentum: $\mathbf{s} = 0$, $\mathbf{r} = 0$
$t \leftarrow 0$
while *stopping criterion not met* **do**
  sample minibatch of size m, input $\{\mathbf{x}_1, \ldots, \mathbf{x}_m\}$ with corresponding output $\{y_1, \ldots, y_m\}$
  compute gradient: $\mathbf{g} \leftarrow \frac{1}{m} \nabla_{\boldsymbol{\theta}} \sum_{i=1}^{m} J(f(\mathbf{x}_i; \boldsymbol{\theta}), y_i)$
  $t \leftarrow t + 1$
  update first biased momentum: $\mathbf{s} \leftarrow \rho_1 \mathbf{s} + (1 - \rho_1)\mathbf{g}$
  update second biased momentum: $\mathbf{r} \leftarrow \rho_2 \mathbf{r} + (1 - \rho_2)\mathbf{g} \circ \mathbf{g}$
  correct first biased momentum: $\hat{\mathbf{s}} \leftarrow \frac{\mathbf{s}}{1 - \rho_1^t}$
  correct second biased momentum: $\hat{\mathbf{r}} \leftarrow \frac{\mathbf{r}}{1 - \rho_2^t}$
  compute update: $\Delta\boldsymbol{\theta} = -\epsilon \frac{\hat{\mathbf{s}}}{\sqrt{\hat{\mathbf{r}}} + \delta}$
  update: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \Delta\boldsymbol{\theta}$
end

5.2 Generalization

The Universal Approximation Theorem first presented by Cybenko (1989) [**?**] showed that any function that is continous on a closed and bounded set of $\mathbb{R}$ can be represented by a single hidden layer feedforward network with the sigmoid activation function and a linear output. Hornik et al. (1991) [**?**] then showed that this is true for a wide range of activation functions. However, we do not know the number of neurons needed and have no guarantee for the ability of our network to learn this representation. Generalization is the ability the model has to predict unobserved data. Empirically, by Bengio et al., 2006 [**?**] and other (Erhan et al., 2009; Bengio, 2009;Mesnil et al., 2011; Ciresan et al., 2012; Krizhevsky et al., 2012; Sermanet et al.,2013; Farabet et al., 2013; Couprie et al., 2013; Kahou et al., 2013; Goodfellowet al., 2014d; Szegedy et al., 2014a), the more layers we add, the better is the generalization of the neural network and the less neurons are needed. To observe how well our model generalizes, it is common to divide the data set into a training and a test set. The training set is used to train the model and the

test set is used to measure the error for unobserved data. If the training error is not sufficiently low, we have what is called underfitting, or high *bias*. If the training error is sufficiently low, but the gap between the training and test error is high, we have overfitting, or high *variance*. Overfitting occurs from sensitivity to small deviations in the data often causing the network to model noise. In deciding on a network architecture, we encounter the bias-variance tradeoff problem. Increased number of hidden units will increase variance and decrease bias causing overfitting, while too few hidden units may lead to underfitting. There are several ways of approaching this problem. (A common approach in machine learning) is the use of cross-validation.

In cross-validation, the training data is split into two subsets, one set to train our model, confusingly enough often called training set, and one validation set to test the generalization error. This can be done in several ways depending on the size of the data. In k-fold cross validation, the data is randomly split into k folds of roughly equal size. For each iteration, $(k-1)$ samples are used as training data and one sample is held out as validation data for which a validation error is estimated to test the model. This is done k times such that each fold is held out exactly once. Then an average of the validation errors is calculated and used as an estimate of the error. In this way, we get a good estimate of how well the model performs on unobserved data (or how well it predicts). Thus this might be a good pointer for which neural network architecture to choose. The k-fold cross validation algorithm is presented in Algorithm **??**. The choice of network architecture is also though, a lot of trail and error.

Algorithm 6: k-fold cross validation

Data: training data, $\mathbb{D}$, such that element i in $\mathbb{D}$ is the input, target pair $(\mathbf{x}^{(i)}, y^{(i)})$
Data: a learning algorithm, A
Data: a loss function, L
Data: number of folds, k
Split $\mathbb{D}$ into k mutually exclusive subsets(such that $\bigcup_{i=1}^{k} \mathbb{D}_i = \mathbb{D}$)
for i *from* 1 *to* k **do**
    $A_i = A(\mathbb{D}/\mathbb{D}_i)$
    for $(\boldsymbol{x}^{(j)}, y^{(j)})$ *in* $\mathbb{D}_i$ **do**
        $e_j = L(A_i(\boldsymbol{x}^{(j)}), y^{(j)})$
    end
end
Return $\boldsymbol{e}$

5.3 Model Setup

5.3.1 Data Calibration

Hutchinson et. al(1994) [?] made the first attempt to solve option pricing using an MLP on both Monte Carlo simulated underlying stock price for option prices as well as real S&P 500 futures and options using only two input parameters, namely time to maturity $(T-t)$ and the moneyness - the ratio between the underlying stock price and the strike price, S/E. The output was the ration between the option price and the strike price, C/E. Reducing the number of input parameters reduces the complexity of the model and necessity for more hidden units. In addition, normalized input data to the same order of magnitude also reduces the complexity of the model as the weight updates are proportional to the input data(source?). Hutchinson et al. (1994) [?] argued for the use of only two input parameters by Theorem 8.9 of Merton (1990) (source!!!) where it is stated that the option pricing formula is homogeneous of degree one in stock price per share and strike price (??). Garcia and Gencay (2000) [?] also showed the use of this assumption in feedforward neural networks greatly improves performance. The authors of Hutchinson et al. (1994) [?] (also) argued that with the assumption that the volatility and the risk-free rate is constant during the life of the option, this will not be picked up by the model. However, this is one of the limitations of the Black-Scholes formula, and providing the model with some measure of the latter two would be useful. For further work, it has been suggested(Fogarasi(2004) [?]) to start out with a lower number of input variables which one increases to achieve more accuracy. This was done (for example) by Amilon (2003) [?], using a total of nine input variables, including 10- and 30-days historical volatilities as well as lagged prices of the underlying stock.

$$C = f(S, E, t; T) \implies \frac{C}{E} = f(\frac{S}{E}, 1, t; T) \tag{5.12}$$

(??) showes the homogeneity assumption for option price C, exercise price E, stock price S and time to maturity T.

In this paper, there is four input parameters - volatility, moneyness, exercise price and risk free rate. The underlying stock prices was collected from 150 different companies on the S&P 500 using Alpha Vantage [?]. Then the strike price, volatility, time to expiry and risk free rate was set in the following way.

- Start price, S_0 for the option was set to the price at the start of the option for 150 different companies.
- Exercise price ratio, E_r was first drawn from a uniform probability distribution $E_r \sim U(0.5, 1.5)$ with a sample size of 10. For each stock price, 10 different exercise prices, E were then calculated by multiplying exercise ratio with the terminal stock price, $S(T)$, for each company. In this way, all exercise ratios $E_r < 1$ will make options expiring out of the money, while exercise ratios $E_r > 1$ will expire in the money and exercise ratios $E_r = 1$ will expire at the money.

- The annualized risk free rate, r was drawn from a uniform probability distribution $r \sim U(0.015, 0.025)$ with a sample size of 100.
- The time to maturity, T was set to 92 days $\approx$ 3 months for all the options and was annualized for 252 trading days per annum.
- The annualized volatility, $\hat{\sigma}$, was calculated using 1 year of historical data prior to the start of the option.

The input data, X, is then structured on the following form

$$X = \begin{bmatrix} \mathbf{x}_1 \\ \mathbf{x}_2 \\ \vdots \\ \mathbf{x}_N \end{bmatrix}$$

such that for $i = 1, \ldots, 150$, $j = 1, \ldots, 100$ and $k = 1, \ldots, 10$

$$\mathbf{x}_{k+10j+10\cdot 100\cdot i} = [\sigma_i, \frac{S_i(0)}{E_{r,j}}, r_k, T]. \tag{5.13}$$

Hence, there are $N = 150000$(and X has dimension 150000×4). The remaining output $\mathbf{y}$ is then calculated as

$$y_{k+10j+10\cdot 100\cdot i} = \frac{e^{-r_k T}}{E_{r,j}} \max\Big(S_i(T) - E_{r,j}, 0\Big) \tag{5.14}$$

(and has dimension 150000×1).

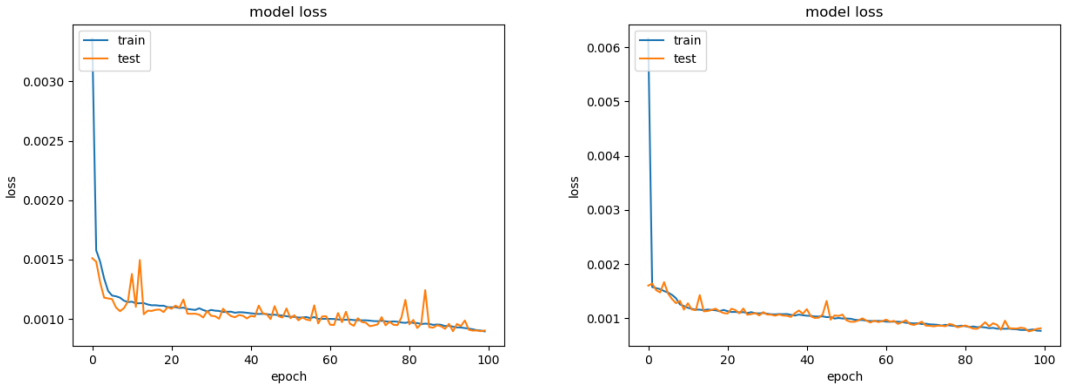
5.3.2 Network Architecture

In Hutchinson et al. (1994) [?], the authors experimented with three different neural networks - Radial Basis Functions(RBFs), Projection Pursuit Regression(PPR) and MLP, discovering that MLP performed best. In combination with it's straightforward configuration, this most likely motivated the further research on the use of MLP to solve Option Pricing. Anders et al. (1996) [?], Gencay and Salih(2003) [?] and Yao et al. (2000) [?] mainly focused on the use of MLP, but used small and shallow networks. Hutchinson et al. (1994) [?] used a one hidden layer network with 4 hidden units and 2 inputs. Benell and Sutcliff(2004) [?] experimentet with one hidden layer with 3-5 hidden units and 3-7 inputs. Anders et al. (1996) [?] experimented with a sparse network(not all units are connected) with one hidden layer, 3 hidden units and 4 inputs. They all still achieved good performance, hence one would expect that a deeper and wider network would do even better. This belief is also supported by studies on better generalization with deeper networks(Bengio et el. (2006) [?]) and by today's computer power it is possible to get results for networks with several hundred hidden units and tens of layers in reasonable time.

(In this paper, it has been experimented with an architecture consisting of $20 - 100$ units per layer for $3 - 6$ layers.) The network has been implemented using Keras [**?**].

Chapter 6

Results and Analysis



(a) Training loss for batch size of 32.

(b) Training loss for batch size of 128.

Figure 6.1

By figure **??** adn **??**, the Black-Scholes formula clearly overprices options (especially higher valued options) and the Monte Carlo simulations underprices(especially higher valued options). (The overpricing Black-Scholes is consistent with the findings in (?)). In figure **??** and **??**, one can see the neural network is more equally spread around the true values in addition to the spread being smaller than for BS and MC. The values from table.. confirmes the smaller spread with lower error values.

Improvements:
For further work, I think it would be interesting to look at more realistic data. In this paper, real underlying stock data was used, then standard pricing rules were applied to calculate option prices, but maybe it would be possible to use even more realistic option pricing rules, look at how the the options are actually quoted(?). Or use real option price data. Also, evaluating performance of options with different maturities.

Table 6.1: Caption

(a) sfsdsdf

Property	Value
Batch size	32
Epochs	100
Optimizer	Adam
Loss Function	MSE
Training size	95999
Validation size	24000
Test size	30001

(b) asd

Layer	No. Neurons	No. Trainable Parameters	Activation Function
1	100	500	ELU
2	80	8080	ELU
3	60	4860	ELU
4	40	2440	ELU
5	20	820	Softplus
6	1	21	

(c) Chose softplus because only takes positive values

	RMSE	MSE	MAE
Training data	0.026871	0.000722	0.014593
Test data	0.027366	0.000749	0.014796

Table 6.2: Caption

(a) sfsdsdf

Property	Value
Batch size	128
Epochs	100
Optimizer	Adam
Loss Function	MSE
Training size	95999
Validation size	24000
Test size	30001

(b) asd

Layer	No. Neurons	No. Trainable Parameters	Activation Function
1	100	500	ELU
2	80	8080	ELU
3	60	4860	ELU
4	40	2440	ELU
5	20	820	Softplus
6	1	21	

(c) Chose softplus because only takes positive values

	RMSE	MSE	MAE
Training data	0.028390	0.000806	0.015940
Test data	0.027876	0.000777	0.015806

Table 6.3: The tables show RMSE, MSE and MAE performance for Black-Scholes(BS), Monte Carlo simulation(MC) and multilayer perceptron(MLP) for training data and test data. In addition, in table c), the whole data set is divided into out-of-the-money(OTM), in-the-money(ITM) and near-the-money(NTM) if the moneyness is respectively $S(T)/E < 0.98$, $S(T)/E > 1.02$ and $< S(T)/E$

(a) some caption

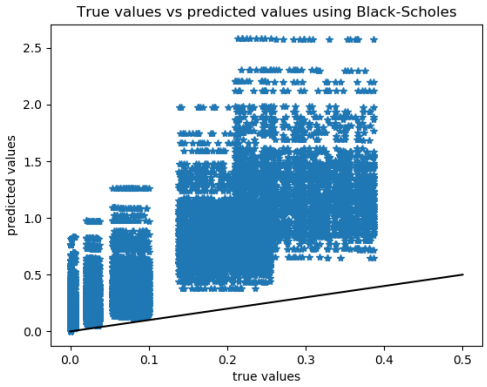
Model	RMSE	MSE	MAE	Observations
BS				119999
MC				
MLP	0.028396	0.000806	0.015998	

(b) some

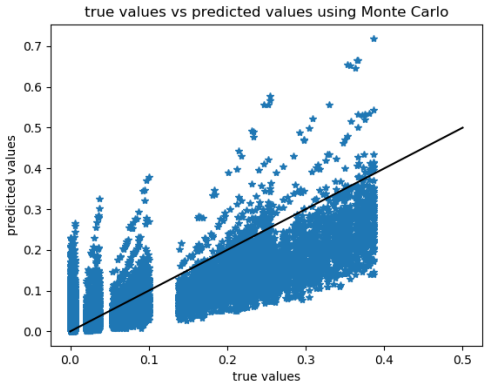
Model	RMSE	MSE	MAE	Observations
BS				30001
MC				
MLP	0.028149	0.000792	0.015915	

(c) skfja

	Model	RMSE	MSE	MAE	Observations
OTM	BS	1	1	1	90957
	MC	1	1	1	
	MLP	0.000163	0.012775	0.005892	
ITM	BS	1	1	1	58116
	MC	1	1	1	
	MLP	0.001609	0.040110	0.028274	
NTM	BS	1	1	1	927
	MC	1	1	1	
	MLP	0.00834	0.028886	0.017241	

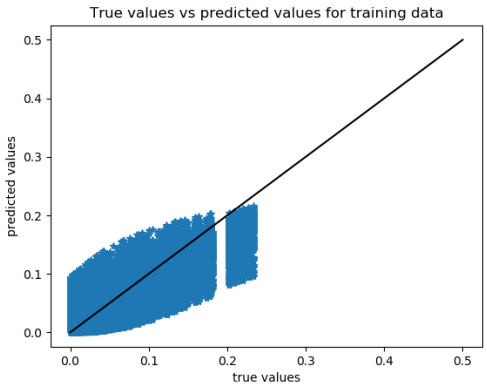


(a) Predicted moneyness plotted against true moneyness using Black-Scholes.

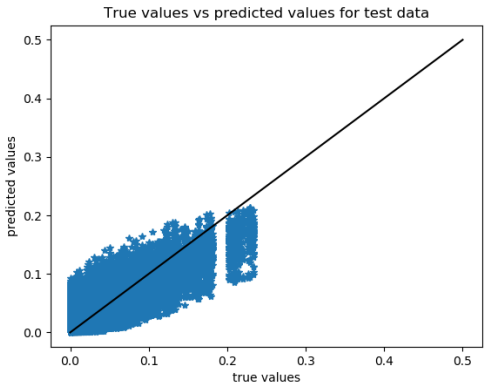


(b) Predicted moneyness plotted against true moneyness using Monte Carlo.

Figure 6.2



(a) Predicted moneyness plotted against true moneyness using MLP for training data.



(b) Predicted moneyness plotted against true moneyness using MLP for test data.

Figure 6.3

To look at performance, it could be informative to also look at the delta hedging performance as done by e.g. Hutchinson et al. (1994) [?], Anders et al. (1996) [?] and to evaluate other performance measures such as R^2.

In this paper, the dividend yield for the stock prices was not included in the Black-Scholes formula and Monte Carlo simulations. This might explain the underpricing of the Monte Carlo simulations as it is based on the expected returns based on historical returns which might have been underestimated if dividend was paid out during the period for which the expected return was calculated. This might also have infected the calculation of the historical volatility. The dividend was ignored for simplicity as there was selected 150 different companies from the S&P 500 that might all have different dividend policies. However, including this could improve the performance of Black-Scholes and Monte Carlo.

Several improvements could also have been done for the multilayer perceptron. For instance, there was no regularization applied such as early stopping, setting a dropout rate or L1/L2 regularization. One could also experiment more with the architecture, use of optimizer and default settings for hyperparameters, activation functions, batch size and number of epochs. One could also experiment with input parameters as done by e.g. Amilon (2003) [?], with more volatility parameters based on different sized historical data, implied volatility, lagged values of the stock price or different measures of the time to maturity if the data set contained options with different maturities.

In this paper, only European call options were evaluated, but it would be interesting to see how a neural network performed on other types of options such as American with free boundary values or path-dependent Asian options, and compare to existing pricing models.

Weaknesses of BS and MC?

Chapter 7

Conclusion

Appendix

Random Walk

Itô's Lemma