UNIVERSITY OF CAPE TOWN

MASTERS THESIS

Stock Option Valuations and Constraint Enforcement Using Neural Networks

Author:
Frans Ignatius NUTT

Supervisor:
Dr. Etienne PIENAAR

A thesis submitted in fulfillment of the requirements for the degree of Master of Actuarial Science

in the

Department of Actuarial Science

April 26, 2022

Declaration of Authorship

I, Frans Ignatius NUTT, declare that this thesis titled, "Stock Option Valuations and Constraint Enforcement Using Neural Networks" and the work presented in it are my own. I confirm that:

- This work was done wholly or mainly while in candidature for a research degree at the University of Cape Town;
- No part or constituent of this dissertation has previously been submitted for a degree or any other qualification at this University or any other institution;
- Where I have consulted the published work of others, this is always clearly attributed;
- Where I have quoted from the work of others, the source is always given and with the exception of such quotations, this thesis is entirely my own work.

Frans Ignatius Nutt
2021

UNIVERSITY OF CAPE TOWN

Abstract

Faculty of Commerce
Department of Actuarial Science

Master of Actuarial Science

Stock Option Valuations and Constraint Enforcement Using Neural Networks

by Frans Ignatius NUTT

Stock option valuations have long been studied, being inherently non-linear financial derivatives. These instruments have a ubiquitous presence in institutional investment practice, and present many favourable and unique benefits to an investment portfolio. Neural Networks on the other hand have become a more familiar concept in recent times. They are by design set to deal with complex, non-linear classification and prediction tasks. Using Neural Networks to predict stock option prices has been studied at length, by various authors in the last 30 years. These studies have considered their relative performance against closed-form pricing solutions like the infamous Black-Scholes-Merton model, as well as in real-world settings. The collective conclusion that is deduced from past literature presents a clear case for their use in finance, albeit that there are some notable pitfalls, like the lack of interpretability and the ability to explicitly enforce certain constraints. Constraints such as option price bounds (upper and lower) and the Put-Call parity, that a stock option's value should satisfy have not been considered in many prior studies. This dissertation sets out to study stock option valuations using Neural Networks with techniques to enforce constraints. First, a functional and appropriately performing Neural Network configuration is derived that outputs European call and put option prices under one model. Thereafter, enforcement of the lower, upper and relative bounds (Put-Call parity) is incorporated into the model. Finally, the Neural Network application is extended to the real-world setting. The performance of the Neural Network model is assessed by means of mean error, as well as percentiles.

Acknowledgements

First and foremost I would like to give thanks to God, whom has blessed me with the talents I have. I would also like to pay special thanks to my supervisor Etienne. From this experience, I have learned more than just what the dissertation is about. Etienne has challenged me to really think about a problem and ask the right questions. I would also like to thank my wife, who has been a pillar and support structure in my life that I cannot possibly repay. Finally, I would like to pay special thanks to my parents. My mother has shown me the meaning of consistency and perseverance, and for that I am eternally grateful.

Contents

List of Figures

List of Tables

List of Abbreviations

AI	**A**rtificial **I**ntelligence
ARE	**A**bsolute **R**elative **E**rror
BSM	**B**lack-**S**choles-**M**erton
CVE	**C**ross-**V**alidation **E**rror
DAE	**D**ifferenced **A**bsolute **E**rror
ETF	**E**xchange **T**raded **F**und
FNN	**F**eedforward **N**eural **N**etwork
GBM	**G**eometric **B**rownian **M**otion
GD	**G**radient **D**escent
MAE	**M**ean **A**bsolute **E**rror
MARE	**M**ean **A**bsolute **R**elative **E**rror
MLP	**M**ulti **L**ayer**P**erceptron
MSE	**M**ean **S**quared **E**rror
NN	**N**eural **N**etwork
PDE	**P**artial **D**ifferential **E**quation
PPR	**P**rojection **P**ursuit **R**egression
RBF	**R**dial **B**asis **F**unction
SBM	**S**tandard **B**rownian **M**otion
SGD	**S**tochastic **G**radient **D**escent
UAT	**U**niversal **A**pproximation **T**heorem

Dedicated to my wife, friends and family

Chapter 1

Introduction

Artificial intelligence, machine learning and the age of big data has been a popular topic as the technological advancement of humanity leaps forward. There have been major developments in this field in recent decades, amplified by the more common availability of powerful computational software and hardware (Goodfellow and Courville, 2016). The rise of social networks, search engines and the internet of things has scaled the amount and format of data that is available to analyse and use in providing improved solutions or entirely new solutions to old problems. The range and applications of this have played a prolific role in the development of machine learning algorithms, being inherently more capable of analysing complex classification and prediction problems (Goodfellow and Courville, 2016). The world of finance has a deep and sophisticated underpinning of data science and analysis inherent to its practice. The use of advanced learning algorithms and techniques has arguably become necessary for participants who aim to remain competitive.

In the sphere of statistical finance, the application of machine learning and artificial intelligence has seen a resurgence of interest with potential applications in the modelling of foreign currency exchange rates, price trajectories in stock markets, prediction of default rates in banking portfolios and macroeconomic forecasting (Huang, 2020). Insurance companies, by means of AI and machine learning can approach long-standing problems in completely new ways. For example, life insurance underwriting can be conducted using machine learning classification algorithms on image data (image of the prospective policyholder) and therefore a life policy can be priced. Although many fields of finance have seen successful application of these technologies, one application has perhaps been less adaptable, namely, the pricing of financial derivatives.

Financial derivatives, in their most simplistic sense, are financial instruments whose value depends on the value of another, namely the underlying asset. Derivatives are ubiquitous in global financial markets perhaps owing to the utility of such instruments both as speculative instruments but also a means of managing the risk-profiles, transaction cost and returns dynamics of more traditional financial instruments such as stocks and commodities (Apanard and Wickramarachi, 2014). Their efficiency in portfolio hedging, re-balancing and exposure management have seen their use become commonplace in portfolio management (Apanard and Wickramarachi, 2014). They provide several benefits as opposed to taking the same economic exposure in the underlying assets, as they also provide certain challenges relating to the difficulty of placing a fair value on them. At the forefront of their benefits is the ability to change portfolio exposures faster, cheaper and more effectively in the sense of hedging systematic risk exposures. While being inherently complex to value, they also pose some operational and credit related risks. The credit risk exists more specifically in the realm of “over-the-counter” markets, where the “counter” is notional and refers to being privately executed trades (not publicly listed on an exchange) and is usually facilitated and sold by investment banks, shadow banks (typically financial intermediaries that are not subject to regulatory oversight) and other market makers. In recent years, the market for derivatives has grown both in size and the range of derivatives under offer(Apanard and Wickramarachi, 2014).

Fischer Black, Myron Scholes and Robert Merton presented a seminal and Nobel prize-winning development for pricing and valuing derivatives in the 1970’s, now known as the novel Black-Scholes-Merton (BSM) model (Hull, 2012). Their approach was borne from the well-known Capital Asset Pricing Model (CAPM), coupled with Merton’s idea of setting up risk-free portfolios in the sense of being "delta-hedged" (Hull, 2012). They essentially developed a closed-form solution, boasting that their European stock option pricing solution is highly interpretable, differentiable and hence measures of risk can be derived. These measures are what are commonly referred to as the “Greeks”, where Greek symbols represent different measures of sensitivity of the options’ value to the underlying and derivative contract specific factors. BSM and related replication strategies have been known to be modified and used by quantitative analysts ("quants") for the past 45 years (Sherif, 2019). The initial BSM model for European type options has an underlying framework in which a risk-neutral environment is used to derive closed-form solutions based on a fundamental partial differential equation. However seminal and useful, the BSM framework is known to be an assumption intensive approach in the sense that the framework under which the closed-form solution was developed lives under some "strong" and arguably unrealistic assumptions (Hull, 2012. The BSM framework is still capable of being modified, where parameters can be altered to calibrate the model to be "self-consistent" in producing realistic pricing dynamics for stock options. Many modifications and advancements have been made to BSM, however its service as baseline appears to be embedded in option valuation practice (Hutchinson, Lo, and Poggio, 1994). This, along with the Greeks, are reasons why the framework is still widely used and adopted, with the various alterations and amendments deployed depending on the particular application at hand (Hutchinson, Lo, and Poggio, 1994). These are all attempts to improve models for a wider spectrum of market climates and thereby improving the hedging informed by such models. Quants are generally more concerned with the real dynamics of market prices, and the traditional BSM framework has often proved to fail when volatility regimes increase or change dramatically, leaving the users under/over-hedged and hence exposed to adversarial portfolio movements (Sherif, 2019). A famous example of this was when a hedge fund, Long-term Capital Management had to be bailed out and eventually liquidated, managed by no less than Black and Merton, due to model error. Detailed account of this can be found in Hull, 2012.

Artificial Intelligence (AI) and its most well known applications like autonomous driving, natural language processing, image and video classification etc., are frequently modelled using a deep learning framework (Goodfellow and Courville, 2016). Neural Networks, being the general framework of deep learning, is a mathematical framework inspired by how the human brain works, and specifically recognises and "learns" complex patterns, relationships and other general dynamics of the process at hand (Culkin, 2017). The term deep learning refers to the fact that there are several hidden layers in a Neural Network (NN) model. The more hidden layers a NN model has, the more non-linear transformations and parameters the model has, and thus the "deeper" the learning is (Goodfellow and Courville, 2016). Culkin, 2017 describes the success of AI to be a confluence of 3 factors, namely; mathematics supporting fast network calibration, depth of the network allowed by major computational advancement, training machines to replicate very complex human phenomena is only possible with big data.

NNs have received a lot of recent attention for their ranging flexibility to perform very well for an array of complex classification or prediction-based problems, that in the past had not been possible to model at the desired accuracy. In particular, they have a unique ability to deal with profound non-linearity inherent to the underlying stochastic processes and boast the benefit of unsupervised feature engineering (Huang, 2020).

In recent years there have been several applications of NNs in finance, ranging from pricing options to time-series forecasts on stock prices (Huang, 2020). Among the many benefits of NNs and more holistically, machine learning, there is a major drawback, besides them not necessarily producing better estimates, which is their susceptibility to spurious accuracy and generally limited model interpretability. These models typically contain vast amounts of parameters that don't relate to any one input feature/explanatory variable. By contrast, frameworks such as structured diffusion models and BSM have the advantage of being parameterised by relatively few parameters, each bearing an understandable interpretation.

It is important to understand the immense benefit that BSM and modifications or extensions of BSM provide users through its closed-form and interpretable model structure. Additionally, such models lend themselves to design/configuration that enforce economic soundness and consistency. A typical NN, often comprised of several thousand parameters or more, does not guarantee economic consistency explicitly. There is therefore no particular way of explicitly enforcing or ensuring certain restrictions, for example, ensuring that the Put-Call parity holds (Beucler et al., 2021).

In this dissertation we will investigate methods to overcome this particular shortcoming of using NNs in pricing stock options. The approach taken to investigate these methods rely on first establishing a suitable NN configuration that will suffice a suitable model to predict stock option prices with. The configured NN model will then be used to investigate how the NN can be taught to enforce economic constraints such as the Put-Call parity.

In the subsequent section, literature around stock option pricing dynamics and previous NN option pricing studies are discussed. Thereafter, in chapter 3, NNs and the various attributes that make up their configuration are elaborated in detail. Chapter 4 follows with a mathematical exposition of NNs and how the cost gradient with respect to (w.r.t.) each model parameter is computed. Chapter 5 sets out the premise of the experimental design pertaining to this dissertation. Chapter 6 exhibits the results, findings and interpretations of the experiments set out. Finally, Chapter 7 concludes this study of option pricing using NNs.

Chapter 2

Derivative Valuations

2.1 Derivative Valuations and Neural Networks

Machine learning and AI is revolutionising virtually all data intensive industries. Unsurprisingly, tech giants such as Google, Facebook and Amazon are at pioneering both the science of AI as well as the real-world applications thereof. Financial markets are well-known for their quantitative nature and the added complexities of behavioral finance materially impacting the levels and movement of financial asset markets. By all accounts, financial markets are more competitive and complex than ever. How data is used to establish a competitive position has arguably become more difficult. The range of approaches that can be taken to solve old problems like valuing derivatives and hence portfolios is notable. AI based solutions in finance boast the ability to process and use wide ranging data formats, often having much higher dimensions i.e. the number of explanatory variables that can be used. Though this is appealing, for particular applications they could present particular difficulties and costs due to the "black box" nature i.e., without a clearly interpretable structure that can be inferred from the underlying model equations, inherent to many of these AI-based approaches.

Traditional techniques for evaluating financial derivatives are built on the BSM framework. Although the BSM framework has many shortcomings and generally a strong and unrealistic assumption basis, the closed-form and structured nature of BSM provides many practical benefits. Overall, BSMs intuitive and interpretable nature is a major benefit to users. Risk-neutral valuation's are typically used in modeling the value of stock options, like BSM. Under risk-neutral valuations, the discounted payoff process is a martingale. Therefore, the drift of such a process is simply the risk-free rate (Hull, 2012). This entails that a risk-free rate like the London Inter-bank Overnight Rate (LIBOR) or other relevant funding rates, is used to discount the expected payoff profile of a derivative to determine their value. BSM is widely referenced and used in many applications relating to option valuations e.g. determining of the VIX index, which is a measure of the implied volatility. The volatility parameter that underlies the payoff process of an option is often described by its own process, and hence allows the BSM framework to be modified and made capable of modelling greater non-linearity, thereby overcoming some of the caveats BSM is known for. Arguably, BSM itself has an impact on how the market values options (Hull, 2012).

Researchers have seen that the use of NNs are very capable of learning complex phenomena, be it in prediction or classification based tasks (Goodfellow and Courville, 2016). Pricing options using NNs is one possible approach that can be taken to introduce modern analytics to an old problem. Researchers have conducted several experiments in the context of derivative valuations using NNs, from "learning" the BSM model, to "learning" to price real-world options prices (Bennell and Sutcliffe, 2004).

In this section, the fundamentals of stock options and BSM are discussed. Option values have certain economic constraints that have to be met in order for any one valuation model to be practically useful. Therefore, when considering the valuation approach, it is essential that the economic constraints are considered over and above the prediction performance of a model. If a model can achieve a prediction error of below say 1%, but fails to meet certain economic constraints, their practical use is limited. As the stochastic process of an options underlying asset may experience changes, while not explicitly designed to be consistent with economic constraints, the performance of such a model may degenerate severely when predicting on data that they were not trained on i.e. out-of-sample data.

Herewith, the literature and foundations of stock options and BSM are set out. In order to investigate the ability of NNs to value stock options, it is fundamental to capture not only the mapping of inputs to outputs, but also the nature of economic constraints that underlie the stock option valuation. This section sets out the representation of economic constraints, referred to as bounds, followed by the fundamentals of BSM and the literature of option pricing using NNs.

2.2 Stock Options: Bounds and Valuation

2.2.1 Introduction

There have been several publications and studies conducted on pricing stock options using NNs over the past decades, especially the most recent one. The famous BSM framework appears at least as a reference point in all of the studies conducted, testament to their established ubiquity in the universe of option valuations. Stock options are generally characterised by two types, namely call options and put options. For these options, there are two distinct settlement styles, namely European and American. European options relate to options where there is a pre-defined exercise date (Hull, 2012). American options are ones where the option can be exercised on a range of dates up until the option maturity date, and hence provides an additional positive value to the long party of them (Hull, 2012). Put options refer to taking negative economic exposure, where the long party has the right, but not the obligation, to sell the underlying asset at a predefined price known as the strike price. Call options on the other hand, the long party takes positive economic exposure and has the right, but not obligation, to buy the underlying asset at the strike price. Put options are therefore valuable when the price falls below the strike price, whereas calls are valuable when the price rises above the strike.

Stock option prices can be seen as bounded spaces in the sense that they have upper and lower absolute bounds, as well as relative bounds. The relative bound is essentially described by what is commonly known as the Put-Call parity. These bounds are important, since if they are not met by the pricing model, they expose the model owners and users to financial risks and/or limitations in using a specific model. The focus of this research is narrowed to European call and put option valuations based on equities, which is by far the most liquid options market, hereafter referred to as calls and puts.

The following expressions represent the payoff's calls (c) and puts (p) at expiry:

$$c = \text{European call option at time zero} = max(S_T - K, 0),$$

$$p = \text{European put option at time zero} = max(K - S_T, 0),$$

where S_T represents the underlying asset at maturity time T, and K represents the strike price.

2.2.2 Option Price Bounds

Lower Bounds

European call and put options are assets, which is naturally bounded on the lower end at zero. More specifically, derivatives are financial assets whose value depends on and is derived from another asset known as the underlying asset. The underlying asset, being stocks, bonds, futures and other more complex assets classes, is bounded by zero mostly, with exceptional cases where the asset price (specifically commodities futures such as oil) can go below zero for brief periods when the cost of carry is rapidly increasing. Besides this, it is accepted and reasonable, especially for equities, that by definition, equity prices are zero only when they are bankrupt, are bounded. Hence we have lower bounds for a derivative, derived from the value of an underlying asset which can, at worst, expire worthless:

$$c \geq 0,$$

and;

$$p \geq 0.$$

Upper Bounds

Calls have upper bounds. Intuitively this can be thought of from the fact that call options, with a strike price greater than 0 must have a value less than the underlying assets. By looking at the above payoff expressions, the payoff given a non-zero value of the underlying, is less than the underlying asset price with probability one (100% certainty) given a positive strike price (deterministically known). Therefore, this alone, without considering the probabilistic nature of the non-zero payoff occurrence ensures that:

$$c \leq S_T$$

at maturity (time T), and therefore;

$$c \leq S_t$$

$\forall t$, for $t < T$ (i.e.prior to maturity).

Puts also have upper bounds. Again considering the probabilistic nature of the payoff expression above, there is a chance with an associated probability measure that the payoff may result in a payoff unknown till the expiry date. Therefore, given that $S_T \geq 0$, we known that $K - S_T \leq K$ and hence:

$$p \leq K$$

as at maturity. Thus, at time $t < T$ (i.e. prior to maturity) the bound is:

$$p \leq K \exp -r(T-t)$$

where r is the risk-free rate that is earned on cash deposits i.e. the short rate.

Relative Bounds - Put-Call Parity

There is very well known and easily derivable parity for European options known as the Put-Call parity. The derivation thereof is based on the law of 1 price, which is what replicating portfolio valuation techniques is fundamentally built on (Hull, 2012). The law of one price states:

If two portfolios have equal payoffs across time, then in an arbitrage-free world they will have equal prices. The law of one price is based on explicit and implicit assumptions based on the underlying asset set under consideration for example, it implicitly assumes that market liquidity is sufficient and equal under various market climates.

The derivation of the Put-Call parity for a non-dividend paying stock is as follows; Consider two portfolios:

1. Portfolio A (PF_A): 1 call option$+Ke^{-r(T-t)}$ units of cash, and;
2. Portfolio B (PF_B): 1 put option$+(S_t)$.

Therefore, at maturity (time T) the following is true:

$$PF_A = max(S_T - K, 0) + K = max(S_T - K + K, K) = max(S_T, K)$$

and;

$$PF_B = max(K - S_T, 0) = max(K - S_T + S_T, S_T) = max(S_T, K)$$

Hence, the two portfolios have an equal value at maturity T. By the law of one price these portfolios have equal value at inception and hence:

$$c + Ke^{-r(T-t)} = p + S_t$$

The above equation is the Put-Call parity. The Put-Call parity is easily adapted to allow for a dividend yielding underlying asset (yielding a rate of q dividends continuously compounded):

$$c + Ke^{-r(T-t)} = p + S_t e^{-q(T-t)}$$

Intuitively the RHS is interpreted as holding $S_t e^{-q(T-t)}$ of the stock and buying additional stock with the yielded dividend so that S_T stock is held at time T.

Additional relative bound statements can be derived for the relationship between the price of European and American stock options. The basis of the derivation is embedded in the fact that American options provide equivalent and equal payoffs at maturity, with additional sample paths that provide profitable payoffs prior to maturity i.e. a greater number of sample paths (scenarios) are profitable, added to the fact that certain sample payoffs are greater on an absolute and discounted basis for prior to maturity exercise, driving an additional value premium for American options. Thus, in a very simple sense, European options are the basis of lower bounds for American options.

2.2.3 Black-Scholes-Merton

Notation and Assumptions

BSM is based on a general assumption set, that individually have some unrealistic basis, but collectively provide a reasonable framework to value options. The assumptions are summarised as follows (Hull, 2012):

(a) Constant drift and volatility of the underlying Wiener processes;

(b) Permitted short selling of securities;

(c) No transactions costs or taxes;

(d) No riskless arbitrage exists;

(e) Continuous security trading;

(f) Constant risk-free interest rate.

The notation is based broadly on the notation used by Hull, 2012:

(i) f = option price;

(ii) T = option maturity;

(iii) S_t = stock price at time t;

(iv) K = strike price;

(v) $F_t = S_t/K$;

(vi) r = risk-free rate (continuously compounded short rate);

(vii) q = dividend yield (continuously compounded);

(viii) σ = volatility of the underlying stock price, which is the standard deviation of the log price process.

Differential Equation

The BSM differential equation provides the basis for constructing a risk-free portfolio over a small time period (dt). The equation is derived using the Geometric Brownian Motion (GBM) diffusion process model and showing that a delta hedged portfolio has no stochastic term over a short time period, and hence is risk free. The delta hedged portfolio (π) consists of:

1. -1 (short) option;

2. $\frac{\partial f}{\partial S}$ shares;

where f is the valuation model, i.e. the BSM equation. Therefore, given this equation, riskless profits could in theory be made if given the valuation model is correct and the market is not perfectly priced at all times. The BSM Partial Differential Equation (PDE) that results is as follows:

$$rf = \frac{\partial f}{\partial t} + rS\frac{\partial f}{\partial S} + \frac{1}{2}\sigma^2\frac{\partial^2 f}{\partial S^2}$$

The derivation of the BSM formulae is based on risk-neutral valuation. This refers to measuring the value under a risk-neutral measure space. The above equation provides clarity that the portfolio return is the risk-free rate and does not depend on a specific expected return (commonly referred to as "alpha") of the security, making it indifferent to risk i.e. risk-neutral. An intuitive interpretation of this is; being indifferent to two investments with the same expected payoff but one posing downside risk (risk of negative returns) and the other not.

Girsanov's Theorem provides a framework where a measure space P can be shifted to find a measure Q such that a Standard Brownian Motion (SBM) process becomes a martingale under Q with certain, generally non-restrictive, bound conditions. The intuition behind this is shifting the real-world measure to a risk-neutral measure space where the discounted payoff process is a martingale (Hull, 2012). Therefore, valuation on the discounted process naturally follows from the properties of martingales which are particularly useful for predicting a certain time-dependent process, given a filtration for the process.

Based on the GBM representation of the underlying process and Girsanov's Theorem, BSM was able to derive a closed-form solution for the price of European put and call options (Hull, 2012). The result of this is the infamous BSM option price formulae:

$$c = S_t e^{-q\times(T-t)}N(d_1) - Ke^{-r\times(T-t)}N(d_2)$$

and

$$p = Ke^{-r\times(T-t)}N(-d_2) - S_t e^{-q\times(T-t)}N(-d_1)$$

where $N()$ is described as the cumulative standard normal distribution function and

$$d_1 = \frac{\ln S_t/K + (r - q + \frac{1}{2}\times\sigma^2)\times(T-t)}{\sqrt{T-t}\times\sigma}$$

and

$$d_2 = d_1 - \sigma\times\sqrt{T-t}.$$

2.3 Option Pricing with Neural Networks

The option pricing problem using NNs has been studied widely over the past few decades, and specifically the most recent decade. The approach to pricing options has been undertaken in two ways; direct pricing or indirect via determining the implied volatility. The former is a very natural approach and allows direct comparison to the outputs of traditional models as well as the price observations on which the model is trained. The latter provides meaningful insights that arguably stretches beyond only the valuation of options. In order to understand this, we consider the notion of implied volatility.

Implied volatility, is the volatility of the log-return process of the underlying asset that would produce the observed market prices via the BSM model or a more general model (Hull, 2012). Implied volatility provides insight not only on how the market values certain assets (relative view), but also how the market anticipates major moves and the probabilistic environment within which option prices can move, i.e. from out-the-money to in-the-money. Empirically, it is evident that the implied volatility on puts are wider than on calls, and has a faster rate of change and convexity (second order rate of change) component. A theory behind this comes from the magnitude and velocity at which sell-offs and bear markets take place as opposed to bull markets. This is because of the inherent nature of "fear trades", portfolio protection and deleveraging as opposed to taking on positive economic exposure (Hull, 2012). Intuitively, investors are far more averse to losses as opposed to missing out on gains (Hull, 2012). Volatility smiles, being a plot of volatility against strike, substantiates this notion, where the volatility is higher for out-the-money options than at-the-money. This presents the "smile", and added to this is the skew (to the downside) effect, referred to as the volatility skew (Hull, 2012). Thus, implied volatility distributions present themselves as negatively skewed distributions.

The implied volatility solution is an inherently founded numerical procedure, drawing its parallel and benefit to NN models. Alternatively stated, there exists no practically used closed-form solution to solve the implied volatility using the BSM model, and hence computational based iterative approaches like Newton-Rhapson are used to solve for it (Hull, 2012). Implied volatility is however highly interpretable, and provides significant information not only on market behaviour and pricing of options by the market, but also in the market of the underlying as well, as it directly ties to a log-return model of the underlying. The VIX index, published by the CBOE, is and index that track the implied volatility on a range of 30 day expiry call and put options on the S&P500 index (Hull, 2012).

Volatility is known to be an inherently heteroskedastic process, which essentially means that the distribution of the volatility changes as the level of the volatility changes (Hull, 2012). Intuitively, stock price volatility is generally higher when the level of markets are higher, more specific to the downside. There are well-developed and studied model classes that perform generally well at modelling volatility. Well known classes are; Exponentially Weighted Moving Average (EWMA) and Generalised Autoregressive Conditional Heteroskedasticity (GARCH) (Hull, 2012). GARCH models are generally considered to overcome the non-constant and stochastic behaviour of volatility itself in a typical time-series fashion (Hull, 2012).

2.3.1 Indirect Pricing: Learning the Implied Volatility

Various experiments were conducted on "learning" volatility that underlies the process of the option values. Volatility has a very natural interpretation in the context of market activity and out right or absolute risk assessment i.e. not drawing specific references to a particular liability set.

The exercise of supervising the implied volatility requires an additional step to derive the option price, i.e. using a function to map the volatility (NN output) to an option price. BSM option formulae, for instance, would map the implied volatility that is "learned" to the observed price by definition. Alternative pricing formulae can also be used to map the volatility prediction to an option price, where the implied volatility definition is adapted to the alternate model instead of BSM. Thus, predicting implied volatility under a closed-form framework, allows the option valuation to be predicted.

Haussamer, 2018 and Mavuso and Singh, 2017 conducted indirect pricing. The important components of their NN configurations are set out below, for each respectively:

(a) 3 vs 4 hidden layers;

(b) 32 vs 64 neurons per each hidden layer;

(c) Softmax and Relu, vs Sigmoid activation functions;

(d) MSE loss function;

(e) $[S_t/K, T-t, r, q, \sigma]$ as input features;

(f) Dropouts of 25%;

(g) Hyperparameter optimisation via grid search;

(h) 302k vs 63k observations;

Haussamer, 2018 found promising results for their NN to effectively learn the implied volatility parameters, which in turn allows an option price solution. Moreover, the limitations of the NN can be assessed by viewing the performance of the model when tested with observations that move outside of the input feature training bounds, specifically considering outcomes when the stock price parameter is allowed to vary more on the downside (Haussamer, 2018). This is analogous to stress testing the model and assessing the tail risks, such as significantly large market movements as seen in March 2020. Haussamer, 2018 further explored the changing of the process parameters dynamic by specifically considering the impact of varying correlation and volatility of volatility parameters in the Heston model that was being learnt in their experiment. Though the NN still performed relatively well when single input bounds were breached, Haussamer, 2018 found notable degeneration of performance when several input bounds were breached and allowed to vary. Haussamer, 2018 concluded that the implication in a practical light would be a re-training/calibration need once assumed bounds are neared are protruded.

2.3.2 Direct Pricing: Learning the Option Price

Bennell and Sutcliffe, 2004, Hutchinson, Lo, and Poggio, 1994, Culkin, 2017, Anders, Korn, and Schmitt, 1998 and Lajbcyguer, 1997 approached the problem by means of direct pricing. They approached the pricing exercise by essentially using the same inputs as the BSM formulae uses, with some simplifying assumptions. Hutchinson, Lo, and Poggio, 1994 for example assumed that r and σ are fixed throughout samples, implicitly also assuming that the simulated data related to a single underlying asset of the call option prices. Bennell and Sutcliffe, 2004 used several different BSM input combinations and considered the performance under different measures of performance, namely; R^2, Mean Error, Mean Proportionate Error, Mean Squared Error and Mean Absolute Error. These error measures were also typically used by the other authors, with some focusing more specifically on one or two error measures as opposed to all of them as Bennell and Sutcliffe, 2004 did.

Referencing to the Universal Approximation Theorem (UAT), which is thoroughly discussed in the next section, the true option pricing function f, is to be learned by the NN. Hutchinson, Lo, and Poggio, 1994 notes that most of their learning networks possess some form of the UAT. Culkin, 2017 and Hutchinson, Lo, and Poggio, 1994 as initial experiments showed that a NN was capable of learning the BSM option pricing formulae for a single instrument i.e. call options. Hutchinson, Lo, and Poggio, 1994 however notes that the UAT results provide no indication on the ease and computational efficiency at which NN achieve good approximations, and to what extent do NNs suffer from the *curse of dimensionality*, that is, does the need for input data grow exponentially as the complexity of phenomena at hand increases.

Hutchinson, Lo, and Poggio, 1994 used three NN frameworks (learning networks as they refer to it) to perform their experiments, namely; Radial Basis Function (RBF) network , Multi Layer Perceptron (MLP) network and Projection Pursuit Regression (PPR). PPRs differ from MLPs (which is the typical and well-known NN framework) primarily in that the non-linear activation functions are also parameterised during learning (Hutchinson, Lo, and Poggio, 1994). Hutchinson, Lo, and Poggio, 1994 does not provide specific information on the hyperparameters they used in their experiments. Culkin, 2017, on the other hand, used a basic feed-forward NN (inherently an MLP), the components of their NN configuration are as follows:

(a) 4 hidden layers and hence 6 layers in total;

(b) 100 neurons per each hidden layer;

(c) Elu, Relu and Leaky Relu activation functions and exponential function on the output;

(d) MSE loss function;

(e) 80:20 training:testing (20 % cross-validation set) data split, with 300k rows in the entire data set i.e. 240k training examples and 80k testing examples;

(f) $[S_t/K, T-t, r, q, \sigma]$ as input features;

(g) Dropouts of 25%;

(h) C/S_t was predicted;

(i) Approximately 31k parameters was use.

Notably, σ is the only feature that is used in BSM that is not directly observable. Culkin, 2017 simulated a data set using ranges for each of the features, and hence the volatility that was assumed in the simulated data, where the dependent variables/observations are generated by the BSM formulae, was used as implied volatility. Additionally, Bennell and Sutcliffe, 2004, Hutchinson, Lo, and Poggio, 1994, Culkin, 2017, Anders, Korn, and Schmitt, 1998 and Lajbcyguer, 1997 used S_t/K as an input, which can be thought of as a kind of normalisation exercise, expressing the underlying in units of strike. Bennell and Sutcliffe, 2004 used different combinations of the BSM inputs in their NN configuration. Anders, Korn, and Schmitt, 1998 and Bennell and Sutcliffe, 2004 both found that using S_t/K and S_t as seperate inputs yields no benefit in prediction performance. Bennell and Sutcliffe, 2004 also found that using S_t and K as independent inputs as opposed to S_t/K produces worse prediction performance.

Anders, Korn, and Schmitt, 1998 approached the selection of their NN configuration in a step-wise fashion using significance tests to select the best one. This resulted in a more parsimonious configuration, with the following components:

(a) 1 hidden layer and hence 3 layers in total;

(b) 100 neurons per each hidden layer;

(c) Tanh activation function;

(d) MSE loss function;

(e) $[S_t/K, T-t, r, q, \sigma]$ as input features;

(f) C/S_t was predicted;

Anders, Korn, and Schmitt, 1998 filtered out data for which time to expiration was less than 15 days, and option data that was out-the-money. The definition of out-the-money was set as; $S_t/K \leq 0.85$ and $S_t/K1.15$.

Bennell and Sutcliffe, 2004 also derived their NN configuration in an additive fashion. They however leveraged past literature, like Hutchinson, Lo, and Poggio, 1994 and Anders, Korn, and Schmitt, 1998 to inform the number of hidden layers to use, being 1 hidden layer. Therefore, they only focused on the numbers of neurons in their hidden layer, starting with two and incrementally increasing this until there was no improvement in the out-of-sample (cross-validated) error. Hutchinson, Lo, and Poggio, 1994 and Bennell and Sutcliffe, 2004 both used in-line weight updates, as opposed to batch training, whereby weights are updated after a batch error is computed to inform that weight parameter update. It is worth noting at this point that the availability of processing power was significantly less at the time when these authors conducted their experiments.

Moreover, Hutchinson, Lo, and Poggio, 1994 laid emphasis on not only the prediction accuracy of their NNs as a performance measure, but also the idea of being able to hedge option's using a replicating portfolio. Thus, Hutchinson, Lo, and Poggio, 1994 considered tracking error over 500 simulated sample paths, whereby a replicating portfolio is assessed against the models option valuation. Average sample daily tracking errors were computed across a six-month time period of the simulated sample paths, therefore generating a distribution of average sample tracking errors. An important phenomenon that Hutchinson, Lo, and Poggio, 1994 makes clear when measuring hedging performance is that the BSM model is a continuous time model, and therefore, at discrete instances, tracking errors will emerge. By contrast, training data for NN applications is sampled at discrete intervals, and thus it can be expected that such trained models can show better hedging performance even though you are hedging an option which is theoretically valued using BSM. This is exactly what was elucidated in the experimental results presented by Hutchinson, Lo, and Poggio, 1994. Thus, such non-parametric approaches may offer significant practical and institutional benefits, as financial reporting and performance reviews take place at discrete time steps in the real-world (Hutchinson, Lo, and Poggio, 1994).

Anders, Korn, and Schmitt, 1998 conducted similar research to Hutchinson, Lo, and Poggio, 1994 relating to assessing the hedging performance, but extended this into further "Greeks", namely; γ and θ. Both Anders, Korn, and Schmitt, 1998 and Hutchinson, Lo, and Poggio, 1994 assessed hedging performance against BSM being the "true" price. This was done by holding r and σ constant when simulating data. Anders, Korn, and Schmitt, 1998 found that the NN model was capable of producing realistic hedging parameters, but suggested that further research must consider the actual hedging performance of the NN using their derived hedging parameters.

Lajbcyguer, 1997, Hutchinson, Lo, and Poggio, 1994 and Anders, Korn, and Schmitt, 1998 further demonstrated that the NN application can be extended to learn "real-world" pricing dynamics. This contrasts with risk-neutral pricing in that risk-neutral pricing uses an assumed volatility that would be consistent with the underlying asset in a risk-neutral world, whereas in the real-world, the volatility would differ. Implied volatility acts as the bridge between the risk-neutral and real-world since the implied volatility is the input volatility that sets the risk-neutral derived pricing model equal to the real-world prices.

Lajbcyguer, 1997 extended their learning experiment on Australian Stock Exchange call options data on the SPI Index Futures. They used the same BSM feature set, akin to Culkin, 2017, however, they "learned" real-world prices. Contrasting with the above experiment where BSM was being "learned". Lajbcyguer, 1997 therefore had to produce an input volatility measure to train the NN. They used a weighted implied volatility as the input volatility as opposed to using a time-series based approach or a volatility that minimises the option pricing error. Lajbcyguer, 1997 describes their reasoning for using the weighted approach because of illiquidity (relative) in certain option markets, with few transactions taking place. Furthermore, notable mean reversion in the volatility was observed, which is easily gauged by viewing the VIX historic chart, coupled with major random movements within wide bid-offer ranges, causing noise. It therefore mitigates biases in the pricing and minimises the impact that noise can have on the model (Lajbcyguer, 1997).

Lajbcyguer, 1997 used a hybrid model approach consisting of a NN model and a modified BSM model, where the outputs were differenced and the residual of the models was assessed. Their NN configuration is vaguely discussed. They undertook had the following hyperparameters:

(a) 3 Hidden layers with 15 neurons in each layer,

(b) 20 % cross-validation set (as well).

An astonishing finding was, akin to similar research conducted on the S&P500 options, near identical residual surfaces resulted, characterised by a ridge at the $S_t/K \approx 1$ level, where the residual space has different signs on either side of the ridge. What this exhibits is that option markets have a non-uniform pricing bias on in and out-the-money calls, supplementing the volatility skew that is empirically validated (Lajbcyguer, 1997). This relates back to "fear trades" and/or de-leveraging events and how the implied volatility distribution is skewed to make allowance for this.

Hutchinson, Lo, and Poggio, 1994 performed the real-world experiment on call options on the S&P500 futures using the hedging tracking error view as a key performance metric. The data set comprised daily closing prices of the futures and futures options from January 1987 to December 1991. Hutchinson, Lo, and Poggio, 1994 partitioned the data into in, near and out-the-money options, and segmented the data further into 10 non-overlapping 6 month periods, thereby training different models over each time period and for each "moneyness" category (in, near and out-the-money). Hutchinson, Lo, and Poggio, 1994 estimated the r and σ parameters by using 3-month Treasuries and a 60 day realised volatility sample, respectively. It was evident that the NNs outperformed BSM in general, and particularly with regards to hedging performance (Hutchinson, Lo, and Poggio, 1994). A very interesting result was the fact that the notorious "Black Monday" crash happened 19 October 1987 (included in the data set used by Hutchinson, Lo, and Poggio, 1994), where the NN models performed better than BSM (Hutchinson, Lo, and Poggio, 1994).

Anders, Korn, and Schmitt, 1998 conducted their real-world experiment on DAX 30 call options that extended up to 9 month expiries. It was clear that the NN models outperformed BSM. Similar to Hutchinson, Lo, and Poggio, 1994, Anders, Korn, and Schmitt, 1998 also had to estimate the r and σ. This was undertaken using interpolation between the overnight, 1, 3, 6 and 12 month lending rates and using the VDAX index, respectively. The VDAX index is a weighted average of volatilities implied by different DAX options, analogous to the VIX index in some ways. Anders, Korn, and Schmitt, 1998 further used BSM residuals, similar to Lajbcyguer, 1997, on the DAX options data as an input into a NN, along with S_t/K and $T-t$ which proved to have as good performance as using the normal BSM inputs into the NN. This approach boasts the merits of simplicity, computational efficiency and ease of application (Anders, Korn, and Schmitt, 1998).

By contrast, Bennell and Sutcliffe, 2004 conducted their experiment on real-world data only, as opposed to training a NN to learn BSM. Instead, they emphasised the comparative performance of BSM and their set of NNs (varying only by different inputs/explanatory variables) on the real-world options data. Bennell and Sutcliffe, 2004 used additional inputs that deviate from the traditional BSM inputs, of which some are a combination of BSM inputs i.e. $\sigma \times \sqrt{t}$. Two inputs that Bennell and Sutcliffe, 2004 used, unrelated to the BSM inputs, were trading volume and open-interest. Their data set comprised of European call options on the FTSE 100 index. They used daily-closing values over the period of January 1998 to March 1999. Bennell and Sutcliffe, 2004 also partitioned their data by "moneyness" category, using in-the-money ($S_t/K < 1$) and out-the-money ($S_t/K \geq 1$) only. Furthermore, the authors created a restricted and unrestricted data set, where $S_t/K > 1.15$ (deep in-the-money) and 200 or more days till maturity is excluded in the restricted data set (Bennell and Sutcliffe, 2004). The results showed that the NN had significantly better results for the restricted data set, and particularly out-the-money options, while they had largely comparative performance on the unrestricted data set. Hence, partitioning data and models by means of "moneyness" seems to be generally appropriate for the option pricing problem, given the broad and heterogeneous ranges of prices that options from different "moneyness" partitions are likely to realise.

Additionally, Bennell and Sutcliffe, 2004 points out a large offsetting effect of the BSM model when considering the error measures, therefore indicating larger error tails compared to the NN models. This was clearly portrayed in the unrestricted data set results when viewing the mean error and mean proportionate error metrics (Bennell and Sutcliffe, 2004). Another interesting result was that the NN models had notable difficulty with deep in-the-money options with long terms to expiry, with a bias appearing on the over-pricing end (Bennell and Sutcliffe, 2004).

Finally, BSM and probably many other closed-form models appear to be more susceptible to bias in the data, while NNs have a seeming advantage in this sense because they have the ability to learn and hence overcome such empirical biases (Bennell and Sutcliffe, 2004). This notion is possibly supported by the results seen by Bennell and Sutcliffe, 2004 where BSM appeared to be more accurate for in-the-money options. Overall, NNs consistently showed tremendous promise across several studies in pricing options.

2.4 Diffusion Processes, State-Dependent Volatility and Kurtosis

The GBM that underlies the BSM framework is a continuous process describing the asset price movement over time (Hull, 2012). The assumption of normality in stock price returns has been empirically disputed, whereby skewness and thicker tails typically exist in the real-world processes of stock price returns (Pienaar, 2016). The typical GBM process is described by:

$$dX_t = (\mu - \frac{1}{2}\sigma^2)dt + \sigma_t dB_t.$$

Pienaar, 2016 further discusses how the disparity of the normality assumption is in part overcome by using stochastic volatility models. This is achieved by representing the volatility parameter (σ_t) by its own diffusion process. Thus yielding a more general expression of the above process (Pienaar, 2016):

$$dX_t = (\mu - \frac{1}{2}\sigma_t^2)dt + \sigma_t dB_t^{(1)},$$

where

$$d\sigma_t^2 = a(\sigma_t^2, t)dt + b(\sigma_t^2, t)dB_t^{(2)}.$$

Real-world processes are often affected by random shocks, which adds exogenous noise to the stock price process. Pienaar, 2016 produced an interesting analysis by considering the empirically estimated kurtosis (rolling) on the S&P500 and its differenced (removing trend) plot over time. When Pienaar, 2016 compared it against the kurtosis threshold under normality, it was often far exceeded. This concludes empirically how stock price processes fundamentally have much larger dispersion and hence tail sizes, than the normality assumption entails. Albeit that the normality of GBMs allow it to be generally well centered and hence capable of predicting well on a first moment measure statistic such as expected value, it is vulnerable to severe degeneration in accuracy due to failing to capture the higher moments of the underlying process (Pienaar, 2016). Thus, it is susceptible to largely understate the area and length under the tail of the true underlying process. This compromises the practical use case of using GBM and related diffusion processes, since higher moments of volatility captures severe financial risk and systemic risk, historically seen during insolvency and liquidity events.

BSM therefore has a fundamental weakness, and so does any model that is trained and configured on data that is; populated disproportionately by "normal" volatility regimes (in terms of skewness and kurtosis) and lacks sufficient data sampled from higher volatility regimes (in terms of skewness and kurtosis).

Barucci, Cherubini, and Landi, 1997 made reference to exactly this when they considered the option pricing problem. In contrast to the above diffusion processes, they considered state-dependent volatility, where the volatility experienced is influenced by the level of the underlying asset, speaking to heteroskedasticity. This is accomplished by making the volatility parameter a function of the state-variable (underlying asset price) (Barucci, Cherubini, and Landi, 1997). Thus, the BSM PDE is expressed with a functional volatility term:

$$\sigma(S) = k_i \sigma^2,$$

for $S_{i-1} < S \leq S_i$ and $i = 1, ..., N+1$, where k_i and σ are constants, yielding a non-parametric, step-wise representation of state-dependent volatility. Therefore, weighted residual method, i.e. Galerkin method, can be used to solve the resulting PDE (Barucci, Cherubini, and Landi, 1997). Barucci, Cherubini, and Landi, 1997 goes further to provide practical methods to derive the implicit probability distributions of option pricing, under the constraints of option bounds and the Put-Call parity, making use of traded instrument strategies like calendar spreads that form a component of the Put-Call parity to derive a kernel functional that spans the market valuation process.

Thus, Barucci, Cherubini, and Landi, 1997 presents a method to solve for the option price process PDE by means of trial functions and basis expansion, in which a functional form for the stochastic volatility term is solved under constraints, effectively capable of describing temporal volatility structure and so it can present a more empirically realistic input into a NN to learn option pricing.

In a more general sense, a NN trained on real-world data, would by design, implicitly capture economic constraints like upper bounds and the Put-Call parity. However, since explicit allowance for these constraints are not allowed, such a model would not only severely suffer from prediction performance degeneration when the data generating process (i.e., stock prices) fundamentally changes, but would also be susceptible to an overall degeneration of constraint consistency. Thus, an approach like that presented by Barucci, Cherubini, and Landi, 1997 may provide remarkable performance, founded on a more realistic, but complex basis.

2.5 Closing

In this dissertation, we endeavour to construct a more robust NN model, which can take various and more continuous ranges of maturities and "moneyness". Preemptively, this will require more iterations in training. The number of parameters used however would not necessarily have to increased substantially compared to the studies cited above.

Moreover, the UAT implicitly assumes that target functions exists and is continuous, which in such a financial application may be a more unrealistic assumption than those that underlie a framework like BSM (Culkin, 2017). In particular, a target function that could exist, is likely to be non-stationary in nature, that is, time inhomogeneous. Therefore, it is important to emphasise that using NNs for this financial application is likely to be presented with many caveats in terms of the target function that is approximated. These caveats relate mostly to the susceptibility to have degenerative performance at a quicker rate than another NN application typically would. Nonetheless, bearing this in mind, the NN configuration will be optimised to best conduct the experiments. Hutchinson, Lo, and Poggio, 1994 and Anders, Korn, and Schmitt, 1998 also describe the shortcomings of the UAT, however, they proceeded to conduct their research.

The number of parameters and regularisation used to minimise and prioritise parsimony in terms of using the least number of parameters was not particularly investigated or studied in the studies cited above. In this study, some attention will be paid to this. The hyperparameter specification that will be derived will have some emphasis on parsimony in terms of the number of parameters that are used to configure the NN. An optimal, but certainly not unique NN configuration that results, will be used throughout the remaining experiments.

More importantly, a NN will be trained to allow for economic constraints described in Section 2.2. It will then be assessed against an equivalent NN model that was not trained to enforce economic constraints. To investigate and assess this, the test set (out-of-sample data) will be stressed, i.e. more testing data will be sampled from the tail of underlying data generating process. This is discussed at length in Section 5.5-5.7. The next section, Section 3, sets out the specific mechanics and literature of NNs.

Chapter 3

Neural Networks

The mobilisation of computational hardware, open-source statistical packages and programming interfaces as well as a surge in freely available information and data have facilitated the major progress in the field of machine learning (Goodfellow and Courville, 2016). When a machine learns, it is inherently dependent on sufficient computing and processing speed. Historically, a single core processing unit (CPU) on a machine was used to train NNs. More recently, graphic processing unit (GPU) and networked CPU's are used to train models at a significantly faster rate (Goodfellow and Courville, 2016). This is because the machine requires many iterations of feeding data through the model and computing a measure of error for each iteration. This in turn allows the machine to learn from their error, and update their model parameters as it learns to classify or predict the phenomena it is set up to model. A typical machine learning model will have a set of hyperparameters and parameters. The parameters are typically calibrated or "learned" during training. The hyperparameters on the other hand relate to the model form and type i.e. the number of parameters used, the measure of error, how data is pre-processed, the optimisation method used e.g. backpropagation, Newton-Rhapson etc.

Neural Networks requisites a set of user-defined hyperparameters, that need to be set before the model can be trained. Some of these consider the type of data and phenomena that is being studied, while others are concerned with the amount and density of the data (Goodfellow and Courville, 2016). The practice of using NNs has a component of theoretical and mathematical rigour on the one side, and on the other it is accompanied by a vast spectrum of empirically driven concepts and practices. In a sense, NNs are a culmination of computational, data and statistical science.

3.1 Introduction to Neural Networks

There are several NN architectures that exist. The most basic and well known architecture is known as the Feedforward Neural Network (FNN). Wide ranging literature on the use of FNNs to predict or evaluate various stochastic financial processes can be found. In general, any NN model will be configured by a set of hyperparameteres and parameters. The hyperparameters pertain to the specific structure and design of the NN. Figure 3.1 illustrates the structure of a typical FNN.

Viewing Figure 3.1, the following can be noted:

(a) An input/design matrix, denoted X,

(b) The number of neurons in the input layer is the number of features (explanatory variables) present in the X input array i.e. number of columns,

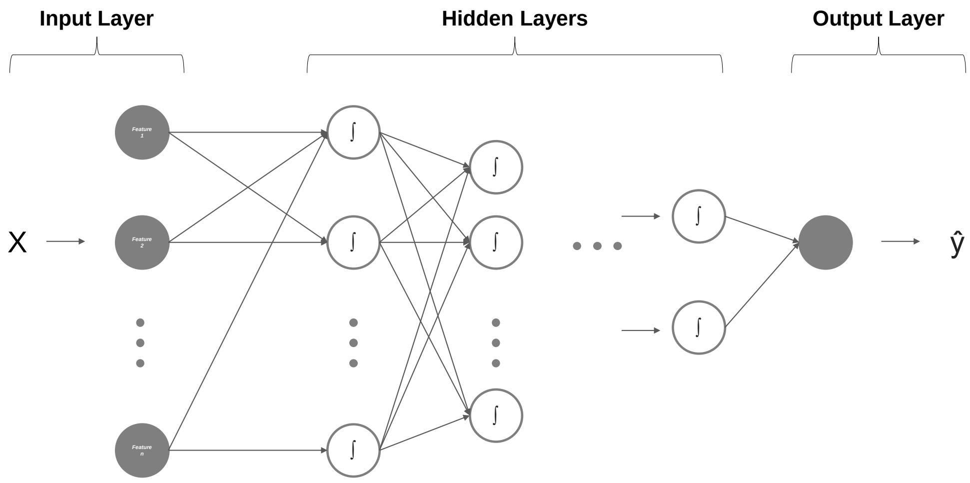

FIGURE 3.1: A simple Feedforward Neural Network

(c) The data frame X (observations by features) is passed through to the next layer by means of:

Multiplying it with a weight matrix;

Then adding a bias to each observation;

Then applying to it an activation function, which is a non-linear transformation,

(d) This is done for each layer till reaching the output layer,

(e) The output is assessed against the dependent variable and a cost (error term) is computed for each observation,

(f) Optimisation on the measure of error can then be undertaken i.e. minimise the error surface.

In general the hyperparameters refer to the design and configuration elements of a particular NN. The following would constitute the general categories of hyperparameters; Number of Layers, Number of Neurons in each layer, Cost Function, Activation Function, Epochs, Batch size, Cross-validation, Learning rate and gradient descent and Regularisation.

The performance of any NN is directly affected by the hyperparameters. Therefore, as part of the exercise of optimising NNs i.e. reducing the overall cost (total error term) in training and testing, an optimal hyperparameter set is sought. Parameters on the other hand falls into 2 general categories; weight parameters and bias parameters. The weight parameters are analogous to coefficients in regression models, and the bias parameters are analogous to the intercept in regression models.

The number of parameters that will have to be optimised is affected by the choice of the hyperparameters. Typically, gradient-based optimisation techniques are used to calibrate the parameters of a NN. Backpropagation is used to calculate explicitly (as opposed to numerically) the partial derivatives of the objective function w.r.t. the parameters of the network (Goodfellow and Courville, 2016). In its simplest terms, backpropagation is the process by which an error term is passed ("propagated") backwards through the network to calculate the derivative of the cost function w.r.t. each parameter in the network, hence the term back propagation. The derivatives are then useful to perform a procedure known as gradient descent, whereby the gradient informs the shape of the cost function, and hence the direction in which parameters should be altered to move towards a lower point on the cost surface (Goodfellow and Courville, 2016).

3.2 Parameters

The nodes exhibited in Figure 3.1 are called neurons. Between the neurons, connections are drawn, which is numerically represented by the values of the weight parameters. The weights are what is "learned" or calibrated during training of the NN. The learning is facilitated by the data, i.e. the model learns from the data it is presented with. The data is parsed forward through the network, from one layer to the next by means of matrix multiplication. Thus, the input layer has the neurons representing the features (explanatory variables) of the data set i.e. each feature is represented by a neuron. After passing the data from one layer to the next, beginning with the input layer, a bias is added to each observation of each neuron. This means that a vector of bias variables is added to each observation, where a particular observation represents a row vector i.e. 1 observation contains an observation of each and every layer neuron. The weights and biases described herewith represent the NN parameters. The mathematical mechanics that underlie the representation and calibration of the NN parameters is discussed in the next chapter.

3.2.1 The Universal Approximation Theorem

The Universal Approximation Theorem (UAT) states (Haussamer, 2018):
For I_{n_1} an $n_1 - dimensional$ unit hypercube space $[0,1]^{n_1}$, and $C(I_{n_1})$ a space of continuous functions on I_{n_1}, then for $f \in C(I_{n_1})$ and $\epsilon > 0 \exists n_2$,an integer, and $a_i, w^l_{ij} \in \mathbb{R}$ with $i \in [1, n_1]$ and $j \in [1, n_2]$ such that:

$$|f_N(X) - f(X)| < \epsilon,$$

Where $f_N(X)$ is an approximation of f(X);

$$f_N(X) = \sum_{i=1}^{n_2} a_i \times g(\sum_{j=1}^{n_1} w^l_{ij} \times x_j + w^l_{i0}),$$

where g is a continuous, non-constant, bounded and monotonically increasing function $\forall X \in [0,1]^{n_1}$.This theorem has been proved using several frameworks and axioms.

In an intuitive sense, there is a true function $f(x)$ that maps the inputs to the true outputs, and the NN is trying to learn this function. The UAT provides a theoretical backing that the NN model, represented by $f_N(X)$, will converge to the true function with high certainty. $f_N(X)$ is a functional representation of the NN, and is widely cited across most literature on NNs. f, mapping the input space to an output space of a different dimension, is not a unique form, and is affected by the configuration and choice of hyperparameters. Additionally, what and how data is used to train f will affect its form. Alternatively stated, the parameter space of the NN is profoundly affected by the data and the hyperparameter set.

In general the following representations are used:

(a) $f_N(X)$ or f_N refers to the NN model;

(b) X refers to the input data array;

(c) $\hat{Y}$ refers to the output data array, i.e. $f_N(X) = \hat{Y}$.

Y in general refers to the dependent variable, i.e. what the predictions are supervised/ assessed against. The next chapter focuses on the mathematical representation of a NN and the mathematical dynamics of backpropogation, which is a common optimisation algorithm used to train NN models.

3.3 Hyperparameters

There are several hyperparameters that can be "improved" or optimised. The number of hidden layers and neurons within each layer is a common place to start when considering the hyperparameter set.

3.3.1 Number of Layers and Neurons

Grid Search

Many applications of hyperparameter optimisation use the technique of grid search to establish an optimal (or more optimal) hyperparameter set to use. This is done by constructing a grid, where different numbers of neurons per layer is on the horizontal, and the number of layers on the vertical. The practicality of the grid search depends on the size of the data set, particularly the number of observations. In this dissertation, the number of observations is less than 100000 and therefore grid search is a viable technique. The error/cost results for each row/column intersection populates the grid. This can by easily built into loop structures but can result in high relative computational costs. However, a grid can be set up in any way desired i.e. rows represent a different layers and neurons per each layer set, and an error/cost is populated for each.

Haussamer, 2018, Culkin, 2017 et. al. noted in their investigations that increasing the number of layers or neurons per layer, and hence the total number of parameters in the overall model does not necessarily improve the efficiency of the model. Efficiency in this sense would pertain to the number of iterations (or "runs"/"parses") the model requires to reach a certain cost level does not necessarily decrease as the layers are added or the average number of neurons increase.

Randomised Search

Another, but similar approach is commonly known as randomised search. The hyperparameters (layers and neurons per layer in this context) are allocated ranges, and from these, random point samples are drawn and the NN model performance is evaluated under this (Nguyen and Bui, 2020). It suggests a lower likelihood of bias and empirical ability to assess a wider range of combinations then the typical grid search technique. It however does face prospect of ranges being inappropriate by either being to wide and hence the more optimal values are under-investigated or to narrow and varying results with no improvement may follow (Nguyen and Bui, 2020).

Bayesian Hyperparameter Optimisation

More recent developments have found other, more theoretically motivated techniques to be useful at optimizing the hyperparameters. One of the most widely known and used of such techniques is known as practical Bayesian Hyperparameter Optimisation. The technique can be summarised as follows(Nguyen and Bui, 2020):

(a) $f(h)$ is a cost or performance measure of the NN, where h is the hyperparameter input,

(b) For H, a hyperparameter space, the following holds:

$H \sim G(.|\theta)$ i.e. the Gaussian assumption for the distribution of θ,

Thus $p(\theta|H) \propto G(.|\theta) \times q(\theta)$ is the Bayesian approach to determine the distribution of the hyperparameter space θ given an observation for it,

(c) This is then extended to F:

$F(p(\theta|H)) \sim N(m, K)$

(d) Therefore, F has a distributional form.

An acquisition function is determined, which is a function to inform how to sample parameter inputs from its updated posterior to achieve certain output ranges for F i.e. choose $\theta|H$ sample to increase the expectation or probability of F above a threshold etc. There are wide ranging applications with several empirical investigations of the form and selection of these acquisition functions and the effectiveness they have on improving the NNs performance.

3.3.2 Gradient Descent and Learning Rate

In Section 3.2 the UAT is put forward. The representation of ϵ is an error measure, akin to regression. This section is based on the general terminology used by Bottou, 2010. To formalize, the following is defined in the context of a single observation:

$$l(\hat{y}, y) = l(f_N, y) = \text{measure of } \epsilon.$$

Note that $l()$ here refers to a loss function and not a likelihood function. The goal is to seek out $f \in \mathcal{F}$, where $\mathcal{F} = C(I_{n_1})$, described above as being a space of functions on $[0,1]^{n_1}$, such that $l(\hat{y}, y)$ is minimised. If dP is a distribution that embodies the laws of nature exactly, then we seek to average the error computation over dP. The distribution of P is thought of as the ground truth distribution, is always unknown, and can vary in terms of complexity, making the measurement of the risk inevitably more difficult (Pan and Jiang, 2014). Therefore, one can only use an empirical approach and average the error over a sample of size n:

$$E(f) = \int l(f_N, y)dP$$

where $E(f)$ is called the expected risk and measures the generalised performance over different samples. dP is however unknown, and hence the true error can only be approximated. The empirical risk measure $E_n(f)$, can be known and used:

$$E_n(f) = \frac{1}{n}\sum_i l(f_N, y).$$

The empirical risk is minimised in general as opposed to the expected risk when $\mathcal{F}$ is considerably restrictive (Bottou, 2010).

Gradient descent earns its name from being an algorithm that uses the gradient of an error or cost surface w.r.t. a particular parameter to descend towards a minima i.e. minimize the error. Intuitively, it applies a rate scalar to the gradient to determine an alteration to make to that particular parameter. For example, if the gradient is positive, then the error curve w.r.t. a parameter in 2 dimensions is upward sloping, and hence moving leftwards (opposite direction to the gradient) implies descending to a lower error point. The lower point could be a saddle point, which is why the rate scalar should not be allowed to be to small, since this will circumvent getting "stuck" in a local minima. In words:

updated parameter value = current parameter value − rate scalar × gradient.

Formally stated, gradient descent is defined by(Bottou, 2010):

$$w_{t+1} = w_t - \gamma \times (\frac{1}{n}\sum_i l(f_N, y) \times \nabla_w)$$

where ∇_w defines the gradient of the error measure w.r.t. the particular and γ is the rate scalar known as the learning rate.

Gradient descent can be contrasted with the well-known maximum likelihood estimation techniques commonplace in linear regression with the enticing feature of closed-form parameter solutions as an alternative optimisation to framework then. Gradient descent is founded on an empirical basis, evident from the above expression. Stochastic gradient descent (SGD) is a version of gradient descent that has widely been used in the study of NNs. SGD is essentially an exercise of drawing random samples from the data, as opposed to using the entire data set, by which the above expression is applied, and so the empirical risk measure is based on a stochastic sample instead of the entire batch (Bottou, 2010). This technique is briefly discussed below, and in more depth further on where its contextualised in the regularisation framework.

3.3.3 Batches and Epochs

From the parameter section, it follows that passing a run through the network by means of various matrix multiplication, addition and transformation operations, an input is taken to produce an output. A single production of outputs on a data frame of observations by features is referred to as a forward parse through the network. Thereafter, a backward parse is exercised to update parameters, which constitutes propagating an error through the network retrospectively, which is commonly referred to as back propagation. A forward parse and backward parse together constitutes a parse through the network.

An epoch refers to the number of times that the learning algorithm works through the training data set (Roy, 2010). Alternatively stated, the number of times the full training data set is used to train the NN. The number of epochs is affected by the type of gradient descent used. Oftentimes, especially when the training data set is very large, batches are created (Roy, 2010). Batching is in essence the procedure of segmenting the training data into several smaller training data frames. The batch size is the number of observations/data-points in a batch and the number of batches is the number of sub data-frames created from training data (Roy, 2010). A batch is parsed through the model to determine a cost output and gradient update, be it as a single run or in several. A sample is taken from the batch and parsed through the network. A sample can be the entire batch, or a subset of the particular batch. The way in which samples are drawn from the batches generally defines the type of gradient descent that is deployed i.e. SGD. There are three general variants of gradient descent (Roy, 2010):

(i) Batch gradient descent (BGD)- size of batch = size of training data set;

(ii) Stochastic gradient descent (SGD) - size of batch = 1 sample;

(iii) Mini-batch gradient descent (MGD) - 1 sample $\leq$ size of batch $\leq$ size of training data set.

The total number of epochs may be fixed, where the efficiency of the model is then assessed on the basis of the accuracy achieved given the number of epochs. Alternatively, the efficiency can be assessed by assessing the number of epochs or total iterations required to a achieve a benchmark, pre-defined accuracy. This can be thought of in the context of a loop and how the stopping condition is stated i.e. a "for" loop versus a "do until" loop.

3.3.4 Cost Function

Calibration of a model requires a measure by which the prediction or classification error is assessed. This measure is then optimised by means of updating and changing the parameters of the model. Prediction type models are models that are set to predict continuous variables, where as classification tasks relate to modelling nominal or categorical responses (Pienaar, 2019).

NNs, having an interpretation as a convoluted series of logistic regressions in multi-dimensional spaces, has parallel optimisation practice to linear models and more broadly Generalised Linear Models. Regression is in general considered to be a supervised learning exercise. This relates to that the model outputs are supervised against a predictor, commonly referred to as the dependent variable, Y_i. The measure of error is often referred to as the cost or loss, and the relating function is referred to as the cost/loss function. It is important for the purpose of NNs and optimisation that the cost function is continuously differentiable on the output-parameter space. The reason is because this ensures that a partial derivatives of the cost function w.r.t. a particular parameter can be determined and used in the process of gradient descent described above. One of the most well-known error functions is called the Mean Squared Error (MSE) (and pervasively used in regression frameworks):

$$C_{MSE} = \frac{1}{N}\sum_i \frac{1}{2}(y_i - \hat{y}_i)^2$$

Where N equals the number of observations and y_i and $\hat{y}_i$ refer to 1-dimensional output features. Clearly, the C_{MSE} is a measure of distance between the model output and predictors. The MSE framework is easily extended to outputs of several features/variables Pienaar, 2019. It is notable that MSE is equivalent to maximum likelihood estimation in the instance where the error space is assumed to be normally distributed. Though this is a largely unrealistic approach, from the various transformations that take place in a NN via the applied activation functions, the use of the MSE on the output is reasonable and proven effective in many predication based applications. Another common loss function that is effectively used for prediction related tasks is the Mean Absolute Error (MAE) (Brownlee, 2019b):

$$C_{MAE} = \frac{1}{N}\sum_i \frac{1}{2}(|y_i - \hat{y}_i|)$$

This cost function assumes that the error space is Gaussian as well. It does however attribute relatively more cost ("weight") to very large absolute losses that would be "squared out" by the MSE, therefore being inherently more considerate of outliers (Brownlee, 2019b).

However, when faced with the aforementioned classification tasks, setting up a distance based measure cost is disadvantageous. The reason is, consider a prediction task with a single output dimension where $y \in (-\infty, \infty)$, then the MSE minimum will be approached from both sides, whereas consider a classification problem $y \in [0, 1]$, then the MSE for $\hat{y}$ will be approached from one side only and hence will form an asymptotic and monotonically convergent trend towards and past 1 from the left hand side. This phenomena has been described as the saturation problem (Pienaar, 2019). The Binary Cross-Entropy Error/Cost function was formulated to overcome this kind of problem. It is defined as:

$$C_{CEE} = -\sum_i (y_i log(\hat{y}_i) + (1 - y_i) log(1 - \hat{y}_i))$$

Several variants of the above functions exist, each posing benefits for different related tasks. In particular, multi-classification (as opposed to binary) cross entropy functions are useful if more then 2 categories exist for the classification task. The cross-entropy cost functions are generally useful in classification related problems(Brownlee, 2019b). The mean squared, absolute and other cost functions relate to prediction related tasks. Hinge loss functions are another class of cost functions, founded in Support Vector Machine practice, that is also commonly used for classification related problems(Brownlee, 2019b).

3.3.5 Activation Functions

The activation functions of a NN is sometimes also referred to as a transfer function. The activation function is a function that has an analogous purpose to the commonly known link functions in Generalised Linear Models. The choice of a specific activation function is often described as arbitrary (Pienaar, 2019). Different activation functions are often used for the hidden layers and the output layer. The output layer interfaces with the data via the cost function that is chosen to measure/penalize miss-prediction or miss-classification Pienaar, 2019.

Activation functions can be categorised into two groups; linear and non-linear Sharma, 2017. The non-linear activation functions provides the NN with ability to model non-linearity. The activation function is applied at each and every layer of the network, after a linear regression type application is applied at that layer. In other words, the neuron observations are taken at each layer, applied weights that maps it to the next layers neurons, then added a bias, which is then applied an activation function. Thereby, linear transformations from 1 layer to the next is made non-linear through the activation function, in order to capture non-linearity. The non-linear transformations in aggregate will have the ability to effectively model significant non-linearity in the underlying process.

In summary, the activation functions are used to convert linear input signals into non-linear output signals, which make possible the learning of high-order non-linearity and convexity in the underlying process. The activation functions have a requisite to be differential and continuous when using the back propagation algorithm i.e. being able to determine derivatives of all network parameters w.r.t. the cost function. The descriptions of the most common activation function frameworks can be found below each with their own relative merits Nwankpa and Marshall, 2018.

Sigmoid Function

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

where $g(x) \in (0,1) \forall x \in \mathbb{R}$.

The Sigmoid function is commonly referred to as the logistic function. The output values of the function lies in the unit range. Sharma, 2017 highlights a major drawback of the Sigmoid being that it experiences sharp damp gradients in deep layers of a network relative to the input layer, which causes gradient saturation, slow convergence and non-zero centered outputs. This speaks to what is known is the the vanishing gradient problem, whereby the relatively small output range of an activation function like the Sigmoid function have certain weights that are small when initialised with small error gradients, resulting in very small weight updates during training (Wang, 2019). This results in some weights slowly converging to zero, and inevitably gets stuck in this cycle whereby it adds no value.

Moreover, Sharma, 2017 points out that the Sigmoid activation function has the ability to cause a network to get "stuck" at training. This refers to descending into a local minima and lacking the ability to move out of the local minima further on. Variations of the typical Sigmoid set out above have been created to overcome some of the drawbacks, where some of the more well-known variations are; the Hard Sigmoid function, Sigmoid-Weighted Linear Units (Silu) and Derivatives of Sigmoid-Weighted Linear Units (dSilu) (Sharma, 2017).

Hyperbolic Tangent (Tanh) Function

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

where $g(x) \in (-1, 1) \forall x \in \mathbb{R}$.

The hyperbolic tangent function is described as "smoother" as opposed to the Sigmoid, and zero centered, as opposed to centered at 0.5. An issue that is particular to the Tanh activation function is that it has a gradient of 1 only when the input value, x, is zero (Nwankpa and Marshall, 2018). In other words, a neuron with a zero value, which is commonly referred to as a dead neuron, is rarely used during network training and computation. The Tanh function also suffers from not overcoming the vanishing gradient problem that the Sigmoid function also suffers from (Nwankpa and Marshall, 2018). Variations such as the HardTanh exist, which overcome some of the issues and prove to have greater computational efficiency for very deep networks. A common use of these are natural language processing applications (Nwankpa and Marshall, 2018). Sharma, 2017 mentions that Tanh is generally used for classification type problems where there are 2 output classes.

Softmax Function

$$g(x_i) = \frac{e^{x_i}}{\sum_j e^{x_j}}$$

where $g(x) \in (0, 1) \forall x \in \mathbb{R}$.

It is typically used in applications where there are multiple output classes(ie the final output has several values) or probabilistic outcomes i.e. a probability distribution is the output of interest (Nwankpa and Marshall, 2018).

Softsign Function

$$g(x) = \frac{x}{|x| + 1}$$

Where $g(x) \in [-1, 1] \forall x \in \mathbb{R}$.

The Softsign function is often described as a quadratic polynomial. It is compared to the Tanh function in that it converges in polynomial form, whereas the Tanh converges exponentially (Nwankpa and Marshall, 2018).

Rectified Linear Unit Function (ReLU)

$$g(x) = \begin{cases} x, & x \geq 0 \\ 0, & x < 0 \end{cases}$$

Where $g(x) \in [0, x] \forall x \in \mathbb{R}$.

The ReLU function boasts very attractive performance attributes such very fast relative calibration (convergence) performance. It is further described to have "sufficient linearity" to preserve linearity of the underlying process that may exist. Nwankpa and Marshall, 2018 points out that it performs a threshold operation to each input, which essentially refers to the fact that inputs are set to zero if they have a value less than 0.

The ReLU function commonly occurs in several NN studies that were conducted in a financial application. The ReLU has been described to be known to overcome the "dead neuron" and "vanishing gradient" problems suffered by Tanh and Sigmoid activation functions. The ReLU activation function also has several variants that have proved very useful for different applications Nwankpa and Marshall, 2018. Among these are the so called; Leaky ReLU, Parametric ReLU, Randomised ReLU and S-shaped ReLU. These variants came about by mostly making factor based or parametric adjustments to the threshold in the ReLU function.

Softplus Function

$$g(x) = log(1 + e^x)$$

where $g(x) \in (0, x] \forall x \in \mathbb{R}$.

The Softplus function is described as a special case of the ReLU, with mostly statistical applications Nwankpa and Marshall, 2018. It has shown to converge faster than the general ReLU and Sigmoid activation functions has in several past experiments (Nwankpa and Marshall, 2018).

Exponential Linear Unit Function (ELU)

$$g(x) = \begin{cases} x, & x > 0 \\ \alpha e^x - 1, & x \leq 0 \end{cases}$$

where $g(x) \in (-1, x] \forall x \in \mathbb{R}$

The ELU activation function is noted to have the major advantage of also overcoming the vanishing gradient problem Nwankpa and Marshall, 2018. The ELU is also said to result in relatively faster convergence when compared to most other activation functions. By having the possibility of negative outputs, the ELU activation function boasts the characteristic of being able to push the mean output closer to zero. Variants of the ELU also exist, such as the parametric and scaled ELU.

Maxout Function

$$g(x) = max(w_1 \times x + b_1, w_2 \times x + b_2)$$

Where $g(x) \in [0, x + max(b)] \forall x \in \mathbb{R}$ and where $max(b)$ is the maximum value that a bias can take.
The Maxout activation function is knownto apply non-linearity by outputting the dot product between weights and the data of a NN Nwankpa and Marshall, 2018. Therefore, it is certainly more computationally expensive, since the number of network parameters are doubled due to it.

Swish Function

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

Where $g(x) \in (-x, x) \forall x \in \mathbb{R}$

The swish function is described as one of the first compound activation functions. It is a product of the input neuron multiplied but the Sigmoid activation function. The function was proposed more recently in 2017 and it uses the automatic search technique to compute the function (Nwankpa and Marshall, 2018).

Exponential Linear Squashing (ELiSH) Function

$$g(x) = \begin{cases} x\frac{1}{1+e^{-x}}, & x \geq 0 \\ x\frac{e^{x}-1}{1+e^{-x}}, & x < 0 \end{cases}$$

Where $g(x) \in [-1, 1] \forall x \in \mathbb{R}$

The ELiSH activation function is an even more recent development than the Swish activation function, brought on in 2018 Nwankpa and Marshall, 2018. It is a combination of the ELU and Sigmoid function. It uses the ELU for positive inputs, and uses the Sigmoid derivative for negative inputs. Nwankpa and Marshall, 2018 states that Sigmoid part improves the information flow between layers while the ELU part, through linearity, overcomes the vanishing gradient problem in part Nwankpa and Marshall, 2018.

It is clear that there is a wide range of activation functions that are more and less useful for particular purposes. Based on the general literature that was reviewed it seems that empirical and practical based investigations were used in arriving at conclusions about the usefulness of each of these functions, with some basic parallels drawn to the general function forms and the monotonic nature and ranges of them and their derivatives. For regression like or continuous prediction problems, linear activation functions have been described to be a generally better choice for the output layer activation (Pienaar, 2019). For classification type problems, an activation function with an output range correspondent to the range of categories should be chosen (Pienaar, 2019). It is worth noting that Nwankpa and Marshall, 2018 tabulates general NN architecture with regards to activation functions, where the use of different activation's functions in the hidden layers and output layer is used. The ReLU function was a common choice for the hidden layer activation's, whereas the Softmax and Sigmoid was used in the output.

3.4 Regularisation

The term regularisation is often used in terms of statistical model and machine learning. Nguyen and Bui, 2018 point out that the definition is often loose, and largely insecure in the realm of deep learning, where it plays a crucial part in producing models that are capable of generalizing to predict well on out-of-sample data sets i.e. testing/validation data sets. Regularisation in a general sense is a technique used to penalize model complexity and ensure over-fitting is mitigated (Pienaar, 2019). Nguyen and Bui, 2018 have loosely defined regularisation as "any modification we make to a learning algorithm that is intended to reduce its test error but not its training error". This is clearly a broad and possibly subjective definition, that is not very clarifying in most senses. In a arguably seminal paper named "Regularisation for deep learning: A Taxonomy" by Nguyen and Bui, 2018, the authors set out to specifically define regularisation in the context of deep learning:
"Regularisation is any supplementary technique that aims at making the model generalize better, i.e. produce better results on the test set".

In their study they provide a structured narrative to holistically understand and define regularisation at the various stages of constructing and training a NN.

Nguyen and Bui, 2018 provides the following categories where regularisation is applied:

(a) Data:

  (i) A transformation is applied to the training data, generating an augmented data space,

  (ii) Some transformations would be feature extraction or pre-processing e.g. dimensionality reduction on the feature space using principal components analysis or some other unsupervised learning framework,

  (iii) Normalisation and/or standardisation is a common pre-processing framework in data-science, it ensures that features with relatively wide or large absolute value ranges are not over-emphasised in its use due to this difference in scale, but rather considered in its relative context.

(b) Network Architectures

  (i) The network architecture is represented by a function f, $f_w : x \to y | w \in W$,

  (ii) Then the set op mappings f_w can realize can be amended by making changes to f_w i.e. how inputs or transformed to outputs,

  (iii) Examples are weight sharing (reusing certain parameters at various intra-layers), activation functions, noisy models (injecting random noise within the network) etc.

(c) Error (Cost or Loss) Function

  (i) The cost function can be adapted to cater for the specific characteristics of the data and output,

  (ii) An example would be where there are class-imbalances and the overall cost function is set to sum over pairwise combinations.

(d) Error Term

   (i) An term commonly referred to as the "regulariser" term can be added to the cost function,
   (ii) This term serves as a mechanism to inscribe other properties that is independent to the data,
   (iii) In other words, these terms are functional without a labelled data point (dependent variable) to be supervised against,
   (iv) It provides inductive bias, which entails placing assumptions/restrictions on f_w's mapping ability other than consistency/spread of outputs with targets.

(e) Optimisation

   (i) Regularisation and optimisation are not separable in the context of deep learning,
   (ii) Optimisation and the cost function has a complicated joint interaction and is inherently complex in how the training initialisation and optimisation affects the parameters space of a network (Nguyen and Bui, 2018).

From the very holistic and structured taxonomy highlighted above and presented by Nguyen and Bui, 2018, focus will placed on three categories, namely; data, error term and optimisation.

3.4.1 Data Regularisation

The data regularisation is discussed in depth by Nguyen and Bui, 2018. The transformation of any data point is presented by a function:

$$r_\theta(x) = x + \theta,$$

Where θ can be stochastic or deterministic. The metrics of distinguishing between these are as follows:

(a) Dependence - θ 's conditional distribution dependence,

(b) Transformation space - Input features, hidden features or input and hidden features,

(c) Stochasticity (θ sampling) - Random or deterministic,

(d) Phase - during training or testing.

The various methods described present different appeals for different underlying processes. Normalisation in general relates to scaling dimensions (axes) of the data space so that all data points fall within a constrained hypercube space.

A well known method to normalize network input data is by Min-Max Normalisation. The following formulae describes this technique:

$$X_{norm} = \frac{X_i - X_{min}}{X_{max} - X_{min}}$$

Where X_{max} is the maximum value X_i can take, and X_{min} is the minimum. If the minimum and maximum of X is known with certainty, and therefore deterministic in nature, then no prior knowledge or assumptions of the distribution is required, and the normalisation will inherently be stable and consistent across the various epochs and partitioning of the data set (into training and testing sets).

Standardisation is a type of normalisation where the dimensions are shifted and scaled so that the data space plots normally around the origin and within the unit hypercube space. Standardisation, in its simplest form, is essentially producing the well known Z scores (Ibrahim, 2014). This however requires distributional properties of a the standardised space. Once axes of the data space is scaled such that all data points are within the unit range, the standardised space must have its axes shifted by the conditional mean of each dimension, and divided by the standard deviation of each i.e. standardizing the normally distributed data.

A more common and widely applicable technique is referred to as batch normalisation. In previous investigation it was shown that for the general case networks converge faster when the inputs are standardised, in other words, transformed to have a sample mean of 0 and sample standard deviation of 1 (Nguyen and Bui, 2018).

Ioffe and Szegedy, 2015 introduced the technique of batch normalisation whereby they describe the concept of Internal Covariate Shift (ICS). ICS refers to the changing distributional form of the aforementioned normalisation samples i.e. covariate shifting. The term batch refers to the fact that the mean and standard deviation used in standardizing the features are sample based. The batch itself is a sample of the training data. This occurs within the layers of the network, and hence the description of internal in ICS. Ioffe and Szegedy, 2015 define ICS as follows:

"Internal Covariate Shift is the change in the distribution of activation's due to the change in network parameters during training."

A batch in this context refers to a subset of the training data. In calibrating a network, stochastic or deterministic optimisation can be used. Stochastic optimisation refers to the process of sampling "batches" from the training data and parsing it through the network to the number of set epochs Ioffe and Szegedy, 2015. The sampling is random and hence the term stochastic optimisation is used to describe it. Deterministic optimisation on the other hand consists of using the full training data and parsing it through the network back and forth to calibrate it by optimizing the cost function.

A mini-batch refers to a smaller sample to draw from the training data that would essentially result in a faster forward and backward parse through the network. The mini-batch is a random sample by design and hence falls under stochastic gradient descent. Literature on using stochastic gradient descent is pervasive and it's benefit in terms of optimisation efficiency is near certain.

It is important to understand that normalisation can be undertaken at each layer within a network. Normalisation is affected by the specific sample, and hence the distribution of activation's changes at each batch or mini-batch. This dynamic has an adversarial impact on the optimisation procedure in the sense that the bias parameter updates at each layer are cancelled out. This is because the bias parameter change that is to be applied during gradient descent is cancelled out because the change is added and then subtracted again (Ioffe and Szegedy, 2015). This is due to the batch

mean subtracted in standardisation and hence the added bias to a particular activation is subtracted (Ioffe and Szegedy, 2015). Ioffe and Szegedy, 2015 notes that this is further exacerbated by the scaling impact in standardizing due to dividing by the sample standard deviation. This coupled with the frequent ICS dynamic described above results in the network calibration possibly diverging and being exposed to inefficiency and inaccuracy.
To overcome this phenomena, Ioffe and Szegedy, 2015 presents a method, referred to as batch normalisation. The procedure is best described mathematically, where the notation is consistent with that used byIoffe and Szegedy, 2015.

Consider a particular observation of any activation in a layer, a_i, which is essentially a transformed representation of the inputs.To standardize this, the following is performed:

$$\hat{a}_i = \frac{a_i - E(a_i)}{\sqrt{Var(a_i)}}$$

Where:

$$E(a_i) = \frac{1}{m} \times \sum_i a_i$$

$$Var(a_i) = \frac{1}{m} \sum_i (a_i - E(a_i))^2$$

And $m =$ mini-batch sample count, $E(a_i)$ and $Var(a_i)$ are the sample mean and variances, respectively, representing distributional characteristics for the activation's at a particular layer.

Therefore,

$$a_i = \hat{a}_i \times \sqrt{Var(a_i)} + E(a_i)$$

represents a function that maps the standardised activation's to its initial form. Clearly there is an impact of shifting (mean) and scaling (standard deviation) being applied here. In order to minimize the impact of the ICS and circumvent bias divergence, a general form of this mapping/linking function can be established and the parameters can be "learned" during optimisation. Consider:

$$B_{\gamma,\beta}(\hat{a}_i) = bn_i = \hat{a}_i \times \gamma + \beta$$

Where γ and β are parameters that serves an inverse standardisation function. These additional parameters are layer specific i.e. there is a batch normalisation function that applies to each layer with its own γ and β. The general function $B_{\gamma,\beta}(\hat{a}_i)$ is referred to as the Batch Normalizing Transform and is completely differentiable. Therefore, the cost function is differentiable w.r.t. each of these parameters, at a very small additional computational cost. Hence, these parameters are "learned" along with the other weight and bias parameters through out the process of backpropagation.

Thus, the exercise of performing batch normalisation can be described as follows:

(a) determine mini-batch mean,

(b) determine mini-batch variance,

(c) Standardize activation's,

(d) Calculate batch normalised activation's, bn_i:

  (i) γ is initialised with the first mini-batch standard deviation,
  (ii) β is initialised with the first mini-batch mean.

3.4.2 Cost Term Regularisation

Regularisation via the cost term is what is the more general interpretation of regularisation. Pienaar, 2019 describes this as applying the brakes on model complexity. This is essential to ensure that the model can generalize beyond the training data, being sufficiently capable of modelling the true nature of the underlying process without being "swayed" by regular random noise in the data. The cost function is directly used in optimizing the network and determining optimal parameters. Regularisation via the cost function relates to adding a "regulariser" term to the cost function.The purpose of this term is to penalize particular model configurations during training to circumvent over-fitting and undue complexity. Generally, regularisation by the cost function can be done through any of the following (Nguyen and Bui, 2018):

(a) f (the NN model),

(b) w and b (weights and biases),

(c) $\frac{\partial f}{\partial w}$ and $\frac{\partial f}{\partial b}$,

(d) $\frac{\partial f}{\partial x}$ (relates to the input features),

(e) $\frac{\partial C}{\partial x}$ (C is the collective error/cost/loss measure).

With the hyperparameter set and configuration held constant, the only way to penalize and alter the model behaviour is by means of "tweaking" the NN parameters (Pienaar, 2019). To achieve this, constraints can be placed on the parameters. This indirectly suffices as a mechanism to limit the model parameters being fitted to very particular random noise in the data set. These constraints allow us the ability to control the flexibility-complexity trade off and levels of the model (Pienaar, 2019). The following representation elucidates this phenomena more clearly, using regularisation via the w terms:

$$R(w) = (\sum_l |w^l|^p)^{1/p} = ||w^p||_p$$

Which is the well-known p-norm term. When p = 1, this is referred to as the L1 penalty, and when p = 2, this is referred to as the L2 penalty, which are frequently referred to as the Lasso and Ridge regulariser terms. The Lasso term takes the general shape of:

$$R(w) = \lambda \times \sum_j |w_j|$$

Clearly, Lasso penalizes the absolute value of any weight and can configure a weight to zero. Ridge, on the other hand, does not allow weights to be set equal to zero like Lasso does:

$$R(w) = \lambda \times \sum_j w_j^2$$

Lasso is therefore known to perform shrinkage and variable selection, whereas Ridge only performs shrinkage, by penalizing the squared size of a weight. More recently, Hastie and Qian, 2014 set out an elastic net framework, which pertains to combining Ridge and Lasso by adding an additional parameter, α, yielding the elastic-net penalty:

$$R(w) = \lambda \times (\alpha(\sum_j |w_j|) + (1-\alpha) \times (w_j^2 \sum_j))$$

Hastie and Qian, 2014 refers to λ as the tuning parameter which controls the overall penalty scale. The elastic-net penalty is controlled by α, which apportions between the L1 and L2 terms, and hence in a way practices trade-off between shrinkage and variable selection.

The $R(w)$ function is incorporated into the cost function as follows:

$$Cost_{new} = Cost + \lambda \times R(w)$$

The λ factor controls the penalisation on the cost (Nguyen and Bui, 2018. Clearly, the higher this factor is, the greater the addition to the cost function is. Increasing this factor therefore results in simplifying the model (Nguyen and Bui, 2018). Its intuitive application is ensuring relatively larger absolute weighting structures are mitigated through proportionately higher cost function outputs via the regulariser. The other categories through which regularisation can be achieved, like $\frac{\partial y}{\partial w}$, allows more dynamic targeting of specific network configurations. Some theory presents techniques to penalise large derivatives i.e. $\frac{\partial y}{\partial w}$ greater than some threshold is penalised.

3.4.3 Optimisation Regularisation

Nguyen and Bui, 2018 describes optimisation and regularisation as being overlapping and inseparable. The optimisation procedure will have effect on 2 particular facets of the model, namely; speed of training and improved generalisation capabilities of the model. There is no established assessment framework to separate these 2 in assessing the optimisation (Nguyen and Bui, 2018). Empirical results and investigations suggest that using regularisation techniques as part of the optimisation provides better training speed (convergence) simultaneously with better generalisation capability. This therefore implies that the use of regularisation techniques during the optimisation procedures is an unambiguously valuable inclusion. Some well-known components of this category is stochastic gradient descent (SGD), drop outs and annealed gradient descent. SGD and drop-outs are discussed in the next section due to its arguably ubiquitous presence in NN model calibration and convergence.

Nguyen and Bui, 2018, in establishing a taxonomy for regularisation, established three categories that constitute the regularisation during optimisation. These are as follows:

(a) Initialisation and warm-start methods,

(b) Update methods,

(c) Termination methods.

Initialisation and Warm-start Methods

There are 2 sub-categories here; pre-training of data and no pre-training of data (Nguyen and Bui, 2018). Specifically, this category relates to how the network is initialised with parameters i.e. weights and biases. The impact of this can be profound on how the network is further calibrated and the resulting parameter set that underlies the network. Moreover, it is easy to imagine that a better initial choice, analogous to the Bayesian framework of selecting an appropriate prior, would inadvertently result in a faster calibration, since the "prior" information provides information that several network forward and backward parses would have had to learn (Nguyen and Bui, 2018).

(a) Without pre-training initialisation examples are as follows:
   (i) Random weights,
   (ii) Orthogonal weights matrices,
   (iii) Data-dependent weight initialisation.

(b) With pre-training initialisation examples are as follows:
   (i) Greedy layer-wise pre-training,
   (ii) Curriculum learning,
   (iii) Spatial contrasting.

Update Methods

There are 2 sub-categories; update rules and, weight and growth filters (Nguyen and Bui, 2018). Update rules are modifications to the update formulae, which is essentially how the learning rate is applied. Weight and growth filters refer to methods of updating whereby the network weight parameters and gradients in backpropogation is applied or transformed by the update formulae in order to update the parameters. Update method examples include; SGD, Adagrad, Nesterov's Accelerated Gradient Method, Learning Rate Schedules and Online Batch Selection (Nguyen and Bui, 2018).

SGD as a procedure, additional to its contextual mention under the gradient descent section, it can be summarised as follows:

(a) Sample a random mini-batch,

(b) Parse forward the mini-batch through the NN,

(c) Compute cost,

(d) Back propagate the cost,

Gradient descent in general takes all the training samples and averages across them in order to supplement a parameter update, while Mini-Batch GD, does not use all the training examples to perform a parameter update, but uses a deterministic sample instead (Pan and Jiang, 2014). SGD consists of sampling at random from the training data. A single iteration is produced where an iteration refers to a forward parse and backward parse of a sample and one update on each of the network parameters(Roy, 2010). SGD requires less computational resources, and there are mathematical proofs that SGD has the ability to process far more asymptotic samples then GD under equal computational resources (Pan and Jiang, 2014). Nguyen and Bui, 2018 describes SGD as an implicit regularisation on the NN. The thought behind it is that through using randomly sampled mini-batches, the network is implicitly assisted to not get "stuck" in saddle points/local minima of the cost surface (Nguyen and Bui, 2018). This therefore assists the NN to be less susceptible to initialisation dependencies affecting the resulting calibration set. This means that through using the random sampling in producing mini-batches, the adverse outcome set caused by poor initialisation weights and spurious initial gradient descent is largely circumvented. SGD therefore takes care of multi-modality issues while providing faster convergence (Pienaar, 2019).

In many applications there is particular difficulty with optimisation and getting stuck in local saddle points, which can be a doom loop of getting stuck in these saddles and descending in and around it (Pan and Jiang, 2014). SGD provides, via is random sampling and hence introduction of noise, a way to overcome this. The convergence of SGD has been widely studied, and Bottou, 2010 points out that it SGD has "almost certain" convergence under certain mild conditions. Specifically, these conditions pertain to the learning rate:

$$\gamma_t^2 < \infty$$

And;

$$\gamma_t^< \infty$$

Weight and growth filter methods on the other hand include; Dropouts, Annealed SGD (AGD) and other variants (Nguyen and Bui, 2018). The aforementioned saddle point problem is exacerbated by a high resolution error space, where the local minima are more abundant. The Mosiac Cost uses "codewords" to produce a simpler, lower resolution error space, where codewords are elements of the code-book, which is simplified approximation to the training set (Pan and Jiang, 2014). This therefore presents the appeal of lesser and "flatter" saddle points when the codewords are parsed forward to produce a cost surface, which mitigates the convergence issue in part. However, the derivation of an appropriate code-book itself requires additional work and poses additional issues (Pan and Jiang, 2014).

Annealed Gradient Descent attempts to effectively alleviate these issues by using k-means clustering, being an unsupervised technique, where k clusters are formed and each clusters' centroid (weighted center of the cluster) is used as the codeword. The clustering technique is used sequentially to add complexity, thereby constructing a hierarchical codebook (Pan and Jiang, 2014). An annealing schedule is constructed to specify what level of the hierarchy that is used in parsing forward, until the lowest level of the hierarchy is finally used, being the closest representation in the hierarchy to the training set (Pan and Jiang, 2014). AGD has proved to converge faster for non-complex applications than the SGD approach (Pan and Jiang, 2014).

Dropouts, on the other hand, is a particularly useful method in the sense that it is easy to apply and simple to understand (Nguyen and Bui, 2018). Regularisation in general has a computational cost, and in this regard, dropouts is relatively computationally cheap (Brownlee, 2019a). This does not present limitations for relatively smaller data sets that are applied to a NN, however, for larger data sets, a cost benefit assessment is due, and hence dropouts presents a highly appealing case from a pragmatist point of view.

Dropouts is applied during training, in between networks by making a random modification to weight parameters that connect neurons. The random modification is essentially a Bernoulli random variable that is applied to the weights (Bouthillier and Memisevic, 2016). By setting certain weights to zero, these neurons are "dropped out" of the network. The network structure is then streamlined through fewer neurons, and more concentrated learning is applied to randomly "not dropped out" weights across various instances of network iterations (forward and backward parses). Dropouts are generally applied in conjunction with SGD and mini-batches to work effectively (Srivastava and Salakhutdinov, 2014).

Optimising hyperparameters is considered a skilled and onerous exercise. Using ensembles of trained NNs of varying hyperparameter configurations and varied training data often provides desired generalisation abilities, but is manually demanding (Srivastava and Salakhutdinov, 2014). Srivastava and Salakhutdinov, 2014 advocates dropouts as a regularisation technique:
"Dropouts is a way of combining approximately exponentially many NNs".

Traditional interpretations of dropouts is described as an application of statistical bagging at test time, or alternatively stated, it implicitly wraps bagging in its application (Bouthillier and Memisevic, 2016). Bouthillier and Memisevic, 2016 establishes an alternative interpretation of dropouts; It is a sophisticated method of data augmentation. Though there are many interpretations, dropouts being inherently straight forward to apply, has a profound presence in NN literature. Srivastava and Salakhutdinov, 2014 conducted several experiments on various data sets, and the inclusion of dropouts provided a faster calibration/cheaper computation with a clear benefit. Additionally, dropouts was proved to be very similar to the usual weight decay regularisation (Bouthillier and Memisevic, 2016).

Termination Methods

This category refers to when optimisation, i.e. backpropagation, is stopped. This requires setting a criteria that is to be met, which can vary from iterations, to meeting a threshold cost/error. There are 2 sub-categories of termination methods; termination using a validation set and termination without using a validation set (Nguyen and Bui, 2018). The latter being used mainly when data is sparse and hence required wholly to train a model (Nguyen and Bui, 2018). By a far amount, most deep learning applications make use of the former termination method, since large data sets are mostly available for NN applications.

Chapter 4

Mathematical Exposition and Model Mechanics of Neural Networks

4.1 Mathematical Notation

The notation used for the mathematical representation of NNs in this section is largely based on Nielsen, 2015. The notation will be as follows:

$$\underset{N\times M}{\mathrm{X}} = \begin{bmatrix} x_{11} & x_{12} & \dots & x_{1M} \\ \vdots & \ddots & & \vdots \\ x_{N1} & \dots & & x_{NM} \end{bmatrix}$$

is the input matrix. In general, each x_{ij} element has the following interpretation:

(i) j refers to the neuron, and since its the input, it refers to the feature,

(ii) i refers to the observations, and the matrix has $N \times M$ elements ($observations \times features$).

The input matrix (X) is passed to the first layer of the network by multiplying a weight matrix with it. The weight matrix has the following notation:

$$\underset{M\times N_1}{\mathrm{W}^1} = \begin{bmatrix} w_{11}^{(1)} & w_{21}^{(1)} & \dots & w_{N_1 1}^{(1)} \\ \vdots & \ddots & & \vdots \\ w_{1M}^{(1)} & \dots & & w_{N_1 M}^{(1)} \end{bmatrix}$$

And in general, each item in the matrix, $w_{ij}^{(l)}$, can be interpreted as the weight that connects the jth neuron in the $l-1th$ layer to the ith neuron in the lth layer. Once the input is "fed forward" into the 1st hidden layer by means of matrix multiplication, a bias is added:

$$\underset{N\times N_1}{\mathrm{Z}^1} = \underset{N\times M}{\mathrm{X}} \times \underset{M\times N_1}{\mathrm{W}^1} + \underset{N\times N_1}{\mathrm{B}^1}, \tag{4.1}$$

where B is defined as:

$$\underset{N\times N_1}{\mathrm{B}^1} = \begin{bmatrix} b_1^{(1)} & b_2^{(1)} & \dots & b_{N_1}^{(1)} \\ \vdots & \ddots & & \vdots \\ b_1^{(1)} & \dots & & b_{N_1}^{(1)} \end{bmatrix},$$

where each and every observation of each neuron in the first layer is added the same bias, i.e. B^1 is populated with a identical row vectors and hence the same bias is added to each first layer observation (Nielsen, 2015). Z is defined as:

$$\underset{N \times N_1}{\mathrm{Z}^1} = \begin{bmatrix} z_{11}^{(1)} & z_{12}^{(1)} & \dots & z_{1N_1}^{(1)} \\ \vdots & \ddots & & \vdots \\ z_{N1}^{(1)} & \dots & & z_{NN_1}^{(1)} \end{bmatrix}$$

In general, each item in the matrix, $z_{ij}^{(l)}$, can be interpreted as follows:

(i) the ith observation,

(ii) for the jth neuron,

(iii) in the lth layer.

Now, a non-linear transformation is applied to each entry in $\underset{N \times N_1}{\mathrm{Z}^1}$ as follows:

$$\underset{N \times N_1}{\mathrm{A}^1} = \sigma(\underset{N \times N_1}{\mathrm{Z}^1}) \tag{4.2}$$

with σ taking the form of a non-linear transformation function. For illustrative purposes the well known Sigmoid function will be used:

$$\sigma(z_{ij}^{(l)}) = \frac{1}{1 + e^{-z_{ij}^{(l)}}}, \tag{4.3}$$

and its derivative, $\frac{\partial \sigma}{\partial z_{ij}^{(l)}}$, is defined as:

$$\sigma\prime(z_{ij}^{(l)}) = \sigma(z_{ij}^{(l)}) \times (1 - \sigma(z_{ij}^{(l)})). \tag{4.4}$$

Therefore the matrix $\underset{N \times N_1}{\mathrm{A}^1}$ can be written as:

$$\underset{N \times N_1}{\mathrm{A}^1} = \begin{bmatrix} \sigma(z_{11}^{(1)}) & \sigma(z_{12}^{(1)}) & \dots & \sigma(z_{1N_1}^{(1)}) \\ \vdots & \ddots & & \vdots \\ \sigma(z_{N1}^{(1)}) & \dots & & \sigma(z_{NN_1}^{(1)}) \end{bmatrix}$$

Thus, the input data is passed into the the first hidden layer, i.e. the second layer of the NN. As the inputs are passed on between layers, a weight operator (scaling), bias operator (shifting) and then non-linear transformation acts on the data at each layer, respectively. In general, for any hidden layer l in the network, we can write $\underset{N \times N_1}{\mathrm{A}^l}$, being the output of the scaled and shifted output of the previous layer, as follows:

$$\begin{aligned} \underset{N \times N_l}{\mathrm{A}^l} &= \sigma(\underset{N \times N_l}{\mathrm{Z}^l}) \\ &= \sigma(\underset{N \times N_{l-1}}{\mathrm{A}^{l-1}} \times \underset{N_{l-1} \times N_l}{\mathrm{W}^l} + \underset{N \times N_l}{\mathrm{B}^l}) \\ &= \begin{bmatrix} \sigma(z_{11}^{(l)}) & \sigma(z_{12}^{(l)}) & \dots & \sigma(z_{1N_1}^{(l)}) \\ \vdots & \ddots & & \vdots \\ \sigma(z_{N1}^{(l)}) & \dots & & \sigma(z_{NN_l}^{(l)}) \end{bmatrix} \end{aligned}$$

$$= \begin{bmatrix} a_{11}^{(l)} & a_{12}^{(l)} & \dots & a_{1N_1}^{(l)} \\ \vdots & \ddots & & \vdots \\ a_{N1}^{(l)} & \dots & & a_{NN_l}^{(l)} \end{bmatrix}$$

Taking an interim view of $\underset{N\times N_l}{\mathrm{Z}^{\mathrm{l}}}$ will add insight into what happens to the data as it is fed forward through the network.

$$\underset{N\times N_l}{\mathrm{Z}^{\mathrm{l}}} = \begin{bmatrix} \sum_k a_{1k}^{(l-1)} \times w_{1k}^{(l)} + b_1^{(l)} & \sum_k a_{1k}^{(l-1)} \times w_{2k}^{(l)} + b_2^{(l)} & \dots & \sum_k a_{1k}^{(l-1)} \times w_{N_lk}^{(l)} + b_{N_l}^{(l)} \\ \vdots & & & \vdots \\ \vdots & & \ddots & \vdots \\ \sum_k a_{Nk}^{(l-1)} \times w_{1k}^{(l)} + b_1^{(l)} & & \dots & \sum_k a_{Nk}^{(l-1)} \times w_{N_lk}^{(l)} + b_{N_l}^{(l)} \end{bmatrix}$$

Therefore, the general case for $z_{ij}^{(l)}$ will be as follows:

$$z_{ij}^{(l)} = \sum_k a_{ik}^{(l-1)} \times w_{jk}^{(l)} + b_j^{(l)}. \tag{4.5}$$

4.2 Single Output Cost and Learning

From section 4.1 it is clear that the factor $z_{ij}^{(l)}$ is a product of multiplying each activation ($a_{ik}^{(l-1)}$) by $w_{jk}^{(l)}$, for a single observation (hence i is fixed in the summation). Therefore taking each feature observation of the ith observation, and multiplying it with the weight $w_{jk}^{(l)}$, which connects the kth feature of an observation in the $l-1th$ layer, to the jth feature in the lth layer, for the ith observation (Nielsen, 2015).

Thus, by summing over k, all the activation's of the ith observation is weighted to the jth feature in the lth layer. Hence a weighted summation of the $l-1th$ layer features, for a particular observation (i), is taken to calculate the lth layer observation of a particular feature (j).

After the above process is followed to feed forward an input matrix of features and observations, the final hidden layers activations ($L-1$), is multiplied with a weight matrix that generates outputs of the dimension which the process is observed, i.e. often an $N \times 1$ output ($observations \times outputs$).

These outputs are considered against the dependent variables (y_i) that are observed. The NN predicts y_i and is optimised by having a measure of error on which parameters are to be calibrated (trained). Thus, we can state predictions, $\hat{y}_i$, in a matrix form as:

$$\underset{N\times 1}{\hat{\mathrm{Y}}} = \underset{N\times 1}{\mathrm{A}^{\mathrm{L}}} = \sigma(\underset{N\times 1}{\mathrm{Z}^{\mathrm{L}}}) = \sigma(\underset{N\times N_L}{\mathrm{A}^{\mathrm{L}-1}} \times \underset{N_L\times 1}{\mathrm{W}^{\mathrm{L}}} + \underset{N\times 1}{\mathrm{B}^{\mathrm{L}}}).$$

There are several optimisation frameworks within which training can be achieved, gradient descent (and stochastic gradient descent) being a very popular and common one.

Optimisation requires a defined measure of error, which is typically referred to as the cost. The optimisation procedure starts with the cost function, C, which can take many forms, the most common being the Mean Squared Error (MSE) function:

$$C = \sum_i C_i = \sum_i \frac{1}{2}(y_i - \hat{y}_i)^2 = \sum_i \frac{1}{2}(y_i - A_i^L)^2, \tag{4.6}$$

and thus, the derivative function of the cost function w.r.t. the predictions ($\hat{y}_i$) is as follows:

$$\frac{\partial C_i}{\partial \hat{Y}_i} = (y_i - \hat{y}_i). \tag{4.7}$$

In order to optimise the network on the input data, and thereby apply gradient descent, a computation of each and every cost observation w.r.t. each parameter within the NN, across the various layers, is required i.e. $\frac{\partial C_i}{\partial w_{ij}^{(l)}}$ and $\frac{\partial C_i}{\partial b_j^{(l)}}$.

The process of propagating the cost output resulting from any particular parameter set for a NN backwards through the NN in order to quantify each $\frac{\partial C_i}{\partial w_{ij}^{(l)}}$ and $\frac{\partial C_i}{\partial b_j^{(l)}}$ is the infamous backpropagation procedure. We then define the vector of cost derivatives w.r.t. each prediction (∇_c) as follows:

$$\nabla_c = \frac{\partial C}{\partial \hat{Y}} = \begin{bmatrix} \frac{\partial c_1}{\partial \hat{y}_1} \\ \frac{\partial c_2}{\partial \hat{y}_2} \\ \vdots \\ \frac{\partial c_N}{\partial \hat{y}_N} \end{bmatrix}$$

Consider first the derivatives of weights w.r.t. the cost function between the last hidden layer ($L-1$) and the output layer (L) in the network i.e. $\frac{\partial C}{\partial w_{ij}^{(L)}}$. By using the chain rule, the following results:

$$\frac{\partial C_i}{\partial w_{jk}^{(L)}} = \frac{\partial C_i}{\partial \hat{y}_i} \times \frac{\partial \hat{y}_i}{\partial \sigma(z_i^{(L)})} \times \frac{\partial \sigma(z_i^{(L)})}{\partial z_i^{(L)}} \times \frac{\partial z_i^{(L)}}{\partial w_{jk}^{(L)}}. \tag{4.8}$$

Now we take a look at each component of equation 4.8:

$$\frac{\partial C_i}{\partial \hat{y}_i} = (y_i - A_i^L)^2, \tag{4.9}$$

$$\frac{\partial \hat{y}_i}{\partial \sigma(z_i^{(L)})} = 1, \tag{4.10}$$

$$\frac{\partial \sigma(z_i^{(L)})}{\partial z_i^{(L)}} = \sigma\prime = \sigma(z_i^{(l)}) \times (1 - \sigma(z_i^{(l)})), \tag{4.11}$$

$$\frac{\partial z_i^{(L)}}{\partial w_{jk}^{(L)}} = a_{ik}^{L-1}. \tag{4.12}$$

The derivatives in equations 4.9 - 4.12 is for a single observation i.e. observation i. Because the bias added in this layer, ${b_i}^L$, does not depend on the weights, it is treated as a constant. Therefore, taking the same derivative across all observations, $\frac{\partial C}{\partial w_{jk}^{(L)}}$ and by the additive property of the chain rule, yields:

$$\frac{\partial C}{\partial w_{jk}^{(L)}} = \sum_i \frac{\partial C_i}{\partial \hat{y}_i} \times \frac{\partial \hat{y}_i}{\partial \sigma(z_i^{(L)})} \times \frac{\partial \sigma(z_i^{(L)})}{\partial z_i^{(L)}} \times \frac{\partial z_i^{(L)}}{\partial w_{jk}^{(L)}}.$$

Since we are considering a NN with 1 neuron in the output layer, i.e. $j = 1$ and each weight connects to the single output neuron, we have:

$$\frac{\partial z_i^{(L)}}{\partial w_{1k}^{(L)}} = a_{ik}^{L-1}.$$

Define dW^L as the weight derivative matrix:

$$dW^L = \begin{bmatrix} \sum_i \frac{\partial C_i}{\partial \hat{y}_i} \times \frac{\partial \hat{y}_i}{\partial \sigma(z_i^{(L)})} \times \frac{\partial \sigma(z_i^{(L)})}{\partial z_i^{(L)}} \times \frac{\partial z_i^{(L)}}{\partial w_{11}^{(L)}} \\ \sum_i \frac{\partial C_i}{\partial \hat{y}_i} \times \frac{\partial \hat{y}_i}{\partial \sigma(z_i^{(L)})} \times \frac{\partial \sigma(z_i^{(L)})}{\partial z_i^{(L)}} \times \frac{\partial z_i^{(L)}}{\partial w_{12}^{(L)}} \\ \vdots \\ \sum_i \frac{\partial C_i}{\partial \hat{y}_i} \times \frac{\partial \hat{y}_i}{\partial \sigma(z_i^{(L)})} \times \frac{\partial \sigma(z_i^{(L)})}{\partial z_i^{(L)}} \times \frac{\partial z_i^{(L)}}{\partial w_{1N_{L-1}}^{(L)}} \end{bmatrix}$$

We are interested in the derivative w.r.t. each weight. Hence the following matrix notation can be used to get all the needed components of dW^L:

$$\sigma\prime = \frac{\partial \sigma(Z^{(L)})}{\partial Z^{(L)}} = \begin{bmatrix} \frac{\partial \sigma(z_1^{(L)})}{\partial z_1^{(L)}} \\ \frac{\partial \sigma(z_2^{(L)})}{\partial z_2^{(L)}} \\ \vdots \\ \frac{\partial \sigma(z_N^{(L)})}{\partial z_N^{(L)}} \end{bmatrix}$$

Therefore, consider:

$$\frac{\partial C}{\partial \hat{Y}} \odot \sigma\prime = \begin{bmatrix} \frac{\partial C_1}{\partial \hat{Y}_1} \times \frac{\partial \sigma(z_1^{(L)})}{\partial z_1^{(L)}} \\ \frac{\partial C_2}{\partial \hat{Y}_1} \times \frac{\partial \sigma(z_2^{(L)})}{\partial z_2^{(L)}} \\ \vdots \\ \frac{\partial C_N}{\partial \hat{Y}_N} \times \frac{\partial \sigma(z_N^{(L)})}{\partial z_N^{(L)}} \end{bmatrix}$$

Clearly a partition of the partial derivative of the cost function w.r.t. a particular weight (the first 2 partial derivatives of equation 4.8) can be written in a vectorised form. Using equations 4.9-4.12, yields:

$$dW^L = \begin{bmatrix} \sum_i (y_i - \hat{y}_i) \times 1 \times \sigma\prime(z_i^{(l)}) \times a_{i1}^{L-1} \\ \sum_i (y_i - \hat{y}_i) \times 1 \times \sigma\prime(z_i^{(l)}) \times a_{i2}^{L-1} \\ \vdots \\ \sum_i (y_i - \hat{y}_i) \times 1 \times \sigma\prime(z_i^{(l)}) \times a_{iN_{L-1}}^{L-1} \end{bmatrix}$$

Therefore, dW^L can be written as:

$$dW^L = \begin{bmatrix} a_{11}^{(L)} & a_{21}^{(L)} & \dots & a_{N1}^{(L)} \\ \vdots & \ddots & & \vdots \\ a_{1N_{L-1}}^{(L)} & \dots & & a_{NN_{L-1}}^{(L)} \end{bmatrix} \times \begin{bmatrix} (y_1 - \hat{y}_1) \times \sigma\prime(z_1^{(L)}) \\ (y_2 - \hat{y}_2) \times \sigma\prime(z_2^{(L)}) \\ \vdots \\ (y_N - \hat{y_N}) \times \sigma\prime(z_N^{(L)}) \end{bmatrix}$$

Thus, dW^L can be written as:

$$dW^L = (\mathrm{A}^{\mathrm{L}-1})^T \times \nabla_c \odot \sigma\prime(Z^L). \tag{4.13}$$

Which, in words, is the transpose of the matrix A^{L-1} multiplied by the derivative matrix, $\frac{\partial C}{\partial \hat{Y}} \odot \sigma\prime$.Hence, the following is defined:

$$\delta^L = \nabla_c \odot \sigma\prime(Z^L), \tag{4.14}$$

$$dW^L = (A^{L-1})^T \times \delta^L. \tag{4.15}$$

It is important to keep in mind what the purpose of the backpropagation is; determining the derivative of a cost observation w.r.t. each parameter in each layer i.e. partial derivatives of the cost function w.r.t. all weights and biases (Nielsen, 2015). Next, we will look at the derivatives for the bias matrix, using some of the above notation. Consider:

$$\underset{N\times 1}{\hat{Y}} = \underset{N\times 1}{\mathrm{A}^{\mathrm{L}}} = \begin{bmatrix} (y_1 - \hat{y}_1) \times \sigma\prime(z_1^{(L)}) \\ (y_2 - \hat{y}_2) \times \sigma\prime(z_2^{(L)}) \\ \vdots \\ (y_N - \hat{y_N}) \times \sigma\prime(z_N^{(L)}) \end{bmatrix}$$

With a NN that has one output neuron (only a single bias and equal bias is added to each of the output observations), the following is true for observation i:

$$\frac{\partial C_i}{\partial b_1^{(L)}} = \frac{\partial C_i}{\partial \hat{y}_i} \times \frac{\partial \hat{y}_i}{\partial \sigma(z_i^{(L)})} \times \frac{\partial \sigma(z_i^{(L)})}{\partial z_i^{(L)}} \times \frac{\partial z_i^{(L)}}{\partial b_1^{(L)}} = (y_i - \hat{y}_i) \times \sigma\prime(z_i^{(L)}) \times 1,$$

since:

$$\frac{\partial z_i^{(L)}}{\partial b_1^{(L)}} = \frac{\partial}{\partial b_1^{(L)}} (\sum_k a_{ik}^{(L-1)} \times w_{1k} + b_1) = 1.$$

Therefore, dB^L can be expressed as:

$$dB^L = \begin{bmatrix} \sum_i (y_i - \hat{y}_i) \times 1 \times \sigma\prime(z_i^{(L)}) \\ \sum_i (y_i - \hat{y}_i) \times 1 \times \sigma\prime(z_i^{(L)}) \\ \vdots \\ \sum_i (y_i - \hat{y}_i) \times 1 \times \sigma\prime(z_i^{(L)}) \end{bmatrix} = \begin{bmatrix} 1 & 1 & \dots & 1 \\ \vdots & \ddots & & \vdots \\ 1 & \dots & & 1 \end{bmatrix} \times \begin{bmatrix} (y_1 - \hat{y_1}) \times \sigma\prime(z_1^{(L)}) \\ (y_2 - \hat{y_2}) \times \sigma\prime(z_2^{(L)}) \\ \vdots \\ (y_N - \hat{y_N}) \times \sigma\prime(z_N^{(L)}) \end{bmatrix}$$

And thus, dB^L can be written as:

$$dB^L = I \times \delta^L \tag{4.16}$$

.

Therefore, we have illustrated how the partial derivatives of the final layer weight and bias parameters can be written in a generalised matrix form. This can be extended to the more general case where the output for each observation is vectorised i.e. has more than one feature. Therefore, the output will be matrix of observations by output features.

4.3 Multiple Output Cost and Learning

We now consider a NN that has more than 1 output, in other words more than 1 predictor. Using similar definitions as in the single output neuron case, we define the matrix of cost derivatives as follows:

$$\nabla_c = \frac{\partial C}{\partial \hat{Y}} = \begin{bmatrix} \frac{\partial c_{11}}{\partial \hat{y_{11}}} & \dots & \frac{\partial c_{1O}}{\partial \hat{y_{1O}}} \\ \frac{\partial c_{21}}{\partial \hat{y_{21}}} & \dots & \frac{\partial c_{2O}}{\partial \hat{y_{2O}}} \\ \vdots & & \vdots \\ \frac{\partial c_{N1}}{\partial \hat{y_{N1}}} & \dots & \frac{\partial c_{NO}}{\partial \hat{y_{NO}}} \end{bmatrix}$$

Consider the derivatives of weights w.r.t. the cost function in the first to last layer in the network i.e. $\frac{\partial C}{\partial w_{ij}^{(L)}}$. By using the chain rule retrospectively yields:

$$\frac{\partial C_i}{\partial w_{jk}^{(L)}} = \frac{\partial C_i}{\partial \hat{y}_i} \times \frac{\partial \hat{y}_i}{\partial \sigma(z_i^{(L)})} \times \frac{\partial \sigma(z_i^{(L)})}{\partial z_i^{(L)}} \times \frac{\partial z_i^{(L)}}{\partial w_{jk}^{(L)}}. \tag{4.17}$$

Now, the components of equation 4.17 yields the following equations:

$$\frac{\partial C_{ij}}{\partial \hat{y_{ij}}} = (y_{ij} - A_{ij}^L)^2, \tag{4.18}$$

$$\frac{\partial \hat{y}_{ij}}{\partial \sigma(z_{ij}^{(L)})} = 1, \tag{4.19}$$

$$\frac{\partial \sigma(z_{ij}^{(L)})}{\partial z_{ij}^{(L)}} = \sigma\prime = \sigma(z_{ij}^{(l)}) \times (1 - \sigma(z_{i}^{(l)})), \tag{4.20}$$

$$\frac{\partial z_{i}^{(L)}}{\partial w_{jk}^{(L)}} = a_{ik}^{L-1}, \tag{4.21}$$

Take note that the derivative is still taken for a single observation i.e. observation i, however there are multiple output features now i.e. $j > 1$. For the final layer, ${b_{ij}}^{L}$ is still a constant. Therefore, $\frac{\partial C}{\partial w_{jk}^{(L)}}$ for multiple output features yields:

$$\frac{\partial C}{\partial w_{jk}^{(L)}} = \sum_{i} \frac{\partial C_{ij}}{\partial \hat{y}_{ij}} \times \frac{\partial \hat{y}_{ij}}{\partial \sigma(z_{ij}^{(L)})} \times \frac{\partial \sigma(z_{ij}^{(L)})}{\partial z_{ij}^{(L)}} \times \frac{\partial z_{ij}^{(L)}}{\partial w_{jk}^{(L)}}. \tag{4.22}$$

Since we have multiple neurons in the output layer now, $j > 1$, the following results:

$$\frac{\partial z_{ij}^{(L)}}{\partial w_{jk}^{(L)}} = a_{ik}^{L-1}. \tag{4.23}$$

Therefore, dW^L, the weight derivative matrix, is now expressible in matrix form as:

$$dW^L = \begin{bmatrix} \sum_i \frac{\partial C_{ij}}{\partial \hat{y}_{ij}} \times \frac{\partial \hat{y}_{ij}}{\partial \sigma(z_{i}^{(L)})} \times \frac{\partial \sigma(z_{ij}^{(L)})}{\partial z_{ij}^{(L)}} \times \frac{\partial z_{ij}^{(L)}}{\partial w_{11}^{(L)}} & \cdots \\ \sum_i \frac{\partial C_{ij}}{\partial \hat{y}_{ij}} \times \frac{\partial \hat{y}_{ij}}{\partial \sigma(z_{ij}^{(L)})} \times \frac{\partial \sigma(z_{ij}^{(L)})}{\partial z_{i}^{(L)}} \times \frac{\partial z_{ij}^{(L)}}{\partial w_{12}^{(L)}} & \cdots \\ \vdots & \\ \sum_i \frac{\partial C_{ij}}{\partial \hat{y}_{ij}} \times \frac{\partial \hat{y}_{ij}}{\partial \sigma(z_{ij}^{(L)})} \times \frac{\partial \sigma(z_{ij}^{(L)})}{\partial z_{ij}^{(L)}} \times \frac{\partial z_{ij}^{(L)}}{\partial w_{1N_{L-1}}^{(L)}} & \cdots \end{bmatrix}$$

Using the same notation as in section 4.2, and considering equations 4.18-4.21 in dW^L yields the same equation as Equation 4.14:

$$\delta^L = \nabla_c \odot \sigma\prime(Z^L).$$

Where δ^L is a matrix of size $N \times O$ (O is the output feature dimension or the amount of output neurons), as opposed to a matrix of size $N \times 1$ as in section 4.2.Thus, the same generalisation holds for dB^L:

$$dB^L = I \times \delta.$$

In the following section, backpropagation and learning is discussed, using the terminology, equations and matrices set out in section 4.2 and 4.3.

4.4 Backpropagation and Learning

The above must be extended to the next (2nd to last) layer, moving backwards through the NN, and even further until the first weight and bias matrix's partial derivatives are computed in matrix form. This is the essence of backpropagation; propagating the prediction/classification error back through the network to compute gradients of the cost w.r.t. all network parameters. For the purpose of notational simplicity, the single output network is used to demonstrate the mathematical dynamics of backpropagation.

Moving to the next layer, via partial derivatives, we need to compute the derivatives of the cost function w.r.t. all w_{jk}^{L-1} and b_{jk}^{L-1}. Consider the partial derivatives w.r.t. w_{jk}^{L-1} first:

$$\underset{N\times N_{L-1}}{\mathrm{Z^L}} = \begin{bmatrix} \sum_k a_{1k}^{(L-2)} \times w_{1k}^{(L-1)} + b_1^{(L-1)} & \sum_k a_{1k}^{(L-2)} \times w_{2k}^{(L-1)} + b_2^{(L-1)} & \dots \\ \vdots & & \vdots \\ \vdots & \ddots & \vdots \\ \sum_k a_{Nk}^{(L-2)} \times w_{1k}^{(L-1)} + b_1^{(L-1)} & \dots & \end{bmatrix}$$

It is clear that each weight in layer $L-1$ is applied to a neuron feature for each observation once. Intuitively, each observation depends on any particular weight once, and so the total derivative w.r.t. each weight will be a weighted sum of activations. Where the weightings applied in this sum is the partial derivative computed moving back through the network to the layer before that weight in consideration was applied to an activation. To illustrate this, consider:

$$\frac{\partial C_i}{\partial w_{jk}^{(L-1)}} = \frac{\partial C_i}{\partial \hat{y}_i} \times \frac{\partial \hat{y}_i}{\partial \sigma(z_i^{(L)})} \times \frac{\partial \sigma(z_i^{(L)})}{\partial z_i^{(L)}} \times \frac{\partial z_i^{(L)}}{\partial a_{ij}^{(L-1)}} \times \frac{\partial a_{ij}^{(L-1)}}{\partial \sigma(z_i^{(L-1)})} \times \frac{\partial \sigma(z_{ij}^{(L-1)})}{\partial z_{ij}^{(L-1)}} \times \frac{\partial z_{ij}^{(L-1)}}{\partial w_{jk}^{(L-1)}}. \tag{4.24}$$

Looking at each component in equation 4.24 yields:

$$\frac{\partial C_i}{\partial \hat{y}_i} = (y_i - A_i^L)^2, \tag{4.25}$$

$$\frac{\partial \hat{y}_i}{\partial \sigma(z_i^{(L)})} = 1, \tag{4.26}$$

$$\frac{\partial \sigma(z_i^{(L)})}{\partial z_i^{(L)}} = \sigma\prime = \sigma(z_i^{(l)}) \times (1 - \sigma(z_i^{(l)})), \tag{4.27}$$

$$\frac{\partial z_i^{(L)}}{\partial a_{ij}^{(L-1)}} = w_{1k}^{(L)}, \tag{4.28}$$

$$\frac{\partial a_{ij}^{(L-1)}}{\partial \sigma(z_i^{(L-1)})} = 1, \tag{4.29}$$

$$\frac{\partial \sigma(z_{ij}^{(L-1)})}{\partial z_{ij}^{(L-1)}} = \sigma(z_{ij}^{(L-1)}) \times (1 - \sigma(z_{ij}^{(L-1)})), \tag{4.30}$$

$$\frac{\partial z_{ij}^{(L-1)}}{\partial w_{jk}^{(L-1)}} = a_{ik}^{(L-2)}. \tag{4.31}$$

Notice that the first three components are already computed by calculating the partial derivatives of the final weights and biases. Of the remaining 4 components, we have all the weights in the final layer, and hence there are only the last 2 terms left. The second to last term is found by applying the Sigmoid derivative to the computed Z's and the final component is already known, being the activation matrix in the 2nd to last layer (2 layers before the output).

Similar to the dW^L matrix expression in section 4.2 and 4.3, we sought a matrix representation of the partial derivative w.r.t. each weight. This will be a weighted sum over all observations. Hence, we define the following:

$$dW^{(L-1)} = \begin{bmatrix} \sum_i \frac{\partial C_i}{\partial \hat{y}_i} \times \frac{\partial \sigma(z_i^{(L)})}{\partial z_i^{(L)}} \times \frac{\partial z_i^{(L)}}{\partial a_{i1}^{(L-1)}} \times \frac{\partial \sigma(z_{i1}^{(L-1)})}{\partial z_{i1}^{(L-1)}} \times \frac{\partial z_{i1}^{(L-1)}}{\partial w_{11}^{(L-1)}} & \cdots \\ \sum_i \frac{\partial C_i}{\partial \hat{y}_i} \times \frac{\partial \sigma(z_i^{(L)})}{\partial z_i^{(L)}} \times \frac{\partial z_i^{(L)}}{\partial a_{i2}^{(L-1)}} \times \frac{\partial \sigma(z_{i1}^{(L-1)})}{\partial z_{i1}^{(L-1)}} \times \frac{\partial z_{i1}^{(L-1)}}{\partial w_{12}^{(L-1)}} & \cdots \\ \vdots & \vdots \\ \sum_i \frac{\partial C_i}{\partial \hat{y}_i} \times \frac{\partial \sigma(z_i^{(L)})}{\partial z_i^{(L)}} \times \frac{\partial z_i^{(L)}}{\partial a_{iN_{L-1}}^{(L-1)}} \times \frac{\partial \sigma(z_{i1}^{(L-1)})}{\partial z_{i1}^{(L-1)}} \times \frac{\partial z_{i1}^{(L-1)}}{\partial w_{1N_{L-1}}^{(L-1)}} & \cdots \end{bmatrix}.$$

With any specific element having the form:

$$\sum_i \frac{\partial C_i}{\partial \hat{y}_i} \times \frac{\partial \sigma(z_i^{(L)})}{\partial z_i^{(L)}} \times \frac{\partial z_i^{(L)}}{\partial a_{ik}^{(L-1)}} \times \frac{\partial \sigma(z_{ij}^{(L-1)})}{\partial z_{ij}^{(L-1)}} \times \frac{\partial z_{ij}^{(L-1)}}{\partial w_{jk}^{(L-1)}}. \tag{4.32}$$

Considering equations 4.25-4.31, yields:

$$dW^L = \begin{bmatrix} \sum_i (y_i - \hat{y}_i) \times \sigma\prime(z_i^{(L)}) \times w_{11}^{(L)} \times \sigma\prime(z_{i1}^{(L-1)}) \times a_{i1}^{(L-2)} & \cdots \\ \sum_i (y_i - \hat{y}_i) \times \sigma\prime(z_i^{(L)}) \times w_{12}^{(L)} \times \sigma\prime(z_{i1}^{(L-1)}) \times a_{i2}^{(L-2)} & \cdots \\ \vdots & \vdots \\ \sum_i (y_i - \hat{y}_i) \times \sigma\prime(z_i^{(L)}) \times w_{1N_{L-2}}^{(L)} \times \sigma\prime(z_{i1}^{(L-1)}) \times a_{iN_{L-2}}^{(L-2)} & \cdots \end{bmatrix}$$

Taking a look at the above matrix, there are several factors that are already defined:

$$\delta^L = \nabla_c \odot \sigma\prime(Z^L)$$

Consider now:

$$\delta^L \times (W^L)^T = \begin{bmatrix} (y_1 - \hat{y_1}) \times \sigma\prime(z_1^{(L)}) \\ \\ (y_2 - \hat{y_2}) \times \sigma\prime(z_2^{(L)}) \\ \vdots \\ (y_N - \hat{y_N}) \times \sigma\prime(z_N^{(L)}) \end{bmatrix} \times \begin{bmatrix} w_{11}^{(L)} & w_{12}^{(L)} & \dots & w_{1N_{L-1}}^{(L)} \end{bmatrix}$$

$$= \begin{bmatrix} (y_1 - \hat{y_1}) \times \sigma\prime(z_1^{(L)}) \times w_{11}^{(L)} & (y_1 - \hat{y_1}) \times \sigma\prime(z_1^{(L)}) \times w_{12}^{(L)} & \dots & (y_1 - \hat{y_1}) \times \sigma\prime(z_1^{(L)}) \times w_{1N_{L-1}}^{(L)} \\ \\ (y_2 - \hat{y_2}) \times \sigma\prime(z_2^{(L)}) \times w_{11}^{(L)} & (y_2 - \hat{y_2}) \times \sigma\prime(z_2^{(L)}) \times w_{12}^{(L)} & \dots & (y_2 - \hat{y_2}) \times \sigma\prime(z_2^{(L)}) \times w_{1N_{L-1}}^{(L)} \\ \vdots & \vdots & & \\ (y_N - \hat{y_N}) \times \sigma\prime(z_N^{(L)}) \times w_{11}^{(L)} & (y_N - \hat{y_N}) \times \sigma\prime(z_N^{(L)}) \times w_{12}^{(L)} & \dots & (y_N - \hat{y_N}) \times \sigma\prime(z_N^{(L)}) \times w_{1N_{L-1}}^{(L)} \end{bmatrix}$$

Notice that the above matrix contains only some of the terms needed to compute $dW^{(L-1)}$. However, the $\sigma\prime(z_N^{(L)})$ terms, a_{ik}^{L-2} and the summation operator is needed to produce equation 4.32 for each weight. Thus, we use the Haddamard product similar to the 1st layer derivatives earlier, as follows:

$$\underset{N\times N_{L-1}}{(\delta^L \times (W^L)^T)} \odot \underset{N\times N_{L-1}}{\sigma\prime(Z^{L-1})} = \begin{bmatrix} (y_1 - \hat{y_1}) \times \sigma\prime(z_1^{(L)}) \times w_{11}^{(L)} \times \sigma\prime(z_{11}^{(L)}) & \\ \\ (y_2 - \hat{y_2}) \times \sigma\prime(z_2^{(L)}) \times w_{11}^{(L)} \times \sigma\prime(z_{21}^{(L)}) & \\ \vdots & \vdots \\ (y_N - \hat{y_N}) \times \sigma\prime(z_N^{(L)}) \times w_{11}^{(L)} \times \sigma\prime(z_{N1}^{(L)}) & \end{bmatrix}$$

And finally, multiplying the result with the transpose of A^{L-2} yields dW^{L-1}:

$$dW^{L-1} = (A^{L-2})^T \times \delta^L \times (W^L)^T \odot \sigma\prime(Z^{L-1}).$$

The same derivation for the partial derivatives w.r.t. b_j^{L-1} as above, with the final partial derivative in equation 4.24 being different, namely 1:

$$dB^{L-1} = I \times \delta^L \times (W^L)^T \odot \sigma\prime(Z^{L-1}). \tag{4.33}$$

Thus, the following equations are derived:

$$\delta^{L-1} = \delta^L \times (W^L)^T \odot \sigma\prime(Z^{L-1}), \tag{4.34}$$

$$dW^{L-1} = (A^{L-2})^T \times \delta^{L-1}, \tag{4.35}$$

$$dB^{L-1} = I \times \delta^{L-1}. \tag{4.36}$$

The general expression relating to any NN layer are as follows:

$$\delta^{L-r} = \begin{cases} \nabla_c \odot \sigma\prime(Z^L) & r = 0 \\ \delta^{L-r+1} \times (W^{L-r+1})^T \odot \sigma\prime(Z^{L-r}) & 0 < r \end{cases} \tag{4.37}$$

And:

$$dW^{L-r} = (A^{L-r-1})^T \times \delta^{L-r}, \tag{4.38}$$

$$dB^{L-r} = I \times \delta^{L-r}. \tag{4.39}$$

4.5 Regularisation Terms

Suppose that regularisation terms are added to the cost, yielding a modified cost function and hence cost surface that is generated by the data. In this section the single output network is again used to simplify notation.

4.5.1 Terms Dependent on Output

Suppose the initial cost function takes the form of MSE function again:

$$C = MSE = \sum_i C_i,$$

where:

$$C_i = \frac{1}{2}(y_i - \hat{y}_i)^2.$$

Regularisation terms are linearly added to the initial cost, where the *mth* regularisation term can be denoted as $R_m(\hat{y})$, and C^* denotes the modified cost function which is expressed as:

$$C^* = MSE + \sum_m R_m(\hat{y}).$$

Thus:

$$C^* = \sum_i C_i + R_{1,i}(\hat{y}) + ... + R_{m,i}(\hat{y}).$$

The R_m terms are required to be differentiable w.r.t. the output ($\hat{y}_i$) and hence the network parameters in order for the error to be propagated back through the network. Consider now each of the individual cost observations and the derivatives:

$$\frac{\partial C_i^*}{\partial \hat{y}_i} = \frac{\partial C_i}{\partial \hat{y}_i} + \frac{\partial R_{1,i}}{\partial \hat{y}_i} + ... + \frac{\partial R_{m,i}}{\partial \hat{y}_i}.$$

The chain rule expression of the new cost derivative w.r.t. a specific weight, $w_{jk}^{(L)}$, is expressed as follows:

$$\frac{\partial C_i^*}{\partial w_{jk}^{(L)}} = \frac{\partial C_i^*}{\partial \hat{y}_i} \times \frac{\partial \hat{y}_i}{\partial \sigma(z_i^{(L)})} \times \frac{\partial \sigma(z_i^{(L)})}{\partial z_i^{(L)}} \times \frac{\partial z_i^{(L)}}{\partial w_{jk}^{(L)}}.$$

Clearly, the added R_m terms only affects the first term in this equation. Therefore, the addition of these terms will only have an impact on the ∇ term in the stated backpropagation equations.

The initial matrix form of the $\frac{\partial C_i}{\partial \hat{y_i}}$ observations is stated as:

$$\nabla_c = \frac{\partial C}{\partial \hat{Y}} = \begin{bmatrix} \frac{\partial c_1}{\partial \hat{y_1}} \\ \frac{\partial c_2}{\partial \hat{y_2}} \\ \vdots \\ \frac{\partial c_N}{\partial \hat{y_N}} \end{bmatrix}$$

And after modifying the cost can be stated as:

$$\nabla_{c^*} = \frac{\partial C}{\partial \hat{Y}} = \begin{bmatrix} \frac{\partial c_1}{\partial \hat{y_1}} + \frac{\partial R_{1,1}}{\partial \hat{y_1}} + ... + \frac{\partial R_{m,1}}{\partial \hat{y_1}} \\ \frac{\partial c_2}{\partial \hat{y_2}} + \frac{\partial R_{1,2}}{\partial \hat{y_1}} + ... + \frac{\partial R_{m,2}}{\partial \hat{y_1}} \\ \vdots \\ \frac{\partial c_N}{\partial \hat{y_N}} + \frac{\partial R_{1,N}}{\partial \hat{y_1}} + ... + \frac{\partial R_{m,N}}{\partial \hat{y_1}} \end{bmatrix}$$

Therefore, the only impact the cost modification has on the entire backpropogation procedure is via the *nabla* term, and therefore will affect all network gradients through affecting δ^L directly:

$$\delta^L = \nabla_c \odot \sigma\prime(Z^L).$$

4.5.2 Terms Dependent Directly on Weights: L2 penalty

Consider now regularisation terms that are dependent directly on the weight. We will focus on the L2 penalty (Ridge penalty), that takes the form of:

$$R(w_k) = \lambda \times \sum_k (w_k)^2,$$

where there are k weights in the network. The for any specific weight w_j, the derivative of $R(w_k)$ w.r.t. w_j is:

$$\frac{\partial R(w_k)}{\partial w_j} = 2 \times w_j.$$

Thus, when assessing the new cost, the following holds:

$$C^* = MSE + \lambda \times \sum_k (w_k)^2,$$

and hence:

$$\frac{\partial C^*}{\partial w_j} = \frac{\partial MSE}{\partial w_j} + 2 \times w_j.$$

To back-propagate the cost, the MSE term is back-propagated as normal, but the $R(w_k)$ has to be treated separately, since it does not dependent on the outputs and therefore any activation function transformations. Thus, a matrix of L2 derivative terms for each layer is added to the MSE cost to form the new cost derivatives used in the gradient descent algorithm:

$$dW^{L-r} = (A^{L-r-1})^T \times \delta^{L-r} + dWR^{L-r},$$

where

$$dWR^{L-r} = 2 \times W^{L-r}.$$

4.6 Additional Transformation on Output

Suppose that you wish to transform the output of the NN by means a function that maps the unit bounded outputs of the Sigmoid function to wider range of values. This will have an impact on the gradients of the outputs w.r.t. the parameters and hence will affect the backpropagation directly. Fortunately, by the convenience of the chain rule, only one term is affected by this. Suppose that the transformation on the output is specified by an equation $g()$ such that:

$$\hat{y}_i = g(A_i^L) = g(\sigma(z_i^{(L)})).$$

Consider again equation 16, which lays out components of the derivative of a particular output observation w.r.t. a particular parameter:

$$\frac{\partial C_i}{\partial w_{jk}^{(L)}} = \frac{\partial C_i}{\partial \hat{y}_i} \times \frac{\partial g(\sigma(z_i^{(L)}))}{\partial \sigma(z_i^{(L)})} \times \frac{\partial \sigma(z_i^{(L)})}{\partial z_i^{(L)}} \times \frac{\partial z_i^{(L)}}{\partial w_{jk}^{(L)}}.$$

Notice that the only term that is affected in the above equation is the second term, namely:

$$\frac{\partial g(\sigma(z_i^{(L)}))}{\partial \sigma(z_i^{(L)})} = g\prime,$$

where g' can in general be stated as:

$$g\prime = g\prime(x) = \frac{\partial g(x)}{\partial x}.$$

This term is equal to 1 if no such output transformation is applied i.e. $g(x) = x$. The impact such a transformation will have on the backpropagation equations is only through an alteration to the δ^L term:

$$\delta^L = \nabla_c \odot g\prime(\sigma(Z^L)) \odot \sigma\prime(Z^L).$$

4.7 Gradient Testing

In order to validate the NN and the proper functionality of the NN model, gradient testing can be used. Gradient testing is an exercise to test the computed derivative matrices i.e. dW^l or dB^l for and across layers i.e. $l \in [1, L]$.The exercise requires a gradient measure against which the derivative matrices can be compared. Therefore, you require a rigorous gradient measure. This can be done as follows, using an approximation based approach. Consider a simple function $g(x)$, then:

$$\frac{\partial g(x)}{\partial x} \approx \frac{g(x+\epsilon) - g(x-\epsilon)}{x+\epsilon-(x-\epsilon)} = \frac{g(x+\epsilon) - g(x-\epsilon)}{2 \times \epsilon}.$$

The cost is a function of the outputs which is a function of the parameters, hence for arguments sake we express the cost as a function of any and specific parameter:

$$C = C(w_{jk}^{(l)}),$$

$$\frac{\partial C}{\partial w_{jk}^{(l)}} \approx \frac{C(w_{jk}^{(l)}+\epsilon) - C(w_{jk}^{(l)}-\epsilon)}{w_{jk}^{(l)}+\epsilon-(w_{jk}^{(l)}-\epsilon)} = \frac{C(w_{jk}^{(l)}+\epsilon) - C(w_{jk}^{(l)}-\epsilon)}{2\times\epsilon}.$$

In order to interpret whether the gradients are correct, the dynamic of the approximation relative to the true gradient needs to be considered. The absolute value of the true gradient under assessment can be classified into three buckets:

(i) Less than 1,

(ii) Equal to 1,

(iii) Greater than 1.

If the true gradient is greater than 1, the cost is changing by more then the change in the weight, therefore, the approximation, having a larger change than the infinitesimal change of the true gradient will result in having a larger value then the true theoretical gradient.

If the true gradient is less than 1, the cost is changing by relatively less than the change in the weight, therefore, the approximation, having a larger change than the infinitesimal change of the true gradient, will result in having a smaller value then the true theoretical gradient.

If the true gradient is equal to 1, the approximation will be equal to the theoretical gradient in absolute value. Thus, the theoretical gradients can be compared against the approximate gradients and validated by the following procedure:

(i) Determine the theoretical gradients (done via a normal single forward and backward parse),

(ii) Determine the approximate gradient:

- Load the initialised weights with $+\epsilon$, parse forward and determine the cost,
- Load the initialised weights with $-\epsilon$, parse forward and determine the cost,

(iii) Using the above 2 cost outputs and ϵ to approximate the gradient,

(iv) Check if the absolute value of the theoretical gradient is less than, equal to or greater than 1, and then confirm that it is greater than, equal to or less than the approximations.

Chapter 5

Methodology

5.1 Overview

In principle, NNs provide a number of benefits over a traditional structured model approach. Most notably, they often presents a greater capability of modelling complex and non-linear relationships between features and responses. However, the aforementioned lack of interpretability and difficulty of enforcing real-world dynamics limits their practical use due to the model lacking essential dynamics such as an explicit self-consistent configuration. In this dissertation we endeavour to train a NN model that is robust in producing the BSM prices of puts and calls in one model, across a broad range of economic climates (r, q, σ) and instrument-specific parameters (T, S, K). The experiments outlined herewith are set up to determine an efficient hyperparameter configuration, and more specifically to investigate approaches to enforce constraints that need satisfaction to bridge the gap from a being theoretical construct to posing a pragmatic and economically sound valuation model.

Formally expressed, the objective of this dissertation is to construct a NN model that is:

1. Sufficiently efficient and capable of learning the pricing model put forth by BSM for European call and put options;
2. Enforcing economic constraints with reference to relative and absolute bounds that call and put option values must satisfy.

The ability to reproduce the BSM valuations is in its own right a validation of the trained NN model. Previous studies have shown that NN's are capable of this for call options with certain range limitations on the inputs i.e. maturity and "moneyness". This ensures that the baseline structure of the NN model will have the capability to value options under a Risk-Neutral measure, as achieved under BSM. Thereafter, enforcing the relative and absolute bounds of the NN model output ensures that there is a degree of pragmatism and self-consistency in producing the model outputs.

In essence, this pertains to being able to use pre-visible features (and to be clear here, we do not mean a pre-visible response, nor predictors sourced from forward in the time-line, only those which are available at valuation date of the contract) to determine the market value of options. The market value itself is a function of underlying factors that are not deemed to be deterministic, and is subject to a systemically changing stochastic process, as the market climate and economic environment changes. For example, market price levels and, perhaps more importantly, volatility (historic or implied) of a specific asset have the ability to contextualise the market climate across an appropriate observation/investment window for that specific asset.

The volatility of options with different underlying assets generally differ. Therefore, in order to produce a valuation model that remains temporally relevant and allows for varying volatility in options based on different underlying assets, the training and testing data sets are to be comprised of varying volatilities. This in a simplistic sense captures the specific volatility state of a specific asset, as well as varying volatility regimes.

Given the ability of the NN model to value options across a range of market climates and instrument specific environments, coupled with the possibility of being trained to account for economic constraints (relative, upper, and lower), such a NN model presents a case for practical use above merely being a theoretical construct only. Thus, such a NN model could possibly be used in further applications such as hedging and portfolio management, where the model performance is relied upon frequently and periodically.

5.2 Generating Experimental Data

For purposes of conducting the experiments in this study, data is simulated for each of the inputs of the BSM model, and hence BSM option prices are simulated. Therefore, it requires random specifications for the underlying asset price (S_0), strike (K), risk-free rate (r), dividend yield (q), time till maturity/expiry ($T-t$) and volatility (σ) in order to produce simulations of the input features and the outputs (labels) on which to train the NN.

The values of $S_0, r, q, T-t$ are determined by a simple simulation using a uniform random number generator, $Rand()$, which generates values between 0 and 1. Scaling and shifting the $Rand()$ distribution therefore allows for the simulation of numbers from different uniform distributions:

$$S_0 \sim \mathrm{U}(10, 100).$$

$$r \sim \mathrm{U}(0.02, 0.04),$$

$$q \sim \mathrm{U}(0, 0.03),$$

and;

$$T \sim \mathrm{U}(0.25, 3).$$

Therefore, the time scale is a quarter year to three year maturities, since $T-t$ is measured in years and concurs with a 365.25 day year. At this stage it is worth pointing out that the simulation of the input variables are a simplification of reality, and likely to unrealistic w.r.t. the distributional properties. However, the ranges within which they are simulated are generally realistic ranges.

5.2.1 Strike and "Moneyness"

"Moneyness" of an option typically refers to how near the underlying is to the strike (K). "Moneyness" is often referred to the relationship of S_t to K, representable by S_t/K. The broader the range of S_t/K, the more in, out and at-the-money options constitute the data. "Moneyness" is however not affected only by the relationship between S_t and K, as it is often used as a simplistic contextualisation of how in, out or at-of-the-money an option is. Instead, the other option parameters play a profound role in how in or, out or at-of-the-money an option is. Volatility and maturity are two notable parameters that have a large effect on "moneyness": Consider an option with S_0/K within 95% and time to maturity of 1 day is typically more out the money then one with S_0/K within 80% with a longer time till maturity. Thus, as a more robust measure of "moneyness" and hence a framework within which to classify the range of "moneyness" a NN is trained for, is set out herewith. Consider $N(d_2)$ in Equation 2.3:

$$N(d_2) = P(S_T > K).$$

In other words, the probability of being in the money at maturity for calls. Similarly, $1 - N(d_2) = N(-d_2)$ is the probability of puts being in the money at maturity. Therefore, $N(d_2)$ and $N(-d_2)$ can be used as a measure of "moneyness". We will use it to define the range of "moneyness" that determines how K is simulated and used for this study.

Let $N(d_2) \leq 0.25$ refer to out-of-the-money call options. Therefore, the following expression can be used to determine an upper bound for K when simulating its values:

$$K_L = S_0 \times \exp -[N^{-1}(0.25) \times \sigma\sqrt{T} + (r - q + 0.5\sigma^2)T]. \tag{5.1}$$

Similarly, let $1 - N(d_2) \leq 0.25$ denote the condition that a put option is out-the-money, and thus the following expression can be used to determine a lower bound for simulating values of K:

$$K_U = S_0 \times \exp +[N^{-1}(0.25) \times \sigma\sqrt{T} + (r - q + 0.5\sigma^2)T]. \tag{5.2}$$

Thus, given that the other parameters are simulated first, coupled with the above expressions, K has a range to simulated from:

$$K \sim \mathrm{U}(K_L, K_U).$$

5.2.2 Volatility

This is perhaps the most important parameter in the analysis of option prices. It essentially reflects the market climate and the markets assessment of the specific risk component of a particular asset. The higher its value, the more probable are larger movements in the underlying. The random number generator used above is not appropriate, since it is a symmetric (uniform) number generator. Volatility is largely asymmetric, typically having a positive or right skewed distribution.

The VIX index, being volatility parameter solution that sets actual market prices as the output of BSM for a range of call and put options on the S&P500 index. Using it in BSM will therefore generate some realistic pricing process of call and put options. Therefore, the distributional form of the VIX represents the measure of volatility under a risk-neutral valuation, exhibited in Figure 5.1.

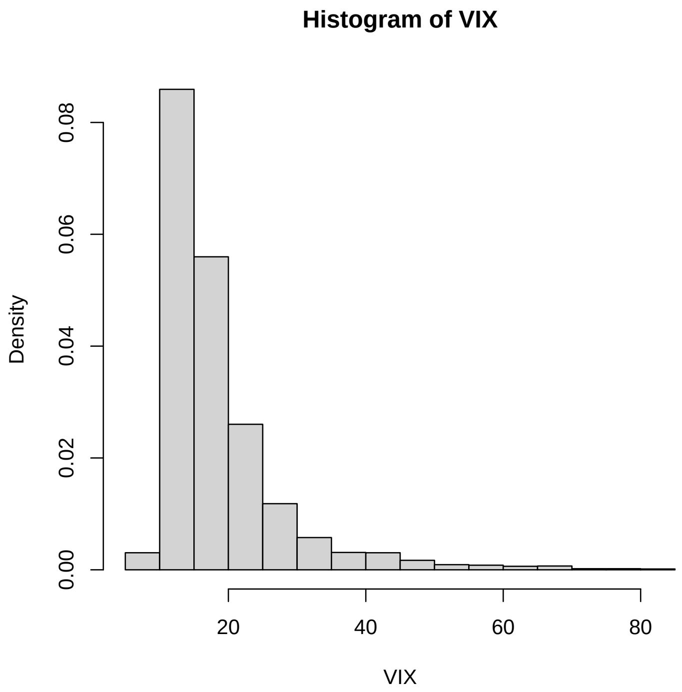

FIGURE 5.1: Histogram of VIX Closing Values: January 2004 - April 2020.

Using the `fitdistrplus` package in R yields the following probability distribution function to generate values with for the volatility parameter:

$$\sigma \sim \text{Gamma}(\alpha = 6.1673, \beta = 3.01378).$$

The distribution was fit on a data set that contains a variety of volatility regimes. When markets experienced higher volatility, the relating data points form part of the tail of the distribution. Thus, σ is simulated using the cumulative distribution function, F_x, of the above probability density distribution:

$$F_x \sim U(0, 1),$$

and thus;

$$\sigma = F_x^{-1}(U(0, 1)). \tag{5.3}$$

We can say that an extreme volatility environment would come from the higher percentiles of the distribution function, whilst lower volatility environments typically come from the lower percentiles. Therefore, we can also use this distribution to simulate stressed market conditions by simulating data from the higher percentiles i.e. the tail of this distribution. Thus, a distressed market environment can be simulated using the cumulative distribution function as follows:

$$\sigma = F_x^{-1}(U(0.8, 1). \tag{5.4}$$

This yields simulations of volatility from a market climate that is typically stressed.

This approach will be used to simulate stressed testing data for Experiment 3 (described below). The `R` script for this experiment can be found in Appendix A.1.

5.3 Neural Network Configuration

The NN model must be validated and configured to a sufficiently performing model before it can be used in further experiments. The hyperparameter domain within which the configuration will take place is discussed in this section. The universe from which we select an appropriate configuration is summarised as follows:

(a) Number of Layers: 1-3 hidden layers and hence 3-5 layers in total;

(b) Number of Neurons in each layer: between 8 and 48 neurons per each hidden layer;

(c) Cross-validation: 80:20 training: testing (20 % cross-validation set) data split;

(d) Learning rate: 0.05;

(e) Gradient descent: Stochastic gradient descent;

(f) Number of batches: 1;

(g) Sample (batch) size: 30 % of training data;

(h) Cost function: MSE;

(i) Activation function: Sigmoid or Tanh.

To improve the convergence of the NN, some scaling of the simulated data will be undertaken. The purpose of normalizing the input features is to assist the NN in faster convergence by re-scaling the input features, since the model parameters in the initial layer will be heavily scale-dependant. Thus it will allow the relative impact of changing features to be more effectively considered in the optimisation procedure i.e. backpropagation. Re-scaling also serves to assist in selecting a hyperparameter configuration of layers and neurons per layer such that particular configurations don't converge faster merely because they have more parameters to deal with the scale on which input features are measured. The following normalisation is used for the respective input features:

(a) S_0 is divided by 100 i.e. $S_0/100$;

(b) K is divided by 100 i.e. S_0/K;

(c) r is divided by the maximum of its range i.e. 0.04;

(d) q is also divided by 0.04, slightly above its range maximum;

(e) $T - t$ is divided by the maximum of its range i.e. 4.

5.3.1 Experiment 0: Layer Transformation and Output Specification

Before proceeding with the experiments that constitute this study, it is requisite to configure a NN model that is capable of predicting call and put options simultaneously, given the input features. Two possible specifications relating to the layer transformations (activation function) and output layer will be considered.

Option 1

Given the existence of upper and lower bounds, the outputs for calls and puts can be stated as a proportion of the underlying asset and strike price respectively. Therefore the outputs of c/S_0 and p/K are mapped unto the unit range i.e. $[0, 1]$. Thus, output's will be specified as follows:

$$\hat{y}_1 = c/S_0,$$

and;

$$\hat{y}_2 = p/K.$$

Define $O(.)$ as a function that operates on the NN outputs to map them to the actual call and put price range as:

$$O(y_j) = \begin{cases} \hat{y}_j \times S_0, & j = 1, \\ \hat{y}_j \times K, & j = 2. \end{cases}$$

A simple exponential or Sigmoid can be used in the output layer. The Sigmoid by design would therefore enforce both the upper and the lower bound of call and put options, i.e., 0 and 1 respectively. A simple exponential function on the output would however only enforce the lower bound. Using the Sigmoid on the output in the present context may easily lead to saturation (as informed by our experience here), and therefore its benefit of enforcing both bounds may be outweighed by its loss of accuracy and ability to learn effectively closer to the lower and upper bound i.e., 0 and 1 respectively. Tanh and Sigmoid activations for the hidden layers will be used.

Option 2

An alternative is to train the log prices of calls and puts. Therefore, the function f that is learned is the prediction of log prices, and so the actual value model would only require applying a simple exponential function to the outputs i.e. $exp(f)$. Thus, output's will be specified as follows:

$$\hat{y}_1 = log(c),$$

and;

$$\hat{y}_2 = log(p).$$

Define $O(.)$ as a function that operates on the NN outputs to map them to the actual call and put price, then:

$$O(\hat{y}_j) = exp(\hat{y}_j).$$

Log prices will take values of less than 0 for options which are more out-of-the-money and therefore $\hat{y}_j$ will have values of less than 1. Thus, a linear function, as opposed to a Sigmoid, will be applied to the output layer. Tanh and Sigmoid are still considered as the activation functions for the hidden layers.

5.4 Experiment 1: Optimal Hyperparameter Configuration

The first step is to establish a hyperparameter configuration that is suited for the task of valuing call and put options in a single NN model. Several previous studies (discussed in Chapter 2) that set out to learn similar phenomena i.e., BSM, had conducted experiments with fixed maturity inputs. Moreover, they focused the model on one instrument type i.e. European call or put options. The inputs were the same inputs used in a typical BSM input.

The authors used grid search to determine a hyperparameter configuration on which to further conduct experiments. Akin to their approach, the first experiment is to further investigate hyperparameter configurations that show efficient convergence as well as not being over-reliant on a small portion of parameters, to ensure that the NN can generalise well.

The objective of Experiment 1 is to determine the configuration that is relatively more parsimonious, while being able to produce sufficiently accurate predictions. The number of parameters used in the model is affected both by the number of hidden layers, and the number of neurons per hidden layer. Using more hidden layers results in using more non-linear transformations on the data. Using more neurons results in using more parameters to model the data.

Experiment 1 will consist of 2 parts; Experiment 1.1 and Experiment 1.2. Experiment 1.1 will consider hyperparameter configurations in general across different hidden layers with different neurons in each layer. Experiment 1.2 will consider, given the number of hidden layers and neurons, which configuration results in the lowest Cross-Validation Error (CVE), where the prediction error that results from using testing data is referred to the CVE (a.k.a. testing error), using the optimal L2 penalty i.e. using the optimal value of λ. This is done while ensuring the model is able to reach a overall prediction accuracy below a certain error margin on the training data.

5.4.1 Experiment 1.1: General Hyperparameter Configurations

The first part of Experiment 1.1 will consist of sampling random hyperparameter configurations with the following attributes:

1. The number of hidden layers are randomly selected as an integer in $1, 2, 3$.
2. The number of neurons for each layer is randomly chosen as integers in $[1 : 8] \times 6$.

This produces a set of 254 unique permutations of hidden layers and neurons per layer i.e. 254 eligible configuration in terms of hidden layers and neuron per layer.

Each will be assessed in a grid search alike fashion, where the prediction error over a number of iterations is assessed.The optimal configuration will be assessed on the following metrics:

(a) MSE for a Given Number of Iterations;

(b) Number of Non-zero and Significant Parameters;

(c) Total Number of Parameters.

MSE for a Given Number of Iterations

The MSE will measure the overall prediction error of the NN on the training data. The use of SGD will cause some additional volatility in the MSE. This, coupled with the impact of using random parameter initialisation may result in spurious assessments of which NN layer and neuron configuration truly is the best. Thus, the lowest MSE (which can be marginal) will not necessarily point to the best performing model. Furthermore, in the absence of regularisation, one should expect termination of the search in local minima on the objective surface. In consequence, we do expect to see some variation in selection, even amongst similar configurations. Nevertheless, for completeness, we conduct the experiment in this way as a reflection of what is, effectively early stopping regularisation, on a "vanilla" NN.

Number of Non-zero and Significant Parameters

It is essential to select a model that is parsimonious as possible. There is no sense in selecting a model with various near or non-zero parameters that essentially adds no or immaterial value, and will then likely be regularised away when a feature selecting type penalty-based regularisation method like Lasso (L1 penalty) is applied. The metric can be stated as:

$$\sum_{HL}\sum_{i} I(P_{HL,i})$$

where;

$$I(P_{HL,i}) = \begin{cases} 1, & |P_{HL,i}| > Th \\ 0, & |P_{HL,i}| \leq Th, \end{cases}$$

and $P_{HL,i}$ is the *ith* parameter in the *HLth* hidden layer, and Th is the threshold. The threshold for the count of near/non-zero parameters is set to 0.005.

Total Number of Parameters

Finally, the total number of parameters will be assessed. This is a simple arithmetic procedure, where all parameters, including the near/non-zero parameters are considered. NN's have 2 categories of parameters; weight and bias parameters. Weight parameters are simply the summed product of the weight matrix dimensions. The bias parameters is the sum of the number of columns in each weight matrix. The more parameters that are is used, the more costly the NN model is in terms of computational resources/time. Therefore, coupled with the number of significant parameters, attention will also be paid to the number of parameters that is used, as opposed to only focusing on the training error metric.

5.4.2 Assessment

The top 25 configurations will be selected on the basis of achieved training error after 1000 iterations. Thereafter, the selected 25 configurations will be run through a grid of 10 different random parameter initialisations for 1000 iterations. This will remove some of the random noise caused by more or less optimal initialisations.

Next, the "best" 15 configurations with their optimal initialisation set will be extracted. The 15 final configurations will then be trained using their optimal initialisation for 30000 iterations.

Finally, between 5 and 10 top performing configurations can be extracted and passed on to experiment 1.2, where the range of configuration choices will be significantly narrowed. Focus will be paid to selecting a final set that has a good spread of; number of hidden layers, neurons per layer and total parameters.

5.4.3 Experiment 1.2: Optimal Cross-Validated Configuration

Given the results of Experiment 1.1, Experiment 1.2 sets out to extract the optimal hyperparameter set with which to further conduct experiments. Experiment 1.2 will consider not only the training data set, but also the testing (validation) data set. Recall that we need to distinguish between training and testing data. Testing data informs model choice and thus the performance of the model on unseen data, i.e. the out-of-sample performance. This is essential when assessing how well any specific model performs at prediction, specifically their ability to generalise over data sets when predicting, and therefore achieve lower (relatively) CVE scores. To ensure that a model generalises well and has lesser over-dependencies on specific parameters, an L2 penalty is introduced in training the NN's.

The L2 norm, commonly referred to as Ridge penalty, yields a smooth cost function as opposed to the L1 norm. It performs "shrinkage" on values, where it sets out to "break" over-dependence on a few parameters, while the L1 performs both "shrinkage" and "weight elimination", it does not yield a smooth, differentiable cost like the L2. The control parameter for L2 is commonly referred to as λ, which typically is a decimal value that scales down the L2 penalty when it is added to the overall cost and before it is backpropagated. An optimal value for λ for a specific NN configuration exists. Experiment 1.2 sets out to find the optimal value of λ for the final set of configurations under consideration. Therefore, to formalise this, the following will be done:

(a) Train each of the resulting NN configurations with L2 regularisation penalty applied across a range of λ values, for 30000 iterations;

(b) Assess the CVE for the various values of λ and determine the optimal configuration and corresponding λ value;

(c) Use the optimal λ for each configuration and train the model till either; an absolute prediction error as a percentage of the output variables/labels is below 4% or, 100000 iterations has been reached;

(d) Extract the NN configuration that converges best within 100000 iterations.

The final configuration determined will be used in the next experiment, whereby the option valuation dynamics and economic constraint enforcement is considered and assessed.

5.5 Experiment 2: Option Bound Penalty Functions for Neural Networks

There are 3 economic constraints that are known for call and put options. These constraints can be stated as bounds, namely; lower bound, upper bound, and relative bound. By design, these are all satisfied by the BSM framework. However, these are not likely achieved by any NN model, and there is no clear approach to explicitly enforce these bounds as with a typical closed-form solution like BSM. The lower bound is enforced by the output transformation for both options (standardised or log price) set out. The upper bound however, is not, for the standardised price NN model. Thus, two penalty approaches are set out and investigated in this experiment, namely; upper bound penalty and relative bound penalty.

Experiment 2 constitutes deriving three additional NN models that follows from the final NN configuration determined in Experiment 1. The three models to derive are as follows:

(a) NN trained with a upper bound penalty;

(b) NN trained with a relative bound penalty;

(c) NN trained with a upper and relative bound penalty.

Similar to λ in the L2 penalty approach in Experiment 1.2, it is essential that a control parameter is introduced for each of the three models. The control parameters ensure that the penalty is applied in an optimal fashion during training, ensuring that the primary objective of learning of option valuations is not compromised by the introduction of penalties.

Experiment 2 will thus be conducted by training each of the models using a grid of control parameters. The optimal values will be assessed viewing the penalty applied, where the penalty should be close to zero for an optimal control parameter, in conjunction with a CVE measure, similar to experiment 1.2. Below, the specifics of each penalty function, their cost surface implication and mathematical expression is set out.

5.5.1 Upper Bound

To enforce the NN to adhere to an upper bound, a penalty approach will be used. The penalty will be added to the overall cost which is backpropagated. It is important to ensure that the penalty that is introduced does not dominate the cost surface and therefore deter the optimisation of prediction error, consequently stifling the learning process of the NN.

Upper bounds for calls and puts can be stated without the discounting component and therefore will be slightly higher then the theoretical upper bound. Given a low or negative interest rate environment, the theoretical use of discounting may be more complicated. Therefore it is neglected. However, there is absolute certainty that calls and puts have a strict upper bound of S_T and K respectively. Formally expressed:

$$c < S_T,$$

and;

$$p < K.$$

For calls and puts respectively. At this stage, a penaliser, $R_u()$ is defined as the penalty for the upper bound:

$$R_u() = R_u(h(\hat{y}_j)),$$

where $j = 1$ for calls and $j = 2$ for puts. The upper bound is to be structured in such a way that the penalty remains relatively flat for predictions below the upper bound, and increases exponentially for predictions above the upper bounds. Additionally, the function $h()$ is used to standardise predictions such that the upper bound for both calls and puts is re-scaled to a value of 1.

Mathematics of $R_u(h(\hat{y}_j))$

It is desired for the bound penaliser to have a certain set of attributes, which will produce a penalty that is added to the MSE cost in a way that ensures when the bound is breached, a severe penalty is applied, while if outputs are not at or above the upper bound, no penalty (or a constant amount) is added to the cost for all such predictions.

To produce a penaliser that is easily implementable and applicable to all observations, the function $h()$ is set up to map NN outputs ($\hat{y}_j$) to the range of c/S_0 and p/K, which is an input to the penaliser. Therefore, the penalty should increase exponentially for values at and above 1, while remaining flat for values below 1. Herewith, standardised prediction refers to $\hat{y_1}/S_0$ and $\hat{y_2}/K$ for calls and puts respectively. Using the standardised predictions allows a simpler R_u to be defined, since all standardised predictions are penalised at and above 1. The attributes R_u can be formally stated as follows:

(a) When the standardised prediction is above 1 (upper bound), then R_u increases exponentially;

(b) When the standardised prediction is between 0 (lower bound) and 1, then R_u remains relatively constant;

(c) R_u is monotonically increasing and;

(d) $R_u \leq \infty \forall h() \in \mathbb{R}$.

The last mentioned attribute is important, since it will ensure that the NN learns with the penalty applied without resulting in numerical issues. Thus, it is required to set out a function with parameters and equations equal to the number of parameters that is to be met in order to parameterise and clearly define R_u. The above attributes can be achieved by using a function for R_u that stems from the Sigmoid function.

The Sigmoid function satisfies the above requirements. By shifting and scaling the Sigmoid, a function can be found that suffices for the purpose of R_u. Thus, R_u is defined as follows:

$$R_u(h(\hat{y})) = \frac{1000}{1 + e^{-(10(h(\hat{y})-2))}}. \tag{5.5}$$

And hence, its derivative R'_u is defined as:

$$R'_u(h(\hat{y}) = \frac{10000}{1 + e^{-(10(h(\hat{y})-2))}} \times (1 - \frac{1}{1 + e^{-(10(h(\hat{y})-2))}}). \tag{5.6}$$

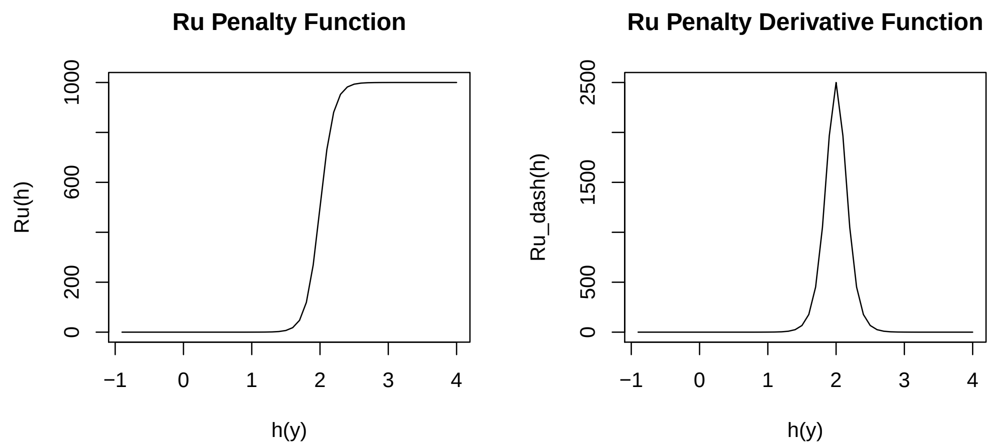

FIGURE 5.2: Plots of R_u and R'_u

Figure 5.2 exhibits the form of the upper bound penalty function and its derivative. Clearly, R_u satisfies the desired attributes set out above. The outputs of R_u is set out in Table 5.1. It is also clear that the derivative function increases substantially when $h(\hat{y})$ is greater than 1 i.e. the upper bound has been breached.

	h(y)	Ru	Ru_dash
1	0.0	0.000	0.000
2	0.5	0.000	0.003
3	0.9	0.017	0.167
4	1.0	0.045	0.454
5	1.2	0.335	3.352
6	1.5	6.693	66.481
7	2.0	500.000	2500.000

TABLE 5.1: Upper Bound Penalty Function

Given the 2 options described above (standardised price and log(price)) outputs, coupled with section 4.5.1, it is necessary to define the gradient of $R_u()$ w.r.t. the outputs $\hat{y}_j$.

Option 1

$$h(\hat{y}_j) = \begin{cases} \frac{\hat{y}_j}{S_0}, & j = 1, \\ \frac{\hat{y}_j}{K}, & j = 2. \end{cases}$$

And therefore;

$$\frac{\partial R_u(h(\hat{y}_j))}{\partial h(\hat{y}_j)} = \frac{\partial R_u(h(\hat{y}_j))}{\partial h(\hat{y}_j)} \times 1.$$

Option 2

$$h(\hat{y}_j) = \begin{cases} \frac{\exp(\hat{y}_j)}{S_0}, & j = 1, \\ \frac{\exp(\hat{y}_j)}{K}, & j = 2. \end{cases}$$

And therefore;

$$\frac{\partial R_u(h(\hat{y}_j))}{\partial \hat{y}_j} = \frac{\partial R_u(h(\hat{y}_j))}{\partial h(\hat{y}_j)} \times \frac{\partial h(\hat{y}_j)}{\partial \hat{y}_j},$$

where;

$$\frac{\partial h(\hat{y}_j)}{\partial \hat{y}_j} = \begin{cases} \frac{\exp(\hat{y}_j)}{S_0}, & j = 1, \\ \frac{\exp(\hat{y}_j)}{K}, & j = 2. \end{cases}$$

Moving onto the implication this will have on the cost surface, the following equation is defined:

$$R_{u_{ij}}' = \frac{\partial R_{uij}}{\partial h(\hat{y_{ij}})} \times \frac{\partial h(\hat{y_{ij}})}{\partial \hat{y_{ij}}}. \tag{5.7}$$

Cost Surface Implication

In order to add the penalty to the cost surface, a control parameter akin to how the λ is used for the L2 penalty. This parameter is used to control the impact of the penalty such that it does not dominate the MSE that is back-propagated, and hence results in sub-optimal overall predictions. The upper bound control parameter will be referred to as β_1. The overall cost function now becomes:

$$C = MSE + \beta_1 \times R_u = MSE + \beta_1 \times \sum_i (R_{ui}),$$

where i represents the index for observations. Then the cost for predictions can be stated as:

$$\frac{\partial C_i}{\partial \hat{y}_i} = (y_i - \hat{y}_i) + \beta_1 \times R_{u_{ij}}',$$

where $j = 1$ for calls and $j = 2$ for puts. Yielding the following total cost matrix that will be back-propagated:

$$\nabla_c = \frac{\partial C}{\partial \hat{Y}} = \begin{bmatrix} (y_{11} - \hat{y_{11}}) + \beta_1 \times R_{u_{11}}'(h(\hat{y_{11}})) & (y_{12} - \hat{y_{12}}) + \beta_1 \times R_{u_{12}}'(h(\hat{y_{12}})) \\ (y_{21} - \hat{y_{11}}) + \beta_1 \times R_{u_{21}}'(h(\hat{y_{21}})) & (y_{22} - \hat{y_{22}}) + \beta_1 \times R_{u_{12}}'(h(\hat{y_{22}})) \\ \vdots & \vdots \\ (y_{N1} - \hat{y_{N1}}) + \beta_1 \times R_{u_{N1}}'(h(\hat{y_{N1}})) & (y_{N2} - \hat{y_{N2}}) + \beta_1 \times R_{u_{N2}}'(h(\hat{y_{N2}})) \end{bmatrix}$$

This will form the new cost that will be propagated backwards in the network for this experiment.

5.5.2 Relative Bound

The relative bound for options refers to the infamous Put-Call parity. The idea to research Put-Call parity consistency when using NN's was put forward by one of the present author's contemporaries, Duncan Saffy - a student completing a Masters on a similar topic concurrently with this one (see Saffy, 2021). Saffy, 2021 provides extensive code and data, however, this study did not reference or use any of the data or code. The Put-Call parity is expressed as follows:

$$c + Ke^{-r(T-t)} = p + S_t e^{-q(T-t)}.$$

Which can be restated as:

$$p + S_t e^{-q(T-t)} - c - Ke^{-r(T-t)} = 0.$$

And dividing the LHS and RHS by K yields:

$$\frac{p}{K} + \frac{S_t}{K} e^{-q(T-t)} - \frac{c}{K} - 1e^{-r(T-t)} = 0.$$

Notably, this expression contains input features, namely; r, q and $T - t$.

Mathematics of $R_{pc}(h(\hat{y}_j))$

Given that the above Put-Call parity expression can be measured for the NN predictions, it is necessary to define a function, h_2, that will measure the Put-Call parity adherence. The outputs $\hat{y}_1$ and $\hat{y}_2$ again denote calls and puts respectively. $h_2(\hat{y}_1, \hat{y}_2)$ will be an input to the penalty function, which will output a penalty, akin to experiment 4.2. Thus, define $h_2(\hat{y}_1, \hat{y}_2)$:

$$h_2(\hat{y}_1, \hat{y}_2) = \frac{\hat{y}_2}{K} + \frac{S_t}{K} e^{-q(T-t)} - \frac{\hat{y}_1}{K} - e^{-r(T-t)}. \tag{5.8}$$

Let $R_{pc}(h_2(\hat{y}_1, \hat{y}_2))$ define the relative bound penalty. The relative bound should be penalised for values not equal to 0. Therefore, a parabola like function with an intercept of $<0, 0>$ corresponding to $<h_2(\hat{y}_1, \hat{y}_2), R_{pc}(h_2(\hat{y}_1, \hat{y}_2))>$ will suffice. The relative bound penaliser, $R_{pc}(h_2(\hat{y}_1, \hat{y}_2))$, is defined as:

$$R_{pc}(h_2(\hat{y}_1, \hat{y}_2)) = (10 \times h_2(\hat{y}_1, \hat{y}_2))^2. \tag{5.9}$$

Yielding a function that is differentiable w.r.t. $h_2(\hat{y}_1, \hat{y}_2)$ and hence $\hat{y}_1$ and $\hat{y}_2$. Thus, for any output (ie $\hat{y}_{ij}$ for $j \in [1, 2]$), the derivative of $R_{pc}(h_2(\hat{y}_1, \hat{y}_2))$ w.r.t. ($\hat{y}_1$ or $\hat{y}_2$), $R'_{pc}(h_2(\hat{y}_1, \hat{y}_2))$ is:

$$\frac{\partial R_{pc_i}}{\partial h_2(\hat{y}_1, \hat{y}_2)} = 20 \times (10 h_2(\hat{y}_1, \hat{y}_2))^1$$

and;

$$\frac{\partial h_2(\hat{y_1}, \hat{y_2})}{\partial \hat{y_{ij}}} = -\frac{1}{K}$$

for $j = 1$, and;

$$\frac{\partial h_2(\hat{y_1}, \hat{y_2})}{\partial \hat{y_{ij}}} = +\frac{1}{K}$$

for $j = 2$.

Moving onto the implication this will have on the cost surface, the following equation is defined:

$$R_{pc}{}'_{ij} = \frac{\partial R_{pc_{ij}}}{\partial h_2(\hat{y_{i1}}, \hat{y_{i2}})} \times \frac{\partial h_2(\hat{y_{i1}}, \hat{y_{i2}})}{\partial \hat{y_{ij}}}. \tag{5.10}$$

Cost Surface Implication

Similar to the upper bound penalty cost surface implication, the relative bound penalty is also added to the overall cost. An important difference to note is that the same relative bound input h_2 is applied to call and put predictions, whereas different upper bound penalties are applied for each call and put observation. The relative bound control parameter will be referred to as β_2. The overall cost function now becomes:

$$C = MSE + \beta_2 \times R_{pc} = MSE + \beta_2 \times \sum_i R_{pc_i},$$

where i represents the observations. The cost for calls can thus be stated as:

$$\frac{\partial C_i}{\partial \hat{y}_i} = (y_i - \hat{y}_i) + \beta_2 \times \frac{\partial R_{pc_i}}{\partial h_2(\hat{y_1}, \hat{y_2})} \times \frac{\partial h_2(\hat{y_1}, \hat{y_2})}{\partial \hat{y_1 i}}.$$

And the cost for puts can be stated as:

$$\frac{\partial C_i}{\partial \hat{y}_i} = (y_i - \hat{y}_i) + \beta_2 \times \frac{\partial R_{pc_i}}{\partial h_2(\hat{y_1}, \hat{y_2})} \times \frac{\partial h_2(\hat{y_1}, \hat{y_2})}{\partial \hat{y_2 i}}.$$

Thus, the following expression is derived:

$$\frac{\partial C_i}{\partial \hat{y}_i} = (y_i - \hat{y}_i) - \beta_2 \times R'_{pc_i}(h_2(\hat{y_1}, \hat{y_2}))$$

for $j = 1$ and

$$\frac{\partial C_i}{\partial \hat{y}_i} = (y_i - \hat{y}_i) + \beta_2 \times R'_{pc_i}(h_2(\hat{y_1}, \hat{y_2}))$$

for $j = 2$. Thus, yielding the following:

$$\nabla_c = \frac{\partial C}{\partial \hat{Y}} = \begin{bmatrix} (y_{11} - \hat{y_{11}}) + \beta_2 \times R'_{pc_{11}} & (y_{12} - \hat{y_{12}}) + \beta_2 \times R'_{pc_{12}} \\ (y_{21} - \hat{y_{21}}) + \beta_2 \times R'_{pc_{21}} & (y_{22} - \hat{y_{22}}) + \beta_2 \times R'_{pc_{22}} \\ \vdots & \vdots \\ (y_{N1} - \hat{y_{N1}}) + \beta_2 \times R'_{pc_{N1}} & (y_{N2} - \hat{y_{N2}}) + \beta_2 \times R'_{pc_{N2}} \end{bmatrix}$$

5.5.3 Upper and Relative Bound

This penalty application to the NN is similar to the above. The control parameter that applies to the penalty is called α, and is applied to the sum of the 2 penalties. Therefore, with the exception of α, everything else is the same.

Cost Surface Implication

The overall cost function now becomes:

$$C = MSE + \alpha \times (R_u + R_{pc}) = MSE + \alpha \times \sum_i (R_{ui} + R_{pc_i}).$$

The ∇ term that is back-propagated now becomes:

$$\nabla_c = \frac{\partial C}{\partial \hat{Y}} = \begin{bmatrix} (y_{11} - \hat{y_{11}}) + \alpha \times (R'_{u11} + R'_{pc_{11}}) & (y_{12} - \hat{y_{12}}) + \alpha \times (R'_{u12} + R'_{pc_{12}}) \\ (y_{21} - \hat{y_{21}}) + \alpha \times (R'_{u21} + R'_{pc_{21}}) & (y_{22} - \hat{y_{22}}) + \alpha \times (R'_{u12} + R'_{pc_{22}}) \\ \vdots & \vdots \\ (y_{N1} - \hat{y_{N1}}) + \alpha \times (R'_{uN1} + R'_{pc_{N1}}) & (y_{N2} - \hat{y_{N2}}) + \alpha \times (R'_{uN2} + R'_{pc_{N2}}) \end{bmatrix}$$

5.5.4 Mean Absolute Relative Error

In proceeding with the experiments, the following measure of error is defined. It is necessary to consider the scale of prediction error, specifically in an out right financial application like valuation or pricing prediction. As such, Mean Absolute Relative Error (MARE) is defined as follows:

$$MARE = \frac{1}{N} \times \frac{(\exp(\hat{y}) - \exp(Y))}{\exp(Y)}. \tag{5.11}$$

The use of MARE casts light on the relativity of the prediction error, taking into account the scale of the data label (output variable) itself. Therefore, it is inherently considerate of the risk associated with capital allocation not determined by market capitalisation. MARE will therefore be used as a stopping error condition in the experiments to follow, as well as an error review framework for this and the next experiments.

5.6 Experiment 3: Analysis of the Four Neural Network Models

Experiment 3 will consist of training and analysing the 4 equivalent NN models set out in Experiment 2. With all but the penalty function applied during training held equal, and using the most optimal hyperparameter configuration that was derived in Experiment 1, the impact of using penalties in training to enforce economic constraints will be analysed and assessed. Herewith the data sets used to analyse and investigate the models are set out, as well as the metrics that will be used to assess and investigate each models performance.

5.6.1 Testing Data

The analysis will be based on 2 sets of testing data, both out-of-sample. Therefore, the four models will be assessed on a CVE basis, taking a view on their ability to generalise and produce accurate predictions. The 2 data sets will be referred to as; test data and stress test data. The former will comprise a normal test set, used during Experiment 1.2 and 2. The latter will be a data set that is generated from a "stressed" financial environment. Thus, the test set is a product of an equal data generating environment as the training data. The stress test data set on the other hand, comes from the tail of the data generating environment. It simulates the changing nature of financial markets and thereby provides a simplified means of investigating how the trained models perform when presented with data generated by a part "foreign" but fundamentally similar process. Fundamentally, it is similar because the valuation of options by the market still adheres to a set of global fundamentals such as economic constraints, even when surges of supply and demand occur i.e. the demand for put options at a certain strike range when a liquidity event/sell-off takes place.

5.6.2 Stress Test Data

The stress test data is generated by using the quantile distribution function set out by Equation 5.1. The stress test data is used to assess how the different models perform under an out-of-sample data set that is generated using extreme and unlikely percentiles of the fitted volatility distribution.

The same definition of "moneyness" and strike price derivation is used as set out by equations 5.1 and 5.2. It is therefore important that the values of K is simulated using a full U(0,1), and thereafter the stressed volatilities using U(0.8, 1.0) is generated, which, coupled with the other features are then used to simulate BSM option valuations. Due to the higher volatility used, the BSM valuations that results is likely to occupy a much wider range of values. This will emphasise how well each of the 4 trained models is able to generalise and produce accurate and economically realistic (with reference to the bounds) predictions.

5.6.3 Assessment Metrics

A set of metrics will be used to compare and assess the performance of the four models. The metrics can be broken down into three categories; prediction error, bound breach and computational efficiency metrics.

Prediction Error Metrics

Naturally, the CVE on both data sets will be used. However, a first order statistic like MSE or MARE provides limited insight. Therefore, more distributional aspects of the resulting CVE space will be considered. The following statistics of the test set predictions will be used:

(a) Mean CVE on a MARE basis;

(b) 25th, 50th, 75th and 100th Percentile of MARE distribution.

The mean CVE is primarily used as a training metric and stopping condition i.e., stop training when a threshold MARE is achieved. The 25th, 50th, 75th and 100th percentile metrics therefore provide good clarity on the distributional aspects of the testing error result of a model.

The distributional aspects of each model's predictions are extremely important. This is because, when considering the impacts of model error at a portfolio level, the size of the prediction error distributions' tail describes the financial risk such a model could pose. Using any mean/average based metric makes an implicit portfolio assumption of an entirely balanced portfolio i.e. equal weighting in each of the underlying. In reality, different portfolios will be unbalanced at varying degrees, and hence the actual loss resulting from the model error could be far worse than the average prediction error makes belief.

Bound Breach Metrics

The breach of bounds will also form part of the metrics, elucidating which models perform better at producing valuations that are consistent with economic constraints. Thus, metrics to view upper and relative bound breaching by a model is necessary. The upper and relative bound breach measures are as follows:

$$UB = \frac{1}{2N} \sum_i (Ind_{ub}(h(\hat{(y)}))$$

with;

$$Ind_{ub}(x) = \begin{cases} 1, & x > 1, \\ 0, & x \leq 1. \end{cases}$$

Such that:

$$RB = \frac{1}{2N} \sum_i (Ind_{rb}(h_2(\hat{(y)})),$$

and;

$$Ind_{ub}(x) = \begin{cases} 1, & x > 0.01, \\ 0, & x \leq 0.01, \end{cases}$$

where N is the total number of observations, and i the index of observations. It is expected that these metrics will be more lower, for the models where the upper and/or relative bound penalties were applied in training.

Computational Efficiency Metric

The computational efficiency is not a primary focus of this dissertation. In lieu of training each model to the same accuracy, it is worth viewing what impact the use of penalties has on the training iterations required to reach the desired accuracy on the training set. Therefore, providing insight on the computational efficiency of each tested model.

5.7 Experiment 4: Real-World Experiment

After Experiment 3 is concluded on a simulated data set, we will consider the same NN models on real-world data. The format of the data frame that is required to train the NNs set out in this dissertation is not available out right. Therefore, for each parameter, a possibly different real-world source is required. Fortunately, Haring, 2020, provides a brilliant guide on how volatility surfaces, and hence many of the required parameters can be attained and/or derived.

Haring, 2020 provides a well described and recent procedure to extract a real-world data set using a web scraping functionality using the R packages `rvest` and `quantmod`. In order to construct the data frame, we consider current option data, as opposed to historical option data. The current option data for a variety of underlying assets will be used. The underlying assets will form some of the more liquid Exchange Traded Funds (ETF's) and large cap stock on which options with sufficient liquidity is traded on option exchanges. Herewith, each feature and how it is attained/derived is set out.

5.7.1 Underlying (S_0) and Dividend Yield (q)

The underlying (stock price) and dividend yield is attained using the `quantmod` package, scraped form Yahoo finance.

5.7.2 Strike, Option Prices and Time to Maturity/Expiry ($K, c, p, T - t$)

This is attained using the `rvest` R package. These values are web-scraped in a table format. Separate tables for calls and puts are scraped. The call price (c) and put price (p) both have a bid and ask values, and for the purposes of constructing the data frame, we use the mid-market value i.e. average of bid and ask for a specific option. The calls and puts are then merged based on the values of K and $T - t$ i.e. strike and time to expiry. This produces a data frame where respective call and put values with the same expiry and strike are in one data frame, as is the simulated data and required to have one NN that predicts prices for both puts and calls in one model.

5.7.3 Risk-Free Rate (r)

The risk-free rate is often assumed flat in many real-world exercises. However, the risk-free (real or nominal) yield curve in reality has a slope. Therefore, a yield curve based approach is taken. The constant maturity yield curves for the 1 month, 3 month, 6 month, 1 year, 2 year, 3 year, 5 year, 10 year, and 20 year treasury bonds is scraped from the St.Louis Fed Reserve website (FRED economic data) using the `quantmod` package in R. Thereafter, a smoothing spline will be fit through the these to produce a yield curve function that more provides the risk-free nominal rate relating to a particular time to maturity.

5.7.4 Implied Volatility

BSM framework provides an ability to engineer the implied volatility of options, from which interpretation and intuition is naturally applicable. Therefore, given that we have all the real-world features that form input to the BSM option formula, with the exception of the volatility (σ), we can derive the implied volatility for any option using an iterative optimiser like Newton-Rhapson.

An implied volatility for puts and calls with the same strike, maturity etc., will therefore be derived, using the call and put option BSM price formulae. In the real-world, we would not expect these to be equal, or narrowly aligned in many instances, since the real-world is known not to follow the same set of assumptions and axioms that underlie a construct like BSM. Therefore, a call and put implied volatility feature will be derived and accordingly form as a feature of the real-world data frame. This sets out the major difference between the simulation based modelling exercise and the extension to the real-world. Two distinct volatilities will form part of the input features for the real-world data set. BSM assumes that the returns of log prices follows a normal distribution with a volatility parameter, however, in the real-world, this assumption does not hold, and option markets dictate the value of a derived implied volatility.

Finally, relating to implied volatility as an engineered feature, will result in "data leakage", which is the notion whereby some signal in the data is lost due to deriving this feature from other, raw data. The implied volatility parameter is however paramount to this study, and arguably to the pricing of options. Thus, the "data leakage" is accepted as a reasonable price to pay given the fundamental importance of this input.

5.7.5 Moneyness Filter

Once all the model input features are derived, we can now calculate $N(d_2)$ and $N(-d_2)$ using the other features of the real-world data frame, and therefore apply a consistent "moneyness" definition filter on the real-world data set as was applied to the simulated data set i.e. $N(d_2) > 0.25$ and $N(-d_2) > 0.25$.

5.7.6 Real-World Bounds

The upper bound that is stated above should hold for real-world data. The time value of money (r) effect was removed, which could have been a driver of some inconsistency, however, no rationale option market participant would for example pay more for an option than the value of the underlying (for calls).

Put-Call parity on the other hand is more complex due to many different narratives, liability sets, liquidity etc. Hence we expect the dynamics of bid and offer for different strike and maturity options by a variety of market participants and under different market liquidities to be misaligned at varying and often significant degrees. Thus, an empirical Put-Call Parity mismatch distribution using Equation 5.8. will be reviewed. Therefore, a training threshold for the R_{pc} penalty can be reviewed for the Real-World data set. The assessment of Put-Call parity may also be useful in identifying data anomalies that can be neglected from the data set.

5.7.7 Closing

The NN configuration that is derived through Experiment 0 - 3 will be used in Experiment 4, i.e. the real-world application. Though it can be argued that the fixing of hyperparameters is naive since the process that is modelled is not ergodic, this is a more general problem that is faced by financial processes where the second order process change is empirically substantiated as non-stationary. Thus, proceeding with a fixed hyperparameter configuration sets an implicit assumption, namely; the time horizon over which the methodology applied is such that the process could be approximately stationary. This is typically more likely to unrealistic in the real-world setting than not.

The next section presents the results of the various experiments. Calibrating and using NN's have a wide range of angles and approaches to focus on. The key focus of this paper is to find a sufficiently performing model in order to perform adequate assessments on how to enforce economic constraints. Therefore, it is likely that a better hyperparameter set exists to model option prices with, and finding such a hyperparameter set would therefore require more focus on the various hyperparameters and procedures to optimise this aspect.

Chapter 6

Experiment Results

6.1 Experiment 0

Two options for specifying the NN output were considered, namely; standardised price prediction or log price prediction respectively. The options were considered in using the activation functions Sigmoid and hyperbolic tangent in the hidden layers. The first option was incapable of being able to effectively train to price deep out-the-money options, specifically put options. The output layer was set to use the Sigmoid and linear function, whereby the former presents the ability to ensure upper and lower bound con. However, it suffers from saturation, where due to the narrow clumping of the outputs by the Sigmoid, it proved incapable of getting an overall prediction performance under 50% within 15000 iterations. Additionally, the Tanh activation seemed to be a better choice for the hidden layers, resulting in generally much better prediction performance and a the range of values prediction fell in closer to the actual range.

Option 2 resulted in far better prediction performance for the same number of iterations and number of parameters. It would after 15000 iterations yield prediction errors consistently below 15%, and often near or below 5% for calls or puts and once for both. The use of the Sigmoid activation proved to have notable poorer performance than the Tanh activation.

Thus, as a preliminary configuration to proceed with, the following was chosen:

(a) Option 2: log price prediction;

(b) Tanh Activation for hidden layers;

(c) Linear output function;

(d) Learning rate of 0.05;

(e) Batch size of 30% for SGD.

6.2 Experiment 1.1

The initial set of hyperparameters described above was used to further endeavour finding an optimal layers and neurons per layer configuration with which to investigate economic constraint/bound inducement. Experiment 1.1 was conducted as what is typically referred to as a hyperparameter grid or grid search technique.

Unique combinations of 1-3 hidden layers and neurons per layer between 0 and 48 were created, resulting in 254 permutations, and hence 254 rows in the initial hyperparameter grid. Each configuration was trained using the preliminary configuration set out in above. These configurations were trained with 1000 iterations each i.e. batches of 30% of the training data was fed forward and the resulting cost was propagated backwards 1000 times using SGD.

The resulting top 25 configurations on a MSE basis was extracted from the grid. Additional to the MSE metric, the number of parameters, number of significant parameters and number of zero parameters were recorded. As set out in section 4.3, we endeavour not only to find the lowest error configuration, but also an efficient one in the sense of having less parameters and a relatively high number of significant parameters.

	Hidden Layers	L1 Neurons	L2 Neurons	L3 Neurons	MSE	# of Parameters	% Zero	% Significant
1	3	40	24	8	0.125	1482	0	99.80
2	1	32	0	0	0.127	290	0	100.00
3	1	16	0	0	0.130	146	0	98.63
4	3	24	16	8	0.139	722	0	99.86
5	3	48	48	8	0.145	3098	0	99.68
6	1	48	0	0	0.146	434	0	99.77
7	2	32	8	0	0.157	506	0	100.00
8	3	48	40	8	0.161	2642	0	99.81
9	3	16	48	8	0.161	1338	0	99.93
10	2	16	16	0	0.163	418	0	99.76
11	2	24	16	0	0.166	602	0	99.83
12	3	24	32	8	0.168	1250	0	99.60
13	2	32	16	0	0.171	786	0	99.75
14	3	24	8	8	0.179	458	0	99.56
15	3	32	48	8	0.184	2218	0	99.91
16	3	32	24	8	0.185	1234	0	99.84
17	3	40	24	16	0.185	1698	0	99.59
18	3	24	40	8	0.187	1514	0	99.80
19	2	48	16	0	0.187	1154	0	99.83
20	3	32	32	16	0.188	1842	0	99.62
21	3	16	8	8	0.188	338	0	100.00
22	3	24	32	16	0.192	1530	0	99.74
23	3	48	40	16	0.195	2986	0	99.67
24	3	16	24	8	0.197	738	0	99.46
25	3	8	24	8	0.200	490	0	98.98

TABLE 6.1: Experiment 1.1: Top 25 MSE Configurations.

In table 6.1 most of the top 25 configurations consisted of NNs with 3 hidden layers, with lesser of 2 and 1 hidden layers respectively. The limited number of iterations at this stage however makes the result susceptible to some initialisations that were a "luckier" random pick than others. However, the range of the MSE, with the top performed at 0.125 and the 25th best at 0.2, does clarify in a sense that the truly better configurations would likely find its way into the top 20 after 1000 iterations, albeit that they were initialised with a relatively less optimal parameter set.

It is notable how the number of parameters in the top 25 varies substantially, from a minimum of 146 to a maximum of 3098 parameters. The number of parameters affects the run time and hence computational cost. Therefore, selecting the configuration with the least number of parameters, given equivalent error performance is consistent with the principle of parsimony i.e. selecting the simplest and most efficient model. Making such a deduction is however more complicated, since a network may appear to perform better but has degenerative learning experience as iterations increase. This can be investigated by viewing how the difference in absolute errors progresses. In other words, difference the error path with itself 1 iteration prior.

Moving on, the selected top 25 layer and neuron configurations were run through another grid search. This time, 10 different parameter initialisations for each was run for 1000 iterations. The best performing error initialisation for each was stored. Thus, some of the noise of sub-optimal random initialisation was reduced. Thereafter, the 25 top configurations were trained for 30000 iterations using more optimal initialisations. The results are again ordered by training error (on a lowest MSE basis). The top 15 performing layer and neuron configurations are tabulated below (Table 6.2).

	Hidden Layers	L1 Neurons	L2 Neurons	L3 Neurons	MSE	# of Parameters	% Zero	% Significant
1	3	24	16	8	0.009	722	0	99.72
2	2	24	16	0	0.012	602	0	99.67
3	3	40	24	16	0.013	1698	0	99.47
4	2	32	16	0	0.013	786	0	99.75
5	3	32	24	8	0.013	1234	0	99.92
6	3	24	8	8	0.013	458	0	99.78
7	3	16	8	8	0.014	338	0	100.00
8	1	16	0	0	0.014	146	0	100.00
9	3	32	48	8	0.014	2218	0	99.68
10	3	48	40	8	0.015	2642	0	99.70
11	2	48	16	0	0.015	1154	0	99.83
12	3	40	24	8	0.015	1482	0	99.73
13	1	32	0	0	0.015	290	0	100.00
14	3	24	40	8	0.016	1514	0	99.74
15	3	24	32	8	0.017	1250	0	99.84

TABLE 6.2: Experiment 1.1: Top 15 MSE configurations using best initialization parameters.

Table 6.2 elucidates a very interesting dynamic, the 1 hidden layer configurations have fallen lower in the table. The range of number of parameters present in the top 15 has also reduced, with the largest amount being 2642.

Notably, all of the configurations have a high proportion of significant parameters i.e. parameters above the value of 0.05. There seems to be a clear inverse relationship between number of parameters and the proportion of the parameters that are significant. The configurations with less parameters generally have the higher % significant score, though the difference is marginal.

At this stage, the final configurations is to be selected from the top 15 configurations set out in Table 6.2. Experiment 1.2 set out to find the optimal L2 parameter (λ) i.e. Ridge parameter, we want to retain the most optimal configurations. The L2 penalty perform parameter shrinkage, and hence it is expected that the % significant score will at least decrease.

The 1st and 2nd configurations in Table 6.2 are selected first since they have relatively less parameters, and the best performing MSE scores. The 3rd configuration will not be used since it has a high number of parameters that is seemingly not necessary based on the top 2 entries in Table 6.2. The 4th, 6th, 7th and 8th configurations are also selected based on having relatively low MSE result and a relatively lower number of parameters. These selections also provide a nice spread across the number of hidden layers (1 to 3), and number of parameters per layer (100 to 800). Thus, these final 6 configurations are taken from Table 6.2 namely; rows (configuration) 1,2,4,6,7 and 8, set out in Table 6.3. The `R` script for this experiment can be found in Appendix A.2.

	Hidden Layers	L1 Neurons	L2 Neurons	L3 Neurons	# of Parameters	% Significant
1	3	24	16	8	722	99.72
2	2	24	16	0	602	99.67
4	2	32	16	0	786	99.75
6	3	24	8	8	458	99.78
7	3	16	8	8	338	100.00
8	1	16	0	0	146	100.00

TABLE 6.3: Experiment 1.1: Final configurations to use in Experiment 1.2.

6.3 Experiment 1.2

The 6 selected configurations are used in a λ grid, where the values of λ are created using an exponential function such that 15 values between 0 and 0.4 are created with a disproportionate amount closer to 0, since this is typically where optimal values of lambda are found.

This experiment is assessed on the CVE basis, where CVE is measured by MSE. Therefore, the configurations are assessed on an out-of-sample basis, where the optimal value of λ can be identified. Each configuration, for a specific value of λ was trained for 30000 iterations. Table 6.4 sets out the results of Experiment 1.2. Generally, the higher the number of parameters, the lower the optimal λ (Lambda Min). This is expected, since the size of the penalty applied is directly affected by the number of parameters. The more parameters, the higher $\sum_i (w_i)^2$ will be in general, and hence a lower λ required to apply the penalty in an optimal manner.

	Hidden Layers	L1 Neurons	L2 Neurons	L3 Neurons	CVE	# of Parameters	% Zero	% Significant	Lambda Min
1	3	24	16	8	0.007	722	0	99.86	0.00021853
2	2	24	16	0	0.008	602	0	99.67	0.00012341
4	2	32	16	0	0.010	786	0	99.75	0.11731917
6	3	24	8	8	0.010	458	0	100.00	0.00673795
7	3	16	8	8	0.011	338	0	100.00	0.20774819
8	1	16	0	0	0.014	146	0	100.00	0.00038698

TABLE 6.4: Final 6 Configurations: Optimal Lambda (L2) by CVE.

Figure 6.1 and 6.2 exhibits how each of the different NN's is descending to a lower point on the cost surface. The variation is a direct result of using SGD. The smaller portion the sample is of the data set, the greater the variation is to be expected, and the faster the training. Figure 6.1 shows the learning for 100 to 1000 iterations. At this early stage of training, configuration 6 (brown curve) is descending to a minimum the fastest.

Figure 6.2 shows the learning progression for 27000 to 30000 iterations. It shows how configuration 1 and 2 have descended to a lower point, and consistently performed the best on the CVE basis. Configuration 6 is not far behind, but on the CVE basis it ranked the lowest. This serves as an early warning that configuration 6 is unable to generalize and may have over-fitted to particular partitions of the training data.

Figure 6.3-6.5 shows the differenced absolute training errors as a percentage of the training error (DAE) at that iteration point. The training errors are difference with itself 1 iteration prior. This is an important visual to consider, since erratically increasing DAEs point to inappropriate configurations that are stagnant, prone to

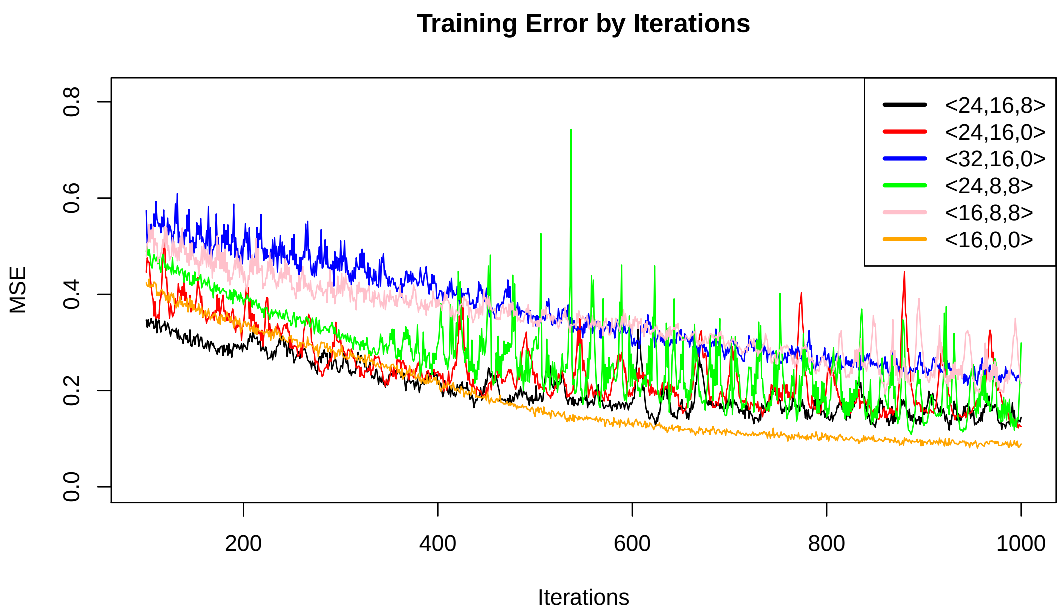

FIGURE 6.1: MSE by first 1000 iterations.

degeneration in accuracy and likely to generalize poorly.

Figure 6.3 and 6.4, and hence the first 4 of the 6 selected configurations points to no alarming outcomes, since the DAEs are generally steeply decreasing at first, becoming flat later on.

Figure 6.5 substantiates the earlier mentioned notion that configuration 6 is inappropriate in performing the task of predicting option prices for this data set. It clearly has an erratic DAE pattern that provides clarity on why its CVE performance is also poor. At this stage, the 6 final configurations are trained on a "do while" basis, where the stopping condition is set to be the earlier of 100000 iterations and a CVE below 4.5 %, where CVE is now assessed using the MARE measure.

Table 6.5 exhibits how configuration 1 and 2 have the ability to effectively converge to a satisfactory prediction ability within 100000 iterations. The other configurations (3-6) don't provide clarity in being effective at predicting option prices. They also appear to be less efficient in terms of training time than configuration 1 and 2.

	Train Error-Call	Test Error-Call	Train Error-Put	Test Error-Put	Train Error-Overall	Test Error-Overall	Iterations
1	3.707	3.950	4.533	4.380	4.120	4.165	82983
2	3.774	4.015	4.711	4.827	4.243	4.421	80079
3	10.583	10.637	10.775	10.784	10.679	10.711	100000
4	12.392	12.370	6.741	6.829	9.566	9.599	100000
5	9.374	9.527	6.129	6.161	7.752	7.844	100000
6	6.921	6.996	6.019	6.028	6.470	6.512	100000

TABLE 6.5: Final Configuration Errors and Iterations: (Prediction-Label)/Label.

Confining focus to configuration 1 and 2, there appears to be no indication of over-fitting since the training MARE and testing MARE do not differ substantially, and the testing MARE is slightly higher then their training counterpart as one would expect. The number of iterations used to reach the goal accuracy of 4 % also differs

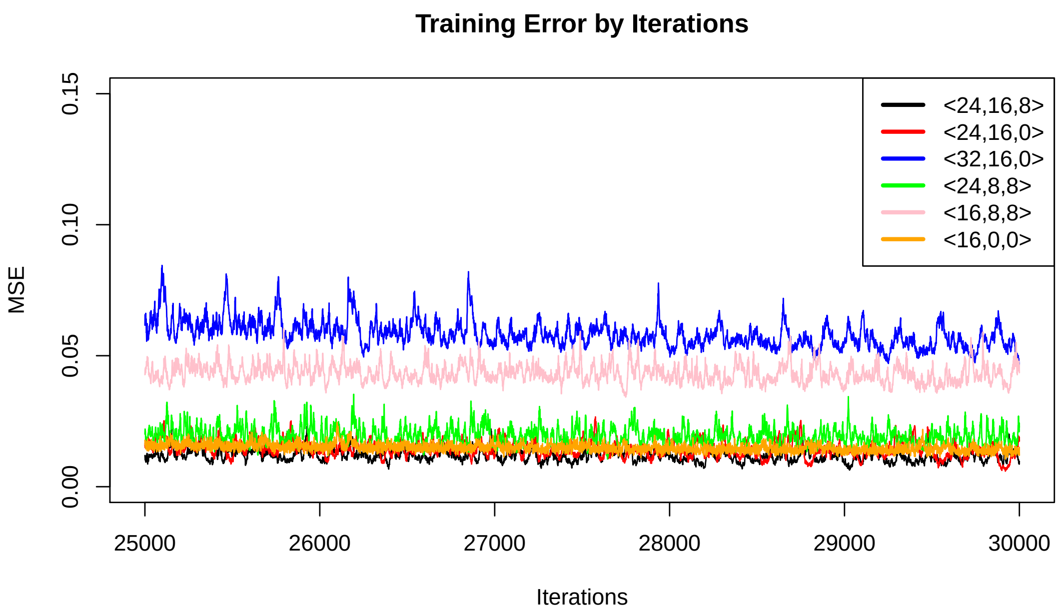

FIGURE 6.2: MSE by last 3000 iterations.

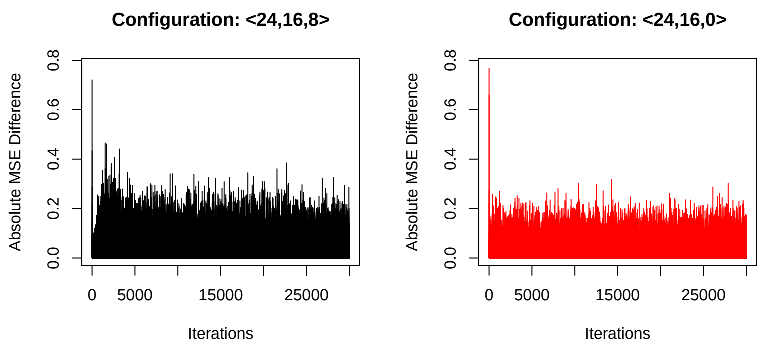

FIGURE 6.3: Absolute Difference in MSE: Configuration 1 and 2.

marginally. However, configuration 1 seemingly presents more balance in the prediction error between calls and puts as opposed to configuration 2. Moreover, configuration 1 appears to generalize better for both calls and puts when viewing the test errors in Table 6.5.

Figure 6.6 and 6.7 exhibits the results of the CVE-λ grid for configuration 1 and 2. Due to the nature of SGD, it is acceptable to experience noise in the CVE by different values of λ. A smoothing spline was fit through each curve of CVE by λ to more clearly elucidate the shape of the curves.

The shape of the spline fitted through the noise in Figure 6.6 validates the existence of a more optimal λ and the effectiveness of applying an L2 penalty in order to produce a NN that is more capable in regularising and producing sufficiently accurate predictions for out-of-sample data.

The shape of the spline fitted through the noise in Figure 6.7 on the other hand,

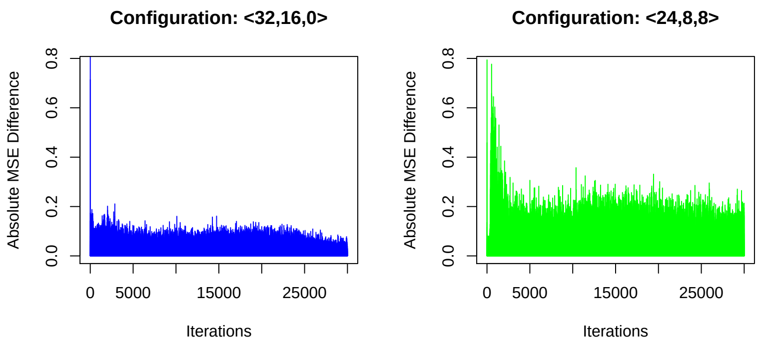

FIGURE 6.4: Absolute Difference in MSE: Configuration 3 and 4.

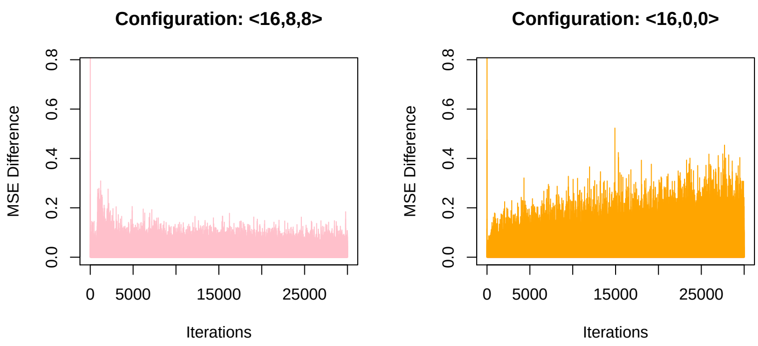

FIGURE 6.5: Absolute Difference in MSE: Configuration 5 and 6.

does not provide a convincing picture. Though the shape is generally consistent with that of Figure 6.6, but much flatter, it is uncertain that the best value of λ for configuration 2 has been derived.

Thus, based on the final 6 configurations derived in the experiment 1.1, coupled with the results set out in Table 6.6, Figure 6.6 and 6.7, configuration 1 will be used. Therefore, a NN with the following hyperparameter set is used going forward:

(a) Log price prediction;

(b) Tanh Activation for hidden layers;

(c) Linear output function;

(d) Learning rate of 0.05;

(e) Batch size of 30% for SGD;

(f) 3 Hidden Layers;

(g) 24, 16 and 8 neurons in layer 1,2 and 3 respectively;

(h) L2 penalty with $\lambda = 0.012$.

Moving on to Experiment 2, where the ability of economic constraint enforcement using a NN model to predict/value call and put options was investigated, the

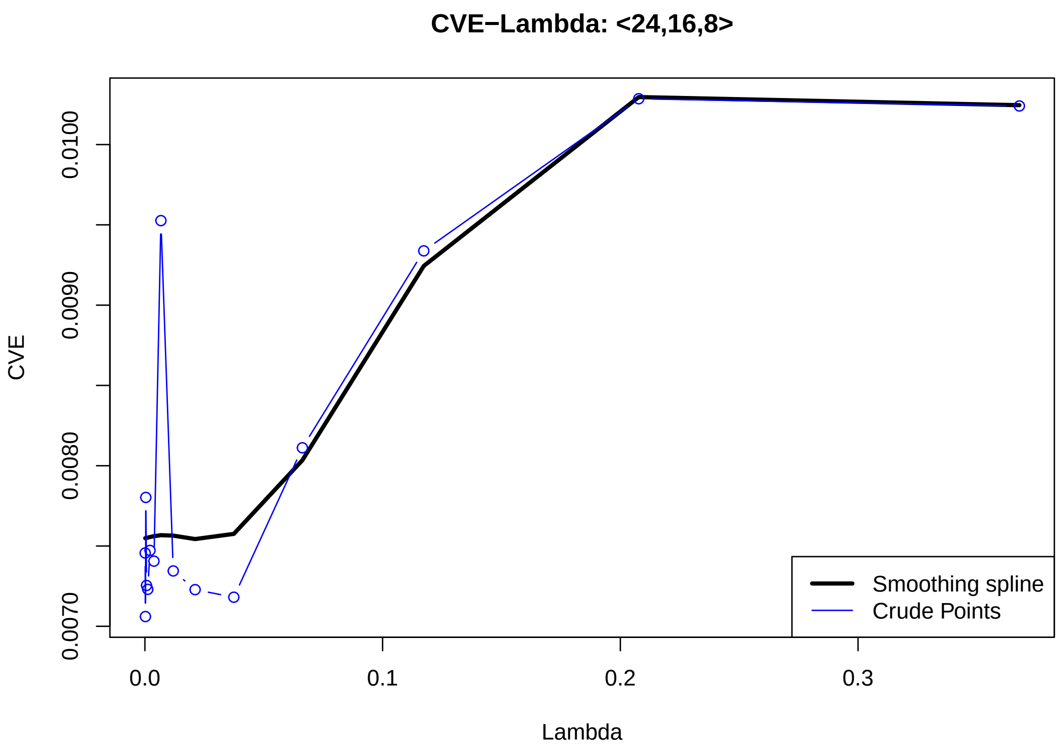

FIGURE 6.6: CVE by λ: Configuration 1.

above hyperparameter set describes the model with which the investigation is to take place. The number of iterations for Experiment 2 will not be investigated, since instead the models will be trained achieve the MARE level of 3 % on the training data, and the number of iterations will be used as an assessment metric when viewing the computational efficiency of the each model in the experiment. The `R` script for this experiment can be found in Appendix A.3.

6.4 Experiment 2 Results

Experiment 2 was also conducted in a grid like fashion. The purpose was to derive β_1, β_2 and α, being the control parameters for R_u, R_pc and $R_u + R_pc$ respectively, as set out in the previous section. Each was trained for a the minimum of 80000 iterations or the number of iterations needed to achieve 3% MARE. Ten equal numbers between 0 and 2 were used to populate the different parameter options, starting with 0.2 and incrementally increasing by 0.2.

Figure 6.8, 6.9 and 6.10 shows the CVE values using MARE as a measure on the primary axis (left) and the penalty cost (R_u, R_pc and $R_u + R_pc$) on the secondary vertical axis. It is essential that a value for each control parameter is selected that achieves a satisfying penalty number i.e. penalty close to zero, and showing promise on the CVE basis (blue line).

Figure 6.8 shows the results for the R_u penalty. Viewing the plot, the shape of the MARE and R_u seems to be relatively correlated, as one would expect. At the lower

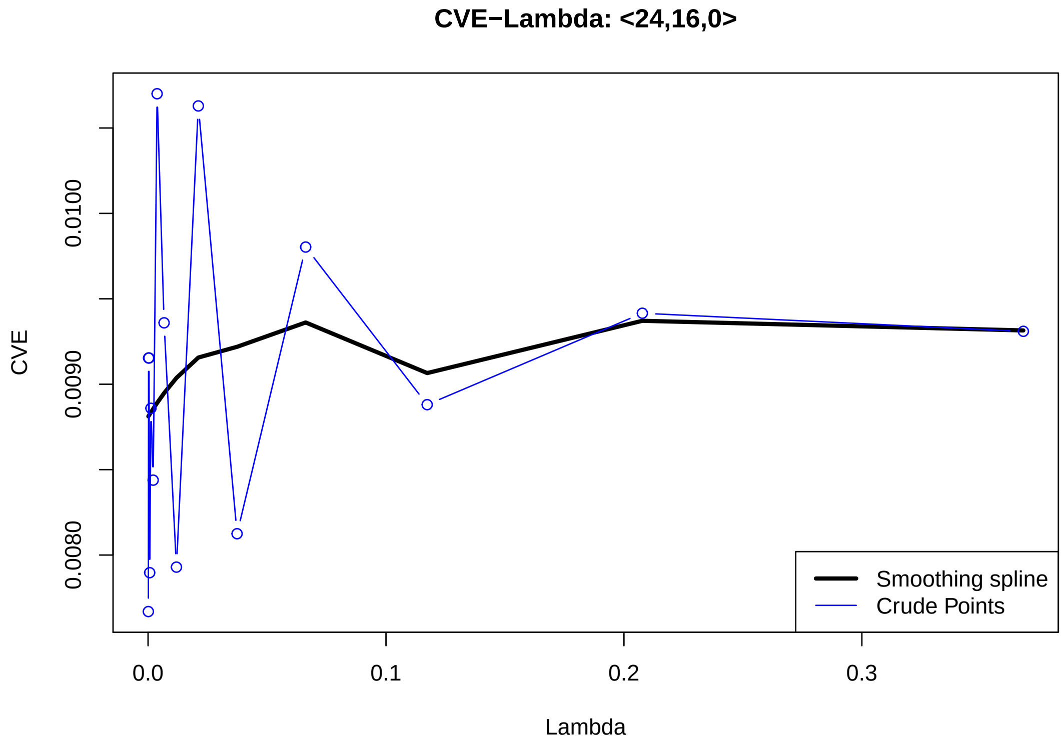

FIGURE 6.7: CVE by λ: Configuration 2.

value range of β_1 (0-0.5), they seem to be negatively correlated, however, this is where the penalty, R_u, achieves its lowest value. At and around 0.5, both the penalty and MARE result seem to be at the best compromise. More weight is attached to ensuring the penalty (red line) is minimised, while using the blue line to assist in selecting a final β_1 value. Thus, the $\beta_1 = 0.4$ will be used in Experiment 3.

Figure 6.9 shows the results for the R_{pc} penalty. The shape of the MARE line (blue) does not appear to follow any specific trend, while R_{pc} (red line) is downward sloping. The R_{pc} line (red) reaches a minimum a β_2 value around 1.6, however it is marginal compared to values of β_2 of 1.6 upwards. Evidently, the optimal value of β_2 is to be found in the range of $\beta_2 > 1.5$. Again, more weight is attached to ensuring the penalty (red line) is minimised, and therefore a value of 2 will be used for β_2. This is because at $\beta_2 = 2$, a very low R_{pc} value corresponds to a slightly larger MARE value when compared to $\beta_2 = 1.6$, which seems to be the result of a noise coming from the SGD usage.

Figure 6.10 shows the results for the $R_u + R_pc$ penalty. The shape of the MARE and $R_u + R_pc$ seems to be relatively more correlated than what Figure 6.9 portrays, similar to Figure 6.8. The lower value range of α is at the left, right and center. This makes sense, since β_1 was at the left end, β_2 was at the right end, with a minimum MARE at 1. Viewing the red line in isolation, it seems that at the higher end of the α range, $R_u + R_pc$ seems to be decreasing, and MARE consistently with it. Therefore, a value of $\alpha = 2$ will be used in Experiment 3.The `R` script for this experiment can be found in Appendix A.4.

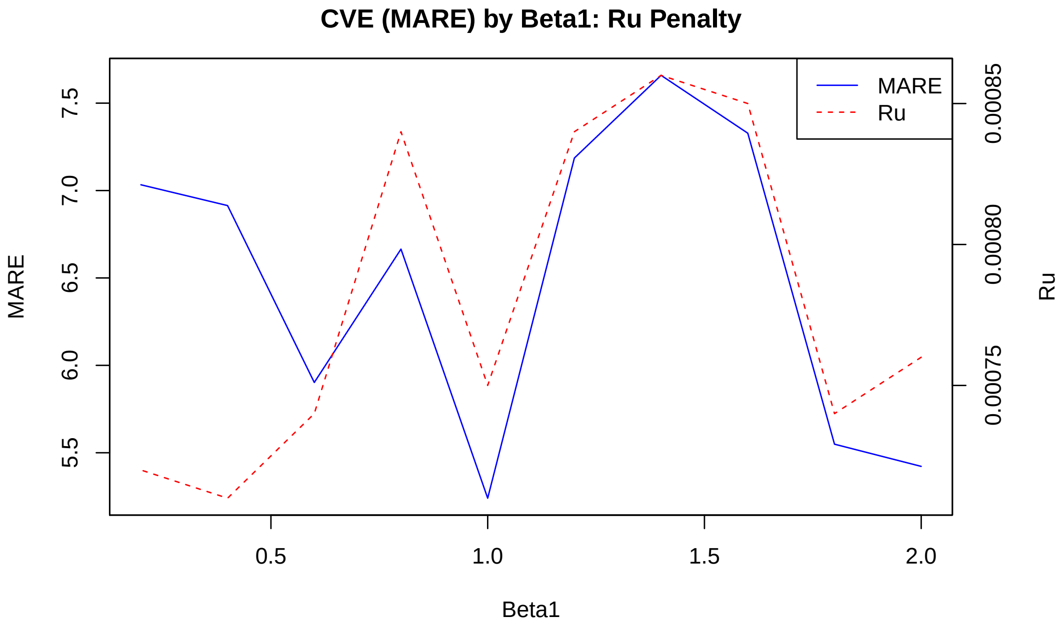

FIGURE 6.8: MARE and R_u plot against β_1 Values.

6.5 Experiment 3

The results of Experiment are set out herewith. A table of results for each of the four models for put, call and both call and put predictions are set out for the 2 cross-validation data sets (testing and stress testing data). The table of results consists of; CVE distributional aspects (MARE, Absolute Relative Error Quantiles), Bound Performance (upper and relative) and Computational Efficiency (iterations to achieve goal) as set out in Chapter 5.

6.5.1 Test Data

The cross-validation results of the 4 NN models is presented in this section. MARE was used as the CVE measure, and the quantiles/distributional aspects of each models error performance is also based on the Absolute Relative Error (ARE) measure. Each model was trained to reach a MARE level below 2% on the training data. The bound performance metrics follows, which is expressed as a percentage of the number of rows i.e. what percentage of the cross-validation (testing) data set breached the bound (upper and relative). The number of iterations required to reach this level is also set out as one of the assessments metrics in the tables below. Herewith, Model 1, Model 2, Model 3 and Model 4 refers to; No Penalty, Upper Bound Penalty (R_u), Relative Bound Penalty (R_{pc}) and Both Penalties ($R_u + R_{pc}$) trained models, respectively.

Call Options

From Table 6.6 it is clear that Model 3 (R_{pc}) performed the best on a MARE basis. When viewing the quantiles, it appears that Model 2 has the lowest 100th percentile and hence the shortest tail in the residual distribution. Model 3 and 4, both having

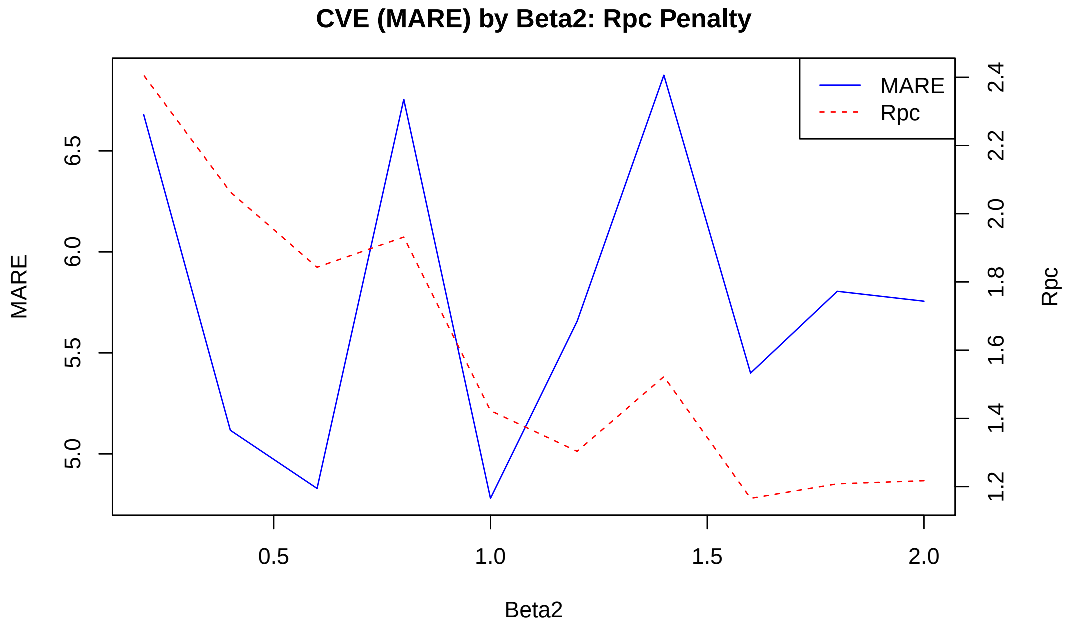

FIGURE 6.9: MARE and R_pc plot against β_2 Values.

the relative bound penalty applied, have much higher 100th percentiles. Viewing the bound metrics, the upper bound was not breached by any of the models. However, the relative bound was. As expected, Model 3 (R_{pc}) and 4 ($R_u + R_{pc}$) performed better by a large margin.

	MARE	25th Perc.	50th Perc.	75th Perc.	100th Perc.	Upper Bound	Relative Bound	Iterations
No Penalty	2.080	0.69625	1.5530	2.68900	24.211	0	2.8	261950
Penalty (Ru)	2.081	0.60400	1.4090	2.70925	19.562	0	2.9	259221
Penalty (Rpc)	1.938	0.52725	1.2910	2.43575	46.830	0	1.2	333670
Penalty (Ru+Rpc)	2.007	0.62975	1.4295	2.48700	46.129	0	1.1	270746

TABLE 6.6: Experiment 3: Call Option Test Data.

Put Options

Table 6.7 presents the corresponding information relating to put option predictions on the testing data set. Table 6.7 paints a different picture compared to Table 6.6, where Model 2 (R_u) is the best performing model based on the MARE score, while Model 1 (No penalty) comes in at second best. Model 1 (No penalty), has the lowest 100% quantile. Notably, Model 1 is the worst performer for the first 2 quantiles for call options and put options, implying that for more than half of the predictions it performs worse than all the other models, often by much. This coupled with its lower 100th percentile in general, still results in some relatively better MARE performance, which averages over the prediction errors.

The bound results for the call and put options are the same. The relative bound is identical, since the same output results for both call and put options, i.e. each call and put option prediction forms an input to this metric (Put-Call parity assessment).

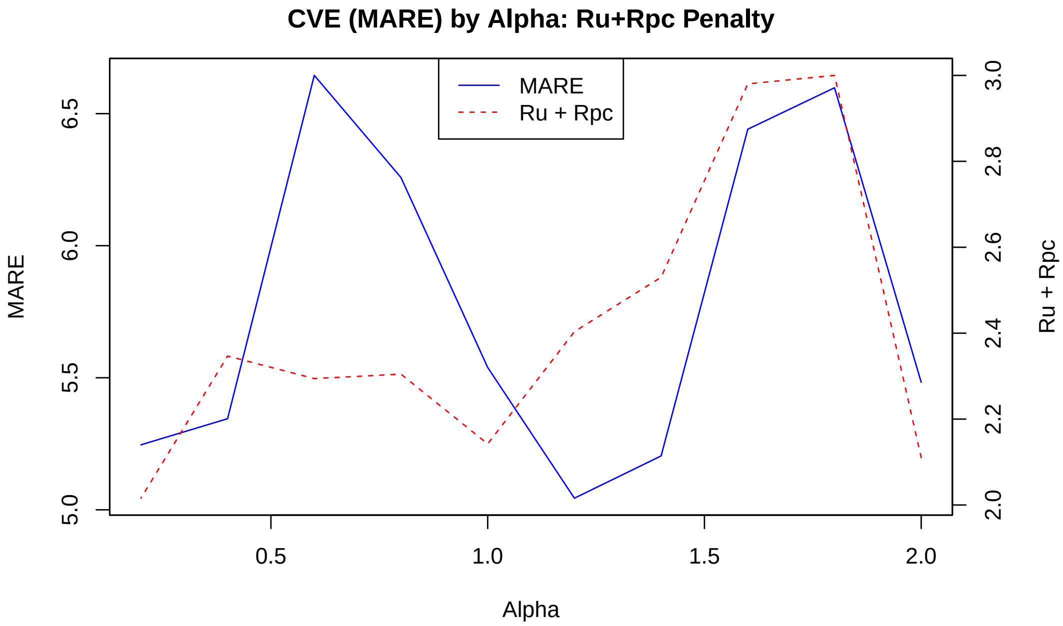

FIGURE 6.10: MARE and $R_u + R_pc$ plot against α Values.

	MARE	25th Perc.	50th Perc.	75th Perc.	100th Perc.	Upper Bound	Relative Bound	Iterations
No Penalty	2.406	0.81750	1.7175	3.10825	40.508	0	2.8	261950
Penalty (Ru)	2.344	0.69375	1.5500	3.16175	44.796	0	2.9	259221
Penalty (Rpc)	2.502	0.71175	1.6425	3.21150	72.527	0	1.2	333670
Penalty (Ru+Rpc)	2.495	0.77100	1.6845	3.13400	86.353	0	1.1	270746

TABLE 6.7: Experiment 3: Put Option Test Data.

Overall: Call and Put Options

Table 6.8 presents the overall table for call and put option predictions, thereby the overall view of each models performance can be gauged. Model 2 appears to have the lowest MARE score (marginally) of the four models, mostly due to its relatively strong out performance on put options. Models 2-4, which had bound penalties applied, all have significantly better performances compared to Model 1 based on the first 2 quantiles (25th and 75th percentiles). Based on the 3rd quantile (75th), the four models had relatively equal performance, while on the 100th percentile basis, Model 1 and Model 2 is much better than Model 3 and 4.

Considering the purpose of the penalties, it appears that the objective to enforce bounds was relatively well achieved. Model 3 and 4 specifically provides clarity on this, since these models were applied the relative bound penalty, which is easily breached even by a model that has a good MARE prediction performance, like we can see with Model 1 in Tables 6.6, 6.7 and 6.8. The next section adds more context to this dynamic, since we deliberately set out to investigate how the bound enforcement dynamic functions for the various models when presented with a data set that is sampled from higher percentiles of the underlying process.

	MARE	25th Perc.	50th Perc.	75th Perc.	100th Perc.	Upper Bound	Relative Bound	Iterations
No Penalty	2.243	0.73300	1.6360	2.88575	40.508	0	2.8	261950
Penalty (Ru)	2.212	0.64975	1.4945	2.91225	44.796	0	2.9	259221
Penalty (Rpc)	2.220	0.62575	1.4715	2.85125	72.527	0	1.2	333670
Penalty (Ru+Rpc)	2.251	0.68750	1.5265	2.81325	86.353	0	1.1	270746

TABLE 6.8: Experiment 3: Put and Call Option Test Data.

6.5.2 Stress Test Data

The stress test data set is generated using the same simulation distributions, however, the volatility parameter is simulated from the extreme end of its distribution i.e. 0.8-1. Therefore, it is reasonable to expect a worse CVE performance from all four models compared to the CVE under the testing data above.

Call Options

Table 6.9 sets out the same metrics for the stress test data set. Interestingly, Model 3 (R_{pc}) had the lowest MARE, followed by Model 4. Model 3 and 4 also consistently performed better on the four quantiles compared to Model 1 and 2. Contrary to the results in Tables 6.6-6.8, Model 1 had the highest 100th percentile under the stress test set, while Model 2-4 had relatively similar scores.

	MARE	25th Perc.	50th Perc.	75th Perc.	100th Perc.	Upper Bound	Relative Bound	Iterations
No Penalty	2.715	0.82800	1.7535	3.3275	32.342	0	11.1	261950
Penalty (Ru)	2.907	1.03775	2.2095	4.0640	23.714	0	14.4	259221
Penalty (Rpc)	2.333	0.80175	1.7820	3.1235	21.633	0	5.5	333670
Penalty (Ru+Rpc)	2.418	0.78550	1.8125	3.2710	22.161	0	5.7	270746

TABLE 6.9: Experiment 3: Call Option Stress Test Data.

Compared with the results presented in Table 6.6, Table 6.8 casts perspective on the how well the models generalised. The fact that the Model 1 achieved such a degeneration in its performance points to that it was likely less regularised and hence less capable of generalizing. A perhaps confounding result is that Model 2-4 saw generally smaller ARE quantile score under the stressed data set. However, when considering the nature of calls, and what the implied volatility represents for the simulated data, it makes sense. Sampling from higher percentiles (upper tail) of the implied volatility distribution would typically mean many calls have a larger likelihood of being in-the-money given that the drift (r) is positive, and so the input feature of higher volatility was " well connected" to the outputs by the parameters in the hidden layers for Model 2-4, and specifically Model 3 and 4. Therefore, the more simple dynamic between volatility and the "moneyness" of calls was effectively learned by these models. Puts on the other hand have a more opaque relationship with implied volatility.

Put Options

Table 6.10 presents the stress test results for put options. Model 3 is again the best performing with reference to the MARE and most of the quantiles, with the exception of the 100th percentile. Model 4 presents a very similar result to Model 3 w.r.t. the MARE and the ARE quantiles, having the worst 100th percentile score. Model 2 on the other hand has the worst performance for the MARE and ARE quantile metrics with the exception of the 100th percentile, where it has the best score. Model 1 does not have a stand out performance relating to any of the error metrics.

	MARE	25th Perc.	50th Perc.	75th Perc.	100th Perc.	Upper Bound	Relative Bound	Iterations
No Penalty	3.330	1.08775	2.2915	3.85600	118.250	0	11.1	261950
Penalty (Ru)	3.264	1.18425	2.3490	4.31400	65.231	0	14.4	259221
Penalty (Rpc)	3.225	0.95125	2.0570	3.97475	138.728	0	5.5	333670
Penalty (Ru+Rpc)	3.248	0.95550	2.0715	4.05350	140.768	0	5.7	270746

TABLE 6.10: Experiment 3: Put Option Stress Test Data.

Considering the quantiles scores presented in Table 6.7 and 6.10, the nature of put options and what is expected under a stressed data set is visible. The 100th percentile score for each of the four models has increased. This makes sense since the implied volatility is generally higher under this test set, and hence the dynamic of "moneyness" and valuation changes sharply, more so than typically for calls. Given this result, and the that portrayed for call option in Table 6.6 and 6.9, the NN models in training have seemingly been calibrated more effectively towards accuracy in call option prices as opposed to put option prices.

Overall: Call and Put Options

Table 6.11 shows the overall performance on the stress test data for each of the four models. The penalty-based approaches perform better overall i.e. looking at the MARE. Model 1 and 2 performs better on the basis of the 100th percentile, Model 2 by quite a margin. When viewing the other percentiles, Model 3 and 4 are the top performing models, having lower MARE and ARE quantile results in general, with the exception of the 100th percentile, caused by poor performance on the put options mostly.

	MARE	25th Perc.	50th Perc.	75th Perc.	100th Perc.	Upper Bound	Relative Bound	Iterations
No Penalty	3.023	0.91650	2.052	3.59525	118.250	0	11.1	261950
Penalty (Ru)	3.086	1.11125	2.269	4.15300	65.231	0	14.4	259221
Penalty (Rpc)	2.779	0.88000	1.880	3.51225	138.728	0	5.5	333670
Penalty (Ru+Rpc)	2.833	0.86275	1.920	3.59950	140.768	0	5.7	270746

TABLE 6.11: Experiment 3: Put and Call Option Stress Test Data.

The upper bound was not breached by any of the models. The relative bound on the other hand, provides more clarity. Model 3 had the best (lowest) relative bound score, with Model 4 narrowly second best. Model 1 and 2, being the models that did not have the relative bound penalty applied, performed significantly worse in absolute terms, both having the relative bound threshold of 1% breached by more than 10% of the test data and generally more than double that of Model 3 and 4.

6.5.3 Overall Comments

Prediction Error Metrics

Considering the MARE and quantile results presented int Tables 6.8 and 6.11, it is evident that the penalty-based approaches in general provide better performing prediction models on a MARE basis. The stress test data (Table 6.8) produced very similar performing models on a MARE basis. However, the penalty-based models (Models 2-4) produced ARE distributions with longer and thinner tails as opposed to Model 1. This was seemingly caused by the put options. In the context of an investment portfolio, or specifically a writer (seller) of both call and put options, this a highly problematic. Since in the context of an unbalanced or concentrated option portfolio, this could expose one to severe financial risk. Be it that such a model is used to only write call options, based on the results here (Table 6.6 and 6.9), Model 3 and Model 4 unambiguously present a more reliable model that generalizes better and exposes the user to significantly less financial and tail risk.

The first three quantiles on the other hand portray a different picture; that Model 3 and 4 have a stronger practical use case for both call and put options. Viewing Tables 6.6-6.11, it is clear that in general Model 3 and Model 4 are the better choices. Model 2 has a strong case only for the put options on test data (Table 6.7), but is generally inconsistent i.e. best for 1 case, worst for others. Model 3 and 4 present very similar error performances, consistently across the 2 test data sets for calls and puts. Specifically, they have generally narrower quantiles, and hence, for most (say 75% of differently balanced portfolios), they would be a better valuation model to rely on for both puts and calls.

Bound Breach Metrics

The upper bound metric can not be assessed beyond the fact that none of the models presented issues in this regard. Model 2 and 4 were trained using the upper bound penalty, and would be expected to perform better on this metric when faced with out-of-sample data that produces valuations near or above the bound. The definition of "moneyness" based on the measure $N(d_2)$ and $N(-d_2)$ is likely the reason that the upper bound was not tested, and therefore provides in itself a validation that using this a definition of "moneyness" at least allows a realistic dynamic with reference to the upper bound.

The relative bound on the other hand has provided some very interesting insights when considering the tables 6.6 - 6.11. The same relative bound % score is in each table, therefore, any table can be viewed for the different data sets. For the test data, Model 4 has the best (lowest) relative bound score, while for the stress test data, Model 3 has the (lowest). Considering the scores relative to Model 1 and 2, it is clear that the objective of enforcing the relative bound has been achieved somewhat by the penalty approach set out. Achieving between 5 and 6 % on the stress test data is relatively satisfactory, given that a very low threshold of 1% was used i.e. $h_2(\hat{y_1}, \hat{y_2}) < 0.01$ for at least 94% of predictions.

Computational Efficiency Metric

The computational efficiency of each of the four models don't seem to differ substantially when viewing the "Iterations" column in any one of Tables 6.6-6.11. All but Model 3 have very similar numbers of iterations that was used to reach the goal MARE of 2%. Given the very similar error performance of Model 3 and Model 4, and the difference between the 2 models being the upper bound penalty applied, this does perhaps propose using the upper bound approach for another reason over and above its primary objective of enforcing the upper bound; faster calibration. Given the relative bound assessments and the marginal difference between Model 3 and 4 on the two data sets with regard to the prediction error and bound breach metrics, this additional and unforeseen benefit may suggests Model 4 is the better one, since it calibrates faster, and is trained to allow for both upper and relative bound constraints.

In the next section we consider a real-world data application. In order to use the above penalties, we will consider empirically how the Put-Call parity holds and derive from it a reasonable threshold. The R script for this experiment can be found in Appendix A.5.

6.6 Experiment 4: real-world Data

As a final experiment, the four models were trained on real-world data and accordingly assessed using the same metrics put forward in Experiment 3. Below, the real-world data set is first described, and some context is placed on the Put-Call Parity for this real-world data set. Thereafter, the same results as Tables 6.6-6.11 are tabulated.

6.6.1 Data

The R script to source and structure the real-world data set can be found in Appendix A.6., using R code written by Haring, 2020 as a starting point. The following underlying assets and relating option information was extracted; QQQ, SPY, AAPL, MSFT, T, GS, JPM, TSLA, F, GOLD, JNJ and IVV. QQQ, SPY and IVV are Exchange Traded Fund (ETF) focused at tracing the Nasdaq, S&P500 and Russel3000 indices respectively. The choice for the remaining symbols was not analytical, but rather on personal whim and selecting from different industries i.e. technology, banking, pharmaceutical etc. Though the ETF tickers are constructed using leverage and other derivatives and hence may not be entirely appropriate, their inclusion largely supplemented the data in terms of adding more observations.

The data that was scraped then had implied volatilities for call and put options calculated separately and appended to the data set. This was used to calculate the "moneyness" using BSM's equation for $N(d_2)$ and $N(-d_2)$ for calls and puts respectively. Thereafter, consistent with Experiment 3, the data set was filtered for values above 0.25. Finally, the Put-Call parity score is calculated using $h_2(\hat{y}_1, \hat{y}_2)$ (described in Chapter 5.5.2). The data points with absolute value scores above 0.1 was filtered out. Table 6.12 exhibits the constituents of the final data set.

	Stock	Data Points
1	AAPL	125
2	JNJ	28
3	MSFT	47
4	QQQ	660
5	SPY	850
6	T	13

TABLE 6.12: Experiment 4: Real-World Data Constituents.

Figure 6.11 exhibits the resulting real-world Put-Call parity score distribution using $h_2(\hat{y_1}, \hat{y_2})$. It is evident that the Put-Call parity holds within some range of values. For the purposes of this exercise, this range was assumed to be anything between -0.1 and 0.1.

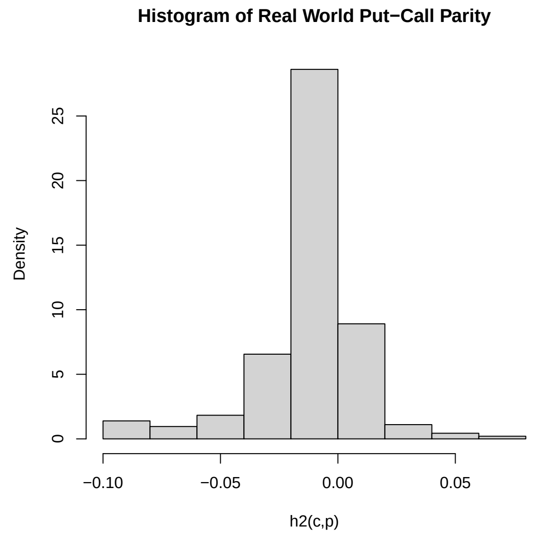

FIGURE 6.11: Real-world Put-Call parity score using $h_2(\hat{y_1}, \hat{y_2})$.

The R script for this experiment can be found in Appendix A.6.1.

6.6.2 Control Parameters

The following NN configuration was used for this deriving the control parameters (β_1, β_2 and α) and assessing the four Models:

(a) Same Neurons and Layers as Experiment 2 and 3;

(b) Tanh Activation Function for hidden layers;

(c) Linear output function for output layer;

(d) Learning rate of 0.07;

(e) Batch size of 70% for SGD.

The NN models that were derived in Experiment 0-3 were based on the simulated BSM data. Thus, it is likely that these are not the optimal NN configurations for the real-world application. Nonetheless, it is likely that they are optimal enough to effectively conduct the real-world experiment with satisfactory performance. The only difference between the NN models here and those in Experiment 3 relates to the input data. The input data in this experiment has an additional feature since an implied call and put volatility is derived as opposed to using a single volatility input. In Figures 6.11-6.13 the respective penalty values are plotted against the control parameters. The range of values for the control parameters were widened to allow for greater complexity in the real-world data when compared to the ranges used in Experiment 2. At this point, we are specifically interested in the values of the control parameters that result in the minimum penalty being applied. The real-world data was more sparse and complex and therefore requisite of training iterations, and hence the plot of MARE was neglected.

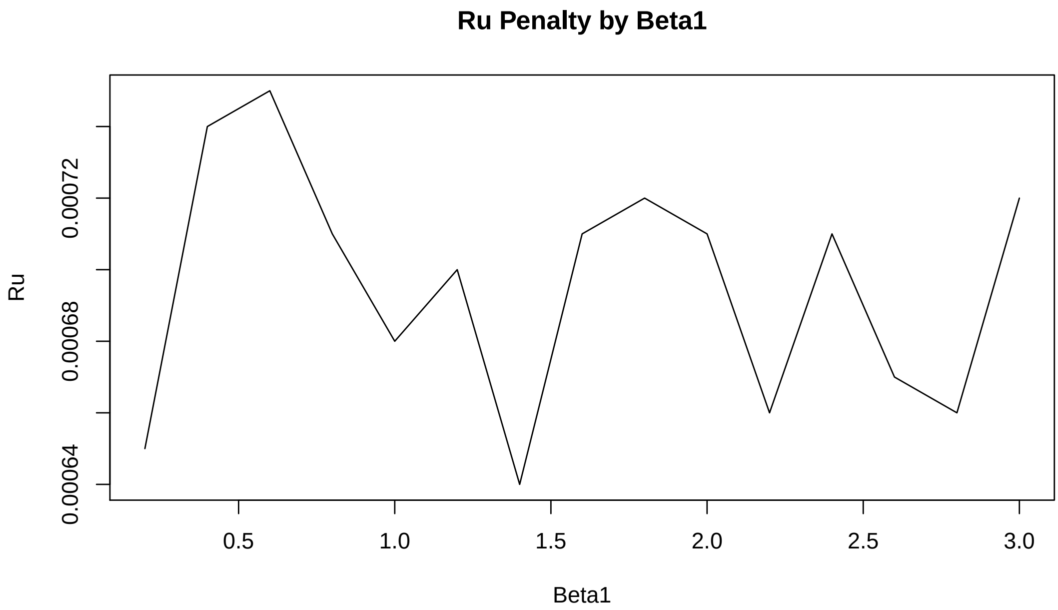

FIGURE 6.12: R_u plot against Beta1 Values.

Figure 6.12 portrays no clear minimum and at the very least it's dominated by noise. β_1 shows little effect but the values of the constraint measure are small, meaning there is little information about that constraint (or violating that constraint) in the data. Towards the center of the β_1 range appears to be a minimum of a mildly visible parabola like shape. $\beta_1 = 1.4$ was therefore chosen.

Figure 6.13 has a remarkably similar shape to Figure 6.9 and compared to Figure 6.12, for certain β_2 values, its constraint is clearly better enforced, supported by the clear parabolic shape. Hence, $\beta_1 = 2.2$ is chosen as the value to use going forward.

Figure 6.13 also appears to have a parabola type shape taking form. Given Figure 6.10 and trying to remain generally consistent, $\alpha = 2.2$ is chosen as the final value to proceed with. This will also allow that the relative bound is penalised at similar scale as in Experiment 2.

6.6.3 Results

The same assessment metrics as in Experiment 3 was used. Herewith the call, put and overall (call and put) results for the metrics as set out in the same table form as Experiment 3. The iterations were limited to a maximum of 500000, of which none of models realised the goal (MARE below 2%) within 500000 iterations. The derived NN from Experiment 1 may not be entirely appropriate to the real-world data, and hence it could take very long to calibrate the NN towards the goal . However, the results after the 500000 provide interesting insights, and can still be used to assess

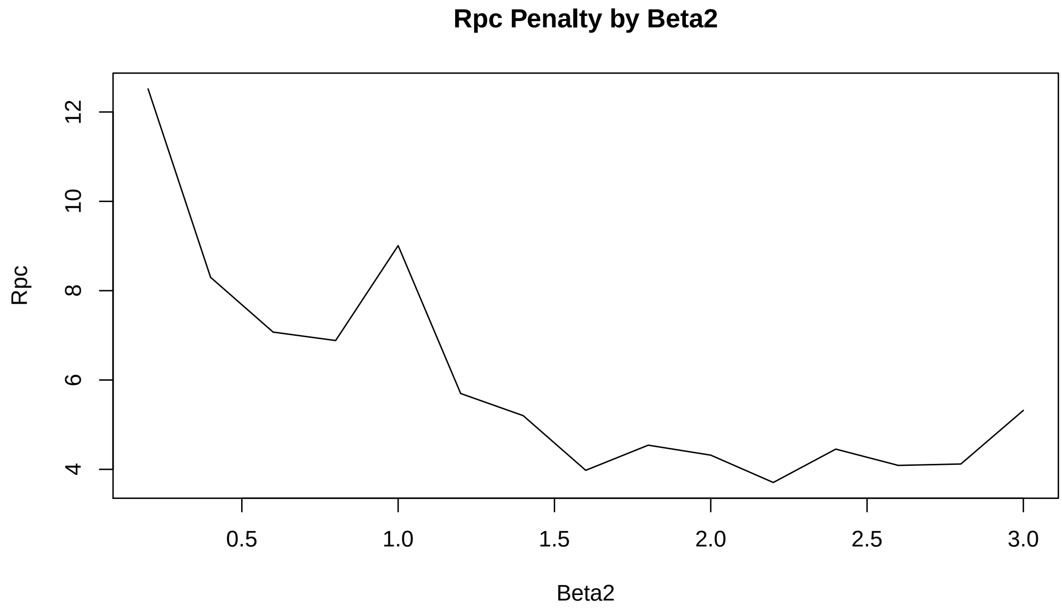

FIGURE 6.13: R_{pc} plot against Beta2 Values.

whether the bounds are enforced.

Call Options

	MARE	25th Perc.	50th Perc.	75th Perc.	100th Perc.	Upper Bound	Relative Bound	Iterations
No Penalty	4.037	1.49125	3.1615	5.52625	26.940	0	4.070	500001
Penalty (Ru)	4.811	1.72300	3.6985	6.36550	38.502	0	3.488	500001
Penalty (Rpc)	4.991	1.66400	3.4065	5.89425	35.227	0	0.291	500001
Penalty (Ru+Rpc)	6.311	3.31800	5.2080	7.93025	28.720	0	0.872	500001

TABLE 6.13: Experiment 3: Call Option Test Data.

Table 6.13 shows how Model 1 (No Penalty) produces the best performing with reference to all the prediction error based metrics. Model 2 and 3 generally have equivalent performance, however, Model 3 presenting generally better distributional characteristics with reference to the quantile performance. Given Model 3 and 4, it seems that the relative bound penalty could have played a positive role in shortening the tail of the ARE distribution for call options.

Put Options

	MARE	25th Perc.	50th Perc.	75th Perc.	100th Perc.	Upper Bound	Relative Bound	Iterations
No Penalty	8.383	5.29750	8.4345	10.72700	45.476	0	4.070	500001
Penalty (Ru)	4.257	1.70050	3.4605	5.64625	19.735	0	3.488	500001
Penalty (Rpc)	7.233	4.22550	6.3735	8.76725	43.946	0	0.291	500001
Penalty (Ru+Rpc)	3.956	1.13475	2.4560	4.52050	50.437	0	0.872	500001

TABLE 6.14: Experiment 3: Put Option Test Data.

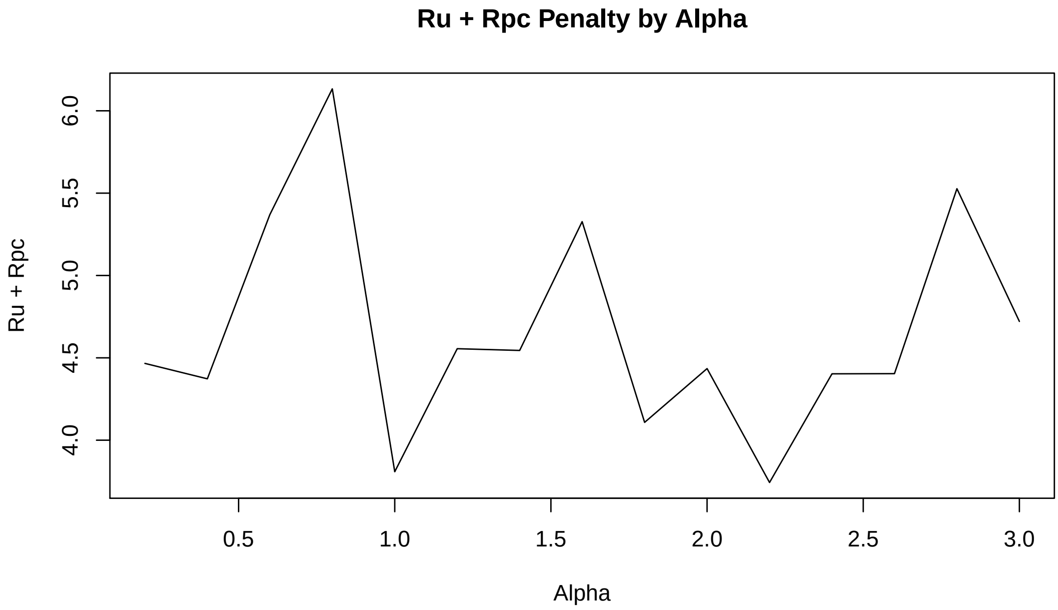

FIGURE 6.14: R_u + R_{pc} plot against Alpha Values.

Table 6.14 portrays an opposing picture to Table 6.13. Model 1 is generally the worst performing model on the ARE quantiles, with the exception of the 100th percentile. Model 4 on the other hand is now the best performer, with the exception of the 100th percentile, and Model 2 is narrowly the second best. It seems that the upper bound penalty played a role in this, as it is the commonality between Model 2 and 4. It is also notable that Model 1 performs substantially worse than the other models.

Overall: Call and Put Options

	MARE	25th Perc.	50th Perc.	75th Perc.	100th Perc.	Upper Bound	Relative Bound	Iterations
No Penalty	6.210	2.41750	5.4810	9.10950	45.476	0	4.070	500001
Penalty (Ru)	4.534	1.71250	3.5645	6.10575	38.502	0	3.488	500001
Penalty (Rpc)	6.112	2.54650	4.8920	7.64525	43.946	0	0.291	500001
Penalty (Ru+Rpc)	5.133	1.90525	3.8395	6.79575	50.437	0	0.872	500001

TABLE 6.15: Experiment 3: Put and Call Option Test Data.

Viewing Table 6.15, the results presented by the four models is somewhat confounding. However, with the exception of the 100th percentile, Models 2-4 appear to be the better overall models given the MARE scores as well as the ARE quantiles.

Considering the bound breach metrics, the upper bound breach provides no more information other than that it was not breached. For the relative bound breach, Model 1 again performs the worst. Furthermore, compared to Model 3 and 4, Model 1 and 2 performs substantially worse in terms of the relative bound breach metric. Given that the relative bound threshold is set relatively wide i.e. 10%, it is expected that the breach of the relative bound will generally be low, i.e. less than 5%, as seen in Table 6.13-6.15. As expected, Model 3 produces the best (lowest) bound breach score (0.291%), since it has the R_{pc} penalty applied during training. Model 4 also produces a relatively low relative bound breach score (0.872%).

Relating to the computational efficiency metric, very limited assessments can be made, other than the fact that the penalised models seem to be calibrating faster and hence more efficiently. Given that the data set is small, and the NN configuration was optimised on simulated data, it is likely sub-optimal for the real-world data set. However, it allowed us to successfully conduct the experiment and focus on the impact the bound penalties had on the distributional aspects of the ARE resulting from each model.

The `R` script for this experiment can be found in Appendix A.6.2.

Chapter 7

Conclusion

In order to conclude this study, the objective with which we set out is revisited, which is; construct a NN model that is:

1. Sufficiently efficient and capable of learning the pricing BSM model for European call and put options;
2. Enforcing economic constraints with reference to relative and absolute bounds that call and put option prices must satisfy.

The framework and findings set out for Experiment 0, 1.1 and 1.2 will hopefully provide future researchers and others with a suitable framework to derive a viable set of NN configurations that predicts European option prices. Some of the findings such as using log prices have in this, and previous studies, shown to be produce better predictions. Experiment 1 does however provide a general idea of how to use a grid search technique to derive a narrower set of candidate configurations within which a final NN specification can be found. However practical this approach was, more effective methods could be used to derive an optimal set of number of hidden layers and neurons per layer, such as a Bayesian hyperparameter optimisation approach.

Considering the results in Experiment 3, it is safe to say that objective 1 has been achieved; we have found a NN sufficiently capable of learning the BSM pricing model for call and put options. Model 1 in particular showed promising capability to predict call and put option prices under a single NN. Moreover, when considering the general MARE results of Experiment 4 (real-world experiment), it is reasonable to infer that NNs can do very well at the task of predicting option prices, especially when trained with more and richer data.

Experiment 2 provides a framework for adding complexity and realism to a NN, tying in with the second objective of this study. Though it is likely that more advanced approaches can be taken to enforce constraints, the penalty-based approach of overcoming problems of non-parametric and non-interpretable models like NNs to make allowance for economic constraints has earned merit. Considering the derivation, structuring and implementation of the upper and relative bound penalties put forward in Chapter 4 (Experiment 2 and Experiment 3), this approach to enforce constraints may provide a useful foundation and reference for further research.

The results of Experiment 3 speaks to whether we have achieved the second objective. Given the nature of NNs, achieving such an objective is not binary as it would be with a closed-form, structured approach. The cost surface is a product of the data that is passed through the NN. The Put-Call parity and relative bound metric produces information for each and every observation, while the upper bound metric, being inherently binary, does not. We are limited in being able to deduce how successful the study was at ensuring upper bound satisfaction, but given the way this experiment was designed, this was achieved, as the lower bound was also satisfied by explicit design using a log price output model specification. Considering how Models 2 - 4 performed, it appears that we were successful at enforcing the upper bound, given the little information we have around this, but we can not be absolutely certain that this approach achieved the enforcement unequivocally.

Like the upper and lower bounds, the Put-Call parity is not specific to BSM, but rather to option prices. The real-world data exhibits that the Put-Call parity has a presence in the markets' option valuation process. Therefore, a NN designed to predict the market valuation mechanism, must itself be able to assess and honour the Put-Call parity. The relative bound aspect produced far more information in the results with which to consider whether this objective has been met as opposed to the upper bound. Comparing the performance of Model 3 and 4 to Model 1 and 2 in Experiment 3, we have found a way to enforce the Put-Call parity to an extent for option pricing using NNs. Given the overall clarity we have on how the penalty-based approach worked on the relative bound, coupled with no upper bound breach, it seems that for the definition of this NN in terms of what "moneyness" options it values, objective 2 has been achieved on the simulated data.

Experiment 3 and 4 together provided noisy, confounding but also part consistent results. We were able to see and conclude that the penalty-based approach in setting out a NN model does enforce the relative bound relatively well, and produces generally better results. However, many of the penalty-based approaches also presented disadvantages relating to the distributional aspects of the absolute relative prediction errors. This was particularly the case for put options and the size of the tail displayed by the 100th percentiles. An approach to measure and penalise wider distributional aspects or using a cost function geared towards the percentile performance in training may circumvent and overcome this. It would certainly be necessary to overcome this additional problem in general.

Many benefits that NNs propose was not explicitly investigated in this study, but abundant they are. Consider for example the variety of additional information that could be used in a NN to value option if they were deemed descriptive of variance by a unsupervised or dimensionality reduction exercise. In the context of being married to structured and closed-form solutions, you would not less capable of using such describers of variance in a parametric model. However, it is always important to think about the disbenefits, before getting carried away by the upside potential.

Moreover, relating specifically to option valuations, though BSM has valid criticism and is severely limited in its raw form, it is immensely useful when studying option valuations. In general, BSM provides a strong basis to measure and therefore assess option valuations in the market. Similarly, like in this study, it can be used to define a model i.e. our model is defined as an option valuation model for in-the-money options, where in-the-money refers to $N(d_2) > 0.25$ for call options. Option pricing with NNs on the other hand is still in its infancy. Though we have demonstrated that some fundamental hurdles in a non-parametric approach like NNs can be overcome to some extent, a more clear and robust solution has to be found before NNs are a valid competitor to option pricing.

The methodology (the application of NNs to option pricing) put forward here is in its infancy and though initial indications are that there could be potential benefits, the fundamental hurdles that need to be overcome in order to be a valid competitor to existing technologies are truly non-trivial. Indeed, setting up the experiments and analysis reveals both technical and philosophical issues such as the use of implied volatility as an input needs to be considered at more length, and contextualised temporally (possibly by a state-dependant model). The lack of risk-neutral martingale theory in the context of the machine learning is perhaps the greatest technical hurdle that needs to be overcome: pricing is not merely a learning problem, and the fundamentals of the pricing theory need to be ironed out before one can claim it as a competing strategy regardless of whether one can deal with the practical aspects of such analysis. Furthermore, the applicability of the UAT, which assumes an approximate target function exists, relies on a continuity. When an underlying process experiences sudden jumps, at discrete intervals, the UATs validity is compromised (Nielsen, 2015). This emphasises a particularly challenging caveat of NNs as such jumps in a process are commonplace in finance.

Further research on this topic is therefore certainly needed. The use of BSM errors as an input could provide many performance benefits, as has been seen in other studies. Setting up different NN models with regards to inputs, combinations thereof, and the impacts this may have on the distributional aspects is another domain that was not investigated in this paper, but is likely to present many insights. Moreover, using stochastic volatility models to produce more realistic volatility and NN contextualisation could be a key focus area for future research. The penalty-based approach to enforce constraints could be further investigated and improved by introducing more control parameters and finding more optimal penalty functions. Beucler et al., 2021 has provided a recent study that may provide a good framework within which to explore constraint enforcement approaches. Consideration of real-world pricing applications and hedging performance is another area that still has to be studied. In future research, it could be very useful to use the market prices of traded strategies like spreads and straddles to inform Put-Call parity and no-arbitrage settings, Barucci, Cherubini, and Landi, 1997 may be a good starting point.

Additionally, it may prove useful to consider separate NN models for put and call options, and enforce constraints in each separately, with derived control parameters. This would however present challenges relating to the relative bound enforcement. Extending the same application to American style options could also be interesting, however, the real-world data aspect may be particularly challenging. This is because American style option are not as frequently traded on exchanges, and so attaining the data may be difficult and costly.

In closing, NNs certainly have a strong case to be a useful and practical class of valuation models. Whether NNs and the bound enforcement undertaken in this study is an optimal way to calibrate a realistic and sound option price prediction model is a separate research question. This study does elucidate that enforcement via penalties provides clear benefits in producing a NN that can predict market prices and make allowance for economic constraints inherent to option prices. However, the model risk presented is certainly more opaque, and could well be much more extreme for certain process filtration's. Thus, such models must be thoroughly studied and put through a sufficient panel of tests before it can be practically motivated as reliable. We hope that the findings in this study are constructive towards bridging the gap from NNs merely being a theoretical construct to being a practically useful and trustworthy valuation model to value options in the complex real-world.

Appendix A

R Code

The R code specific to the NN model extends from code that Pienaar, 2019 provided me.

A.1 Data Generating Code

```
dx =  read.csv("vixcurrent.csv", header = T,
  stringsAsFactors = F)
#install.packages("fitdistrplus")
require(fitdistrplus)
my_data <- dx$VIX.Close

fit_w  <- fitdist(my_data, "weibull")
fit_g  <- fitdist(my_data, "gamma")
fit_ln <- fitdist(my_data, "lnorm")
summary(fit_ln)
hist(dx$VIX.Close)

pdf(file="VIX_Hist.pdf", width = 8.27, height = 4)
hist(dx$VIX.Close, main  = "Histogram of VIX", xlab = "VIX"
  )
dev.off()


par(mfrow = c(2,2))
plot.legend <- c("Weibull", "lognormal", "gamma")
denscomp(list(fit_w, fit_g, fit_ln), legendtext = plot.
  legend)
cdfcomp (list(fit_w, fit_g, fit_ln), legendtext = plot.
  legend)
qqcomp  (list(fit_w, fit_g, fit_ln), legendtext = plot.
  legend)
ppcomp  (list(fit_w, fit_g, fit_ln), legendtext = plot.
  legend)
summary(fit_g)

shape = fit_g$estimate[1]
scale = fit_g$estimate[2]

N = 5000
```

```
#Generate Data
  Sim   = 1:N
  So    = runif(N,50,150)
  Tt    = runif(N,0.25,3)
  r     = runif(N,0.02,0.04)
  q     = runif(N,0,0.03)
  xxx   = runif(N, 0, 1)
  Vol = qgamma(xxx,fit_g$estimate[1], scale = 1/fit_g$
    estimate[2])/100

    th = qnorm(0.25,0,1)
    mn = So* exp(-1*(th*Vol*sqrt(Tt)-(r-q -0.5*Vol^2)*Tt))
    mx = So* exp(th*Vol*sqrt(Tt)+(r-q -0.5*Vol^2)*Tt)
    K  = ceiling(runif(N,mx,mn))

  d1 = 1/(Vol*sqrt(Tt))*(log(So/K)+(r-q +0.5*Vol^2)*Tt)
  d2 = d1-Vol*sqrt(Tt)
  Call.Price = So*exp(-q*Tt)*pnorm(d1)-K*exp(-r*Tt)*pnorm(
    d2)
  Put.Price = K*exp(-r*Tt)*pnorm(-d2) - So*exp(-q*Tt)*pnorm
    (-d1)
  Data = as.data.frame(cbind(Sim,So,Tt,r,q,Vol,K, Put.Price
    , Call.Price))

#Save Data
save(Data, file = "Put_Call.Rdata")
```

A.2 Experiment 1.1 Code

```
rm(list = ls())
# Define activation functions and their first derivitives.
# Define the sum-of-squares objective g and its first
  derivitive w.r.t. the final layer of activations.
sig = function(x)
{
  x
}
sig. = function(x)
{
  1
}
sig2 = function(x)
{
  (2/(1+exp(-2*x)))-1
}
sig2. = function(x)
{
  ((4*exp(-2*x))/(1+exp(-2*x))^2)
}
g     = function(yhat,y)
{
```

```
  (yhat - y)^2
}
g.    = function(yhat,y)
{
  2*(yhat - y)
}

# N = observations
# p = input features
# q = outputs
# m = neurons in layer

neural_net = function(X,Y, theta,m,lam)
{

  N   = dim(X)[1]
  p   = dim(X)[2]
  q   = dim(Y)[2]
  m = append(m,q)
  Neuron = m
  L   = length(m)
  #l = list()
  W   = list()
  B   = list()
  A   = list()
  d   = list()
  dW = list()
  dB = list()


  a0    = X
  ones =  matrix(1,N,1)

  for (i in 1:L)
  {
    if (i ==1) {
      index   = 1:(p*Neuron[i])
      W[[i]] = matrix(theta[index],p,Neuron[i])
      index   = max(index)+1:(Neuron[i])
      B[[i]] = matrix(theta[index],1,Neuron[i])

      dW[[i]]= W[[i]]*0
      dB[[i]]= B[[i]]*0

      A[[i]] = sig2(a0%*%W[[i]]+ones%*%B[[i]])

      rdg = sum(W[[i]]^2)
    } else if (i ==L){
      index   = max(index)+1:(Neuron[i-1]*Neuron[i])
      W[[i]] = matrix(theta[index],Neuron[i-1],Neuron[i])
      index   = max(index)+1:(Neuron[i])
```

```
    B[[i]] = matrix(theta[index],1,Neuron[i])

    dW[[i]]= W[[i]]*0
    dB[[i]]= B[[i]]*0

    A[[i]] = sig(A[[i-1]]%*%W[[i]]+ones%*%B[[i]])

    rdg = rdg + sum(W[[i]]^2)
  } else {
    index  = max(index)+1:(Neuron[i-1]*Neuron[i])
    W[[i]] = matrix(theta[index],Neuron[i-1],Neuron[i])
    index  = max(index)+1:(Neuron[i])
    B[[i]] = matrix(theta[index],1,Neuron[i])

    dW[[i]]= W[[i]]*0
    dB[[i]]= B[[i]]*0

    A[[i]] = sig2(A[[i-1]]%*%W[[i]]+ones%*%B[[i]])

    rdg = rdg + sum(W[[i]]^2)
  }

}

for (j in 1:(L-1))
{
  if (j ==1){
    d[[L+1-j]]  = g.(A[[L+1-j]],Y)*sig.(A[[L-j]]%*%W[[L
      +1-j]]+ones%*%B[[L+1-j]])
    dB[[L+1-j]] = t(ones) %*%d[[L+1-j]]
    dW[[L+1-j]] = (t(A[[L-j]]) %*% d[[L+1-j]]) + 2*lam*W
      [[L+1-j]]
  } else {
    d[[L+1-j]]  = (d[[L+2-j]] %*% t(W[[L+2-j]]))*sig2.(A
      [[L-j]]%*%W[[L+1-j]]+ones%*%B[[L+1-j]])
    dB[[L+1-j]] = t(ones) %*%d[[L+1-j]]
    dW[[L+1-j]] = (t(A[[L-j]]) %*% d[[L+1-j]]) + 2*lam*W
      [[L+1-j]]
  }
}
d[[1]]  = (d[[2]] %*% t(W[[2]]))*sig2.(a0%*%W[[1]]+ones
  %*%B[[1]])
dB[[1]] = t(ones) %*%d[[1]]
dW[[1]] = (t(a0) %*% d[[1]]) + 2*lam*W[[1]]

out   = A[[L]]
error = (Y - out)^2

E1    = sum(error) #pure error
E2    = E1 + lam*(rdg) #error with ridge
grad = c(dW[[1]], dB[[1]])
```

```
  for (i in 2:L){
    grad = c(grad, dW[[i]], dB[[i]])
  }
  return(list(grad = grad/(2*N),out = out, E1 = E1/(2*N),
    E2 = E2/(2*N)))
}



# theta_start = initialized parameters, W and B
# learn = learning rate
# iterations = epochs
grad_dec = function(theta_start,learn,iterations)
{
  theta = theta_start #initialize
  res   = neural_net(X,Y,theta,m,lambda) #feed forward
  errors= rep(NA,iterations) #error vector
  errors[1] = res$E2 #first error
  theta_mat = matrix(NA,length(theta),iterations) #matrix
    with all para as rows, columns as iteration
  theta_mat[,1] = theta
  for(i in 2:iterations)
  {
    learn = ifelse(i<5000, learn, ifelse(i<10000, learn,
      learn))
    theta      = theta-learn*res$grad #update parameters
    theta_mat[,i] = theta #put in para matrix
    set        = sample(1:N,N*0.3, replace = F) #take a
      batch, SGD,
    res        = neural_net(matrix(X[set,],ncol = p),matrix(
      Y[set,],ncol = q),theta,m,lambda) #feed forward
    errors[i] = res$E1 #get new error

  }
  ret = list(theta_hat = theta,errors = errors, theta_mat =
     theta_mat)
  return(ret)
}


# read in data
load("Put_Call.Rdata")

Data$So = Data$So/100
Data$K = Data$K/100
Data$r = Data$r/(0.04)
Data$q = Data$q/(0.04)
Data$Tt = Data$Tt/3
Data$logCall = log(Data$Call.Price)
Data$logPut = log(Data$Put.Price)
```

```
X = as.matrix(Data[,2:7])
colnames(X) = NULL
Y = as.matrix(Data[,10:11])

N = nrow(X)
p = dim(X)[2]
q = dim(Y)[2]


#build grid
neur1 = c(1:6)*8
neur2 = c(0:6)*8
neur3 = c(0:6)*8

g = expand.grid(neur1, neur2, neur3)
g = g[-which((g[,2]==0 & g[,3]>0)),]
g = g[order(g[,1], g[,2], g[,3]),]
g$Lay = ifelse(g$Var2>0&g$Var3>0, 3, ifelse(g$Var2>0, 2, 1)
  )
g = g[,c("Lay","Var1","Var2","Var3")]
g$error = 0
g$pars = 0
g$zeros = 0
g$sign = 0
dim(g)

lambda = 0 #set to zero ie no L2 regularisation
tmr = Sys.time()
#now run the grid - neurons and layers
for (j in 1:nrow(g)){
  cat(paste0("Conf:",j))
  L = as.double(g[j,1])
  if(L==3){m =  c(as.matrix(g[j,2:4]))}else if (L==2){m = c
    (as.matrix(g[j,2:3]))}else{ m=g[j,2] }
  mm = append(m, q)
  npars = (p+1) * mm[1]
  for (i in 2:(length(mm))){
    npars = npars +((mm[i-1]+1)*mm[i])
  }
  theta_rand = runif(npars,-2,2) #initialize para
  res_opt = grad_dec(theta_rand, 0.05,1000) #train
  res_fit = neural_net(X,Y,res_opt$theta_hat,m, lambda) #
    use trained NN and get error

  ch = c(res_fit$E1, npars,length(which(res_opt$theta_hat
    ==0)), length(which(abs(res_opt$theta_hat)>0.005)))
  g[j,5:8] = ch #update grid
  cat('\014')
}
tmr = as.double(Sys.time() - tmr)
```

```
#order by best error on top
g = g[order(g$error,decreasing = F), ]
rownames(g) = NULL

gx = g #gx contains large grid


#now run the grid of initializing weights
g = g[1:25,] #select best 25
THETA = list() #store best initializing weights
lg = matrix(rep(0, 1*nrow(g)),nrow(g), 1)
#now run the grid
tmr2 = Sys.time()
for (j in 1:nrow(g)){
  #j=1
  cat(paste0("Config:",j))
  L = as.double(g[j,1])
  if(L==3){m =  c(as.matrix(g[j,2:4]))}else if (L==2){m = c
    (as.matrix(g[j,2:3]))}else{ m=g[j,2] }
  mm = append(m, q)
  npars = (p+1) * mm[1]
  for (i in 2:(length(mm))){
    npars = npars +((mm[i-1]+1)*mm[i])
  }

  for (jj in 1:10){

    theta_rand = runif(npars,-2,2) #initialize para
    res_opt = grad_dec(theta_rand, 0.05,1000) #optimize
    res_fit = neural_net(X,Y,res_opt$theta_hat,m, lambda) #
      fit and get train error
    #res_fit_test = neural_net(Test_X,Test_Y,res_opt$theta_
      hat,m, lambda) #fit and get test error

    # store the param scores of lowest test error
    if (jj==1){
      lg[j,] = g[j,5] = res_fit$E1
      ch = c(length(which(res_opt$theta_hat==0)), length(
        which(abs(res_opt$theta_hat)>0.005)))
      g[j,7:8] = ch
      THETA[[j]] = theta_rand
    }else if (res_fit$E1<lg[j,]){
      lg[j,] = g[j,5] = res_fit$E1
      ch = c(length(which(res_opt$theta_hat==0)), length(
        which(abs(res_opt$theta_hat)>0.005)))
      g[j,7:8] = ch
      THETA[[j]] = theta_rand #store best initialization
        weights in THETA object

    }
```

```
  }
  cat('\014')
}
tmr2 = as.double(Sys.time() - tmr2)
#g = g[order(g$error,decreasing = F), ]


# g$order = 1:nrow(g)
# g = g[order(g$error,decreasing = F), ]

tmr = Sys.time()
#now run the grid - neurons and layers, more iterations
for (j in 1:nrow(g)){
  cat(paste0("Conf:",j))
  L = as.double(g[j,1])
  if(L==3){m =  c(as.matrix(g[j,2:4]))}else if (L==2){m = c
    (as.matrix(g[j,2:3]))}else{ m=g[j,2] }
  theta_rand = c(unlist(THETA[j]))#initialize para
  npars = length(theta_rand)
  res_opt = grad_dec(theta_rand, 0.05,30000) #train
  res_fit = neural_net(X,Y,res_opt$theta_hat,m, lambda) #
    use trained NN and get error

  ch = c(res_fit$E1, npars,length(which(res_opt$theta_hat
    ==0)), length(which(abs(res_opt$theta_hat)>0.005)))
  g[j,5:8] = ch #update grid
  cat('\014')
}
tmr = as.double(Sys.time() - tmr)

save(gx, g, THETA,  file = "Exp1.1.Rdata")
save(gx, g, file = "Exp1.1.2.Rdata")
```

A.3 Experiment 1.2 Code

```
rm(list = ls())
# Define activation functions and their first derivitives.
# Define the sum-of-squares objective g and its first
  derivitive w.r.t. the final layer of activations.
sig = function(x)
{
  x
}
sig. = function(x)
{
  1
}
sig2 = function(x)
{
  (2/(1+exp(-2*x)))-1
}
```

```
sig2. = function(x)
{
  ((4*exp(-2*x))/(1+exp(-2*x))^2)
}
g     = function(yhat,y)
{
  (yhat - y)^2
}
g.    = function(yhat,y)
{
  2*(yhat - y)
}


# N = observations
# p = input features
# q = outputs
# m = neurons in layer

neural_net = function(X,Y, theta,m,lam)
{

  N   = dim(X)[1]
  p   = dim(X)[2]
  q   = dim(Y)[2]
  m = append(m,q)
  Neuron = m
  L   = length(m)
  #l = list()
  W   = list()
  B   = list()
  A   = list()
  d   = list()
  dW  = list()
  dB  = list()


  a0    = X
  ones =  matrix(1,N,1)

  for (i in 1:L)
  {
    if (i ==1) {
      index   = 1:(p*Neuron[i])
      W[[i]]  = matrix(theta[index],p,Neuron[i])
      index   = max(index)+1:(Neuron[i])
      B[[i]]  = matrix(theta[index],1,Neuron[i])

      dW[[i]]= W[[i]]*0
      dB[[i]]= B[[i]]*0
```

```
      A[[i]] = sig2(a0%*%W[[i]]+ones%*%B[[i]])

      rdg = sum(W[[i]]^2)
    } else if (i ==L){
      index  = max(index)+1:(Neuron[i-1]*Neuron[i])
      W[[i]] = matrix(theta[index],Neuron[i-1],Neuron[i])
      index  = max(index)+1:(Neuron[i])
      B[[i]] = matrix(theta[index],1,Neuron[i])

      dW[[i]]= W[[i]]*0
      dB[[i]]= B[[i]]*0

      A[[i]] = sig(A[[i-1]]%*%W[[i]]+ones%*%B[[i]])

      rdg = rdg + sum(W[[i]]^2)
    } else {
      index  = max(index)+1:(Neuron[i-1]*Neuron[i])
      W[[i]] = matrix(theta[index],Neuron[i-1],Neuron[i])
      index  = max(index)+1:(Neuron[i])
      B[[i]] = matrix(theta[index],1,Neuron[i])

      dW[[i]]= W[[i]]*0
      dB[[i]]= B[[i]]*0

      A[[i]] = sig2(A[[i-1]]%*%W[[i]]+ones%*%B[[i]])

      rdg = rdg + sum(W[[i]]^2)
    }

  }

  for (j in 1:(L-1))
  {
    if (j ==1){
      d[[L+1-j]]  = g.(A[[L+1-j]],Y)*sig.(A[[L-j]]%*%W[[L
        +1-j]]+ones%*%B[[L+1-j]])
      dB[[L+1-j]] = t(ones) %*%d[[L+1-j]]
      dW[[L+1-j]] = (t(A[[L-j]]) %*% d[[L+1-j]]) + 2*lam*W
        [[L+1-j]]
    } else {
      d[[L+1-j]]  = (d[[L+2-j]] %*% t(W[[L+2-j]]))*sig2.(A
        [[L-j]]%*%W[[L+1-j]]+ones%*%B[[L+1-j]])
      dB[[L+1-j]] = t(ones) %*%d[[L+1-j]]
      dW[[L+1-j]] = (t(A[[L-j]]) %*% d[[L+1-j]]) + 2*lam*W
        [[L+1-j]]
    }
  }
  d[[1]]  = (d[[2]] %*% t(W[[2]]))*sig2.(a0%*%W[[1]]+ones
    %*%B[[1]])
  dB[[1]] = t(ones) %*%d[[1]]
  dW[[1]] = (t(a0) %*% d[[1]]) + 2*lam*W[[1]]
```

```
  out    = A[[L]]
  error = (Y - out)^2

  E1     = sum(error) #pure error
  E2     = E1 + lam*(rdg) #error with ridge
  grad = c(dW[[1]], dB[[1]])
  for (i in 2:L){
    grad = c(grad, dW[[i]], dB[[i]])
  }
  return(list(grad = grad/(2*N),out = out, E1 = E1/(2*N),
    E2 = E2/(2*N)))
}



# theta_start = initialized parameters, W and B
# learn = learning rate
# iterations = epochs
grad_dec = function(theta_start,learn,iterations)
{
  theta = theta_start #initialize
  res    = neural_net(X,Y,theta,m,lambda) #feed forward
  errors= rep(NA,iterations) #error vector
  errors[1] = res$E2 #first error
  #theta_mat = matrix(NA,length(theta),iterations) #matrix
    with all para as rows, columns as iteration
  theta_opt = theta
  mn = res$E1
  for(i in 2:iterations)
  {
    learn = ifelse(i<5000, learn, ifelse(i<10000, learn,
      learn))
    theta      = theta-learn*res$grad #update parameters
    #theta_mat[,i] = theta #put in para matrix
    set        = sample(1:N,N*0.3, replace = F) #take a
      batch, SGD,
    res        = neural_net(matrix(X[set,],ncol = p),matrix(
      Y[set,],ncol = q),theta,m,lambda) #feed forward
    errors[i] = res$E2 #get new error
    if (res$E1 < mn){
      mn = res$E1
      theta_opt = theta}
  }
  ret = list(theta_hat = theta,errors = errors, theta_opt =
    theta_opt)
  return(ret)
}

# read in data
load("Put_Call.Rdata")
```

```
#normalize
Data$So = Data$So/100
Data$K = Data$K/100
Data$r = Data$r/(0.04)
Data$q = Data$q/(0.04)
Data$Tt = Data$Tt/3
Data$logCall = log(Data$Call.Price)
Data$logPut = log(Data$Put.Price)

Dat = Data

## Training Data
obs = nrow(Dat)
train = 0.8
N = obs * train
Data = Data[1:N,]
X = as.matrix(Data[,2:7])
colnames(X) = NULL
Y = as.matrix(Data[,10:11])

##Testing Data
Test_Data = Dat[(N +1):obs,]
Test_X = as.matrix(Test_Data[,2:7])
colnames(Test_X) = NULL
Test_Y = as.matrix(Test_Data[,10:11])

p = dim(X)[2]
q = dim(Y)[2]

#load Exp1.1 Data
load('Exp1.1.Rdata')
g$order = 1:nrow(g)
g$lambdamin = 0
g = g[order(g$error,decreasing = F), ]
g = g[c(1:2,4,6:8),]

#Generate Lambda Grids
gg = exp(seq(-9,-1,length = 15))
lg = matrix(rep(0, length(gg)*nrow(g)),nrow(g), length(gg))
lgg = matrix(rep(0, length(gg)*nrow(g)),nrow(g), length(gg)
  )
lgcall = matrix(rep(0, length(gg)*nrow(g)),nrow(g), length(
  gg))
lgput = matrix(rep(0, length(gg)*nrow(g)),nrow(g), length(
  gg))
res = list()
#now run the grid
for (j in 1:nrow(g)){
  cat(paste0("Config:",j))
  L = as.double(g[j,1])
```

```
  if(L==3){m =  c(as.matrix(g[j,2:4]))}else if (L==2){m = c
    (as.matrix(g[j,2:3]))}else{ m=g[j,2] }

  theta_rand = c(unlist(THETA[g$order[j]]))#initialize para
  npars = length(theta_rand)

  for (jj in 1:length(gg)){

    lambda = lam = gg[jj]
    res_opt = grad_dec(theta_rand, 0.05,30000) #optimize
    res_fit = neural_net(X,Y,res_opt$theta_opt,m, lambda) #
      fit and get train error
    res_fit_test = neural_net(Test_X,Test_Y,res_opt$theta_
      opt,m, lambda) #fit and get test error
    lg[j,jj] = res_fit_test$E1
    lgg[j,jj] = mean(abs(exp(res_fit_test$out) - exp(Test_Y
      ))/exp(Test_Y))
    lgcall[j,jj] = mean(abs(exp(res_fit_test$out[,1]) - exp
      (Test_Y[,1]))/exp(Test_Y[,1]))
    lgput[j,jj] = mean(abs(exp(res_fit_test$out[,2]) - exp(
      Test_Y[,2]))/exp(Test_Y[,2]))
    # store the param scores of lowest test error
    if (jj==1){
      ch = c(res_fit_test$E1, npars,length(which(res_opt$
        theta_hat==0)), length(which(abs(res_opt$theta_hat)
        >0.005)))
      g[j,5:8] = ch #update grid
      lgmin = res_fit_test$E1
      g[j,10] = gg[jj]
      res[[j]] = res_opt
    }else if (lg[j,jj]<lgmin){

        ch = c(res_fit_test$E1, npars,length(which(res_opt$
          theta_hat==0)), length(which(abs(res_opt$theta_
          hat)>0.005)))
        g[j,5:8] = ch #update grid
        lgmin = res_fit_test$E1
        g[j,10] = gg[jj]
        res[[j]] = res_opt
    }

  }
  cat('\014')
}

save(lg, g, gg, THETA, gx, lgg, lgcall, lgput, res, file =
  "Exp1.2.Rdata")
```

A.4 Experiment 2 Code

```
rm(list = ls())
```

```
# Define activation functions and their first derivitives.
# Define the sum-of-squares objective g and its first
  derivitive w.r.t. the final layer of activations.
sig = function(x)
{
  x
}
sig. = function(x)
{
  1
}
sig2 = function(x)
{
  (2/(1+exp(-2*x)))-1
}
sig2. = function(x)
{
  ((4*exp(-2*x))/(1+exp(-2*x))^2)
}
g     = function(yhat,y)
{
  (yhat - y)^2
}
g.    = function(yhat,y)
{
  2*(yhat - y)
}

Ru = function(x)
{
  1000/(1+exp(-10*(x-2)))
}


Ru. = function(x)
{
  10000/(1+exp(-10*(x-2)))*(1-1/(1+exp(-10*(x-2))))
}



h = function(x,s){

  exp(x)/s
}
h. = function(x,s){

  exp(x)/s
}

Rpc = function(x){
  (10*x)^2
```

```
}

Rpc. = function(x){
  20*x
}


# N = observations
# p = input features
# q = outputs
# m = neurons in layer

neural_net = function(X,Y, SK, theta,m,lam, pen, beta1,
  beta2, alpha)
{


  N  = dim(X)[1]
  p  = dim(X)[2]
  q  = dim(Y)[2]
  m = append(m,q)
  Neuron = m
  L  = length(m)

  Ix = cbind(matrix(rep(-1,nrow(Y)*1), nrow(Y), 1), matrix(
    rep(1,nrow(Y)*1), nrow(Y), 1))
  K = cbind(SK[,2],SK[,2])
  QR = cbind(exp(-(3*X[,2])*(0.04*X[,4])),-exp(-(3*X[,2])
    *(0.04*X[,3])))

  #l = list()
  W  = list()
  B  = list()
  A  = list()
  d  = list()
  dW = list()
  dB = list()


  a0   = X
  ones =  matrix(1,N,1)

  for (i in 1:L)
  {
    if (i ==1) {
      index  = 1:(p*Neuron[i])
      W[[i]] = matrix(theta[index],p,Neuron[i])
      index  = max(index)+1:(Neuron[i])
      B[[i]] = matrix(theta[index],1,Neuron[i])

      dW[[i]]= W[[i]]*0
```

```
      dB[[i]]= B[[i]]*0

      A[[i]] = sig2(a0%*%W[[i]]+ones%*%B[[i]])

      rdg = sum(W[[i]]^2)
    } else if (i ==L){
      index  = max(index)+1:(Neuron[i-1]*Neuron[i])
      W[[i]] = matrix(theta[index],Neuron[i-1],Neuron[i])
      index  = max(index)+1:(Neuron[i])
      B[[i]] = matrix(theta[index],1,Neuron[i])

      dW[[i]]= W[[i]]*0
      dB[[i]]= B[[i]]*0

      A[[i]] = sig(A[[i-1]]%*%W[[i]]+ones%*%B[[i]])

      rdg = rdg + sum(W[[i]]^2)
    } else {
      index  = max(index)+1:(Neuron[i-1]*Neuron[i])
      W[[i]] = matrix(theta[index],Neuron[i-1],Neuron[i])
      index  = max(index)+1:(Neuron[i])
      B[[i]] = matrix(theta[index],1,Neuron[i])

      dW[[i]]= W[[i]]*0
      dB[[i]]= B[[i]]*0

      A[[i]] = sig2(A[[i-1]]%*%W[[i]]+ones%*%B[[i]])

      rdg = rdg + sum(W[[i]]^2)
    }

  }

  pc = ((exp(A[[L]]) * Ix) + (SK * QR))/K
  PC = pc[,1]+pc[,2]
  PC = cbind(PC, PC) #same penalty applies to put and call

  for (j in 1:(L-1))
  {
    if (j ==1){
      if (pen ==1){ #upper bound only
        d[[L+1-j]]  = (g.(A[[L+1-j]],Y) + beta1*Ru.(h(A[[L
          +1-j]],SK))*h.(A[[L+1-j]],SK))
      } else if (pen ==2){ #rel bound only
        d[[L+1-j]]  = (g.(A[[L+1-j]],Y) + beta2*Rpc.(PC)*h
          .(A[[L+1-j]],K)*Ix)
      } else if (pen ==3){ #both penalties
        d[[L+1-j]]  = (g.(A[[L+1-j]],Y) + alpha*Ru.(h(A[[L
          +1-j]],SK))*h.(A[[L+1-j]],SK)
                      + alpha*Rpc.(PC)*h.(A[[L+1-j]],K)*Ix
                        )
```

```
      }
      else {d[[L+1-j]]  = (g.(A[[L+1-j]],Y))}

      d[[L+1-j]]  = d[[L+1-j]]*sig.(A[[L-j]]%*%W[[L+1-j]]+
        ones%*%B[[L+1-j]])
      dB[[L+1-j]] = t(ones) %*%d[[L+1-j]]
      dW[[L+1-j]] = (t(A[[L-j]]) %*% d[[L+1-j]]) + 2*lam*W
        [[L+1-j]]
    } else {
      d[[L+1-j]]  = (d[[L+2-j]] %*% t(W[[L+2-j]]))*sig2.(A
        [[L-j]]%*%W[[L+1-j]]+ones%*%B[[L+1-j]])
      dB[[L+1-j]] = t(ones) %*%d[[L+1-j]]
      dW[[L+1-j]] = (t(A[[L-j]]) %*% d[[L+1-j]]) + 2*lam*W
        [[L+1-j]]
    }
  }
  d[[1]]  = (d[[2]] %*% t(W[[2]]))*sig2.(a0%*%W[[1]]+ones
    %*%B[[1]])
  dB[[1]] = t(ones) %*%d[[1]]
  dW[[1]] = (t(a0) %*% d[[1]]) + 2*lam*W[[1]]

  out   = A[[L]]
  error = (Y - out)^2
  pc = ((exp(A[[L]]) * Ix) + (SK * QR))/K
  PC = pc[,1]+pc[,2]
  PC = cbind(PC, PC) #same penalty applies to put and call

  E1    = sum(error) #pure error
  E2    = E1 + lam*(rdg) #error with ridge
  if (pen ==1){E3    = E2 + beta1*sum(Ru(h(out,SK)))}
  else if (pen ==2){E3    = E2 + beta2*sum(Rpc(PC))}
  else if (pen ==3){E3    = E2 + alpha*sum(Ru(h(out,SK))) +
     alpha*sum(Rpc(PC))}
  else {E3    = E2}
  #E3    = E1 + lam*(rdg) + sum(Ru(h(out,SK)))
  UB   = length(which(h(out,SK)>1))
  RB   = length(which(PC>0.1))
  grad = c(dW[[1]], dB[[1]])
  for (i in 2:L){
    grad = c(grad, dW[[i]], dB[[i]])
  }
  return(list(grad = grad/(2*N),out = out, E1 = E1/(2*N),
    E2 = E2/(2*N), E3 = E3/(2*N),
              UB= UB*100/(2*N), RB= RB*100/(2*N)))
}



# theta_start = initialized parameters, W and B
# learn = learning rate
# iterations = epochs
```

```
grad_dec = function(theta_start,learn,iterations,goal, pen,
   beta1, beta2, alpha)
{
  theta = theta_start #initialize
  res   = neural_net(X,Y,SK,theta,m,lambda,pen, beta1,
    beta2, alpha) #feed forward
  errors  = rep(NA,iterations) #error vector
  errors2 = rep(NA,iterations) #error vector
  ubound  = rep(NA,iterations) #error vector
  rbound  = rep(NA,iterations)
  errors[1] = res$E2 #first error
  errors2[1] = res$E3
  ubound[1] = res$UB
  rbound[1] = res$RB

  theta_mat = matrix(NA,length(theta),iterations) #matrix
    with all para as rows, columns as iteration
  theta_mat[,1] = theta
  x = 100
  i=2
  #for(i in 2:iterations)
  while(x > goal)
  {
    learn = ifelse(i<5000, learn, ifelse(i<100000, learn,
      learn/2))
    theta      = theta-learn*res$grad #update parameters
    theta_mat[,i] = theta #put in para matrix
    set        = sample(1:N,N*0.3, replace = F) #take a
      batch, SGD,
    res        = neural_net(matrix(X[set,],ncol = p),matrix(
      Y[set,],ncol = q),
                           matrix(SK[set,],ncol = q), theta
                             ,m,lambda,pen, beta1, beta2,
                             alpha) #feed forward
    errors[i] = res$E2 #get new error
    errors2[i] = res$E3
    ubound[i] = res$UB
    rbound[i] = res$RB
    trainer   = round(100*mean(abs(exp(res$out[,1:2]) - exp
      (matrix(Y[set,],ncol = q)[,1:2]))/exp(matrix(Y[set,],
      ncol = q)[,1:2])),3)
    x = trainer
    itx = iterations
    if(i == itx){x = 0}
    i = i+1
  }
  ret = list(theta_hat = theta,errors = errors, errors2 =
    errors2,
             ubound = ubound,rbound = rbound,theta_mat =
               theta_mat, runs = i)
  return(ret)
```

```
}

# read in data
load("Put_Call.Rdata")
# DAT = list()
#normalize
Data$So = Data$So/100
Data$K = Data$K/100
Data$r = Data$r/(0.04)
Data$q = Data$q/(0.04)
Data$Tt = Data$Tt/3
Data$logCall = log(Data$Call.Price)
Data$logPut = log(Data$Put.Price)

Dat = Data

## Training
obs = nrow(Dat)
train = 0.8
N = obs * train
Data = Data[1:N,]
X = as.matrix(Data[,2:7])
colnames(X) = NULL
Y = as.matrix(Data[,10:11])
SK= as.matrix(Data[,c(2,7)])*100

##Testing
Test_Data = Dat[(N +1):obs,]
# DAT[1] = Test_Data
Test_X = as.matrix(Test_Data[,2:7])
colnames(Test_X) = NULL
Test_Y = as.matrix(Test_Data[,10:11])
Test_SK= as.matrix(Test_Data[,c(2,7)])*100

## Stress Testing
load("Put_Call_Stress.Rdata")

#normalize
Datax$So = Datax$So/100
Datax$K = Datax$K/100
Datax$r = Datax$r/(0.04)
Datax$q = Datax$q/(0.04)
Datax$Tt = Datax$Tt/3
Datax$logCall = log(Datax$Call.Price)
Datax$logPut = log(Datax$Put.Price)

# DAT[2] = Datax
STest_X = as.matrix(Datax[,2:7])
colnames(STest_X) = NULL
STest_Y = as.matrix(Datax[,10:11])
STest_SK= as.matrix(Datax[,c(2,7)])*100
```

```
p = dim(X)[2]
q = dim(Y)[2]
#load data from Exp1.2 and select lambda, initialization,
  Layers, Neurons

load("Exp1.2.Rdata")

j = 1
L = as.double(g[j,1])
if(L==3){m =  c(as.matrix(g[j,2:4]))}else if (L==2){m = c(
  as.matrix(g[j,2:3]))}else{ m=g[j,2] }

theta_rand = c(unlist(THETA[g$order[j]]))#initialize para
npars = length(theta_rand)

c1 = which.min(apply(matrix(lg[j,],1,length(gg)),MARGIN=2,
  min))
lambda = gg[c1]
#first train for 10 iterations - make sure Ru doesn't cause
   numerical issues

lr = 0.05
iter = 3
goal = 3
pen = 0
b1 = 0.2
b2 = 0.4
a1 = 0.5
res_opt = grad_dec(theta_rand, lr,iter,goal, pen , b1 , b2,
   a1) #optimize
theta_rand = res_opt$theta_hat #use new weights
res_fit = neural_net(X,Y,SK,res_opt$theta_hat,m, lambda,
  pen , b1 , b2, a1)




#beta1 grid
Bone = c(10:1)/5
Gb1  = matrix(rep(0, 1*length(Bone)),1, length(Bone))
gr1 = matrix(rep(0, 2*length(Bone)),2, length(Bone))

for (i in 1:length(Gb1)){
  lr = 0.05
  iter = 80000
  goal = 3
  pen = 1
  b1 = Bone[i]
  b2 = 0.4
```

```
  a1 = 0.5
  res_opt = grad_dec(theta_rand, lr,iter,goal, pen , b1 ,
    b2, a1) #optimize

  res_fit = neural_net(X,Y,SK,res_opt$theta_hat,m, lambda,
    pen , b1 , b2, a1)
  res_fit_test = neural_net(Test_X,Test_Y,Test_SK,res_opt$
    theta_hat,m, lambda, pen , b1 , b2, a1)

  gr1[1,i] = round(100*(res_fit_test$E3 - res_fit_test$E2)/
    b1,5)
  gr1[2,i] = round(100*mean(abs(exp(res_fit_test$out[,1:2])
      - exp(Test_Y[,1:2]))/exp(Test_Y[,1:2])),3)
}
#beta2 grid
Btwo = c(10:1)/5
Gb2  = matrix(rep(0, 1*length(Btwo)),1, length(Btwo))
gr2  = matrix(rep(0, 2*length(Btwo)),2, length(Btwo))

for (i in 1:length(Gb2)){
  lr = 0.05
  iter = 80000
  goal = 3
  pen = 2
  b1 = 0.2
  b2 = Btwo[i]
  a1 = 0.5
  res_opt = grad_dec(theta_rand, lr,iter,goal, pen , b1 ,
    b2, a1) #optimize

  res_fit = neural_net(X,Y,SK,res_opt$theta_hat,m, lambda,
    pen , b1 , b2, a1)
  res_fit_test = neural_net(Test_X,Test_Y,Test_SK,res_opt$
    theta_hat,m, lambda, pen , b1 , b2, a1)

  gr2[1,i] = round(100*(res_fit_test$E3 - res_fit_test$E2)/
    b2,5)
  gr2[2,i] = round(100*mean(abs(exp(res_fit_test$out[,1:2])
      - exp(Test_Y[,1:2]))/exp(Test_Y[,1:2])),3)
}
#alpha grid
Alph  = c(10:1)/5
Galph = matrix(rep(0, 1*length(Alph)),1, length(Alph))
gr3 = matrix(rep(0, 2*length(Alph)),2, length(Alph))
for (i in 1:length(Alph)){
  iter = 80000
  goal = 3
  pen = 3
  b1 = 0.2
  b2 = 0.4
  a1 = Alph[1]
```

```
  res_opt = grad_dec(theta_rand, lr,iter,goal, pen , b1 ,
    b2, a1) #optimize

  res_fit = neural_net(X,Y,SK,res_opt$theta_hat,m, lambda,
    pen , b1 , b2, a1)
  res_fit_test = neural_net(Test_X,Test_Y,Test_SK,res_opt$
    theta_hat,m, lambda, pen , b1 , b2, a1)

  gr3[1,i] = round(100*(res_fit_test$E3 - res_fit_test$E2)/
    a1,5)
  gr3[2,i] = round(100*mean(abs(exp(res_fit_test$out[,1:2])
      - exp(Test_Y[,1:2]))/exp(Test_Y[,1:2])),3)
}

save(gr1,gr2,gr3,Alph, file = "Exp2_beta_alpha.Rdata")
```

A.5 Experiment 3 Code

```
rm(list = ls())
# Define activation functions and their first derivatives.
# Define the sum-of-squares objective g and its first
  derivative w.r.t. the final layer of activations.
sig = function(x)
{
  x
}
sig. = function(x)
{
  1
}
sig2 = function(x)
{
  (2/(1+exp(-2*x)))-1
}
sig2. = function(x)
{
  ((4*exp(-2*x))/(1+exp(-2*x))^2)
}
g     = function(yhat,y)
{
  (yhat - y)^2
}
g.    = function(yhat,y)
{
  2*(yhat - y)
}

Ru = function(x)
{
  1000/(1+exp(-10*(x-2)))
}
```

```
Ru. = function(x)
{
  10000/(1+exp(-10*(x-2)))*(1-1/(1+exp(-10*(x-2))))
}


h = function(x,s){

  exp(x)/s
}
h. = function(x,s){

  exp(x)/s
}

Rpc = function(x){
  (10*x)^2
}

Rpc. = function(x){
  20*x
}


# N = observations
# p = input features
# q = outputs
# m = neurons in layer

neural_net = function(X,Y, SK, theta,m,lam, pen, beta1,
  beta2, alpha)
{


  N  = dim(X)[1]
  p  = dim(X)[2]
  q  = dim(Y)[2]
  m = append(m,q)
  Neuron = m
  L  = length(m)

  Ix = cbind(matrix(rep(-1,nrow(Y)*1), nrow(Y), 1), matrix(
    rep(1,nrow(Y)*1), nrow(Y), 1))
  K = cbind(SK[,2],SK[,2])
  QR = cbind(exp(-(3*X[,2])*(0.04*X[,4])),-exp(-(3*X[,2])
    *(0.04*X[,3])))

  #l = list()
  W  = list()
```

```
B  = list()
A  = list()
d  = list()
dW = list()
dB = list()


a0   = X
ones =  matrix(1,N,1)

for (i in 1:L)
{
  if (i ==1) {
    index  = 1:(p*Neuron[i])
    W[[i]] = matrix(theta[index],p,Neuron[i])
    index  = max(index)+1:(Neuron[i])
    B[[i]] = matrix(theta[index],1,Neuron[i])

    dW[[i]]= W[[i]]*0
    dB[[i]]= B[[i]]*0

    A[[i]] = sig2(a0%*%W[[i]]+ones%*%B[[i]])

    rdg = sum(W[[i]]^2)
  } else if (i ==L){
    index  = max(index)+1:(Neuron[i-1]*Neuron[i])
    W[[i]] = matrix(theta[index],Neuron[i-1],Neuron[i])
    index  = max(index)+1:(Neuron[i])
    B[[i]] = matrix(theta[index],1,Neuron[i])

    dW[[i]]= W[[i]]*0
    dB[[i]]= B[[i]]*0

    A[[i]] = sig(A[[i-1]]%*%W[[i]]+ones%*%B[[i]])

    rdg = rdg + sum(W[[i]]^2)
  } else {
    index  = max(index)+1:(Neuron[i-1]*Neuron[i])
    W[[i]] = matrix(theta[index],Neuron[i-1],Neuron[i])
    index  = max(index)+1:(Neuron[i])
    B[[i]] = matrix(theta[index],1,Neuron[i])

    dW[[i]]= W[[i]]*0
    dB[[i]]= B[[i]]*0

    A[[i]] = sig2(A[[i-1]]%*%W[[i]]+ones%*%B[[i]])

    rdg = rdg + sum(W[[i]]^2)
  }

}
```

```
pc = ((exp(A[[L]]) * Ix) + (SK * QR))/K
PC = pc[,1]+pc[,2]
PC = cbind(PC, PC) #same penalty applies to put and call

for (j in 1:(L-1))
{
  if (j ==1){
    if (pen ==1){ #upper bound only
      d[[L+1-j]]  = (g.(A[[L+1-j]],Y) + beta1*Ru.(h(A[[L
        +1-j]],SK))*h.(A[[L+1-j]],SK))
    } else if (pen ==2){ #rel bound only
      d[[L+1-j]]  = (g.(A[[L+1-j]],Y) + beta2*Rpc.(PC)*h
        .(A[[L+1-j]],K)*Ix)
    } else if (pen ==3){ #both penalties
      d[[L+1-j]]  = (g.(A[[L+1-j]],Y) + alpha*Ru.(h(A[[L
        +1-j]],SK))*h.(A[[L+1-j]],SK)
                    + alpha*Rpc.(PC)*h.(A[[L+1-j]],K)*Ix
                      )
    }
    else {d[[L+1-j]]  = (g.(A[[L+1-j]],Y))}

    d[[L+1-j]]  = d[[L+1-j]]*sig.(A[[L-j]]%*%W[[L+1-j]]+
      ones%*%B[[L+1-j]])
    dB[[L+1-j]] = t(ones) %*%d[[L+1-j]]
    dW[[L+1-j]] = (t(A[[L-j]]) %*% d[[L+1-j]]) + 2*lam*W
      [[L+1-j]]
  } else {
    d[[L+1-j]]  = (d[[L+2-j]] %*% t(W[[L+2-j]]))*sig2.(A
      [[L-j]]%*%W[[L+1-j]]+ones%*%B[[L+1-j]])
    dB[[L+1-j]] = t(ones) %*%d[[L+1-j]]
    dW[[L+1-j]] = (t(A[[L-j]]) %*% d[[L+1-j]]) + 2*lam*W
      [[L+1-j]]
  }
}
d[[1]]  = (d[[2]] %*% t(W[[2]]))*sig2.(a0%*%W[[1]]+ones
  %*%B[[1]])
dB[[1]] = t(ones) %*%d[[1]]
dW[[1]] = (t(a0) %*% d[[1]]) + 2*lam*W[[1]]

out   = A[[L]]
error = (Y - out)^2
pc = ((exp(A[[L]]) * Ix) + (SK * QR))/K
PC = pc[,1]+pc[,2]
PC = cbind(PC, PC) #same penalty applies to put and call

E1    = sum(error) #pure error
E2    = E1 + lam*(rdg) #error with ridge
if (pen ==1){E3    = E2 + beta1*sum(Ru(h(out,SK)))}
else if (pen ==2){E3    = E2 + beta2*sum(Rpc(PC))}
```

```
  else if (pen ==3){E3     = E2 + alpha*sum(Ru(h(out,SK))) +
     alpha*sum(Rpc(PC))}
  else {E3     = E2}
  #E3     = E1 + lam*(rdg) + sum(Ru(h(out,SK)))
  UB    = length(which(h(out,SK)>1))
  RB    = length(which(PC>0.01))
  grad = c(dW[[1]], dB[[1]])
  for (i in 2:L){
    grad = c(grad, dW[[i]], dB[[i]])
  }
  return(list(grad = grad/(2*N),out = out, E1 = E1/(2*N),
    E2 = E2/(2*N), E3 = E3/(2*N),
                UB= UB*100/(2*N), RB= RB*100/(2*N)))
}



# theta_start = initialized parameters, W and B
# learn = learning rate
# iterations = epochs
grad_dec = function(theta_start,learn,iterations,goal, pen,
   beta1, beta2, alpha)
{
  theta = theta_start #initialize
  res    = neural_net(X,Y,SK,theta,m,lambda,pen, beta1,
    beta2, alpha) #feed forward
  errors   = rep(NA,iterations) #error vector
  errors2  = rep(NA,iterations) #error vector
  ubound   = rep(NA,iterations) #error vector
  rbound   = rep(NA,iterations)
  errors[1] = res$E2 #first error
  errors2[1] = res$E3
  ubound[1] = res$UB
  rbound[1] = res$RB

  theta_mat = matrix(NA,length(theta),iterations) #matrix
    with all para as rows, columns as iteration
  theta_mat[,1] = theta
  x = 100
  i=2
  #for(i in 2:iterations)
  while(x > goal)
  {
    learn = ifelse(i<5000, learn, ifelse(i<200000, learn,
      learn/1))
    theta      = theta-learn*res$grad #update parameters
    theta_mat[,i] = theta #put in para matrix
    set        = sample(1:N,N*0.3, replace = F) #take a
      batch, SGD,
    res        = neural_net(matrix(X[set,],ncol = p),matrix(
      Y[set,],ncol = q),
```

```
                                    matrix(SK[set,],ncol = q), theta
                                      ,m,lambda,pen, beta1, beta2,
                                      alpha) #feed forward
     errors[i] = res$E2 #get new error
     errors2[i] = res$E3
     ubound[i] = res$UB
     rbound[i] = res$RB
     trainer   = round(100*mean(abs(exp(res$out[,1:2]) - exp
       (matrix(Y[set,],ncol = q)[,1:2]))/exp(matrix(Y[set,],
       ncol = q)[,1:2])),3)
     x = trainer
     itx = iterations
     if(i == itx){x = 0}
     i = i+1
  }
  ret = list(theta_hat = theta,errors = errors, errors2 =
    errors2,
             ubound = ubound,rbound = rbound, runs = i)
  return(ret)
}

# read in data
load("Put_Call.Rdata")

#normalize
Data$So = Data$So/100
Data$K = Data$K/100
Data$r = Data$r/(0.04)
Data$q = Data$q/(0.04)
Data$Tt = Data$Tt/3
Data$logCall = log(Data$Call.Price)
Data$logPut = log(Data$Put.Price)

Dat = Data

## Training
obs = nrow(Dat)
train = 0.8
N = obs * train
Data = Data[1:N,]
X = as.matrix(Data[,2:7])
colnames(X) = NULL
Y = as.matrix(Data[,10:11])
SK= as.matrix(Data[,c(2,7)])*100

##Testing
Test_Data = Dat[(N +1):obs,]
Dat1 = Test_Data
Test_X = as.matrix(Test_Data[,2:7])
colnames(Test_X) = NULL
Test_Y = as.matrix(Test_Data[,10:11])
```

```
Test_SK= as.matrix(Test_Data[,c(2,7)])*100

## Stress Testing
load("Put_Call_Stress.Rdata")

#normalize
Datax$So = Datax$So/100
Datax$K = Datax$K/100
Datax$r = Datax$r/(0.04)
Datax$q = Datax$q/(0.04)
Datax$Tt = Datax$Tt/3
Datax$logCall = log(Datax$Call.Price)
Datax$logPut = log(Datax$Put.Price)

Dat2 = Datax
STest_X = as.matrix(Datax[,2:7])
colnames(STest_X) = NULL
STest_Y = as.matrix(Datax[,10:11])
STest_SK= as.matrix(Datax[,c(2,7)])*100


p = dim(X)[2]
q = dim(Y)[2]
#load data from Exp1.2 and select lambda, initialization,
  Layers, Neurons

load("Exp1.2.Rdata")

j = 1
L = as.double(g[j,1])
if(L==3){m =  c(as.matrix(g[j,2:4]))}else if (L==2){m = c(
  as.matrix(g[j,2:3]))}else{ m=g[j,2] }

theta_rand = c(unlist(THETA[g$order[j]]))#initialize para
npars = length(theta_rand)

c1 = which.min(apply(matrix(lg[j,],1,length(gg)),MARGIN=2,
  min))
lambda = gg[c1]
#first train for 10 iterations - make sure Ru doesn't cause
   numerical issues

lr = 0.07
iter = 5
goal = 2.5
pen = 0
b1 = 0.4
b2 = 2
a1 = 2
res_opt = grad_dec(theta_rand, lr,iter,goal, pen , b1 , b2,
   a1) #optimize
```

```
theta_rand = res_opt$theta_hat #use new weights
res_fit = neural_net(X,Y,SK,res_opt$theta_hat,m, lambda,
  pen , b1 , b2, a1)

res_fit$E3 - res_fit$E2
max(h(res_fit$out, SK))
max(Ru(h(res_fit$out, SK)))

#train NN with no bound penalty

res = list()
res_f = list()
res_f_test = list()
res_f_Stest = list()




lr = 0.07
iter = 400000
goal = 2
b1 = 0.4
b2 = 2
a1 = 2


for (z in 1:4){
  pen = z-1
  res[[z]] = grad_dec(theta_rand, lr,iter, goal, pen , b1 ,
     b2, a1) #optimize
  res_f[[z]] = neural_net(X,Y,SK,res_opt$theta_hat,m,
    lambda,pen , b1 , b2, a1) #fit and get train error
  res_f_test[[z]] = neural_net(Test_X,Test_Y,Test_SK,res_
    opt$theta_hat,m, lambda,pen , b1 , b2, a1) #CVE on
    normal data
  res_f_Stest[[z]] = neural_net(STest_X,STest_Y,STest_SK,
    res_opt$theta_hat,m, lambda,pen , b1 , b2, a1) #CVE on
    stressed dat

}


res_f_test[[1]]$E3-res_f_test[[1]]$E2
res_f_test[[2]]$E3-res_f_test[[2]]$E2
res_f_test[[3]]$E3-res_f_test[[3]]$E2
res_f_test[[4]]$E3-res_f_test[[4]]$E2

#tables
TSTcall  = list()
TSTput   = list()
TST      = list()
```

```
TT = matrix(rep(0,32),4,8)
TT = data.frame(TT)
rownames(TT) = c("No Penalty", "Penalty (Ru)", "Penalty (
  Rpc)", "Penalty (Ru+Rpc)")
names(TT) = c("MAE", "25th Perc.", "50th Perc.", "75th Perc
  .", "100th Perc.", "Upper Bound", "Relative Bound","
  Iterations")

Tcall = TT
Tput = TT

for (Dz in 1:2){
  if (Dz ==1){Test_Data = Dat1}else{Test_Data = Dat2}
  # Test_Data = DAT[Dz]
  Test_X = as.matrix(Test_Data[,2:7])
  colnames(Test_X) = NULL
  Test_Y = as.matrix(Test_Data[,10:11])
  Test_SK= as.matrix(Test_Data[,c(2,7)])*100
  #


  for (z in 1:4){

    lr = 0.05
    goal = 2
    b1 = 0.4
    b2 = 1.6
    a1 = 2

    pen = z-1
    # if (Dz ==1){
    #   res_fit_test = res_f_test[[z]]
    # }else{
    #   res_fit_test = res_f_Stest[[z]]
    # }
    # res_fit        = res_f[[z]]
    # res_fit_test   = res_f_test[[z]]
    # res_fit_Stest  = res_f_Stest[[z]]
    res_opt          = res[[z]]
    res_fit_test     = neural_net(Test_X,Test_Y,Test_SK,res_
      opt$theta_hat,m, lambda,pen , b1 , b2, a1)

    Tcall[z,1]       = round(100*mean(abs(exp(res_fit_test$out
      [,1]) - exp(Test_Y[,1]))/exp(Test_Y[,1])),3)
    Tput[z,1]        = round(100*mean(abs(exp(res_fit_test$out
      [,2]) - exp(Test_Y[,2]))/exp(Test_Y[,2])),3)
    TT[z,1]          = round(100*mean(abs(exp(res_fit_test$out
      [,1:2]) - exp(Test_Y[,1:2]))/exp(Test_Y[,1:2])),3)
```

```
    q1  = round(100*abs(exp(res_fit_test$out[,1]) - exp(
      Test_Y[,1]))/exp(Test_Y[,1]),3)
    q2  = round(100*abs(exp(res_fit_test$out[,2]) - exp(
      Test_Y[,2]))/exp(Test_Y[,2]),3)
    qb  = round(100*abs(exp(res_fit_test$out[,1:2]) - exp(
      Test_Y[,1:2]))/exp(Test_Y[,1:2]),3)
    # qs1 = round(100*abs(exp(res_fit_Stestx$out[,1]) - exp
      (STest_Y[,1]))/exp(STest_Y[,1]),3)
    # qs2 = round(100*abs(exp(res_fit_Stestx$out[,2]) - exp
      (STest_Y[,2]))/exp(STest_Y[,2]),3)
    # qsb = round(100*abs(exp(res_fit_Stestx$out[,1:2]) -
      exp(STest_Y[,1:2]))/exp(STest_Y[,1:2]),3)

    Tcall[z,2:5] = quantile(q1, probs = c(0.25,0.5, 0.75,
      1))
    Tput[z,2:5]  = quantile(q2, probs = c(0.25,0.5, 0.75,
      1))
    TT[z,2:5]    = quantile(qb, probs = c(0.25,0.5, 0.75,
      1))

    Tcall[z,6]   = round(length(which(h(res_fit_test$out
      [,1],Test_SK[,1])>1))/nrow(Test_SK) *100,3)
    Tput[z,6]    = round(length(which(h(res_fit_test$out
      [,2],Test_SK[,2])>1))/nrow(Test_SK) *100,3)
    TT[z,6]      = round(res_fit_test$UB*100,3)

    Tcall[z,7]   = round(res_fit_test$RB,3)
    Tput[z,7]    = round(res_fit_test$RB,3)
    TT[z,7]      = round(res_fit_test$RB,3)

    Tcall[z,8]   = res_opt$runs
    Tput[z,8]    = res_opt$runs
    TT[z,8]      = res_opt$runs

  }
  TSTcall[[Dz]]  = Tcall
  TSTput[[Dz]]   = Tput
  TST[[Dz]]      = TT
}

TSTcall[[1]]
TSTcall[[2]]

TSTput[[1]]
TSTput[[2]]

TST[[1]]
TST[[2]]


plot(res[[1]]$errors, type = 'l')
```

```
plot(res[[3]]$errors, type = 'l')
which(res[[4]]$errors == min(res[[4]]$errors))
which(res[[3]]$errors == min(res[[3]]$errors))
library(xtable)

t3 = TSTcall[[1]]
x = xtable(t3, auto = TRUE, type = "latex", caption = '
  Experiment 3: Call Option Test Data')
print(x, file = "Exp3.Call.T_alt.tex", caption.placement =
  'bottom',scalebox='0.65')
t3 = TSTcall[[2]]
x = xtable(t3, auto = TRUE, type = "latex", caption = '
  Experiment 3: Call Option Stress Test Data')
print(x, file = "Exp3.Call.ST_alt.tex", caption.placement =
   'bottom',scalebox='0.65')


t3 = TSTput[[1]]
x = xtable(t3, auto = TRUE, type = "latex", caption = '
  Experiment 3: Put Option Test Data')
print(x, file = "Exp3.Put.T_alt.tex", caption.placement = '
  bottom',scalebox='0.65')
t3 = TSTput[[2]]
x = xtable(t3, auto = TRUE, type = "latex", caption = '
  Experiment 3: Put Option Stress Test Data')
print(x, file = "Exp3.Put.ST_alt.tex", caption.placement =
  'bottom',scalebox='0.65')


t3 = TST[[1]]
x = xtable(t3, auto = TRUE, type = "latex", caption = '
  Experiment 3: Put and Call Option Test Data')
print(x, file = "Exp3.Overall.T_alt.tex", caption.placement
   = 'bottom',scalebox='0.65')
t3 = TST[[2]]
x = xtable(t3, auto = TRUE, type = "latex", caption = '
  Experiment 3: Put and Call Option Stress Test Data')
print(x, file = "Exp3.Overall.ST_alt.tex", caption.
  placement = 'bottom',scalebox='0.65')


save(TST,TSTcall,TSTput,res, res_f, res_f_Stest, res_f_test
  , file = "Exp3.2alt_results.Rdata")
```

A.6 Experiment 4 Code

A.6.1 Real World Data

```
library(quantmod)
library(rvest)
library(dplyr)
```

```
#real world data script


gBSM = function(S, X, sigma, r, q, ttm, type){
  #S = stock price
  #X = strike price
  #sigma = volatility
  #r = risk free interest rate
  #q = dividend yield
  #ttm = time to maturity in days
  #type = option type

  t = ttm/365.25

  d1 = (log(S / X) + (r - q + sigma ^ 2 / 2) * t) / (sigma
    * sqrt(t))
  d2 = d1 - sigma * sqrt(t)

  if(type == "call"){
    price <- S * exp((-q) * t) * pnorm(d1) - X * exp(-r * t
      ) * pnorm(d2)
  }else if (type == "put"){
    price <-  (X * exp(-r * t) * pnorm(-d2) - S * exp((-q)
      * t) * pnorm(-d1))
  }

  return(price)
}



volOptimFun = function(sigma, price, S, K, r, q, ttm, type)
  {
  abs(price - gBSM(S, K, sigma, r, q, ttm, type))
}


getIV <- function(x, type){
  if(type == "call"){
    res <- optimize(volOptimFun, interval = c(0, 4), price
      = as.numeric(x["Call.Price"]), S = as.numeric(x["So"
      ]),
                    K = as.numeric(x["strike"]), r = as.
                      numeric(x["r"]), q = as.numeric(x["q"
                      ]), ttm = as.numeric(x["ttm"]), type
                      = type)
  }else if (type == "put"){
    res <- optimize(volOptimFun, interval = c(0, 4), price
      = as.numeric(x["Put.Price"]), S = as.numeric(x["So"])
      ,
```

```
                    K = as.numeric(x["strike"]), r = as.
                      numeric(x["r"]), q = as.numeric(x["q"
                      ]), ttm = as.numeric(x["ttm"]), type
                      = type)
  }
  # res <- optimize(volOptimFun, interval = c(0, 2), price
    = as.numeric(x["ask"]), S = S,
  #                 K = as.numeric(x["strike"]), r = r, q =
     q, ttm = as.numeric(x["ttm"]), type = type)
  return(res$minimum)
}


getND2 <- function(x,sigma, type){
  S = as.numeric(x["So"])
  X = as.numeric(x["strike"])
  r = as.numeric(x["r"])
  q = as.numeric(x["q"])
  t = as.numeric(x["ttm"])/365.25
  if (type == "call"){
    sigma = as.numeric(x["VolCall"])
    d1 = (log(S / X) + (r - q + sigma ^ 2 / 2) * t) / (
      sigma * sqrt(t))
    d2 = d1 - sigma * sqrt(t)
    res <- pnorm(d2)
  }else if (type == "put"){
    sigma = as.numeric(x["VolPut"])
    d1 = (log(S / X) + (r - q + sigma ^ 2 / 2) * t) / (
      sigma * sqrt(t))
    d2 = d1 - sigma * sqrt(t)
    res <- pnorm(-d2)
  }
  return(res)
}

#first get the yield curve as the risk free rate

TBs = c("DGS1MO","DGS3MO","DGS6MO","DGS1","DGS2","DGS3","
  DGS5","DGS10","DGS20")
YC = matrix(rep(0,length(TBs)* 2),length(TBs),2 )
YC[,1] = c(1/12,0.25,0.5,1,2,3,5,10,20)
for (t in 1:length(TBs)){
  tb = TBs[t]
  rs = getSymbols(tb,src="FRED", auto.assign=FALSE) # FX
    rates from FRED
  lst = length(rs)
  YC[t,2] = rs[lst]
}

#test it
f = smooth.spline(YC[,2]~YC[,1],df =8)
```

```
Ttm = c(0.4,0.8,3,4.5)
r = predict(f,Ttm)[2]
r = unlist(r)# use f and term to maturity to get r for
  options

B = list()
tickers = c('QQQ', 'SPY', 'AAPL', 'MSFT', 'GOOGL', 'T', 'GS
  ', 'JPM', 'TSLA', 'F', 'GOLD', 'JNJ','IVV')
tickers = c('QQQ', 'SPY', 'AAPL', 'MSFT', 'T', 'GS', 'JPM',
   'TSLA', 'F', 'GOLD', 'JNJ','IVV')
tst = matrix(rep(0, 2*length(tickers)), length(tickers), 2)
# tickers = c('QQQ', 'SPY', 'MSFT', 'T', 'GS', 'JPM',  'F',
   'GOLD', 'JNJ','IVV')
for (j in 1:length(tickers)){
  symbol = tickers[j]

  #now get q and So

  priceInfo <- getQuote(symbol)
  lastPrice <- priceInfo$Last
  divYield <- getQuote(symbol, what = yahooQF("Dividend
    Yield"))$`Dividend Yield`
  if(is.na(divYield)){divYield <- 0}
  date <- as.Date(priceInfo$`Trade Time`)


  #settings for moneyness and time to maturity
  #moneynessBoundaries <- c(0.85,1.15)
  ttmBoundaries <- c(90, 2000)

  #scrape full site
  baseUrl <- paste0("https://finance.yahoo.com/quote/",
    symbol,"/options")
  baseHTML <- read_html(baseUrl)

  #get available expiries and convert to time to maturity
  expiriesUNIX <- baseHTML %>% html_nodes("option") %>%
    html_attr("value")
  expiries <- as.Date((baseHTML %>% html_nodes("option")
    %>% html_text()), format = "%b %d, %Y")
  timeToMats <- as.numeric(expiries - date)

  #select applicable expiries
  sel <- timeToMats >= ttmBoundaries[1] & timeToMats <=
    ttmBoundaries[2]

  expiriesUNIX <- expiriesUNIX[sel]
  expiries <- expiries[sel]
  timeToMats <- timeToMats[sel]

  tst[j,1] = symbol
```

```
tst[j,2] = length(expiriesUNIX)
#loop over expiries to get calls and puts
calls <- NULL
puts <- NULL

print(symbol)
for(i in 1:length(expiriesUNIX)){
  expiryUrl <- paste0(baseUrl,"?date=",expiriesUNIX[i])
  expiryHTML <- read_html(expiryUrl)
  tmpCalls <- expiryHTML %>% html_nodes(".calls") %>%
    html_table()
  if(length(tmpCalls) > 0){
    tmpCalls <- tmpCalls[[1]]
    #sometimes column names are in uppercase, sometimes
      not
    colnames(tmpCalls) <- tolower(colnames(tmpCalls))
    #remove thousand separator and convert to numeric if
      applicable
    tmpCalls$strike <- as.numeric(gsub(",","",tmpCalls$
      strike))
    #add time to maturity
    tmpCalls$ttm <- timeToMats[i]
    #calculate moneyness
    tmpCalls$moneyness <- lastPrice/tmpCalls$strike
    #convert yahoo finance IV to numeric
    tmpCalls$ivOrig <- as.numeric(gsub("%","",gsub(",",""
      ,tmpCalls$`implied volatility`)))/100
    calls <- rbind(calls, tmpCalls)
  }

  tmpPuts <- expiryHTML %>% html_nodes(".puts") %>% html_
    table()
  if(length(tmpPuts) > 0){
    tmpPuts <- tmpPuts[[1]]
    #sometimes column names are in uppercase, sometimes
      not
    colnames(tmpPuts) <- tolower(colnames(tmpPuts))
    #remove thousand separator and convert to numeric if
      applicable
    tmpPuts$strike <- as.numeric(gsub(",","",tmpPuts$
      strike))
    #add time to maturity
    tmpPuts$ttm <- timeToMats[i]
    #calculate moneyness
    tmpPuts$moneyness <- tmpPuts$strike/lastPrice
    #convert yahoo finance IV to numeric
    tmpPuts$ivOrig <- as.numeric(gsub("%","",gsub(",","",
      tmpPuts$`implied volatility`)))/100
    puts <- rbind(puts, tmpPuts)
  }
}
```

```
  if((length(puts) > 0 ) | (length(calls) > 0)){
    feats = c('strike', 'ttm', 'last price')
    calls = calls[, feats]
    puts  = puts[,feats]

    calls$Call.Price = calls$`last price`
    puts$Put.Price   = puts$`last price`

    both = merge(calls, puts, by = c('strike', 'ttm'))
    both = both[,c('strike', 'ttm', 'Call.Price', 'Put.
      Price')]
    both$r = unlist(predict(f,both$ttm/365.25)[2])/100
    both$q = divYield
    both$So = lastPrice

    both$VolCall = apply(both,1, getIV,  type = "call")
    both$VolPut = apply(both,1, getIV,  type = "put")
    both$Vol  = (both$VolPut + both$VolCall)/2

    #calculating IV
    # calls$iv <- apply(calls, 1, getIV, S = lastPrice, r =
       0.0011, q = divYield, type = "call")
    # puts$iv <- apply(puts, 1, getIV, S = lastPrice, r =
      0.0011, q = divYield, type = "put")


    #now moneyness filter d2 applies

    both$ND2 = apply(both,1, getND2,  type = "call")
    both$NnegD2 = apply(both,1, getND2,  type = "put")
    both$tckr = symbol
    B[[j]] = both
    both = both[(both$ND2 > 0.25 & both$NnegD2 > 0.25), ]

    if (j==1){
      Data = both
    }else{Data = rbind(Data, both)}

  }else{
    baseUrl <- NULL
    baseHTML <- NULL

  }


}

dim(Data)
hist(B[[1]]$ND2)
```

```
save(Data, B, file = "Real_World_Data.Rdata")

setwd("C:/Users/Frans/Desktop/UCT Masters/CSV Files")

#Now check data and Put Call Parity
load("Real_World_Data.Rdata")
Data$Call.Price = as.numeric(Data$Call.Price)
Data$Put.Price = as.numeric(Data$Put.Price)
Data$Tt = Data$ttm/365.25
Data = Data[, c('tckr','So','r', 'q', 'Tt', 'strike', '
  VolCall', 'VolPut', 'Put.Price', 'Call.Price')]
names(Data) = c('tckr','So','r', 'q', 'Tt', 'K', 'VolCall',
   'VolPut', 'Put.Price', 'Call.Price')

Data$PCPar = Data$Call.Price+exp(-Data$r*Data$Tt)*Data$K-
  Data$Put.Price-Data$So*exp(-Data$q*Data$Tt)

mean(Data$PCPar/Data$K)
hist(Data$PCPar/Data$K)
quantile(Data$PCPar/Data$K)

length(which(abs(Data$PCPar/Data$K)>0.1))
nrow(Data[which(abs(Data$PCPar/Data$K)<0.1),])
Data = Data[which(abs(Data$PCPar/Data$K)<0.1),]
hist(Data$PCPar/Data$K)

pdf(file="RW_Put_Call.pdf", width = 6, height = 6)
hist(Data$PCPar/Data$K, main = 'Histogram of Real World Put
  -Call Parity', freq = F, xlab = 'Put-Call Difference')
dev.off()

Data = Data[, c('tckr','So','r', 'q', 'Tt', 'K', 'VolCall',
   'VolPut', 'Put.Price', 'Call.Price')]


scrmbl = sample(1:nrow(Data), nrow(Data), replace = F)
Data = Data[scrmbl,]
save(Data, file = "Real_World_Data_Use.Rdata")
```

A.6.2 Experiment 4.0: Penalty Control Parameters

```
rm(list = ls())
# Define activation functions and their first derivitives.
# Define the sum-of-squares objective g and its first
  derivitive w.r.t. the final layer of activations.
sig = function(x)
{
  x
}
sig. = function(x)
{
```

```
  1
}
sig2 = function(x)
{
  (2/(1+exp(-2*x)))-1
}
sig2. = function(x)
{
  ((4*exp(-2*x))/(1+exp(-2*x))^2)
}
g      = function(yhat,y)
{
  (yhat - y)^2
}
g.     = function(yhat,y)
{
  2*(yhat - y)
}

Ru = function(x)
{
  1000/(1+exp(-10*(x-2)))
}


Ru. = function(x)
{
  10000/(1+exp(-10*(x-2)))*(1-1/(1+exp(-10*(x-2))))
}


h = function(x,s){

  exp(x)/s
}
h. = function(x,s){

  exp(x)/s
}

Rpc = function(x){
  (10*x)^2
}

Rpc. = function(x){
  20*x
}


# N = observations
# p = input features
```

```
# q = outputs
# m = neurons in layer

neural_net = function(X,Y, SK, theta,m,lam, pen, beta1,
  beta2, alpha)
{


  N   = dim(X)[1]
  p   = dim(X)[2]
  q   = dim(Y)[2]
  m = append(m,q)
  Neuron = m
  L   = length(m)

  Ix = cbind(matrix(rep(-1,nrow(Y)*1), nrow(Y), 1), matrix(
    rep(1,nrow(Y)*1), nrow(Y), 1))
  K = cbind(SK[,2],SK[,2])
  QR = cbind(exp(-(3*X[,2])*(0.04*X[,4])),-exp(-(3*X[,2])
    *(0.04*X[,3])))

  #l = list()
  W   = list()
  B   = list()
  A   = list()
  d   = list()
  dW = list()
  dB = list()


  a0    = X
  ones =  matrix(1,N,1)

  for (i in 1:L)
  {
    if (i ==1) {
      index   = 1:(p*Neuron[i])
      W[[i]] = matrix(theta[index],p,Neuron[i])
      index   = max(index)+1:(Neuron[i])
      B[[i]] = matrix(theta[index],1,Neuron[i])

      dW[[i]]= W[[i]]*0
      dB[[i]]= B[[i]]*0

      A[[i]] = sig2(a0%*%W[[i]]+ones%*%B[[i]])

      rdg = sum(W[[i]]^2)
    } else if (i ==L){
      index   = max(index)+1:(Neuron[i-1]*Neuron[i])
      W[[i]] = matrix(theta[index],Neuron[i-1],Neuron[i])
      index   = max(index)+1:(Neuron[i])
```

```
      B[[i]] = matrix(theta[index],1,Neuron[i])

      dW[[i]]= W[[i]]*0
      dB[[i]]= B[[i]]*0

      A[[i]] = sig(A[[i-1]]%*%W[[i]]+ones%*%B[[i]])

      rdg = rdg + sum(W[[i]]^2)
    } else {
      index  = max(index)+1:(Neuron[i-1]*Neuron[i])
      W[[i]] = matrix(theta[index],Neuron[i-1],Neuron[i])
      index  = max(index)+1:(Neuron[i])
      B[[i]] = matrix(theta[index],1,Neuron[i])

      dW[[i]]= W[[i]]*0
      dB[[i]]= B[[i]]*0

      A[[i]] = sig2(A[[i-1]]%*%W[[i]]+ones%*%B[[i]])

      rdg = rdg + sum(W[[i]]^2)
    }

  }

  pc = ((exp(A[[L]]) * Ix) + (SK * QR))/K
  PC = pc[,1]+pc[,2]
  PC = cbind(PC, PC) #same penalty applies to put and call

  for (j in 1:(L-1))
  {
    if (j ==1){
      if (pen ==1){ #upper bound only
        d[[L+1-j]]  = (g.(A[[L+1-j]],Y) + beta1*Ru.(h(A[[L
          +1-j]],SK))*h.(A[[L+1-j]],SK))
      } else if (pen ==2){ #rel bound only
        d[[L+1-j]]  = (g.(A[[L+1-j]],Y) + beta2*Rpc.(PC)*h
          .(A[[L+1-j]],K)*Ix)
      } else if (pen ==3){ #both penalties
        d[[L+1-j]]  = (g.(A[[L+1-j]],Y) + alpha*Ru.(h(A[[L
          +1-j]],SK))*h.(A[[L+1-j]],SK)
                      + alpha*Rpc.(PC)*h.(A[[L+1-j]],K)*Ix
                        )
      }
      else {d[[L+1-j]]  = (g.(A[[L+1-j]],Y))}

      d[[L+1-j]]  = d[[L+1-j]]*sig.(A[[L-j]]%*%W[[L+1-j]]+
        ones%*%B[[L+1-j]])
      dB[[L+1-j]]  = t(ones) %*%d[[L+1-j]]
      dW[[L+1-j]]  = (t(A[[L-j]]) %*% d[[L+1-j]]) + 2*lam*W
        [[L+1-j]]
    } else {
```

```
      d[[L+1-j]]  = (d[[L+2-j]] %*% t(W[[L+2-j]]))*sig2.(A
        [[L-j]]%*%W[[L+1-j]]+ones%*%B[[L+1-j]])
      dB[[L+1-j]] = t(ones) %*%d[[L+1-j]]
      dW[[L+1-j]] = (t(A[[L-j]]) %*% d[[L+1-j]]) + 2*lam*W
        [[L+1-j]]
    }
  }
  d[[1]]  = (d[[2]] %*% t(W[[2]]))*sig2.(a0%*%W[[1]]+ones
    %*%B[[1]])
  dB[[1]] = t(ones) %*%d[[1]]
  dW[[1]] = (t(a0) %*% d[[1]]) + 2*lam*W[[1]]

  out   = A[[L]]
  error = (Y - out)^2
  pc = ((exp(A[[L]]) * Ix) + (SK * QR))/K
  PC = pc[,1]+pc[,2]
  PC = cbind(PC, PC) #same penalty applies to put and call

  E1    = sum(error) #pure error
  E2    = E1 + lam*(rdg) #error with ridge
  if (pen ==1){E3    = E2 + beta1*sum(Ru(h(out,SK)))}
  else if (pen ==2){E3    = E2 + beta2*sum(Rpc(PC))}
  else if (pen ==3){E3    = E2 + alpha*sum(Ru(h(out,SK))) +
     alpha*sum(Rpc(PC))}
  else {E3    = E2}
  #E3    = E1 + lam*(rdg) + sum(Ru(h(out,SK)))
  UB    = length(which(h(out,SK)>1))
  RB    = length(which(PC>0.05))
  grad = c(dW[[1]], dB[[1]])
  for (i in 2:L){
    grad = c(grad, dW[[i]], dB[[i]])
  }
  return(list(grad = grad/(2*N),out = out, E1 = E1/(2*N),
    E2 = E2/(2*N), E3 = E3/(2*N),
                UB= UB*100/(2*N), RB= RB*100/(2*N)))
}



# theta_start = initialized parameters, W and B
# learn = learning rate
# iterations = epochs
grad_dec = function(theta_start,learn,iterations,goal, pen,
   beta1, beta2, alpha)
{
  theta = theta_start #initialize
  res   = neural_net(X,Y,SK,theta,m,lambda,pen, beta1,
    beta2, alpha) #feed forward
  errors  = rep(NA,iterations) #error vector
  errors2 = rep(NA,iterations) #error vector
  ubound  = rep(NA,iterations) #error vector
```

```
  rbound  = rep(NA,iterations)
  errors[1] = res$E2 #first error
  errors2[1] = res$E3
  ubound[1] = res$UB
  rbound[1] = res$RB

  theta_mat = matrix(NA,length(theta),iterations) #matrix
    with all para as rows, columns as iteration
  theta_mat[,1] = theta
  x = 100
  i=2
  #for(i in 2:iterations)
  while(x > goal)
  {
    learn = ifelse(i<5000, learn, ifelse(i<200000, learn,
      learn/1))
    theta      = theta-learn*res$grad #update parameters
    theta_mat[,i] = theta #put in para matrix
    set        = sample(1:N,N*0.3, replace = F) #take a
      batch, SGD,
    res        = neural_net(matrix(X[set,],ncol = p),matrix(
      Y[set,],ncol = q),
                             matrix(SK[set,],ncol = q), theta
                               ,m,lambda,pen, beta1, beta2,
                               alpha) #feed forward
    errors[i] = res$E2 #get new error
    errors2[i] = res$E3
    ubound[i] = res$UB
    rbound[i] = res$RB
    trainer    = round(100*mean(abs(exp(res$out[,1:2]) - exp
      (matrix(Y[set,],ncol = q)[,1:2]))/exp(matrix(Y[set,],
      ncol = q)[,1:2])),3)
    x = trainer
    itx = iterations
    if(i == itx){x = 0}
    i = i+1
  }
  ret = list(theta_hat = theta,errors = errors, errors2 =
    errors2,
             ubound = ubound,rbound = rbound,theta_mat =
               theta_mat, runs = i)
  return(ret)
}

# read in data
load("Real_World_Data_Use.Rdata")

Data$So = Data$So/100
Data$K = Data$K/100
Data$r = Data$r*100/(0.4)
Data$q = Data$q/(0.1)
```

```
Data$Tt = Data$Tt/3
Data$logCall = log(Data$Call.Price)
Data$logPut = log(Data$Put.Price)

Dat = Data


## Training
obs = nrow(Dat)
train = 0.8
N = obs * train
Data = Data[1:N,]
X = as.matrix(Data[,2:8])
colnames(X) = NULL
Y = as.matrix(Data[,11:12])
SK= as.matrix(Data[,c(2,6)])*100

##Testing
Test_Data = Dat[(N +1):obs,]
Dat1 = Test_Data
Test_X = as.matrix(Test_Data[,2:8])
colnames(Test_X) = NULL
Test_Y = as.matrix(Test_Data[,11:12])
Test_SK= as.matrix(Test_Data[,c(2,6)])*100

p = dim(X)[2]
q = dim(Y)[2]
#load data from Exp1.2 and select lambda, initialization,
  Layers, Neurons

load("Exp1.2.Rdata")

j = 1
L = as.double(g[j,1])
if(L==3){m =  c(as.matrix(g[j,2:4]))}else if (L==2){m = c(
  as.matrix(g[j,2:3]))}else{ m=g[j,2] }
mm = append(m, q)
npars = (p+1) * mm[1]
for (i in 2:(length(mm))){
  npars = npars +((mm[i-1]+1)*mm[i])
}
theta_rand = runif(npars,-2,2) #initialize para
# theta_rand = c(unlist(THETA[g$order[j]]))#initialize para
#
# npars = length(theta_rand)
theta_init = theta_rand

c1 = which.min(apply(matrix(lg[j,],1,length(gg)),MARGIN=2,
  min))
lambda = gg[c1]
```

```
#first train for 10 iterations - make sure Ru doesn't cause
   numerical issues
lr = 0.07
iter = 5
goal = 2.5
pen = 0
b1 = 0.4
b2 = 2
a1 = 2



res_opt = grad_dec(theta_rand, lr,iter,goal, pen , b1 , b2,
   a1) #optimize
theta_rand = res_opt$theta_hat #use new weights
res_fit = neural_net(X,Y,SK,res_opt$theta_hat,m, lambda,
  pen , b1 , b2, a1)




#beta1 grid
Bone = c(15:1)/5
Gb1  = matrix(rep(0, 1*length(Bone)),1, length(Bone))
gr1 = matrix(rep(0, 2*length(Bone)),2, length(Bone))

for (i in 1:length(Gb1)){
  lr = 0.07
  iter = 200000
  goal = 3
  pen = 1
  b1 = Bone[i]
  b2 = 0.4
  a1 = 0.5
  res_opt = grad_dec(theta_rand, lr,iter,goal, pen , b1 ,
    b2, a1) #optimize

  res_fit = neural_net(X,Y,SK,res_opt$theta_hat,m, lambda,
    pen , b1 , b2, a1)
  res_fit_test = neural_net(Test_X,Test_Y,Test_SK,res_opt$
    theta_hat,m, lambda, pen , b1 , b2, a1)

  gr1[1,i] = round(100*(res_fit_test$E3 - res_fit_test$E2)/
    b1,5)
  gr1[2,i] = round(100*mean(abs(exp(res_fit_test$out[,1:2])
     - exp(Test_Y[,1:2]))/exp(Test_Y[,1:2])),3)
}
#beta2 grid
Btwo = c(15:1)/5
Gb2  = matrix(rep(0, 1*length(Btwo)),1, length(Btwo))
gr2  = matrix(rep(0, 2*length(Btwo)),2, length(Btwo))
```

```
for (i in 1:length(Gb2)){
  lr = 0.07
  iter = 200000
  goal = 3
  pen = 2
  b1 = 0.2
  b2 = Btwo[i]
  a1 = 0.5
  res_opt = grad_dec(theta_rand, lr,iter,goal, pen , b1 ,
    b2, a1) #optimize

  res_fit = neural_net(X,Y,SK,res_opt$theta_hat,m, lambda,
    pen , b1 , b2, a1)
  res_fit_test = neural_net(Test_X,Test_Y,Test_SK,res_opt$
    theta_hat,m, lambda, pen , b1 , b2, a1)

  gr2[1,i] = round(100*(res_fit_test$E3 - res_fit_test$E2)/
    b2,5)
  gr2[2,i] = round(100*mean(abs(exp(res_fit_test$out[,1:2])
     - exp(Test_Y[,1:2]))/exp(Test_Y[,1:2])),3)
}
#alpha grid
Alph  = c(15:1)/5
Galph = matrix(rep(0, 1*length(Alph)),1, length(Alph))
gr3 = matrix(rep(0, 2*length(Alph)),2, length(Alph))
for (i in 1:length(Alph)){
  lr = 0.07
  iter = 200000
  goal = 3
  pen = 3
  b1 = 0.2
  b2 = 0.4
  a1 = Alph[1]
  res_opt = grad_dec(theta_rand, 0.02,iter,goal, pen , b1 ,
     b2, a1) #optimize

  res_fit = neural_net(X,Y,SK,res_opt$theta_hat,m, lambda,
    pen , b1 , b2, a1)
  res_fit_test = neural_net(Test_X,Test_Y,Test_SK,res_opt$
    theta_hat,m, lambda, pen , b1 , b2, a1)

  gr3[1,i] = round(100*(res_fit_test$E3 - res_fit_test$E2)/
    a1,5)
  gr3[2,i] = round(100*mean(abs(exp(res_fit_test$out[,1:2])
     - exp(Test_Y[,1:2]))/exp(Test_Y[,1:2])),3)
}

save(gr1,gr2,gr3,Alph, theta_init, file = "Exp4_beta_alpha.
  Rdata")
Bone = Btwo = Alph
```

```
plot(Bone, gr1[1,], type = 'l', xlab = "Beta1", ylab = "Ru"
  , main = "CVE (MAE) by Lambda: Ru Penalty")
plot(Btwo, gr2[1,], type = 'l',xlab = "Beta2", ylab = "Rpc"
  , main = "CVE (MAE) by Lambda: Rpc Penalty")
plot(Alph, gr3[1,], type = 'l',xlab = "Alpha", ylab = "Ru +
   Rpc", main = "CVE (MAE) by Lambda: Ru + Rpc Penalty")


plot(Bone, gr1[2,], type = 'l', xlab = "Beta1", ylab = "MAE
  ", main = "CVE (MAE) by Lambda: Ru Penalty")
plot(Btwo, gr2[2,], type = 'l',xlab = "Beta2", ylab = "MAE"
  , main = "CVE (MAE) by Lambda: Rpc Penalty")
plot(Alph, gr3[2,], type = 'l',xlab = "Alpha", ylab = "MAE"
  , main = "CVE (MAE) by Lambda: Ru+Rpc Penalty")

pdf(file="Exp4_Beta1.pdf", width = 8.27, height = 5)
plot(Bone, gr1[1,], type = 'l', xlab = "Beta1", ylab = "Ru"
  , main = "Ru Penalty by Beta1")
dev.off()

pdf(file="Exp4_Beta2.pdf", width = 8.27, height = 5)
plot(Btwo, gr2[1,], type = 'l',xlab = "Beta2", ylab = "Rpc"
  , main = "Rpc Penalty by Beta2")
dev.off()

pdf(file="Exp4_Alph.pdf", width = 8.27, height = 5)
plot(Alph, gr3[1,], type = 'l',xlab = "Alpha", ylab = "Ru +
   Rpc", main = "Ru + Rpc Penalty by Alpha")
dev.off()
```

A.6.3 Experiment 4.1: Real World Models

```
rm(list = ls())
# Define activation functions and their first derivitives.
# Define the sum-of-squares objective g and its first
  derivitive w.r.t. the final layer of activations.
sig = function(x)
{
  x
}
sig. = function(x)
{
  1
}
sig2 = function(x)
{
  (2/(1+exp(-2*x)))-1
}
sig2. = function(x)
{
  ((4*exp(-2*x))/(1+exp(-2*x))^2)
```

```
}
g      = function(yhat,y)
{
  (yhat - y)^2
}
g.     = function(yhat,y)
{
  2*(yhat - y)
}

Ru = function(x)
{
  1000/(1+exp(-10*(x-2)))
}


Ru. = function(x)
{
  10000/(1+exp(-10*(x-2)))*(1-1/(1+exp(-10*(x-2))))
}


h = function(x,s){

  exp(x)/s
}
h. = function(x,s){

  exp(x)/s
}

Rpc = function(x){
  (10*x)^2
}

Rpc. = function(x){
  20*x
}


# N = observations
# p = input features
# q = outputs
# m = neurons in layer

neural_net = function(X,Y, SK, theta,m,lam, pen, beta1,
  beta2, alpha)
{


  N  = dim(X)[1]
```

```
p  = dim(X)[2]
q  = dim(Y)[2]
m = append(m,q)
Neuron = m
L  = length(m)

Ix = cbind(matrix(rep(-1,nrow(Y)*1), nrow(Y), 1), matrix(
  rep(1,nrow(Y)*1), nrow(Y), 1))
K = cbind(SK[,2],SK[,2])
QR = cbind(exp(-(3*X[,2])*(0.04*X[,4])),-exp(-(3*X[,2])
  *(0.04*X[,3])))

#l = list()
W  = list()
B  = list()
A  = list()
d  = list()
dW = list()
dB = list()


a0   = X
ones =  matrix(1,N,1)

for (i in 1:L)
{
  if (i ==1) {
    index  = 1:(p*Neuron[i])
    W[[i]] = matrix(theta[index],p,Neuron[i])
    index  = max(index)+1:(Neuron[i])
    B[[i]] = matrix(theta[index],1,Neuron[i])

    dW[[i]]= W[[i]]*0
    dB[[i]]= B[[i]]*0

    A[[i]] = sig2(a0%*%W[[i]]+ones%*%B[[i]])

    rdg = sum(W[[i]]^2)
  } else if (i ==L){
    index  = max(index)+1:(Neuron[i-1]*Neuron[i])
    W[[i]] = matrix(theta[index],Neuron[i-1],Neuron[i])
    index  = max(index)+1:(Neuron[i])
    B[[i]] = matrix(theta[index],1,Neuron[i])

    dW[[i]]= W[[i]]*0
    dB[[i]]= B[[i]]*0

    A[[i]] = sig(A[[i-1]]%*%W[[i]]+ones%*%B[[i]])

    rdg = rdg + sum(W[[i]]^2)
  } else {
```

```
    index   = max(index)+1:(Neuron[i-1]*Neuron[i])
    W[[i]] = matrix(theta[index],Neuron[i-1],Neuron[i])
    index   = max(index)+1:(Neuron[i])
    B[[i]] = matrix(theta[index],1,Neuron[i])

    dW[[i]]= W[[i]]*0
    dB[[i]]= B[[i]]*0

    A[[i]] = sig2(A[[i-1]]%*%W[[i]]+ones%*%B[[i]])

    rdg = rdg + sum(W[[i]]^2)
  }

}

pc = ((exp(A[[L]]) * Ix) + (SK * QR))/K
PC = pc[,1]+pc[,2]
PC = cbind(PC, PC) #same penalty applies to put and call

for (j in 1:(L-1))
{
  if (j ==1){
    if (pen ==1){ #upper bound only
      d[[L+1-j]]  = (g.(A[[L+1-j]],Y) + beta1*Ru.(h(A[[L
        +1-j]],SK))*h.(A[[L+1-j]],SK))
    } else if (pen ==2){ #rel bound only
      d[[L+1-j]]  = (g.(A[[L+1-j]],Y) + beta2*Rpc.(PC)*h
        .(A[[L+1-j]],K)*Ix)
    } else if (pen ==3){ #both penalties
      d[[L+1-j]]  = (g.(A[[L+1-j]],Y) + alpha*Ru.(h(A[[L
        +1-j]],SK))*h.(A[[L+1-j]],SK)
                      + alpha*Rpc.(PC)*h.(A[[L+1-j]],K)*Ix
                        )
    }
    else {d[[L+1-j]]  = (g.(A[[L+1-j]],Y))}

    d[[L+1-j]]  = d[[L+1-j]]*sig.(A[[L-j]]%*%W[[L+1-j]]+
      ones%*%B[[L+1-j]])
    dB[[L+1-j]] = t(ones) %*%d[[L+1-j]]
    dW[[L+1-j]] = (t(A[[L-j]]) %*% d[[L+1-j]]) + 2*lam*W
      [[L+1-j]]
  } else {
    d[[L+1-j]]  = (d[[L+2-j]] %*% t(W[[L+2-j]]))*sig2.(A
      [[L-j]]%*%W[[L+1-j]]+ones%*%B[[L+1-j]])
    dB[[L+1-j]] = t(ones) %*%d[[L+1-j]]
    dW[[L+1-j]] = (t(A[[L-j]]) %*% d[[L+1-j]]) + 2*lam*W
      [[L+1-j]]
  }
}
d[[1]]  = (d[[2]] %*% t(W[[2]]))*sig2.(a0%*%W[[1]]+ones
  %*%B[[1]])
```

```
  dB[[1]] = t(ones) %*%d[[1]]
  dW[[1]] = (t(a0) %*% d[[1]]) + 2*lam*W[[1]]

  out   = A[[L]]
  error = (Y - out)^2
  pc = ((exp(A[[L]]) * Ix) + (SK * QR))/K
  PC = pc[,1]+pc[,2]
  PC = cbind(PC, PC) #same penalty applies to put and call

  E1    = sum(error) #pure error
  E2    = E1 + lam*(rdg) #error with ridge
  if (pen ==1){E3    = E2 + beta1*sum(Ru(h(out,SK)))}
  else if (pen ==2){E3    = E2 + beta2*sum(Rpc(PC))}
  else if (pen ==3){E3    = E2 + alpha*sum(Ru(h(out,SK))) +
     alpha*sum(Rpc(PC))}
  else {E3    = E2}
  #E3    = E1 + lam*(rdg) + sum(Ru(h(out,SK)))
  UB   = length(which(h(out,SK)>1))
  RB   = length(which(PC>0.05))
  grad = c(dW[[1]], dB[[1]])
  for (i in 2:L){
    grad = c(grad, dW[[i]], dB[[i]])
  }
  return(list(grad = grad/(2*N),out = out, E1 = E1/(2*N),
    E2 = E2/(2*N), E3 = E3/(2*N),
             UB= UB*100/(2*N), RB= RB*100/(2*N)))
}



# theta_start = initialized parameters, W and B
# learn = learning rate
# iterations = epochs
grad_dec = function(theta_start,learn,iterations,goal, pen,
   beta1, beta2, alpha)
{
  theta = theta_start #initialize
  res   = neural_net(X,Y,SK,theta,m,lambda,pen, beta1,
    beta2, alpha) #feed forward
  errors  = rep(NA,iterations) #error vector
  errors2 = rep(NA,iterations) #error vector
  ubound  = rep(NA,iterations) #error vector
  rbound  = rep(NA,iterations)
  errors[1] = res$E2 #first error
  errors2[1] = res$E3
  ubound[1] = res$UB
  rbound[1] = res$RB

  theta_mat = matrix(NA,length(theta),iterations) #matrix
    with all para as rows, columns as iteration
  theta_mat[,1] = theta
```

```
  x = 100
  i=2
  #for(i in 2:iterations)
  while(x > goal)
  {
    learn = ifelse(i<5000, learn, ifelse(i<200000, learn,
      learn/1))
    theta      = theta-learn*res$grad #update parameters
    theta_mat[,i] = theta #put in para matrix
    set        = sample(1:N,N*0.7, replace = F) #take a
      batch, SGD,
    res        = neural_net(matrix(X[set,],ncol = p),matrix(
      Y[set,],ncol = q),
                           matrix(SK[set,],ncol = q), theta
                             ,m,lambda,pen, beta1, beta2,
                             alpha) #feed forward
    errors[i] = res$E2 #get new error
    errors2[i] = res$E3
    ubound[i] = res$UB
    rbound[i] = res$RB
    trainer    = round(100*mean(abs(exp(res$out[,1:2]) - exp
      (matrix(Y[set,],ncol = q)[,1:2]))/exp(matrix(Y[set,],
      ncol = q)[,1:2])),3)
    x = trainer
    itx = iterations
    if(i == itx){x = 0}
    i = i+1
  }
  ret = list(theta_hat = theta,errors = errors, errors2 =
    errors2,
             ubound = ubound,rbound = rbound,theta_mat =
               theta_mat, runs = i)
  return(ret)
}

# read in data
load("Real_World_Data_Use.Rdata")

Data$So = Data$So/100
Data$K = Data$K/100
Data$r = Data$r*100/(0.4)
Data$q = Data$q/(0.1)
Data$Tt = Data$Tt/3
Data$logCall = log(Data$Call.Price)
Data$logPut = log(Data$Put.Price)

Dat = Data


## Training
obs = nrow(Dat)
```

```
train = 0.8
N = obs * train
Data = Data[1:N,]
X = as.matrix(Data[,2:8])
colnames(X) = NULL
Y = as.matrix(Data[,11:12])
SK= as.matrix(Data[,c(2,6)])*100

##Testing
Test_Data = Dat[(N +1):obs,]
Dat1 = Test_Data
Test_X = as.matrix(Test_Data[,2:8])
colnames(Test_X) = NULL
Test_Y = as.matrix(Test_Data[,11:12])
Test_SK= as.matrix(Test_Data[,c(2,6)])*100

p = dim(X)[2]
q = dim(Y)[2]
#load data from Exp1.2 and select lambda, initialization,
  Layers, Neurons

load("Exp1.2.Rdata")

j = 1
L = as.double(g[j,1])
if(L==3){m =  c(as.matrix(g[j,2:4]))}else if (L==2){m = c(
  as.matrix(g[j,2:3]))}else{ m=g[j,2] }
mm = append(m, q)

load("Exp4_beta_alpha.Rdata")
theta_rand = theta_init
npars = length(theta_rand)

c1 = which.min(apply(matrix(lg[j,],1,length(gg)),MARGIN=2,
  min))
lambda = gg[c1]
#first train for 10 iterations - make sure Ru doesn't cause
   numerical issues

iter = 5
goal = 2.5
pen = 0
b1 = 0.6
b2 = 3
a1 = 2.6
res_opt = grad_dec(theta_rand, 0.07,iter,goal, pen , b1 ,
  b2, a1) #optimize
theta_rand = res_opt$theta_hat #use new weights
res_fit = neural_net(X,Y,SK,res_opt$theta_hat,m, lambda,
  pen , b1 , b2, a1)
```

```
res_fit$E3 - res_fit$E2
max(h(res_fit$out, SK))
max(Ru(h(res_fit$out, SK)))

#train NN with no bound penalty

res = list()
res_f = list()
res_f_test = list()


lr = 0.07
iter = 800000
goal = 2
b1 = 0.6
b2 = 3
a1 = 2.6


for (z in 1:4){
  pen = z-1
  res[[z]] = grad_dec(theta_rand, lr,iter, goal, pen , b1 ,
     b2, a1) #optimize
  res_f[[z]] = neural_net(X,Y,SK,res_opt$theta_hat,m,
    lambda,pen , b1 , b2, a1) #fit and get train error
  res_f_test[[z]] = neural_net(Test_X,Test_Y,Test_SK,res_
    opt$theta_hat,m, lambda,pen , b1 , b2, a1) #CVE on
    normal data

}


res_f_test[[1]]$E3-res_f_test[[1]]$E2
res_f_test[[2]]$E3-res_f_test[[2]]$E2
res_f_test[[3]]$E3-res_f_test[[3]]$E2
res_f_test[[4]]$E3-res_f_test[[4]]$E2

for (z in 1:4){
  res_opt       = res[[z]]
  res_fit_test  = neural_net(Test_X,Test_Y,Test_SK,res_opt$
    theta_hat,m, lambda,pen , b1 , b2, a1)
  print(round(100*mean(abs(exp(res_fit_test$out[,1]) - exp(
    Test_Y[,1]))/exp(Test_Y[,1])),3))
  print(round(100*mean(abs(exp(res_fit_test$out[,2]) - exp(
    Test_Y[,2]))/exp(Test_Y[,2])),3))
  print(round(100*mean(abs(exp(res_fit_test$out[,1:2]) -
    exp(Test_Y[,1:2]))/exp(Test_Y[,1:2])),3))
  print(res_opt$runs)
}

strt = 20000
```

```
for (z in 1:4){
  res_opt        = res[[z]]
  cols = c('black','blue', 'red', 'green')
  if (z ==1){
    plot(res_opt$errors[strt:res_opt$runs], type = 'l',
      ylim = c(0,0.1), col = cols[z])
  }else{
    lines(res_opt$errors[strt:res_opt$runs], type = 'l',
      col = cols[z])
  }

}

TT = matrix(rep(0,32),4,8)
TT = data.frame(TT)
rownames(TT) = c("No Penalty", "Penalty (Ru)", "Penalty (
  Rpc)", "Penalty (Ru+Rpc)")
names(TT) = c("MAE", "25th Perc.", "50th Perc.", "75th Perc
  .", "100th Perc.", "Upper Bound", "Relative Bound","
  Iterations")

Tcall = TT
Tput = TT



  for (z in 1:4){

    lr = 0.05
    goal = 2
    b1 = 0.4
    b2 = 2
    a1 = 2

    pen = z-1


    res_opt        = res[[z]]
    res_fit_test   = neural_net(Test_X,Test_Y,Test_SK,res_
      opt$theta_hat,m, lambda,pen , b1 , b2, a1)

    Tcall[z,1]     = round(100*mean(abs(exp(res_fit_test$out
      [,1]) - exp(Test_Y[,1]))/exp(Test_Y[,1])),3)
    Tput[z,1]      = round(100*mean(abs(exp(res_fit_test$out
      [,2]) - exp(Test_Y[,2]))/exp(Test_Y[,2])),3)
    TT[z,1]        = round(100*mean(abs(exp(res_fit_test$out
      [,1:2]) - exp(Test_Y[,1:2]))/exp(Test_Y[,1:2])),3)

    q1  = round(100*abs(exp(res_fit_test$out[,1]) - exp(
      Test_Y[,1]))/exp(Test_Y[,1]),3)
```

```
    q2  = round(100*abs(exp(res_fit_test$out[,2]) - exp(
      Test_Y[,2]))/exp(Test_Y[,2]),3)
    qb  = round(100*abs(exp(res_fit_test$out[,1:2]) - exp(
      Test_Y[,1:2]))/exp(Test_Y[,1:2]),3)
    # qs1 = round(100*abs(exp(res_fit_Stestx$out[,1]) - exp
      (STest_Y[,1]))/exp(STest_Y[,1]),3)
    # qs2 = round(100*abs(exp(res_fit_Stestx$out[,2]) - exp
      (STest_Y[,2]))/exp(STest_Y[,2]),3)
    # qsb = round(100*abs(exp(res_fit_Stestx$out[,1:2]) -
      exp(STest_Y[,1:2]))/exp(STest_Y[,1:2]),3)

    Tcall[z,2:5] = quantile(q1, probs = c(0.25,0.5, 0.75,
      1))
    Tput[z,2:5]  = quantile(q2, probs = c(0.25,0.5, 0.75,
      1))
    TT[z,2:5]    = quantile(qb, probs = c(0.25,0.5, 0.75,
      1))

    Tcall[z,6]   = round(length(which(h(res_fit_test$out
      [,1],Test_SK[,1])>1))/nrow(Test_SK) *100,3)
    Tput[z,6]    = round(length(which(h(res_fit_test$out
      [,2],Test_SK[,2])>1))/nrow(Test_SK) *100,3)
    TT[z,6]      = round(res_fit_test$UB,3)

    Tcall[z,7]   = round(res_fit_test$RB,3)
    Tput[z,7]    = round(res_fit_test$RB,3)
    TT[z,7]      = round(res_fit_test$RB,3)

    Tcall[z,8]   = res_opt$runs
    Tput[z,8]    = res_opt$runs
    TT[z,8]      = res_opt$runs

  }

Tcall
Tput
TT

library(xtable)
t3 = Tcall
x = xtable(t3, auto = TRUE, type = "latex", caption = '
  Experiment 3: Call Option Test Data')
print(x, file = "Exp4.Call.RW.tex", caption.placement = '
  bottom',scalebox='0.65')

t3 = Tput
x = xtable(t3, auto = TRUE, type = "latex", caption = '
  Experiment 3: Put Option Test Data')
print(x, file = "Exp4.Put.RW.tex", caption.placement = '
  bottom',scalebox='0.65')
```

```
t3 = TT
x = xtable(t3, auto = TRUE, type = "latex", caption = '
  Experiment 3: Put and Call Option Test Data')
print(x, file = "Exp4.Overall.RW.tex", caption.placement =
  'bottom',scalebox='0.65')

save(TT,Tcall,Tput,res, res_f, res_f_Stest, res_f_test,
  file = "Exp4_results.Rdata")
```

Bibliography

Anders, Ulrich, Olaf Korn, and Christian Schmitt (1998). "Improving the pricing of options: a neural network approach". In: *Journal of Forecasting* 17.5-6, pp. 369–388. DOI: https://doi.org/10.1002/(SICI)1099-131X(1998090)17:5/6<369::AID-FOR702>3.0.CO;2-S. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/%28SICI%291099-131X%281998090%2917%3A5/6%3C369%3A%3AAID-FOR702%3E3.0.CO%3B2-S. URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/%28SICI%291099-131X%281998090%2917%3A5/6%3C369%3A%3AAID-FOR702%3E3.0.CO%3B2-S.

Apanard P., Savard K. and H. Wickramarachi (2014). *Deriving the Economic Impact of Derivatives*. Milken Institute.

Barucci, Emilio, Umberto Cherubini, and Leonardo Landi (1997). "Neural Networks for Contingent Claim Pricing via the Galerkin Method". In: *Computational Approaches to Economic Problems*. Ed. by Hans Amman, Berc Rustem, and Andrew Whinston. Boston, MA: Springer US, pp. 127–141. ISBN: 978-1-4757-2644-2. DOI: 10.1007/978-1-4757-2644-2_9. URL: https://doi.org/10.1007/978-1-4757-2644-2_9.

Bennell, Julia and Charles Sutcliffe (Oct. 2004). "Black-Scholes Versus Neural Networks in Pricing FTSE 100 Options". In: *University of Southampton - Department of Accounting and Management Science, Papers* 12. DOI: 10.1002/isaf.254.

Beucler, Tom et al. (2021). "Enforcing analytic constraints in neural networks emulating physical systems". In: *Physical Review Letters* 126.9, p. 098302.

Bottou, L. (2010). *Large-Scale Machine Learningwith Stochastic Gradient Descent*.

Bouthillier X., Konda K. Vincent P. and R. Memisevic (2016). *DROPOUT AS DATA AUGMENTATION*.

Brownlee, J. (2019a). *A Gentle Introduction to Dropout for Regularizing Deep Neural Networks*. URL: https://machinelearningmastery.com/dropout-for-regularizing-deep-neural-networks/.

— (2019b). *How to Choose Loss Functions When Training Deep Learning Neural Networks*. URL: https://machinelearningmastery.com/how-to-choose-loss-functions-when-training-deep-learning-neural-networks/.

Culkin R. Das, SR. (2017). "Machine Learning in Finance: The Case of Deep Learning for Option Pricing". In: *Annalen der Physik* 10. DOI: http://dx.doi.org/10.1002/andp.19053221004.

Goodfellow I., Bengio Y. and A. Courville (2016). *Deep Learning*. MIT Press. URL: http://www.deeplearningbook.org.

Haring, S. (2020). *Implied Volatility in R — assess options risk correctly*. URL: https://towardsdatascience.com/implied-volatility-in-r-assess-options-risk-correctly-70fe36843474.

Hastie, T. and J. Qian (2014). *Glmnet Vignette*. URL: https://web.stanford.edu/~hastie/glmnet/glmnet_alpha.html#poi.

Haussamer, N. (2018). "Model Calibration with Machine Learning". MA thesis. University of Cape Town.

Huang J., Chai J. Cho S. (2020). "Deep learning in finance and banking: A literature review and classification". In: *Frontiers of Business Research in China* 13.

Hull, JC. (July 2012). *Options, Futures and Other Derivatives*. 8th ed. Vol. 4. 10. An optional note. The address: Pearson. ISBN: 3257227892.

Hutchinson, James M., Andrew W. Lo, and Tomaso Poggio (1994). "A Nonparametric Approach to Pricing and Hedging Derivative Securities Via Learning Networks". In: *The Journal of Finance* 49.3, pp. 851–889. ISSN: 00221082, 15406261. URL: http://www.jstor.org/stable/2329209.

Ibrahim, AN. (2014). *Data Normalization and Standardization for Neural Networks Output Classification*. URL: https://ahmedhanibrahim.wordpress.com/2014/10/10/data-normalization-and-standardization-for-neural-networks-output-classification/c.

Ioffe, S. and C. Szegedy (2015). *Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift*.

Lajbcyguer PR. Connor, JT. (1997). *Improved Option Pricing Using Artificial Neural Networks and Bootstrap Methods*.

Mavuso M., Marr A. Mitoulis N. and A. Singh (2017). *Model calibration with neural networks*. URL: https://www.quantlib.org/slides/qlum17/hernandez.pdf.

Nguyen, H. and M. Bui (2018). *Regularization for Deep Learning: A Taxonomy*.

— (2020). *Improving neural network's performance with Bayesian Optimization*. URL: https://medium.com/@hiepnguyen034/improving-neural-networks-performance-with-bayesian-optimization-efbaa801ad26.

Nielsen, Micheal (2015). *Neural Networks and Deep Learning*. URL: https://neuralnetworksanddeeplearning.com.

Nwankpa CE., Ijomah W. Gachagan A. and S. Marshall (2018). "Activation Functions: Comparison of Trends in Practice and Research for Deep Learning". In: *Faculty of Engineering, Univeristy of Strathclyde* 20.

Pan, H. and H. Jiang (2014). *Annealed Gradient Descent for Deep Learning*.

Pienaar, E. (2016). "Non-Linear Diffusion Processes and Applications". PhD dissertation. University of Cape Town.

Pienaar, E.A.D (2019). *Analytics Part 2: Introduction to Neural Network*.

Roy, R (2010). *ML | Stochastic Gradient Descent (SGD)*. URL: https://www.geeksforgeeks.org/ml-stochastic-gradient-descent-sgd/.

Saffy, D. (2021). "Using Neural Networks to Price and Delta-Hedge Options under a Put-Call Parity Constraint". MSc. thesis. University of Cape Town.

Sharma, S. (2017). *Improving neural network's performance with Bayesian Optimization*. URL: https://towardsdatascience.com/activation-functions-neural-networks-1cbd9f8d91d6.

Sherif, N. (2019). *Deep hedging and the end of the Black-Scholes era*. URL: https://www.risk.net/derivatives/6875321/deep-hedging-and-the-end-of-the-black-scholes-era.

Srivastava N., Hinton G. Krizhevsky A. Sutskever I. and R. Salakhutdinov (2014). "Dropout: A Simple Way to Prevent Neural Networks from Overfitting". In.

Wang, Chi-Feng (2019). *The Vanishing Gradient Problem*. URL: https://towardsdatascience.com/the-vanishing-gradient-problem-69bf08b15484.