Artificial Intelligence for Option Pricing

A Robust Alternative for Pricing European-styled Call- and Put Options

Master's thesis in Mathematical Sciences, specialisation in Finance Mathematics

Emil Hietanen

Department of Mathematical Sciences
CHALMERS UNIVERSITY OF TECHNOLOGY | UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2022

MASTER'S THESIS 2022

Artificial Intelligence for Option Pricing

A Robust Alternative for Pricing European-styled Call- and Put Options

Emil Hietanen

UNIVERSITY OF GOTHENBURG

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Mathematical Sciences
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2022

Artificial Intelligence for Option Pricing
A Robust Alternative for Pricing European-styled Call- and Put Options
Emil Hietanen

Supervisor: Annika Lang, Department of Mathematical Sciences
Examiner: David Cohen, Department of Mathematical Sciences

Master's Thesis 2022
Department of Mathematical Sciences
Chalmers University of Technology
University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Gothenburg, Sweden 2022

Artificial Intelligence for Option Pricing
A Robust Alternative for Pricing European-styled Call- and Put Options
Emil Hietanen
Department of Mathematical Sciences
Chalmers University of Technology
University of Gothenburg

Abstract

This thesis addresses the issue of vulnerable underlying assumptions used in option pricing methodology. More precisely; underlying assumptions made on the financial assets and markets make option pricing theory vulnerable to changes in the financial framework. To enhance the robustness of option pricing, an alternative approach using artificial intelligence is introduced.
Artificial intelligence is an advantageous tool for pricing financial assets and instruments, in particular; the use of deep neural networks as one does not have to make any assumptions. Instead, the neural network learns the underlying patterns of the asset and market directly from the input data.
To test the proposed pricing alternative, an error metrical analysis, a log-returns distribution fit, and a volatility-smile fit is performed. Four mathematical option pricing models are used as reference models; Black–Scholes, Merton jump-diffusion model, Heston stochastic volatility model and Bates stochastic volatility with jumps. In addition, three types of neural networks are used; multilayer perceptron (MLP), long short-term memory (LSTM), and convolutional neural network (CNN).
All methods included in the thesis require some predefined set of parameters, therefore, a parameter calibration method is required. A non-linear least square method can be used for cases where the number of combinations is sufficiently small. However, as the possible number of parameter combinations increases, the method becomes too computationally heavy. To combat this, an evolutionary reinforcement machine learning algorithm is introduced to find a set of calibrated parameters in a more efficient approach.
First versions of option pricing neural networks show great promise, with significantly better results than the reference models. In addition, the networks show good coherence to existing stylized facts of options, in terms of the empirical frequency distribution of log-returns and volatility smile fit.

Keywords: Options, calls, puts, pricing, artificial neural networks, models, volatility, comparison.

Acknowledgements

I wish to express my gratitude towards my supervisor Annika Lang for the support and help during this thesis. Thank you for answering all my questions and for always being just an email away. Also, many thanks to my family and friends for always being there, supporting me, and cheering me on no matter what. You truly mean a lot.

Emil Hietanen, Gothenburg, 2022

List of Acronyms

Below is the list of acronyms that are used throughout this thesis listed in alphabetical order:

AI	Artificial Intelligence
BP	Backward Propagation
CNN	Convolutional Neural Network
GA	Genetic Algorithm
IV	Implied Volatility
LSTM	Long Short-Term Memory
MLP	Multilayer Perceptron
MSE	Mean Squared Error
NN	Neural Network
RNN	Recurrent Neural Network
WTI	West Texas Intermediate
FP	Forward Propagation

Contents

List of Figures

List of Tables

1 Introduction

This first section begins with a background description that explains the problems with existing option pricing models. Thereafter the aim is stated which specifies the intended outcome, and, lastly, the limitations of the thesis are declared.

1.1 Background

At the beginning of the 20th century, the first option pricing model was introduced by Louis Bachelier, which laid the foundation for using mathematics when pricing options. Since the first introduction, there have been several advances in the field, with Black–Scholes being one of the most prominent models. However, there are several problems concerning the assumptions of the newly introduced model. For example, the model assumes constant values for the risk-free rate of return and volatility over the option duration, which is not the case in the real world. Another issue was recognized in the middle of 2020, where the WTI oil futures dropped below zero [34]. This was problematic for oil options since one of the assumptions is that the price of the underlying asset follows a geometric Brownian motion, which is non-negative. Thus, contradicting the assumption and making the option pricing models obsolete. This caused huge problems since firms and investors could not correctly perform risk assessments. Instead, investors and firms had to switch to the Bachelier model, which does not assume that the underlying assets are non-negative.

There are potential problems with most of the existing mathematical option pricing models, i.e., underlying assumptions incoherent to real-world market behavior [5, 11, 44, 47, 61]. The threat to the validity of the option pricing models can be exaggerated through history. When Bachelier introduced the option pricing model, many investors and academians were skeptical about modeling the stock dynamics with a Brownian motion, which allows for underlying negative values [61]. Thus, when the Black–Scholes model was introduced, the majority was positive towards the change for not allowing negative values for the underlying assets, as negative values were seen as nonsensical. As described in the previous paragraph, the price of an underlying asset went negative. Thus, we can conclude that there is always a threat to the validity when making assumptions, even though the alternative might seem nonsensical.

As there could be an issue with the robustness of making an assumption on the financial market, it seems logical to use as few assumptions as possible. This is where the advances in the field of artificial intelligence could be useful – more specifically, deep neural networks – which make no underlying assumption. Instead, the networks use historical input data to learn the structure of the data. Thus, such a network could theoretically learn the statistical properties of the financial instrument, asset, and market [30].

1.2 Aim

This master's thesis aims to investigate possible AI alternatives for pricing options. More specifically, we can divide the purpose into three parts:

i. Calibrate the set of parameters for each model and neural network.
ii. Evaluate a metric error analysis for the performance of the AI alternatives with mathematical option pricing models as reference.
iii. Investigate how the different methods fit two common stylized facts of option prices; the frequency distribution of log-returns have heavy tails and the option prices form a volatility-smile [1].

1.3 Limitations

The exploration of possible neural network structures is limited to investigate three different neural network architectures:

i. multilayer perceptron,
ii. long short-term memory,
iii. convolutional neural network.

Furthermore, the number of possible reference option pricing models is limited to four:

i. Black–Scholes model,
ii. Merton (jump-diffusion) model,
iii. Heston (stochastic volatility) model,
iv. Bates (stochastic volatility with jumps) model.

1.4 Thesis outline

Section 2 initially introduces the financial framework used in the thesis. Thereafter, the section resumes by presenting an overview of each included option pricing model.

Section 3 gives a theoretical foundation of the included neural network architectures.

Section 4 briefly introduces the two selected parameter calibration methods; non-linear least squares and genetic algorithm.

[1]A graph shape, where the forecasted standard deviation of a sequence of returns, known as implied volatility, against the strike prices creates a convex-shaped graph, refereed to as a smile.

Section 5 presents the data gathered and how the data is processed.

Section 6 presents the error metrical and stylized coherence results of each option pricing approach.

Section 7 confers the result.

section 8 presents potential future work.

2
Financial Foundation

This chapter presents the underlying financial foundation of the thesis. In the first section, the financial framework used in the thesis is stated. In section 2.2, the included reference mathematical option pricing models are introduced together with a brief theoretical explanation.

2.1 Financial Framework

Let's consider a filtered probability space $(\Omega, \mathcal{F}, \{\mathcal{F}_t\}_{0\leq t\leq T}, \mathbb{P})$, where $\{\mathcal{F}_t\}_{0\leq t\leq T}$ is a filtration which satisfies $\mathcal{F}_T = \mathcal{F}$, while $T \in \mathbb{R}^+$ represents the maturity of all investments, and $\mathbb{P}$ represents a real-world probability measure. We assume that there are two types of assets in the market: a risk-free asset and a risky asset whose related prices are respectively modeled by the following $\mathcal{F}_t$-adapted stochastic processes: $\{B_t\}_{0\leq t\leq T}$, and $\{S_t\}_{0\leq t\leq T}$. The risk-free asset guarantees a pre-specified future interest rate, which models the money market account. Let's consider a stock to be the underlying risky asset of a European style option, which is a type of contract that gives its holder the right, but not the obligation, to buy, or sell, one unit of the underlying stock for a predetermined strike price $K > 0$, at maturity time T. The payoff of such an option that gives the holder the right to buy one unit, referred to as a call option, is defined by

$$(S_T - K)^+ = \begin{cases} S_T - K & \text{if } S_T > K, \\ 0 & \text{if } S_T \leq K. \end{cases}$$

Suppose that $S_T > K$, the holder will exercise the option and make a profit equal to $S_T - K$, buying the stock for K and selling it immediately at the market price S_T. If $S_T \leq K$, the option is not exercised.

The payoff of a European option that gives the holder the right to sell one unit, referred to as a put option, is defined by

$$(K - S_T)^+ = \begin{cases} K - S_T & \text{if } S_T < K, \\ 0 & \text{if } S_T \geq K. \end{cases}$$

Suppose that $S_T < K$, the holder will exercise the option and make a profit equal to $K - S_T$, selling the stock for K and buying it immediately at the market price S_T. If $S_T \geq K$, the option is not exercised.

Following [20] and [41], we make various assumptions concerning the financial market characteristics. More specifically,

- the risk-free interest rate $r > 0$ is known and constant through time;
- the stock does not pay dividends;
- frictionless markets: trading occurs continuously in time, with no restrictions on borrowing and short selling and with no transaction costs nor taxes;
- there are no risk-free arbitrage opportunities. Shortly, it means that there is no way of trading on the stock so that an agent has a probability equal to one to gain a strictly positive quantity of money.

We will consider four different ways to model the asset dynamics, each of which leads to various pricing formulas for the same European option.

Remark 2.1.1 None of the assumptions made are used by any of the NNs considered in this thesis.

2.2 Mathematical Option Pricing Models

2.2.1 Black–Scholes

In the early 1970s, Fischer Black and Myron Scholes developed a mathematical model, the Black–Scholes (BS) model, to treat specific financial quantities, publishing related results in the article "The Pricing of Options and Corporate Liabilities" [20]. The model quickly gained popularity, as the model had an easy-to-use closed-form solution. However, the BS model is founded on the so-called risk-neutral pricing assumption, significantly simplifying the associated derivatives analysis. In particular, in the classical BS model, we assume:

- The volatility[1] σ is assumed to be constant over all times t.
- The returns on the underlying asset are log-normally distributed.
- The returns X_t are assumed to be an infinitesimal random walk with drift.

Given the underlying assumptions of the Black–Scholes model, the stock price dynamics is given by

$$dS_t = \mu S_t \mathrm{d}t + \sigma S_t \mathrm{d}W_t, \quad 0 < t \leq T, \quad S(0) = S_0 > 0, \tag{2.1}$$

where μ is the expected infinitesimal rate of return on the stock, σ is the volatility and S_t is the price of the asset at time t and $\{W_t\}_{0\leq t\leq T}$ is a Brownian motion under $\mathbb{P}$, see [25], [33] and [52] for further details related to stochastic calculus (e.g. Brownian motion, Itô calculus, etc.). Now, using the Itô's lemma, which appears as lemma 4.4.4 in [52], one obtains the exact solution

$$S_t = S_0 \exp\left\{(\mu - \frac{\sigma^2}{2})t + \sigma W_t\right\}, \quad 0 \leq t \leq T. \tag{2.2}$$

[1]For any asset that evolves randomly with time, volatility is defined as the standard deviation of a sequence of random variables, where each of these random variables is the return of the asset over some corresponding sequence of times.

We recall that this is the definition of a geometric Brownian motion (gBm) [25]. Moreover, since the distribution of a gBm at time t is log-normal, for $0 \leq t \leq T$, we have

$$\begin{aligned} \log(S_t) &= \log(S_0) + \sigma W_t + (\mu - \frac{\sigma^2}{2})t \\ &\sim \mathcal{N}\left(\log(S_0) + (\mu - \frac{\sigma^2}{2})t, \sigma^2 t\right), \end{aligned} \tag{2.3}$$

where $\mathcal{N}$ denotes the normal distribution. Then, the log-returns are given by

$$X_t = \log(S_t) - \log(S_0) = \sigma W_t + (\mu - \frac{\sigma^2}{2})t \sim \mathcal{N}\left((\mu - \frac{\sigma^2}{2})t, \sigma^2 t\right). \tag{2.4}$$

Given this framework, it is well known that the market model is complete, see, e.g., [52]. Mathematically, a complete market means that any contingent claim can be replicated as a stochastic integral of a sequence of semi-martingales. Indeed, guaranteeing the existence and uniqueness of the martingale measure $\mathbb{Q}$, which makes it possible to perfectly hedge a short position, see, e.g., [52]. Thus, the given martingale dynamic gives us

$$\begin{cases} S_0 = S_0 > 0, \\ dS_t = rS_t\,dt + \sigma S_t\,dW_t^{\mathbb{Q}}, \end{cases} \tag{2.5}$$

where

$$W_t^{\mathbb{Q}} := W_t + \frac{\mu - r}{\sigma}t,\ 0 \leq t \leq T,$$

is a $\mathbb{Q}$-Brownian motion, by the Girsanov theorem, which appears as theorem 5.1 in [25].

The risk-neutral pricing formula, which expresses the European call option price at time t with underlying price x , is given by

$$C(t,x) = e^{-r\tau}\tilde{\mathbb{E}}[(S_T - K)^+ | S_t = x], \quad \tau = T - t, \tag{2.6}$$

where $\tilde{\mathbb{E}}$ denotes the expectation under $\mathbb{Q}$. Equation (2.6) can then be simplified to gives us the Black–Scholes formula, see, e.g., [52]. In particular, indicating with C_{BS} the call price under the Black–Scholes approach, we have

$$C_{BS}(t,x) = x\Phi(d_+(\tau,x)) - e^{-r\tau}K\Phi(d_-(\tau,x)), \tag{2.7}$$

where Φ is the cumulative distribution function for the standard normal random variable, namely

$$\Phi(z) = \frac{1}{\sqrt{2\pi}}\int_{-\infty}^{z} e^{-\frac{y^2}{2}}dy,$$

and

$$d_{\pm}(\tau,x) := \frac{1}{\sigma\sqrt{\tau}}\left\{\log\left(\frac{x}{K}\right) + \left(r \pm \frac{\sigma^2}{2}\right)\tau\right\}.$$

The pricing formula for put options follows directly from put–call parity with discount factor $D_t = e^{-r(T-t)}$.

Theorem 2.2.1 (Put–call parity for European styled options [26]) *If the market gives no risk-free arbitrage opportunities, then the relationship between a put and a call is given by*

$$C(t,x) - P(t,x) = S_t - K \cdot D_t. \tag{2.8}$$

Thus, using (2.8) we obtain the BS put formula

$$P_{BS}(t,x) = Ke^{-r(T-t)} - S_t + C_{BS}(t,x). \tag{2.9}$$

2.2.2 Merton Jump-Diffusion

After the Black–Scholes publication, the model was met with skepticism towards the assumptions of the model and the actual financial markets. As of today, there is a consensus among empirical studies that volatility changes through time, exhibit volatility clustering, and most often returns are not log-normally distributed [5, 11, 44, 47, 61]. These stylized facts were not considered in the original BS model. However, in 1976 Robert C. Merton published a new model to capture one of these facts with the use of jumps [41].

Remark 2.2.1 Parameters that are not specifically defined share the same definitions as previous models.

The Merton model makes the same assumptions on the underlying asset as the BS model, except for log-normally distributed returns.

We shall use the notation $t-$ to indicate a time right before the index in question. In other words, when writing S_{t-} we mean the same as

$$\lim_{s \to t-} S_s,$$

i.e., the left limit.

Let us introduce what follows the Merton approach, as in the original specification of the model [41], starting with the stock price dynamics

$$\begin{cases} S_0 = S_0 > 0, \\ dS_t = (\mu - \lambda k)S_t \, \mathrm{d}t + \sigma S_t \, \mathrm{d}W_t + S_{t-}Q_t, \ 0 < t \le T, \end{cases} \tag{2.10}$$

where λ is the intensity of the Poisson process, μ is the yield, k is the mean random percentage jump size conditional on the jump occurring, $\{W_t\}_{0 \le t \le T}$ is a Brownian motion, and $\{Q_t\}_{0 \le t \le T}$ is a compound Poisson process

$$Q_t = \sum_{i=1}^{N_t} (Y_i - 1), \tag{2.11}$$

where

$$Y_i = \frac{S_{T_i}}{S_{T_{i-}}} > 0, \ i \in \mathbb{N}, \tag{2.12}$$

is the price ratio associated with the i-th jump of the stock price at random time $T_i > 0$. Moreover, we assume that the random variables $\{Y_i\}_{i\in\mathbb{N}}$ are independent and identically distributed (i.i.d.) and that they are also independent of both W_t and N_t, where N_t is a Poisson process with intensity $\lambda > 0$. We also assume that

$$V_i := \log Y_i \sim \mathcal{N}(m, \delta^2),$$

with probability density

$$f_V(y) = \frac{1}{\delta\sqrt{2\pi}} e^{-\frac{(y-m)^2}{2\delta^2}},\ y \in \mathbb{R}.$$

Hence,

$$\mathbb{E}[Q_t] = \lambda k t,\ 0 \leq t \leq T,$$

where

$$k := \mathbb{E}[Y_i - 1] = e^{m+\frac{\delta^2}{2}} - 1,$$

independent of i and therefore well defined. This gives us that equation (2.10) with the compensated version of Q_t, by definition is a martingale [41]. As in the BS model we use this fact together with the risk-neutral pricing formula, to get the explicit solution

$$S_t = S_0 \exp\{\sigma W_t + \left(\mu - \lambda k - \frac{1}{2}\sigma^2\right) t\} \prod_{i=1}^{N_t} Y_i, \quad 0 \leq t \leq T. \tag{2.13}$$

Let the log-returns for the Merton model be conditional on the event $\{N_t = j\}$, then

$$\begin{aligned}\log(S_t) - \log(S_0) &= \sigma W_t + \left(\mu - \lambda k - \frac{1}{2}\sigma^2\right) t + \sum_{i=1}^{j} V_i \\ &\sim \mathcal{N}\left((\mu - \lambda k - \frac{1}{2}\sigma^2)t + jm, \sigma^2 t + j\delta^2\right),\end{aligned} \tag{2.14}$$

which gives us its probability density as a quickly converging series. Indeed, for an arbitrary $A \subseteq \mathbb{R}$,

$$\mathbb{P}\left(\log\left(\frac{S_t}{S_0}\right) \in A\right) = \sum_{j=0}^{\infty} \mathbb{P}\left(\log\left(\frac{S_t}{S_0}\right) \in A | N_t = j\right) \mathbb{P}(N_t = j),$$

with the related probability density at time t as follow

$$\phi_t(y) = e^{-\lambda t} \sum_{j=0}^{\infty} \frac{(\lambda t)^j \exp\left\{-\frac{(y-(\mu-\lambda k-\frac{1}{2}\sigma^2)t-jm)^2}{2(\sigma^2 t+j\delta^2)}\right\}}{j!\sqrt{2\pi((\sigma^2 t + j\delta^2))}},\ y \in \mathbb{R}, \tag{2.15}$$

expressed as a weighted sum of normal densities.

Contrary to the BS model, such a jump-diffusion model is not complete, in which the set of probability measures under which the discounted stock price is a martingale is infinite. Hence, we can choose different martingale measures, $\mathbb{Q} \sim \mathbb{P}$, such that the discounted price $e^{-rt}S_t$ is a martingale [52]. In particular, we can with direct computation get another martingale for the discounted price $e^{-rt}S_t$ as

$$\begin{aligned}\mathrm{d}(e^{-rt}S_t) &= e^{-rt}(\mu - r - \lambda k)S_t\,\mathrm{d}t + e_t^{-rt}(\mathrm{d}W_t^{\mathbb{Q}} - \theta\,\mathrm{d}t) + e^{-rt}S_{t-}\,\mathrm{d}Q_t \\ &= e^{-rt}(\mu - r - \lambda k - \sigma\theta + \lambda^{\mathbb{Q}}k^{\mathbb{Q}})S_t\,\mathrm{d}t \\ &\quad + e^{-rt}\sigma S_t\,\mathrm{d}W_t^{\mathbb{Q}} + e^{-rt}S_{t-}\mathrm{d}(Q_t - \lambda^{\mathbb{Q}}k^{\mathbb{Q}}t), \qquad (2.16)\end{aligned}$$

and setting the differential of (2.16) equal to zero, we obtain the market price of the risk equation

$$\mu - r - \lambda k - \sigma\theta + \lambda^{\mathbb{Q}}k^{\mathbb{Q}} = 0, \qquad (2.17)$$

where θ is such that

$$W_t^{\mathbb{Q}} := W_t + \theta t,\ 0 \leq t \leq T,$$

is a $\mathbb{Q}$-Brownian motion, by the Girsanov theorem, see theorem 5.1 in [25], while $\lambda^{\mathbb{Q}} > 0$ is the new arrival rate of jumps and $k := \tilde{\mathbb{E}}[Y_i - 1]$. Merton proposed the following choice for the change of measure:

$$\begin{aligned}\lambda^{\mathbb{Q}} &= \lambda, \\ f_V^{\mathbb{Q}}(y) &= f_V(y), \\ k^{\mathbb{Q}} &= k, \\ \theta &= \frac{\mu - r}{\sigma}.\end{aligned}$$

Merton justified leaving the jump segment unchanged, with the assumption that the jump risk is diversifiable and no risk premium is attached to it. Thus, the risk-neutral properties of the jump segment of the process S_t are assumed to be the same as its statistical properties. In particular, the dynamics of the stock price under $\mathbb{Q}$ are then

$$S_t = S_0\mathrm{exp}\left\{\sigma W_t^{\mathbb{Q}} + (r - \lambda k - \frac{1}{2}\sigma^2)t\right\}\prod_{i=1}^{N_t} Y_i, \quad 0 \leq t \leq T. \qquad (2.18)$$

We can rewrite the European call price formula, assuming that the jumps are of Gaussian type length [41], as

$$\begin{aligned}C_M(\tau, x) &= \sum_{j=0}^{\infty} e^{-\lambda\tau}\frac{(\lambda\tau)^j}{j!}\tilde{\mathbb{E}}[e^{-r\tau}(S_T - K)^+|S_t = x] \\ &= \sum_{j=0}^{\infty} e^{\lambda\tau}\frac{(\lambda\tau)^j}{j!}C_{BS}(\tau, x), \qquad (2.19)\end{aligned}$$

where

$$\sigma_j^2 = \sigma^2 + \frac{j\delta^2}{\tau} \text{ and } x_j = x\mathrm{exp}\left\{jm + \frac{j\delta^2}{2} - \lambda\tau e^{m+\frac{\delta^2}{2}} + \tau\right\}$$

are used in $C_{BS}(\tau, x)$.

Remark 2.2.2 *If $\lambda = 0$ then $C_M(\tau, x) = C_{BS}(\tau, x)$, with all the terms in the sum equal to 0, except for $j = 0$, when $x_0 = x$ and $\sigma_0 = \sigma$.*

2.2.3 Heston

In 1993, Steven Heston proposed a model with stochastic volatility, where an asset's price and volatility follow a Brownian motion process [47]. Thus, the model introduced by Heston could incorporate the three known stylized facts presented in the previous section: volatility is not constant through time, returns are non-normally distributed, and volatility clusters.

Let us introduce what follows the Heston, or stochastic volatility, approach, starting with the stock price dynamics

$$\begin{aligned} \mathrm{d}S_t &= rS_t\,\mathrm{d}t + \sqrt{V_t}S_t\,\mathrm{d}W_{1t}, \\ \mathrm{d}V_t &= \kappa(\theta - V_t)\mathrm{d}t + \sigma\sqrt{V_t}\,\mathrm{d}W_{2t}, \end{aligned} \tag{2.20}$$

where the newly introduced parameters are defined as

- $\sqrt{V_t}$: volatility of the volatility of the asset price;
- θ: long-term price variance;
- κ: rate of reversion to the long-term mean price variance;
- W_{1t}: Brownian motion of the asset price;
- W_{2t} : Brownian motion of the asset's price variance;
- ρ: correlation coefficient for W_{1t} and W_{2t}.

For the last two option pricing models (Heston and Bates), we will use the Lewis framework, see [4], which takes advantage of the fact that each probability density function (PDF) can be described using a characteristic function. Now, before we consider such methods, it is worth mentioning why we would consider such an approach. First, recall from Section 2.1 that the price of a European call option on a single asset S_T can be written as:

$$C = \mathbb{E}[(S_T - K)^+]. \tag{2.21}$$

As (2.21) is an expectation, it can be calculated via integration, given that we know the PDF in closed-form (as with the BS model). However, for multiple models, the PDF is either unknown in closed-form or quite unmanageable. Then, its characteristic function is often much easier to work with, which is the case for Heston and Bates.

Definition 2.2.1 *A characteristic function for a random variable X is defined by*

$$\phi_X(u) = \mathbb{E}(e^{iuX})$$

for arbitrary real numbers u, where $i = \sqrt{-1}$.

If $f_X(x)$ is the PDF of the random variable then the integral

$$\phi_X(u) = \int_{-\infty}^{\infty} e^{iux} f_X(x)\,\mathrm{d}x, \tag{2.22}$$

is the expected value and is by definition, see Section 2.1 in [45], the Fourier transform of the density function $f_X(x)$ denoted by $\mathbb{F}[f_X(x)]$. For a given u, $\phi_X(u)$ is a single random variable and for $-\infty < u < \infty$ we have a stochastic process.

We know that Itô's lemma in the three variable case for a general stochastic differential equation (SDE) involving a time dependent stochastic process of three variables, t, X_t and V_t, is given by the following system of two standard stochastic differential equations

$$\begin{aligned} \mathrm{d}X_t &= \mu_x \,\mathrm{d}t + \sigma_x \,\mathrm{d}W_{1t}, \\ \mathrm{d}V_t &= \mu_v \,\mathrm{d}t + \sigma_v \,\mathrm{d}W_{2t}. \end{aligned} \tag{2.23}$$

Now, let W_{1t} and W_{2t} have correlation ρ, where $-1 \leq \rho \leq 1$. For a given continuous, twice differentiable, scalar function $g(X_t, V_t, t)$, we want to find the derivative. With the multi-variable Taylor series expansion of 2nd order and omitting the higher order terms, we get that the derivative of a function of three variables involving two stochastic processes equals the following expression:

$$\begin{aligned} \mathrm{d}g(X_t, V_t, t) = &\left[\mu_x g_x + \mu_v g_v + g_t + g_{xv}\sigma_x\sigma_v\rho + \frac{1}{2}(g_{xx}\sigma_x^2 + g_v v\sigma_v^2)\right] \mathrm{d}t \\ &+ [\sigma_x g_x]\,\mathrm{d}W_{1t} + [\sigma_v g_v]\,\mathrm{d}W_{2t}. \end{aligned}$$

To get a sense of why higher order terms can be omitted, see the proof of Itô's lemma, which appears as lemma 4.4.4 in [52].

The characteristic function of the Heston model is a function of S_t, V_t, and t, where Itô's lemma can be used to get the derivative of the characteristic function. Also, we know that the characteristic function for a three variable stochastic process has the following exponential affine form [47]:

$$\phi_{S_t,V_t}(u) = \mathbb{E}[e^{A(T-t)+B(T-t)S_t+C(T-t)V_t+iuS_t}].$$

Letting $T - t = \tau$, the explicit form of the Heston model's characteristic function appears below.

$$\phi_{S_t,V_t}(u) = \mathbb{E}[e^{A(\tau)+B(\tau)S_t+C(\tau)V_t+iuS_t}], \tag{2.24}$$

where

$$\begin{aligned} A(\tau) &= riu\tau + \frac{\kappa}{\sigma^2}\left[-(\rho\sigma iu - k - M)\tau - 2\ln(\frac{1 - Ne^{M\tau}}{1 - N})\right], \\ B(\tau) &= 0, \\ C(\tau) &= \frac{(e^{M\tau} - 1)(u\sigma iu - \kappa - M)}{\sigma^2(1 - Ne^{M\tau})}, \\ M &= \sqrt{(\rho\sigma iu - \kappa)^2 + \sigma^2(iu + u^2)}, \\ N &= \frac{\rho\sigma iu - k - M}{\rho\sigma iu - \kappa - M}. \end{aligned}$$

With the known characteristic function, we can use the Lewis methodology. However, we will first need some basic knowledge of Fourier transformation.

Let us introduce the inversion theorem as it is the fundamental theorem of the theory of characteristic functions since it links the characteristic function back to its probability distribution via an inverse Fourier transform.

Definition 2.2.2 (Absolutely Integrable Functions) *A function f is absolutely integrable if the integral of its absolute value over $\mathbb{R}$ is finite, i.e.,*

$$\int_{-\infty}^{\infty} |f(x)|\, dx < \infty. \tag{2.25}$$

Absolutely integrable functions give us an important relation since for a Fourier transform and its inverse to exist, then the condition in equation (2.25) must hold [32].

Another important property of characteristic functions is their one-to-one relationship with distribution functions. In particular, every random variable possesses a unique characteristic function, and the characteristic function indeed characterizes the distribution uniquely [32].

With the fundamental results from Lévy [45], who gave a general inversion formula, Gil-Pelaez [27] developed a useful representation of the Inversion Theorem.

In the following, we use the form given by Gil-Pelaez. See [32] for a review on inversion methods.

Theorem 2.2.2 *For a univariate random variable X, if x is a continuity point of the cumulative distribution function F_X then*

$$F_X(x) = \frac{1}{2} - \frac{1}{2\pi} \int_{-\infty}^{\infty} \frac{[e^{iux}\phi_X(u)]}{iu} du. \tag{2.26}$$

From (2.26) we note that have a distribution function expressed as an integral in terms of the characteristic function. Taking the derivative of $F_X(x)$ yields the probability density function $f_X(x)$

$$f_X(x) = \mathbb{F}^{-1}[\phi_X(u)] = \frac{1}{2\pi} \int_{-\infty}^{\infty} e^{-iu}\phi_X(u) du, \tag{2.27}$$

where $\mathbb{F}^{-1}[\phi_X(u)]$ denotes the inverse Fourier transform of the characteristic function.

Next, some relevant properties of Fourier transforms.

Theorem 2.2.3 (Elementary Properties of the Fourier Transform [46]) *We denote the Fourier transform from $f(x)$ as $\hat{f}(u) = \mathbb{F}[f](u)$*

- ***Linearity.*** *For arbitrary a, b the transform is a linear operator*

$$\mathbb{F}[af(x) + bg(x)](u) = a\hat{f}(u) + b\hat{g}(u). \tag{2.28}$$

- ***Translation.*** *The Fourier transform turns a multiplication by a variable x_0 into translation*

$$\mathbb{F}[f(x - x_0)](u) = e^{iux_0}\hat{f}(u). \tag{2.29}$$

- ***Differentiation.*** *If $n > 0$ is an integer, $f^{(n)}$ is piecewise continuously differentiable, and each derivative is absolutely integrable on the entire real line, then*

$$\mathbb{F}[f^{(n)}](u) = (-iu)^n\hat{f}(u). \tag{2.30}$$

Thus, a differentiation converts to multiplication in Fourier space.

- ***Convolution.*** *Define the convolution by*

$$(f * g)(x) = \int_{-\infty}^{\infty} f(x - y)g(y)\,\mathrm{d}y, \tag{2.31}$$

where $*$ denotes the convolution operator. Then

$$\mathbb{F}[f * g](u) = \hat{f}(u)\hat{g}(u), \tag{2.32}$$

and

$$\mathbb{F}[fg](u) = \hat{f}(u) * \hat{g}(u). \tag{2.33}$$

We have until now defined the characteristic function in terms of real-valued transform variables u. However, we could also integrate a characteristic function along the complex plane $u \to z \in \mathbb{C}$ by a line parallel to the real axis. Then, characteristic function is well-behaved, i.e., Lipschitz continuous [15]. The set of values for $z = z_r + iz_i$ for which the expectation in Eq. (2.22) is well-defined and within some strip of regularity $\mathcal{S}_X$ with $a < \Im[z] < \beta$ parallel to the real $z-$axis.

With the extension to the complex plane, $\phi_T(z)$ is called the generalized Fourier transform, with the inverse of this generalized Fourier transform

$$f(x) = \frac{1}{2\pi}\int_{iz_i-\infty}^{iz_i+\infty} e^{-izx}\phi(z)\,dz. \tag{2.34}$$

Generally, the properties for the ordinary Fourier transform also apply with little or no modification to the generalized Fourier transform.

Theorem 2.2.4 (Plancherel–Parseval identity) *If the function f and g are Lipschitz continuous at z_i, then by an integration along a straight line parallel to the real axis*

$$\int_{-\infty}^{\infty} f(x)\bar{g}(x)\,dx = \frac{1}{2\pi}\int_{-\infty}^{\infty} \mathbb{F}[f(x)](z_r + iz_i)\overline{\mathbb{F}[g(x)]}(z_r + iz_i)\,dz_r. \tag{2.35}$$

For more detailed information on the theory of characteristic functions, see [15], and for a more detailed discussion in the context of Fourier transforms, see [17, 59].

Now, with the brief explanation of Fourier transforms, we introduce the Lewis framework. The idea is to express the option value as the convolution of generalized Fourier transforms and then apply the Plancherel–Parseval identity.

For each option there exists a known payoff function $w(S_T)$. The payoff function for a call option is given by $w(x) = (e^x - K)^+$ with $x = \ln S_T$. Lewis suggested that every payoff function could also be represented in Fourier space $\hat{w}(z) = \int_{-\infty}^{\infty} e^{izx} w(x) dx$. However, $(e^x - K)^+$ is an unbounded function, which does not belong to L^1. Thus, as the defining integral is not finite when $\hat{w}(z) = \mathbb{F}[w(x)] \to \infty$ the Fourier transform does not exist. If we use a modified payoff, we have a regular Fourier transform, since $(e^x - K)^+ e^{-z_i x} \to 0$ as $x \to \infty$ for some appropriate z_i. This approach corresponds integrating a characteristic along the complex plane by a line parallel to the real axis. Hence, the call option payoff yields

$$\begin{aligned}
\hat{w}(z) &= \int_{-\infty}^{\infty} e^{izx} (e^x - K)^+ \, dx \\
&= \int_{\ln K}^{\infty} e^{izx} (e^x - K) \, dx \\
&= \left(\frac{e^{(iz+1)x}}{iz+1} - K \frac{e^{izx}}{iz} \right)_{\ln K}^{x=\infty} \\
&= 0 - \left(\frac{K^{iz+1}}{iz+1} - K \frac{K^{iz}}{iz} \right) \\
&= -\frac{K^{iz+1}}{z^2 - iz},
\end{aligned} \tag{2.36}$$

tit

where z is a complex-valued number. The upper limit $x = \infty$ only exists under the condition that $\Im(z) > 1$ implying that the Fourier transform is Lipschitz continuous only within a certain strip of regularity $\mathcal{S}_w$ in the complex domain. Note that the transformed payoff for a put option has the same functional form, but is defined in a different strip in the complex plane with $z_i < 0$.

If we assume that the characteristic function $\phi_T(z)$ is Lipschitz continuous with $z \in \mathcal{S}_X$ with a transformed payoff $\hat{w}(z)$, $z \in \mathcal{S}_X$, and an arbitrary price dynamic and a we can apply martingale pricing, which gives us the option value

$$\begin{aligned}
V(S_0, K, T) &= e^{-rt} \mathbb{E}^{\mathbb{Q}}[w(x)] \\
&= \frac{e^{-rT}}{2\pi} \mathbb{E}^{\mathbb{Q}} \left[\int_{iz_i-\infty}^{iz_i+\infty} e^{izx} \hat{w}(z) \, dz \right] \\
&= \frac{e^{-rT}}{2\pi} \int_{iz_i-\infty}^{iz_i+\infty} \mathbb{E}^{\mathbb{Q}} \left[e^{izx} \right] \hat{w}(z) \, dz \\
&= \frac{e^{-rT}}{2\pi} \int_{iz_i-\infty}^{iz_i+\infty} \phi_T(-z) \hat{w}(z) \, dz.
\end{aligned} \tag{2.37}$$

Note from above that we have stated that the extended characteristic function $\phi_T(z)$ with $z \in \mathbb{C}$ is Lipschitz continuous in $\mathcal{S}_X$. However, the reflection symmetry property for a real-valued function $\bar{\phi_T}(z) = \phi_T(-\bar{z})$, $\phi_T(-z)$ is also Lipschitz continuous in the conjugate strip of regularity $\bar{\mathcal{S}_X}$, where the real z-axis is the line of symmetry. If we get a $z \in (\bar{\mathcal{S}_x} \cap \mathcal{S}_w)$ then the whole integral would be Lipschitz continuous. In addition, if we take the inversion contour along $\bar{\mathcal{S}_x} \cap \mathcal{S}_w$, then the integral converges absolutely, and we were allowed to change the order of integration by Fubini's theorem, see theorem 2.3.2 in [15].

Using the payoff transform of a call option (2.36), the call price is given by

$$C(S_0, K, T) = -\frac{Ke^{-rt}}{2\pi}\int_{iz_i-\infty}^{iz_i+\infty} e^{-izk}\phi_T(-z)\frac{dz}{z^2 - iz}, \tag{2.38}$$

with $k = \ln\frac{S_0}{K} + rT$ in the phase factor e^{-izk} and $z_i \in \mathcal{S}_V$.

The call price in (2.38) is regular in the strip $\bar{\mathcal{S}_V}$, excluding for when $z^2 - iz$ is zero, which will occur at two poles, at $z = 0$ and $z = i$. For the pole at $z = i$, the residue is given by

$$\text{Res}(i) = \lim_{z\to i}\left[(z - i)\left(-\frac{Ke^{-rT}}{2\pi}e^{-izk}\frac{\phi_T(-z)}{z^2 - iz}\right)\right] = \frac{S\phi_T(-i)i}{2\pi} = \frac{iS}{2\pi}. \tag{2.39}$$

Recall that $z = z_r + i_i$, then if we move integration contour to $z_i \in (0, 1)$, then the valuation function for a call option must equal the integral along $\Im(z) = z_i - 2\pi i \cdot \text{Res}(i) = S$, according to the Cauchy's residue theorem, which appears as theorem 3.2.5 in [32]. Hence, we get

$$C(S_0, K, T) = S_0 - \frac{Ke^{-rT}}{2\pi}\int_{iz_i-\infty}^{iz_i+\infty} e^{izk}\phi_T(-z)\frac{dz}{z^2 - iz}. \tag{2.40}$$

For the put option a similar contour can be used, however, the strip is then defined by $\Im(z) < 0$. By shifting the contour crossing both poles at $z = 0$ and $z = i$ we additionally pick up the residue for $z = 0$ and find that

$$\text{Res}(i) = \lim_{z\to 0}\left[\left(-\frac{Ke^{-rT}}{2\pi}e^{-izk}\frac{\phi_T(-z)}{z^2 - iz}\right)\right] \quad = \frac{e^{-rT}K\phi_T(0)i}{2\pi} = \frac{iKe^{-rT}}{2\pi}. \tag{2.41}$$

With the use of Cauchy's residue theorem, we obtain the formulation for the put option

$$P(S_0, K, T) = Ke^{-rT} - \frac{Ke^{-rT}}{2\pi}\int_{iz_i-\infty}^{iz_i+\infty} e^{-izk}\phi_T(-z)\frac{dz}{z^2 - iz}. \tag{2.42}$$

The introduced framework can now be used with substituting in the Heston characteristic function (2.24) at time T to obtain prices for call and put options.

For further details on the Lewis framework, see [3, 4].

2.2.4 Bates

The Bates model (1996) incorporates both stochastic volatilities similar to the Heston model and jump-diffusion similar to the Merton model [14]. Analogously to the Heston model, the Bates model embodies the three known stylized facts presented in Section 2.2.2.

Let us introduce what follows the Bates, or stochastic volatility with jump-diffusion, approach, starting with the stock price dynamics.

$$dS_t = (r - \lambda k)S_t dt + \sqrt{V_t} S_t dW_{1t} + Q_t S_t, \tag{2.43}$$

$$dV_t = k(\theta - V_t)dt + \sigma\sqrt{V_t} dW_{2t}. \tag{2.44}$$

Similarly to the Heston model, we can use the Lewis framework and use the characteristic function for the Bates model instead of the Heston model.

The characteristic function for the Bates model [14] with $\tau = T - t$ is given by

$$\phi_{S_t,V_t}(u) =$$

$$\exp\left\{(C + D\sigma + iu \ln S_t)\left[\lambda\tau\left((1+r)^{i\phi}\exp\left\{\delta^2(-\frac{1}{2}iu + \frac{(iu)^2}{2})\right\} - 1\right) - \lambda\tau riu\right]\right\},$$

where

$$\begin{aligned} C &= ri\phi\tau + \frac{k\theta}{\sigma^2}\left[(b - \rho\sigma iu - d)\tau - 2\ln\left(\frac{1-\epsilon e^{-d\tau}}{1-\epsilon}\right)\right], \\ D &= \frac{b - \rho\sigma iu - dk}{\sigma^2}\left(\frac{1-e^{-d\tau}}{1-\epsilon e^{-d\tau}}\right), \\ \epsilon &= \frac{b - \rho\sigma iu + d}{b - \rho\sigma iu - d}, \\ d &= \sqrt{(b-\rho\sigma iu)^2 - \sigma^2(-iu - u^2)}, \\ b &= k + \lambda - \rho\sigma. \end{aligned}$$

3
Neural Networks

This section consists of a theoretical foundation for understanding the selected neural network (NN) architectures. Furthermore, as each network is different from the other, a theoretical subsection for each network is included. However, to get an adequate theoretical foundation, the section initially covers the perceptron before reviewing the selected networks.

3.1 Perceptron

The perceptron [21], also referred to as a McCulloch–Pitts neuron, is the first and simplest NN model, making it a natural starting point. Furthermore, extending this model with multiple neurons will result in the so-called Rosenblatt's single-layer perceptron, used for the classification of linearly separable patterns [21].

3.1.1 One-neuron perceptron

For a one-neuron perceptron [21], the network topology is shown in Fig. 3.2, and the net input to the neuron is given by

$$net = \sum_{i=1}^{n} w_i x_i - \theta = \mathbf{w}^T \mathbf{x} - \theta, \tag{3.1}$$

$$\hat{y} = g(net), \tag{3.2}$$

where $\mathbf{x} = [x_1, x_2, ..., x_n]^T$ are the inputs, $\mathbf{w} = [w_1, .., w_n]^T$ are the weights, n are the number of inputs, θ are the bias and g is the activation function.

The one-neuron perceptron with an activation function can be used for the classification of a vector $\mathbf{x}$ into two classes. Moreover, the classification process is determined by the decision areas, separated by a hyperplane,

$$\mathbf{w}^T \mathbf{x} - \theta = 0,$$

where the threshold $\theta \in \mathbb{R}$ is a parameter used to shift the decision boundary away from the origin.

The activation functions are used as functions that help the network adapt to complex patterns in the data. The activation function is attached to each neuron in the network and determines whether it should be activated or not. This decision is made based on whether the input for each neuron is relevant for the model prediction [9]. The simplest form of an activation function is a linear function, but there are also non-linear activation functions. The two most popular ones are [55]:

rectified linear unit (ReLU)

$$g(z) = \max(0, z), \; g'(z) = \begin{cases} 1, & z > 0, \\ 0, & z \leq 0, \end{cases} \tag{3.3}$$

and the sigmoid

$$g(z) = (1 + e^{-z})^{-1}, \; g'(z) = g(z)(1 - g(z)).$$

The reason one might use a non-linear activation function is that they introduce non-linearity into the network. Therefore, if one is working with non-linear data it is necessary to use a non-linear function, and this is the case no matter the depth of the network [9]. We can illustrate the problem of trying to classify non-linear data with a linear activation function through a graphical example in Fig. 3.1. From the figure, we note that the data are not classifiable by a linear activation function, but can be done with a non-linear activation function.

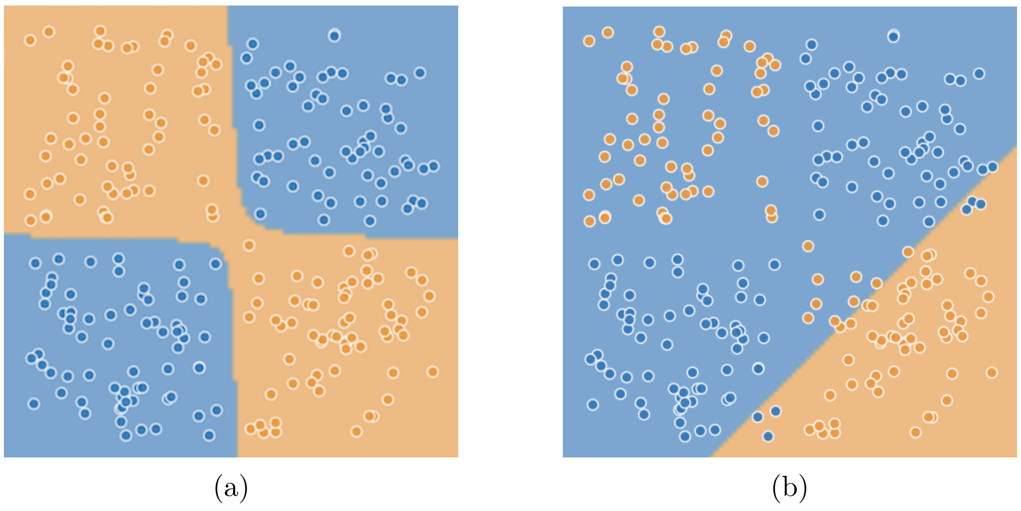

Figure 3.1: Comparison of classifying positives (blue) and negatives (orange) by a hyperplane, based on (a) a non-linear activation (ReLU) function or (b) a linear activation function.

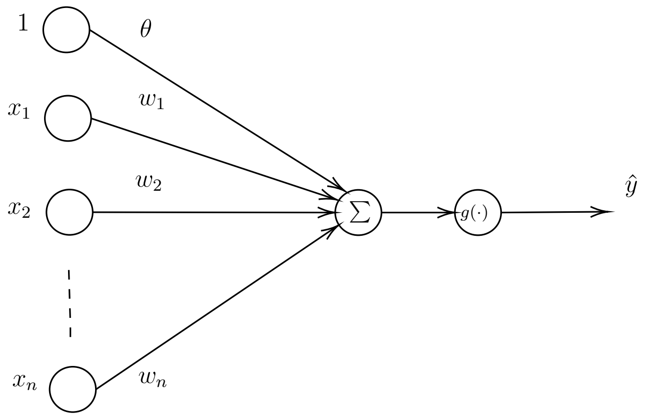

Figure 3.2: Network topology of the McCulloch-Pitts neuron with one output.

3.1.2 Single-layer perceptron

When more neurons are added to the perceptron network from the previous section, we obtain a single-layer perceptron, the network topology is shown in Fig. 3.3.

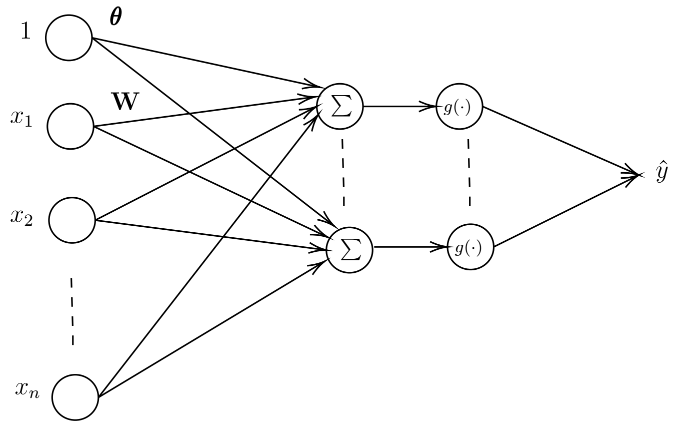

Figure 3.3: Rosenblatt's single-layer perceptron with one output.

Compared to the one-neuron perceptron, the single-layer perceptron can be used to classify input vector $\mathbf{x}$ into more than two classes [21]. For a perceptron with n inputs and m neurons, the state is updated by

$$\mathbf{net} = \mathbf{W}^T\mathbf{x} - \boldsymbol{\theta}, \tag{3.4}$$

$$\hat{\mathbf{y}} = g(\mathbf{net}), \tag{3.5}$$

where $\mathbf{W}$ is the weights matrix, $\mathbf{net} = (net_1, ..., net_m)^T$ is the net input vector, $\hat{\mathbf{y}} = (\hat{y}_1, ..., \hat{y}_m)^T$ is the output vector, and $\boldsymbol{\theta} = (\theta_1, ..., \theta_m)^T$ is the biases in the second layer. The adaptation of $\mathbf{W}$ is error driven, and this process is known as a learning algorithm. To learn the basics of a learning algorithm, we shall look closer at one specific algorithm, Rosenblatt's perceptron learning algorithm [21, 22].

3.1.2.1 Perceptron learning algorithm

In what follows, we introduce Rosenblatt's perceptron convergence theorem for classification problems [21]. Let us start with defining two key concepts, linear separability and gradient descent.

Definition 3.1.1 (Linear separability) *A data set is said to be* linear separable *if there exists a weight vector $\boldsymbol{x}^*$, and a bias term θ^*, such that all features $\boldsymbol{x}$ in the training data X have predicted sign equal to their true class label l [21].*

Definition 3.1.2 (Gradient descent method) *The weights are updated by*

$$\Delta w_{jk} = \alpha \frac{\partial E_r}{\partial w_{j,k}} = \alpha \sum_{i=1}^{N} \rho(\epsilon_i, \beta) \frac{\partial \epsilon_i}{\partial w_{j,k}}, \tag{3.6}$$

where $\alpha \in \mathbb{R}^+$ is the learning rate, $\beta \in \mathbb{R}^+$ is the cutoff parameter, ϵ_i is the estimated error for input i, i.e., the network residuals and $\rho : \mathbb{R} \times \mathbb{R}^+ \to \mathbb{R}$ is the influence function, given by

$$\rho(\epsilon_i, \beta) = \frac{\partial \sigma(\epsilon_i, \beta)}{\partial \epsilon_i}, \tag{3.7}$$

where σ is the loss function, which is a symmetric function with a unique minimum at zero. The loss function represents a loss associated with an event, for example a squared error loss function [9].

Theorem 3.1.1 (Perceptron convergence) *Given a one-neuron perceptron and input $\boldsymbol{x} \in X$ from two linear separable classes. Let the patterns be presented in an arbitrary sequence of update cycles, also known as epochs. Then, starting from an arbitrary initial state, the perceptron learning procedure always converges and yields a decision hyperplane between the two classes in finite time .*

For a proof we refer readers to [21, 22].

The theorem states that the weight of the perceptron will converge to a fixed point within a finite number of epochs for a separable data set. Therefore, we can use the theorem to understand the learning algorithm, which is used in the proof of the theorem.

Remark 3.1.1 *The perceptron convergence theorem can be extended to the single-layer perceptron by extending the perceptron learning algorithm from one neuron to multiple neurons [13, 38].*

The perceptron learning algorithm is given as

$$net_{t,j} = \sum_{i=1}^{J_1} x_{t,i} w_{ij}(t) - \theta_j = \mathbf{w}_j^T \mathbf{x}_t - \theta_j, \tag{3.8}$$

$$\hat{y}_{t,j} = \begin{cases} 1, & net_{t,j} > 0, \\ 0, & net_{t,j} \leq 0, \end{cases} \tag{3.9}$$

$$e_{t,j} = y_{t,j} - \hat{y}_{t,j}, \tag{3.10}$$

$$w_{ij}(t+1) = w_{ij}(t) + \alpha x_{t,i} e_{t,j}, \tag{3.11}$$

where J_1 is the number of neurons in layer 1, $i = 1, 2, ..., n$, is the number of inputs, $j = 1, 2, ..., m$ are the neurons, and α is the learning rate. All weights are initialized randomly and continuously updated until the errors, given by equation (3.10), are sufficiently small. Thus, one could interpret the equations (3.8)-(3.10) as a loop that breaks for a given level of error, $e_{t,j}$.

The selection of learning rate, α, does not affect the stability of perceptron learning, meaning, the learning algorithm will converge, independent of the learning rate. Instead, the learning rate only affects the convergence speed of the nonzero initial weight vector [21, 22].

3.2 Multilayer perceptrons

Multilayer perceptrons (MLPs) are feedforward networks with one or more layers of units between the input and output layers. The network topology is illustrated in Fig. 3.4, where M is the number of layers.

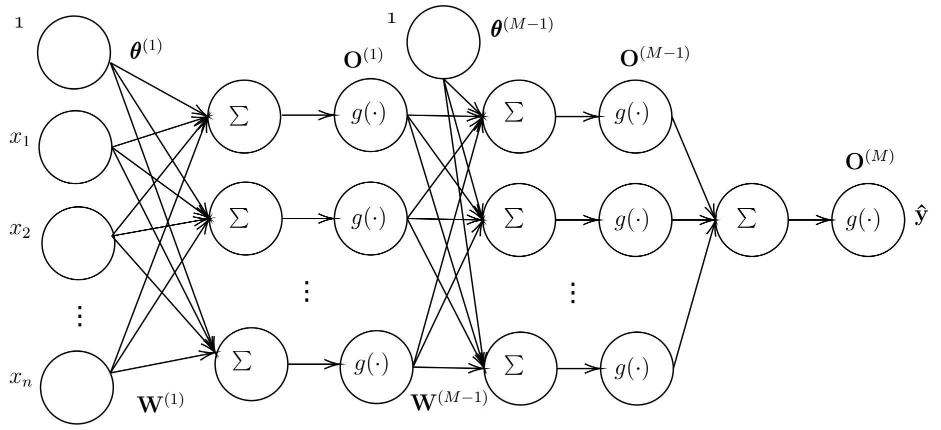

Figure 3.4: Multilayer perceptrons with one output.

For $m = 2, ..., M$ and the pth node we have the following relations:

$$\hat{\mathbf{y}}_p = \mathbf{O}_p^{(M)}, \quad \mathbf{O}_p^{(1)} = \mathbf{x}_p, \tag{3.12}$$

$$\mathbf{net}_p^{(m)} = \left[\mathbf{W}^{(m-1)}\right]^T \mathbf{O}_p^{(m-1)} - \boldsymbol{\theta}^{(m)}, \tag{3.13}$$

$$\mathbf{O}_p^{(m)} = g^{(m)}(\mathbf{net}_p^{(m)}), \tag{3.14}$$

where $\mathbf{O}^{(m)}$ is an output vector for each layer.

It has been shown that a three-layer sigmoid activated MLP with an arbitrary number of neurons is a universal approximator, i.e., the MLP can approximate any continuous multivariate functions to any accuracy [24].

3.2.1 Backpropagation learning algorithm

Backpropagation (BP) learning is the most popular learning rule for performing supervised learning tasks [9]. It is not only used to train feedforward networks such as MLP but also is adapted to recurrent neural networks (RNNs). Moreover, the perceptron convergence theorem (theorem 3.1.1) can be extended for MLPs, stating that the BP algorithm converges to an optimal solution for linearly separable input data with no upper bound on the learning rate [13, 38].

The BP algorithm propagates backward the error between the desired signal and the network output through the network. After providing an input pattern, the output of the network is then compared with a given target pattern, and the error of each output unit is calculated. This error signal is propagated backward, and a closed-loop control system is thus established. The weights can be adjusted by the use of a gradient descent algorithm.

To implement the BP algorithm, a continuous, nonlinear, monotonically increasing, differentiable activation function is needed [13]. In the following, we derive the BP algorithm for MLP with a ReLu activation function.

Remark 3.2.1 BP algorithms for other neural network models can be derived similarly [13, 38].

The objective function for optimization, given a sample size N and a training set S, is defined as the mean square error (MSE) between the actual network output $\mathbf{y}_p$ and the desired output $\hat{\mathbf{y}}_p$ for all the training pairs $(\mathbf{x}_p, \mathbf{y}_p) \in S$. Thus, the objection function for the size of the sample set N can be written as

$$E = \frac{1}{N} \sum_{p \in S} E_p = \frac{1}{2N} \sum_{p \in S} ||\hat{\mathbf{y}}_p - \mathbf{y}_p||^2, \tag{3.15}$$

where

$$E_p = \frac{1}{2} ||\hat{\mathbf{y}}_p - \mathbf{y}_p||^2 = \frac{1}{2} e_p^T e_p, \tag{3.16}$$

and

$$e_p = \hat{\mathbf{y}}_p - \mathbf{y}_p. \tag{3.17}$$

Remark 3.2.2 The factor $\frac{1}{2}$ is used in E_p for the convenience of the derivation.

The network parameters $\mathbf{W}^{(m-1)}$ and $\theta^{(m)}$ for layers $m = 2, ..., M$ can be defined as the matrix of the combinations, $\mathbf{W} = [w_{ij}]$. This means that the bias term for the error function represented by (3.15) or (3.16) can be minimized using gradient descent, see definition 3.1.2. Thus, minimizing E_p gives us

$$\Delta_p \mathbf{W} = -\alpha \frac{\partial E_p}{\partial \mathbf{W}} = -\alpha \frac{\partial E_p}{\partial w_{ij}}, \tag{3.18}$$

Using the chain rule on (3.18) gives the expression

$$\frac{\partial E_p}{\partial w_{uv}^{(m)}} = \frac{\partial E_p}{\partial net_{p,v}^{(m+1)}} \frac{\partial net_{p,v}^{(m+1)}}{\partial w_{uv}^{(m)}}, \tag{3.19}$$

where the second factor is derived from (3.13)

$$\frac{\partial net_{p,v}^{(m+1)}}{\partial w_{uv}^{(m)}} = \frac{\partial}{\partial w_{uv}^{(m)}} \left(\sum_{\omega=1}^{J_m} w_{\omega v}^{(m)} O_{p,\omega}^{(m)} + \theta_v^{(m+1)} \right) = O_{p,u}^{(m)}, \tag{3.20}$$

and J_m is the number of neurons in layer m.

The first factor of (3.19) can be derived using the chain rule and (3.14)

$$\begin{aligned} \frac{\partial E_p}{\partial net_{p,v}^{(m+1)}} &= \frac{\partial E_p}{\partial O_{p,v}^{(m+1)}} \frac{\partial O_{p,v}^{(m+1)}}{\partial net_{p,v}^{(m+1)}} \\ &= \frac{\partial E_p}{\partial O_{p,v}^{(m+1)}} g_v^{(m+1)} \left(net_{p,v}^{(m+1)} \right). \end{aligned} \tag{3.21}$$

If we consider two situations for the output unit ($m = M - 1$) and for the hidden unit ($m = 1, ..., M - 2$) we can solve the first factor of (3.21):

$$\frac{\partial E_p}{\partial O_{v,p}^{(m+1)}} = e_{p,v} = y_{p,v} - \hat{y}_{p,v}, \tag{3.22}$$

for $m = M - 1$, and

$$\begin{aligned} \frac{\partial E_p}{\partial O_{p,v}^{(m+1)}} &= \sum_{\omega=1}^{J_m+2} \left(\frac{\partial E_p}{\partial net_{p,\omega}^{(m+2)}} \frac{\partial net_{p,\omega}^{(m+2)}}{\partial O_{p,v}^{(m+1)}} \right) \\ &= \sum_{\omega=1}^{J_m+2} \left[\frac{\partial E_p}{\partial net_{p,\omega}^{(m+2)}} \frac{\partial}{\partial O_{p,v}^{(m+1)}} \left(\sum_{u=1}^{J_m+1} w_{u\omega}^{(m+1)} O_{p,u}^{(m+1)} + \theta_\omega^{(m+2)} \right) \right] \\ &= \sum_{\omega=1}^{J_m+2} \frac{\partial E_p}{\partial net_{p,w}^{m+2}} w_{v\omega}^{m+1}, \end{aligned} \tag{3.23}$$

for $m = 1, ..., M - 2$. Let us define the delta function as

$$\delta_{p,v}^{(m)} = -\frac{\partial E_p}{\partial netp, v^{(m)}}, \text{for } m = 2, ..., M. \tag{3.24}$$

Now, we substitute (3.19), (3.23) and (3.24) into (3.51), obtaining the output units and hidden units

$$\delta_{p,v}^{(M)} = -e_{p,v} g_v^{(M)} \left(net_{p,v}^{(M)} \right), \quad m = M - 1, \tag{3.25}$$

$$\delta_{p,v}^{(m+1)} = g_v^{(m+1)} \left(net_{p,v}^{m+1} \right) \sum_{\omega=1}^{J_m+2} \delta_{p,\omega}^{(m+2)} w_{v\omega}^{(m+1)}, \quad m = 1, ..., M - 2. \tag{3.26}$$

With the use of (3.25) and (3.26) we have a recursive method to solve $\delta_{p,v}^{(m+1)}$ for the whole network. Thus, the weights $\mathbf{W}$ can be adjusted by

$$\frac{\partial E_p}{w_{uv}^{(m)}} = -\delta_{p,v}^{m+1} O_{p,u}^{(m)}. \tag{3.27}$$

Remark 3.2.3 An adjustment of the weights $\mathbf{W}$ refers to the process of adding the weight adjustments $\Delta\mathbf{W}$ to the old weights $\mathbf{W}$. Furthermore, from (3.18) we know that $\Delta\mathbf{W}$ is the product of the learning rate $\alpha > 0$ and the gradient, multiplied by -1, which implies that the objection function E_p always decreases.
In addition to updating the weights, we can also update the biases. Let's first define two key concepts.

Definition 3.2.1 (Autocorrelation) *Let $\{X_n,\, n \in \mathbb{N}\}$ be a sequence of random variables. The autocorrelation coefficient between two terms of the sequence X_n and X_{n+k} is*

$$\rho(n, n+k) = \frac{Cov[X_n, X_{n+k}]}{\sqrt{Var[X_n]\, Var[X_{n+k}]}}.$$

Definition 3.2.2 (Eigenvalue) *Let $\mathbf{A}$ be a matrix. If there is a vector $\mathbf{X} \in \mathbb{R}^n \neq 0$ such that*

$$A\mathbf{X} = \lambda\mathbf{X},$$

for some scalar λ, then λ is called an eigenvalue of $\mathbf{A}$ with corresponding (right) eigenvector $\mathbf{X}$.

The biases can be updated with the gradient descent method concerning $\boldsymbol{\theta}^{(m)}$, by following the above methodology. However, the biases can be treated as a special kind of weight, which are usually omitted in practical applications.

The BP algorithm given by (3.19) can be rewritten as

$$\Delta_p\mathbf{W}(t) = -\alpha\frac{\partial E_p}{\partial \mathbf{W}}. \tag{3.28}$$

The algorithm is convergent in the mean if $\alpha > 0$ is two times smaller than the inverse of the largest eigenvalue of the autocorrelation matrix, which is a square matrix giving the autocorrelation between each pair of elements of a given random input vector $\mathbf{x}$, denoted by $\mathbf{R}$. However, if α is too small, the likelihood of getting stuck at a local minimum of the error function increases. If instead, the learning rate is too high the likelihood of falling into oscillatory traps is higher, i.e. swinging backward and forward like a pendulum without any progress [8]. The difference in learning rates can be visualized with synthetic data by a simple illustration, see Fig. 3.5.

Remark 3.2.4 In practice, α is usually chosen between 0 and 1 [9].

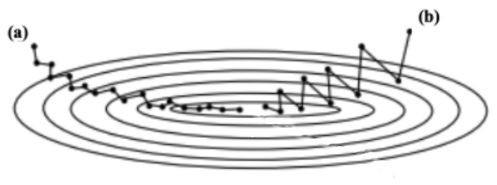

Figure 3.5: Descent in weight space, the space of all possible weight values, for a (**a**) small learning rate and (**b**) large learning rate. The smaller learning rate takes more, but smaller, steps to converge to the center. The larger learning rate takes less, but larger, steps to converge to the center.

A pseudo algorithm flowchart for a three-layer MLP is shown in Pseudo Algorithm 3.1.2.

Pseudo Algorithm 3.1.2 (BP for three-layer MLP).

```
 Let all units have the same activation function $g$, and all
biases be combined into the weight matrices.
  1. Initialize $\mathbf{W}^{(1)}$ and $\mathbf{W}^{(2)}$.
  2. Use (3.15) to calculate $E$.
  3. For each epoch:
       • Use (3.15) to calculate $E$.
       • If $E$ is less than a threshold $\epsilon$, return weights.
       • For all $\mathbf{x}_p$, $p = 1, ..., N$ :
         (a) Forward pass:
              (i) Compute $\mathbf{net}_p^{(2)}$ by (3.13) and $\mathbf{o}_p^{(2)}$ by (3.14).
             (ii) Compute $\mathbf{net}_p^{(3)}$ by (3.13) and $\mathbf{o}_p^{(3)} = \hat{\mathbf{y}}_p$ by (3.14).
            (iii) Compute $\mathbf{e}_p$ by (3.17).
         (b) Backward pass (learning), for all neurons:
              (a) Compute $\delta_{p,v}^{(3)}$ by (3.25).
              (b) Update $\mathbf{W}^{(2)}$ by (3.28).
              (c) Compute $\delta_{p,v}^{(2)}$ by (3.26).
              (d) Update $\mathbf{W}^{(1)}$ by (3.28).
  4. End.
```

3.3 Long short-term memory

Long short-term memory (LSTM) is an artificial recurrent neural network (RNN) architecture [39, 42]. Unlike the standard feedforward neural networks, see the previous section, RNN has feedback connections. These feedback connections enable the propagation of data from the earlier event to the current processing, which builds a memory of time-dependent events.

3.3.1 Basics

RNNs range from partly to fully connected, see Fig. 3.6 and Fig. 3.7. The Elman network is a simple RNN similar to a three-layer MLP network, with an additional layer for saving the outputs of the hidden layer in so-called context cells, see Fig. 3.8 [35, 28]. Then the output of these context cells is fed back to the hidden neurons along with the originating signals.

The difference in structure between the standard feedforward neural networks and recurrent ones also means that they also need to be trained differently. This is because, for RNNs, information needs to be propagated through the recurrent connections inbetween steps. In this thesis, we will examine two common learning algorithms for supervised temporal learning, backpropagation through time (BPTT) and real-time recurrent learning (RTRL) [39, 42]. The original formulation of LSTM networks used a combination of BPTT and RTRL. Therefore, we will cover both learning algorithms in short.

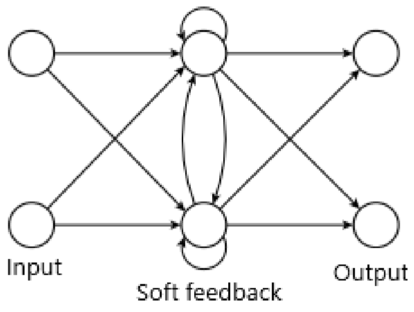

Figure 3.6: Partly connected RNN. The input comes from the left and goes through the feedback connections in the hidden layer which enable the propagation of data from the earlier event to the current processing and then return outputs.

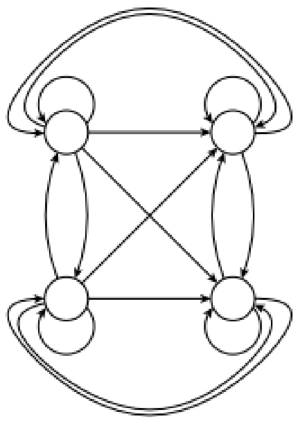

Figure 3.7: Fully connected RNN. In addtion to having feedback connection in the hidden layers as a partly connected RNN, a fully connected RNN have feedback connections everywhere.

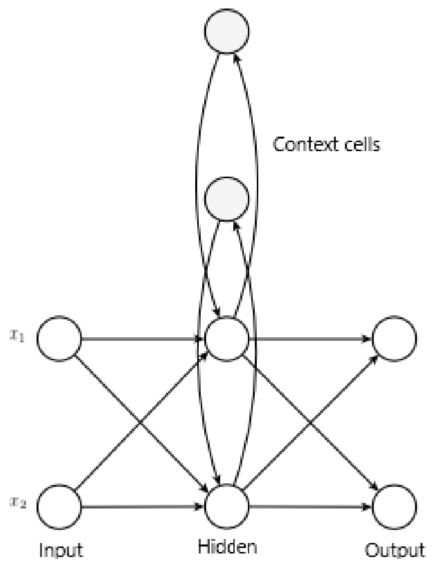

Figure 3.8: Elman network.

3.3.2 Backpropagation Through Time

For a finite period of time, there is a feedforward network identical to every RNN [39]. Thus, the BPTT uses this fact by unfolding the RNN in time. Fig. 3.9 (a) shows a fully recurrent neural network with one two-neuron layer. The corresponding feedforward neural network, shown in Fig. 3.9 (b), consists of separate layers for each given timestep with the same weight for all layers. Hence, if the weights are identical, both networks show the same behavior.

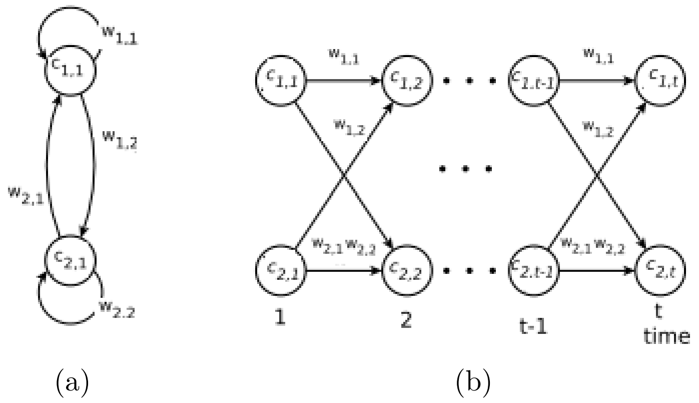

Figure 3.9: (a) shows a fully recurrent neural network with a two-neuron layer. Same network structure but unfolded trough time is shown in (b).

As the unfolded network is an MLP network, it can be trained using the same backpropagation algorithm described in Section 3.2.1.

We use the following iterative backpropagation algorithm to calculate the error signal for a unit for all steps in a pass. We consider discrete timesteps $\tau = 1, 2, 3...$ The network starts at a point in time t' and runs until a final time t. Now, we let U

be the set of non input units, and let g_u be the non-linear differentiable, sigmoid function of the unit $u \in U$; the output $y_u(\tau)$ of u at time τ is given by

$$y_u(\tau) = g_u(z_u(\tau)), \tag{3.29}$$

with the weighted input

$$\begin{aligned} z_u(\tau+1) &= \sum_l W_{[u,v]} X_{[l,u]}(\tau+1) \\ &= \sum_v W_{[u,v]} y_v(\tau) + \sum_i W_{[u,i]} y_i(\tau+1) y_v(\tau), \end{aligned} \tag{3.30}$$

where $W_{[u,v]}$ are the weight connected to neurons u and v, $l \in K(u)$, $K(u)$ is the set of units v that proceeds u, $v \in U \cap K(u)$ and $i \in I$, which is the set of input units. The inputs at $\tau + 1$ to u are of two types: the recurrent output from all non-input units in the network produced at time τ and the environmental input that arrives at time $\tau + 1$ by the input units. Thus, if the network is fully connected we have that $U \cap K(u)$ is equal to the set U of non-input units. If $T(\tau)$ is the set of non-input units for which, at time τ, the output value $y_u(\tau)$ of the unit $u \in T(\tau)$ is equal to the target value $d_u(\tau)$. The cost function is the sum of all errors $E_{total}(t', t)$ for the epoch $t', t'+1, ..., t$ and we want to minimise this cost function using a learning algorithm.

Hence, the total error is defined by

$$E_{total}(t', t) = \sum_{\tau=t'}^{t} E(\tau), \tag{3.31}$$

with the error $E(\tau)$ at time τ defined using the squared error as an objective function by

$$E(\tau) = \frac{1}{2} \sum_{u \in U} (e_u(\tau))^2,$$

and with the error $e_u(\tau)$ of the non-input unit u at time τ defined by

$$e_u(\tau) = \begin{cases} d_u(\tau) - y_u(\tau), & \text{if } u \in T(\tau) \\ 0, \ \text{otherwise} \end{cases}$$

We update the weights, using the error signal $\psi_u(\tau)$ of a non-input unit u at a time τ, which is defined by

$$\psi_u(\tau) = \frac{\partial E(\tau)}{\partial z_u(\tau)}. \tag{3.32}$$

When we unroll ψ_u over time, we obtain the equality

$$\psi_u(\tau) = \begin{cases} g'_u(z_u(\tau)) e_u(\tau), \ \text{if } \tau = t, \\ g'_u(z_u(\tau)) (\sum_{k \in U} W_{[k,u]} \psi_k(\tau+1)), \ \text{if } t' \leq \tau < t. \end{cases}$$

When the backpropagation is performed until time t', we calculate the weight update $\Delta W_{[u,v]}$ for the recurrent version of the network:

$$\Delta W_{[u,v]} = -\alpha \frac{\partial E_{total}(t', t)}{\partial W_{[u,v]}}, \tag{3.33}$$

with

$$\frac{\partial E_{total}(t', t)}{\partial W_{[u,v]}} = \sum_{\tau=t'}^{t} \psi_u(\tau) \frac{\partial z_u(\tau)}{\partial W_{[u,v]}} \tag{3.34}$$

$$= \sum_{\tau=t'}^{t} \psi_u(\tau) X_{[u,v]}(\tau). \tag{3.35}$$

For more details on BPTT see [39, 13, 43].

3.3.3 Real-Time Recurrent Learning

For the RTRL algorithm, we do not require error propagation. Instead, all necessary information for the computation of the gradient is collected using an input stream. Thus, we do not need dedicated training intervals. This approach also comes with a high computational cost, and the information is not stored locally, which means that we need a notion for the sensitivity of the output. However, the memory required is only dependent on the size of the network and not the input.

The network units $v \in I \cup U$ and $u, k \in U$, with the timesteps $t' \leq \tau \leq t$. We assume the existence of a labeled data point $d_k(\tau)$ at every time τ for every non-input unit k. Hence, the training objective is to minimize the overall error, which is at timestep τ given by

$$E(\tau) = \frac{1}{2} \sum_{k \in U} (d_k(\tau) - y_k(\tau))^2. \tag{3.36}$$

From equation (3.31) we note that the gradient of the total error is equal to the sum of the gradient of all previous steps and the current:

$$\Delta_W E_{total}(t', t+1) = \Delta E_{total}(t', t) + \Delta_W E(t+1), \tag{3.37}$$

where Δ_W is the gradient of the weights.

When presenting the time series to the network, we need to collect the gradient for each timestep, which also means that we can track the deltas, or weight changes, $\Delta W_{[u,v]}(\tau)$. Thus, the overall deltas for $W_{[u,v]}$ is given by

$$\Delta W_{[u,v]} = \sum_{\tau=t'+1}^{t} \Delta W_{[u,v]}(\tau). \tag{3.38}$$

To obtain the deltas, we must calculate

$$\Delta W_{[u,v]}(\tau) = -\alpha \frac{\partial E(\tau)}{\partial W_{[u,v]}}, \tag{3.39}$$

for each timestep t, where α is the learning rate. Using the expansion by gradient decent together with equation (3.36), we get

$$\begin{align}\Delta W_{[u,v]}(\tau) &= -\alpha \sum_{k \in U} \frac{\partial E(\tau)}{\partial y_k(\tau)} \frac{\partial y_k(\tau)}{\partial W_{[u,v]}} \tag{3.40}\\ &= -\alpha \sum_{k \in U} (d_k(\tau) - y_k(\tau)) \left(\frac{\partial y_k(\tau)}{\partial W_{[u,v]}} \right). \tag{3.41}\end{align}$$

Since the error $e_k(\tau) = d_k(\tau) - y_k(\tau)$ is known, we only need to calculate the second factor. The measure of sensitivity are given by

$$p^k_{uv}(\tau) = \frac{\partial y_k(\tau)}{\partial W_{[u,v]}}, \tag{3.42}$$

where $y_k(\tau)$ is the output of unit at time τ to change in the weight $W_{[u,v]}$. Since unit k does not have to be connected with the weight $W_{[u,v]}$, it makes the algorithm non-local. In RTRL, the gradient information is forward propagated. Using equations (3.29) and (3.30), the output $y_k(t+1)$ at timestep $t+1$ is given by

$$y_k(t+1) = g_k(z_k(t+1)), \tag{3.43}$$

with the weighted input

$$z_k(t+1) = \sum_l W_{[k,l]} X_{[k,l]}(t+1) = \sum_{v \in U} W_{[k,v]} y_v(t) + \sum_{i \in I} W_{[k,i]} y(t+1), \tag{3.44}$$

where $l \in K(k)$ with $K(k)$ denoting the set of units with connections to a unit k, and $X_{[k,l]}$ denotes the input of unit u coming from a unit v.

We differentiate equations (3.42), (3.43) and (3.44) to calculate the results for all timesteps $\geq t+1$ with

$$\begin{align}p^k_{uv}(t+1) = \frac{\partial y_k(t+1)}{\partial W_{[u,v]}} &= \frac{\partial}{\partial W_{[u,v]}} \left[g_k \left(\sum_{l \in Pre(k)} W_{[k,l]} X_{[k,l]}(t+1) \right) \right] \\ &= g'_k(z_k(t+1)) \left[\frac{\partial}{\partial W_{[u,v]}} \left(\sum_{l \in K(k)} W_{[k,l]} X_{[k,l]}(t+1) \right) \right] \\ &= g'_k(z_k(t+1)) \left[\left(\sum_{l \in K(k)} \frac{\partial W_{[k,l]}}{\partial W_{[u,v]}} X_{[k,l]}(t+1) \right) + \left(\sum_{l \in K(k)} W_{[k,l]} \frac{\partial X_{[k,l]}(t+1)}{\partial W_{[u,v]}} \right) \right] \\ &= g'_k(z_k(t+1)) \left[\delta_{uk} X_{[u,v]}(t+1) + \left(\sum_{l \in U} W_{[k,l]} \frac{\partial y_l(t)}{\partial W_{[u,v]}} + \sum_{i \in l} W_{[k,i]} \frac{\partial y_i(t+1)}{\partial W_{[u,v]}} \right) \right] \\ &= g'_k(z_k(t+1)) \left[\delta_{uk} X_{[u,v]}(t+1) + \sum_{l \in U} W_{[k,l]} p^l_{uv}(t) \right], \tag{3.45}\end{align}$$

where δ_{uk} is the Kronecker delta, i.e.,

$$\delta_{uk} = \begin{cases} 1, & \text{if } u = k, \\ 0, & \text{otherwise.} \end{cases}$$

Assuming that the initial network state has no functional dependency on the weights, the derivative for the first step is given by

$$p_{uv}^{k}(t') = \frac{\partial y_k(t')}{\partial W_{[u,v]}} = 0. \tag{3.46}$$

Equation (3.45) shows how $p_{uv}^{k}(t+1)$ can be calculated in terms of $p_{uv}^{k}(t)$. Thus, we have an incremental learning algorithm, which allows for learning in real-time instead of using backpropagation through time.

Given the initial value for p_{uv}^{k} at time t' from (3.46), we can recursively calculate p_{uv}^{k} for the first and all subsequent timesteps using (3.45). Note that $p_{uv}^{k}(\tau)$ uses the values of $W_{[u,v]}$ at t', and not values inbetween t' and τ. Using the combination of these two values with the error vector $e(\tau)$ and (3.14), we get the negative error gradient $\Delta WE(\tau)$. The final deltas for $W_{[u,v]}$ can be calculated using equations (3.14) and (3.13).

For a more detailed description of the RTRL algorithm, see [42] and [43].

3.3.4 Vanishing Error Problem

Backpropagation error signals tend to change with each timestep. Hence, if we propagate over many steps, the error typically blows up or vanishes [60, 50]. A standard RNN cannot bridge more than 5–10 timesteps [18]. If the error blows up, it leads straight to oscillating weights, while a vanishing error leads to learning taking an unacceptable amount of time or does not work at all.

Using the standard backpropagation algorithm, we update the weights from time t' to time t using the formula

$$\Delta W_{[u,v]} = -\alpha \frac{\partial E_{total}(t',t)}{\partial W_{[u,v]}}, \tag{3.47}$$

with

$$\frac{\partial E_{total}(t',t)}{\partial W_{[u,v]}} = \sum_{\tau=t'}^{t} \psi_u(\tau) X_{[u,v]}(\tau),$$

where the backpropagated error signal at time τ (with $t' \le \tau < t$) of the unit u is

$$\psi_u(\tau) = g_u'(z_u(\tau)) \left(\sum_{v \in U} W_{vu} \psi_v(\tau+1) \right). \tag{3.48}$$

Assuming that we have a fully recurrent neural network with a set of non-input units U, the error that occurs at any output-layer neuron $o \in O$, at time-step τ, is backpropagated through time for $t - t'$ time-steps, with $t' < t$ to an arbitrary neuron v. Thus, the error is scaled by the following factor:

$$\frac{\partial \psi_v(t')}{\partial \psi_0(t)} = \begin{cases} g'_v(z_v(t'))W_{[o,v]}, \text{ if } t - t' = 1, \\ g'_v(z_v(t')) \left(\sum_{u \in U} \frac{\partial \psi_u(t'+1)}{\partial \psi_0(t)} W_{[u,v]}\right), \text{ if } t - t' > 1. \end{cases} \tag{3.49}$$

We unroll the above equation over time and let u_τ be a non-input-layer neuron in one of the replicas in the unrolled network at time τ for $t' \leq \tau \leq t$. Now, by setting $u_t = v$ and $u_{t'} = o$, we obtain

$$\frac{\partial \psi_v(t')}{\partial \psi_0(t)}) = \sum_{u_{t'} \in U} \cdots \sum_{u_{t-1} \in U} \left(\prod_{\tau = t'+1}^{t} g'_{u_\tau}(z_{u_\tau}(t - \tau + t'))W_{[u_\tau, u_{\tau-1}]} \right). \tag{3.50}$$

It follows

$$|g_{u_\tau}(z_{u_\tau}(t - \tau + t'))W_{[u_\tau, u_{\tau-1}]}| < 1, \tag{3.51}$$

for all τ, the error vanishes, as the product decreases exponentially [18]. In other words, as each iteration receives a weight proportional to the partial derivative of the error function with respect to the current weight, the update will become so small that the weights can not be changed. Thus, the network is unable to learn within an acceptable time period or in some cases not at all [51]. Moreover, the sum

$$\sum_{o \in O} \frac{\partial \psi_v(t')}{\partial \psi_0(t)}, \tag{3.52}$$

indicates that if local error vanishes, then the global error also vanishes.
For a more detailed analysis of the problem, see [51].

One solution to the vanishing error problem is to use a gradient based method; more specifically, we can use a long short-term memory (LSTM) [49, 48, 18, 19]. The LSTM network can bridge time lags of more than 1000 discrete timesteps. This is due to constant error carousels (CECs), which enforce a constant error flow within special cells.

3.3.5 Constant Error Carousel

Let u be the only unit connected to itself. Then, from (3.48) it follows that the local error backflow of u at a timestep is given by

$$\psi_u(\tau) = g'_u(z_u(\tau))W_{[u,u]}\psi_u(\tau + 1). \tag{3.53}$$

From equations (3.50) and (3.51), we note, if we want to ensure a constant error flow through u, we must have

$$g'_u(z_u(\tau))W_{[u,u]} = 1, \tag{3.54}$$

and by integration we have

$$g_u(z_u(\tau)) = \frac{z_u(\tau)}{W_{[u,u]}}. \tag{3.55}$$

Hence, g_u is linear and the output of unit u is given by

$$y_u(\tau+1) = g_u(z_u(\tau+1)) = g_u(y_u(\tau))W_{[u,v]}) = y_u(\tau), \tag{3.56}$$

remains constant over time. Using the identity function $g_u = id$ and setting $W_{[u,u]} = 1$ ensures the linearity and consistency over time, which is the so-called constant error carousel (CEC). CEC is a key feature of the LSTM network, which allows for short-term memory to be stored over extended periods [18]. However, the CEC does not consider the connection from other units to u, which is where the different components of LSTM emerge.

3.3.6 Memory Blocks

We note from the previous section that the CEC's backflow is constant if there are no new cell inputs. However, this is not a realistic assumption as the CEC is a part of a neural network, which means it is also connected to other units in the network [49]. Therefore, we need to account for these in- and outputs. Furthermore, the incoming connection to a neuron can be conflicted by weight update signals since the same weights can be used to control the storage and discernment of inputs.

The LSTM networks address the problem with conflicting weight updates by implementing input and output gates connected to the input layer and other memory cells, referred to as a memory block.

The input gates control the signals from the network to the memory cells by scaling them with the use of a sigmoid activation function ranging between 0 and 1 [48]. The input gates can also learn to discard irrelevant signals to preserve the stored content. The output gates control the access to the memory cell contents, which can protect other memory cells from irrelevant information originating from u. Therefore, we note that the functionality of the gates is to either allow or deny access to the constant error flow via the CEC.

See Fig. 3.10 for a illustration of a standard LSTM memory block.

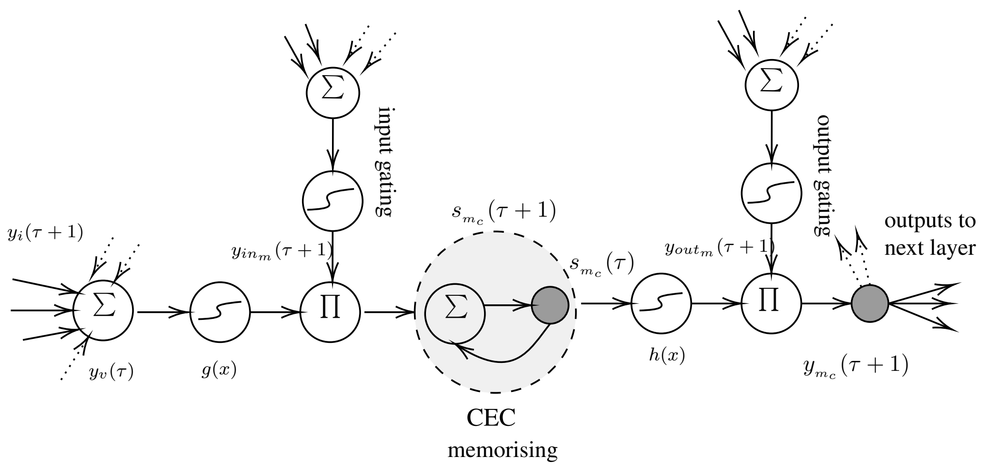

Figure 3.10: A standard LSTM memory block. The block contains (at least) one cell with a recurrent self-connection (CEC)

3.3.7 Training LSTM

The original LSTM network used a combination of the two learning algorithms: BPTT to components located after the cells and RTRL to train cells and components located before the cells.

Remark 3.3.1 *We assume that the gradient of the cells is only allowed to be propagated through time, limiting the rest of the gradients for the other recurrent connections.*

Suppose that we have discrete timesteps in the form $\tau = 1, 2, 3, ...$, where each step has a forward- and backward pass. The forward pass is used to calculate the output/activation of all units, and the backward pass is used to calculate the error signals for all weights.

3.3.8 The Forward Pass

Let $W_{[u,v]}$ be a weight connecting unit u to unit v, M be the set of memory blocks and m_c be the $c-$th memory cell in the memory block m.

As in the original LSTM formulation, we suppose that each memory block m is associated with one input gate in_m and one output gate out_m. Moreover, the internal state of a memory cell m_c at time $\tau + 1$ is updated in relation to its state $s_{m_c}(\tau)$ and in coherance with the weighted input $z_{m_c}(\tau + 1)$ multiplied by the activation of the input gate $y_{in_m}(\tau + 1)$. Then, the activation for output gate $z_{out_m}(\tau + 1)$ is calculate for the activation of the cell $y_{m_c}(\tau + 1)$.

Recall that $K(u)$ denotes the set of units with connections to a unit u, $W_{[v,u]}$ denotes the weight that connects the unit v to the unit u, y_u denotes the output of unit u, z_u denotes the weighted input of unit u, g_u denotes the activation function of unit u, U is the set of non input units, I is the set of input unit and $X_{[v,u]}$ denotes the input of a unit u coming from a unit v.

The output of the input gate is computed as

$$y_{in_m}(\tau+1) = g_{in_m}(z_{in_m}(\tau+1)), \tag{3.57}$$

with the gate input

$$\begin{aligned} z_{in_m}(\tau+1) &= \sum_u W_{[in_m,u]} X_{[u,in_m]}(\tau+1) \\ &= \sum_{u\in U} W_{[in_m,v]} y_v(\tau) + \sum_{i\in I} W_{[in_m,i]}(\tau+1), \end{aligned} \tag{3.58}$$

where $u \in K(in_m)$ with $K(in_m)$ denoting the set of units with connections to a input gate unit m. The output of the output gate is computed as

$$y_{out_m}(\tau+1) = g_{out_m}(z_{out_m}(\tau+1)), \tag{3.59}$$

with

$$\begin{aligned} z_{out_m}(\tau+1) &= \sum_u W_{[out_m,u]} X_{[u,out_m]}(\tau+1) \\ &= \sum_{v\in U} W_{[out_m,v]} y_v(\tau) + \sum_{i\in I} W_{[out_m,i]} y_i(\tau+1), \end{aligned} \tag{3.60}$$

where $u \in K(out_m)$. Then, the gates are scaled using the non-linear activation function $g_{in_m} = g_{out_m} = g$, defined by

$$g(s) = \frac{1}{1+e^{-s}}, \tag{3.61}$$

so that they are within the range $[0,1]$. Hence, the input for the memory cell will only pass if the signal at the input gate is sufficiently close to one.

For a memory cell m_c in the memory block m, the weighted input $z_{m_c}(\tau+1)$ is given by

$$z_{m_c}(\tau+1) = \sum_u W_{[m_c,u]} X_{[u,m_c]}(\tau+1) = \sum_{v\in U} W_{[m_c,v]} y_v(\tau) + \sum_{i\in I} W_{[m_c,i]} y_i(\tau+1). \tag{3.62}$$

The internal state $s_{m_c}(\tau+1)$ of the unit in the memory cell at time $\tau+1$ is computed differently, as mentioned before. The corresponding equation is

$$s_{m_c}(\tau+1) = s_{m_c}(\tau) + y_{in_m}(\tau+1) g(z_{m_c}(\tau+1)), \tag{3.63}$$

with $s_{m_c}(0) = 0$ and the non-linear activation function for the cell input

$$g(z) = \frac{4}{1+e^{-z}} - 2, \tag{3.64}$$

which scales the result to within the range $[-2,2]$.

We get the output y_{m_c} by using the output function and multiplying s_{m_c} by y_{out_m}

$$y_{m_c}(\tau+1) = y_{out_m}(\tau+1)h(s_{m_c}(\tau+1)), \tag{3.65}$$

with the non-linear activation function

$$h(z) = \frac{2}{1+e^{-z}} - 1, \tag{3.66}$$

with the range $[-1, 1]$.

Suppose we have layered RNN with standard input, standard output and hidden layer consisting of memory blocks. Then the activation of the output unit o is given by

$$y_0(\tau+1) = g_o(z_o(\tau+1)), \tag{3.67}$$

where

$$z_o(\tau+1) = \sum_{u \in U-G} W_{[o,u]} y_u(\tau+1), \tag{3.68}$$

where U is the set of non input units and G is the set of gate units.

3.3.9 Forget Gates

In order to preserve cell state over time, the self-connection of a regular LSTM has a fixed weight of 1. However, the cell states grow linearly during the continued progress of the input stream, which can lead to the network losing its memorizing property, and function like a regular RNN.

We could limit the growth by manually resetting the state of the cell at the beginning of each sequence. While this seems to be a simple solution, it is not practical for continuous inputs as there would be no way of distinguish the start of a sub-sequence from the whole sequence.

One possible solution to this problem is to attach an adaptive forget gate to the self-connection [18]. The forget gates can learn to reset the cell state if the information is not needed. Thus, we can replace the weight of one for the self-connection from the constant error carousel (CEC) with a forget gates activation y_{ϕ_m} of memory block m, given by

$$y_{\phi_m}(\tau+1) = g_{\phi_m}(z_{\phi_m}(\tau+1) + b_{\phi_m}), \tag{3.69}$$

where $g \in [0, 1]$ is the activation function from equation (3.61), b_{ϕ_m} is the bias of the forget gate, and

$$\begin{aligned} z_{\phi_m}(\tau+1) &= \sum_u W_{[\phi_m,u]} X_{[u,\phi_m]}(\tau+1) && (3.70) \\ &= \sum_{v \in U} W_{[\phi_m,u]} y_v(\tau) + \sum_{i \in I} W_{[\phi_m,i]} y_i(\tau+1). && (3.71) \end{aligned}$$

We fix b_{ϕ_m} to 1 for improved performance of the LSTM [40].

The internal cell state s_{m_c} updated with a forget gate is given by

$$s_{m_c}(\tau+1) = s_{m_c}(\tau) y_{\phi_m}(\tau+1) + y_{in_m}(\tau+1) g(z_{m_c}(\tau+1)), \tag{3.72}$$

with $s_{m_c}(0) = 0$, $y_{\phi_m}(\tau+1) = 1$ if there is no forget gate, and the activation function is given by equation (3.64), within the range $[-2, 2]$. If we replace equation (3.63) with equation (3.72), we get an extended forward pass.

Initially, the bias weight of the input is set to be negative, while the weights for the forget gate are initially positive. Thus, the forget gate activation will be close to 1 at the beginning of the training. Then the memory cell behaves like a standard LTSM, which prevents the network from forgetting when it has not yet learned anything.

3.3.10 Backward Pass

We separate units into two groups: those whose weight is computed with BPTT and those calculated with RTRL. This separation is considered in this thesis since an LSTM network incorporates both BPTT and RTRL [39, 42].

Using the notations from previous subsections, and with the use of equations (3.31) and (3.32), we express the overall network error at timestep τ as

$$E(\tau) = \frac{1}{2} \sum_{o \in O} (d_o(\tau) - y_o(\tau))^2, \tag{3.73}$$

where $d_o(\tau) - y_o(\tau) = e_o(\tau)$ and O is the set of output units.

We initially consider the units that use the BPTT. We define the notion of individual error of a unit u at time τ by

$$\psi_u(\tau) = -\frac{\partial E(\tau)}{\partial z_u(\tau)}, \tag{3.74}$$

where z_u is the weighted input of the unit. In addition, we we also define notion of weight update as

$$\begin{aligned} \Delta W_{[u,v]}(\tau) &= -\alpha \frac{\partial E(\tau)}{\partial W_{[u,v]}} && (3.75) \\ &= -\alpha \frac{\partial E(\tau)}{\partial z_u(\tau)} \frac{\partial z_u(\tau)}{\partial W_{[u,v]}}, && (3.76) \end{aligned}$$

where $\frac{\partial z_u(\tau)}{\partial W_{[u,v]}}$ is the input signal that comes from unit v to unit u. However, depending on the essence of u, the individual error varies. Let us consider u equal to an output unit o, then

$$\psi_o = g'_o(z_o(\tau))(d_o(\tau) - y_o(\tau)). \tag{3.77}$$

Then, the weight contribution of output units is given by

$$\Delta W_{[o,v]}(\tau) = \alpha \psi_o X_{[v,o]}(\tau). \tag{3.78}$$

If u is equal to a hidden unit h located between cells and output units, then

$$\psi_h(\tau) = g'_h(z_h(\tau)) \left(\sum_{o \in O} W_{[o,h]} \psi_o(\tau) \right), \tag{3.79}$$

where O is the set of output units, and the weight contribution of hidden units is

$$\Delta W_{[h,v]}(\tau) = \alpha \psi_h(\tau) X_{[v,h]}(\tau). \tag{3.80}$$

Finally, if u is equal to the output gate out_m of the memory block m, then

$$\psi_{out_m} \overset{truncated}{=} g'_{out_m}(z_{out_m}(\tau)) \left(\sum_{m_c \in m} h(s_{m_c}(\tau)) \sum_{o \in O} W_{[o,m_c]} \psi_o(\tau) \right), \tag{3.81}$$

where $\overset{truncated}{=}$ is defined as; the equality only holds if and only if the error is truncated, i.e., the error from propagating is not allowed to go backwards to its unit via its own feedback connection.

For the output gates, the weight contribution is given by

$$\Delta W_{[out_m,v]}(\tau) = \alpha \psi_{out_m}(\tau) X_{[v,out_m]}(\tau). \tag{3.82}$$

Let us now consider units that use RTRL. We define the individual error of the cell m_c of the memory block m by

$$\begin{aligned}
\psi_{m_c}(\tau) &\overset{truncated}{=} -\frac{\partial E(\tau)}{\partial s_{m_c}} + \psi_{m_c}(\tau+1) y_{\phi_m}(\tau+1) \\
&\overset{truncated}{=} \frac{\partial y_{m_c}(\tau)}{\partial s_{m_c}} \left(\sum_{o \in O} \frac{\partial z_o(\tau)}{\partial y_{m_c}(\tau)} \left(-\frac{\partial E(\tau)}{\partial z_o(\tau)} \right) \right) + \psi_{m_c}(\tau+1) y_{\psi_m}(\tau+1) \\
&\overset{truncated}{=} y_{out_m}(\tau) h'(s_{m_c}(\tau)) \left(\sum_{o \in O} W_{[o,m_C]} \psi_o(\tau) \right) + \psi_{m_c}(\tau+1) y_{\phi_m}(\tau+1),
\end{aligned} \tag{3.83}$$

where $\psi_{m_c}(\tau+1) y_{\phi_m}(\tau+1)$ is a recurrent connection.

Note that propagating back in time only accounts for the error through its recurrent connection, with the influence of the forget gate. Thus, the following partial derivatives expand the weight contribution for the cell as

$$\begin{aligned}
\Delta W_{[m_c,v]}(\tau) &= -\alpha \frac{\partial E(\tau)}{\partial W_{[m_c,v]}} \\
&= -\alpha \frac{\partial E(\tau)}{s_{m_c}(\tau)} \frac{s_{m_c}(\tau)}{\partial W_{[m_c,v]}} \\
&= \alpha \psi_{m_c}(\tau) \frac{\partial s_{m_c}(\tau)}{\partial W_{[m_c,v]}},
\end{aligned} \tag{3.84}$$

and the weight contribution for forget and input gates as

$$\begin{aligned}\Delta W_{[m_c,v]}(\tau) &= -\alpha\frac{\partial E(\tau)}{\partial W_{[m_c,v]}}\\ &= -\alpha\sum_{m_c\in m}\frac{\partial E(\tau)}{s_{m_c}(\tau)}\frac{s_{m_c}(\tau)}{\partial W_{[u,v]}}\\ &= \alpha\sum_{m_c\in m}\psi_{m_c}(\tau)\frac{\partial s_{m_c}(\tau)}{\partial W_{[u,v]}}.\end{aligned} \tag{3.85}$$

where $\frac{\partial s_{m_c}(\tau+1)}{\partial W_{[u,v]}}$ depends on the the unit u. If u is equal to the cell m_c, then

$$\frac{\partial s_{m_c}(\tau+1)}{\partial W_{[u,v]}} \overset{truncated}{=} \frac{\partial s_{m_c}(\tau)}{\partial W_{[u,v]}}y_{\phi_m}(\tau+1) + g'(z_{m_c}(\tau+1))g_{in_m}(z_{in_m}(\tau+1))y_v(\tau). \tag{3.86}$$

If u is equal to the input gate in_m, then

$$\frac{\partial s_{m_c}(\tau+1)}{\partial W_{[in_m,v]}} \overset{truncated}{=} \frac{\partial s_{m_c}(\tau)}{\partial W_{[in_m,v]}}y_{\phi_m}(\tau+1) + g(z_{m_c}(\tau+1))g'_{in_m}(z_{in_m}(\tau+1))y_v(\tau). \tag{3.87}$$

Finally, if u is equal to a forget gate ϕ_m, then

$$\frac{\partial s_{\phi_c}(\tau+1)}{\partial W_{[\phi_m,v]}} \overset{truncated}{=} \frac{\partial s_{m_c}(\tau)}{\partial W_{[\phi_m,v]}}y_{\phi_m}(\tau+1) + s_{m_c}(\tau))g'_{\phi_m}(z_{\phi_m}(\tau+1))y_v(\tau), \tag{3.88}$$

with $s_{m_c}(0)=0$.

For a more detailed version of the LSTM backward pass, see [18].

A pseudo algorithm flowchart for an LTSM network is shown in Pseudo Algorithm 3.1.3.

Pseudo Algorithm 3.1.3 (Training for a LSTM memory block).

```
Let all units have the same activation function $g$, and all
biases be combined into the weight matrices.
 1. Initialize Weights.
 2. For each epoch:
     • Use $(3.36)$ to calculate overall network error, $E$.
     • If $E$ is less than a threshold $\epsilon$, return weights.
     • For all inputs $\mathbf{x}_p$, $p = 1, ..., N$ :
       (a) Forward pass:
           (i) Compute input gate $(3.57)$ and output gate $(3.59)$.
          (ii) Compute forget gate $(3.72)$.
         (iii) Compute output unit $(3.67)$.
         (iii) Compute cell output $(3.62)$.
       (b) Backward pass (learning):
         – BPTT:
           (i) Update output units $(3.78)$.
          (ii) Update hidden units $(3.80)$.
         (iii) Update output gates $(3.82)$.
         – RTRL:
           (i) Update input gates $(3.85)$.
          (ii) Update forget gates $(3.85)$.
         (iii) Update cells $(3.84)$.
 3. End.
```

3.3.11 Bidirectional LSTM

Standard RNNs use just one direction for a given point in a sequence: the past. The work published in [2] explores extending a standard RNN to also incorporate future values as well, refereed to as bidirectional. We can use this to extend the regular LSTM to a bidirectional LSTM. A bidirectional LSTM presents input values backwards and forward to two different LSTM networks, which are both connected to the same output layer. As the LSTM in this thesis aims to learn financial data, we want to preserve information from both past and future which would make the network better suited for predictions on time dependent data [2]. Note, when mentioning future values, it is used to describe values contained in its training data sequence while still positioned ahead of the current input value.

3.3.12 Convolution Neural Network

In this section, we shall briefly introduce a one-dimensional (1D) convolution neural network (CNN). CNNs are a fundamental tool of deep learning and are most often used within image/object recognition and classification [40]. However, 1D CNN's can be used for time series, which also have certain advantages. One of the essential advantages is their low computational requirements, making the 1D CNN more suitable for applications in real-world conditions where the only operation with significant computation cost is the 1D convulsions [40, 29, 35].

Let f be an input vector and g a vector, which is slid across f and multiplied with the input such that the output is enhanced in a certain desirable manner, also know as a kernel. Let f have length n and g have length m. Then the convolution $f * g$ of f and g is given by:

$$conv1D(f, g) := (f * g)(i) = \sum_{j=1}^{m} g(j) \cdot f(i - j + m/2). \tag{3.89}$$

The 1D CNN is predominantly trained using the forward- and backward propagation method, sharing the same fundamental as previously introduced. Moreover, the linear operation, 1D convolution, can run in parallel with the forward- and backward propagation, making the learning process significantly faster.

The structure of typical 1D CNN, as illustrated in Fig. 3.11, consists of:

1. CNN layers;
2. Pooling;
3. MLP Layers.

The MLP component of the CNN is identical to the network introduced in Section 3.4. Thus, we shall only focus on the details of the other components; more specifically the forward- and backward propagation in CNN layers.

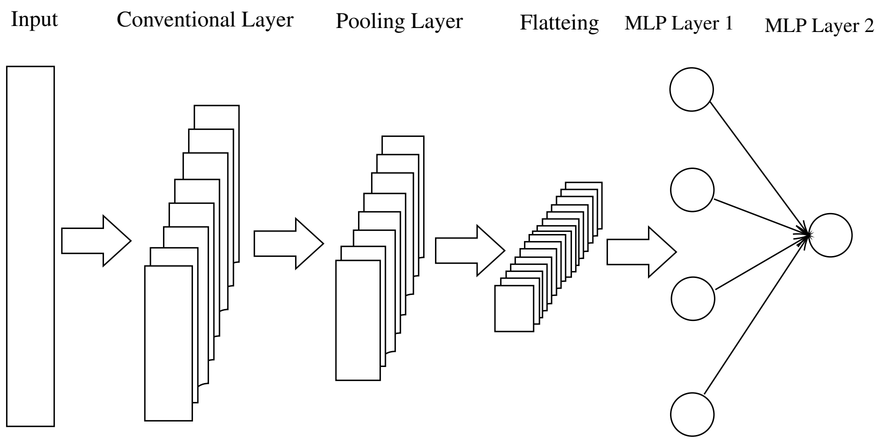

Figure 3.11: A typical 1D CNN configuration with 1 CNN layer and 2 MLP layers.

3.3.13 Forward and backward propagation in CNN-layers

For each CNN layer, 1D forward propagation (FP) is expressed as follows:

$$x_k^l = b_k^l + \sum_{i=1}^{N_{t-1}} conv1D(w_{ik}^{l-1}, s_i^{l-1}), \tag{3.90}$$

where x_k^l is the input, b_k^l is the bias of the kth neuron at layer l, s_i^{l-1} is the output of the ith neuron at layer $l-1$, w_{ik}^{l-1} is the kernel from the ith neuron at layer $l-1$ to the kth neuron at layer l. In addition, we note that the $conv1D$ given by (3.89) has no zero-padding. Zero-padding refers to the process of adding zeros on the edges of the input data to keep the dimensions of the input the same after the operation. Hence, the input array dimension is less than the output array dimension, and the intermediate output, y_k^l, can be expressed by the input x_k^l passing through the activation function, f,

$$y_k^l = f(x_k^l) \text{ and } s_k^l = y_k^l \downarrow ss, \tag{3.91}$$

where s_k^l is output of the kth neuron of the layer, l, and $\downarrow ss$ is the down sampling operation, also refereed to as pooling, with a scale factor ss.

The backpropagation (BP) algorithm error starts from the output MLP layer. Let $l = 1$ for the input layer, $l = L$ for the output layer, and N_L be the number of classes in the data. Given an input vector p, let the target be given by $\mathbf{t}^p$ and the output vectors be given by $[y_1^l, ..., y_{N_L}^l]T$. Thus, with the MSE as the error metric, we express the total error E_p as

$$E_p = \text{MSE}(\mathbf{t}^p, [y_1^l, ..., y_{N_L}^l]^T) = \sum_{i=1}^{N_L}(y_i^L - t_i^p)^2. \tag{3.92}$$

Next, we calculate the delta error, $\Delta_k^l = \frac{\partial E_p}{\partial x_k^l}$ which is used for updating the bias of that neuron and all weights of the neurons in the preceding layer. Using the chain-rule of derivatives, we obtain

$$\frac{\partial E_p}{\partial w_{ik}^{l-1}} = \Delta_k^l y_i^{l-1}, \tag{3.93}$$

$$\frac{\partial E_p}{\partial b_k^l} = \Delta_k^l. \tag{3.94}$$

From the first MLP layer to the last CNN layer, the regular BP is performed as

$$\frac{\partial E_p}{\partial s_k^l} = \Delta s_k^l y_i^{l-1} = \sum_{i=1}^{N_{t+1}} \frac{\partial E_p}{\partial x_i^{l+1}} \frac{\partial x_i^{l+1}}{\partial s_k^l} = \sum_{i=1}^{N_{t+1}} \Delta_i^{l+1} w_{ki}^l. \tag{3.95}$$

When the first BP is executed for l, the BP for the next layer, $l+1$, is executed, and once the BP is performed for $l+1$, then the input layer l can carry over the delta error Δ_k^l. Let the zero-order up-sampled map be $us_k^l = up(s_k^l)$; then the delta errror is given by

$$\Delta_k^l = \frac{\partial E_p}{\partial y_k^l}\frac{\partial y_k^l}{\partial x_k^l} = \frac{\partial E_p}{\partial us_k^l}\frac{\partial us_k^l}{\partial y_k^l} g'(x_k^l) = up(\Delta s_k^l)\beta g'(x_k^l), \tag{3.96}$$

where $\beta = (ss)^{-1}$. The BP of the delta error can then be expressed as,

$$\Delta s_k^l = \sum_{i=1}^{N_{t+1}} conv1Dz(\Delta_i^{l+1}, rev(w_{ki}^l)), \tag{3.97}$$

where $rev()$ is used to reverse the array and $conv1Dz()$ is used to perform full 1D convolution with zero-padding. We can express the weight and bias sensitivities by

$$\frac{\partial E_p}{\partial w_{ik}^l} = conv1D(s_k^l, \Delta_i^{l+1}) \text{ and } \frac{\partial E_p}{\partial b_k^l} = \sum_{i=1}^{N_{t+1}} \Delta_i^l. \tag{3.98}$$

The weight and bias sensitivities can be used to update biases and weights with the learning factor, α, as

$$w_{ik}^{l-1}(t+1) = w_{ik}^{l-1} - \alpha\frac{\partial E_p}{\partial w_{ik}^{l-1}} \text{ and } b_k^l(t+1) = b_k^{\prime}(t) - \alpha\frac{\partial E_p}{\partial b_k^l}. \tag{3.99}$$

A pseudo algorithm flowchart for a CNN is shown in Pseudo Algorithm 3.1.4.

Pseudo Algorithm 3.1.4 (Training for a CNN).

```
 Let all units have the same activation function $g$, and all
biases be combined into the weight matrices.
  1. Initialize Weights.
  2. For each epoch:
       • Use (3.92) to calculate overall network error, $E_p$.
       • If $E_p$ is less than a threshold $\alpha$, return weights.
       • For all inputs $\mathbf{x}_p$, $p = 1, ..., N$ :
         (a) Forward propagation:
              (i) Compute outputs of each neuron at each layer by
                  (3.90).
         (b) Backward propagation:
              (i) Compute delta error for output layer by (3.96).
             (ii) Backpropagate the delta error to the first
                  hidden layer using (3.97).
       • Compute the weight and bias sensitivities by (3.98).
       • Update the weights and biases by (3.99).
  3. End.
```

For further details of the CNN learning algorithm, see [36].

4
Parameter Calibration

This chapter presents two methods for calibrating the fixed parameters of each included method. The first calibration method is a non-linear least squares method. The second calibration method is a genetic algorithm, which is a method that relies on evolutionary reinforcement machine learning techniques.

4.1 Non-linear least squares

The first method calibrates the parameter using a non-linear least squares problem

$$\psi_m = arg\, min_{\theta \in \Psi} \sum_{i=1}^{N} (C_i - C_i^m)^2, \tag{4.1}$$

where m is the option pricing model (see section 2.2), N is the number of options, Ψ is the set of parameters, C_i is the option price of option i, C_i^m is the option price of option i obtained by model m. We approximate the bounds of each parameter, see table 4.1, using economic intuition of financial markets [6].

	σ	λ	m	v	ρ	κ	θ	v_0
B–S	$[0,1]$							
Merton	$[0,1]$	$[0,10]$	$[-1,1]$	$[0,1]$				
Heston	$[0,1]$				$[-1,1]$	$[0,1]$	$[0,1]$	$[0,1]$
Bates	$[0,1]$	$[0,10]$	$[-1,1]$	$[0,1]$	$[-1,1]$	$[0,1]$	$[0,1]$	$[0,1]$

Table 4.1: Intuitively approximated bounds for each option pricing model.

If we want to calibrate a model with a sufficiently small set of parameters on a sufficiently small data set, the method is viable. Trying to calibrate a model with multiple parameters on a large data set makes a non-linear least square method untenable due to computational intensity [10]. However, the notion of sufficiently small is in the eye of the beholder and could therefore theoretically be applied in all cases. In addition, the approximated bounds of the parameters can be wider or smaller. For this thesis, we do not consider the method viable for models with multiple parameters and/or an extensive data set. Thus, we take a random sample of 10% from the training data set to calibrate on and then extrapolate to the whole data set.

4.2 Genetic algorithm

The genetic algorithm is a machine learning search algorithm. The algorithm uses a biologically evolutionary process when searching for the best solution as introduced in [62].

Let us start by introducing the method. First, we note that each solution, i.e. a set of calibrated parameters, is represented in a binary string. For example, if we have the solution $\psi = (1, 2, 3, 4)$, it is represented as

$$1\ 10\ 11\ 100.$$

The algorithm starts by creating a population of random possible solutions that are binary represented. The next step is the evolutionary aspect, which refers to a solution evolving over multiple generations in the pursuit of a better solution. The evolution step starts with a new empty population, referred to as next generation, that gets populated by solutions based on a probability collection process that adapts over each generation, where the fittest solutions are more likely to be picked into the next generation. Fitness is determined by a pre-defined objective function, for example, minimizing the MSE. After the population collection process, i.e., the next generation is full, each solution gets randomly paired with another solution in the next generation, the pair is then referred to as parents. The parents are then crossed over, which creates two new individuals, referred to as children. The children adopt the binary characteristics of the parents with some randomness for non-shared characteristics of the parents, referred to as a mutation. Solutions can mutate in each generation with a pre-determined probability, where the mutation is the process of changing one random bit in the representation of the solution. The algorithm stops when obtaining a pre-determined level of fitness or number of generations.

Compared to the previous calibration method, the genetic algorithm is far less computationally heavy. Partly due to the inexpensive binary operations, but primarily, due to the searching configuration. Instead of testing each possible solution, the algorithm only looks at a fraction of possible solutions. The reason that the algorithm only needs to look at a fraction of solutions is induced by the reinforcement learning principle, which is to learn from each iteration. In addition, as the method uses reinforced learning, it does not require labeled data, making the method applicable in cases where data is unlabeled. Note, the performance is determined by the set level of fitness which is similar to determining the parameters for the non-linear method. Thus, both methods could achieve the same level of fitness but at different rates [62, 10].

As the genetic algorithm can be incorporated into the mathematical option pricing models, they inherit advantages of a machine learning approach, making the models a hybrid version. The hybrid versions can solve common problems of calibrating the regular option pricing models. For example, a common investor practice is to take a rolling average of the historical volatility as the implied volatility [1] input

[1] Implied volatility is the forecasted volatility.

parameter. The practice works if one assumes that the market is a random walk or a market with the perfect competition since the historical volatilities should be the same independent of the rolling average window since otherwise there exist arbitrage opportunities. However, research suggests that market inefficiencies exist, indicating that the historical volatility depends on the window size for the rolling average of the historical volatility, presenting the investor with an additional choice [6]. Thus, choosing input parameters can be a complex issue, where a machine learning approach only use historical values to find the most suitable value regardless of mathematical formulas.

A pseudo algorithm flowchart for the genetic algorithm is shown in Pseudo Algorithm 4.2.

Pseudo Algorithm 4.2 (Genetic Algorithm).

```
1. Collect a population of random candidate solutions.
2. For each generation until conditions met:
   (a) Create new empty population (next generation).
   (b) While the new population is not full:
        i. Choose two solutions at random from the population,
           where the fitness determines the likelihood of
           being picked.
       ii. Cross-over the two solutions to produce two new.
   (c) Let each solution in the new population have a random
       chance to mutate.
   (d) Replace the population with the new population.
3. Select the solution with the highest fitness.
```

For more details and examples, see [54]

5
Data

The historical option price used in this thesis is written on Nifty 50, a weighted index for the 50 largest companies listed on India's National stock exchange [37]. The data set provides information about maturity date T, strike price K, and observed stock price S at T. In addition, we have complemented the option prices with a proxy for risk–free income and the past 30 days from T of stock prices. For the risk-free proxy r, we use the Indian 10-year bond yield [16]. In order to find the best proxy for r, we use the yield that is closest to maturity T.

For the option pricing models presented in section 2.2 we need to adjust the data to fit with the underlying assumptions of the financial framework. Furthermore, Anders et al. [56] and Stark [31] suggest using exclusion criteria to remove the ”non-representative” options of the market. These would, for example, be illiquid or extraordinary option circumstances. However, these criteria will not be applied since this thesis aims to see the actual real-world application. Consequently, the data was only cleaned enough not to violate any underlying assumptions, see section 2.2, i.e., not contradicting any of the model assumptions. The data used by the neural network will not be cleaned at all and just presented in its entirety.

The data set contains around 2.2 million European-style options between the periods 2020-06-01 and 2018-06-01, where roughly half are call options, and the other half are put options.

For the partition of the data into training and test data, we use 5% of the data set as test data, and the remaining 95% is used as training data. Note, the partitioning to test data is based on maturity data T. Thus, no future data can be included in the training data.

6 Results

This chapter contains the result of the thesis. In the first section, we define the general NN option pricing architecture. In section 6.2, the calibrated parameters for each method are presented. In section 6.3, the error metrical analysis is presented, concurrently with a definition of each metric. Lastly, section 6.4 presents the empirical frequency distribution of log-returns and resulting volatility smile fit.

6.1 Neural Network Architecture

With the acquired underlying theory of neural networks, we introduce the overall pricing network structure (see Fig. 6.1). The network configuration can be separated into two parts. The first part of the network will be one of the three neural networks presented in section 3. The second part of the network is a fully connected network, or an MLP, see section 3.2, which uses the input from the first part of the network as a feature to train on, together with time to maturity, risk-free rate, and strike price. The second part of the network then outputs the computed price of the option.

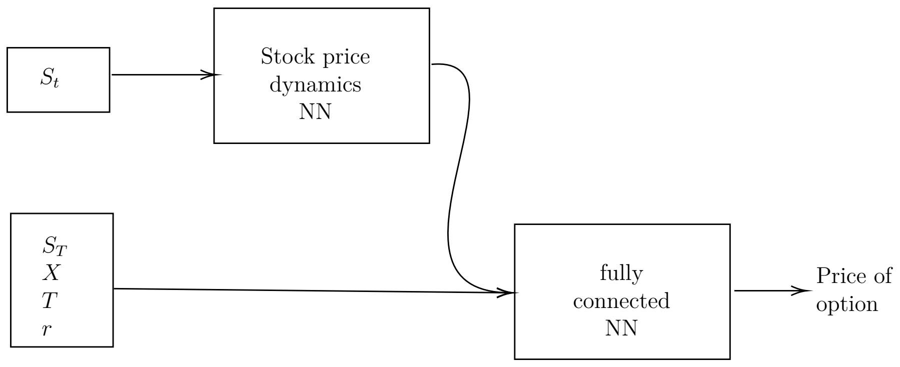

Figure 6.1: General neural network architecture for pricing an option.

6.2 Calibration

We have two structurally different option pricing methods that ought to be calibrated:

1. mathematical option pricing models, and
2. neural networks.

The calibration methods are described in section 4. Let us start with the calibration for the neural networks.

6.2.1 Neural Networks

The option pricing network architecture contains two separated neural networks, implying two sets of hyperparameters ought to be calibrated, one for each part. One could theoretically first calibrate the stock price dynamic network on the historical stock prices and then use the resulting parameters for the complete network and calibrate the second part of the network on option prices. However, these should be calibrated in conjunction since the output depends on both parts of the network simultaneously. Furthermore, the calibrated parameters for the first part of the network do not necessarily need to be the best calibrated parameters for the overall network performance.

We note that the put-call parity theoretically explains the relationship between calls and puts, one could therefore assume that the calibrated hyperparameters for one option type should be viable for the other type of the option — this relies on the notion that a neural network should learn such a relationship. However, as the purpose of this thesis is to investigate an alternative for option pricing that uses no underlying assumptions, we separate the calibration by calls or puts. In addition, the choice to separate between the two option types allows for a comparison in performance for each option type.

Each neural network have an unlimited number of possible hyperparameters combinations; we, therefore, make some simplifying restrictions to make the calibration less computation-intensive:

1. batch size is 4900,
2. ADAM is used as optimization algorithm[1],
3. loss function is a mean-squared error,
4. learning rate is 0.0001,
5. the activation functions is a ReLU (see equation (3.3)), and
6. each hidden layer has the same number of neurons.

The resulting sets of calibrated hyperparameters is presented in table 6.1.

	Stock Price dynamics NN		fully connected NN	
Network type	Layers	Neurons	Layers	Neurons
LSTM	4	8	11	3008
MLP	3	147	5	100
CNN	6	38	12	673

Table 6.1: Calibrated hyperparameters for each neural network type.

[1]The ADAM optimization algorithm is an extension to the stochastic gradient descent method, which is used to change the attributes of your neural network.

6.2.2 Mathematical Option Pricing Models

For the mathematical option pricing models, we can see the resulting calibrated parameters in table 6.2.

		σ	λ	k	δ^2	ρ	κ	θ	v_0
Call	BS	0.23							
	BS[1]	0.25							
	Merton	0.19	1.01	1.70	0.45				
	Merton[1]	0.13	0.06	0.51	0.03				
	Heston	0.34				0.91	0.22	0.15	0.05
	Heston[1]	0.46				-0.16	0.09	0.23	0.07
	Bates	0.80	0.70	0.14	0.01	0.91	0.92	0.01	0.08
	Bates[1]	0.76	2.10	0.04	0.07	-0.91	0.37	0.06	0.19
Put	BS	0.24							
	BS[1]	0.25							
	Merton	0.16	0.96	1.06	0.32				
	Merton[1]	0.13	0.06	0.51	0.03				
	Heston	0.41				0.78	0.31	0.22	0.08
	Heston[1]	0.46				-0.16	0.09	0.23	0.07
	Bates	0.80	0.61	0.19	0.04	0.94	0.56	0.03	0.06
	Bates[1]	0.76	2.10	0.04	0.07	-0.91	0.37	0.06	0.19

[1]Calibrated with the genetic algorithm.

Table 6.2: Calibrated parameters for each option pricing model.

Recall that σ is the volatility, λ is the intensity, δ^2 is the variance of the jumps, ρ is the correlation between $W_{1,t}$ and $W_{2,t}$, k is the mean jump size conditional on the jump occurring and v_0 is the initial variance of the asset.

Note that the parameters are not the same for the two calibration methods, given any of the pricing models, which is not enough to make any reasonable comparison. Although the parameters in each model are intended to capture some observable behavior in the market, one could make an argument for which model that is the most reasonable but that entails assumptions on financial markets and assets, which contradicts the purpose of the thesis. Thus, we strictly look at the metrics and do not apply any financial intuition to the results.

6.3 Error Analysis

Our error analysis for all models are reported in Table 6.3, where the error metrics, with θ denoting the actual data point and $\hat{\theta}$ denoting the predicted data point, are defined as follows;

$$\text{Root mean square error } (RMSE) = \sqrt{\mathbb{E}((\hat{\theta}-\theta)^2)}, \tag{6.1}$$

$$\text{Absolute percentage error } (APE) = 100 * \left|\frac{\hat{\theta}-\theta}{\theta}\right|, \tag{6.2}$$

$$\text{Average absolute percentage error } (AAPE) = 100 * \mathbb{E}\left(APE\right), \tag{6.3}$$

$$\text{Median absolute percentage error } (MAPE) = 100 * \text{median}(APE), \tag{6.4}$$

$$\text{Median percent error (BIAS) } = 100 * \text{median}(\frac{\hat{\theta}-\theta}{\theta}), \tag{6.5}$$

$$\text{Percent error within the } \pm X\% \text{ of the actual price } (PEX) = \frac{100}{n} * \sum_{i:\, APE<X}^{n} i. \tag{6.6}$$

From Table 6.3, we can select the best performing neural network and mathematical option pricing model for call options.

Error metric	Neural network	Option pricing model	$\frac{\text{Option pricing model}}{\text{NN}}$
RMSE	LSTM	Bates	67.3%
BIAS	LSTM	Bates	88%
APPE	LSTM	BS[1]	8.9%
MAPE	LSTM	Heston	74.9%
PE5	LSTM	Merton	28.8%
PE10	LSTM	Heston	36.6%
PE20	LSTM	Bates	49.9%

Table 6.4: Best performing neural network and option pricing model for pricing European styled call options.

The result in table 6.4 suggests that the LSTM is the best performing neural network, regardless of included metric, while the best performing option pricing model varies for each metric. The comparison between the two best performers indicates that the best performing neural network, the LSTM, is significantly better performing, with a minimum of 9% less error in the $APPE$ and a maximum of 88% less BIAS. For the capturing metrics, we note that the best performing option pricing model capture 28% less of the options within ±5%. In addition, we note that the gap in capture performance increases towards the tails, indicating that the LSTM is better at capturing out-of-money options, deep in the money options, and options with extraordinary conditions. Furthermore, as the training $RMSE$ is close to the testing $RMSE$ there is no reason to suspect any overtraining for the neural networks.

		LSTM	CNN	MLP	BS	BS[1]	Merton	Merton[1]	Heston	Heston[1]	Bates	Bates[1]
Call	RMSE	234.14	299.17	360.21	1001.80	952.66	916.46	928.71	901.49	864.65	714.77	960.87
	RMSE (training)	234.42	249.11	374.79								
	BIAS	0.19%	2.20%	4.64%	20.69%	16.51%	-6.68%	3.80%	-18.66%	-1.70%	-1.55%	-12.95%
	AAPE	50.05%	51.83%	1427.12%	58.37%	54.95%	63.52%	59.50%	202.61%	60.35%	292.58%	90.99%
	MAPE	9.62%	14.15%	17.94%	57.45%	50.93%	38.97%	44.83%	38.46%	39.33%	33.98%	40.43%
	PE5	37.09%	31.02%	17.47%	9.20%	9.52%	10.67%	10.01%	10.22%	7.56%	8.20%	9.95%
	PE10	50.67%	43.34%	33.20%	15.52%	16.15%	18.32%	17.16%	18.55%	16.58%	17.60%	17.04%
	PE20	64.92%	56.93%	53.18%	25.85%	27.10%	31.49%	29.16%	31.73%	25.70%	32.40%	29.61%
Put	RMSE	230.15	298.23	356.01	1000.89	960.11	926.45	940.54	900.12	890.73	821.35	970.32
	RMSE (training)	230.02	267.66	344.83								
	BIAS	0.22%	2.23%	5.1%	19.17%	16.04%	-3.55%	3.92%	-18.80%	-1.28%	-2.03%	-10.62%
	AAPE	49.45%	51.30%	1410.38%	56.17%	53.07%	61.94%	58.22%	206.78%	61.63%	282.41%	92.04%
	MAPE	9.34	15.67%	14.08%	58.97%	48.10%	34.87%	47.60%	36.74%	39.41%	32.84%	41.05%
	PE5	37.81%	31.62%	17.89%	9.68%	9.44%	10.67%	10.01%	10.22%	7.61%	8.16%	10.67%
	PE10	52.05%	42.26%	33.77%	16.17%	16.03%	18.32%	17.16%	18.55%	17.09%	17.57%	18.19%
	PE20	60.11%	55.13%	54.96%	26.65%	26.99%	31.49%	29.16%	31.73%	26.22%	31.78%	30.31%

[1]Calibrated with the genetic algorithm.

Table 6.3: Error metric result for comparing four pre-defined option pricing models, see section 2.2, with three pre-defined neural network types, see section 3, for pricing European-style call- and put options

Analogously, we can make a similar comparison for the put options, see Table 6.5.

Error metric	Neural network	Option pricing model	$\frac{\text{Option pricing model}}{\text{NN}}$
RMSE	LSTM	Bates	71.9%
BIAS	LSTM	Heston[1]	88.8%
APPE	LSTM	BS[1]	6.8%
MAPE	LSTM	Bates	73.2%
PE5	LSTM	Bates[1]	28.2%
PE10	LSTM	Heston	35.61%
PE20	LSTM	Bates	52.8%

Table 6.5: Best performing neural network and option pricing model for pricing European styled put options.

Similarly to call options, the result in table 6.4 suggests that the LSTM is the best performing neural network, regardless of included metric, while the best performing option pricing model varies for each metric. The comparison between the two best performers indicates that the best performing neural network, the LSTM, is significantly better performing, with a minimum of 6.8% less error in the $APPE$ and a maximum of 88.8% less BIAS. For the capturing metric, we note that the best-performing option pricing model capture 28.2% less of the options within $\pm 5\%$. Similarly to the call options, we note that the gap in capture performance increases towards the tails, indicating that the LSTM is better at capturing out-of-money options, deep in the money options, and options with extraordinary conditions. Also, as the training $RMSE$ is close to the testing $RMSE$ there is no reason to suspect any overtraining for the neural networks.

Note that an option is a leveraged financial instrument, meaning that a shift in the price of the underlying asset can induce a much larger shift in the price of the option. Thus, the financial effects of pricing an option wrongly could be highly leveraged, and thus potentially more harmful to investors, making any small error in pricing a potentially large financial loss.

We observe that the two different calibration methods make a difference in the resulting parameters. However, one should note that the calibration with non-linear least squares was calibrated on a sample of the data and extrapolated, which means that the overall performance could potentially be lowered. Although the error metrics could be potently lowered, it is not realistic to implement such a method in a real-world environment since the computation is too expensive for implementation. Hence, the genetic algorithm, although not always better performing, is more robust, not as expensive, and does not need to be calibrated on any labeled data.

The overall result of the error analysis indicates that the use of neural networks in option pricing significantly outperforms the historical mathematical option pricing models, both for call and put options. Comparing the best performing NN for calls and puts indicates a slight advantage for the pricing put option but not large enough to make any conclusions.

6.4 Empirical Frequency Distribution of Log-returns and Volatility Smile

In addition to error metrical analysis, we also investigate how the neural networks perform in terms of stylized facts, more specifically; the distribution of the log-returns and the volatility smile fitting.

For the empirical frequency distribution, we illustrate the resulting frequency distribution of log-returns for each method and the true distribution as a reference point in Fig. 6.2.

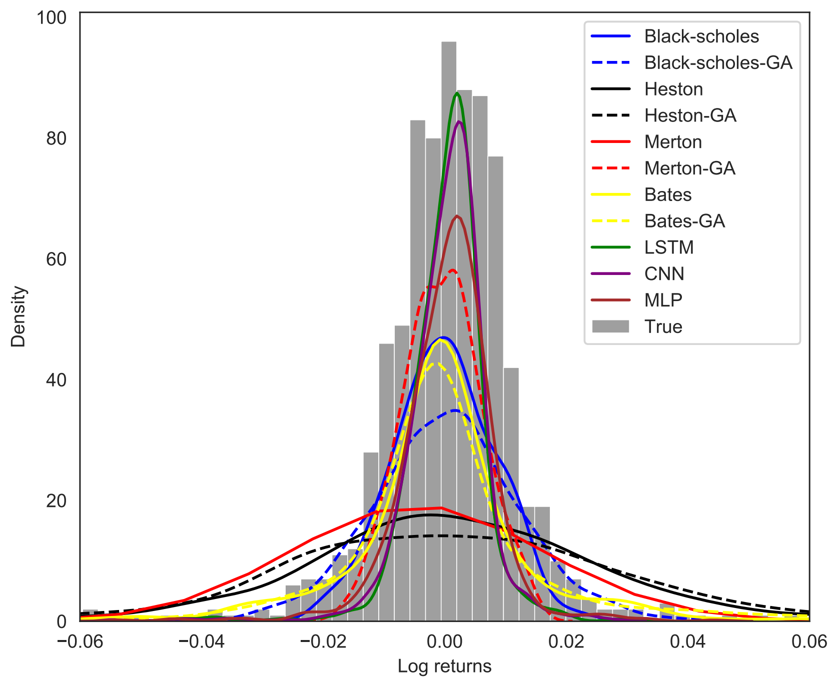

Figure 6.2: The empirical frequency distribution of log-returns for the NIFTY 50 index prices against each theoretical prices.

For a more concrete metrical comparison, we use the sum of squared residuals (SSR):

$$SSR = \sum_{i=i}(\hat{\theta}_i - \theta_i)^2. \tag{6.7}$$

In what follows, we see the resulting SSR for each method in table 6.6.

	SRR
Black-Scholes	0.197
Black–Scholes[1]	0.274
Merton	0.634
Merton[1]	0.736
Heston	4.169
Heston[1]	0.184
Bates	2.419
Bates[1]	1.118
LSTM	0.175
CNN	0.180
MLP	0.208

Table 6.6: Sum of squared residuals for the true log-returns of NIFTY50 against the theoretical log-returns of each option pricing method.

From table 6.6, we note that LSTM is the best performer which might be as expected as the network is created to learn order dependence in sequences [39, 42]. Also, on average, the NN outperforms the classical option pricing models in terms of fit to the true values. Moreover, we can see from Fig. 6.2 that option pricing models with incorporated jumps have heavier tails, that is Merton and Bates, than their counterparts, which is to be expected since they have been modeled to have larger kurtosis. In addition, as the stylized facts of stock dynamics support heavy tails characterization of the frequency distribution of log-returns rather than a normal distribution, see section 2.2, it seems reasonable to introduce jumps. However, this might not always be the best-performing model. More specifically, we note that both the Merton and the Bates model perform worst, in terms of SSR, out of all methods. In addition, the jump models do not perform best in the overall error analysis either. Thus, we note that coherence to stylized facts does not necessarily determine the overall performance.

For the IV smile fitting, let's assume that the market prices come from the use of the Black–Scholes model. Then, we can use that the Black–Scholes model has a closed analytical formula for pricing calls and puts. Thus, equation (2.7) can be used to choose the implied volatility that minimizes the difference between the theoretical price and the true value. Note that all input values in the BS models are known from the data, i.e., we can fix all values and only let implied volatility be the unknown.

Recall that equation (2.7) gives the call price under the BS approach,

$$C = x\Phi(d_+(\tau, x)) - e^{-r\tau}K\Phi(d_-(\tau, x)). \tag{6.8}$$

We want to minimize

$$\sigma_i^{IV} = \arg\min_{\sigma_i \in \Psi}(C_i^m - C_i)^2, \tag{6.9}$$

where C_i^m is the calculated theoretical price of method m for option i and C_i are the BS calculated option price for option i. The bounds are arbitrarily set to $[0, 1]$ but the implied volatility could theoretically much lager than 1. The non-linear least squares method is then used to find the IV for each option.

As the volatility smile is a graphed form, we need to visualize the outcome. Thus, we choose a limited number of three randomly selected sets of fixed values, i.e option type, writing date, time to maturity.

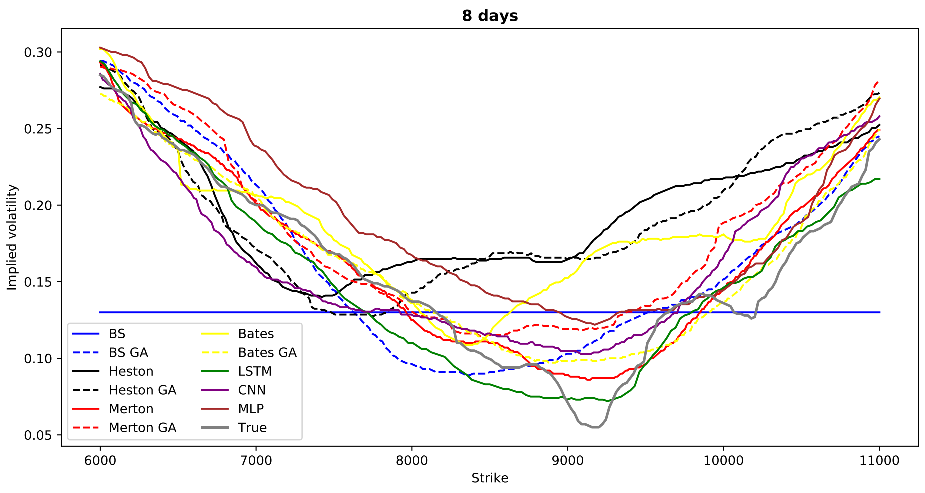

Figure 6.3: IV against strike price for all call options in the data set with 8 days to maturity and written on the same date.

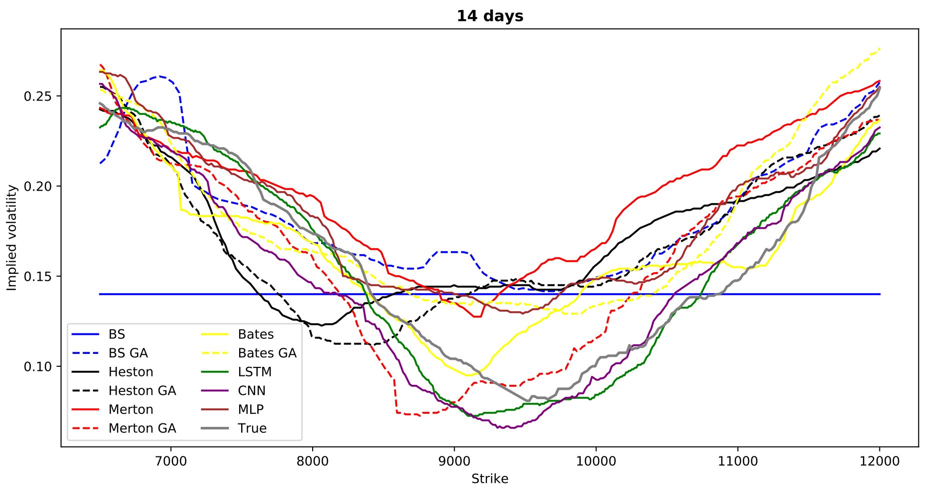

Figure 6.4: IV against strike price for all put options in the data set with 14 days to maturity and written on the same date.

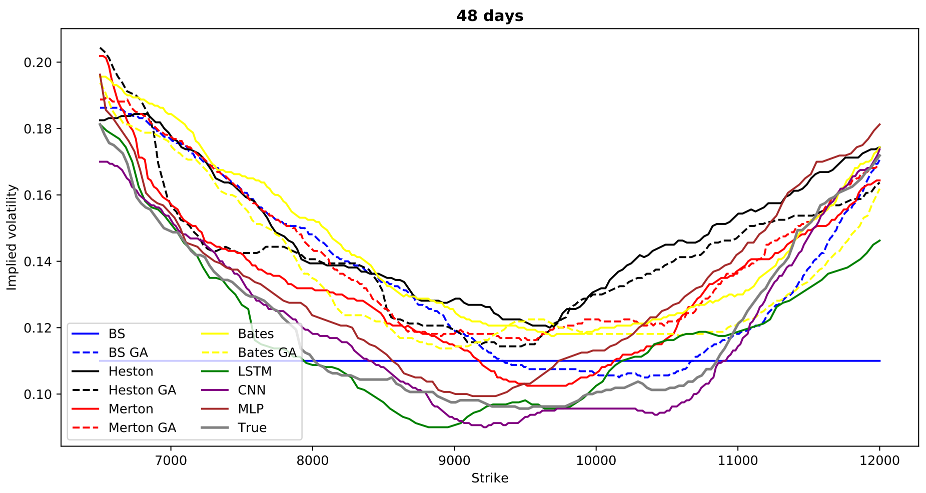

Figure 6.5: IV against strike price for all call options in the data set with 48 days to maturity and written on the same date.

From Fig. 6.3, 6.4 and 6.5, we note that all but one option pricing model, the BS model, seem to display a smile-shaped feature. The reason the BS model has a constant IV is due to the assumption that the BS price is the market price, which was necessary to calculate the IV of the other methods. Although the output of the methods could be considered relative close, it is important to remember how IV impacts options prices; a lower IV would make the option less expensive, and a higher IV would make the option more expensive. In addition, options are a leveraged financial instrument. Thus, incremental differences in IV could drastically affect the price.

One could also calculate a metric for the best fit to the true IV given by the data set. However, the true IV values are still a forecast of future volatility. Thus, there is no advantage to being close to the true IV if it was not correctly forecasted, which is unknown. Instead, the IV was only used to see if the NN priced options follow real-world market behavioral patterns.

7 Conclusion

In this thesis, we have compared the performances of NNs and mathematical option pricing models, in terms of fit to stylized fact and error metrical analysis. The four related mathematical market models are similar in many aspects, particularly because the financial assumptions behind them are largely the same. Financial assumptions have in the past been contradicted, see Section 1.1. The robustness of all mathematical option pricing models can therefore be questioned. Simultaneously, the NNs make no assumptions at all. However, NN is still viewed as a black box which makes them less reliable. Nevertheless, NNs are applied in different high-risk aspects of society, e.g. medicine, military, and self-driving cars [1, 57, 23]. Thus, one could assume that NNs could reach a high level of reliability.

It has been shown that in addition to being a more robust alternative, the NNs significantly outperforms its counterparts in terms of error metrics. One could argue that the lack of data cleaning meant that the mathematical models were not subjected to the best type of data. However, as the used data were actual market data, there is no reason to exclude it. In real-world practice, a pricing model should price the asset, independent of underlying market conditions. Moreover, the data was partly cleaned to fit the underlying assumption on the market and asset. For example, an option that posed as an arbitrage opportunity or was only traded on its intrinsic value was not allowed for mathematical models. Yet, all options included in the data set were still included in the NNs. Thus, in addition to outperforming the mathematical models, the NNs priced more options under more complex options conditions.

The NNs coherence to the chosen stylized facts further strengthens the NNs as a viable alternative, indicating that the NNs can learn market behaviors and asset structures. However, as discussed, coherence to stylized facts does not necessarily imply a better overall result.

8
Future work

In this section, some questions that have been raised during the work are mentioned and discussed as possible future work.

- NNs depend on the input data, which most often determines the performance, in terms of learning the input data patterns. In this work, we were able to use a large amount of historical data. However, for future research, it would be interesting to see the effects of including more data features, e.g. ESG data, related financial assets, and macro economical variables.
- Implementations of the NNs in a production environment seem promising, but as the presented alternatives are only a first prototype, there are a lot of improvements to be made with fine-tuning but also testing alternative NN types.
- The included option were European-styled but it would be interesting to see how NNs would perform when pricing more complex option styles, e.g. barrier options, Bermuda options and quantity-adjusting options.

Bibliography

[1] A. Egba and R. Okonkwo. Artificial neural networks for medical diagnosis: a review of recent trends, International Journal of Computer Science and Engineering Survey (2020).

[2] A. Graves and J. Schmidhuber. Framewise phoneme classification with bidirectional LSTM networks, In Proc. of the Int. Joint Conf. on Neural Networks (2005), volume 18, 2047-2052, Oxford, UK.

[3] A. Lewis. Option Valuation under Stochastic Volatility: With Mathematica Code, Finance Press (2000), Newport Beach.

[4] A. Lewis. A simple option formula for general jump-diffusion and other exponential Lévy processes, Envision Financial Systems (2001), California.

[5] B. Eraker. Do stock prices and volatility jump? Reconciling evidence from spot and option prices, Journal of Finance (2004), 59, 1367-1403.

[6] B. G. Malkiel. The efficient market hypothesis and its critics, Princeton University, CEPS Working Paper No. 91 (2003).

[7] B. Widrow and M.E. Hoff. Adaptive switching circuits. IRE Eastern Electronic Show and Convention, Convention Record 4 (1960), 96-104.

[8] B. Widrow and S.D. Stearns. Adaptive signal processing englewood cliffs, NJ: Prentice-Hall (1985).

[9] C. C. Aggarwal. Neural Networks and Deep Learning: A Textbook, Springer International Publishing (2018)

[10] C. Matonoha, J. Vlček and L. Luksan. Problems for nonlinear least squares and nonlinear equations (2018).

[11] D. Duffe, J. Pan, and K. Singleton. Transform analysis and asset pricing for affine jump-diffusions, Econometrica (2000), 1343-1376.

[12] D. Filipović. A general characterization of one factor affinene term structure models, Finance and Stochastics 5(3) (2001), 389-412.

[13] D.E. Rumelhart, G.E. Hinton and R.J. Williams. Learning internal representations by error propagation, parallel distributed processing: explorations in the microstructure of cognition, Foundation (1986), 318-362.

[14] D. S. Bates. Jumps and stochastic volatility: exchange rate processes implicit in Deutsche Mark Options, Review of Financial Studies 9(1) (1996), 69-107.

[15] E. Lukacs. Characteristic functions, 2ed ed., Charles Griffin and Co (1970), London.

[16] Federal Reserve Economic Data. Interest rates: long-term government bond yields: 10-year: main (including benchmark) for India (2020). Source: `https://fred.stlouisfed.org/series/INDIRLTLT01STM` (Gathered: 2020-05-20).

[17] E.C Titchmarsh. Introduction to the Theory of Fourier Integrals, Reprint of 2nd ed., Oxford University Press (1975), London.

[18] F. A. Gers, J. Schmidhuber, and F. Cummins. Learning to forget: continual prediction with LSTM, Neural Computation (2000), 2451- 2471.

[19] F. A. Gers, N. N. Schraudolph, and J. Schmidhuber. Learning precise timing with LSTM recurrent networks, Journal of Machine Learning Research (JMLR) (2002), 3(1):115-143.

[20] F. Black, and M. Scholes. The pricing of options and corporate liabilities, The Journal of Political Economy, 81, No. 3 (1973), 637-654, 10.1086/260062.

[21] F. Rosenblatt. The Perceptron: a probabilistic model for information storage and organization in the brain. Psychological Rev., 65 (1958), 386-408.

[22] F. Rosenblatt. Principles of Neurodynamics, New York: Spartan Books (1962).

[23] G. Biggi and J. Stilgoe. Scientometric and bibliometric analysis artificial intelligence in self-driving cars research and annovation: a scientometric and bibliometric analysis (2020).

[24] G. Cybenko, Approximation by superposition of a sigmoid function. Math. of Contr., Signals and Syst., 2 (1989), 303-314.

[25] I. Karatzas and S.E Shreve. Brownian Motion and Stochastic Calculus, Springer-Verlag (1988).

[26] J. Hull. Options, Futures and Other Derivatives (5th ed.), Prentice Hall.

[27] J. Gil-Pelaez, Note on the inversion theorem, Biometrika 38(3-4) (1951), 481-482.

[28] J. L. Elman. Finding structure in time, Cognitive Science (1990), 14(2):179-211.

[29] J. Ruiz, J. Pérez and J. Blázquez. Arrhythmia detection using convolutional neural models, Int. Symp. Distrib. (2018).

[30] J. W. Goodell, S. Kumar, W. M. Lim, D. Pattnaik. Artificial intelligence and machine learning in finance: identifying foundations, themes, and research clusters from bibliometric analysis, Journal of Behavioral and Experimental Finance (2021).

[31] L. Stark. Machine learning and options pricing: a comparison of Black–Scholes and a deep neural network in pricing and hedging dax 30 index options. 2017.

[32] L.A Waller, Does the characteristic function numerically distinguish distributions, The American Statistician 49(2) (1995), 150-152.

[33] L.C.G. Rogers and D. Williams. Diffusions, Markov processes and martingales, Cambridge Mathamatical Library (2000).

[34] M. French. Crude oil prices briefly traded below $0 in spring 2020 but have since been mostly flat, U.S. Energy Information Administration (2020).

[35] M. I. Jordan. Attractor dynamics and parallelism in a connectionist sequential machine, In Proc. of the Eigth Annual Conf. of the Cognitive Science Society (1986), 531-546.

[36] M. Zihlmann, D. Perekrestenko, M. Tschannen, Convolutional recurrent neural networks for electrocardiogram classification, Computing (2017).

[37] National stock exchange, India.
Source: `https://www.nseindia.com/option-chain` (Gathered: 2020-05-23).

[38] P.J. Werbos. Beyond regressions: new tools for prediction and analysis in the behavioral sciences, Harvard University (1974).

[39] P.J. Werbos. Backpropagation through time: What it does and how to do it. Proc. of the IEEE (1990), 1550-1560.

[40] R. Jozefowicz, W. Zaremba, and I. Sutskever. An empirical exploration of recurrent network srchitectures, Int. Conf. on Machine Learning (2015), 2342-2350.

[41] R.C Merton. Option pricing when underlying stock returns are discontinuous, Journal of Financial Economics, 3 (1976), 125-144.

[42] R. J. Williams and D. Zipser. A learning algorithm for continually running fully recurrent neural networks, Neural Computation (1989), 1(2):270- 280.

[43] R. J. Williams and D. Zipser. Gradient-based learning algorithms for recurrent networks and their computational complexity, In Back-propagation: Theory, Architectures and Applications (1995), 1-45. L. Erlbaum Associates Inc.

[44] P. Christoffersen, K. Jacobs, C. Ornthanalai, and Y. Wang. Option valuation with long-run and short-run volatility components, Journal of Financial Economics (2008), 90, 272- 297.

[45] P. Lévy. Calcul des probabilités, Gauthier-Villars et Cie (1925), Paris.

[46] S. Haykin and M. Moher, Introduction to analog and digital communications (2nd ed.), John Wiley and Sons, Inc. (2007).

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

[48] S. Hochreiter, A. S. Younger, and P. R. Conwell. Learning to learn using gradient descent. In G. Dorffner, H. Bischof, and K. Hornik, Proc. of the Int. Conf. of Artificial Neural Networks, Springer Berlin Heidelberg (2001).

[49] S. Hochreiter and J. Schmidhuber. Long short-term memory, Neural computation (1997), 9(8):1735-1780.

[50] S. Hochreiter and J. Schmidhuber. LSTM can solve hard long time lag problems, Neural Information Processing Systems (1997), 473-479.

[51] S. Hochreiter, Y. Bengio, P. Frasconi, and J. Schmidhuber. Gradient flow in recurrent nets: the difficulty of learning long-term dependencies, Wiley-IEEE Press (2001), page 15.

[52] S. E. Shreve. Stochastic Calculus for Finance II: Continuous-Time Models, Springer, USA (2004).

[53] S. Kiranyaz, T. Ince, R. Hamila, M. Gabbouj. Convolutional neural Networks for patient-specific ECG classification, Proc. Annu. Int. Conf. IEEE Eng. Med. Biol. Soc. EMBS (2015).

[54] S. Thede, An introduction to genetic algorithms, Journal of Computing Sciences in Colleges (2004).

[55] T. Szandała. Review and comparison of commonly used activation functions for deep neural networks (2021).

[56] U. Anders, O. Korn, and C. Schmitt. Improving the pricing of options: a neural network approach, Journal of Forecasting, 17(5-6):369-388, 1998.

[57] W. P. WEBSTER, Artificial neural networks and their application to weapons, Naval Engineers Journal (1991)

[58] W. Rudin. Functional analysis, International Series in Pure and Applied Mathematics. Vol. 8 (2rd ed.), McGraw-Hill (1991).

[59] W. Rudin. Real and Complex Analysis (International Series in Pure and Applied Mathematics, 3rd ed.), McGraw-Hill (1987).

[60] Y. Bengio, P. Simard, P. Frasconi, and P. Frasconi Yoshua Bengio, P. Simard. Learning long-term dependencies with gradient descent is difficult, IEEE trans. on Neural Networks / A publication of the IEEE Neural Networks Council (1994), 5(2):157-66.

[61] X. Xu and S. Taylor. The term structure of volatility implied by foreign exchange options, Journal of Financial and Quantitative Analysis (1994), 29, 57-74.

[62] Z. Michalewicz. Genetic algorithms + data structures = evolution programs, Springer-Verlag (1992).

A
Source code

The source code is available at:
`https://github.com/dunderlime/OptionPricingProject`