UNIVERSITY OF BIRMINGHAM

BIRMINGHAM BUSINESS SCHOOL

2018-2019

MSc DISSERTATION COVER SHEET

I confirm that I have read and understood the regulations on plagiarism* and have acknowledged the work of others that I have included in this dissertation.

Please read the following statement and **tick ONE box** regarding permission, or denial thereof, to view your dissertation by other students:

- **I AGREE** to allow my dissertation to be seen by future students. ☑

By signing this form, I agree to allow access to students of the Business School, as part of the University of Birmingham, to view my dissertation, or part thereof, for guidance as an example of good practice. For its part, the University will grant access to Birmingham Business School students as it deems appropriate, but in so doing forbids anyone to copy or use my dissertation in any other way or for any other purpose.
I understand that my dissertation will be available to view via Canvas and that any personal references will be anonymised.
I further understand that the University has no control over the actions of third parties, and should I have any concerns, my permission may be withdrawn, at any time, by advising the Business School in writing.

- **I DO NOT AGREE** to allow my dissertation to be seen by future students. ☐

Print Name: Dalvir Singh Mandara

Student ID: 1423101

Date: 11/09/2019……
---

*Plagiarism, in this context, is the reproduction of material from books and articles without acknowledgement. It is the act of passing off another person's work as your own, copying a fellow student's work or reproducing work submitted by a past student. Such actions are seen as a form of cheating and, as such, are penalised by examiners according to their extent and gravity.

You should not quote existing work without quotation marks and appropriate referencing. An attempt to present the work of someone else as your own may lead to your dissertation being awarded a mark of zero. You are required to state the full references of all sources that you use. If quotations are made, they must be explicitly and fully referenced, including stating the relevant page number(s). You will be penalised very severely if examiners find that you have presented a section of a book, an article or a paper without appropriate referencing. If you are not sure about how to quote an existing work, please ask for advice from your supervisor.

Artificial Neural Networks for Black-Scholes Option Pricing and Prediction of Implied Volatility for the SABR Stochastic Volatility Model

DALVIR MANDARA

MSc Mathematical Finance - 2018/19

UNIVERSITY OF BIRMINGHAM

Supervisor: Dr Daniel J. Duffy
Student ID: 1423101

A thesis submitted in partial fulfilment of
the requirements for the degree of
MSc Mathematical Finance

School of Business
School of Mathematics
University of Birmingham

11 September 2019

Abstract

We present an artificial neural network (ANN) based framework for pricing options under the Black-Scholes model whilst ensuring that the network preserves the first and second derivatives and the positivity of option prices. This is done by taking into account the order of continuity of the activation functions used on the network and ensuring output activation functions can only take values on the positive domain. We also present an ANN based framework for predicting the implied volatility generated by the SABR stochastic volatility model with an extension to an image-based implicit learning method used to predict the SABR implied volatility surface. Explicit formulae exist for the options and the SABR implied volatility expansions we consider meaning we are able to generate a large amount of data to robustly test our methods.

The experiments show that our ANN architectures are able to accurately predict the price of calls and puts under Black-Scholes as well as distinguish between the two pricing formulae based on an input flag. Additionally, our ANNs are able to accurately predict both individual implied volatility and an implied volatility surface under the normal and lognormal SABR regimes. In the analysis we also conclude that the ANNs are able to generate a volatility surface faster than repeatedly evaluating the SABR expansion for each point on the surface.

Acknowledgements

I would like to express my gratitude to Dr Daniel J. Duffy for the continued guidance, mentoring and fruitful discussions throughout the duration of this thesis, without his expertise and resourceful input this project would not have been possible. Secondly, I would like thank my course director, Dr Colin Rowat, for assembling such a challenging but rewarding masters program. Last but not least, I am extremely grateful to my family and friends for their continued support.

Contents

CHAPTER 0

The Big Picture

When tackling any problem it is useful to sub-divide the process into actionable steps, this is the approach we take for this thesis. We define 4 main stages:

- Define the goal.
- Formulate the procedure.
- Execute the plan.
- Evaluate the results.

The goal is to investigate the feasibility of artificial neural networks for option pricing problems as well as their use for predicting implied volatility for stochastic volatility models (we consider SABR). Feasibility in our context includes the accuracy of the neural networks on unseen test data and run-time performance. Formulating the procedure involves the data generation and preparation process, the choice of computer language/resources and the ANN design patterns. Deployment of the neural network followed by numerical results and error analysis form the latter 2 steps of the 4 above.

Figure 0.1 shows how we approached the research, starting with selecting a financial model, then learning an ANN and processing the output. The outline is kept as general as possible to serve as a blueprint for potential future work, specifics involved at each stage include the data generation method, the scaling/normalisation methods and the ANN training regime (pointwise or image-based). Only the specifics at each step change depending on the context and the goal of the networks in each of our experiments. Initially, C++ was the language of choice for this thesis. However, it quickly became apparent that the main C++ neural network libraries (OpenCV, tiny-dnn etc) did not have the flexibility for customising key elements of

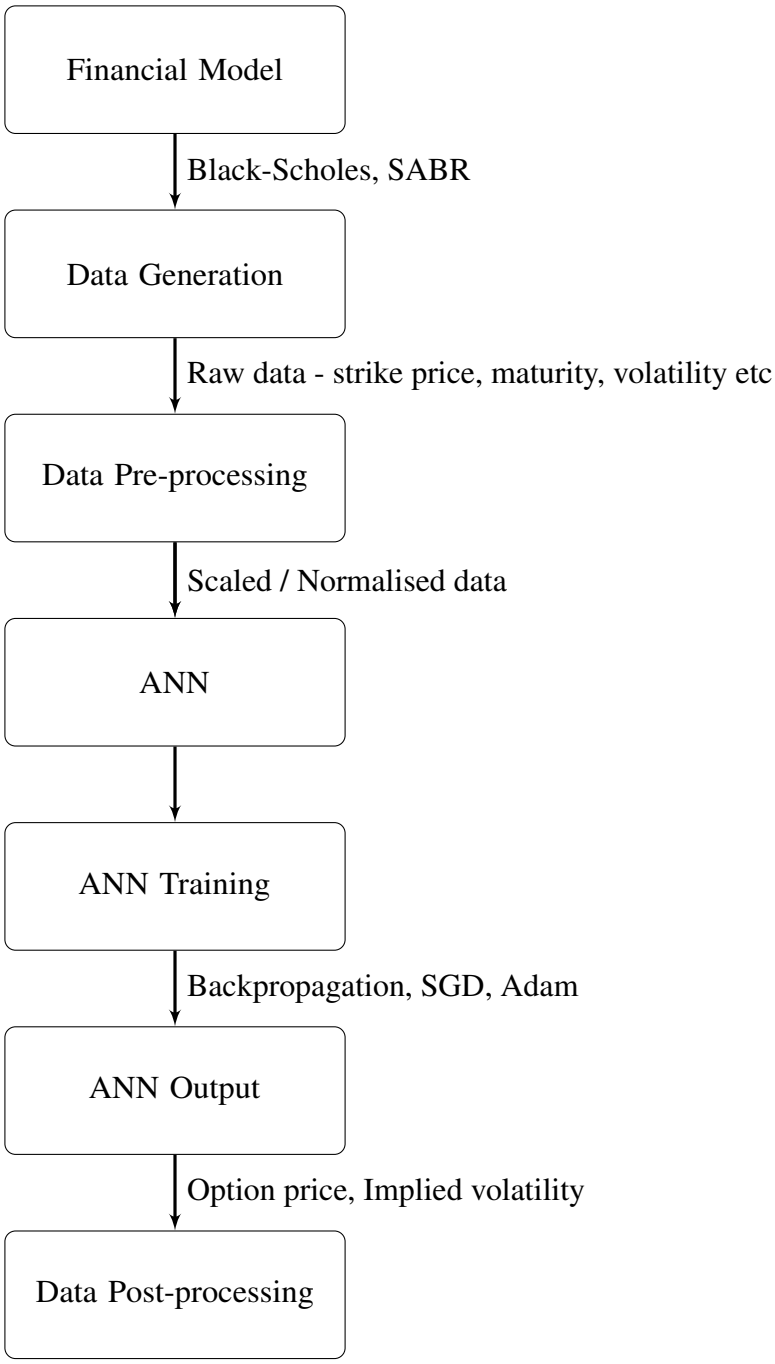


FIGURE 0.1. *Methodology Flow Diagram.*

the networks. After a discussion with Daniel Duffy (thesis supervisor) it was decided that Python would be a more suitable choice, making use of the Keras deep learning library.

CHAPTER 1

Introduction

Recent advances in research regarding the use of artificial neural networks to capture some of the extremely complex non-linear functional relationships between state variables in the market, option prices and implied volatility surfaces provide the the motivation for this thesis. Machine learning in the more general sense has already been at use for a long time in quantitative finance, some of its earlier uses include the domain of financial time series prediction. The Efficient-Markets hypothesis states that under the assumption of perfect information it should be impossible to beat the market by predicting how a market will move at some point in the future (Malkiel 1989), however, machine learning techniques have long since been used to successfully predict how certain markets will change with remarkable accuracy.

The use of artificial neural networks (ANNs) to learn complex functional mappings between observed market variables in order to supplement more traditional methods is a hot topic due the vast availability of data and potential monetary gains. Some benefits of using ANNs to represent functions such as option pricing and implied volatility functions is that they can allow for faster functional evaluation due to the deterministic relationship between inputs and outputs obtained by the networks as opposed to other numerical methods. The power of ANNs in this domain is underpinned by a pioneering result of George Cybenko known as the Universal Approximation Theorem (Cybenko 1989). The Theorem essentially states that any n-dimensional function that is continuous on the domain $[0, 1]^n$, can be approximated by a neural network with just one hidden layer to a degree of accuracy controlled by any $\epsilon > 0$. We make use of this result with extensions to multiple layers in order to investigate the feasibility of neural networks for option pricing problems and representing the functional

form of the implied volatility for the SABR (Hagan et al. 2002) stochastic volatility model. We subsume, and build on, the design patterns of (Horvath et al. 2019) who introduce the idea of image-based implicit learning for the prediction of volatility surfaces. Much of the current literature has focused on a more traditional pointwise regime in which model parameters are mapped to a single value such as an implied volatility. The image-based implicit learning method instead maps inputs to the neural network to a grid of 'pixels' which can thought of as an image where each 'pixel' represents a particular implied volatility. The authors (Horvath et al. 2019) have shown that this results in being able to use a much simpler network to get accurate results which leads to quicker training and ultimately more efficiency.

In (McGhee 2018) it was shown that a neural network with a single hidden layer is able to accurately represent the SABR approximation for implied volatility (Hagan et al. 2002). This thesis will instead investigate the use of the image-based implicit learning method for the SABR model using a neural network with 3 layers and a slightly more nuanced network architecture. Whilst the use of ANNs for option pricing and approximating other complex functions in finance has been promising, (Itkin 2019) draws attention to some pitfalls in the current implementations, such as lack of consideration in ensuring that the activation functions on the networks are differentiable at least twice to allow for option Greeks to be calculated. We address some of these issues and implement some recommended solutions as part of the Black-Scholes framework as a proof of concept.

The remainder of this thesis is structured as follows: Chapter 2 reviews some important literature. In Chapter 3 the financial models considered in this paper will be introduced: Black-Scholes and SABR. Chapter 4 introduces neural computation and explains their fundamentals along with some important theorems regarding the power of depth (Eldan and Shamir 2016) and universal approximation (Cybenko 1989). Chapter 5 explains our data sources and generation methods, this is followed by chapter 6 where we detail our network architectures. In chapter 7, the results of our implementations are examined and finally Chapter 8 concludes the findings along with a discussion of potential future work in this area.

CHAPTER 2

Literature review

Much of the power of neural networks for quantitative finance boils down to their speed and efficiency. An important emphasis in the literature is put on the fact that these neural networks can be trained offline meaning the training phase does not enter the total computation time for valuing options, calibrating models or computing implied volatilities. This means evaluating the functional relationship learned by the neural network in real time for a particular set of input parameters is extremely fast. This has resulted in bold claims such as the work by (McGhee 2018) on a neural network representation of the SABR model being 10,000 times faster and more accurate than a corresponding finite difference scheme. (Ferguson and Green 2018) focused on the use of deep neural nets to value European basket options and reported a remarkable 1 million times speed up compared to a Monte-Carlo method along with more accurate results. The strength of these results are hinged upon the ability of the networks to accurately represent the pricing function at hand.

In the context of option pricing (Itkin 2019) has brought to light some subtle issues regarding the way neural networks are used in the current literature. One of the key aspects he touches upon is the necessity for quants and risk managers to have access to option Greeks in order to investigate sensitives and for hedging activities. An important result by (Hornik et al. 1990) essentially states that if a neural network uses an activation function that is n-times differentiable to approximate some function G that is also n-times differentiable, then the network approximates G and all its n derivatives. For option Greeks we require the first and sometimes second derivatives to exist meaning that these properties should also be shared by the activation functions used on a network to price an option. Consider the work discussed above by (Ferguson and Green 2018) using deep neural nets for basket option pricing, the

authors used the ReLU activation function. Whilst this function is popular due to its fast convergence on the network, it is not differentiable at the point 0 and so its first derivative is not smooth which could create problems for the calculation of option Greeks. Another key issue (Itkin 2019) brings up is the arbitrage-free condition required for option pricing, this brings constraints into the picture which have commonly been ignored in the literature, consequently, such networks may not guarantee arbitrage-free pricing. The constraints also make it harder to implement the neural networks in practice as often complex loss functions are required in order to handle them and the networks may still not guarantee arbitrage-free pricing out of sample.

(Horvath et al. 2019) applied 3-layer neural networks to a variety of stochastic volatility models. The authors use what they call image-based implicit learning where multiple implied volatilities (the target variable) characterised by the strike price and the maturity date are used to train the network at the same time as opposed to the more traditional method of considering one implied volatility at a time, moving from the objective of predicting implied volatility to predicting an implied volatility surface.

The calibration problem can be thought of as selecting a particular set of model parameters in such a way that minimises the difference between market quoted prices and the price the model generates for a given set of market data. As a minimal example consider the Black-Scholes model where the only parameter to calibrate would be the volatility parameter. (Liu et al. 2019) use what they refer to as CaNN (calibration neural networks) to calibrate models by splitting the calibration process along the neural network into two steps: a forward pass where the pricing equation is learnt (via learning of the neural network weights and biases) and a backward pass where those same weights and biases are used again except the objective is now to learn the previously un-learnable input parameters given output data (market data). The authors report great success and accuracy using these caNNs to calibrate the Heston model and higher dimensional models such as the Bates model (Liu et al. 2019). (Itkin 2019) proposes another deep-learning approach to model calibration which builds upon the work of (Hernandez 2016) using a neural network to invert the learnt pricing map and directly return the optimal model parameters removing the need for any global optimisation step altogether.

Crucially, since the networks can be trained offline, on-line inference becomes extremely fast. (Ardizzone et al. 2018) identify what they refer to as invertible neural networks that determine hidden parameters and their properties given observable/experimental data and showcase their success in various domains of natural sciences. These types of networks are a good candidate for financial model calibration problems too. Despite the power of these invertible approaches (Horvath et al. 2019) identify a key flaw in their use in quantitative finance: from a standpoint of risk analysis it is not clear what relationships these model parameters have to the original market data since they are simply churned out of a somewhat 'black box' and there is no real direct control over the generated inverted pricing map.

CHAPTER 3

Financial Models

3.1 Black-Scholes Framework

In 1973 the Black-Scholes model for valuing options was presented in the Journal of Political Economy (Black and Scholes 1973) wherein the following PDE describing the value of an option is presented:

$$\frac{1}{2}\sigma^2 S^2 \frac{\partial^2 V}{\partial^2 S^2} + \frac{\partial V}{\partial t} - rV + rS\frac{\partial V}{\partial S} = 0 \tag{3.1}$$

where $V = V(S,t)$ is the value of the option, S is the value of the underlying asset which has volatility equal to σ, t is time and r is the risk-free interest rate. Moreover, the value of an option at expiry time $t = T$, where T is the exercise date of the option, is simply given by the payoff function evaluated at time T. The PDE is contingent on the assumption that the instantaneous log return of the asset price is a geometric Brownian motion given by

$$dS_t = rS_t dt + \sigma S_t dZ_t, \tag{3.2}$$

where Z_t is a Wiener process (also called a standard Brownian motion) (Hida 1980). Sufficient conditions under which PDE (3.1) will hold are as follows (Black and Scholes 1973):

- The risk-free interest rate, r, is assumed to be constant.
- The volatility, σ, is assumed to be constant.
- There are no dividends paid out.
- Zero transaction costs.
- Zero penalties to short selling.

Under these assumptions along with the restriction of focusing on European style options where the option can only be exercised at the maturity date T, there exists analytical formulae for the value of call and put options derived by solving the corresponding diffusion equation.

Definition 3.1 (Value of a European Vanilla Call). The value of a European (Vanilla) Call option at time t on an underlying asset S is defined by

$$C(S,t) = SN(d_1) - Pv(X)N(d_2) \tag{3.3}$$

where X is the strike price of the option, $Pv(X) = Xe^{-r(T-t)}$, N is the cumulative Gaussian distribution function and $d_{1,2}$ are defined as

$$d_1 = \frac{ln\left(\frac{S}{X}\right) + \left(r + \frac{\sigma^2}{2}\right)(T-t)}{\sigma\sqrt{T-t}} \tag{3.4}$$

$$d_2 = d_1 - \sigma\sqrt{T-t}. \tag{3.5}$$

Definition 3.2 (Value of a European Vanilla Put). The value of a European (Vanilla) Put option at time t on an underlying asset S is defined by

$$P(S,t) = Pv(X)N(-d_2) - SN(-d_1) \tag{3.6}$$

where $d_{1,2}$, X and $Pv(X)$ are defined as in definition 3.1.

It is easy to relax the assumption of zero dividends in which case the Black-Scholes PDE (3.1) becomes

$$\frac{1}{2}\sigma^2 S^2 \frac{\partial^2 V}{\partial^2 S^2} + \frac{\partial V}{\partial t} - rV + (r-q)S\frac{\partial V}{\partial S} = 0 \tag{3.7}$$

where q is the (constant) dividend yield. Furthermore, the value of a European Vanilla Calls and Puts at time t on underlying S with dividend yield q also have analytical formulae under the Black-Scholes framework.

Definition 3.3 (Value of a European Vanilla Call with Dividends). The value at time t of a European (Vanilla) Call option on underlying asset S paying out (constant) dividends with

dividend yield equal to q is defined by

$$C(S,t)_q = Pv_q(S)N(d_1) - Pv(X)N(d_2) \tag{3.8}$$

where X is the strike price of the option, $Pv(X) = Xe^{-r(T-t)}$, $Pv_q(S) = Se^{-q(T-t)}$, N is the cumulative Gaussian distribution function and $d_{1,2}$ are defined as

$$d_1 = \frac{ln\left(\frac{S}{X}\right) + \left(r - q + \frac{\sigma^2}{2}\right)(T-t)}{\sigma\sqrt{T-t}} \tag{3.9}$$

$$d_2 = d_1 - \sigma\sqrt{T-t}. \tag{3.10}$$

Definition 3.4 (Value of a European Vanilla Put with Dividends). The value at time t of a European (Vanilla) Put option on underlying asset S paying out (constant) dividends with dividend yield equal to q is defined by

$$P(S,t)_q = Pv(X)N(-d_2) - Pv_q(S)N(-d_1) \tag{3.11}$$

where $d_{1,2}$, X, $Pv(X)$ and $Pv_q(S)$ are defined as in definition 3.3.

3.2 Black-76 Model

Fischer Black extended the Black-Scholes framework to account for options and contracts involving forwards (Black 1976) including derivatives such as futures, caps and floors. We will refer to the model as the Black model. Black's model assumes the forward (forward rate, forward stock price etc), F, is a log-normal process that is described by the following SDE:

$$dF = \sigma_{black} F dZ \tag{3.12}$$

where σ_{black} is a constant volatility term and dZ is a Wiener process. There are explicit formulae for the price of calls and puts on futures under Black's model following the same methodology as the standard Black-Scholes approach.

Definition 3.4 (Black Value of a Call Option). The value at time t of a European (Vanilla) Call option under the Black model with maturity date T on a futures contract F with settlement

date $\hat{T} \geq T$ is defined by

$$C(F,t)_B = Pv(F)N(d_1) - Pv(X)N(d_2) \tag{3.13}$$

where X is the strike price, $Pv(F) = Fe^{-r(T-t)}$, $Pv(X) = Xe^{-r(T-t)}$. N is the cumulative Gaussian distribution function and $d_{1,2}$ are defined as

$$d_1 = \frac{ln\left(\frac{F}{X}\right) + \left(\frac{\sigma^2}{2}\right)(T-t)}{\sigma\sqrt{T-t}} \tag{3.14}$$

$$d_2 = d_1 - \sigma\sqrt{T-t}. \tag{3.15}$$

Definition 3.5 (Black Value of a Put Option). The value at time t of a European (Vanilla) Put option under the Black model with maturity date T on a futures contract F with settlement date $\hat{T} \geq T$ is defined by

$$P(F,t)_B = Pv(X)N(-d_2) - Pv(F)N(-d_1) \tag{3.16}$$

where $d_{1,2}$, X, $Pv(X)$ and $Pv(F)$ are defined as in definition 3.4.

3.3 Bachelier Model

Bachelier's model (Bachelier 1900) assumes that the underlying asset is normally distributed as opposed to log-normally distributed. Accordingly, it is often referred to as the normal model. In this case the forward process is described by the following SDE:

$$dF = \sigma_{normal} dZ \tag{3.17}$$

where σ_{normal} is constant velocity and dZ is a Wiener process. In this model the forward (rate) F does not enter the right hand side of the SDE, consequently the solution (using Ito calculus) to the SDE is simply given by

$$F = F(0) + \sigma Z.$$

Definition 3.6 (Normal Value of a Call Option). The value at time t of a European (Vanilla) Call option under the normal model with maturity T on a futures contract F with settlement date $\hat{T} \geq T$ is defined by

$$C(F,t)_N = Pv(F-X)N(d) + Pv(\sigma\sqrt{T-t})n(d) \tag{3.18}$$

where

$$Pv(F-X) = (F-X)e^{-r(T-t)},$$
$$Pv(\sigma\sqrt{T-t}) = \sigma\sqrt{T-t}\cdot e^{-r(T-t)},$$

N is the cumulative Gaussian distribution function, n is the Gaussian density function and d is defined as

$$d = \frac{(F-X)}{\sigma\sqrt{T-t}} \tag{3.19}$$

Definition 3.7 (Normal Value of a Put Option). The value at time t of a European (vanilla) Put option under the normal model with maturity T on a futures contract with settlement date $\hat{T} \geq T$ is defined by

$$P(F,t)_N = Pv(X-F)N(-d) + Pv(\sigma\sqrt{T-t})n(d) \tag{3.20}$$

where

$$Pv(X-F) = (X-F)e^{-r(T-t)}$$

and the other terms are defined the same as in definition 3.6.

A key difference between the normal model (Bachelier's Model) and the log-normal model (Black's Model) is that the normal model allows the underlying and strikes to take negative values (e.g negative forward rates) due to the normality assumption. Negative interest rates are indeed possible and observed. Bachelier's model does not rely on the log-normal assumption involving the natural logarithm so it does not break down below the 0 boundary. Nonetheless, we choose to work on the positive domain.

3.4 SABR Model

The SABR model (Hagan et al. 2002) comes under a class of pricing models known as stochastic volatility models. Such models relax the assumption of constant volatility that regimes such as Black-Scholes (Black and Scholes 1973) and Bachelier (Bachelier 1900) rely on so that the volatility is modelled as a stochastic process.

SABR is a widely used model for forward prices for two main reasons. Firstly, it has the ability to capture the observed relationship between asset prices and market smiles: if the asset price goes up, the smile shifts up and vice-versa. Popular local volatility models used at the time pioneered by (Dupire et al. 1994) where not able to capture this relationship, in fact, they predicted the complete opposite despite being self-consistent (Hagan et al. 2002). Secondly, Hagan's SABR model provides closed form approximations for the implied volatility that are actually very accurate making it extremely efficient to implement in practice. In the SABR model the forward price, F, is described by the following system (Hagan et al. 2002):

$$dF = \sigma F^{\beta} dZ^1, \tag{3.21}$$

$$d\sigma = \alpha \sigma dZ^2, \tag{3.22}$$

where $dZ^1 \cdot dZ^2 = \rho dt$, F is the forward price, σ is the volatility, α is the volatility of volatility, and dZ^1, dZ^2 are correlated Wiener processes with correlation equal to ρ.

In (Hagan et al. 2002) Hagan uses singular perturbation techniques (Lagerstrom and Casten 1972) in order to derive the SABR price of European call and put options using what he refers to as a 'small volatility expansion' (Hagan et al. 2002) where the analysis considers $\bar{\sigma} = \epsilon\sigma$ and $\bar{\alpha} = \epsilon\alpha$ for a sufficiently small $\epsilon \ll 1$. Afterwards, the resulting expressions for option prices and implied volatilities can be expressed in the standard terms by setting $\epsilon = 1$. We do not reproduce the entire derivation here, instead, we choose to highlight some key parts that allow us arrive at the implied volatility approximations that serve as the starting point for our neural networks.

Hagan derives two different approximations for the implied volatility for the SABR model, each under a different pricing regime. The normal implied volatility - obtained using the Bachelier model shown in equation (3.17) and the log-normal implied volatility - obtained using Black's model, see equation (3.12). Consider the following system using a general function $J(F)$ and the transformed 'small' parameters described above:

$$dF = \bar{\sigma} J(F) dZ^1, \tag{3.23}$$

$$d\bar{\sigma} = \bar{\alpha}\bar{\sigma} dZ^2, \tag{3.24}$$

where $\bar{\sigma} = \epsilon\sigma$, $\bar{\alpha} = \epsilon\alpha$ and $\epsilon \ll 1$.

The value of a European Call option under the SABR model (disregarding the discount factor) is given by (Hagan et al. 2002):

$$C(F,t,\sigma) = max(F-X,0) + \frac{|F-X|}{4\sqrt{\pi}} \int_{\frac{x^2}{2texp}-\epsilon^2\theta}^{\infty} \frac{e^{-q}}{q^{\frac{3}{2}}} dq \tag{3.25}$$

where

$$\epsilon^2\theta = ln\left(\frac{\epsilon\sigma z}{F-X}\sqrt{Q(0)Q(\epsilon\sigma z)}\right) + ln\left(\frac{xI^{\frac{1}{2}}(\epsilon\sigma z)}{z}\right) + \frac{1}{4}\epsilon^2\rho\alpha\sigma b z^2, \tag{3.26}$$

$$z = \frac{1}{\epsilon\sigma}\int_X^F \frac{1}{J(F')} dF', \quad Q(\epsilon\sigma z) = J(F), \tag{3.27}$$

$$b = \frac{Q'(\epsilon\sigma z_0)}{Q(\epsilon\sigma z_0)}, \quad x = z[1+O(\epsilon)]. \tag{3.28}$$

3.4.1 Normal Implied Volatility

By assigning $J(F) = 1$, $\alpha = 0$ and $\bar{\sigma} = \sigma_{normal}$ in system (3.23)-(3.24) we get the SDE:

$$dF = \sigma_{normal} dZ \tag{3.29}$$

which is exactly the SDE from the normal model (Bachelier's model), see equation (3.12). We now work through (3.25)-(3.28):

$$J(F) = 1 \implies z = \frac{1}{\epsilon\sigma}\int_X^F 1 dF' = \frac{1}{\epsilon\sigma}(F-X) \tag{3.30}$$

Using the fact that $x = z[1 + O(\epsilon)]$ we have

$$x^2 = \frac{(F-X)^2}{\epsilon^2\alpha^2} = \frac{(F-X)^2}{\bar{\sigma}^2} \tag{3.31}$$

Now applying $\bar{\sigma} = \sigma_{normal} \implies \bar{\sigma}^2 = \sigma^2_{normal}$ yields

$$x^2 = \frac{(F-X)^2}{\sigma^2_{normal}} \tag{3.32}$$

The lower limit of the integral in (3.25) is given by

$$\frac{x^2}{2t_{exp}} - \epsilon^2\theta = \frac{(F-X)^2}{2\sigma^2_{normal}t_{exp}} \tag{3.33}$$

Since $\epsilon^2\theta$ goes to 0 as the 2 logarithms and the final term in (3.26) all reduce to 0. Evaluating the full expression and the integral would then give the expression for the call option price under the Bachelier model. Hagan (Hagan et al. 2002) shows that the price of the option under Bachelier's model is equal to the price of the option under the SABR model if and only if σ_{normal} is as follows (Hagan et al. 2002):

$$\sigma_{normal}(X) = \frac{\epsilon\sigma(F-X)}{\int_X^F \frac{1}{J(G)}dG} \cdot \left(\frac{\delta}{x(\delta)}\right) \cdot \left\{1 + \left[\frac{(2\phi_2 - \phi_1^2)\sigma^2 J^2(\hat{F})}{24} + \frac{\rho\alpha\sigma\phi_1 J(\hat{F})}{4} + \frac{(2-3\rho^2)\alpha^2}{24}\right] \cdot \epsilon^2 t_{exp}\right\} \tag{3.34}$$

where

$$\hat{F} = \sqrt{FX}, \quad \phi_1 = \frac{J'(\hat{F})}{J(\hat{F})}, \quad \phi_2 = \frac{J''(\hat{F})}{J(\hat{F})}, \tag{3.35}$$

and

$$\delta = \frac{\alpha(F-X)}{\sigma J(\hat{F})}, \quad x(\delta) = ln\left(\frac{\sqrt{1-2\rho\delta+\delta^2} - \rho + \delta}{1-\rho}\right). \tag{3.36}$$

Recall that the SABR model is simply system (3.23)-(3.24) with $J(F) = F^\beta$ as shown by system (3.21)-(3.22). Accordingly, to derive the normal implied volatility (approximation) for the SABR model we will work through (3.34)-(3.36) with $J(F) = F^\beta$. To ease the notation, we express (3.34) as follows

$$\sigma_{normal}(X) = A \cdot B \cdot \left\{1 + [C + D + E] \cdot \epsilon^2 t_{exp}\right\} \tag{3.37}$$

with

$$A = \frac{\epsilon\sigma(F-X)}{\int_X^F \frac{1}{J(G)}dG}, \quad B = \left(\frac{\delta}{x(\delta)}\right), \quad C = \frac{(2\phi_2 - \phi_1^2)\sigma^2 J^2(\hat{F})}{24}, \tag{3.38}$$

$$D = \frac{\rho\alpha\sigma\phi_1 J(\hat{F})}{4}, \quad E = \frac{(2-3\rho^2)\alpha^2}{24}. \tag{3.39}$$

Each part will be tackled separately. The following facts follow on from the definitions of $J(F) = F^\beta, \phi_1$ and ϕ_2:

$$J(F) = F^\beta, \quad J^{'}(F) = \beta F^{\beta-1}, \quad J^{''}(F) = \beta(\beta-1)F^{\beta-2}, \tag{3.40}$$

$$\phi_1 = \frac{J^{'}(\hat{F})}{J(\hat{F})} = \frac{\beta\hat{F}^{\beta-1}}{\hat{F}^\beta} = \beta\hat{F}^{-1}, \tag{3.41}$$

$$\phi_2 = \frac{J^{''}(\hat{F})}{J(\hat{F})} = \frac{\beta(\beta-1)\hat{F}^{\beta-2}}{\hat{F}^\beta} = \beta(\beta-1)\hat{F}^{-2}. \tag{3.42}$$

$$2\phi_2 - \phi_1^2 = 2\beta(\beta-1)\hat{F}^{-2} - (\beta\hat{F}^{-1})^2 = -\beta(2-\beta)\hat{F}^{-2} \tag{3.43}$$

Now consider the integral on the denominator of part A:

$$\int_X^F \frac{1}{J(G)}dG = \int_X^F G^{-\beta}dG = \left[\frac{G^{1-\beta}}{1-\beta}\right]_X^F = \frac{F^{1-\beta} - X^{1-\beta}}{1-\beta} \tag{3.44}$$

Part A then follows on from the result of the integral

$$A = \frac{\epsilon\sigma(F-X)(1-\beta)}{F^{1-\beta} - X^{1-\beta}} \tag{3.45}$$

Part B is given by:

$$B = \left(\frac{\delta}{x(\delta)}\right) \tag{3.46}$$

where

$$\delta = \frac{\alpha(F-X)}{\sigma\hat{F}^\beta}, \quad x(\delta) = ln\left(\frac{\sqrt{1-2\rho\delta+\delta^2} - \rho + \delta}{1-\rho}\right). \tag{3.47}$$

Part C is given by:

$$C = \frac{\beta(2-\beta)\sigma^2}{24} \cdot \hat{F}^{-2}\hat{F}^{2\beta} = \frac{\beta(2-\beta)\sigma^2}{24} \cdot \hat{F}^{2\beta-2} = \frac{-\beta(2-\beta)\sigma^2}{24\hat{F}^{2-2\beta}}. \tag{3.48}$$

Part D is given by:

$$D = \frac{\rho\alpha\sigma\phi_1 J(\hat{F})}{4} = \frac{\rho\alpha\sigma\beta\hat{F}^{-1}\hat{F}^\beta}{4} = \frac{\rho\alpha\sigma\beta}{4\hat{F}^{1-\beta}}. \tag{3.49}$$

Finally, part E is the same as in (3.34):

$$E = \frac{(2-3\rho^2)\alpha^2}{24}. \tag{3.50}$$

Substituting A, B, C, D and E into (3.37) and letting $\epsilon = 1$ gives us the analytical expression for the normal implied volatility approximation (Hagan et al. 2002). The same process can be repeated for European put options using put-call parity or starting from the beginning, however, the implied volatility is the same.

Definition 3.8 (Normal Implied Volatility for the SABR model). Let F be the forward price, let t_{exp} be the expiry date of a European option on the forward F with strike price X and let σ be the initial volatility. The normal implied volatility, which is the value of volatility in Bachelier's pricing model such that the Bachelier option price is equal to the SABR option price, is defined as (Hagan et al. 2002):

$$\sigma_{normal}(X) = \frac{\sigma(F-X)(1-\beta)}{F^{1-\beta}-X^{1-\beta}} \cdot \left(\frac{\delta}{x(\delta)}\right) \cdot \left\{1 + \left[\frac{-\beta(2-\beta)\sigma^2}{24\hat{F}^{2-2\beta}} + \frac{\rho\alpha\sigma\beta}{4\hat{F}^{1-\beta}} + \frac{(2-3\rho^2)\alpha^2}{24}\right] \cdot t_{exp}\right\} \tag{3.51}$$

where all extra terms are defined as in the above derivation.

3.4.2 Log-normal Implied Volatility

To derive the log-normal implied volatility expansion the same preceding analysis taken for the normal implied volatility can be carried out using the Black model, which is described in section 3.2, instead of Bachelier's model. In most cases option prices are quoted in terms of the log-normal implied volatility (Hagan et al. 2002).

Definition 3.9 (Log-normal Implied Volatility for the SABR model) let F be the forward price, let t_{exp} be the expiry date of a European option on the forward F with strike price X and let σ be the initial volatility. The log-normal implied volatility, which is the volatility from Black's pricing model (Black 1976) such that the Black option price is equal to the

SABR option price, is defined as (Hagan et al. 2002):

$$\sigma_{ln}(X) = \frac{\sigma\delta}{(FK)^{\frac{(1-\beta)}{2}}\left\{1 + \frac{(1-\beta)^2 \cdot ln^2\left(\frac{F}{X}\right)}{24} + \frac{(1-\beta)^4 \cdot ln^4\left(\frac{F}{X}\right)}{1920}\right\} \cdot x(\delta)} \cdot \left\{1 + \left[\frac{(1-\beta)^2\sigma^2}{24(FX)^{1-\beta}} + \frac{\rho\sigma\alpha\beta}{4(FK)^{\frac{(1-\beta)}{2}}} + \frac{(2-3\rho^2)\alpha^2}{24}\right] \cdot t_{exp}\right\} \quad (3.52)$$

where

$$\delta = \frac{\alpha(FX)^{\frac{(1-\beta)}{2}} \cdot ln\left(\frac{F}{X}\right)}{\sigma}, \quad x(\delta) = ln\left(\frac{\sqrt{1 - 2\rho\delta + \delta^2} - \rho + \delta}{1 - \rho}\right). \quad (3.53)$$

3.4.3 Underlying Forward Process

Much of the current literature focuses on the log-normal approximation, however, we do not neglect the normal approximation in our research and endeavour to learn both implied volatility functions given by definitions 3.8 and 3.9 with our neural network architecture.

To summarise some key points, the SABR model subsumes one of two pricing regimes: Bachelier or Black. From this, the relevant implied volatility expansions are obtained. It is usual to calibrate the model parameters, however, of all the parameters ($\alpha, \beta, \sigma, \rho$), β, the parameter controlling the distribution of the underlying asset, is commonly predetermined based on some prior belief of what process the underlying follows. There are 3 main choices, each resulting in a different process.

- $\beta = 0$ results in a stochastic normal underlying process (Hagan et al. 2002) characterised by the system:

$$dF = \sigma dZ^1, \quad (3.54)$$

$$d\sigma = \alpha\sigma dZ^2. \quad (3.55)$$

- $\beta = 0.5$ results in a stochastic CIR type underlying process (Hagan et al. 2002) characterised by the system:

$$dF = \sigma\sqrt{F}dZ^1, \tag{3.56}$$

$$d\sigma = \alpha\sigma dZ^2. \tag{3.57}$$

- $\beta = 1$ results in a stochastic log-normal underlying process (Hagan et al. 2002) characterised by the system:

$$dF = \sigma F dZ^1, \tag{3.58}$$

$$d\sigma = \alpha\sigma dZ^2. \tag{3.59}$$

By stochastic we are emphasising that in each of the above systems the volatility itself is a stochastic process. We investigate neural network performance on each of these types of underlying processes as part of our research.

CHAPTER 4

Neural Computation

Feedforward neural networks use input-output pairs (training examples) and some cost function C to approximate a function by learning the optimal parameters of the network (weights and biases) such that the cost function is minimised. It does so by differentiating the cost function with respect to the weights and biases at each layer in the network using a process called backpropagation and then uses an optimiser such a stochastic gradient descent (SGD) to minimise the cost function to find the optimal weights and biases.

We will describe the neural network architecture using computational graphs in which nodes represent variables and edges represent functional dependencies between variables. For example, an edge from node x to node y indicates that y is a functions of x.

4.1 Feedforward Neural Networks

For any node in a computational graph, H_b^a, the superscript indicates which layer we are at and the subscript indicates what unit within the layer we are at. Figure 4.1 shows an example of a feedforward neural network with 3 input parameters, a single hidden layer with 3 nodes and a single output.

The depth of the network is equal to the number of layers and the width corresponds to how many units there are in each hidden layer, in the example above there are only 3 units in the single hidden layer so the network has a width of 3. Define $l \in (1, \ldots, L)$ to be a layer in the neural network where L is the total number of layers, layer 1 is called the input layer and layer L is the output layer. Figure 4.2 shows what is happening in more detail at each

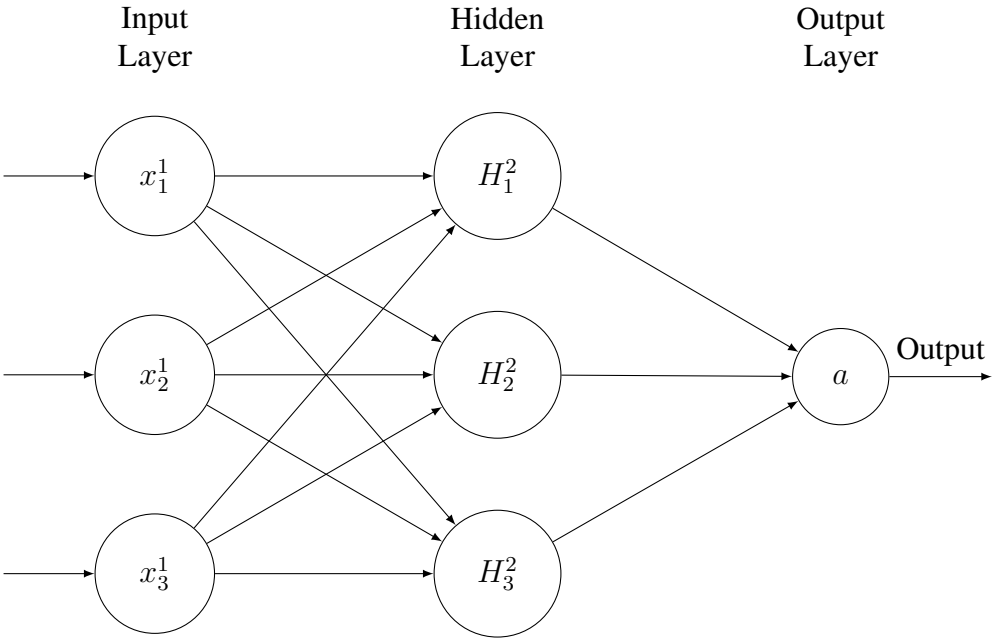


FIGURE 4.1. *Feedforward neural network with one hidden layer.*

particular node from the hidden layers in the neural network, it will be the starting point from where we develop the understanding of how these networks are trained. Firstly, we give some notation that will be used throughout adopted from (Nielsen 2015):

- $w^l_{j,k}$ is the weight of the edge from unit k in layer $l-1$ to unit j in layer l.
- b^l_j is known as the bias of the $j'th$ unit in layer l.
- $H^l_j = (\sum_{k=1}^{m} (w^l_{jk} \cdot a^{l-1}_k)) + b^l_j$ where m is the number of units in the previous layer.
- $a^l_j = \phi(H^l_j)$ is the activation of unit j in layer l, where ϕ is the activation function which transforms H^l_j.

Importantly, each node H^l_j depends on all of the activation nodes from the previous layer weighted by the parameter $w^l_{j,k}$ with an additional bias term b^l_j added. These weights and biases are the learnable parameters on the network.

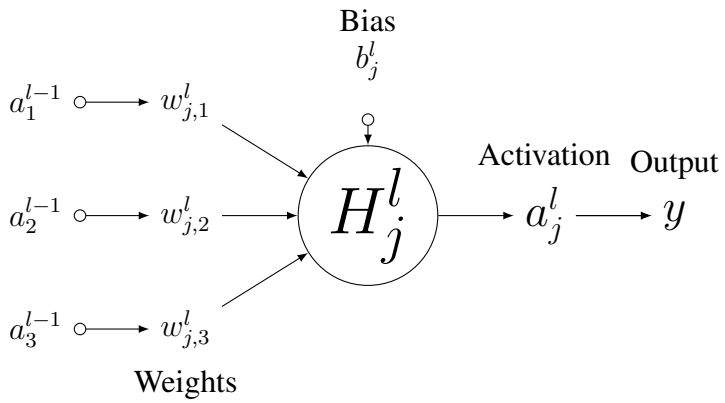


FIGURE 4.2. *Working of a single unit from a feedforward neural network.*

4.1.1 Training a Feedforward Neural Network

Let $N(w)$ be a neural network and let $F(x)$ be some function that we wish to approximate using $N(w)$. Assume we have n input-output training pairs: $\left\{(x^{(1)}, y^{(1)}), \ldots, (x^{(n)}, y^{(n)})\right\}$ and we have some cost function given by $C = \frac{1}{n}\sum_{i=1}^{n} C^{(i)}$ where $C^{(i)}$ is the error on the $i'th$ training example for a suitably defined loss function such as the mean squared error: $C^{(i)} = \frac{1}{2}(y^{(i)} - a^L)^2$, where a^L is the output of the neural network. To train the neural network we want to solve the following optimisation problem:

$$w^* = \underset{w}{\operatorname{argmin}}\, C(N(w), F).$$

That is, to calibrate the parameters of the network such that they minimise the error between the neural network and the function we wish to approximate across all training examples. In order to solve this optimisation problem using a gradient descent based method the partial derivatives of the cost functions with respect to the network parameters (weights and biases), $\frac{\partial C}{\partial w_{j,k}^l}$ and $\frac{\partial C}{\partial b_j^l}$, are required. Backpropagation allows us to calculate these derivatives. Recall that the outputs of a neural network are activation units which are simply the last layer nodes H^L fed through the activation function ϕ as follows: $a^L = \phi(H^L)$. We will use this setup to derive the partial derivatives in question following closely the intuitive explanation given in (Nielsen 2015) . Using the fact that the units are given by

$$H_j^l = \left(\sum_{k=1}^{m} w_{jk}^l \cdot a_k^{l-1}\right) + b_j^l$$

and the activation units are given by

$$a_j^l = \phi(H_j^l).$$

We can use the chain rule to write

$$\frac{\partial C}{\partial w_{j,k}^l} = \frac{\partial C}{\partial H_j^l} \cdot \frac{\partial H_j^l}{\partial w_{j,k}^l} = \frac{\partial C}{\partial H_j^l} \cdot a_k^{l-1}, \tag{4.1}$$

$$\frac{\partial C}{\partial b_j^l} = \frac{\partial C}{\partial H_j^l} \cdot \frac{\partial H_j^l}{\partial b_j^l} = \frac{\partial C}{\partial H_j^l}. \tag{4.2}$$

Define the common quantity in the above two equations as follows

$$\tau_j^l = \frac{\partial C}{\partial H_j^l}. \tag{4.3}$$

Backpropagation provides this quantity, τ_j^l, for each node at each layer in the network. Notice that τ_j^l depends on the cost function C so it will be different depending on the cost function being used. We start at the output layer L and work backwards through the network. Using the chain rule again we obtain

$$\tau_j^L = \frac{\partial C}{\partial H_j^L} = \frac{\partial C}{\partial a_j^L} \cdot \frac{\partial a_j^L}{\partial H_j^L} = \frac{\partial C}{\partial a_j^L} \cdot \phi^{'}(H_j^L). \tag{4.4}$$

Now we work through the hidden layers and calculate τ_j^l for $l \in \{2, \ldots, L-1\}$ as follows

$$\tau_j^l = \frac{\partial C}{\partial H_j^l} = \frac{\partial C}{\partial a_j^l} \cdot \frac{\partial a_j^l}{\partial H_j^l}. \tag{4.5}$$

Using the chain rule with respect to $\frac{\partial C}{\partial a_j^l}$ we obtain:

$$\tau_j^l = \left(\sum_k \frac{\partial C}{\partial H_k^{l+1}} \cdot \frac{\partial H_k^{l+1}}{\partial a_j^l} \right) \cdot \phi^{'}(H_j^l) \tag{4.6}$$

Finally, replacing $\frac{\partial C}{\partial H_k^{l+1}}$ with the definition of τ_j^{l+1} given by (4.3) and noticing that $\frac{\partial H_k^{l+1}}{\partial a_j^l} = w_{k,j}^{l+1}$ we arrive at

$$\tau_j^l = \phi^{'}(H_j^l) \cdot \left(\sum_k \tau_k^{l+1} \cdot w_{k,j}^{l+1} \right) \tag{4.7}$$

So, given we know the value of τ^{l+1} we can compute τ^l. Starting from the output layer one can backpropogate to retrieve the values of τ_j^l at all hidden layers providing the partial derivatives of the cost function with respect to the weights and biases at every node on every layer on the network.

Once the partial derivatives of the cost function have been calculated gradient descent based techniques can be used to optimise the cost function to select the network parameters that produce the smallest error. Some of the optimisers that have been used in the experiments will now briefly be discussed.

4.1.1.1 Optimisation Algorithms

The main benefits SGD has in comparison to standard gradient descent is the fact that shuffling the data and the stochastic nature of the algorithm ensures that it is exploring more of the search space which helps it get out of local optima and prevents overfitting. However, a drawback is that SGD uses a fixed learning rate.

Algorithm 1: Stochastic Gradient Descent (SGD).

Input: cost function $J : \mathbb{R}^m \to \mathbb{R}$, learning rate $\epsilon \in \mathbb{R}, \epsilon > 0$.
$x \leftarrow$ Initial point in $\mathbb{R}^m$;
while *termination condition not satisfied* **do**
  Shuffle training set randomly;
  for *i = 1:n* **do**
    $x \leftarrow x - \epsilon \cdot \nabla J_i(x)$;
  end for
end while
return x;

The learning rate is a parameter that heavily impacts the time taken to train a neural network, it controls the extent to which weights are updated. Too high a learning rate may result in large movements across the surface of the cost function which may cause the optimum to be overlooked and too low a learning rate may result in slow convergence. Cost functions have different gradients in different directions due to their shape, therefore, we require different learning rates for the different directions. Ideally, we would like the learning rate to be small

Algorithm 2: Adam (Kingma and Ba 2014).

Input: cost function J, decay rates $\delta_1, \delta_2 \in (0, 1)$, step size ϵ, $\zeta = 10^{-8}$.
Set $r, s, t = 0$;
Choose an initial parameter θ;
while *termination condition not satisfied* **do**
  $t \leftarrow t + 1$;
  $b \leftarrow \nabla_\theta J(\theta)$;
  $s \leftarrow \delta_1 \cdot s + (1 - \delta_1)\, b$;
  $r \leftarrow \delta_2 \cdot r + (1 - \delta_2)\, b \odot b$;
  $\hat{s} \leftarrow \frac{s}{1-\delta_1^t}$;
  $\hat{r} \leftarrow \frac{r}{1-\delta_2^t}$;
  $z \leftarrow -\epsilon \cdot \frac{\hat{s}}{\sqrt{\hat{r}}+\zeta}$;
  $\theta \leftarrow \theta + z$;
end while
return θ;

in directions where the gradient has been large over time and larger where the gradient has been small over time. Adam (Kingma and Ba 2014) combines the effects of two other popular optimisers called RMSProp (Tieleman and Hinton 2012) and AdaGrad (Duchi et al. 2011) to adapt the learning rates by updating the first and second moments of the gradients at each time step. The algorithm is known to perform well in problems with large dimensions and there is very little parameter tuning to be done since the authors (Duchi et al. 2011) report effective defaults that rarely need to be altered.

4.1.1.2 Activation Functions

The activation functions serve a purpose to introduce non-linearity to the network, indeed most phenomena in quantitative finance are characterised by non-linear relationships between state variables. Without the non-linearity neural nets could only be used for linear classification or regression problems. There are a few properties that we would like activation functions to enjoy to ensure optimality:

- Non-linear for hidden layers to capture complex relationships.
- Zero-centred output.
- Resistance to exploding and vanishing gradients.

Remark 4.1 (The vanishing and exploding gradients problem). When a neurons activation saturates close to 0 or 1, the gradients of the activation at these regions are close to zero. During backpropagation this gradient, given by $\phi^{'}(H_j^l)$ in our notation, is a multiplicative factor in the calculation of τ_j^l (see equations (4.4), (4.6) and (4.7)). Consequently, the gradients of the weights will become extremely small which will prevent the weights from updating and hinder training (the opposite is true for the exploding gradients problem).

If the outputs are not zero centred the output is always the same sign meaning during backpropagation the gradient of the weights will all have to move in the same direction which makes optimisation harder and so a zero-centred activation function is favoured for faster convergence.

In practice there is nearly always a trade off between these properties but it is important to realise that the type of activation function chosen is problem specific, we will see there are various constraints in option pricing and stochastic volatility models that make some activation functions more suitable than others.

Definition 4.1 (Linear/Identity). The *Linear/Identity* activation function is defined as

$$\phi(x) = x. \tag{4.8}$$

The linear activation is often used in regression problems for the output layer since it makes the derivative of the cost function with respect to the network output simple to calculate.

Definition 4.2 (Rectified Linear Unit (ReLU)). The *ReLU* (Glorot et al. 2011) activation function is defined as

$$\phi(x) = max(0, x) \in [0, \infty). \tag{4.9}$$

The ReLU function is extremely cheap to implement and it does not suffer from gradient saturation so convergence on the network is fast, however, the output is unbounded from above which can cause the activation to explode.

Definition 4.3 (Exponential Linear Unit (ELU)). The *ELU* (Clevert et al. 2015) activation function is defined as

$$\phi(\alpha, x) = \begin{cases} x & \text{if } x > 0, \\ \alpha(e^x - 1) & \text{if } x \leq 0. \end{cases} \tag{4.10}$$

The ELU function takes values in the range $(-\alpha, \infty)$, usually $\alpha = 1$ is chosen, so it is unbounded from above which can cause the activation to blow up. However, since it can take negative values the average output is closer to 0 which speeds up convergence.

Definition 4.4 (Modified ELU (MELU)). The *MELU* (Itkin 2019) activation function is defined as

$$\phi(\alpha, x) = \begin{cases} \frac{0.5x^2 + (1-2\alpha)x}{x - 2 + \frac{1}{\alpha}} & \text{if } x > 0, \\ \alpha(e^x - 1) & \text{if } x \leq 0. \end{cases} \tag{4.11}$$

Sharing similar properties to ELU, MELU has been proposed by the authors (Itkin 2019) to be a suitably defined activation function that is of class C^2 for option pricing neural networks to allow option Greeks to be calculated.

Definition 4.5 (SoftPlus). The *SoftPlus* activation function is defined as

$$\phi(x) = \ln(1 + e^x) \in (0, \infty)\,. \tag{4.12}$$

The SoftPlus function is of class C^∞ and it is monotonic. This function will prove useful for ensuring that the output of a neural network is positive whilst preserving their continuous second and first order derivatives.

4.1.2 Universal Approximation Theorem

Definition 4.5 (Discriminatory Functions) (Cybenko 1989) $\phi : \mathbb{R} \to \mathbb{R}$ is known as discriminatory if

$$\int_{I_n} \phi\left(w^T x + b\right) d\mu(x) = 0$$

$\forall w \in \mathbb{R}^n, b \in \mathbb{R} \implies \mu = 0$ for some measure μ . Note that this definition is making use of the Lebesgue integral (Bartle and Bartle 1995).

Theorem 4.1 (Universal Approximation Theorem (UAT)) (Cybenko 1989) Let ϕ be any discriminatory continuous function, then for any function $f \in c(I_n)$ (f is continuous on the domain I_n) and any $\epsilon > 0$ $\exists$ a finite sum of the form

$$M(x) = \sum_{j=1}^{m} \alpha_j \cdot \phi(w_j^T x + b_j)$$

such that $|M(x) - f(x)| < \epsilon \; \forall x \in I_n$ where $\alpha_j, b_j \in \mathbb{R}$.

We refer the reader to (Cybenko 1989) for a full treatment of the proof. This means we can chose ϵ as arbitrarily small as we like and there is never a neural network, $M(x)$, that is different by more than ϵ from the function that we wish to approximate. Consequently, provided there are an adequate number of units, a neural network with a single hidden layer can approximate any continuous function. (McGhee 2018) stayed in line with this result and approximated SABR volatility smiles with impressive accuracy using one hidden layer with up to 1000 units.

4.1.3 The Power of Depth

There are 2 main issues that the UAT does not address:

- It does not give any indication as to how many units are required in the hidden layer.
- It does not give any indication of how easy or difficult it is to train the network to approximate a given function.

In practice, if a very large number of units are required in the hidden layer to achieve a suitable degree of accuracy, single layer ANNs can become non-viable due to the number of parameters to be learnt increasing exponentially and susceptibility to overfitting. (Eldan and Shamir 2016) study the benefits of depth and present a useful result that shows how much expressive power is gained from going from 2 layers to 3. The result is reproduced in Theorem 4.2.

Theorem 4.2 (Eldan and Shamir 2016). If the activation function ϕ satisfies some weak assumptions then there is a function $h : \mathbb{R}^n \to \mathbb{R}$ and a probability measure μ on $\mathbb{R}^n$ such that

- h is expressible by a neural network with 3 layers of width $O(n^{\frac{19}{4}})$
- Every function p approximated by a neural network with 2 layers of width at most ce^{cn} satisfies

$$\mathbb{E}_{\mathbf{z}\sim\mu}\left(p(\mathbf{z}) - h(\mathbf{z})\right)^2 \geq c.$$

This means that there exists some function that can be approximated to a sufficient degree of accuracy with a 3 layer ANN and a polynomial number of units in each layer, however, to express that same function with a 2 layer ANN exponentially many units are needed and even then, the expected error may be large. In a nutshell, you gain a lot of expressive power by moving from 2 layers to 3. (Lu et al. 2017) experiment with the opposite approach from the width standpoint using ReLU networks, their results corroborate the power of depth even further. In our network architecture we favour the result of Theorem 4.2 and utilise depth, in doing so the number of units needed in each layer to produce good results can be controlled more, along with a reduced propensity to overfit to the training data.

4.2 Representing Financial Models with Neural Networks

Let $M^{n_1}_{n_2,n_3}(w, \delta)$ define a neural network that has n_1 layers, input dimension n_2, output dimension n_3 that takes in as input a vector of model parameters $\delta \in \mathbb{R}^{n_2}$. In order to use the network to price options the following setup will be used. Let $P(\delta)$ denote the pricing map of the model in question that takes in a vector of input parameters δ, the goal will be to learn a

neural network $M^{n_1}_{n_2,n_3}(w^*, \delta)$ such that

$$M^{n_1}_{n_2,n_3}(w^*, \delta) \approx P(\delta), \text{ where } w^* = \underset{w}{\operatorname{argmin}}\, C\left(M^{n_1}_{n_2,n_3}(w, \delta), P(\delta)\right)$$

across all training examples for a suitably defined cost function, C, such as the mean squared error. In the case of the mean squared error, the loss on the $i'th$ training example is given by

$$C^{(i)} = \left(M^{n_1}_{n_2,n_3}(w, \delta_i) - P(\delta_i)\right)^2.$$

For the Black-Scholes model there are 6 input parameters the stock price, strike price, volatility, risk-free interest rate, the dividend yield, and the expiry date. Denote the Black-Scholes pricing map as

$$P_{BS}(S, K, \sigma, r, d, T) = P_{BS}(\delta)$$

where $\delta = [S, K, \sigma, r, q, T]$ is the vector of input parameters. The approximation becomes:

$$M^{n_1}_{n_2,n_3}(w^*, \delta) \approx P_{BS}(\delta) \in \mathbb{R}, \text{ where } w^* = \underset{w}{\operatorname{argmin}}\, C\left(M^{n_1}_{n_2,n_3}(w, \delta), P_{BS}(\delta)\right).$$

For approximating the implied volatility let $\sigma_{imp}(\delta)$ be the implied volatility map for the model being used where δ is the vector of model parameters. The task is to learn a neural network such that

$$M^{n_1}_{n_2,n_3}(w^*, \delta) \approx \sigma_{imp}(\delta) \in \mathbb{R}, \text{ where } w^* = \underset{w}{\operatorname{argmin}}\, C\left(M^{n_1}_{n_2,n_3}(w, \delta), \sigma_{imp}(\delta)\right).$$

4.2.1 Image-Based Implicit Method

So far, the neural network setups that have been described take a traditional pointwise learning approach (Horvath et al. 2019) in which there are explicit input vectors that map to a single output, such as an option price. (Horvath et al. 2019) introduce what they refer to as image-based implicit learning, the idea is to train the model using model parameters as inputs and a grid in which each point on the grid represents a 'pixel' in an image as target variables (Horvath et al. 2019). For implied volatilities, this corresponds to grid of a implied volatilities defined by the strike price and the maturity date. In this setup, given a training set of input

vectors (including expiry T and strike price K) δ, the task is to learn a neural network

$$M_{n_2,n_3}^{n_1}(w^*, \hat{\delta}) \approx \sigma_{imp}(T_i, K_j)_{i=1,\dots,l, j=1,\dots,m} \in \mathbb{R}^{l \times m}$$

where $\hat{\delta} = \delta \setminus \{T, K\}$. So the inputs into the neural network do not explicitly include the expiry date or the strike price, they are implicitly part of the implied volatility grids used to train the network (Horvath et al. 2019).

To summarise, the model parameters excluding expiry date and the strike prices are the inputs to the network, and an implied volatility 'grid', or surface, corresponding to a total of $l \cdot m$ individual implied volatilites is the output of the network. The number of implied volatilities generated by the network for a given set of input parameters is O(lm), so a larger grid implies more volatilites and vice-versa.

Some of the benefits of this approach compared to the more traditional pointwise approach are:

- It may be more suitable when considering multiple options (on the same asset) with varying maturities and strikes as multiple strikes and maturities are evaluated at once.
- It has a better chance of preserving distinctness between the model parameters and the implied volatilities since for any given input vector it is much more unlikely that a different set of inputs will generate the same set of implied volatilities than would be the case if mapping to just one output (Horvath et al. 2019).
- It is easier to add to the network: if more strike prices and maturities need to be considered, the corresponding implied volatilities can be appended to the end of the existing implied volatilities without the need to add more individual training points.
- More information is given to the network. The network can make use of the multiple implied volatilities to learn the non linear relationship between them themselves as well as between the input parameters and the outputs.

CHAPTER 5

Data

In this chapter we explain our data flow, in particular, what type of data is required, how it is generated, it's structure and the pre-processing routine. We expand on the data-driven part of the flow diagram presented in figure 0.1 to give a clear picture of what our data-flow process entails in figure 5.1.

5.1 Data Generation

Each of the models considered in the research require different types of data and understanding the data that is fed into the neural networks is extremely important. If we cannot understand the data, we cannot understand the results, thus, we find it useful to briefly explain where our data comes from and how it is generated.

5.1.1 Black-Scholes Data

For completeness, we choose to work with the Black-Scholes equation with dividends shown in equation (3.7). Accordingly, we require data for the dividend yield, risk-free interest rate, spot price, strike price, maturity time and volatility. Due to the analytical nature of the Black-Scholes framework we are able to synthetically generate accurate training data for the

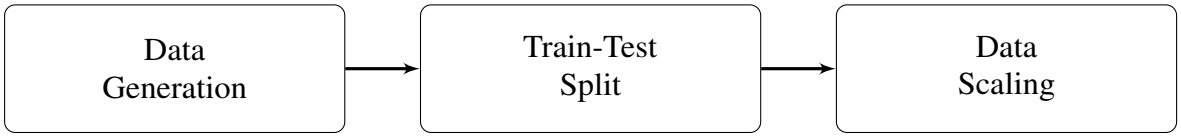


FIGURE 5.1. *Data-flow process.*

TABLE 5.1. *Black-Scholes Parameters. Cyan row(s) = ANN Input. Gray row(s) = ANN Output.*

Parameter	Range
Spot Price (S)	[1, 50)
Strike Price (K)	[40, 55)
Risk-free Rate (r)	[0, 1)
Dividend Yield (q)	[0, 1)
Volatility (σ)	[0, 1)
Maturity in years (T)	[0, 1)
Option Type (Call or Put)	$\{-1, 1\}$
Option Price (y)	$\mathbb{R}^+$

neural networks. We choose work to with 3,000,000 uniformly randomly generated samples of each model parameter over the ranges shown in table 5.1 used to generate a corresponding option price. The gray row represents the output variable which in this case is the option price and the cyan rows are inputs to the neural network. The option prices are calculated by implementing formulae (3.8) and (3.11), in Python. Note that S and the output option price, y, are first scaled by dividing by K, so K itself does enter the network.

5.1.2 SABR Data

For the SABR model we focus mainly on the image-based approach, however, we also implement the pointwise approach first as a proof of concept that a neural network can indeed learn the complex implied volatility formulae given by definitions 3.8 and 3.9. This means the structure of the data used for each method is different.

5.1.2.1 SABR Pointwise Data

Data for the model parameters are sampled uniformly randomly and used to calculate the corresponding implied volatility with Python implementations of the analytical volatility expansions given by definitions 3.8 and 3.9. For the SABR model we require data for the forward, strike price, expiry, initial volatility, volatility of volatility, β and correlation. Again, we choose to work with 3,000,000 data points, and the parameter ranges chosen are shown in table 5.2. The gray row shows the output variable we are trying to predict, the implied

TABLE 5.2. *Pointwise SABR Parameters. Cyan row(s) = ANN Input. Gray row(s) = ANN Output.*

Parameter	Range
Spot forward (F)	[0.3, 1.5)
Strike Price (K)	[0.5, 2)
Initial Volatility (σ)	[0.2, 0.8)
Volatility of Volatility (α)	[0.2, 0.5)
Correlation (ρ)	[-0.95, 0.95)
Beta (β)	$\{0, 0.5, 1\}$
Maturity in years (t_{exp})	[0.1, 5)
Implied Volatility (σ_{imp})	$\mathbb{R}^+$

volatility and the cyan rows are inputs to the network. Notice that β is not fed into the neural network, this is because we are prespecifying its value before the various experiments as this parameter is usually predetermined in practice (Hagan et al. 2002).

5.1.2.2 SABR Image-based Data

For the image-based method the same model parameters are used to generate the training data but the structure of the data is slightly more complicated. Table 5.3 shows which parameters are used for input (the cyan rows) and output (gray row). In particular, as explained in section 4.2.1, the network does not explicitly take in the maturity date and strike price as inputs, instead they are implicitly part of the implied volatility grid used to train the network. For each input vector of parameters the implied volatilites are calculated by iteratively running the calculation altering only the maturity date and strike price. Per input vector $[F, \sigma, \alpha, \rho]$ there are $l \cdot m$ implied volatility outputs where l is the number of maturity dates and m is the number of strike prices. Adhering to the design patterns of (Horvath et al. 2019) we choose $l = 8$ and $m = 11$ giving us a total of 88 implied volatilites. Using $200,000$ samples we choose to store all of this data in a matrix $H \in \mathbb{R}^{200000\times 92}$ where we denote each row as a 'data point' consisting of the 4 input parameters and the 88 implied volatilities. With $200,000$ data points there are in fact a total of $200000 \cdot 92 = 18400000$ individuals pieces of data in total (model parameters and individual implied volatilities).

```
# Define the parameter ranges
sample_size = 200000
K = np.linspace(0.5, 2.0, 11) # Strike Prices
texp = np.linspace(0.1, 5, 8) # Expiry
F = np.random.uniform(0.3, 1.5, sample_size) # Spot Forward Price
sigma = np.random.uniform(0.2, 0.8, sample_size) # Initial Volatility
alpha = np.random.uniform(0.2, 0.5, sample_size) # Vol of vol
rho = np.random.uniform(-0.95, 0.95, sample_size) # Correlation

# Matrix to store the data
matrix = np.zeros((sample_size, len(texp)*len(K) + 4))

# Generate the dataset in the required matrix format
for i in range(sample_size):
    imp_vol = []
    model_params = []

    for j in range(len(texp)):
        for k in range(len(K)):
            vol = impvol_lognormal(K[k], F[i], texp[j], sigma[i], 1,
                                                    alpha[i], rho[i])
            imp_vol.append(vol)

    imp_vol = np.array(imp_vol)
    model_params.append(F[i])
    model_params.append(alpha[i])
    model_params.append(rho[i])
    model_params.append(sigma[i])
    model_params = np.array(model_params)
    row = np.concatenate((model_params, imp_vol))
    matrix[i, :] = row
```

FIGURE 5.2. *Generating The Data Matrix in Python*

It will be beneficial to look a bit closer at the way the implied volatility grid data is organised. For a given vector of input parameters $[F, \sigma, \alpha, \rho]$ we have 2 consecutive for loops over the expiry dates and the strike prices respectively as shown in figure 5.2. This results in 11 implied volatilities per expiry date, one for each strike price, providing 88 volatilities per input vector. Each of these 11 implied volatilities corresponds to an individual volatility smile, and all 8 sets of the 11 form the implied volatility grid use to train the network. Once we have generated the matrix, we store it locally using the NumPy .npy extension.

Taking a vector of 88 implied volatilites and reshaping it into an 8×11 grid gives a clearer insight into the structure which we illustrate in table 5.4. Each element,(l, m), of the grid is an implied volatility value for the parameters $\left[F, \sigma, \alpha, \rho, t_{\exp_l}, K_m\right]$. Therefore, each row

TABLE 5.3. *Imaged Based SABR Parameters. Cyan row(s) = ANN Input. Gray row(s) = ANN Output.*

Parameter	Range
Spot forward (F)	[0.3, 1.5)
Strike Price (K)	[0.5, 2)
Initial Volatility (σ)	[0.2, 0.8)
Volatility of Volatility (α)	[0.2, 0.5)
Correlation (ρ)	[-0.95, 0.95)
Beta (β)	$\{0, 0.5, 1\}$
Maturity in years (t_{exp})	[0.1, 5)
Implied Volatility Grid ($\sigma_{imp} \in \mathbb{R}^{l \times n}$)	$\mathbb{R}_{+}^{l \times n}$

TABLE 5.4. *Implied volatility grid for an input vector $[F, \sigma, \alpha, \rho]$ used to train the ANNs. Row indices = maturity dates, columns indices = strike prices.*

		K										
		0.5	0.65	0.8	0.95	1.1	1.25	1.4	1.55	1.7	1.85	2.0
t_{exp}	0.1	·	·	·	·	·	·	·	·	·	·	·
	0.8	·	·	·	·	·	·	·	·	·	·	·
	1.5	·	·	·	·	·	·	·	·	·	·	·
	2.2	·	·	·	·	·	·	·	·	·	·	·
	2.9	·	·	·	·	·	·	·	·	·	·	·
	3.6	·	·	·	·	·	·	·	·	·	·	·
	4.3	·	·	·	·	·	·	·	·	·	·	·
	5.0	·	·	·	·	·	·	·	·	·	·	·

corresponds to an implied volatility smile, and the entire matrix together corresponds to an implied volatility grid. As a more concrete example consider the full grid shown in table 5.5, the implied volatilities highlighted in blue form the volatility smile across the 11 strike prices with maturity date 0.1. We plot this particular smile in figure 5.3. To summarise, there are 8 such smiles per volatility grid and the entire grid is the target variable for the neural network, so for a given input vector the network is predicting 88 outputs. Finally, it is important to point that the dimensions of the grid can be chosen to be as large or as small as needed, of course, the more maturity dates and strike prices used to define the grid the more outputs the neural network will be predicting.

TABLE 5.5. *Implied volatility grid example.*

		K										
		0.5	0.65	0.8	0.95	1.1	1.25	1.4	1.55	1.7	1.85	2.0
t_{exp}	0.1	0.567	0.559	0.551	0.545	0.541	0.538	0.537	0.538	0.540	0.544	0.549
	0.8	0.571	0.562	0.555	0.549	0.545	0.542	0.541	0.541	0.544	0.547	0.553
	1.5	0.575	0.566	0.559	0.553	0.549	0.546	0.545	0.545	0.547	0.551	0.556
	2.2	0.579	0.570	0.563	0.557	0.552	0.549	0.548	0.549	0.551	0.555	0.560
	2.9	0.583	0.574	0.567	0.561	0.556	0.553	0.552	0.553	0.555	0.559	0.564
	3.6	0.587	0.578	0.571	0.564	0.560	0.557	0.556	0.556	0.559	0.563	0.568
	4.3	0.591	0.582	0.574	0.568	0.564	0.561	0.560	0.560	0.562	0.566	0.572
	5.0	0.595	0.586	0.578	0.572	0.567	0.565	0.563	0.564	0.566	0.570	0.576

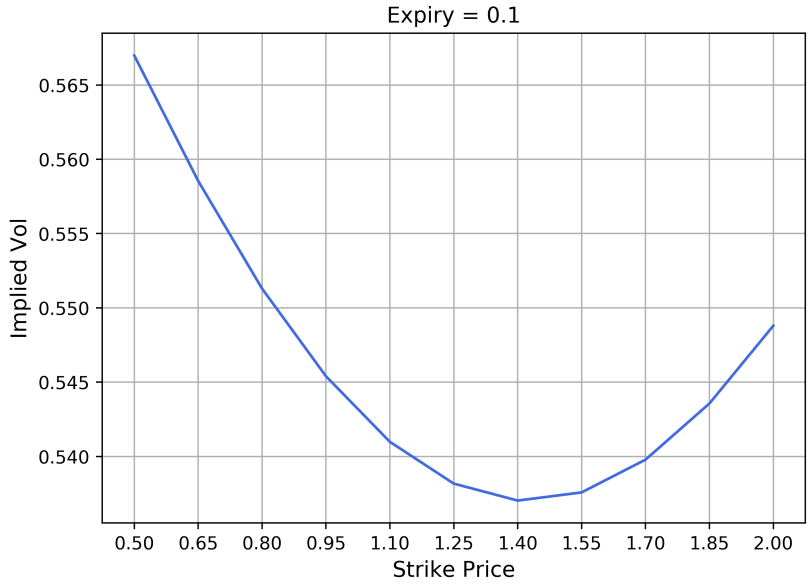


FIGURE 5.3. *Example SABR volatility smile from the volatility grid.*

5.2 Data Preprocessing

Since we are working with data over a variety of ranges it is necessary to scale/normalise the data to ensure that each parameter equally influences the results and therefore the accuracy of the neural networks. Additionally, if the input data is scaled the cost function is typically more symmetrical on average. This results in faster optimisation of the cost function than would be the case if the data was not appropriately scaled because there is less oscillation on average on the path toward the minimum.

For the SABR pointwise regime we normalise the input parameters by subtracting the mean and dividing by the standard deviation across each parameter, this reduces each parameter set to a 0 mean and a unit variance. For an input parameter value δ (for example strike price) we use the following transformation:

$$\frac{\delta - \mu}{s} \tag{5.1}$$

where μ is the mean across all samples for the particular parameter and s is the standard deviation.

For the image-based regime we normalise the input parameters using the method presented in (Horvath et al. 2019, p21). For an input parameter value δ we perform:

$$\frac{2\delta - \max(\delta) - \min(\delta)}{\max(\delta) - \min(\delta)} \tag{5.2}$$

where $\max(\delta)$ and $\min(\delta)$ are the maximum and minimum values of the parameter across all examples respectively. To scale the implied volatility grids we use the same method as in equation (5.1) for each grid, where the mean and standard deviation are across all the implied volatility grids.

The Black-Scholes variables are kept the same except we scale the stock price, S, and the option price, y by dividing each by the strike price K as follows, $S \mapsto S/K$ and $y \mapsto y/K$.

CHAPTER 6

Network Design

In subsequent explanations when we refer to the number of layers, we are referring to all layers excluding the input layer, so we are considering the hidden layers and the output layer where layer 1 is the first hidden layer and so on.

6.1 Black-Scholes Network Architecture

Keeping in line with Theorem 4.2 (The Power of Depth) we choose to use 3 hidden layers. Since this is a regression problem in which we are trying to map inputs to a continuous output, the mean squared error loss function is chosen so that larger deviations from the truth are penalised more than smaller.

An epoch is 1 pass through the training data. Slowly reducing the learning rate as the number of epochs grows means that initially when the learning rate is relatively large, we benefit from faster learning as the steps taken toward the optimum are larger, but as it is gradually decreased the steps become increasingly smaller meaning the algorithm oscillates in a tighter bound around the optimum as learning approaches convergence. This process is known as learning rate decay and it helps prevent the loss from diverging later on in training. We select the Adam optimiser shown in Algorithm 2 for its adaptive properties, however, in its native form Adam will adapt the learning rate at any stage to within the range $[0, init]$ where $init$ is the initial learning rate (default init = 0.001). To ensure the upper bound on the learning rate at each epoch is reduced as the number of epochs grows we use Adam with a time-based

learning rate decay that follows the update rule

$$lr_{n+1} = \frac{lr_n}{1 + \epsilon n}, \tag{6.1}$$

where ϵ is the decay rate which we set as $\epsilon = \frac{init}{N}$, $n = 1, \ldots, N$ is the current epoch of training and N is the total number of epochs.

6.1.1 Black-Scholes Neural Network 1

This network is designed to learn the pricing formulae for puts (definition 3.2) and calls (definition 3.1) simultaneously. This is achieved by adding an extra input to the neural network, a binary flag that is 1 if the option is a call and -1 if the option is a put in the hopes that the ANN is able to discern between the two. We use a batch size of 128 and train for 50 epochs.

1. Hidden layer 1 with 128 nodes and ELU (4.2) activation function.
2. Hidden layer 2 with 128 nodes and ELU activation function.
3. Hidden layer 3 with 128 nodes and ELU activation function.
4. Output layer with a linear activation function (4.1) and output dimension equal to 1.

The configuration results in 34,049 trainable parameters. A breakdown of the network and the trainable parameters is given by the Keras summary shown in figure 6.1.

6.1.2 Black-Scholes Neural Network 2

We implement some changes suggested by (Itkin 2019) to ensure the output is positive and has smooth first and second derivatives. The suggested setup uses 2 hidden layers instead of 3 and the MELU (see definition 4.4) activation function on the hidden layers. Instead, we propose a slightly different architecture utilising the ISRLU (Carlile et al. 2017) activation function but still adhering to the suggested principles.

Layer (type)	Output Shape	Param #
input_1 (InputLayer)	(None, 6)	0
dense_1 (Dense)	(None, 128)	896
dense_2 (Dense)	(None, 128)	16512
dense_3 (Dense)	(None, 128)	16512
dense_4 (Dense)	(None, 1)	129

Total params: 34,049
Trainable params: 34,049
Non-trainable params: 0

FIGURE 6.1. *Black-Scholes neural network 1 - Keras summary.*

The *inverse square root linear unit (ISRLU)* activation function is defined as

$$\phi(\alpha, x) = \begin{cases} \frac{x}{\sqrt{1+\alpha x^2}}, & \text{if } x < 0, \\ x & \text{if } x \geq 0. \end{cases} \tag{6.2}$$

In addition to ISRLU and its derivative being monotonic, it also has the desirable property that it is approximately the identity at the origin, these 3 features all help speed up convergence on the network (Aghdam and Heravi 2017). Importantly, ISRLU is also of class C^2 meaning it has smooth first and second derivatives (Carlile et al. 2017). To ensure the positivity of the ANN outputs we use the SoftPlus (4.5) activation function on the output layer which takes values in the range $(0, \infty)$. Finally, we utilise 3 hidden layers instead of 2 as a consequence of Theorem 4.2. The batch size used is 64 and we train for 50 epochs.

1. Hidden layer 1 with 128 nodes and ISRLU activation function.
2. Hidden layer 2 with 128 nodes and ISRLU activation function.
3. Hidden layer 3 with 128 nodes and ISRLU activation function.
4. Output layer with SoftPlus (4.5) activation function and output dimension equal to 1.

The configuration results in 34,049 trainable parameters. The Keras summary is shown in 6.2.

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_1 (InputLayer)         (None, 6)                 0
_________________________________________________________________
dense_1 (Dense)              (None, 128)               896
_________________________________________________________________
dense_2 (Dense)              (None, 128)               16512
_________________________________________________________________
dense_3 (Dense)              (None, 128)               16512
_________________________________________________________________
dense_4 (Dense)              (None, 1)                 129
=================================================================
Total params: 34,049
Trainable params: 34,049
Non-trainable params: 0
_________________________________________________________________
```

FIGURE 6.2. *Black-Scholes neural network 2 - Keras summary.*

6.2 SABR Network Architecture

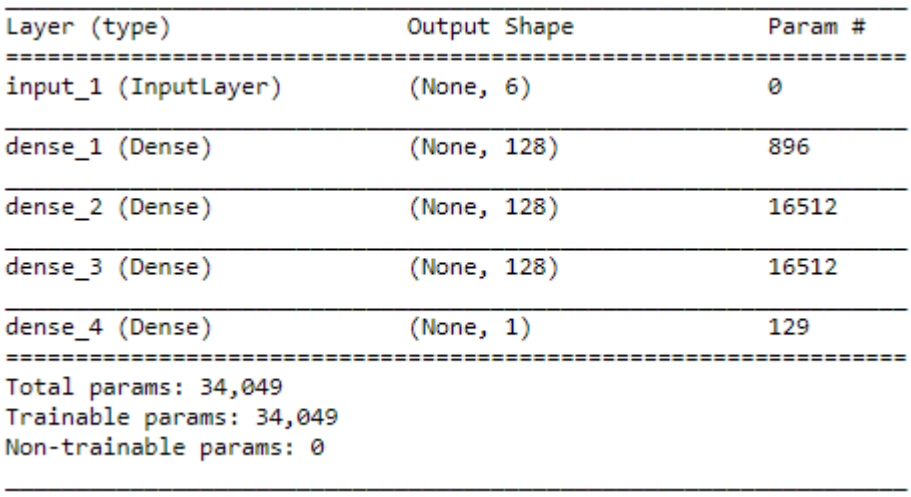
Once again we use 3 hidden layers and stick with the mean squared error loss function coupled with the Adam optimiser with a time-based learning rate decay for the same reasons described in section 6.1.

6.2.1 SABR Pointwise Neural Network

The pointwise regime follows the same idea as the Black-Scholes networks since the network is only trying to predict a single output given an input vector. We use a batch size of 128 and train for 50 epochs.

1. Hidden layer 1 with 128 nodes and ELU (4.2) activation function.
2. Hidden layer 2 with 128 nodes and ELU activation function.
3. Hidden layer 3 with 128 nodes and ELU activation function.
4. Output layer with a SoftPlus activation function and output dimension equal to 1.

The configuration results in 34,049 trainable parameters. The Keras summary for this model is shown in figure 6.3.



Layer (type)	Output Shape	Param #
input_1 (InputLayer)	(None, 6)	0
dense_1 (Dense)	(None, 128)	896
dense_2 (Dense)	(None, 128)	16512
dense_3 (Dense)	(None, 128)	16512
dense_4 (Dense)	(None, 1)	129

Total params: 34,049
Trainable params: 34,049
Non-trainable params: 0

FIGURE 6.3. *SABR Pointwise regime network - Keras summary.*

6.2.2 SABR Image-based Neural Network

For the image based regime we mainly subsume the design patterns of (Horvath et al. 2019, p20), however, we choose the mean squared error loss function and utilise 80 neurons in each hidden layer as opposed to 30. Furthermore, we use Adam with learning rate decay instead of vanilla Adam. Finally, we chose a batch size of 128 as this choice yielded the best performance and we train for 500 epochs.

1. Hidden layer 1 with 80 nodes and ELU (4.2) activation function.
2. Hidden layer 2 with 80 nodes and ELU activation function.
3. Hidden layer 3 with 80 nodes and ELU activation function.
4. Output layer with a linear activation function (4.1) and output dimension equal to 88.

The configuration results in 20,488 trainable parameters. The Keras summary for this model is shown in figure 6.4. Since this network has multiple outputs we make use of the Keras functional API.

Layer (type)	Output Shape	Param #
input_1 (InputLayer)	(None, 4)	0
dense_1 (Dense)	(None, 80)	400
dense_2 (Dense)	(None, 80)	6480
dense_3 (Dense)	(None, 80)	6480
dense_4 (Dense)	(None, 88)	7128

Total params: 20,488
Trainable params: 20,488
Non-trainable params: 0

FIGURE 6.4. *SABR Image-based regime network - Keras summary.*

6.3 Model Selection and Evaluation

The neural networks need to be tested against various accuracy and performance metrics in order to justify the experiments. The choice of how we evaluate our models dictates the choice of which model, hypyerparameterrs, and algorithms we ultimately decide to use. The problem of predicting option prices or implied volatility is a regression problem, therefore, we choose the mean squared error (MSE), mean absolute error (MAE), the coefficient of determination (R^2) and mean absolute percentage error (MAPE) as accuracy metrics. Let Y_{true} be the true value and Y_{pred} be the predicted value.

$$MSE = \frac{1}{n}\sum_{i=1}^{n}\left(Y_{true_i} - Y_{pred_i}\right)^2 \tag{6.3}$$

$$MAE = \frac{1}{n}\sum_{i=1}^{n}\left|Y_{true_i} - Y_{pred_i}\right| \tag{6.4}$$

$$MAPE = \frac{100\%}{n}\sum_{i=1}^{n}\left|\frac{Y_{true_i} - Y_{pred_i}}{Y_{true_i}}\right| \tag{6.5}$$

$$R^2 = 1 - \frac{\sum_{i=1}^{n}\left(Y_{true_i} - Y_{pred_i}\right)^2}{\sum_{i=1}^{n}\left(Y_{true_i} - \overline{Y_{true}}\right)^2} \tag{6.6}$$

6.3.1 Train-Test Split

Of the data generated we choose to keep aside 15% as unseen test samples to evaluate the network performance. Of the 85% of the training data used to train the model, 65% is used to explicitly train the network and 20% is used to validate the model. After each epoch of training, the mean squared error on the training set and the validation set is calculated and after the total number of epochs the model that performs the best overall across the training and validation losses is selected. The process is shown in Algorithm 3 where mse denotes the mean squared error. We keep track of the validation loss at each epoch using *Keras Callbacks* which allow us to view the models internal state during training. This is crucial to ensure that the networks are able to perform well out of sample as well as in sample, after all, the goal is to be able to make accurate predictions on unseen data.

Algorithm 3: Train-Validate-Test

Input: T = Training Set, V = Validation Set, ζ = Test Set, ANN = Neural Network, $ecochs \in \mathbb{Z}$, $\theta \in \mathbb{R}$ (current lowest mse)

$\theta \leftarrow \infty$;
for *i = 1:epochs* **do**
  Train ANN_i on T;
  Calculate mse_i^T on T;
  Calculate mse_i^V on V;
  if $mse_i^V < \theta$ **then**
    $\theta = mse_i^V$;
  end if
end for
Save model ANN^* that yields the lowest validation error θ;
Calculate mse on the test set T using ANN^* to evaluate model performance;
return ANN^*;

6.3.2 Cross Validation

Cross validation provides a robust procedure to quantify how well a model is able to generalise. It is a highly computationally expensive task and since the training of ANNs is costly in itself, it is often neglected in deep learning research. However, we recognise its benefit to the accuracy and error analysis and deem it appropriate to use. We implement *K-Fold* cross

validation in which the training data is split into K sets and the model is trained, that is, the network weights and biases are learned, using $K-1$ sets of data and validated on the remaining set. The learned model is then applied to the separate unseen test data and the corresponding loss recorded. The process is repeated K times where each set will have been used as the validation set once. At the end, this provides us with K losses on the test set which can be averaged over to assess the model performance on average across all K models, each trained and validated on a different combination of data. The procedure is shown in Algorithm 4 where mse denotes the mean squared error. The Scikit-Learn Python library has a KFold method that gives us the indices of training and testing data to be used at each fold for a training data set.

Algorithm 4: K-Fold Cross Validation.

Input: T = Training Set, ANN = Neural Network, err = List of mse scores
Divide T into K sets $\{T_i\}_{i=1}^{K}$;
for *i = 1:K* **do**
    $H = T - T_i$;
    Train ANN_i on H;
    Calculate mse_i^V on validation set T_i;
    Add mse_i^V to the list err;
end for
Compute the average loss $\overline{mse}$ = $\frac{1}{K}\sum_{i=1}^{K} err_i$ over the test sets;
return $\overline{mse}$;

CHAPTER 7

Results and Analysis

All training and evaluation was done on a system with an Intel Core i3 2100 at 3.1GHz, 16GB dual channel DDR3 RAM at 665MHz using Windows 10 with a 64bit installation of Python through Anaconda. In subsequent analysis, in-sample refers to the training data and out-sample refers to the unseen test data, MSE refers to mean squared error, MAE refers to mean absolute error and R^2 refers to the coefficient of determination.

7.1 Black-Scholes Option Pricing

We present the results for option pricing under the Black-Scholes framework for each of the networks described in section 6.1.

7.1.1 Black-Scholes Neural Network 1 Results

Figure 7.1 shows out-sample and in-sample scaled analytical option prices and the scaled neural network predicted prices. The line plotted in red indicates all points at which the analytical price exactly matches the ANN predicted price. It is clear from the plotted prices (in black) that the ANN is able to accurately predict the Black-Scholes option prices for calls and puts. The distribution of the errors, specifically, the difference between the ANN predicted price and the analytical price is shown in figure 7.2. The fact that the network is able to predict the prices of options with excellent accuracy also tells us that it is able to distinguish correctly between the call and put option pricing formulae given the input flag (1 for calls and -1 for puts).

TABLE 7.1. *Error scores for Black-Scholes Neural Network 1.*

	MSE	MAE	R^2
Out-sample	3.78958×10^{-7}	0.00040	0.999994
In-sample	3.81794×10^{-7}	0.00040	0.999994

The R^2 value for the out of sample predictions is 0.999994, the MSE achieved is 3.78958×10^{-7} or 0.0038 basis points (bps) and the MAE achieved is 0.00040. These results are summarised in table 7.1. The average time taken per epoch of training was 76 seconds and we track the mean squared loss value across all 50 epochs for the training and validation data which is shown in figure 7.3. Initially, the loss values fall rapidly then begin to level off as the number of epochs grows. If there was a sign of overfitting to the training data the validation loss line would begin to pull away and diverge from the training loss line, we can see this is not the case, this, coupled with the out of sample accuracy results confirm there is no overfitting.

The cross validation (CV) scores over 5-folds are shown in table 7.2, there is an average MSE and MAE of 7.82140×10^{-7} and 0.00057 respectively. We attribute the marginally higher errors than those seen in table 7.1 to the fact that cross validation was performed using 30 epochs at each stage as opposed to the 50 used to train the network originally so as to save on computation time. The cross validation scores over the 5 folds are extremely consistent with each other showing that this network architecture is able to consistently, and accurately, predict the option prices.

Whilst the accuracy of the network is impressive, there are 2 main points that need to be addressed. The ELU function does not have smooth first and second derivatives and the network does not guarantee the positivity of the output due to the linear activation function (see definition 4.1) used on the output layer. These are the subject of the next set of results for Black-Scholes Neural Network 2.

TABLE 7.2. *5-Fold cross validation scores for Black-Scholes Neural Network 1 using 30 epochs.*

	MSE	MAE
Fold 1	8.35265×10^{-7}	0.00058
Fold 2	6.70458×10^{-7}	0.00054
Fold 3	6.02270×10^{-7}	0.00048
Fold 4	8.69874×10^{-7}	0.00061
Fold 5	9.32832×10^{-7}	0.00065
Mean	7.82140×10^{-7}	0.00057

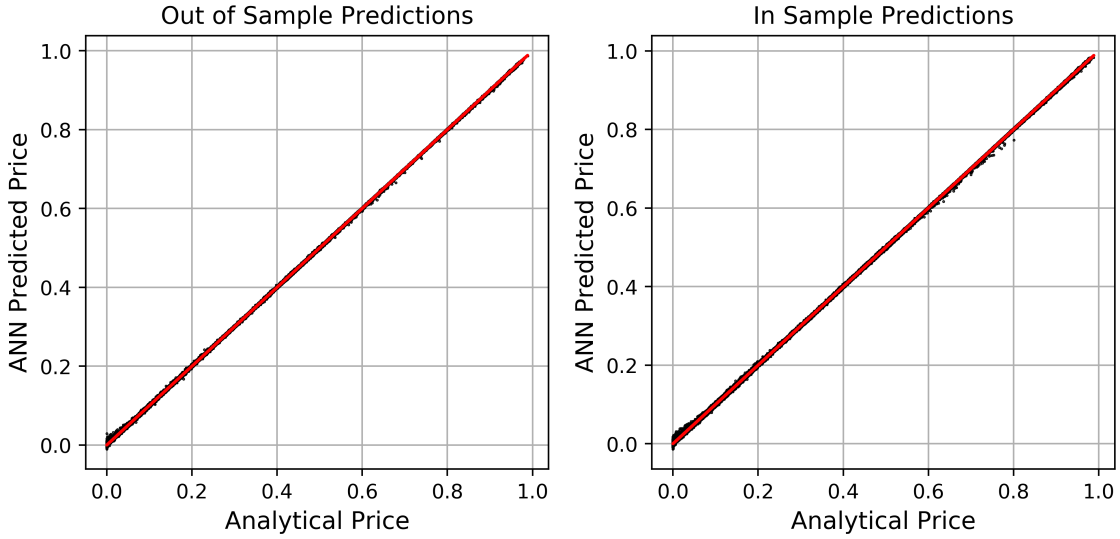


FIGURE 7.1. *Black-Scholes Neural Network 1 - Out of sample and in sample prices (analytical and ANN predicted) for European calls and puts, option prices are scaled prices.*

7.1.2 Black-Scholes Neural Network 2 Results

Figure 7.4 shows the out of sample and in sample results for Black-Scholes neural network 2, again the prices shown are scaled analytical prices and scaled ANN predicted prices. The difference between this network and Black-Scholes Neural Network 1 is that these outputs are certain to be positive and have smooth first and second derivatives allowing Greeks to be calculated. The ANN is able to accurately predict option prices for calls and puts as well as distinguish between the call and put pricing formulae based upon on the input flag.

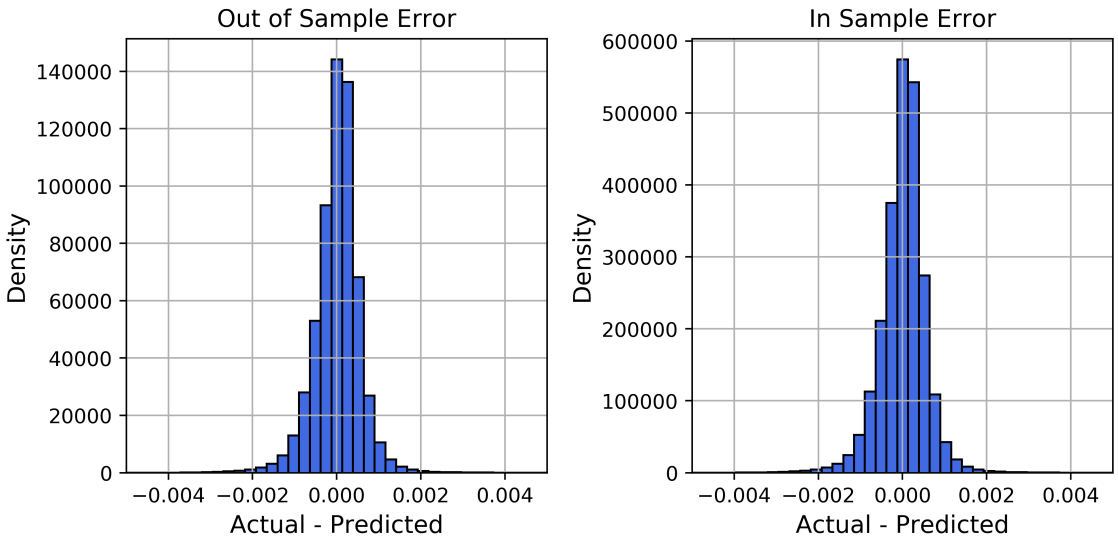


FIGURE 7.2. *Error distribution for Black-Scholes Neural Network 1 predictions: ANN predicted values - analytical values.*

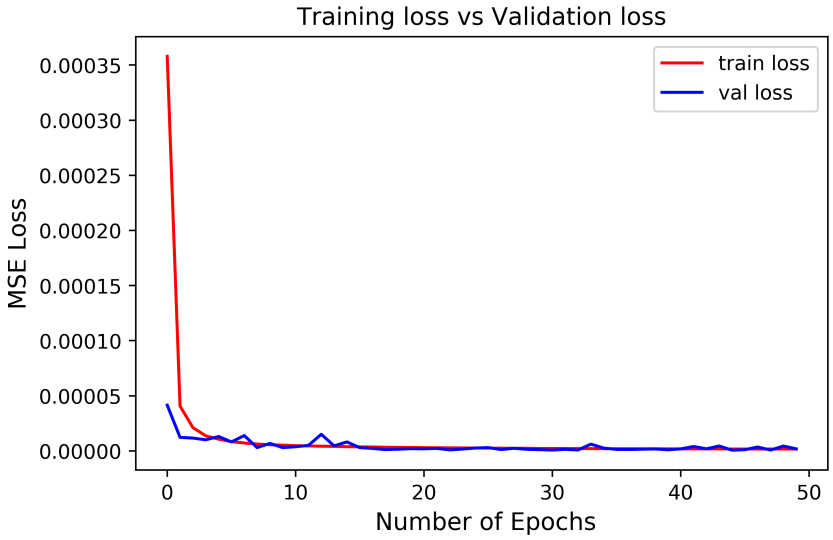


FIGURE 7.3. *Training loss vs Validation loss for Black-Scholes Neural Network 1.*

The R^2 value for out of sample predictions was 0.999989, the MSE is 6.79353×10^{-7} or 0.0068 bps and the MAE achieved is 0.00043. The errors are summarised in table 7.3 and their distribution illustrated in figure 7.5. The average time taken per epoch of training was 75 seconds and we track the training and validation loss over 50 epochs in figure 7.6, both loss lines rapidly drop initially before levelling off. The graph shows there is no evidence of overfitting as the validation loss line does not significantly pull away from the training loss

TABLE 7.3. *Error scores for Black-Scholes Neural Network 2.*

	MSE	MAE	R^2
Out-sample	6.79353×10^{-7}	0.00043	0.999989
In-sample	6.90774×10^{-7}	0.00043	0.999989

TABLE 7.4. *5-Fold cross validation scores for Black-Scholes Neural Network 2 using 30 epochs.*

	MSE	MAE
Fold 1	7.75716×10^{-7}	0.00059
Fold 2	7.16497×10^{-7}	0.00055
Fold 3	8.23186×10^{-7}	0.00062
Fold 4	8.00848×10^{-7}	0.00062
Fold 5	7.26213×10^{-7}	0.00059
Mean	7.68492×10^{-7}	0.00058

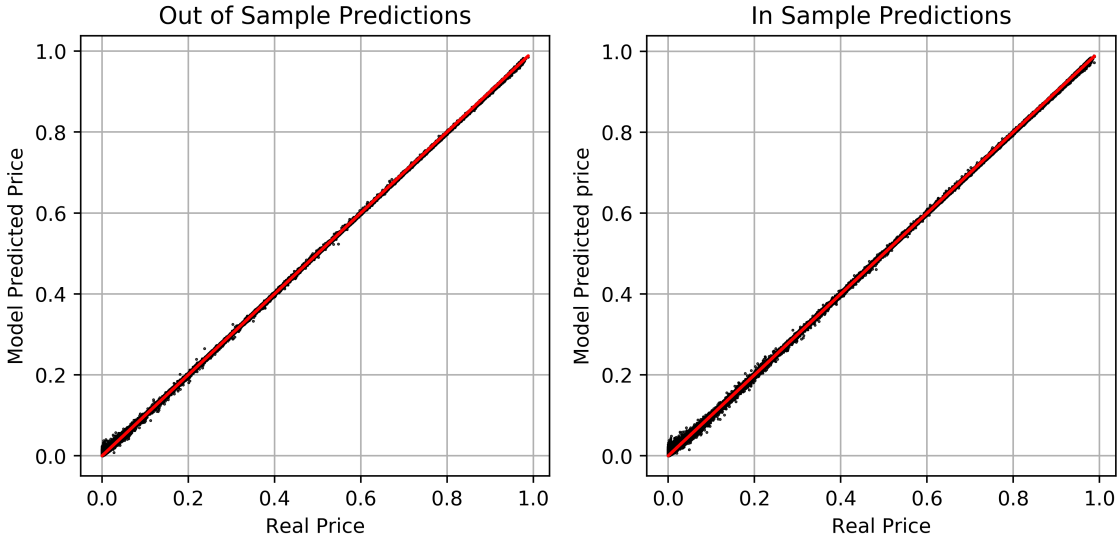


FIGURE 7.4. *Black-Scholes Neural Network 2 - Out of sample and in sample prices (analytical and ANN predicted) for European calls and puts, option prices are scaled prices.*

line, this is corroborated by the out of sample accuracy. Moreover, the 5-fold cross validation scores with an average MSE and MAE of 7.68492×10^{-7} and 0.00058 respectively confirm the robustness of the network.

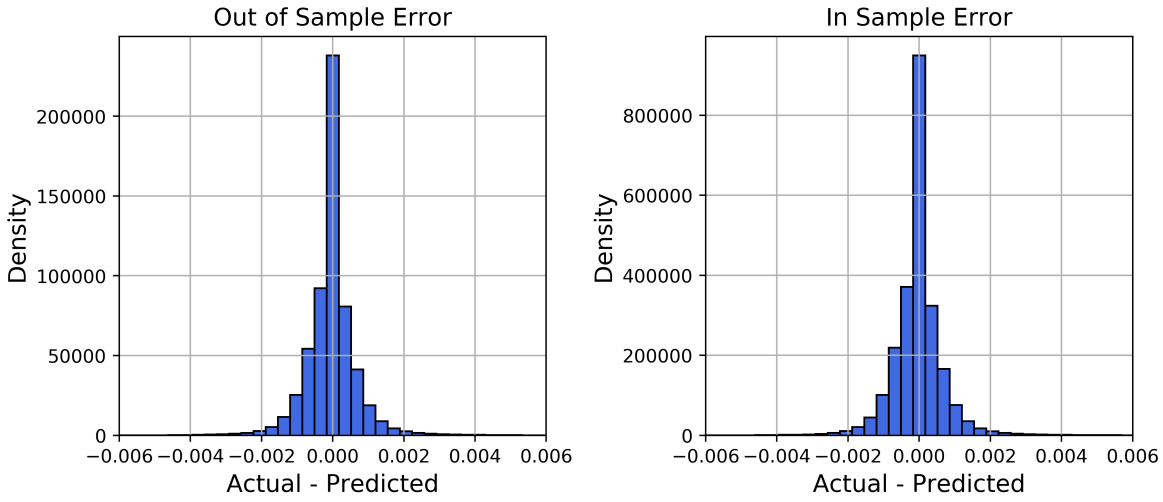


FIGURE 7.5. *Error distribution for Black-Scholes Neural Network 2 predictions: ANN predicted values - analytical values.*

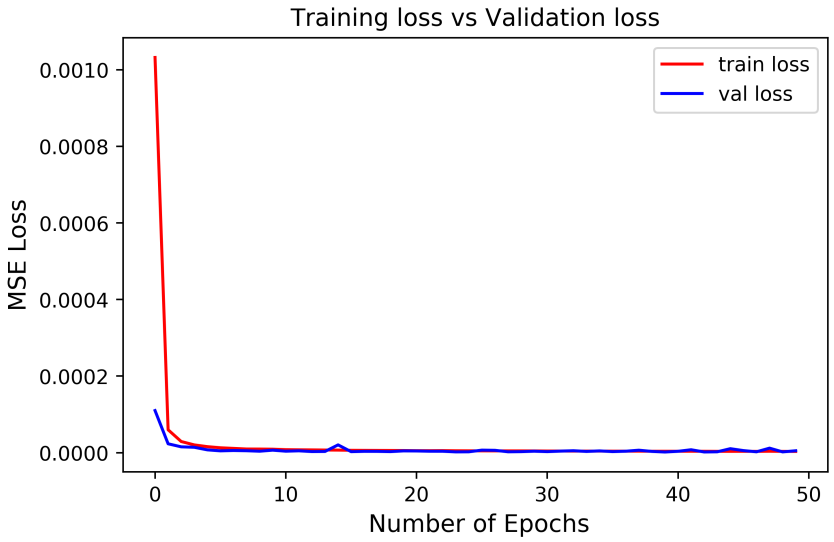


FIGURE 7.6. *Training loss vs Validation loss for Black-Scholes Neural Network 2.*

7.2 Lognormal SABR Model

We present the results of our neural networks used to predict the log normal implied volatility (see definition 3.9) for the SABR model under the pointwise learning regime (single output). The analysis of the results is conducted in a similar fashion as was for the Black-Scholes option pricing results.

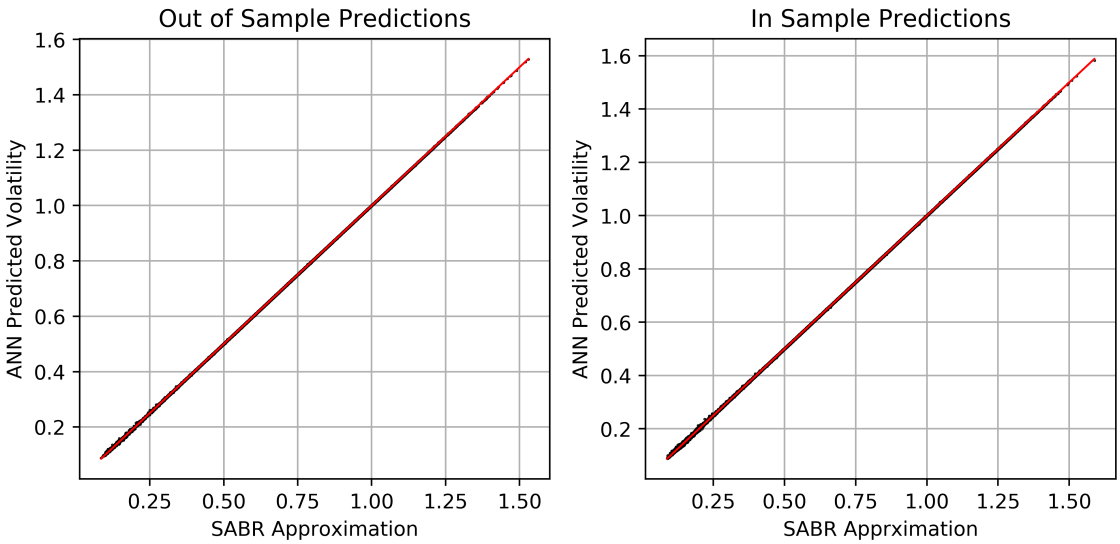


FIGURE 7.7. *Lognormal SABR Pointwise Neural Network - Out of sample and in sample implied volatilities when* $\beta = 1$.

TABLE 7.5. *Error scores for lognormal SABR pointwise Neural Network*

	MSE	MAE	R^2
Out-sample	1.24853×10^{-7}	0.00025	0.999997
In-sample	1.23912×10^{-7}	0.00025	0.999997

7.2.1 Lognormal SABR Pointwise Neural Network Results

For the pointwise regime we only consider the case when the underlying follows a stochastic log-normal distribution, i.e when $\beta = 1$.

Figure 7.7 shows the out sample and in sample lognormal implied volatilites predicted by the ANN and the corresponding SABR approximation (see definition 3.9). Visually, it is clear that the network is very accurate since it is hard to differentiate the plotted points from the red line at which the prediction and the approximation exactly match. The R^2 value between out of sample predictions and the true SABR lognormal implied volatilities is 0.999997, the MSE achieved is 1.24853×10^{-7} or 0.0012 bps and the MAE achieved is 0.00025. The results are summarised in table 7.5 and the distribution of the errors is shown in figure 7.8.

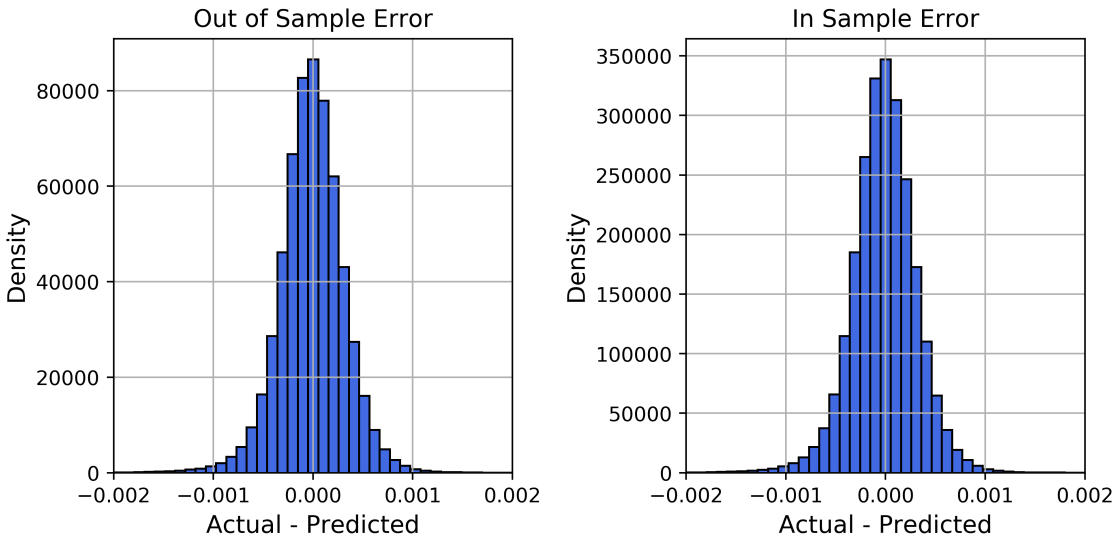


FIGURE 7.8. *Error distribution for lognormal SABR pointwise neural network.*

TABLE 7.6. *5-Fold cross validation scores for Lognormal SABR Pointwise Neural Network using 30 epochs.*

	MSE	MAE
Fold 1	1.74756×10^{-7}	0.00029
Fold 2	2.05858×10^{-7}	0.00032
Fold 3	1.84796×10^{-7}	0.00030
Fold 4	2.01843×10^{-7}	0.00033
Fold 5	1.99345×10^{-7}	0.00032
Mean	1.93320×10^{-7}	0.00031

The average time taken per epoch of training was 49 seconds and the training and validation MSE loss was tracked over 50 epochs and is shown in figure 7.9. Both the validation and training loss lines smoothly converge without any sign of overfitting which is further backed up by the out of sample accuracy. Moreover, the 5-fold cross validation scores shown in table 7.6 show an average MSE and MAE of 1.93320×10^{-7} and 0.00031 respectively, supporting the out of sample evidence that our ANN can accurately and consistently predict the SABR lognormal implied volatility. The results of the pointwise experiment show that our neural network can very accurately learn the SABR log normal implied volatility expansion and provides the proof of concept we require to go ahead with the image-based implicit learning method.

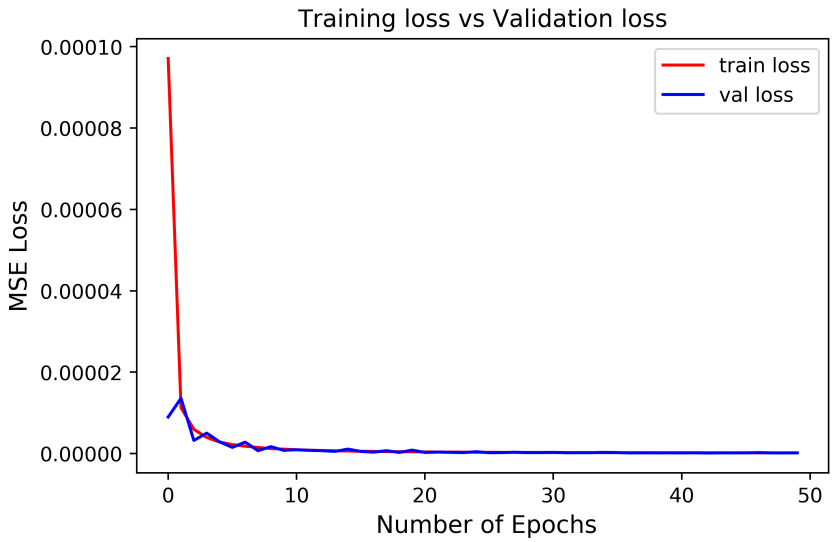


FIGURE 7.9. *Training loss vs Validation loss for lognormal SABR pointwise neural network.*

7.2.2 Lognormal SABR Image-Based Neural Network Results

We present the results for the image-based implicit method for the lognormal SABR model. We reiterate that in this case we are concerned with predicting the entire 8×11 implied volatility grid given the inputs F, α, ρ, σ into the network. The average time taken per epoch of training for these networks is approximately 4 seconds.

7.2.2.1 Stochastic Lognormal Model: $\beta = 1$

Figure 7.10 shows the out of sample mean absolute percentage error (MAPE) of the ANN predictions across the volatility grid over all instances of the test set. That is, each element (l, m) of the matrix shown in figure 7.10 represents the MAPE between the SABR implied volatility at expiry l and strike price m, and the ANN predicted implied volatility, across the entire test set. The MAPE at any point on the 8×11 grid is in the range $(0.042\%, 0.086\%)$. The errors seem to be largest (relatively) for the shorter and longer expiries, but overall the ANN is able to accurately predict the implied volatility grid. The out of sample MSE and MAE achieved is 2.89495×10^{-6} and 0.00122 respectively and the R^2 value is 0.999997, the error scores are summarised in table 7.7.

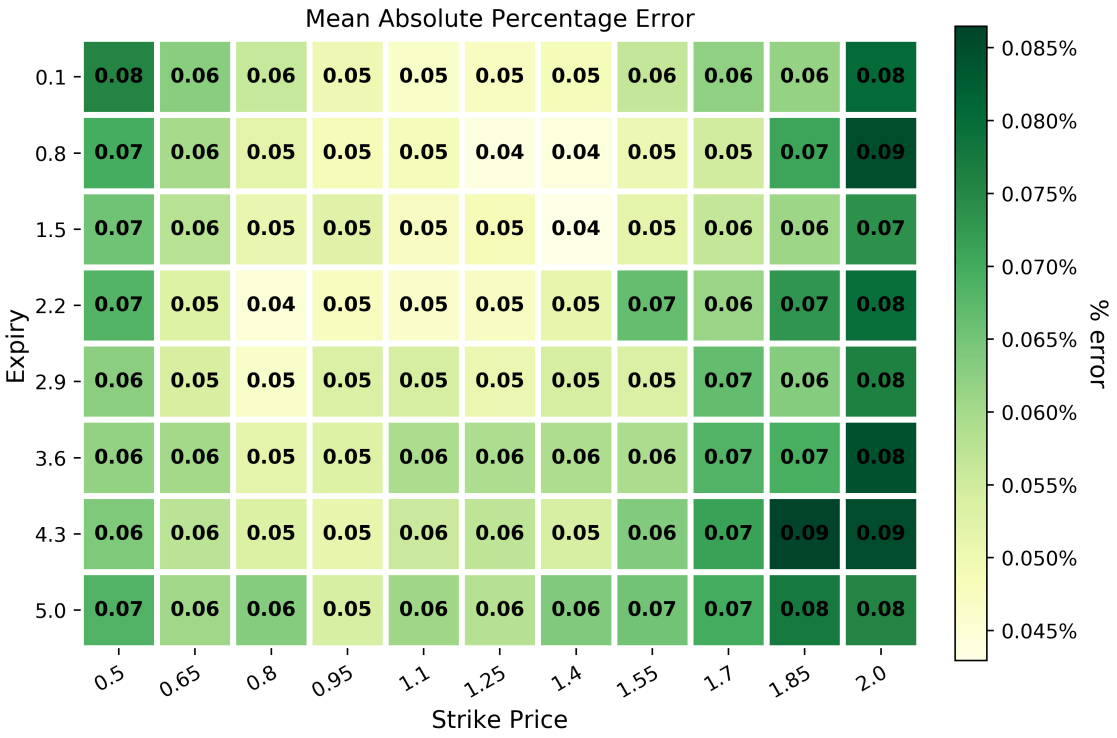


FIGURE 7.10. *Out of sample mean absolute percentage error across the implied volatility grid for lognormal SABR with* $\beta = 1$.

TABLE 7.7. *Error scores for lognormal SABR image-based neural network with* $\beta = 1$.

	MSE	MAE	R^2
Out-sample	2.89495×10^{-6}	0.00122	0.999997
In-sample	2.92880×10^{-6}	0.00122	0.999997

TABLE 7.8. *5-Fold cross validation scores for lognormal SABR* ($\beta = 1$) *image-based neural network using 200 epochs.*

	MSE	MAE
Fold 1	8.33274×10^{-6}	0.0020
Fold 2	8.31752×10^{-6}	0.0020
Fold 3	7.78586×10^{-6}	0.0019
Fold 4	8.45512×10^{-6}	0.0021
Fold 5	8.64836×10^{-6}	0.0021
Mean	8.30791×10^{-6}	0.0021

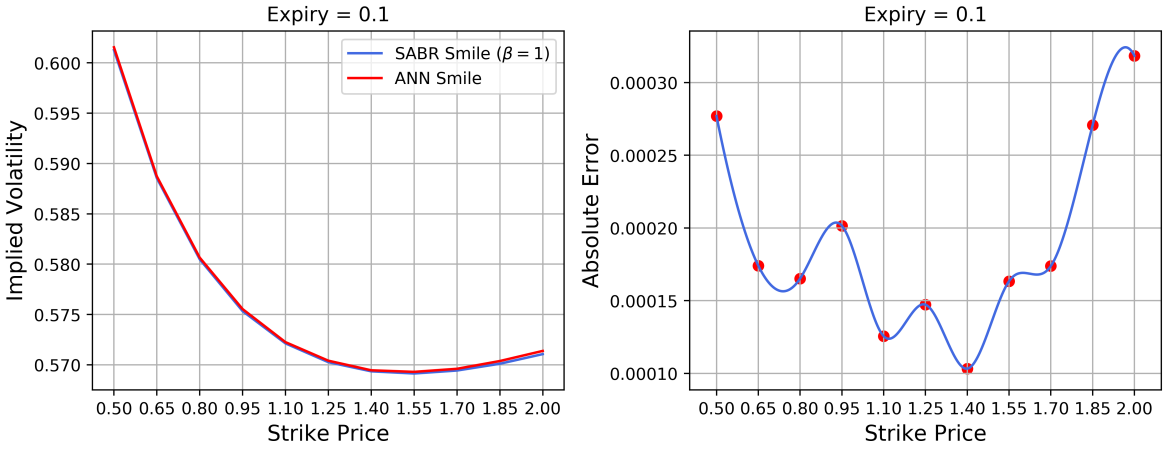


FIGURE 7.11. *Example smile and absolute error between SABR implied volatility and ANN implied volatility taken from the grid in figure 7.12.*

Figure 7.12 shows an example of an implied volatility grid from the SABR approximation and the corresponding ANN predicted grid for a random set of input parameters F, α, ρ, σ taken from the test set. The grid is composed of 8 smiles across 11 different strike prices as detailed in section 5.1.2.2. Visually, it is clear that the network is able to accurate predict the entire volatility surface since it is almost impossible to differentiate between the blue lines (SABR) and the red lines (ANN). To examine further, we plot the SABR and ANN smile corresponding to the maturity 0.1 and the absolute error in figure 7.11, the red dots plot the absolute error between the SABR implied volatility and the ANN implied volatility at each strike price. In the error plot cubic splines are interpolated through the 11 absolute error points to make the results easier to interpret. For this particular smile from the MSE and MAE is 4.13319×10^{-8} and 0.00019 respectively. Additionally, we run a 5-fold cross validation yielding an average MSE and MAE of 8.30791×10^{-6} and 0.0021 respectively, the CV scores are shown in table 7.8. To save on computation time the CV was run with 200 epochs in each iteration instead of 500 used to train our network which explains the slightly higher errors than those seen in table 7.7. However, the results support the conclusion that the network is accurate and consistent in predicting the implied volatility surface.

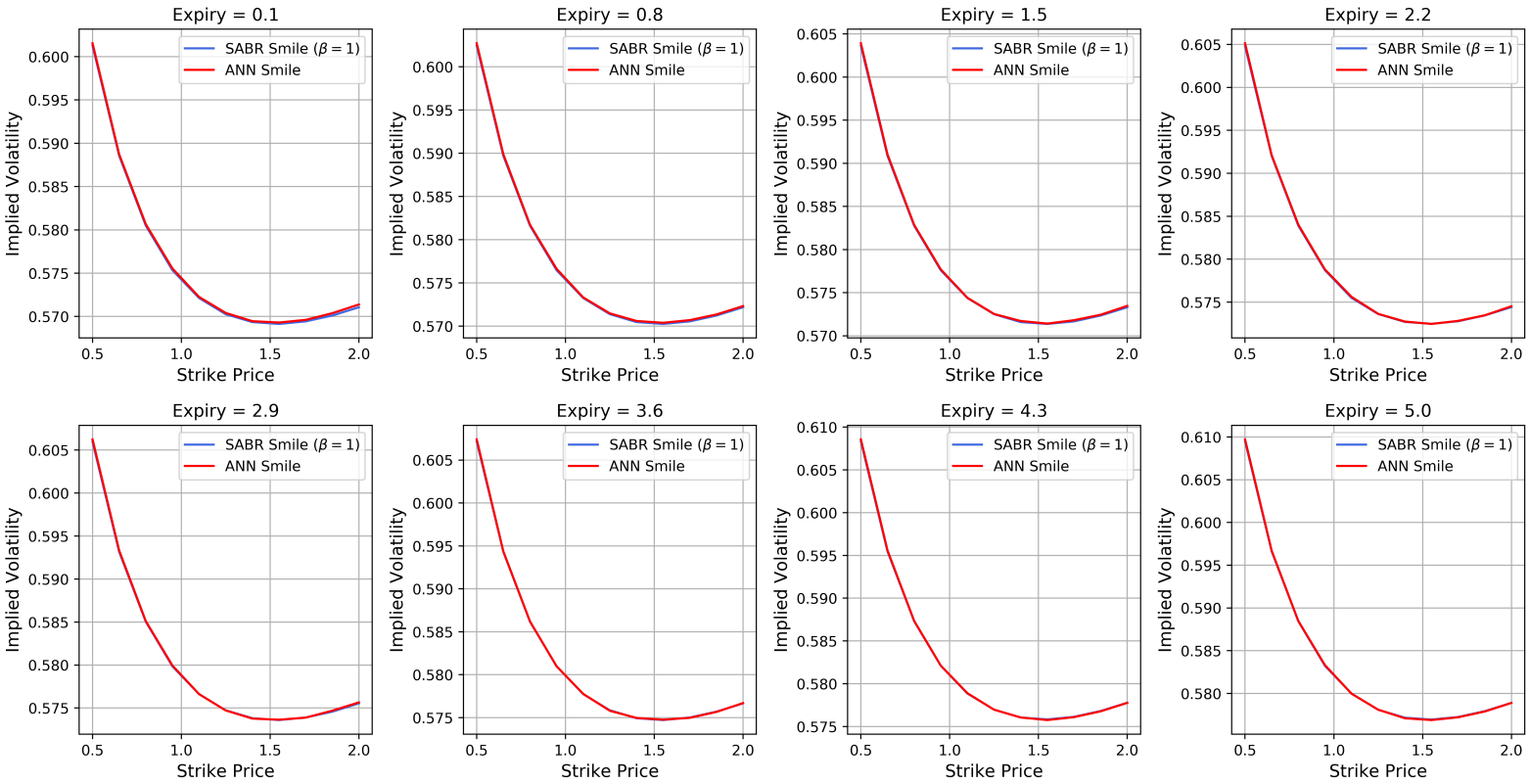


FIGURE 7.12. *Implied volatility grid predicted by the neural network and the actual implied volatility grid given by the lognormal SABR approximation with* $\beta = 1$.

FIGURE 7.13. *Out of sample mean absolute percentage error across the implied volatility grid for lognormal SABR with* $\beta = 0.5$.

7.2.2.2 Stochastic CIR Model: $\beta = 0.5$

Figure 7.13 shows the out of sample MAPE between the test set and the ANN predicted implied volatilites across the entire volatility grid over all instances of the test set. At any point on the implied volatility grid the MAPE is in the range $(0.043\%, 0.082\%)$, as with the stochastic lognormal case the errors seem to be relatively largest at short and long maturities but the overall accuracy is slightly better than the stochastic lognormal case. The out of sample MSE and MAE achieved is 2.46866×10^{-6} and 0.00109 respectively and the R^2 value is 0.999998, the error scores are summarised in table 7.9. The 5-fold CV scores are shown in table 7.10, the average MSE and MAE on the test set over the 5-folds are 6.97522×10^{-6} and 0.0018 respectively.

Figure 7.15 shows an example of an implied volatility surface from the SABR approximation and the corresponding ANN predicted surface for a set of random input parameters, F, α, ρ, σ, taken from the test set. Visually, it is clear that the network can accurately predict the volatility

TABLE 7.9. *Error scores for lognormal SABR image-based neural network with* $\beta = 0.5$.

	MSE	MAE	R^2
Out-sample	2.46866×10^{-6}	0.00109	0.999998
In-sample	2.46075×10^{-6}	0.00109	0.999998

TABLE 7.10. *5-Fold cross validation scores for lognormal SABR (*$\beta = 0.5$*) image-based neural network using 200 epochs.*

	MSE	MAE
Fold 1	7.12863×10^{-6}	0.0018
Fold 2	7.07138×10^{-6}	0.0019
Fold 3	5.91801×10^{-6}	0.0017
Fold 4	6.73641×10^{-6}	0.0018
Fold 5	8.02165×10^{-6}	0.0020
Mean	6.97522×10^{-6}	0.0018

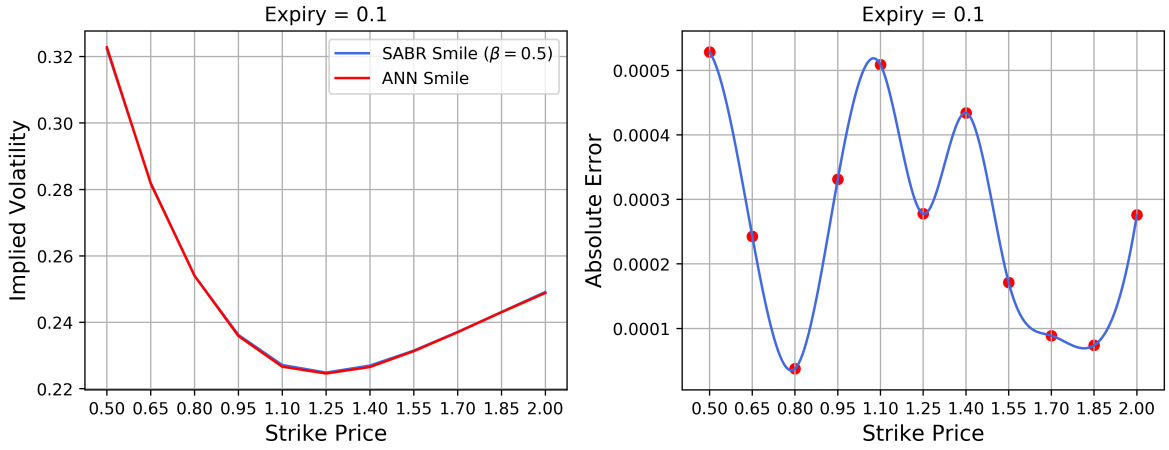


FIGURE 7.14. *Example smile and absolute error between SABR implied volatility and ANN implied volatility taken from the grid in figure 7.15.*

surface. Figure 7.14 examines the first smile from the grid and tracks the absolute error at each strike price. For this particular smile the MSE and MAE was 9.92646×10^{-8} and 0.00027 respectively. We conclude that the ANN is able to consistently and accurately predict the implied volatility surface.

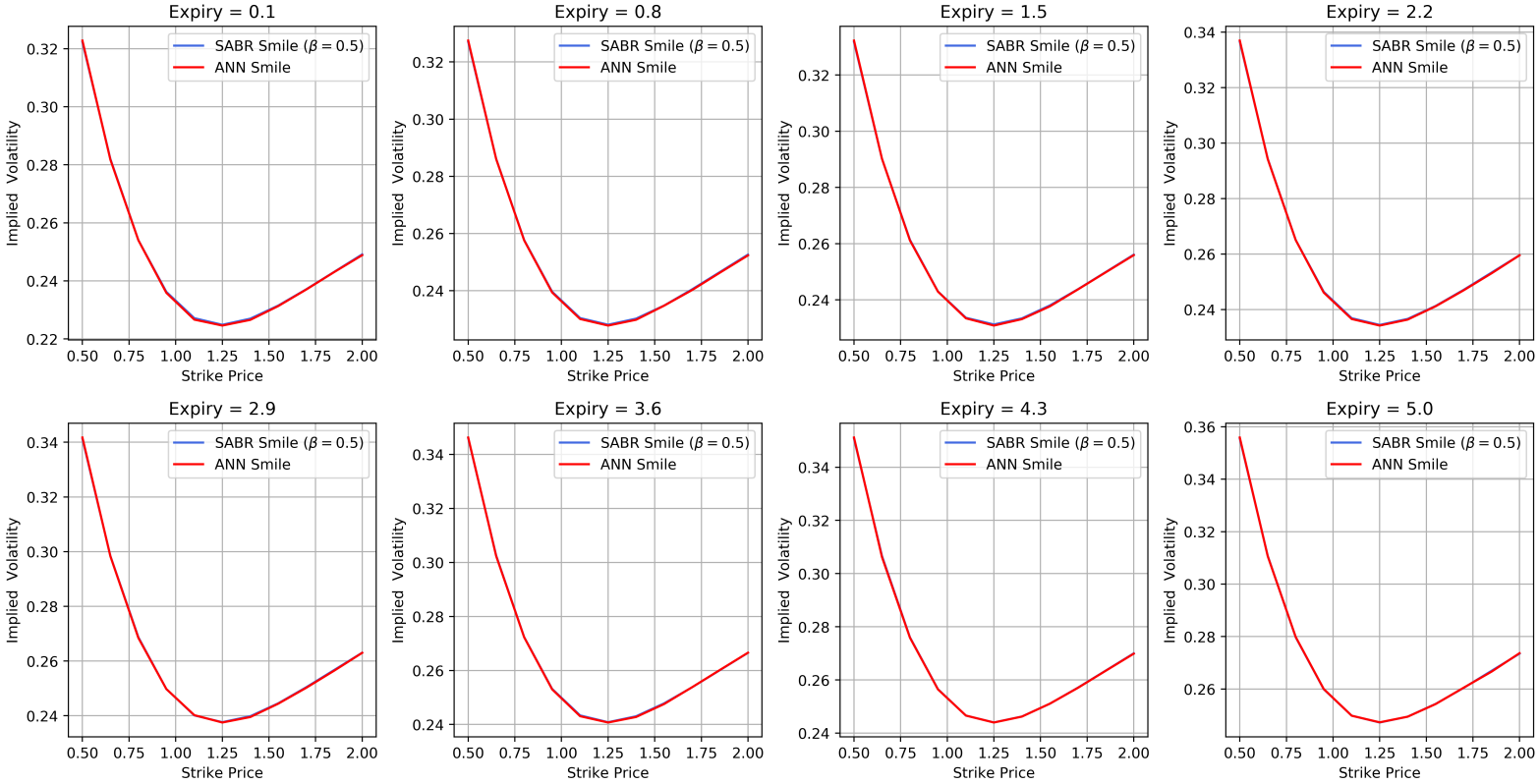


FIGURE 7.15. *Implied volatility grid predicted by the neural network and the actual implied volatility grid given by the lognormal SABR approximation with* $\beta = 0.5$.

FIGURE 7.16. *Out of sample mean absolute percentage error across the implied volatility grid for lognormal SABR with* $\beta = 0$.

7.2.2.3 Stochastic Normal Model: $\beta = 0$

Figure 7.16 shows the out of sample MAPE across the volatility grid and table 7.11 summarises the other error statistics. The out of sample MSE and MAE are 2.31908×10^{-6} and 0.00106 respectively and the R^2 is 0.999998. At any point on the implied volatility grid the MAPE is in the range $(0.047\%, 0.094\%)$ and the errors are relatively larger for very short and long maturities.

Figure 7.18 shows an example of an ANN predicted grid and the corresponding SABR grid. Figure 7.17 examines the first smile from this grid in more detail, the MSE and MAE for this

TABLE 7.11. *Error scores for lognormal SABR image-based neural network with* $\beta = 0.5$.

	MSE	MAE	R^2
Out-sample	2.31908×10^{-6}	0.00106	0.999998
In-sample	2.31256×10^{-6}	0.00106	0.999998

TABLE 7.12. *5-Fold cross validation scores for lognormal SABR ($\beta = 0$) image-based neural network using 200 epochs.*

	MSE	MAE
Fold 1	6.66435×10^{-6}	0.0018
Fold 2	6.44737×10^{-6}	0.0018
Fold 3	5.60939×10^{-6}	0.0016
Fold 4	6.19001×10^{-6}	0.0017
Fold 5	7.11604×10^{-6}	0.0018
Mean	6.40543×10^{-6}	0.0018

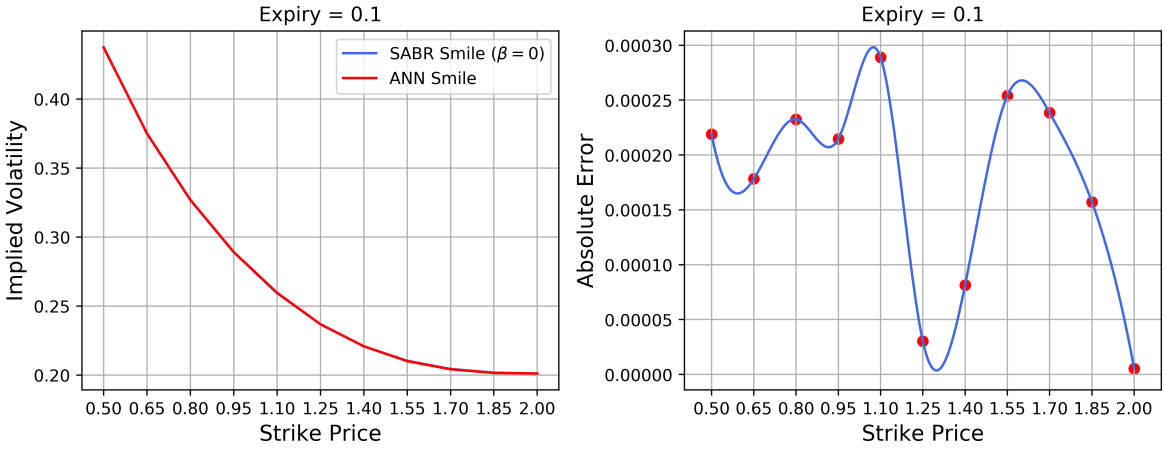


FIGURE 7.17. *Example smile and absolute error between SABR implied volatility and ANN implied volatility taken from the grid in figure 7.18.*

smile are 3.79198×10^{-8} and 0.00017 respectively. These results, along with the 5-fold CV scores shown in table 7.12 confirm that the ANN performs accurately and is consistent.

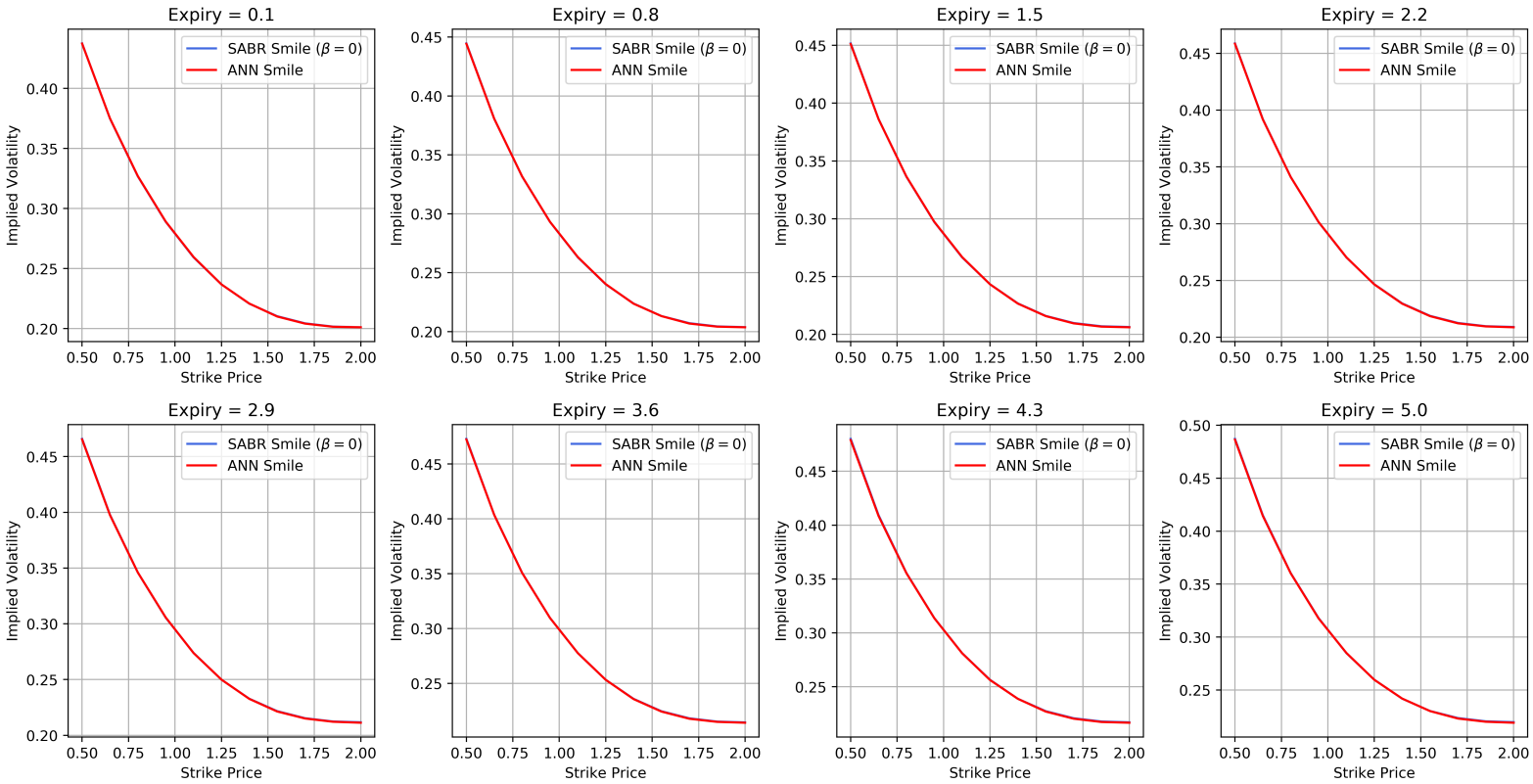


FIGURE 7.18. *Implied volatility grid predicted by the neural network and the actual implied volatility grid given by the lognormal SABR approximation with* $\beta = 0$.

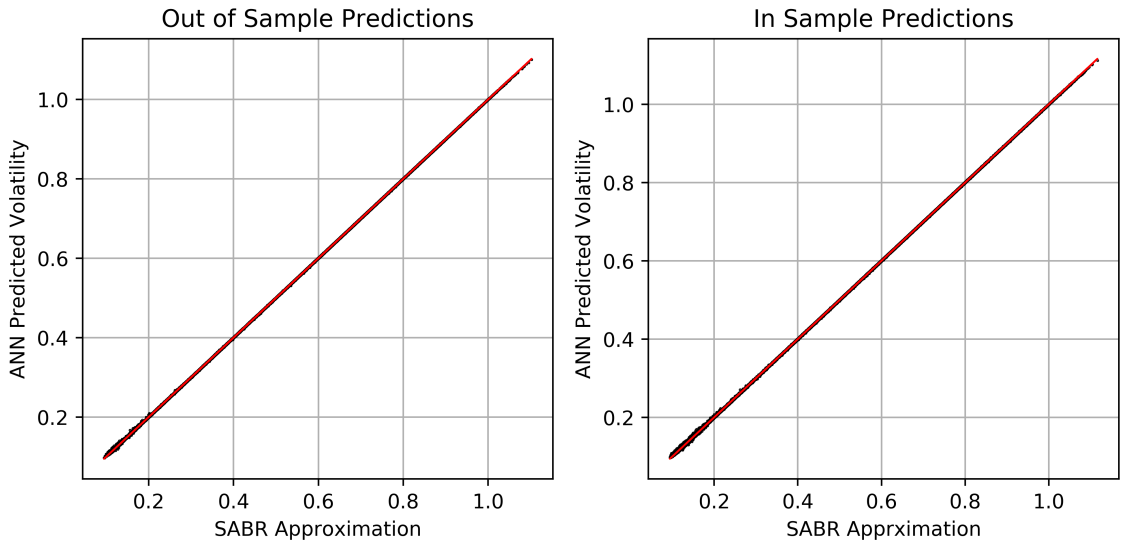


FIGURE 7.19. *Normal SABR Pointwise Neural Network - Out of sample and in sample implied volatilities when* $\beta = 0$.

TABLE 7.13. *Error scores for normal SABR pointwise Neural Network*

	MSE	MAE	R^2
Out-sample	6.27015×10^{-8}	0.00017	0.999998
In-sample	6.14239×10^{-8}	0.00017	0.999998

7.3 Normal SABR Model

7.3.1 Normal SABR Pointwise Neural Network Results

For the pointwise regime for the normal SABR model we only consider the case when the underlying follows a stochastic normal distribution, i.e when $\beta = 0$. Figure 7.19 shows the out sample and in sample normal implied volatilities predicted by our ANN and the SABR approximation (see definition 3.8). It is almost impossible to distinguish the implied volatilites from the red line at which the prediction is the same as the SABR approximation. The out of sample R^2 value between predicted implied volatilites and SABR approximations is 0.999998, the MSE is 6.27015×10^{-8} or 0.0006 bps and the MAE is 0.00017, table 7.13 summarises these error statistics. The distribution of the errors, the difference between the SABR approximations and the ANN predictions, is shown in figure 7.20.

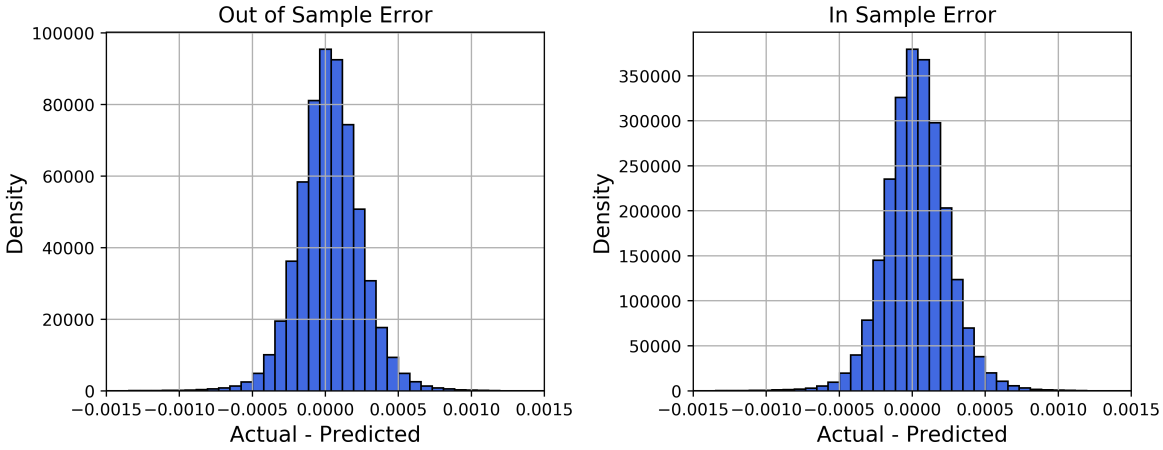


FIGURE 7.20. *Normal SABR Pointwise Neural Network - Out of sample and in sample implied volatilities when* $\beta = 0$.

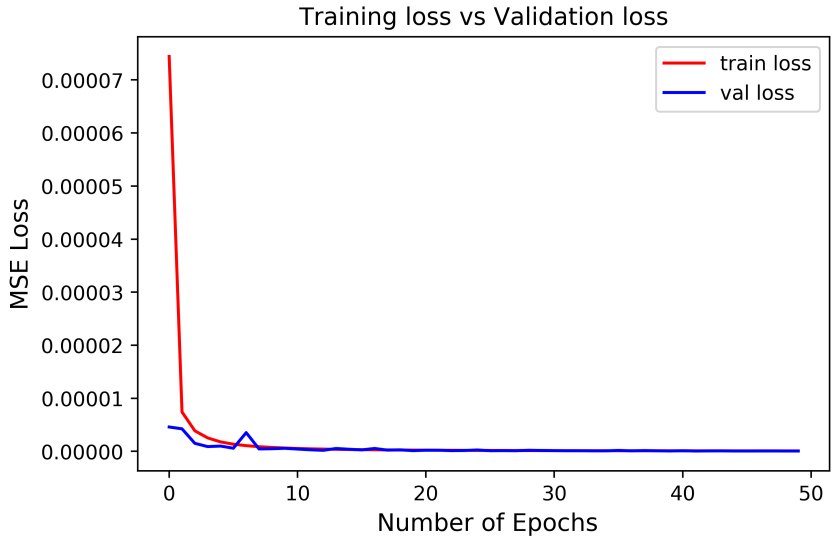


FIGURE 7.21. *Training loss vs Validation loss for normal SABR pointwise neural network.*

The average time taken per epoch of training was 48 seconds and the training and validation loss was tracked over all 50 epochs and is shown in figure 7.21. As has been the case for all previous networks there is no sign of overfitting to the training data as the two loss lines do not diverge. The 5-fold cross validation error scores obtained are shown in table 7.14, the average MSE and MAE over the 5 folds was 1.00132×10^{-7} and 0.00022 respectively, showing that the ANN consistently and accurately predicts the normal implied volatility. We attribute the marginally higher error scores compared to those in table 7.13 to the fact that the

TABLE 7.14. *5-Fold cross validation scores for normal SABR pointwise neural network using 30 epochs.*

	MSE	MAE
Fold 1	9.57564×10^{-8}	0.00021
Fold 2	8.91472×10^{-8}	0.00021
Fold 3	1.16753×10^{-7}	0.00024
Fold 4	8.89997×10^{-8}	0.00021
Fold 5	1.10003×10^{-7}	0.00024
Mean	1.00132×10^{-7}	0.00022

TABLE 7.15. *Error scores for normal SABR image-based neural network with* $\beta = 1$.

	MSE	MAE	R^2
Out-sample	2.80786×10^{-6}	0.0012	0.999997
In-sample	2.79794×10^{-6}	0.0012	0.999997

CV was done over 30 epochs instead of 50. As an observation, these are the most accurate results out of all our experiments. The results of the pointwise experiment show that our ANN can learn the SABR normal implied volatility expansion which provides the proof of concept required before proceeding with the image-based method.

7.3.2 Normal SABR Image-Based Neural Network Results

Again, in this case we are concerned with predicting the entire 8×11 implied volatility grid given the inputs F, α, ρ, σ into the network. The average time taken per epoch of training for these networks is approximately 4 seconds.

7.3.2.1 Stochastic Lognormal Model: $\beta = 1$

Figure 7.22 shows the out of sample MAPE across the volatility grid and table 7.15 shows the remaining error scores. At any point on the implied volatility grid the MAPE is in the range $(0.047\%, 0.092\%)$, the out of sample MSE and MAE achieved is 2.80786×10^{-6} and 0.0012 respectively and the R^2 value is 0.999997.

FIGURE 7.22. *Out of sample mean absolute percentage error across the implied volatility grid for normal SABR with* $\beta = 1$.

TABLE 7.16. *5-Fold cross validation scores for normal SABR* ($\beta = 1$) *image-based neural network using 200 epochs.*

	MSE	MAE
Fold 1	7.57436×10^{-6}	0.0019
Fold 2	7.54576×10^{-6}	0.0020
Fold 3	7.28132×10^{-6}	0.0019
Fold 4	7.31869×10^{-6}	0.0019
Fold 5	7.64233×10^{-6}	0.0020
Mean	7.47249×10^{-6}	0.0019

Figure 7.24 shows an example ANN predicted surface and the matching SABR surface and figure 7.23 examines the first smile from this grid in more detail. For the smile in figure 7.23 the MSE and MAE are 3.90027×10^{-7} and 0.00052 respectively. The MSE and MAE between the predicted surface and the actual surface shown in figure 7.24 are 5.71244×10^{-7} and 0.00064 respectively. Finally, the 5-fold CV scores are shown in table 7.16 with an average MSE and MAE of 7.47249×10^{-6} and 0.0019 respectively.

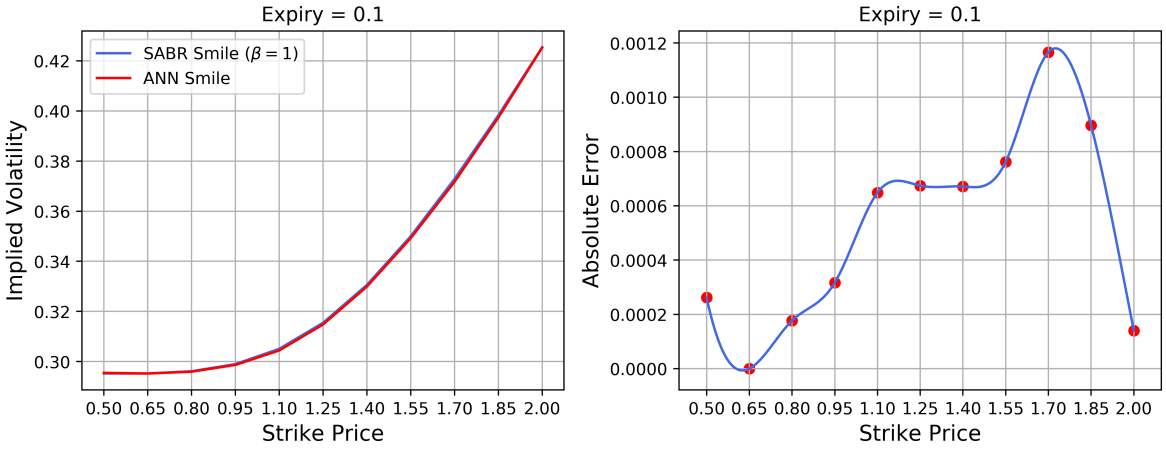


FIGURE 7.23. *Example smile and absolute error between SABR implied volatility and ANN implied volatility taken from the grid in figure 7.24.*

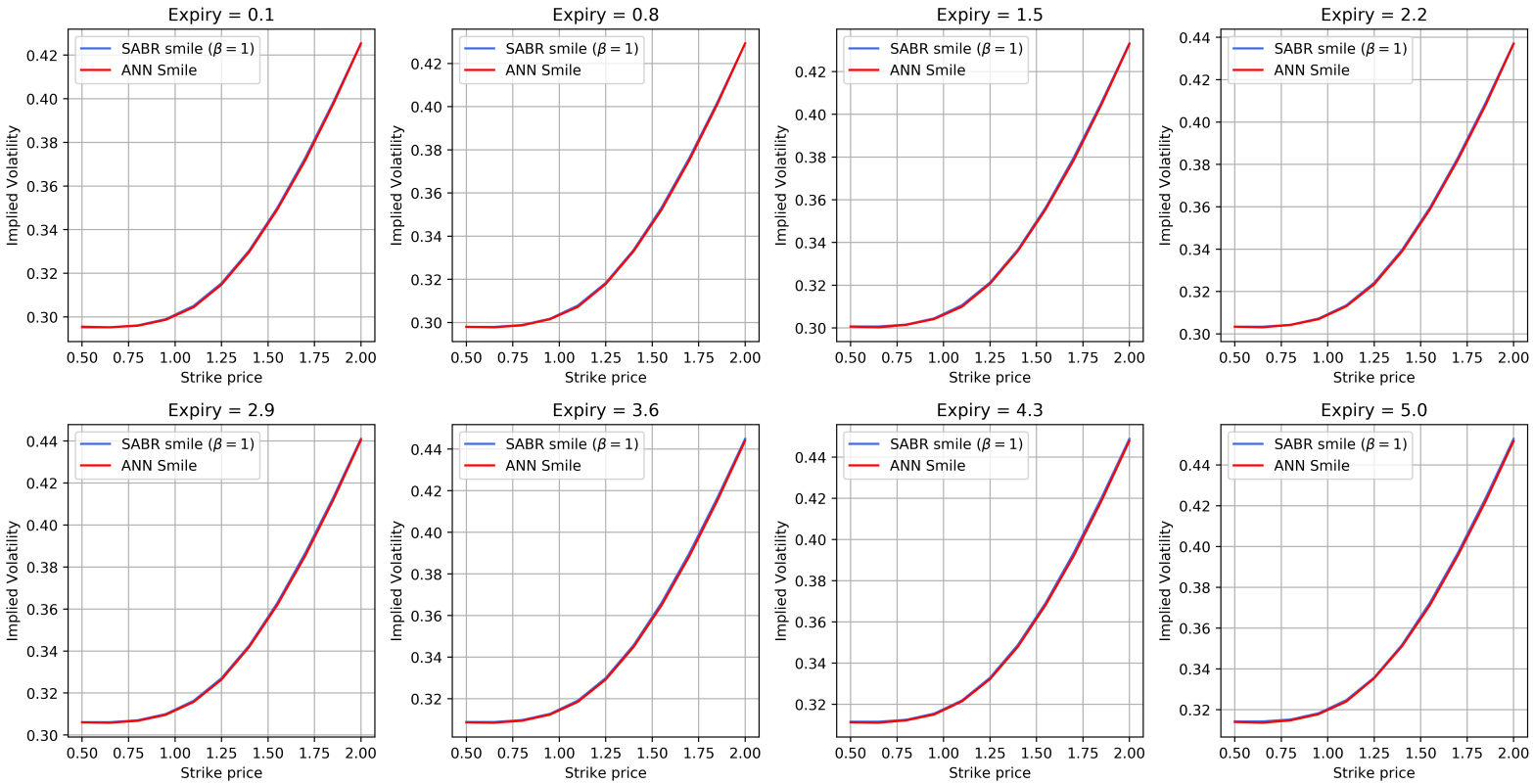


FIGURE 7.24. *Implied volatility grid predicted by the neural network and the actual implied volatility grid given by the normal SABR approximation with* $\beta = 1$.

FIGURE 7.25. *Out of sample mean absolute percentage error across the implied volatility grid for normal SABR with* $\beta = 0.5$.

7.3.2.2 Stochastic CIR Model: $\beta = 0.5$

Figure 7.25 shows the out of sample MAPE across the implied volatility grid and table 7.17 shows the MSE, MAE and R^2 values. At any point on the implied volatility grid the MAPE is in the range $(0.042\%, 0.083\%)$, the MSE and MAE across the test set are 2.85730×10^{-6} and 0.0012 respectively whilst the R^2 is 0.999997. Table 7.18 shows the 5-fold CV scores with an average MSE and MAE on the test set of 7.76258×10^{-6} and 0.0020 respectively, this, along with the out of sample accuracy strongly support the accuracy of the network. Figure 7.27 shows an example ANN predicted grid and the corresponding SABR grid and

TABLE 7.17. *Error scores for normal SABR image-based neural network with* $\beta = 0.5$.

	MSE	MAE	R^2
Out-sample	2.85730×10^{-6}	0.0012	0.999997
In-sample	2.82215×10^{-6}	0.0012	0.999997

TABLE 7.18. *5-Fold cross validation scores for normal SABR ($\beta = 0.5$) image-based neural network using 200 epochs.*

	MSE	MAE
Fold 1	6.85820×10^{-6}	0.0018
Fold 2	6.67380×10^{-6}	0.0018
Fold 3	9.08180×10^{-6}	0.0021
Fold 4	8.30442×10^{-6}	0.0021
Fold 5	7.89468×10^{-6}	0.0019
Mean	7.76258×10^{-6}	0.0020

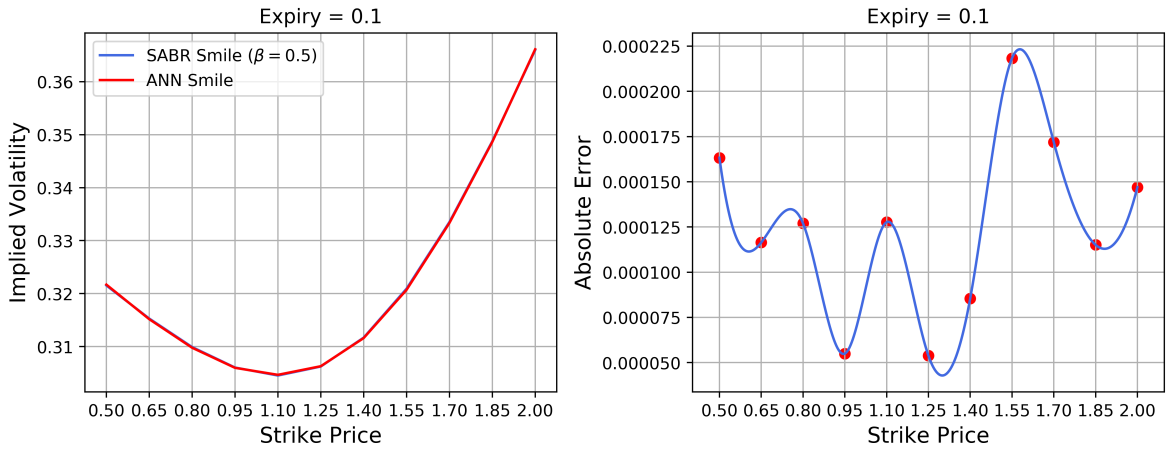


FIGURE 7.26. *Example smile and absolute error between SABR implied volatility and ANN implied volatility taken from the grid in figure 7.27.*

figure 7.26 looks in more detail at the first smile from this grid. For this smile the MSE and MAE are 1.79939×10^{-8} and 0.00013 respectively, whilst the MSE and MAE for the entire example grid is 2.69688×10^{-8} and 0.00014 respectively.

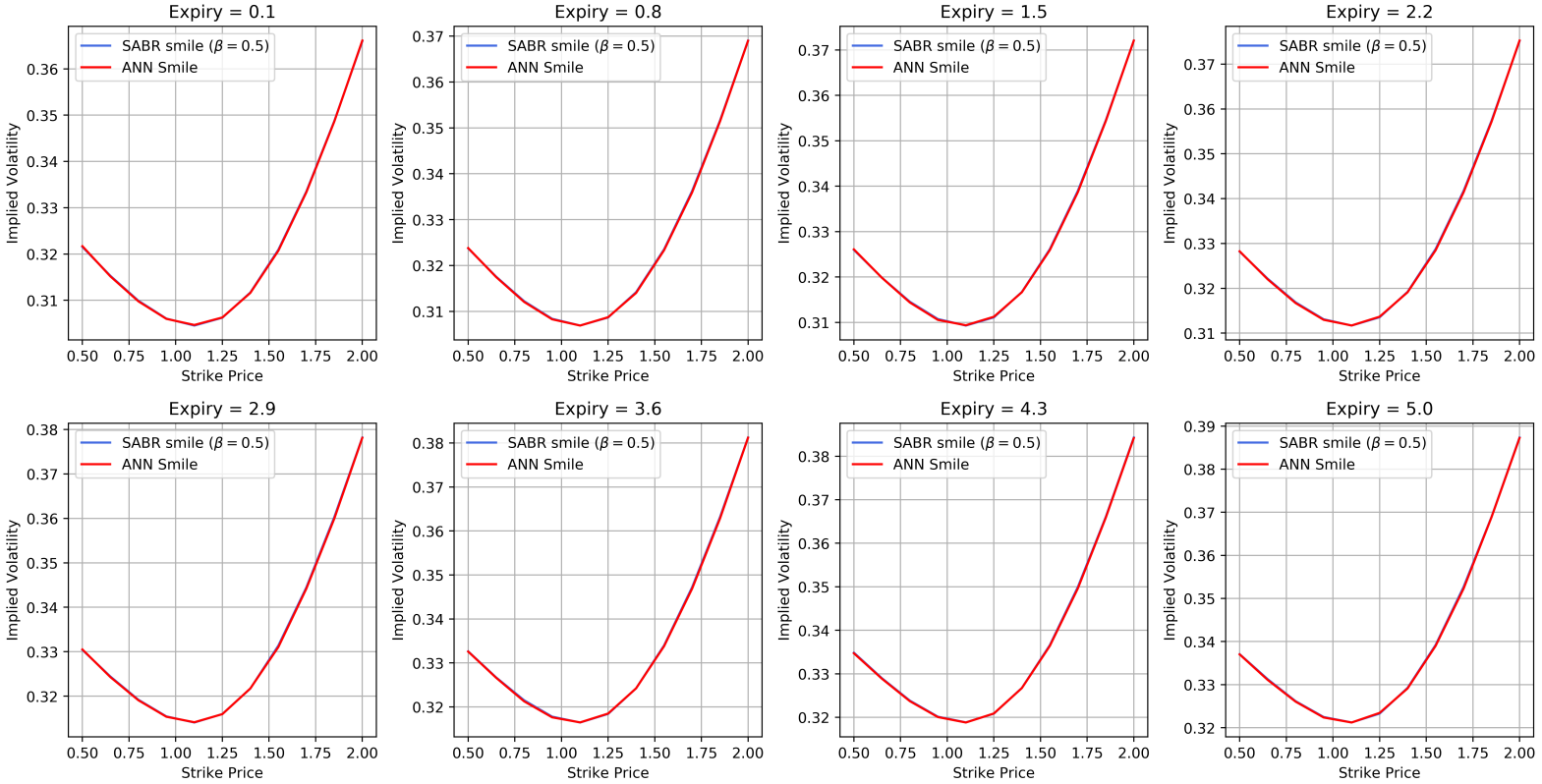


FIGURE 7.27. *Implied volatility grid predicted by the neural network and the actual implied volatility grid given by the normal SABR approximation with* $\beta = 0.5$.

FIGURE 7.28. *Out of sample mean absolute percentage error across the implied volatility grid for normal SABR with* $\beta = 0$.

7.3.2.3 Stochastic Normal Model: $\beta = 0$

Figure 7.28 shows the out of sample MAPE across the implied volatility surface, at any point on the grid the MAPE is in the range $(0.038\%, 0.078\%)$. The out of sample MSE and MAE are 2.79836×10^{-6} and 0.0012 respectively, table 7.19 summarises these results. Figure 7.30 shows an example ANN predicted surface and the corresponding SABR model surface, the MSE and MAE for the predicted surface are 5.02490×10^{-8} and 0.00019 respectively. Figure 7.29 examines the first smile from the grid in more detail, the MSE and MAE of the smile are 6.02275×10^{-8} and 0.00021 respectively. Finally, the 5-fold CV scores are shown

TABLE 7.19. *Error scores for normal SABR image-based neural network with* $\beta = 0$.

	MSE	MAE	R^2
Out-sample	2.79836×10^{-6}	0.0012	0.999997
In-sample	2.76285×10^{-6}	0.0012	0.999997

TABLE 7.20. *5-Fold cross validation scores for normal SABR ($\beta = 0$) image-based neural network using 200 epochs.*

	MSE	MAE
Fold 1	6.69640×10^{-6}	0.0018
Fold 2	6.91258×10^{-6}	0.0018
Fold 3	5.85344×10^{-6}	0.0017
Fold 4	6.62384×10^{-6}	0.0018
Fold 5	6.93271×10^{-6}	0.0019
Mean	6.60379×10^{-6}	0.0018

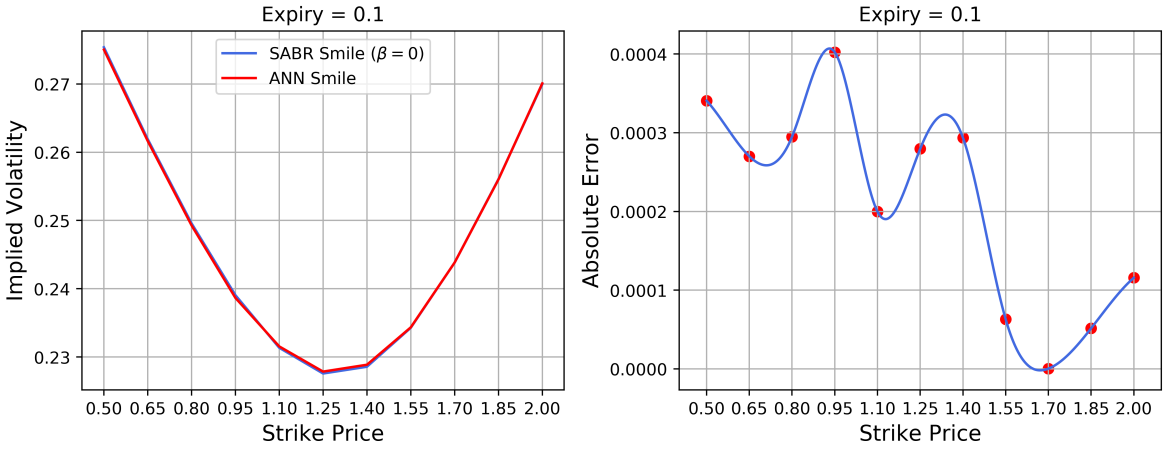


FIGURE 7.29. *Example smile and absolute error between SABR implied volatility and ANN implied volatility taken from the grid in figure 7.30.*

in table 7.20 with an average MSE and MAE on the test set of 6.60379×10^{-6} and 0.0018 respectively. We conclude that the network is able to accurately and consistently predict the implied volatility surface

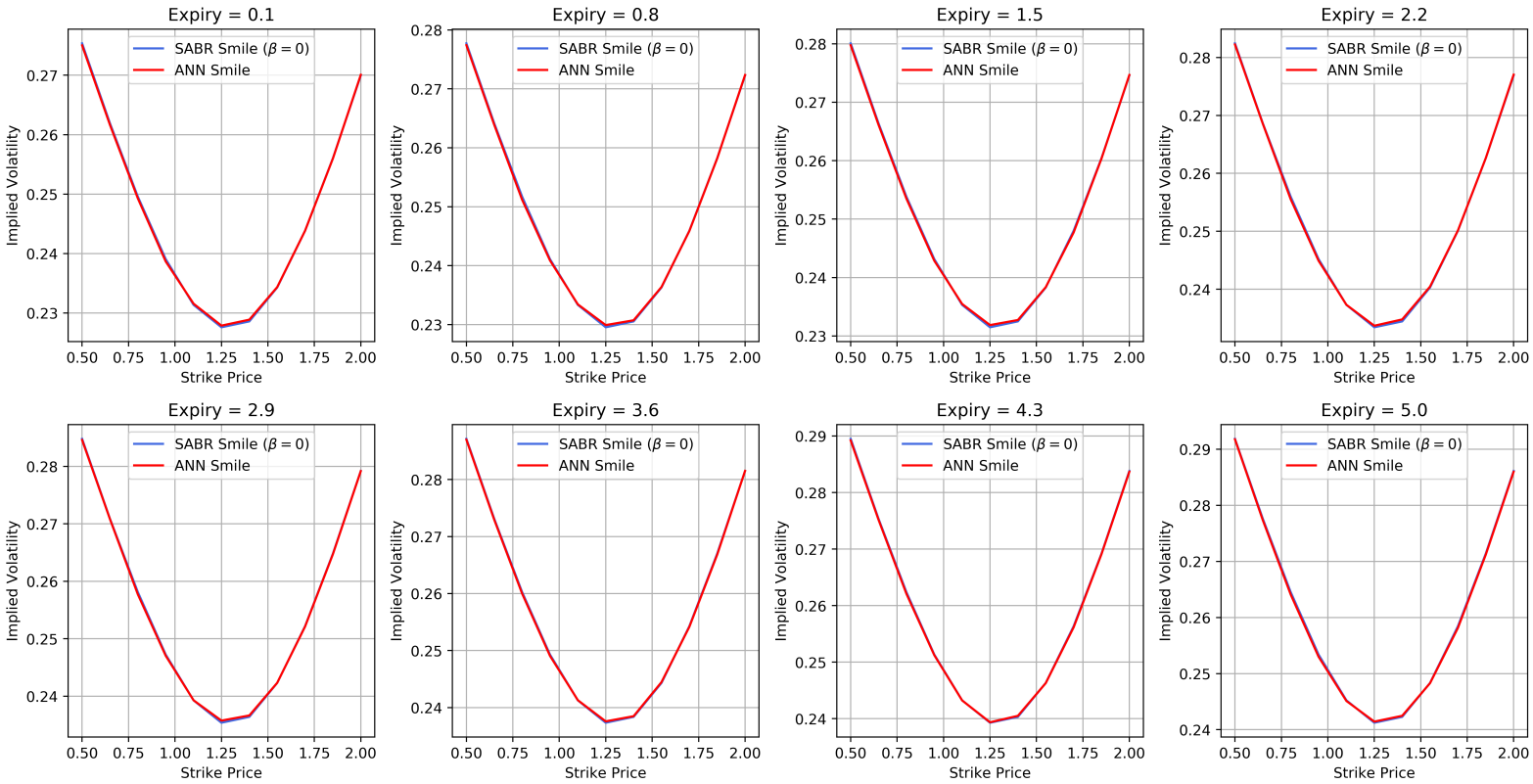


FIGURE 7.30. *Implied volatility grid predicted by the neural network and the actual implied volatility grid given by the normal SABR approximation with* $\beta = 0$.

TABLE 7.21. *Run time performance, image-based ANN vs SABR approximation for normal SABR with random input vector from the test set.*

Method	$\beta = 0$	$\beta = 0.5$	$\beta = 1$
ANN	0.321ms	0.349ms	0.295ms
SABR Approximation	5.372ms	5.381ms	4.527ms

TABLE 7.22. *Run time performance, image-based ANN vs SABR approximation for lognormal SABR with random input vector from the test set.*

Method	$\beta = 0$	$\beta = 0.5$	$\beta = 1$
ANN	0.327ms	0.333ms	0.397ms
SABR Approximation	6.091ms	4.937ms	6.522ms

7.4 Speed Test

Tables 7.21 and 7.22 show the time taken in milliseconds to generate the implied volatility surface for an input vector when using the ANN vs repeatedly using the SABR approximations to obtain each implied volatility value on the surface. The ANN's are on average between 16 and 17 times faster than the SABR expansions.

CHAPTER 8

Conclusion

In this thesis we have investigated the use of neural networks for 2 key problems in quantitative finance: option pricing and implied volatility calculation. The results show that ANNs can accurately capture the relationship between market state variables and option prices/implied volatility for the Black-Scholes model and the SABR stochastic volatility model. We have addressed some inconsistencies in option pricing neural networks currently in the literature and have shown that they are certainly a viable method due to the high accuracy of predictions whilst preserving first and second order derivatives for option Greeks. Moreover, our approach of learning the pricing functions of calls and puts together on the same network have shown that a single network can differentiate between the two. We have also considered the SABR model in some depth, starting with learning the implied volatility expansions to predicting implied volatility surfaces. In both cases we were able to predict implied volatility accurately, the ANNs for the normal model were generally slightly more accurate than those for the lognormal model. However, all of the neural networks have been shown to consistently and accurately predict the implied volatility surface, the resulting ANNs are also approximately 16 to 17 times faster than repeatedly evaluating the SABR expansions, these speed ups could be crucial when considering strategies that trade volatility for example.

8.1 Future outlook

It would be fruitful to consider 2 variations of the original SABR model, the SABR integration (McGhee 2011) and a SABR 2 factor finite difference scheme (McGhee 2018). There is large scope for creativity with our network architectures, experimenting with different activation

functions and more exotic optimisers such as AdaBound (Luo et al. 2019) would be useful. Of particular interest, would be the use of gradient-free optimisers such as differential evolution on the neural networks. Exploring robust hyper parameter optimisation and the use of a weighted ensemble of different neural networks to reduce the variation of the predictions would provide valuable insight (see Appendix C). Further work on ANNs for option pricing could include models such as Heston (Heston 1993) and Bates' (Bates 1996) along with consideration of more exotic options, particularly American options.

Bibliography

Aghdam, Hamed Habibi and Elnaz Jahani Heravi (2017). 'Guide to Convolutional Neural Networks'. In: *New York, NY: Springer. doi* 10, pp. 978–3.

Ardizzone, Lynton et al. (2018). 'Analyzing inverse problems with invertible neural networks'. In: *arXiv preprint arXiv:1808.04730*.

Bachelier, Louis (1900). 'Théorie de la spéculation'. In: *Annales scientifiques de l'École normale supérieure*. Vol. 17, pp. 21–86.

Bartle, Robert Gardner and Robert G Bartle (1995). *The elements of integration and Lebesgue measure*. Vol. 27. Wiley Online Library.

Bates, David S (1996). 'Jumps and stochastic volatility: Exchange rate processes implicit in deutsche mark options'. In: *The Review of Financial Studies* 9.1, pp. 69–107.

Black, Fischer (1976). 'The pricing of commodity contracts'. In: *Journal of financial economics* 3.1-2, pp. 167–179.

Black, Fischer and Myron Scholes (1973). 'The pricing of options and corporate liabilities'. In: *Journal of political economy* 81.3, pp. 637–654.

Carlile, Brad et al. (2017). 'Improving deep learning by inverse square root linear units (ISRLUs)'. In: *arXiv preprint arXiv:1710.09967*.

Clevert, Djork-Arné, Thomas Unterthiner and Sepp Hochreiter (2015). 'Fast and accurate deep network learning by exponential linear units (elus)'. In: *arXiv preprint arXiv:1511.07289*.

Cybenko, George (1989). 'Approximation by superpositions of a sigmoidal function'. In: *Mathematics of control, signals and systems* 2.4, pp. 303–314.

Duchi, John, Elad Hazan and Yoram Singer (2011). 'Adaptive subgradient methods for online learning and stochastic optimization'. In: *Journal of Machine Learning Research* 12.Jul, pp. 2121–2159.

Dupire, Bruno et al. (1994). 'Pricing with a smile'. In: *Risk* 7.1, pp. 18–20.

Eldan, Ronen and Ohad Shamir (2016). 'The power of depth for feedforward neural networks'. In: *Conference on learning theory*, pp. 907–940.

Ferguson, Ryan and Andrew David Green (2018). 'Deeply learning derivatives'. In: *Available at SSRN 3244821*.

Glorot, Xavier, Antoine Bordes and Yoshua Bengio (2011). 'Deep sparse rectifier neural networks'. In: *Proceedings of the fourteenth international conference on artificial intelligence and statistics*, pp. 315–323.

Hagan, Patrick S et al. (2002). 'Managing smile risk'. In: *The Best of Wilmott* 1, pp. 249–296.

Hernandez, Andres (2016). 'Model calibration with neural networks'. In: *Available at SSRN 2812140*.

Heston, Steven L (1993). 'A closed-form solution for options with stochastic volatility with applications to bond and currency options'. In: *The review of financial studies* 6.2, pp. 327–343.

Hida, Takeyuki (1980). 'Brownian motion'. In: *Brownian Motion*. Springer, pp. 44–113.

Hornik, Kurt, Maxwell Stinchcombe and Halbert White (1990). 'Universal approximation of an unknown mapping and its derivatives using multilayer feedforward networks'. In: *Neural networks* 3.5, pp. 551–560.

Horvath, Blanka, Aitor Muguruza and Mehdi Tomas (2019). 'Deep learning volatility'. In: *Available at SSRN 3322085*.

Itkin, Andrey (2019). 'Deep learning calibration of option pricing models: some pitfalls and solutions'. In: *arXiv preprint arXiv:1906.03507*.

Kingma, Diederik P and Jimmy Ba (2014). 'Adam: A method for stochastic optimization'. In: *arXiv preprint arXiv:1412.6980*.

Lagerstrom, PA and RG Casten (1972). 'Basic concepts underlying singular perturbation techniques'. In: *Siam Review* 14.1, pp. 63–120.

Liu, Shuaiqiang et al. (2019). 'A neural network-based framework for financial model calibration'. In: *arXiv preprint arXiv:1904.10523*.

Lu, Zhou et al. (2017). 'The expressive power of neural networks: A view from the width'. In: *Advances in neural information processing systems*, pp. 6231–6239.

Luo, Liangchen et al. (2019). ‘Adaptive gradient methods with dynamic bound of learning rate’. In: *arXiv preprint arXiv:1902.09843*.

Malkiel, Burton G (1989). ‘Efficient market hypothesis’. In: *Finance*. Springer, pp. 127–134.

McGhee, William (2011). ‘An efficient implementation of stochastic volatility by the method of conditional integration”’. In: *ICBI Conference, Paris*.

McGhee, William A (2018). ‘An artificial neural network representation of the SABR stochastic volatility model’. In: *Available at SSRN 3288882*.

Nielsen, Michael A (2015). *Neural networks and deep learning*. Vol. 25. Determination press San Francisco, CA, USA:

Tieleman, Tijmen and Geoffrey Hinton (2012). ‘Lecture 6.5-rmsprop: Divide the gradient by a running average of its recent magnitude’. In: *COURSERA: Neural networks for machine learning* 4.2, pp. 26–31.

1 Appendix A: Using Keras.

We use Keras with TensorFlow backend, to install using pip run *pip install –upgrade tensorflow* and *pip install keras*. The Keras deep learning library offers two interfaces, the sequential API and the funtional API. The sequential API does not permit multiple outputs whereas the functional API does, for this reason, along with the fact that it is easier to understand the layer structure, we work exclusively with the functional API, even when we only have one output. Consider a generic network with input dimension x, 3 hidden layers using y neurons and output dimension z. The following syntax is used to generate the network.

```
from keras.layers import Dense, Input
from keras.models import Model

input_layer = Input(shape=(x,)) # Input layer
a = Dense(y, activation='elu')(input_layer) # Hidden layer 1
b = Dense(y, activation='elu')(a) # Hidden layer 2
c = Dense(y, activation='elu')(b) # Hidden layer 3
output_layer = Dense(z, activation='linear')(c) # Output layer
NN = Model(inputs=input_layer, outputs=output_layer) # Define NN
NN.compile(loss='mse', optimizer='adam') # Compile NN
```

The network can then be trained by calling the *.fit* method and then used to make predictions using the *.predict* method.

By default, Keras will not return the best model (lowest validation MSE) over all training epochs, instead, it returns a model with weights and biases determined by the final epoch of training. As far as we are aware, Keras does not intrinsically offer this feature. To get around this, we make use of Keras callbacks, a feature that allows us to view the networks internal state during training. We use a *ModelCheckpoint* callback to monitor the validation MSE at each epoch, when the MSE is lower than that of a previous epoch the network weights and biases are saved to a *.hdf5* file. After all epochs, the file contains the weights and biases of the best model, which we load into the network using the *load_weights* method. The general syntax is as follows.

```
from keras.callbacks import ModelCheckpoint
from keras.models import load_model
X # Some input data
y # Some output data
cp = ModelCheckpoint('weights.hdf5', monitor='val_loss',
          save_best_only=True, mode='min') # Callback to monitor MSE
NN.fit(X, y, batch_size=128, validation_split=0.2,
        epochs=500, callbacks=[cp], shuffle=True) # Fit ANN
NN.load_weights('weights.hdf5') # Load into network the best weights
```

2 Appendix B: Code Structure

There are 3 core classes, *Data.py*, *ANN.py* and *ModelEvaluation.py*. The *ANN* class has 4 methods, *generate_ANN* which creates, compiles and returns an ANN, *fit_ANN* which fits the ANN to the data using the Keras Callback to keep track of the best model over all epochs, *load_best _model* which loads into the ANN the weights of the best model and *cross_validation* which implements K-Fold cross validation. The *Data* class implements the pre-processing routine for the SABR image based data. The *ModelEvaluation* class implements all plotting and error analysis. The notebooks *Data_generation_script.ipynb*, *Black-Scholes.ipynb*, *Normal_SABR.ipynb* and *Lognormal_SABR.ipynb* show the application of our methods.

3 Appendix C: An Experiment With Ensemble Methods

Let z be some data associated with a function $g = g(x) + u$ where $u \sim (0, \sigma^2)$ is noise, assume that we wish to approximate this function using a model $p = p(x)$. Consider the squared error cost function, $C = (z - p)^2$, the expectation can be expanded as

$$\mathbb{E}\left[C\right] = \sigma^2 + \underbrace{Var[p]}_{\text{Variance}} + \underbrace{(z - \mathbb{E}[p])^2}_{\text{Bias}}.$$

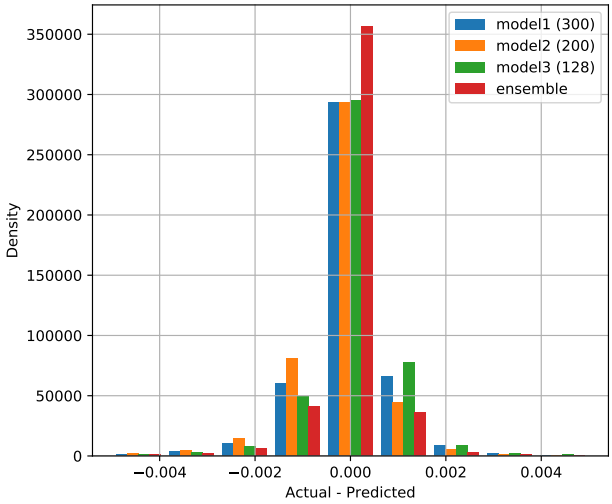


FIGURE .1. *Ensemble error distributions*

TABLE .1. *Ensemble errors*

Model	MSE	MAE	R^2
Model 1 (300 neurons)	1.24481×10^{-6}	0.00059	0.999981
Model 2 (200 neurons)	1.32424×10^{-6}	0.00060	0.999980
Model 3 (128 neurons)	1.56981×10^{-6}	0.00060	0.999980
Ensemble	9.42831×10^{-7}	0.00042	0.999990

Minimising the cost function requires minimising the bias and the variance, which are usually conflicting. This is known as the *bias-variance trade off*.

Ensemble methods combine different models and aggregate their outputs to reduce the variance of the ensemble. One option is to average the outputs of n models. Errors for combining the outputs of 3 ANNs using the same architecture presented for Black Scholes Neural Network 2 using 300, 200 and 128 neurons in each respectively and *30 epochs* are shown in figure .1 and table .1. Averaging out the predictions of the 3 networks performs better than any single network individually.