Model Calibration with Machine Learning

Nicolai Cornell Haussamer

A dissertation submitted to the Faculty of Commerce, University of Cape Town, in partial fulfilment of the requirements for the degree of Master of Philosophy.

September 22, 2018

MPhil in Mathematical Finance,
University of Cape Town.

Declaration

I declare that this dissertation is my own, unaided work. It is being submitted for the Degree of Master of Philosophy to the University of Cape Town. It has not been submitted before for any degree or examination to any other university.

Nicolai Cornell Haussamer

September 22, 2018

Abstract

This dissertation focuses on the application of neural networks to financial model calibration. It provides an introduction to the mathematics of basic neural networks and training algorithms. Two simplified experiments based on the Black-Scholes and constant elasticity of variance models are used to demonstrate the potential usefulness of neural networks in calibration. In addition, the main experiment features the calibration of the Heston model using model-generated data. In the experiment, we show that the calibrated model parameters reprice a set of options to a mean relative implied volatility error of less than one per cent. The limitations and shortcomings of neural networks in model calibration are also investigated and discussed.

Acknowledgements

I would like to thank Ralph Rudd for his help with this research. He has patiently guided me through the research process, and has often given invaluable advice. I have learned a lot from him, and I am grateful for the opportunity to have done so.

Many thanks to James Taylor, who originally proposed this most interesting and rewarding research topic. Thank you also to Feroz Bhamani for the conversations and updates throughout our parallel research processes.

Lastly, I owe thanks to the friends and family who supported and encouraged me throughout this academic undertaking.

Contents

List of Figures

Chapter 1

Introduction

1.1 Derivatives trading and the calibration problem

Two of the key functions performed by traders are the pricing and hedging of unlisted derivatives. Pricing, in particular, needs to be performed in an arbitrage-free manner. To be able to fulfil these functions, practitioners rely on models of asset prices and other market-related variables, such as interest rates. To ensure that prices are arbitrage-free, and that hedges are set up and maintained correctly, they need to calibrate the model that they are using to observable market information, such as the prices of listed derivatives. This calibration process may take place daily, for instance.

Ideally, calibration would be conducted using a calibration function,

$$\Phi : \mathbb{R}^N \to \mathbb{R}^M : \underline{V} \mapsto \underline{\phi},$$

which takes an N-sized vector, $\underline{V}$, of listed derivative prices and values of other market-observable variables, and outputs an M-sized vector, $\underline{\phi}$, of parameter values associated with the financial model being used. Most derivative pricing formulae cannot be inverted to find such a calibration function explicitly. Instead, calibration is usually performed using numerical techniques.

The amount of time required to perform calibration will depend on the model being used. Typically, models that are relatively simple, but which do not realistically describe the dynamics in a given market, can be calibrated quickly. The Black-Scholes framework is a good example of this. Conversely, models that are better at describing a market's dynamics are usually more complex and take longer to calibrate. In practice, the models for which the calibration time is extensive may be excluded from use, owing to a lack of practicality. Hernandez (2016) points out that, depending on the calibration time, whole classes or sub-classes of models that have otherwise desirable properties may be excluded as a result.

1.2 Using neural networks to solve calibration problems

One of the useful properties of neural networks is that the computation of a neural network can take place very quickly on most computers (Hernandez, 2016). This speed forms the primary motivation for attempting to use neural networks for model calibration. If it can be shown that neural networks are able to calibrate more complex models adequately, then the calibration time for those models would no longer be the dominant model selection criterion (Hernandez, 2016). This, in turn, means that the model could be judged primarily on its merits, such as its efficacy in pricing and hedging, rather than the computational time required to calibrate that model.

In order to access this speed advantage, though, a neural network must first be trained for its specific application. For example, a neural network may be trained to calibrate a specific stochastic volatility model, and may then be used on an equity derivative trading desk. Hernandez (2016) stresses that the training for model calibration is carried out in an *offline* manner. In other words, the training process, which is computationally expensive and which may take a long time, would take place on a computer system separate from that used by trading desks. Once this training process is complete, however, the trained neural network can be brought *online*, that is, it can be used on trading desks for calibration. Training neural networks and using them for calibration thus takes place in two distinct phases.

Another useful property of neural networks is that they can be used to approximate non-linear and complicated functions well (Ng, 2013). As such, neural networks appear to be suitable for solving calibration problems. This stems partly from their composition, an example of which is shown in Figure 1.1. The figure depicts a *feedforward neural network*, which is the type of neural network that is focused on in this dissertation.

Neural networks are made up of multiple layers of neurons, resembling the analogous neurological structures in organisms. Neural networks process information by receiving it through the *input layer* (orange, left) and passing it to the *hidden layers* (blue), in which various computations are performed. In particular, the neurons in each of the hidden layers receive the outputs of neurons in the preceding layer, as shown by the arrows in Figure 1.1. Within these neurons, non-linear *activation functions* are applied, thereby introducing the aforementioned non-linearity to the neural network. After all of these computations are performed, an output is produced through the *output layer* (orange, right). It should therefore be clear that a neural network can be regarded as a function, $f_{\mathcal{N}}$ (Haykin, 1994).

To reiterate, Hernandez (2016) and Mavuso *et al.* (2017) point out that using neu-

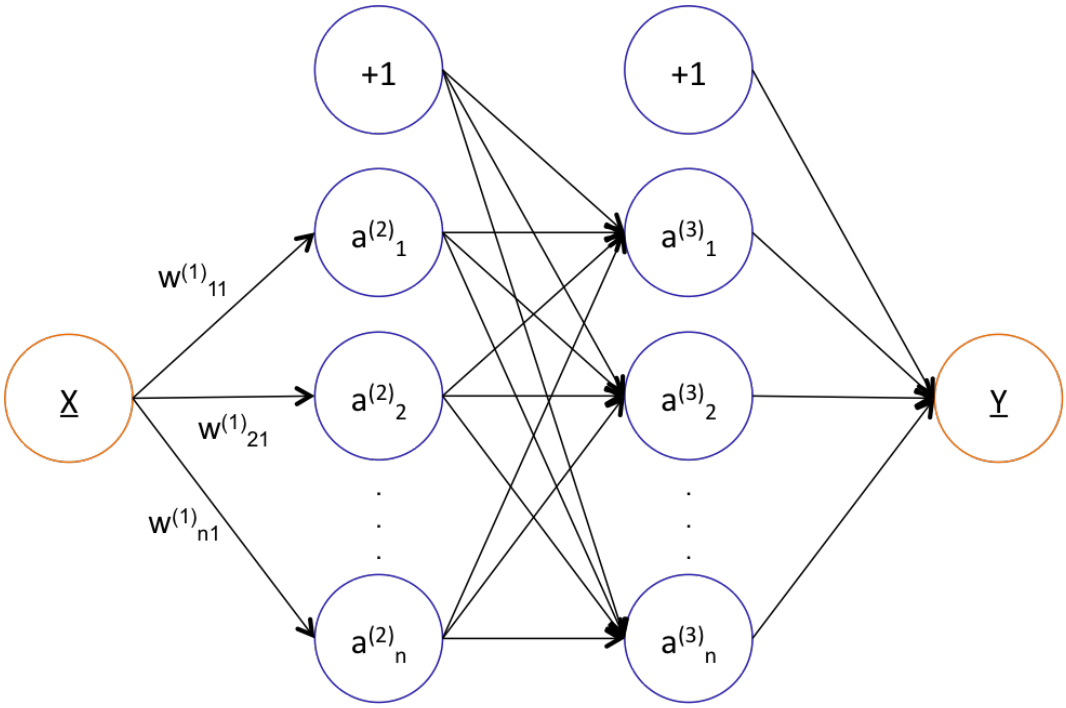

Fig. 1.1: A simple feedforward neural network

ral networks for calibration is a two-stage process: firstly, historical and simulated data is used to train the neural network, with the goal being to approximate the relevant calibration function. In other words, the idea is to train a neural network such that

$$f_{\mathcal{N}} \approx \Phi.$$

This training would need to take place relatively infrequently (Hernandez, 2016). Once training is complete, the second stage of the process, namely, applying the neural network to calibration, may commence. As highlighted above, the computation of the neural network using current market information can take place very quickly.

One of the idiosyncrasies of using neural networks in model calibration is that two sets of parameters are involved: those of the financial model being calibrated, and the *hyperparameters* of the neural network (Hernandez, 2016). This latter set of parameters determines the architecture and other features of the neural network, including the number of hidden layers present in the neural network, the number of neurons per layer, and the activation functions used in the hidden and output layers.

Not all neural network architectures will produce the same results or perform

comparably. As such, the optimal hyperparameters need to be found for each calibration problem (Hernandez, 2016). This involves training a range of different neural networks, with each being specified by a unique combination of hyperparameter values.

1.3 Previous work on neural-network calibration

Hernandez (2016) recently produced the seminal paper incorporating neural networks into financial model calibration. In it, he demonstrates the effective application of neural-network calibration to the (one-factor) model specified by Hull and White (1993). In order to discuss the calibration of this model, we define r_t to be the short rate at time t, with dynamics specified by the Hull-White model,

$$dr_t = (\theta(t) - \alpha r_t)\, dt + \sigma\, dW_t \qquad r_0 = r \text{ (constant)},$$

where $\theta(t)$ is the level of mean reversion of the short rate at time t, α specifies the rate of mean reversion, σ is the volatility of the short rate, and W_t is a standard Brownian motion.

Hernandez (2016) states that $\theta(t)$ is chosen to replicate the current yield curve. Brigo and Mercurio (2006) outline how this is done. Firstly, define $P^{mkt}(0,T)$ to be the market discount factor for maturity T. The market instantaneous forward rate at time 0 for maturity T, $f^{mkt}(0,T)$, is given by

$$f^{mkt}(0,T) = -\frac{\partial \ln P^{mkt}(0,T)}{\partial T}.$$

The level of mean reversion of the short rate at time t is then fixed by setting

$$\theta(t) = \frac{\partial f^{mkt}(0,T)}{\partial T}\bigg|_{T=t} + \alpha f^{mkt}(0,t) + \frac{\sigma^2}{2\alpha}(1 - e^{-2\alpha t}).$$

By fixing the values of $\theta(t)$, the calibration problem is reduced to finding

$$\underline{\phi} = (\alpha, \sigma)^\top.$$

As inputs to the neural-network calibration, Hernandez (2016) supplies 156 swaption prices and 44 points on a bootstrapped yield curve from the period between 2 January 2013 and 1 June 2016, so that the calibration function is

$$\Phi : \mathbb{R}^{200} \to \mathbb{R}^2.$$

Overall, Hernandez (2016) shows that the performance of this neural-network calibration (in terms of the error metrics defined in the paper) is comparable to that

of traditional numerical calibration methods, for periods up to 6-12 months from initial training. In light of this, he recommends retraining the neural network every few months.

The work by Mavuso *et al.* (2017) builds on this foundation, and looks at a related calibration problem. Their innovation is to apply neural-network calibration to a mixture model consisting of two Hull-White models, where the dynamics of the factors driving the short rates, $r_t^{(1)}$ and $r_t^{(2)}$, are given by

$$dr_t^{(1)} = (\beta_1(t) - \alpha_1 r_t^{(1)})\, dt + \sigma_1\, dW_t,$$

and

$$dr_t^{(2)} = (\beta_2(t) - \alpha_2 r_t^{(2)})\, dt + \sigma_2\, dW_t,$$

with the modelled short rate, r_t, being determined by

$$\mathbb{P}(r_t \leq x) = \pi \mathbb{P}\big(r_t^{(1)} \leq x\big) + (1-\pi)\mathbb{P}\big(r_t^{(2)} \leq x\big),$$

where $\pi \in [0, 1]$. Unlike Hernandez (2016), Mavuso *et al.* (2017) also calibrate the level of mean reversion parameters, $\beta_1(t)$ and $\beta_2(t)$, for all 44 maturities from which the yield curve is constructed. Mavuso *et al.* (2017) thus train a calibration function $\Phi : \mathbb{R}^{200} \to \mathbb{R}^{49}$. As for Hernandez (2016), the results produced by Mavuso *et al.* (2017) compared favourably with those derived from traditional calibration techniques. Both sets of results point clearly to the suitability of neural networks to model calibration, even in a calibration problem as complex as that undertaken by Mavuso *et al.* (2017).

1.4 Calibrating stock price models with neural networks

The previous research into neural-network calibration, described above, has shown the effective use of such calibration for interest-rate modelling. The focus of this dissertation is on calibrating models of stock prices. In the examples considered here, we calibrate these models using European call option prices. We look at model calibration for the model by Black and Scholes (1973) and the constant elasticity of variance (CEV) model by Cox (1975) in simplified experiments to determine the suitability of neural-network calibration to these models. Subsequently, the main experiment and results focus on the Heston model. The unique feature of this model, compared to the other two, is that it incorporates stochastic volatility. Finally, we modify this experiment to investigate the limitations of neural-network calibration.

In order to avoid a ”black box” approach to constructing and training neural networks, Chapter 2 of this dissertation is dedicated to laying out the formal mathematics underlying neural-network computations and training algorithms. Furthermore, the code that is deployed in the two simplified experiments described in Chapter 3 is based on the mathematics developed in Chapter 2. For the benefit of those to whom neural networks are unfamiliar, the key facets of working with neural networks, such as hyperparameter optimisation and the use of different training procedures, are also described.

Chapter 2

The Mathematics of Feedforward Neural Networks

2.1 Overview

The intuitive understanding of the architecture and computation of feedforward neural networks was presented in the previous chapter. This chapter serves as an exposition of the associated formal mathematics.

This exposition is important for several reasons. It provides a clear framework through which to understand, at a technical level, both the neural networks and the training algorithms that are described in the subsequent chapters. This helps practitioners and others who are interested in neural networks to be able to identify the neural networks that are suitable for a particular problem.

In addition, a clear description of the mathematics of neural networks allows the architecture and computation of neural networks to be translated into code relatively easily. This theoretical exposition is also useful for identifying, understanding, and resolving the potential difficulties that may be encountered in neural-network training and use.

The sections below cover the mathematical representation of feedforward neural networks (Section 2.2) and the gradient-descent training algorithm (Section 2.3).

2.2 Mathematical representation of feedforward neural networks

The material and notation used in this section is derived from that presented by Ng (2013). Consider a feedforward neural network, $f_{\mathcal{N}}$, with L layers, that is, $L-2$ hidden layers. Layer l, where $1 \leq l \leq L$ and $l \in \mathbb{Z}$, is comprised of n_l neurons. The value of the output of the i^{th} neuron in layer l, where $0 \leq i \leq n_l$ and $i \in \mathbb{Z}$, is denoted $a_i^{(l)}$ and is a function of the outputs of the neurons in layer $l-1$. We thus

define

$$\underline{a}^{(l)} := \begin{bmatrix} a_0^{(l)} \\ a_1^{(l)} \\ \vdots \\ a_i^{(l)} \\ \vdots \\ a_{n_l}^{(l)} \end{bmatrix} \tag{2.1}$$

to be the vector containing the outputs of the neurons in layer l. We also define the matrices

$$\underline{w}^{(l-1)} := \begin{bmatrix} w_{10}^{(l-1)} & w_{11}^{(l-1)} & \cdots & w_{1n_{l-1}}^{(l-1)} \\ w_{20}^{(l-1)} & w_{21}^{(l-1)} & \cdots & w_{2n_{l-1}}^{(l-1)} \\ \vdots & \vdots & \ddots & \vdots \\ w_{n_l0}^{(l-1)} & w_{n_l1}^{(l-1)} & \cdots & w_{n_ln_{l-1}}^{(l-1)} \end{bmatrix}. \tag{2.2}$$

These matrices contain the transition weights, $w_{ij}^{(l-1)}$, that are applied to the output from neuron j in layer $l-1$ when calculating the output of neuron i in layer l. Note from (2.2) that $1 \leq i \leq n_l$, and $0 \leq j \leq n_{l-1}$.

The calculation of $a_i^{(l)}$ is performed in two steps. Firstly, an intermediate value, $z_i^{(l)}$, is computed for each neuron in layer l as the weighted sum of the outputs of the neurons in layer $l-1$, that is,

$$\begin{aligned} z_i^{(l)} &= \sum_{j=0}^{n_{l-1}} w_{ij}^{(l-1)} a_j^{(l-1)} \\ &= \underline{w}_{i\cdot}^{(l-1)} \underline{a}^{(l-1)}. \end{aligned} \tag{2.3}$$

Note that these are not weights in the traditional sense of summing to 1. Furthermore, $\underline{w}_{i\cdot}^{(l-1)}$ denotes the i^{th} row of the matrix defined in (2.2). Since the second line of (2.3) is a vectorised representation of the summation in the first line, it can be implemented via matrix multiplication.

Secondly, an activation function, g, is applied to these intermediate values to get the outputs of the neurons in layer l, that is,

$$a_i^{(l)} = g(z_i^{(l)}),$$

or

$$\underline{a}^{(l)} = g(\underline{z}^{(l)}),$$

where $\underline{z}^{(l)}$ is the (column) vector of intermediate values for layer l, calculated in the previous step. Note that the activation function applied in the output layer may differ from that used in the hidden layers. In the second line above, g is applied element-wise. Furthermore, let $a_0^{(l)} = 1$ for $1 \leq l < L$. This corresponds with what are known as *bias units*. Bias units, which occur in the hidden layers, are neurons

that output a constant value of 1, and hence do not take any input from neurons in the preceding layers. They are analogous to the intercept term used in regression. The above expressions can now be used to specify the algorithm for computing a forward pass of the neural network, that is, evaluating $f_{\mathcal{N}}$ for a given input vector. Suppose that we have an input vector, $\underline{x}$, and apply the activation functions g and g_o in the hidden and output layers, respectively. We then compute the forward pass as follows:

FORWARD PASS OF A FEEDFORWARD NEURAL NETWORK

1. Set $a_i^{(1)} = x_i$
2. For $2 \leq l \leq L-1$ and $1 \leq i \leq n_l$:
   (a) calculate $z_i^{(l)}$
   (b) calculate $a_i^{(l)} = g(z_i^{(l)})$
3. For $l = L$ and $1 \leq i \leq n_L$:
   (a) calculate $z_i^{(L)}$
   (b) calculate and set $\hat{\underline{y}} = f_{\mathcal{N}}(\underline{x}) = g_o(\underline{z}^{(L)})$

Using the mathematical representation developed above, we can state one of the key results pertaining to neural networks, namely, the Universal Approximation Theorem. The following version of the theorem is laid out in Haykin (1994), whereas a proof can be found in Cybenko (1989).

Theorem 2.1. *Let g be a non-constant, bounded, and monotone-increasing continuous function. Let I_{n_1} denote the n_1-dimensional unit hypercube $[0,1]^{n_1}$. The space of continuous functions on I_{n_1} is denoted by $\mathcal{C}(I_{n_1})$. Then, given any function $f \in \mathcal{C}(I_{n_1})$ and $\epsilon > 0$, there exist an integer n_2 and sets of real constants α_i and $w_{ij}^{(1)}$, where $i = 1, ..., n_2$ and $j = 1, ..., n_1$, such that we may define*

$$f_{\mathcal{N}}(\underline{x}) = \sum_{i=1}^{n_2} \alpha_i \cdot g\left(\sum_{j=1}^{n_1} w_{ij}^{(1)} x_j + w_{i0}^{(1)} \right)$$

as an approximate realisation of the function f; that is,

$$|f_{\mathcal{N}}(\underline{x}) - f(\underline{x})| \quad < \quad \epsilon$$

for all $\underline{x} \in [0,1]^{n_1}$.

In simple terms, this theorem shows that any continuous function can be approximated by a neural network. Furthermore, the result states that a single hidden layer is sufficient for such an approximation. However, as an existence result, the theorem does not indicate how many neurons a single hidden layer would require to achieve a suitable approximation for a function f, nor does it imply that

neural networks with a single hidden layer are optimal for training (Haykin, 1994). Nonetheless, it is this result that underlies the utility of neural networks in approximating various functions, as discussed in Section 1.2.

This mathematical framework can now be used to describe algorithms used to train neural networks. In particular, the relatively simple gradient-descent algorithm is presented in the following section.

2.3 Training neural networks using gradient descent with backpropagation

In the context of model calibration, we are interested in *supervised learning*. This is a machine learning procedure whereby a function is learned (approximated) using known pairs of inputs to and outputs from that function. In contrast, *unsupervised learning* is the process of inferring a function that represents the structure of data for which there is no prior classification or categorisation (Haykin, 1994; Ng, 2013).

Our goal is to train a neural network to approximate the calibration function for a particular financial model. In order to carry out this training, we need a dataset comprised of two sets of vectors. The vectors in the first set, denoted $\underline{x}^{(k)}$, consist of market data, such as observed derivative prices, asset prices, and possibly other market variables. The vectors in the second set, paired with the vectors in the first set and denoted $\underline{y}^{(k)}$, consist of corresponding parameter values for the financial model for which a calibration function is sought (Mavuso *et al.*, 2017; Hernandez, 2016). Each vector pair in a dataset that is used for supervised learning is referred to as a *training example*.

Most of the training algorithms that are used today are adaptations of the gradient-descent algorithm (Ruder, 2016). In general, gradient descent is used as an optimisation algorithm, that is, it is used to find the minimum of a given function. In the context of supervised learning problems, the gradient-descent algorithm is used to minimise some cost function. This cost function, $E(\underline{x}, \underline{y}; \underline{w})$, is a function of the training data and the weights used in the neural network. It quantifies the discrepancy between the output of the neural network, $\hat{\underline{y}}^{(k)} = f_{\mathcal{N}}(\underline{x}^{(k)})$, and the values that the neural network is being trained to estimate, $\underline{y}^{(k)}$.

Although there are a number of different algorithms used to train neural networks, gradient descent with backpropagation is one of the simplest and most commonly-used algorithms (Ng, 2013). As such, it has been implemented for the Black-Scholes and CEV model experiments described in the next chapter.

It is important to note that training a neural network amounts to adjusting the weights used in the neural network. With this in mind, training by gradient descent is carried out by making incremental adjustments to the values of the weights, based on the partial derivatives of the cost function with respect to each of the weights in the neural network. The magnitude of these incremental adjustments is based on a learning rate parameter, α. Moreover, the above-mentioned partial derivatives are calculated using the training data.

Training algorithms thus work as follows. The values $\hat{\underline{y}}^{(k)} = f_{\mathcal{N}}(\underline{x}^{(k)})$ are calculated for all of the training examples in the dataset. The calculated $\hat{\underline{y}}^{(k)}$ values are

then used to compute the cost function and above-mentioned partial derivatives, after which the weights are adjusted (Ng, 2013). Each iteration of this process is referred to as an *epoch*.

In order to specify and implement the gradient-descent algorithm, we need to be able to explicitly calculate the partial derivatives of the cost function with respect to individual weights in the neural network. Given that a feedforward neural network essentially consists of a composition of functions, we thus require the following key result for finding those partial derivatives. To obtain this result, we draw on the specifications of the gradient-descent algorithm given by Ng (2013) and Nielsen (2015) [1].

Theorem 2.2. *Let $f_{\mathcal{N}}$ be a feedforward neural network with L layers, n_l neurons in each layer, and weights $w_{ij}^{(l-1)}$. Let g be a continuously differentiable activation function applied in the hidden layers of the neural network, and further let $E(\underline{x}, \underline{y}; \underline{w})$ be the cost function associated with the neural network.*

Then for all i, j and $2 \leq l \leq L-1$, we have the (backwards recursive) relationship

$$\frac{\partial E}{\partial w_{ij}^{(l-1)}} = \delta_i^{(l)} \cdot a_j^{(l-1)} = g'(z_i^{(l)}) \cdot a_j^{(l-1)} \sum_{k=1}^{n_{l+1}} \delta_k^{(l+1)} \cdot w_{ki}^{(l)}$$

where

$$\delta_i^{(l)} := \frac{\partial E}{\partial a_i^{(l)}} \cdot \frac{\partial a_i^{(l)}}{\partial z_i^{(l)}}.$$

Proof. From a simple application of the chain rule, it is clear that

$$\begin{aligned} \frac{\partial E}{\partial w_{ij}^{(l-1)}} &= \frac{\partial E}{\partial a_i^{(l)}} \cdot \frac{\partial a_i^{(l)}}{\partial z_i^{(l)}} \cdot \frac{\partial z_i^{(l)}}{\partial w_{ij}^{(l-1)}} \\ &= \delta_i^{(l)} \cdot \frac{\partial z_i^{(l)}}{\partial w_{ij}^{(l-1)}}. \end{aligned} \tag{2.4}$$

Now express each of the terms on the right-hand side of 2.4 as

$$\frac{\partial z_i^{(l)}}{\partial w_{ij}^{(l-1)}} = a_j^{(l-1)}$$

and

$$\delta_i^{(l)} = \frac{\partial E}{\partial a_i^{(l)}} \cdot \frac{\partial a_i^{(l)}}{\partial z_i^{(l)}} = \frac{\partial E}{\partial a_i^{(l)}} \cdot g'(z_i^{(l)}).$$

In order to find $\frac{\partial E}{\partial a_i^{(l)}}$, consider E to be a function of the (non-bias) neuron outputs in the $(l+1)^{th}$ layer, that is, $E\left(a_1^{(l+1)}, a_2^{(l+1)}, ..., a_{n_{l+1}}^{(l+1)}\right)$, and note that each $a_k^{(l+1)}$ $(1 \leq k \leq n_{l+1})$ is a function of $a_i^{(l)}$ by construction.

[1] The "Backpropagation" web page, authored by John McGonagle and others, was also consulted.

Then, by an application of the chain rule, we get the total derivative

$$\begin{aligned}\frac{\partial E}{\partial a_i^{(l)}} &= \left(\frac{\partial E}{\partial a_1^{(l+1)}}\right)\left(\frac{\partial a_1^{(l+1)}}{\partial a_i^{(l)}}\right) + \left(\frac{\partial E}{\partial a_2^{(l+1)}}\right)\left(\frac{\partial a_2^{(l+1)}}{\partial a_i^{(l)}}\right) + \ldots + \left(\frac{\partial E}{\partial a_{n_{l+1}}^{(l+1)}}\right)\left(\frac{\partial a_{n_{l+1}}^{(l+1)}}{\partial a_i^{(l)}}\right) \\ &= \sum_{k=1}^{n_{l+1}} \frac{\partial E}{\partial a_k^{(l+1)}} \cdot \frac{\partial a_k^{(l+1)}}{\partial a_i^{(l)}} \\ &= \sum_{k=1}^{n_{l+1}} \delta_k^{(l+1)} \cdot \frac{\partial z_k^{(l+1)}}{\partial a_i^{(l)}}, \text{ since } \delta_k^{(l+1)} = \frac{\partial E}{\partial a_k^{(l+1)}} \cdot \frac{\partial a_k^{(l+1)}}{\partial z_k^{(l+1)}} \\ &= \sum_{k=1}^{n_{l+1}} \delta_k^{(l+1)} \cdot w_{ki}^{(l)}.\end{aligned}$$

Therefore,

$$\delta_i^{(l)} = g'(z_i^{(l)}) \cdot \sum_{k=1}^{n_{l+1}} \delta_k^{(l+1)} \cdot w_{ki}^{(l)}.$$

Substituting the derived expressions for $\delta_i^{(l)}$ and $\frac{\partial z_i^{(l)}}{\partial w_{ij}^{(l-1)}}$ into 2.4 yields the result.

□

Corollary 2.3. *For the feedforward neural network described in Theorem 2.2, and assuming that g_o is a continuously differentiable activation function used in the output layer, the partial derivatives of the cost function, with respect to the weights applied to the neuron outputs of the last hidden layer, are given by*

$$\frac{\partial E}{\partial w_{ij}^{(L-1)}} = \delta_i^{(L)} \cdot a_j^{(L-1)} = \frac{\partial E}{\partial \hat{y}_i} \cdot g_o'(z_i^{(L)}) \cdot a_j^{(L-1)}.$$

It should be clear from the statement and proof of Theorem 2.2 that there is a backwards recursive relationship between the $\delta_i^{(l)}$ terms for adjacent layers. Since the $\delta_i^{(L)}$ terms are used to calculate the $\delta_i^{(L-1)}$ terms, and so on, the calculations of the partial derivatives $\frac{\partial E}{\partial w_{ij}^{(l-1)}}$ are made simpler. This also illustrates the origin of the term backpropagation.

To carry out a training procedure, the architecture of the neural network, and the cost function used to train the neural network, must first be specified. In addition, the weights used in the neural network need to be randomised, and these are typically centred around 0 (Ng, 2013). The learning rate parameter, α, also needs to be set. Care needs to be taken in setting this value, since an α value that is too large will prevent the gradient-descent algorithm from converging, and the computed cost will blow up (Ng, 2013). Finally, some convergence criterion (the magnitude of the cost function value), or a fixed number of epochs, must be set to determine how many iterations of the algorithm must be performed.

One final result, based on the specification of the gradient-descent algorithm given by Nielsen (2015), is needed to specify the gradient-descent algorithm.

Lemma 2.4. *Let $(\underline{x}^{(k)}, \underline{y}^{(k)})$ $(1 \leq k \leq K, k \in \mathbb{Z})$ be training examples used to train a feedforward neural network via the gradient-descent algorithm. Then the partial derivative of the cost function with respect to a given weight in the neural network is the mean of the same partial derivatives calculated for all training examples, that is,*

$$\frac{\partial E}{\partial w_{ij}^{(L-1)}} = \frac{1}{K}\sum_{k=1}^{K} \frac{\partial E}{\partial w_{ij}^{(L-1)}}\bigg|_{x^{(k)}} = \frac{1}{K}\sum_{k=1}^{K} \left(g'(z_i^{(l)}) \cdot a_j^{(l-1)} \sum_{p=1}^{n_{l+1}} \delta_p^{(l+1)} \cdot w_{pi}^{(l)} \right)\bigg|_{x^{(k)}}.$$

With these preliminaries in place, and given the above results, it is now possible to specify an iteration of the gradient-descent training algorithm as follows:

GRADIENT-DESCENT TRAINING ALGORITHM (ONE EPOCH)

1. For all training examples, perform a forward pass of the neural network to calculate $\hat{\underline{y}}^{(k)} = f_{\mathcal{N}}(\underline{x}^{(k)})$
2. For $l = L$:
   (a) Calculate $\delta_i^{(L)} = \frac{\partial E}{\partial \hat{y}_i} \cdot g_o'(z_i^{(L)})$ for all $1 \leq i \leq n_L$
   (b) Use this to calculate $\frac{\partial E}{\partial w_{ij}^{(L-1)}}$
   (c) Adjust weights by $\Delta w_{ij}^{(L-1)} = -\alpha \frac{\partial E}{\partial w_{ij}^{(L-1)}}$, that is, set $w_{ij}^{(L-1)} = w_{ij}^{(L-1)} + \Delta w_{ij}^{(L-1)}$
3. For $L - 1 \geq l \geq 2$:
   (a) Calculate $\frac{\partial E}{\partial w_{ij}^{(l-1)}}$ as per Theorem 2.2 and Lemma 2.4, using $\delta_i^{(l)}$
   (b) Adjust weights by $\Delta w_{ij}^{(l-1)} = -\alpha \frac{\partial E}{\partial w_{ij}^{(l-1)}}$, that is, set $w_{ij}^{(l-1)} = w_{ij}^{(l-1)} + \Delta w_{ij}^{(l-1)}$

This algorithm is what appears in the code given in Appendix B.

This concludes the exposition of the mathematics of feedforward neural networks, as well as the simple gradient-descent algorithm used to train such neural networks. To reiterate, the significance of this chapter lies in providing a clear understanding of how neural networks function and how they are trained. It is also important for understanding the experiments described in the following chapters.

Chapter 3

Neural-network Training and Model Calibration in Simplified Experiments

3.1 Overview

The aim of this chapter is to use the mathematics developed in Chapter 2 to construct and train neural networks. This will be done in two simplified experiments. The simplification here stems from the fact that these model-based experiments do not completely mirror real-world calibration. Nonetheless, these experiments are important for demonstrating the ability of neural networks to learn the kinds of non-linear mathematical relationships that are prevalent in calibration problems.

The two experiments are based on the Black-Scholes and constant elasticity of variance (CEV) models. In the Black-Scholes experiment, a neural network is trained to calibrate one parameter, namely, implied volatility. The CEV experiment is an extension of the Black-Scholes experiment, in that most of the elements of the Black-Scholes experiment are retained, with the modification that two parameters are calibrated instead of one.

In addition to showing the successful translation of the neural-network theory and mathematics into usable code that performs as expected, these experiments help to illustrate two important facets of training neural networks. Specifically, the Black-Scholes experiment is used to show hyperparameter optimisation, and the selection of the best-performing neural network out of a group of neural networks trained on the same data. The CEV experiment shows how training differs according to the optimisation scheme (that is, the training algorithm) used, and how this difference translates into the results obtained.

3.2 Black-Scholes experiment

Consider the time-t price of a stock that does not yield dividends, S_t. Merton (1973) specifies dynamics for the stock price as

$$dS_t = \mu S_t\, dt + \sigma S_t\, dW_t \qquad S_0 = s \text{ (constant)},$$

Tab. 3.1: Values used for hyperparameter optimisation in the Black-Scholes experiment

Activation function	softplus	ReLU			
No. of hidden layers	1	2	3	4	
No. of neurons per hidden layer	4	8	16	32	64

where W_t is a standard Brownian motion, and μ and σ are the constant drift and implied volatility parameters, respectively. The analytical formula for the risk-neutral price of a European call option, derived by Black and Scholes (1973), appears in Appendix A for completeness.

The idea underlying this experiment is to train a neural network to estimate implied volatility, given the price of a European call option. More formally, let V be the price of a European call option. Then the neural network is trained to approximate the function

$$\Phi : \mathbb{R} \to \mathbb{R} : V \mapsto \sigma.$$

This implies fixing the parameters S_t, K, T, and r. As such, it is clear that this experiment does not mimic a real-world calibration scenario, in which the aforementioned parameters (S_t in particular) would vary. Rather, as stated before, the objective is to demonstrate the ability of neural networks to approximate such non-linear relationships in a simplified setting. Moreover, the experiment has been designed in this way to reflect a similar experiment implemented by Mavuso *et al.* (2017) to establish comparable results.

In light of the above, the parameter values $S_t = 100$, $K = 100$, $(T - t) = 1$, and $r = 0$ are chosen and fixed, and are similar to those used by Mavuso *et al.* (2017). The price of a European call option thus becomes a function of one parameter, namely σ.

To generate data, we let $\sigma \in [0.01, 1.5]$ and take 50000 values spaced evenly-apart on that interval as the sample of parameter values. Each implied volatility value, $\sigma^{(k)}$, is then used to generate the corresponding European call option price, $V^{(k)}$, using the Black-Scholes option pricing formula given in Appendix A. These option price and implied volatility pairs are used as the training examples. In order to specify the neural-network architectures, we utilise the softplus and rectified linear unit (ReLU) activation functions, defined as

$$\begin{aligned} \text{softplus}(x) &:= \ln(1 + e^x) \\ \text{ReLU}(x) &:= \max(x, 0). \end{aligned}$$

Hyperparameter optimisation is carried out by training 40 different neural networks, based on a grid search through the hyperparameter values given in Table 3.1. These values are chosen based on those considered by Hernandez (2016) and Mavuso *et al.* (2017) in their hyperparameter optimisations. The two activation functions mentioned are used in the hidden layers, while the linear activation function, $g_o(x) = x$, is used in the output layer.

Although the ReLU activation function violates the differentiability requirements of Theorem 2.2, it is nonetheless useful for comparison, since it is commonly

Tab. 3.2: Black-Scholes calibration – mean squared error values for different neural network architectures

Softplus

Neurons \ Layers	1	2	3	4
4	0.011000	0.000130	0.000106	0.000200
8	0.010000	0.000700	0.000150	0.000200
16	0.008100	0.000170	0.000090	0.000190
32	0.009700	0.000200	0.000077	0.000200
64	0.006800	0.000200	0.000090	0.000110

ReLU

Neurons \ Layers	1	2	3	4
4	0.000250	0.000185	0.000110	0.090000
8	0.092000	0.000110	0.000120	0.011000
16	0.000156	0.000120	0.000120	0.000170
32	0.000097	0.000120	0.000110	0.000085
64	0.000109	0.000097	0.000079	0.000078

used in practice regardless of this shortcoming. To avoid producing NaNs in the partial derivative calculations of the gradient-descent algorithm, the derivative of the ReLU function is set to 1 where $x = 0$ [1].

The MATLAB code that is used to construct and train these neural networks, and which implements a simple gradient-descent algorithm for training, is given in Appendix B. For all hyperparameter combinations, a standard 500 epochs are used to train the corresponding neural network, in line with the methods employed by Hernandez (2016) and Mavuso *et al.* (2017). Finally, the mean squared error (MSE) metric is selected as the cost function. In this context, the MSE is defined by

$$\text{MSE}(\underline{\sigma}, \underline{\hat{\sigma}}; \underline{w}) := \frac{1}{K} \sum_{k=1}^{K} (\sigma^{(k)} - \hat{\sigma}^{(k)})^2,$$

where

$$\hat{\sigma}^{(k)} = f_{\mathcal{N}}(V^{(k)}).$$

Table 3.2 shows the mean squared error calculated for each neural network using a smaller testing dataset consisting of 5000 examples. The mean squared error is used as the criterion for selecting the optimal neural network in this experiment. Naturally, a lower error is regarded as better: it indicates that the neural network is better at calibrating implied volatility, given the option price.

From Table 3.2, we see that the neural network that uses the softplus activation function, and which consists of 3 hidden layers and 32 neurons per hidden layer, is

[1] This is the recommendation given by a practitioner, James McCaffrey, on his personal website ("Two Ways to Deal with the Derivative of the ReLU Function").

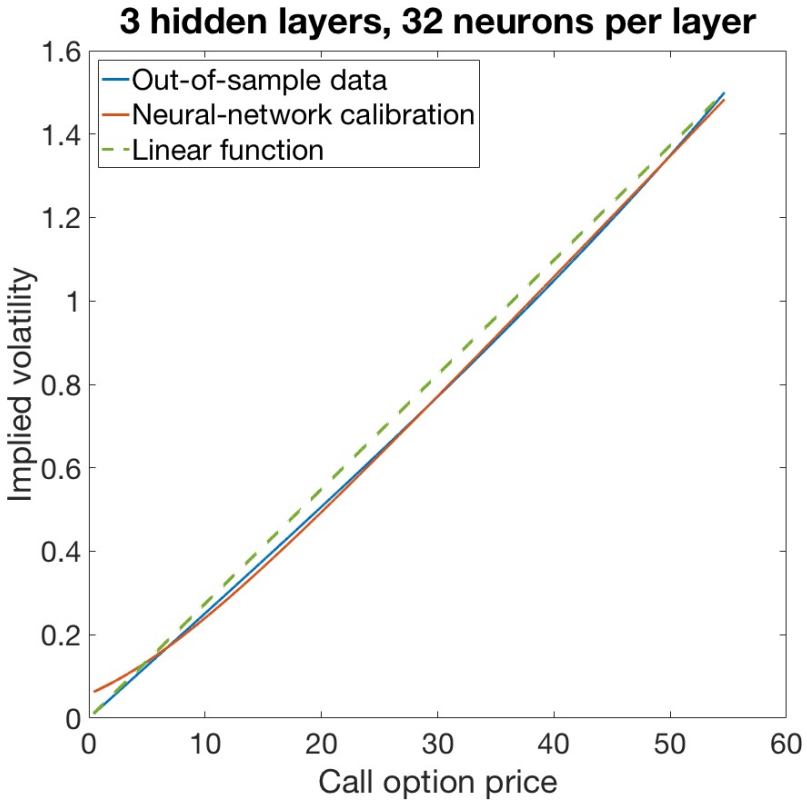

Fig. 3.1: Black-Scholes calibration — Neural network: softplus activation, 3 hidden layers, 32 neurons per layer

the best at calibrating implied volatility out of those that have been trained. One of the implications of this result is that neural networks with more hidden layers and more neurons per hidden layer do not necessarily perform better at a given task than other, smaller neural networks.

Figure 3.1 gives us a visual indication of how well the optimal neural network calibrates implied volatility. The blue line shows the true implied volatility values that correspond with the option prices, while the orange line shows the implied volatility values derived from calibration. Overall, these true and calibrated implied volatility values are close together, indicating that this neural network calibrates implied volatility relatively well. Note that the green dashed line in the graph is a linear function, and is plotted to illustrate the non-linear nature of the relationship between call option price and implied volatility in the Black-Scholes framework.

This result can be compared to that obtained by Mavuso *et al.* (2017) in their version of the Black-Scholes experiment, in which a neural network using the sigmoid activation function, and consisting of 4 hidden layers with 64 neurons in each hidden layer, is used for calibration. For clarity, the sigmoid activation function is defined as

$$\text{sigmoid}(x) := \frac{1}{1+e^{-x}}.$$

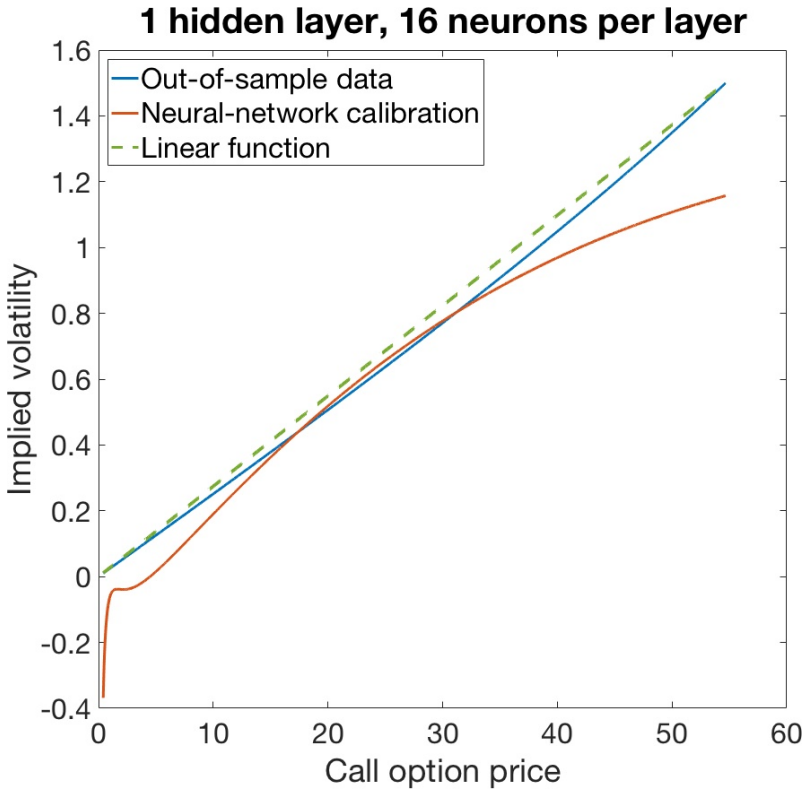

Fig. 3.2: Black-Scholes calibration — Neural network: softplus activation, 1 hidden layer, 16 neurons per layer

The implied volatility values calibrated by Mavuso *et al.* (2017) are very close to the true values in the lower range of call option prices. However, in the higher range of prices, the calibrated values diverge significantly from the true values, and approach 1 asymptotically. It is worth noting that the sigmoid activation function also approaches 1 asymptotically, and that their result can thus be replicated by applying the sigmoid activation function in both the hidden layers and the output layer of the neural network. Nonetheless, the result obtained in our experiment compares favourably with that obtained by Mavuso *et al.* (2017).

For contrast, and to illustrate how poorly neural-network calibration can perform if the wrong architecture is chosen, Figure 3.2 shows the calibration performed by a neural network that uses the softplus activation function as before, but this time consists of 1 hidden layer, with 16 neurons in that layer. The lack of suitability of this neural network to the calibration problem should be obvious, particularly since some of the calibrated implied volatility values are negative, contrary to the assumptions of the Black-Scholes model.

It is clear that even for the best neural-network calibration shown here, applied to a problem as simple as calibrating implied volatility in the Black-Scholes framework, the result is imperfect. This is not necessarily a negative indication for the application of neural networks to model calibration, though, since the hyperparameter optimisation was constrained to a relatively small grid, and thus relatively

few neural networks were trained, applied, and compared. Moreover, and perhaps most importantly, the algorithm used here is not optimised for neural-network training, whereas many of the commonly-used algorithms present in machine-learning code libraries are.

Rather, as stated above, one of the aims of this experiment has been to demonstrate the successful translation of the neural-network theory and mathematics into usable code that performs as expected. In addition, the other objectives have been to use this code to train a neural network to approximate a non-linear relationship between variables, and to demonstrate hyperparameter optimisation for neural networks. In this simple Black-Scholes setting, all three of these aims have been achieved, which allows us to proceed by increasing the complexity of the problem that is addressed. In the following CEV experiment, neural networks are trained to calibrate two parameters instead of one.

3.3 CEV experiment

Under the constant elasticity of variance model, Cox (1975) specifies the dynamics of the stock price, S_t, as

$$dS_t = \mu S_t \, dt + \sigma S_t^{\alpha} \, dW_t \qquad S_0 = s \text{ (constant)},$$

where μ is the drift parameter, σ and α are constants, and W_t is a standard Brownian motion. The corresponding analytical formula for the price of a European call option is derived by Schroder (1989). The statement of this formula given by McWalter (2017) is shown in Appendix A.

As in the Black-Scholes experiment, we fix the values of the market-observable parameters, this time defining five different call options for the purpose of calibrating the two model parameters, α and σ. We set the following values:

$$\begin{array}{ll} S_t = 100, & (K_1, K_2, K_3, K_4, K_5)^\top = (80, 90, 100, 110, 120)^\top, \\ T = 1, & r = 0.05. \end{array}$$

The aim of this experiment is thus to train a neural network to approximate

$$\Phi : \mathbb{R}^5 \to \mathbb{R}^2 : \underline{V} \mapsto (\alpha, \sigma)^\top.$$

To generate training data, we set up a grid of values for α and σ on which $0.5 \leq \alpha \leq 0.9$ and $0.3 \leq \sigma \leq 0.75$, and take all possible combinations of the evenly-spaced parameter values to price all five call options. For clarity, the general setup is shown in Figure 3.3: each point that makes up the surface corresponds with a vector of parameter values, $(\alpha^{(k)}, \sigma^{(k)})^\top$, and the call option prices calculated from those parameter values for a given strike price, $\underline{V}_i^{(k)}$.

Neural-network training is carried out over 500 epochs, using a training dataset consisting of 62500 training examples. The ReLU activation function is used in the hidden layers of the neural networks. The mean squared error is used as the cost function once again, since the magnitudes of values for the α and σ parameters are

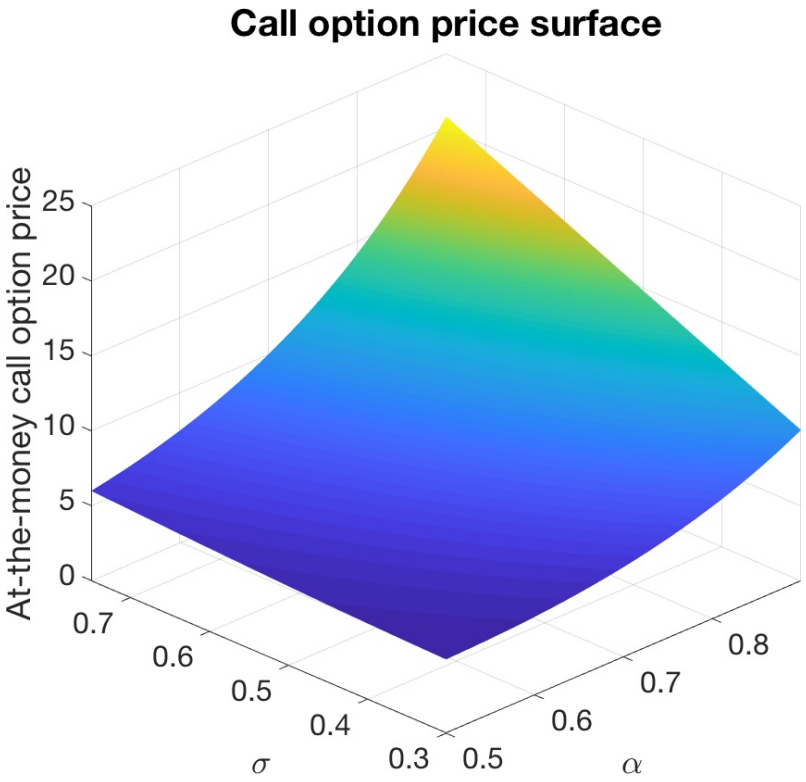

Fig. 3.3: CEV call option pricing surface (at-the-money)

similar. Taking into account that the training examples and neural-network outputs are vectors, the MSE is now given by

$$\text{MSE}(\underline{\phi}, \underline{\hat{\phi}}; \underline{w}) = \frac{1}{K} \sum_{k=1}^{K} ||\underline{\phi}^{(k)} - \underline{\hat{\phi}}^{(k)}||^2,$$

where $||\cdot||$ is the standard Euclidian norm,

$$\underline{\phi}^{(k)} = (\alpha^{(k)}, \sigma^{(k)})^\top$$

is a vector of true parameter values, and

$$\underline{\hat{\phi}}^{(k)} = f_{\mathcal{N}}(\underline{V}^{(k)}) = (\hat{\alpha}^{(k)}, \hat{\sigma}^{(k)})^\top$$

is the corresponding vector of calibrated parameter values.

In this experiment, both the MATLAB code and the Python Keras library are used. The Keras library, written by Chollet (2015), is a high-level machine learning library that can be used to construct, train, and compute neural networks. The reason for training neural networks using these two different sets of code is to compare the results using different training procedures for the same problem. In addition, this provides an opportunity to check whether the implementation and training of neural networks in the MATLAB code, based on the mathematics outlined in Chapter 2, is consistent with those from the Keras library. As in the Black-Scholes experiment, the MATLAB code carries out training using the gradient-descent algorithm, while the Adam (adaptive momentum) optimiser[2] is used in the Python code. One of the advantages of the Adam optimiser is that it incorporates momentum into gradient descent. In other words, the optimiser updates the learning rate according to the rates of change of the partial derivatives of the cost function, and thus typically leads to quicker convergence to the minima of the cost function (Kingma and Ba, 2014).

The graphs in Figures 3.4 and 3.5 show the results for out-of-sample calibration, which is performed using 4900 examples. Recall from the description above that the neural network computes α and σ values from the call option prices, which indicates how these graphs should be read and understood. For the neural network trained in MATLAB, we see that, given price, the neural network's calibration occurs along a line or path on the surface of prices. In other words, it seems that the neural network will calibrate to some set of parameter values, not necessarily the true parameter values, that will approximately reprice the call option correctly.

In contrast, the second graph shows different neural-network behaviour: it appears that the neural network attempts to calibrate to the true parameter values. This is implied by the fact that the neural network calibrates to a surface, rather than a line or path.

While this seems to indicate inconsistency between the MATLAB and Python code implementations, increasing the number of epochs over which the MATLAB code performs training leads to a result that is very similar to that obtained from

[2] The optimisers in the Keras library are improvements to the gradient descent algorithm that have been implemented to make it more efficient.

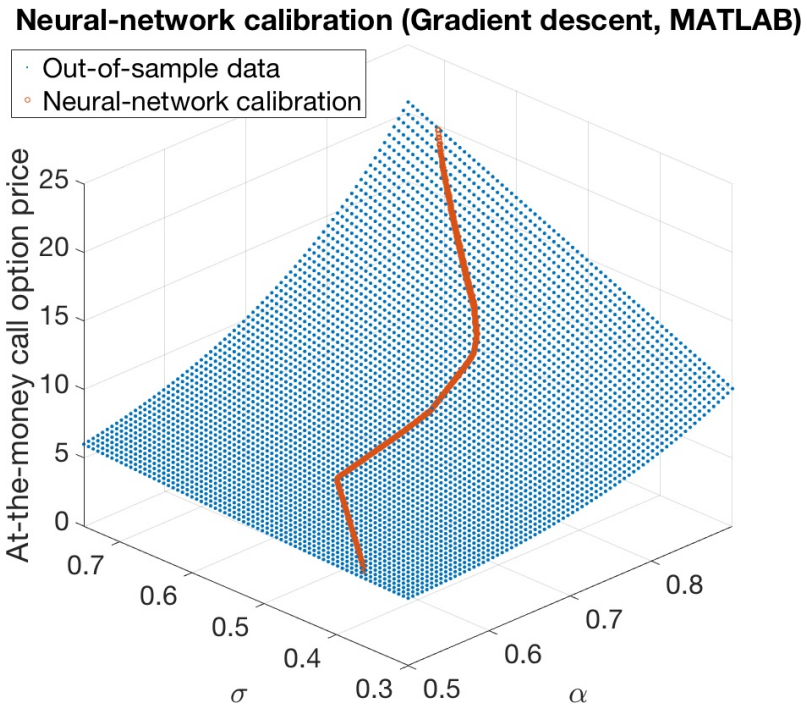

Fig. 3.4: CEV calibration — Training in MATLAB

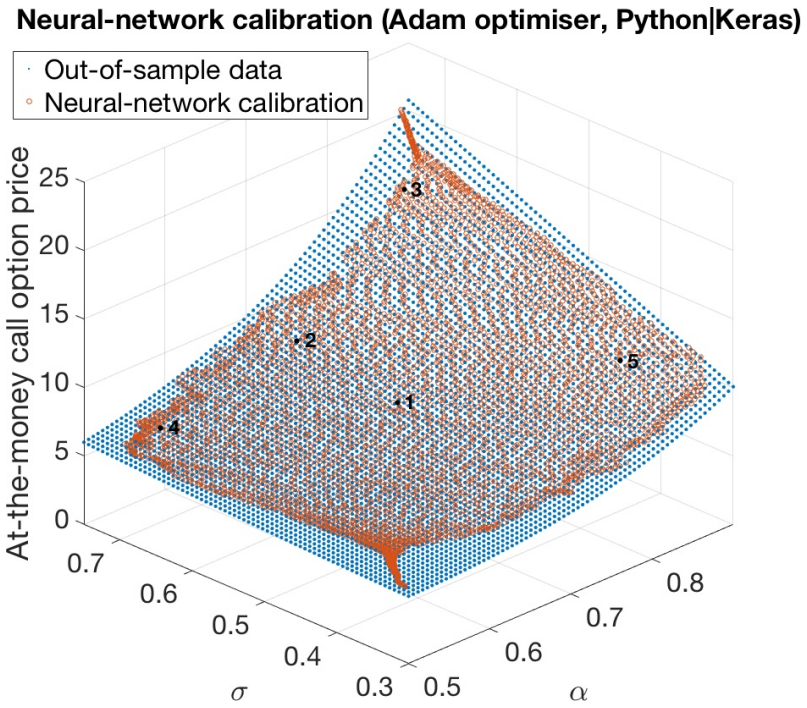

Fig. 3.5: CEV calibration — Training in Python with Keras library

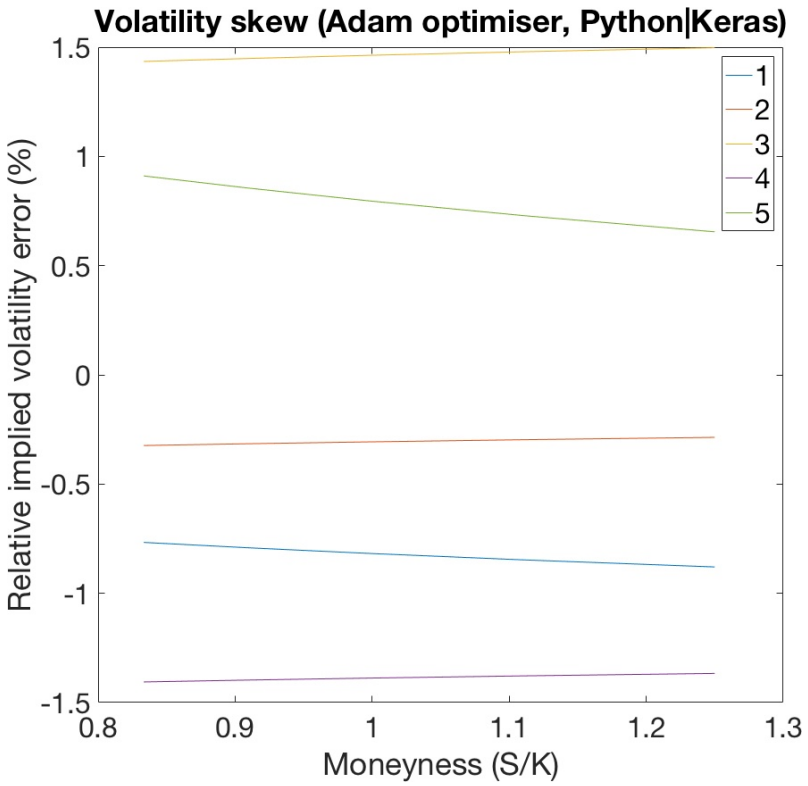

Fig. 3.6: CEV calibration in Python — Volatility skews for out-of-sample points

the Python code. This is in line with our expectations, since the gradient-descent algorithm in the MATLAB code is not modified or optimised for better neural-network training, whereas the Keras library is.

Five out-of-sample points are plotted in Figure 3.5. The calibrated α and σ values associated with these points are used to reprice the five call options. To measure the performance of the calibration, we define the relative implied volatility error (RIVE) for the i^{th} option as

$$\text{RIVE}(V_i) := \frac{IV_i^{mod} - IV_i^{\mathcal{N}}}{IV_i^{mod}},$$

where IV_i^{mod} is the implied volatility corresponding with the model-generated price of the i^{th} option, and $IV_i^{\mathcal{N}}$ is the implied volatility derived from repricing the i^{th} option using the calibrated parameter values. The resulting error skews are shown in Figure 3.6.

The implied volatility values derived through calibration all fall within 1.5% of the true (that is, model-generated) value. It is reasonable to suggest that through hyperparameter optimisation, this result could be further improved.

From this, we can conclude that the experiment successfully shows the ability of neural networks to calibrate two model parameters, albeit in a simplified setting. Moreover, it is clear that the training algorithm and optimisers in the Keras library

converge to a result faster than that does the gradient-descent algorithm implemented in the MATLAB code, as expected. Most importantly, consistency between the results for the two sets of code gives us the confidence going forward that the objective of implementing neural-network construction and training in code, based on the mathematics, has been successful. This understanding is translated to the Heston model experiments, in which only Python code and the Keras library are used for training. Finally, the results of the CEV experiment provide another indication as to the suitability of neural networks to calibration problems. We thus progress to the calibration of five parameters in the Heston experiment.

Chapter 4

Simulating Realistic Calibration Procedures with the Heston Model

4.1 Overview

In the previous two experiments, we considered simplified scenarios that did not realistically depict calibration as it would be carried out in practice. Specifically, the calibration in those experiments took place at only one point in time.

Here, we focus on neural-network calibration that resembles realistic calibration procedures. In this experiment, calibration takes place over time and is performed daily.

Define S_t and v_t to be the stock price and variance processes, respectively. Heston (1993) specifies the dynamics of these processes as

$$dS_t = \mu S_t\, dt + \sqrt{v_t} S_t\, dW_t^1 \qquad S_0 = s \text{ (constant)},$$

$$dv_t = \kappa(\theta - v_t)\, dt + \sigma\sqrt{v_t}\, dW_t^2 \qquad v_0 = v \text{ (constant)},$$

and

$$dW_t^1\, dW_t^2 = \rho\, dt,$$

where μ is the drift of the stock price process, κ, θ, and σ are, respectively, the rate of mean reversion, level of mean reversion, and volatility of volatility, and ρ is the correlation between the two Brownian motions, W_t^1 and W_t^2.

Specifying the stock price dynamics in this way means that the number of parameters to be calibrated is now greater. In order to price the European call options used in this experiment, a Fourier transformation technique is applied, using the *little trap* formulation of the Heston characteristic function from Albrecher *et al.* (2006).

As in the previous two experiments, we generate the data that is used in this calibration problem. To generate the out-of-sample stock price and variance paths, we fix the following parameter values for the Heston model: $S_0 = 100$, $v_0 = 0.06$, $\kappa = 3$, $\theta = 0.05$, $\sigma = 0.1$, $\rho = -0.4$. These values are based on an example given by McWalter (2017) to represent standard market conditions. The out-of-sample paths are generated using the Milstein approximation scheme given by Rouah (2013).

The paths that are generated span 91 days. In order to limit the amount of training to be performed, the maximum and minimum values of both the stock

Tab. 4.1: Values used for hyperparameter optimisation in the Heston experiment

Activation function	softplus	ReLU	ELU
No. of hidden layers	2	3	4
No. of neurons per hidden layer	16	32	64

price and variance paths are used to define the parameter value ranges used in training.

The training dataset is generated in a manner similar to that in the CEV experiment: ranges are defined for all of the Heston model parameters, and evenly-spaced points are sampled along each of these ranges. This produces a hypercube filled with points described by the different combinations of these parameter values. These points are then used to price a number of different call options, with varying strike prices and maturities. The *validation* and testing datasets are generated by taking random parameter values within the specified ranges and using these values to price the same call options as for the training dataset. The validation dataset is used by the training algorithm to avoid overfitting to the training dataset.

4.2 Hyperparameter optimisation

In order to carry out hyperparameter optimisation, 100000 training examples are generated for 15 different call options. We also introduce and define another activation function, the exponential linear unit (ELU), as

$$\mathrm{ELU}(x;\alpha) := \begin{cases} x & x \geq 0 \\ \alpha(e^x - 1) & \text{otherwise,} \end{cases}$$

and limit ourselves to the case where $\alpha = 1$, since this case alone meets the differentiability requirements of Theorem 2.2 (Barron, 2017).

Hyperparameter optimisation is performed as in Section 3.2, via a grid search over the values given in Table 4.1. Training takes place over 500 epochs using the Adam optimiser and the mean squared error cost function. The neural network consisting of 4 hidden layers and 32 neurons per hidden layer, with the ELU activation function applied in the hidden layers, is selected. When applied to the out-of-sample data, the performance of this neural network calibration is neither particularly good nor poor: the implied volatility values derived from the calibrated parameter values differ up to 6% from the true implied volatility values. However, it is clear that this result can be improved by modifying the training procedure and focusing on the selected neural network.

4.3 Improved training and Heston model calibration

The neural network selected by the hyperparameter optimisation is retained and subjected to a more focused training procedure. The following ranges of the Heston

model parameters are used to generate datasets:

$$\begin{aligned} 95 \leq & S_t \leq 115 \\ 0.05 \leq & v_t \leq 0.08 \\ 2 \leq & \kappa \leq 4 \\ 0.02 \leq & \theta \leq 0.08 \\ 0.05 \leq & \sigma \leq 0.3 \\ -0.5 \leq & \rho \leq -0.3. \end{aligned}$$

In addition, a greater number of call options is used than before, with maturities (in years)

$$T = (0.5, 1, 1.5, 2)^\top,$$

and strike prices

$$K = (80, 85, 90, 95, 100, 105, 110, 115, 120)^\top.$$

Owing to limited computational resources, the resulting hypercube of parameter values is kept relatively small. Nonetheless, 302500 training examples are generated. Each training example includes a vector of 36 option prices and the stock price. An additional 45000 examples are generated for each of the validation and testing datasets. The testing set is used to perform hyperparameter optimisation on a narrower scope, using the Adam optimiser and focusing on two different cost functions. The first is the the MSE cost function used previously. The second is the mean absolute percentage error (MAPE), defined as

$$\text{MAPE}(\underline{\phi}, \hat{\underline{\phi}}; \underline{w}) := \frac{1}{M} \sum_{i=1}^{M} \left[\frac{100}{K} \sum_{k=1}^{K} \left| \frac{\phi_i^{(k)} - \hat{\phi}_i^{(k)}}{\phi_i^{(k)}} \right| \right],$$

where $\underline{\phi}^{(k)}$ and $\hat{\underline{\phi}}^{(k)}$ are the vectors of true and calibrated parameter values, respectively, and M is the number of parameters being calibrated. In the case of the Heston model,

$$\begin{aligned} \underline{\phi}^{(k)} &= (v_t^{(k)}, \kappa^{(k)}, \theta^{(k)}, \sigma^{(k)}, \rho^{(k)})^\top, \\ \hat{\underline{\phi}}^{(k)} &= (\hat{v}_t{}^{(k)}, \hat{\kappa}^{(k)}, \hat{\theta}^{(k)}, \hat{\sigma}^{(k)}, \hat{\rho}^{(k)})^\top, \end{aligned}$$

and hence

$$M = 5.$$

The MAPE cost function is chosen so that the cost-function minimisation in the neural-network training does not focus unduly on the parameters with higher absolute values. Instead, it ensures that the relative differences between the true parameter values and those calibrated by the neural network are minimised across all parameters. In the context of this calibration problem, we find that the MAPE cost function is better for training neural networks.

Furthermore, the neural network selected from the preceding hyperparameter optimisation is compared to one with 64 neurons in each hidden layer as a check

that the former neural network remains optimal for this calibration problem. This is confirmed by the additional training.

To measure the performance of neural-network calibration, we define the mean relative implied volatility error (MRIVE) as

$$\text{MRIVE}(\underline{V}) := \frac{1}{N}\sum_{i=1}^{N}\frac{IV_i^{mkt} - IV_i^{\mathcal{N}}}{IV_i^{mkt}},$$

where IV_i^{mkt} is the implied volatility derived from the (out-of-sample) market price of the i^{th} option, and $IV_i^{\mathcal{N}}$ is the implied volatility derived from repricing the i^{th} option using the calibrated parameter values, as before.

The main result from this experiment is depicted in Figure 4.1. The mean relative implied volatility error, taken over the 36 call options, is less than 0.9% throughout the 91-day out-of-sample period. In addition, examples of the error surfaces on several different days are shown in Figure 4.2. These results clearly show an improvement in training over the initial hyperparameter optimisation.

For comparison, we calibrate using a numerical optimiser. This latter calibration is highly accurate, as expected, with the mean relative implied volatility error being of the order 10^{-5}. Clearly, on the basis of error alone, traditional numerical calibration still performs better than the neural-network calibration shown here.

We note, however, that numerical calibration takes significantly longer than neural-network calibration. For this experiment, training a neural network takes approximately 60 minutes, which corresponds with the initial *offline* phase of neural-network calibration. However, once this training is completed, the neural-network calibration takes less than a second to calibrate over the entire out-of-sample time series. In contrast, the numerical calibration takes approximately 60 seconds to calibrate the parameters on each day of the out-of-sample period, totalling approximately 90 minutes to calibrate the entire time series.

It is worthwhile considering how these two calibration methods would perform and compare in practice, where we would expect to have a greater number of derivative instruments on each underlying, and possibly a greater number of parameters to calibrate, depending on the models being used.

As the numbers of instruments and parameters increase, the neural-network training time (in the *offline* phase) would increase significantly. Hernandez (2016) states that training could take between several hours and several days in practice. To put this into context, however, Hernandez (2016) recommends that the *offline* training phase is conducted once every few months, as stated before. Once training is complete, the neural-network calibration would still take place very quickly. This is because calibrating a greater number of parameters, and using a greater number of instruments for calibration, merely increases the sizes of the matrices that are multiplied during neural-network computation. Again, as Hernandez (2016) states, this computation can be performed very quickly on most computers, and so the effect on neural-network calibration time would likely be marginal.

Numerical calibration, however, would take significantly longer every time calibration is performed. As the number of instruments used in calibration increases, numerical calibration takes longer to minimise the cost function, since this

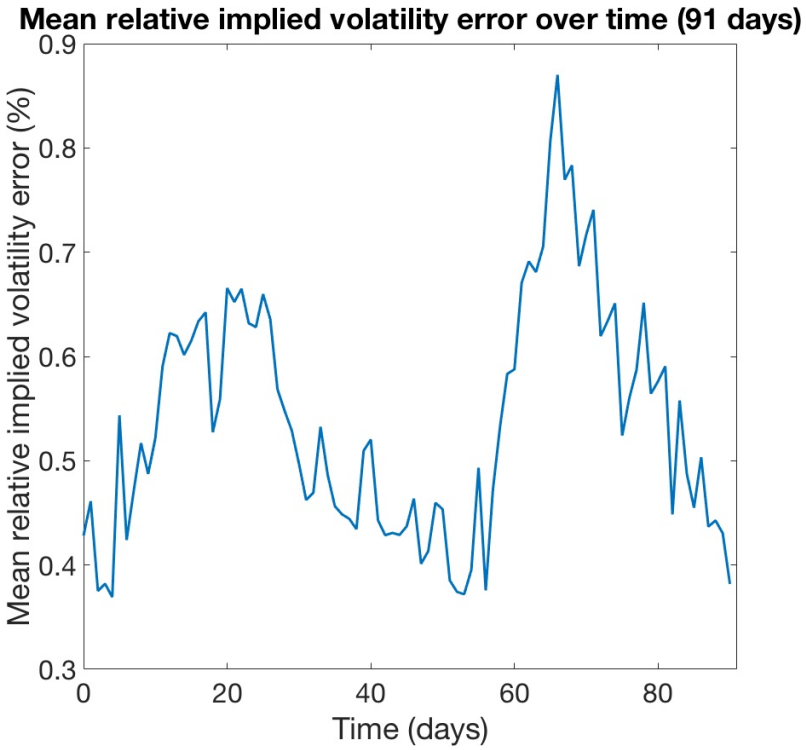

Fig. 4.1: Mean relative implied volatility error for Heston calibration with a neural network

is a function of all of the instrument prices (Hernandez, 2016). Furthermore, numerical calibration involves finding the vector of parameter values that minimises the cost function. This implies that the optimisation of the cost function is multi-dimensional. As such, calibrating a greater number of parameters significantly increases the calibration time. This again illustrates why numerical calibration may be impractical for certain classes of models, and why we are interested in using neural networks for calibration.

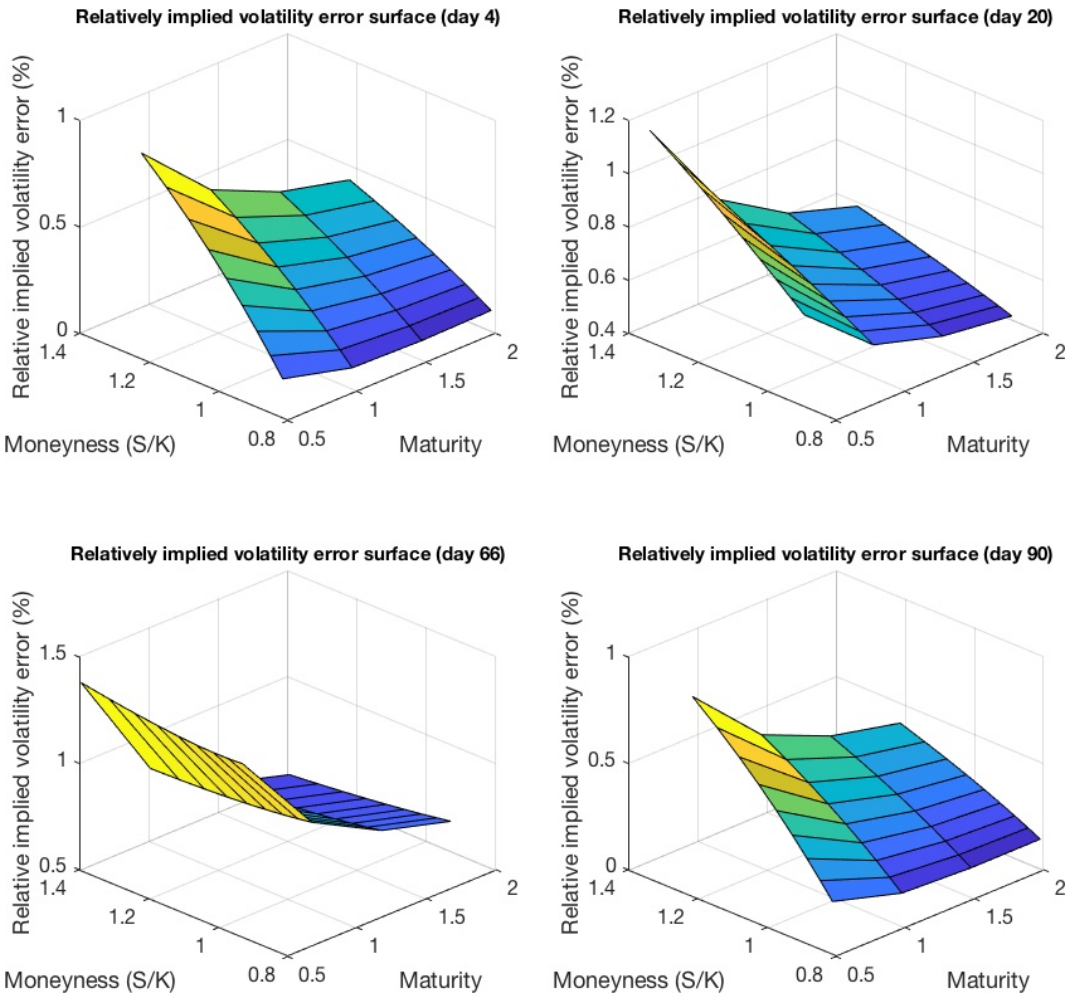

Fig. 4.2: Relative implied volatility error surfaces for Heston calibration with a neural network

Chapter 5

Exploring the Limitations of Neural-Network Calibration

5.1 Overview

In the previous chapter, we demonstrated neural-network calibration for the Heston model, using out-of-sample model-generated data. To reiterate, the mean implied volatility error (that is, taken over all of the calibration instruments) was less than 0.9%. However, the experiment gave no obvious indications regarding the limitations of neural-network calibration.

Hernandez (2016) and Mavuso *et al.* (2017) briefly discuss such limitations, but these discussions pertain mostly to the performance of neural-network calibration over time, that is, several months after training. Based on his experiment, Hernandez (2016) notes that there is a significant degradation in calibration performance after 6 months to 1 year since that neural network was trained. As a solution, he proposes retraining neural networks every 2-3 months. Here we are interested in the other potential shortcomings and limitations of neural networks in calibration. In particular, we use the Heston model to study how neural-network calibration performs when model assumptions are violated, and additionally look at calibration in instances where model parameters move out of the ranges for which the neural network has been trained. We will refer to the bounds of these ranges as the *neural-network training bounds*.

Under the Heston model, the parameters $(\kappa, \theta, \sigma, \rho)^\top$ are assumed constant. Here we look at scenarios in which two of these parameters, the correlation, ρ, and volatility of volatility, σ, change over time, first individually and then together. We consider these two parameters because the calibration of correlation is often difficult or problematic in practice, while varying the volatility of volatility may give us an indication as to how neural-network calibration performs in uncertain or changing market conditions.

In addition, we are interested in seeing how these particular parameters are calibrated relative to their true, underlying values. More importantly, we want to see whether the overall performance of neural-network calibration, measured by the mean relative implied volatility error, is comparable to that shown in Chapter 4.

We begin with an experiment in which parameter values are kept constant,

however, and specifically look at examples of bear market scenarios in which the stock price alone moves below its training bounds. The aim of this experiment is to see how much of an effect this movement out of the training bounds has on neural-network calibration.

Thereafter, we consider scenarios in which the true values of the model parameters vary and remain within their neural-network training bounds. This is done to isolate and study the effect of the violation of the model assumptions, so that these results are not conflated with the effects of trying to calibrate in cases where the true parameter values exceed the training bounds.

Finally, we extend the above-mentioned experiments involving ρ and σ to scenarios in which the true values of these parameters also exceed the neural-network training bounds.

To review, we investigate the following:

1. Calibration where the stock price moves out of the neural-network training bounds
2. Calibration where Heston model parameter values vary within the neural-network training bounds:
   (a) Correlation varies within bounds
   (b) Volatility of volatility varies within bounds
   (c) Correlation and volatility of volatility vary simultaneously within bounds
3. Calibration where Heston model parameter values vary and exceed the neural-network training bounds:
   (a) Correlation varies and exceeds bounds
   (b) Volatility of volatility varies and exceeds bounds
   (c) Correlation and volatility of volatility simultaneously and exceed bounds

Apart from the features described above, these experiments are set up in the same way, and use the same values, as described in Chapter 4. The neural network used in all of these experiments is the same one that was trained and used in the preceding experiment.

5.2 Stock price movements out of the neural-network training bounds

We are first interested in seeing how neural-network calibration performs when the stock price moves out of the bounds for which the neural network has been trained. The motivation for this is twofold. Firstly, we want to determine whether it is necessary to retrain the neural network for calibration in bear market scenarios, which include abrupt decreases in a given stock price. Secondly, one of the concerns for the experiments following this one is that any movement of the stock

Fig. 5.1: First bear market scenario – stock price path

price out of the neural-network training bounds will distort the results and thus the interpretation of results.

In generating the two stock price paths studied here, we retain the Heston model parameter values used in Chapter 4. Along the first sample path, shown in Figure 5.1, the stock price drops below the neural-network training lower bound value of 95 on just over half of the days in the out-of-sample period. The corresponding mean relative implied volatility error is shown in Figure 5.2. It is clear that the neural-network calibration performs poorly in this experiment, with the error reaching nearly 20% at one point. Considering both of these figures, there appears to be a relationship between the error and the stock price when the latter falls below the training bound. Specifically, the error seems to be greater the further the stock price falls below the bound. As such, we compare the stock price (relative to this lower bound) to the corresponding error. This comparison is shown in Figure 5.3. From the plot, we see what appears to be an exponential relationship between the two aforementioned quantities.

The above procedure is repeated for a second sample path, shown in Figure 5.4, over which the stock price is below the relevant lower bound for 71 out of the 91 days. In this case, the error (Figure 5.5) is even greater than for the first scenario, reaching and remaining close to 45% over much of the out-of-sample period. Again, there appears to be a relationship between the extent to which the stock

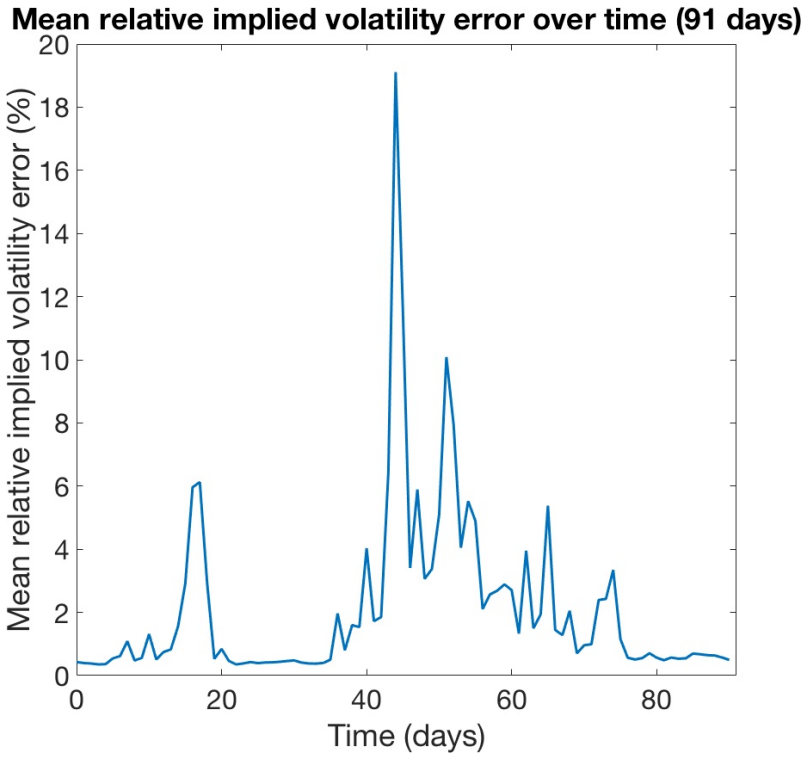

Fig. 5.2: First bear market scenario – mean relative implied volatility error

price drops below the lower bound and the magnitude of the error. For this sample path, the relationship is depicted in Figure 5.6. Although there are some outlying points, this relationship largely appears to be exponential, as it does for the first bear market scenario.

From this experiment, it is clear that even for fixed parameter values, that is, even if the stock price dynamics are perfectly described by the model, neural-network calibration is severely and negatively impacted if the stock price moves out of the neural-network training bounds. The implication for the use of neural-network calibration in practice is that market conditions need to be monitored and checked against the last round of neural-network training and, in particular, the training data. If a critical point is reached, such as a stock price being near one of the bounds, the neural networks can be appropriately retrained in line with the recommendation by Hernandez (2016).

For the remaining experiments, therefore, we only study paths where the stock price remains within its training bounds, in order to isolate the effect of what we are investigating, namely, changing parameter values.

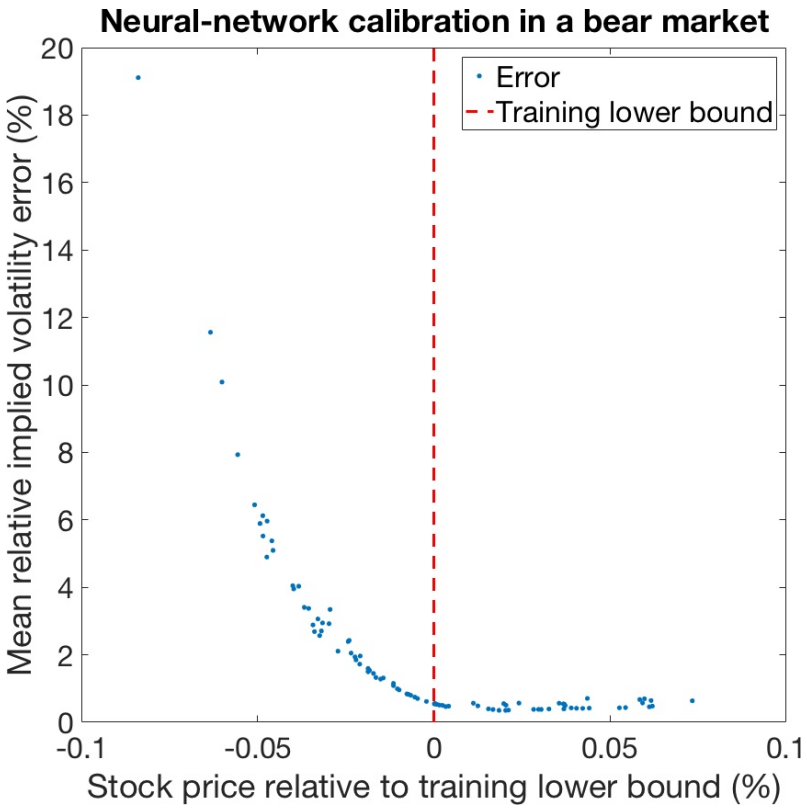

Fig. 5.3: First bear market scenario – comparison of stock price and error

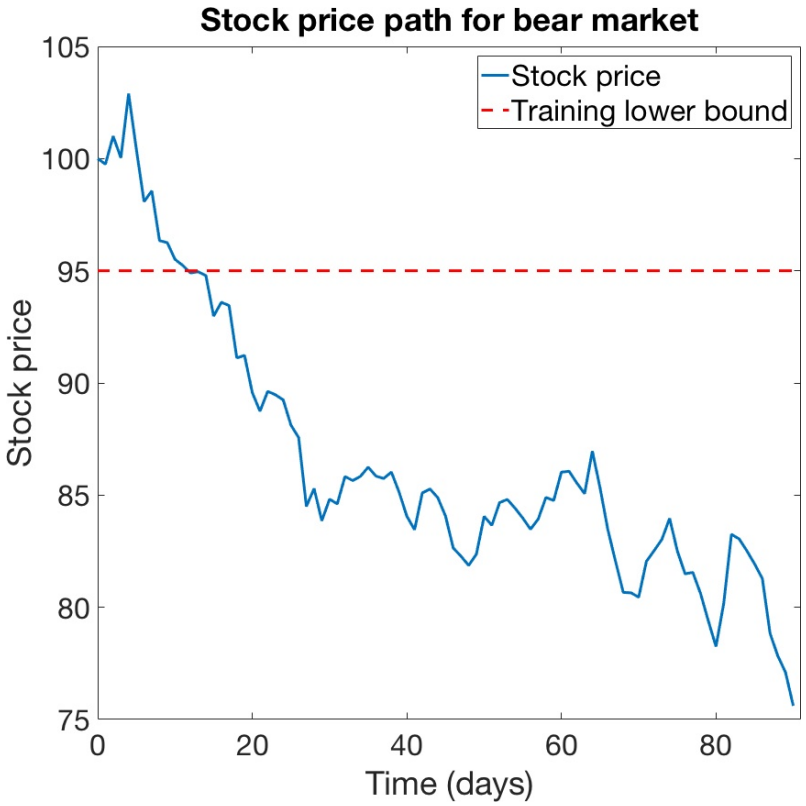

Fig. 5.4: Second bear market scenario – stock price path

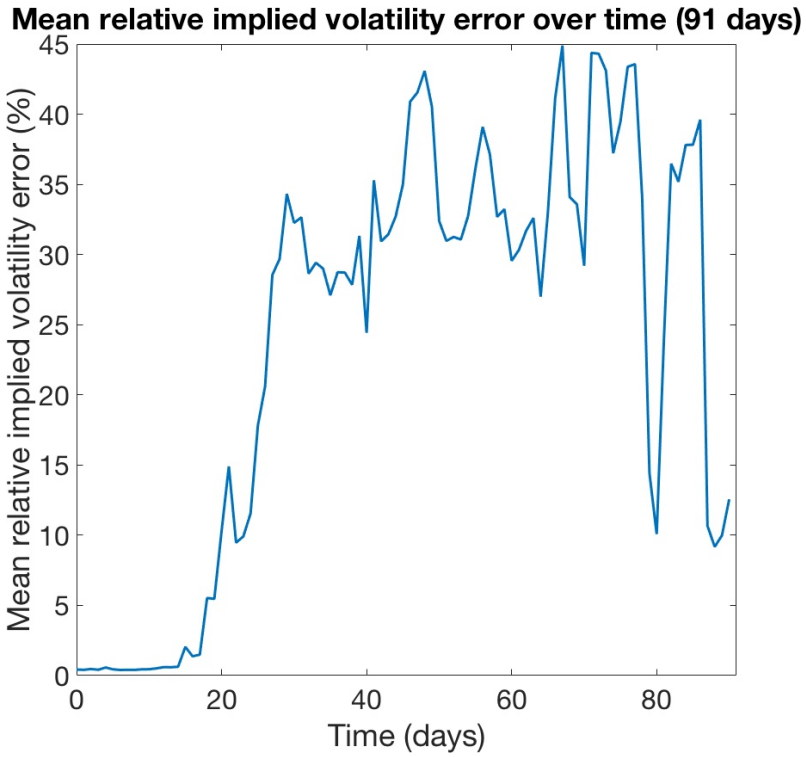

Fig. 5.5: Second bear market scenario – mean relative implied volatility error

5.3 Violation of model assumptions within neural-network training bounds

In the first of this group of experiments, we hold all parameter values constant except for the correlation parameter value. The initial correlation value is set at -0.3, the upper bound of the range of correlation values for which the neural network has been trained, and decrease its value by 0.025 every 10 days when generating the out-of-sample data. Over the last 10 days, its value is -0.5, which is the lower of the neural-network training bounds for correlation.

From Figure 5.7 (left), we see that the calibrated correlation values track the true values relatively closely over most of the 10-day periods, with the exception of days 61-70, over which the calibrated correlation values deviate the most from the true values. Moreover, the calibrated values during this shorter period are greater than the true values, unlike the rest of the out-of-sample period, for which correlation is under-estimated. Additionally, the mean relative implied volatility error for this experiment is less than or equal to 1.01% over the entire period. This is similar to the corresponding number for the experiment described in the previous chapter.

In the next experiment, we vary the value of the volatility of volatility parameter, rather than the correlation parameter value. We use the lower of the neural-network training bounds for this parameter, 0.05, as the initial value, and increase the parameter value by 0.0025 every 10 days until it reaches 0.25 during the last 10

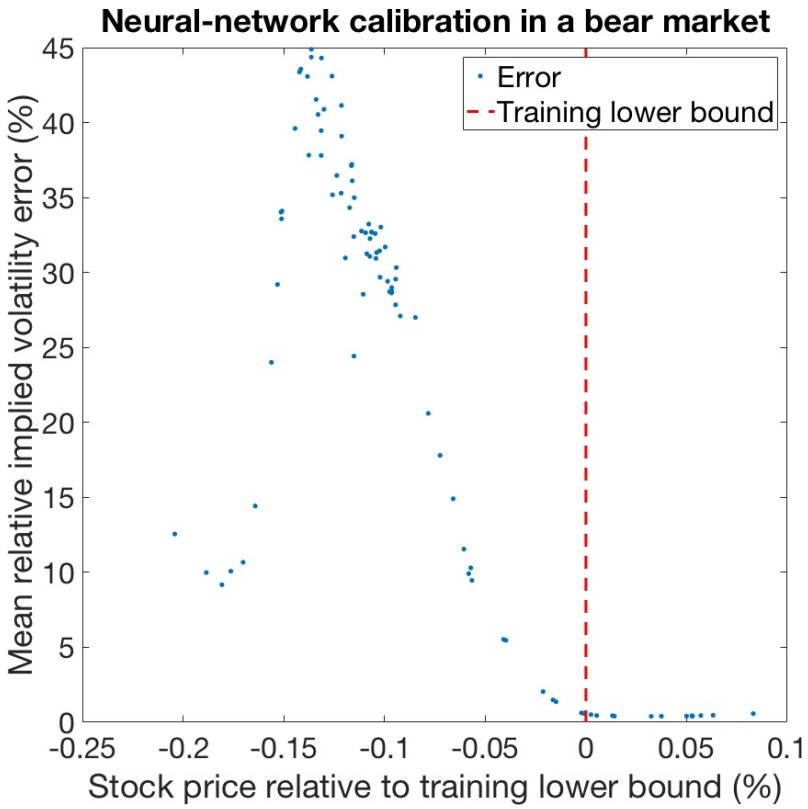

Fig. 5.6: Second bear market scenario – comparison of stock price and error

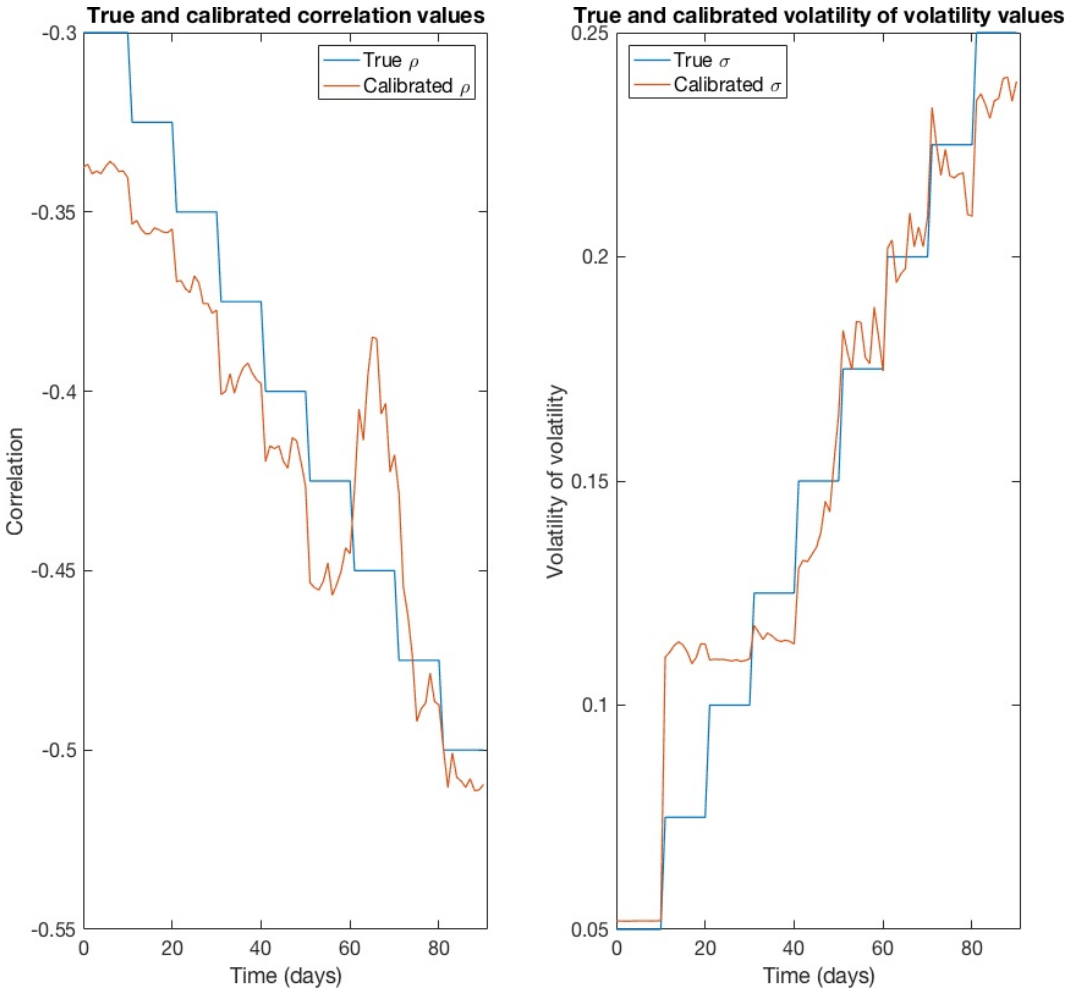

Fig. 5.7: Calibrated correlation and volatility of volatility values – one parameter varies within training bounds

days of the experiment.

As in the correlation-based experiment, we see in Figure 5.7 (right) that the calibrated volatility of volatility values track the true values relatively closely, with the exception of the period over days 11-20. Again, there is similarity with the preceding experiment in that the error for the out-of-sample period is less than or equal to 1.05% and thus comparable to that in the original Heston experiment (Chapter 4).

In the last of this group of experiments, we combine the two preceding experiments, varying correlation and volatility of volatility simultaneously in the manner described above. The results are shown in Figure 5.8. The calibrated volatility of volatility values (right) lie relatively closely to the true values as in the prior experiment. In contrast, however, there is clearly a large discrepancy between the true and calibrated correlation values (left) over most of the out-of-sample period,

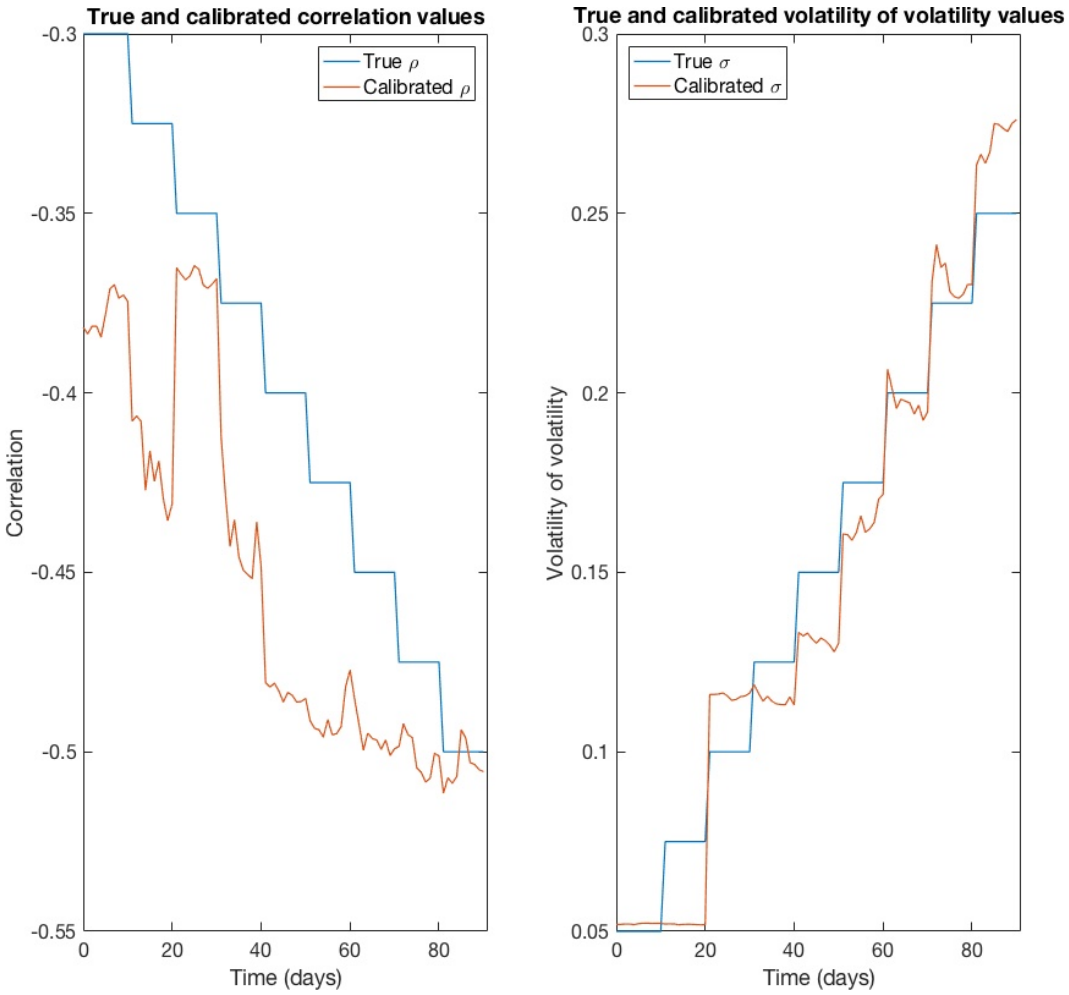

Fig. 5.8: Calibrated correlation and volatility of volatility values – both parameters vary within training bounds

unlike in the first experiment. Nonetheless, in the case where the values of both parameters vary within their training bounds, the calibration performance of the neural network is sill comparable to that shown in Chapter 4, with a mean relative implied volatility error of less than 1% over the out-of-sample period.

Our expectation for these three experiments was that, despite model assumptions being violated, the neural network would calibrate relatively well. This expectation is based on an understanding of the trained neural network as an interpolation tool, capable of interpolating between points within a defined hypercube. The results of these experiments are in line with our expectations, with the neural-network calibration performing similarly to that in the original Heston model experiment.

5.4 Violation of model assumptions outside of neural-network training bounds

In order to set up the first in this group of experiments, we specify correlation values that change every 10 days, as in Section 5.3. However, these values alternate between lying within and outside of the neural-network training bounds for successive 10-day periods. The true and calibrated correlation values are shown in Figure 5.9 (left). When the true correlation values lie within the neural-network training bounds, the neural-network calibration estimates correlation relatively well, with the possible exception of the period between 21-30 days. What this shows is that even if the parameter values exceed the training bounds during some periods, the neural network is still able to calibrate the parameter values relatively well in subsequent periods where the true values lie within the training bounds, as we would expect.

We also see in Figure 5.9 (right) that the mean relative implied volatility error remains below 2.5%. While this error is not nearly as large (in magnitude) as the value of the corresponding error for the experiments described in Section 5.2, for example, it is nonetheless large relative to the error values calculated for the original Heston model experiment (Chapter 4), as well as for the experiments in the previous section. Lastly, it is interesting to note that in this experiment, the error for the periods where correlation falls below its training lower bound is less than that for the periods where correlation exceeds its training upper bound, although it is not immediately clear why this is the case here, nor whether this is a phenomenon that would persist over multiple iterations of the experiment.

For the next experiment, we specify volatility of volatility values that vary similarly to how the correlation values vary in the preceding experiment. The true and calibrated values of this parameter are shown in Figure 5.10 (left). As for the related correlation experiment, the calibrated parameter values for volatility of volatility are relatively close to the true values when the latter are within the training bounds. Figure 5.10 (right) shows that the mean relative implied volatility error remains below 2% over the out-of-sample period, which is slightly lower than for the correlation experiment. Again, this value is large relative to the errors calculated in some of the previous experiments.

For the last of these experiments, we combine the preceding correlation and volatility of volatility experiments. The true and calibrated parameters are shown in Figure 5.11. These results are similar to those for the individual correlation and volatility of volatility experiments, except that each series of calibrated parameter values contains significantly under- and over-estimated correlation and volatility of volatility values, respectively, between days 31-40. In particular, correlation is calibrated to values smaller than -1, which contradicts the definition of correlation. The associated error is shown in Figure 5.12. For the periods where the two parameter values exceed their respective training bounds, the magnitude of the error is large, especially between days 31-40.

All of the experiments described in this section reveal a significant degradation in calibration performance in even the simplest cases where only one parameter's true value exceeds the neural-network training bounds. This degradation is worse

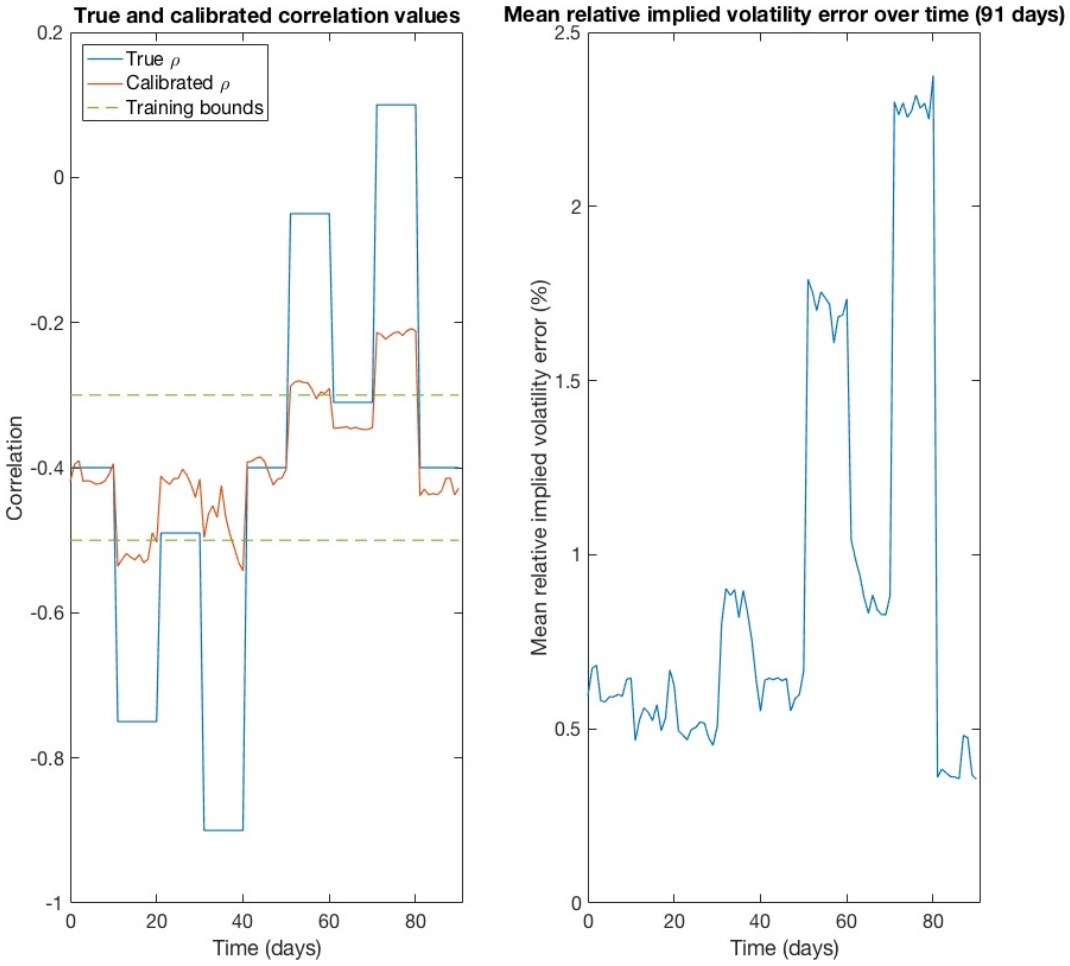

Fig. 5.9: Calibrated correlation values and error – one parameter varies, exceeds training bounds

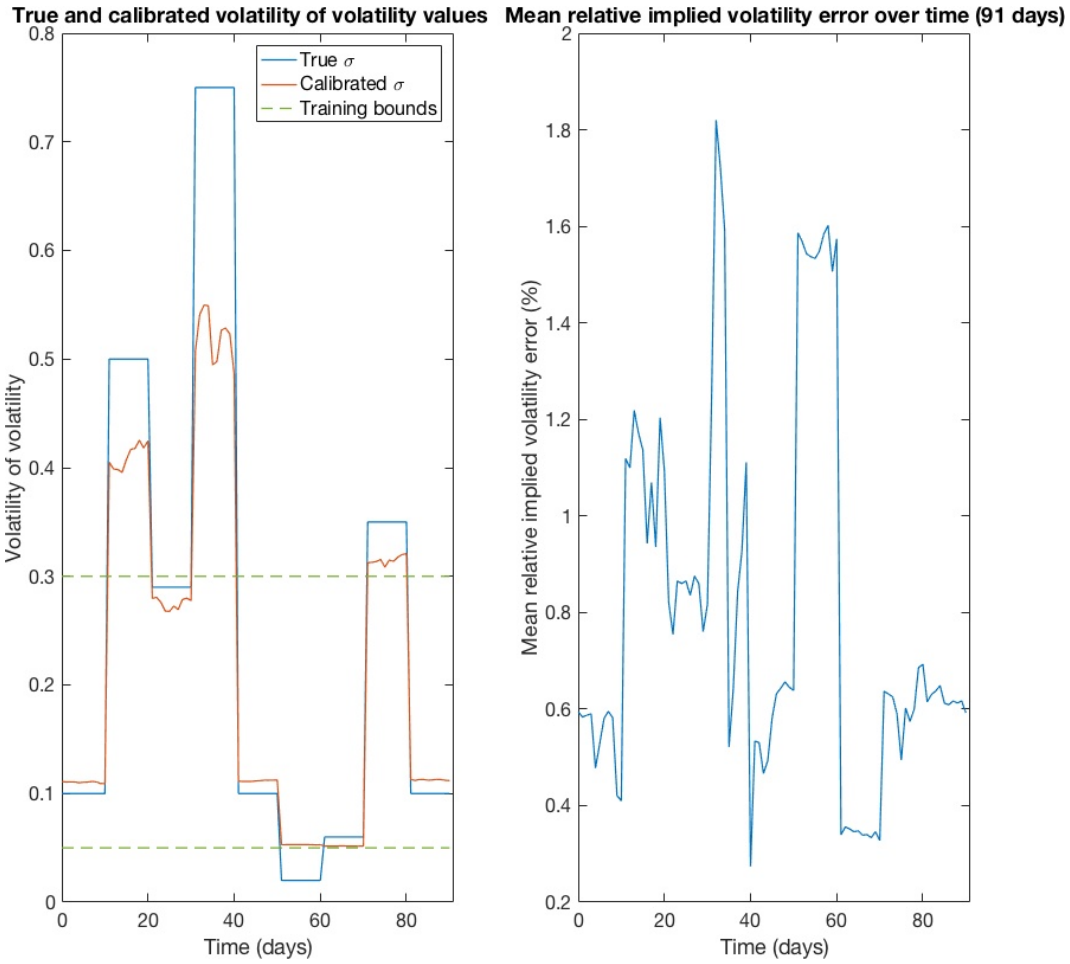

Fig. 5.10: Calibrated volatility of volatility values and error – one parameter varies, exceeds training bounds

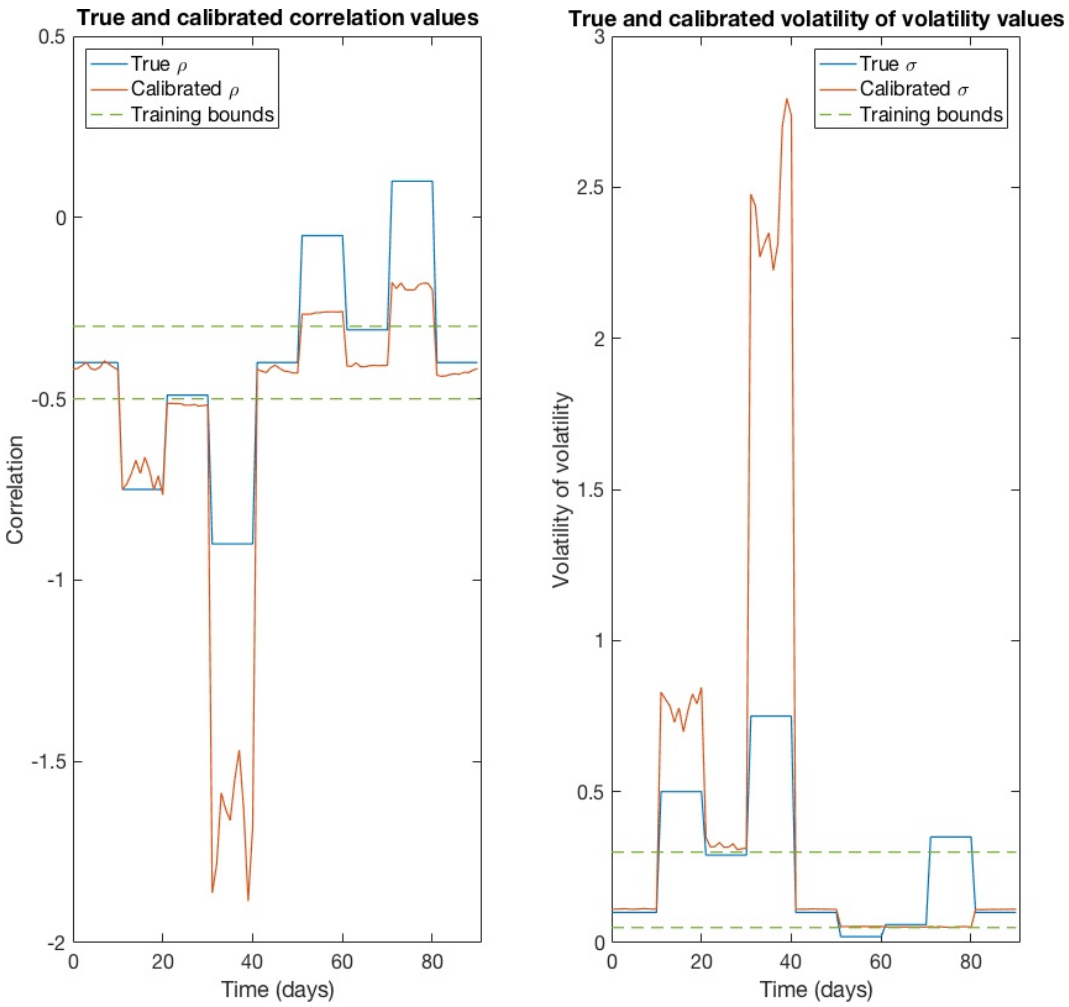

Fig. 5.11: Calibrated correlation and volatility of volatility values – both parameters vary, exceed training bounds

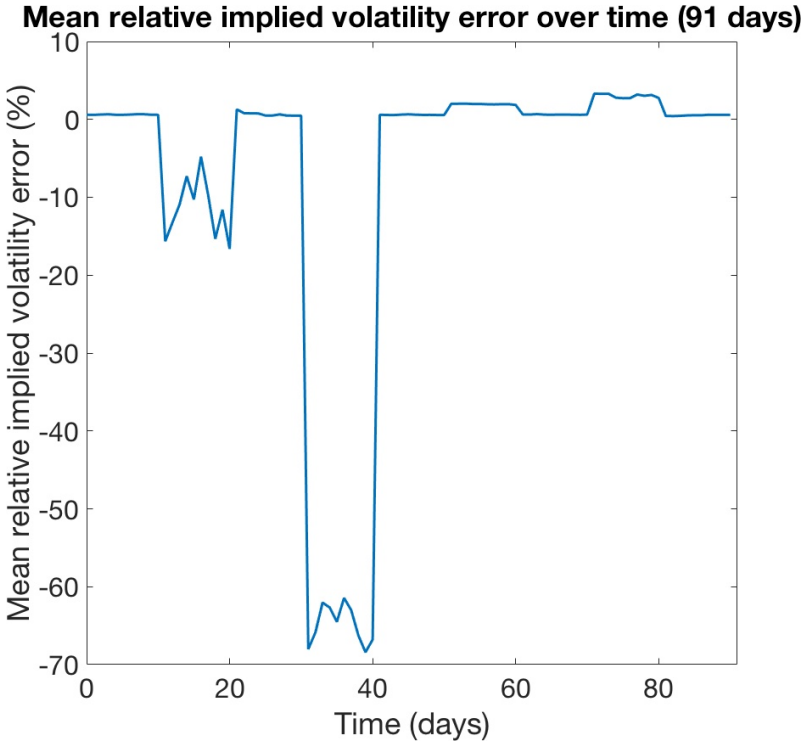

Fig. 5.12: Both parameters vary, exceed training bounds – mean relative implied volatility error

when two parameters vary in this manner. Based on the same hypothesis and reasoning outlined in Section 5.3, this is in line with our expectations for these experiments: the calibrating neural network is, in a sense, trained to interpolate within a hypercube, but it is not necessarily capable of extrapolating out of that hypercube.

However, despite the extent of performance degradation in these experiments, we note that neural-network calibration does not fail catastrophically. The parameter values produced by neural-network calibration track the movements of the true parameter values when the latter move out of their training bounds. In some instances, the calibrated parameter values exceed the training bounds. In other words, when the true parameter values exceed the upper training bounds, the calibrated parameter values approach or exceed the upper training bounds as well, with the same holding for the lower training bounds. Although there are large differences between the true and calibrated parameter values when the training bounds are exceeded, the neural-network calibration does not produce any oscillating parameter values, or parameter values that move in a direction opposite to that of the true values, for example.

These experiments highlight the importance of having an adequate set of training data that spans sufficiently broad ranges of parameter values. This also provides an indication as to why the method specified by Hernandez (2016), and also utilised by Mavuso *et al.* (2017), to derive training datasets produces neural networks that perform calibration to market data well. The dataset used by Hernandez (2016) consists, firstly, of historical market data and numerically-calibrated parameter values that correspond with a given model. This data effectively specifies the stock price and model parameter ranges within which calibration will likely take place in future, which thus determines the neural-network training bounds. Furthermore, this method consists of using the empirical statistical properties of the data, such as the correlations between parameter values, to generate additional training examples. This is akin to filling the aforementioned hypercube with more points, which supports more thorough and robust neural-network training.

Chapter 6

Conclusion

In this dissertation, we explored multiple facets of feedforward neural networks and the application of neural networks to the calibration of financial models. This began with the specification of the mathematics and algorithm needed to calculate a forward pass of a feedforward neural network. In addition, we outlined the mathematics underlying the gradient-descent training algorithm.

This theoretical content was then used to code the neural networks utilised in two subsequent experiments pertaining to the Black-Scholes and constant elasticity of variance models. Through these experiments, we demonstrated the hyperparameter optimisation of neural networks. We also showed the use of different optimisation procedures in the training process.

Furthermore, these experiments provided further indication that neural networks, as a calibration tool, can be used effectively to map the (non-linear) relationships between market prices and model parameter values for a given financial model.

Another experiment, based on the Heston model, demonstrated neural-network calibration in a manner similar to real-world calibration, with calibration taking place and analysed over a series of days. For this experiment, the Keras library was used to train neural networks (Chollet, 2015).

A hypercube of parameter values was defined and used to generate training data. We defined 36 call options, varying in maturity and strike price, as the instruments used in calibration. With an out-of-sample time horizon of 91 days, a stock price path and corresponding option prices were generated to test the neural-network calibration.

The mean relative implied volatility error associated with the neural-network calibration of the Heston model remained below 0.9% throughout the out-of-sample period. While the corresponding error for numerical calibration was much lower (that is, in the region of thousandths of one per cent), we noted that that neural-network calibration for the entire time series took less than a second, while the numerical calibration took approximately 90 minutes. This time advantage points to the potential usefulness of neural networks for calibration problems where traditional numerical techniques take too long to be of practical value.

Another significant component of this work was the investigation into the shortcomings and limitations of neural-network calibration. We provided some evidence to suggest that, in instances where the assumptions underlying financial models are violated (such as where parameters that are assumed constant are not),

the performance of neural-network calibration is comparable to that in an idealistic scenario in which model assumptions are not violated. One caveat to this, however, is that the neural network must be adequately trained to handle the ranges of stock prices and parameter values that are encountered.

Conversely, we showed examples of situations in which neural-network calibration clearly performed poorly, to varying degrees. These include stock price movements out of the neural-network training bounds, as well as movements in parameter values beyond their training bounds. As stated before, this illustrates the importance of adequate training, in terms of the size and scope of training datasets.

Moreover, neural networks are evidently not a perfect solution to calibration problems. Hernandez (2016) suggests that auxiliary systems may need to be used to systematically monitor the performance of neural-network calibration, such as parallel numerical calibration that is run periodically as a check. In addition, Hernandez (2016) recommends retraining neural networks every few months.

While this dissertation explored multiple aspects of neural-network calibration, it is obvious that there is much more research that can be conducted on the application of neural networks to calibration. Firstly, from the hyperparameter optimisation carried out by Hernandez (2016), Mavuso *et al.* (2017), and in this work, it is not immediately clear what constitutes a good neural-network architecture for calibration problems. Another important question that arises is whether feedforward neural networks are the best for such problems.

Hernandez (2016) briefly discusses an attempted use of convolutional neural networks for Hull-White calibration, but notes that these neural networks did not show significantly better performance than feedforward neural networks, despite taking much longer to train. It may be the case, however, that convolutional neural networks are better for different types of calibration problems, and this should be investigated.

Furthermore, while it has provided some insights, our investigation of the shortcomings and limitations of neural networks has been limited. Particularly with the practical implementation and use of neural-network calibration in mind, these limitations need to be investigated and tested more thoroughly. Finally, for completeness, it will be valuable to investigate the performance of hedging strategies derived from neural-network calibration and to compare these to hedging based on traditional, numerical calibration techniques.

Bibliography

Albrecher, H., Mayer, P., Schoutens, W. and Tistaert, J. (2006). The little heston trap. KU Leuven Section of Statistics Technical Report.

Barron, J. T. (2017). Continuously differentiable exponential linear units, *arXiv preprint arXiv:1704.07483* .

Black, F. and Scholes, M. (1973). The pricing of options and corporate liabilities, *Journal of political economy* **81**(3): 637–654.

Brigo, D. and Mercurio, F. (2006). *Interest rate models-theory and practice: with smile, inflation and credit*, Springer Science & Business Media.

Chollet, F. (2015). Keras, https://github.com/keras-team/keras.

Cox, J. (1975). Note on option pricing 1: constant elasticity of diffusion. Unpublished note.

Cybenko, G. (1989). Approximation by superpositions of a sigmoidal function, *Mathematics of control, signals and systems* **2**(4): 303–314.

Haykin, S. (1994). *Neural networks: a comprehensive foundation*, Prentice Hall PTR.

Hernandez, A. (2016). Model calibration with neural networks, *SSRN* . Accessed: 2017.08.14.

Heston, S. L. (1993). A closed-form solution for options with stochastic volatility with applications to bond and currency options, *The review of financial studies* **6**(2): 327–343.

Hull, J. and White, A. (1993). One-factor interest-rate models and the valuation of interest-rate derivative securities, *Journal of financial and quantitative analysis* **28**(2): 235–254.

Kingma, D. P. and Ba, J. (2014). Adam: A method for stochastic optimization, *arXiv preprint arXiv:1412.6980* .

Mavuso, M., Marr, A., Mitoulis, N. and Singh, A. (2017). Model calibration with neural networks. ACQuFRR.

McWalter, T. (2017). Local volatility. Lecture notes for Numerical Methods in Finance (unpublished).

Merton, R. C. (1973). Theory of rational option pricing, *The Bell Journal of economics and management science* pp. 141–183.

Ng, A. (2013). Machine learning. Coursera [MOOC].

Nielsen, N. A. (2015). *Neural Networks and Deep Learning*, Determination Press.

Rouah, F. D. (2013). *The Heston Model and Its Extensions in Matlab and C*, John Wiley & Sons.

Ruder, S. (2016). An overview of gradient descent optimization algorithms, *arXiv preprint arXiv:1609.04747* .

Schroder, M. (1989). Computing the constant elasticity of variance option pricing formula, *the Journal of Finance* **44**(1): 211–219.

Appendix A

Formulae

A.1 Black-Scholes analytical call option pricing formula

$$V(S_t, K, T, r; \sigma) = S_t\, N(d_1) - Ke^{-r(T-t)}\, N(d_2),$$
$$d_1 = \frac{1}{\sigma\sqrt{T-t}}\left[\ln\left(\frac{S_t}{K}\right) + \left(r + \frac{\sigma^2}{2}\right)(T-t)\right],$$

and

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

where K is the strike price, T is the maturity of the option, r is the risk-free rate of interest, σ is the implied volatility, and N denotes the Gaussian cumulative distribution function.

A.2 CEV analytical call option pricing formula

$$V(S_t, K, T, r; \alpha, \sigma) = S_t\,[1 - \chi^2(y; z, x)] - Ke^{-r(T-t)}\,\chi^2(x; z-2, y),$$

where K, T, and r are the strike price, maturity, and risk-free rate of interest, respectively. The function denoted $\chi^2(\cdot; n, \lambda)$ is the cumulative distribution function of the non-central chi-square distribution with n degrees of freedom and non-centrality parameter λ, while x, y, and z are given by

$$x = \kappa S_t^{2(1-\alpha)} e^{2r(1-\alpha)(T-t)},$$
$$y = \kappa K^{2(1-\alpha)},$$
$$z = 2 + \frac{1}{1-\alpha},$$

and

$$\kappa = \frac{2r}{\sigma^2(1-\alpha)(e^{2r(1-\alpha)(T-t)} - 1)}.$$

Appendix B

Code

B.1 MATLAB code for neural-network construction and training

```
clear;
clc;
close all;

rng(0)

%% Get data and determine the number of training examples

load('data.mat');
epochs = 500;

X = prices;
Y = pars;

n_inst = size(X,1);
n_pars = size(Y,1);

K = size(X,2);

%% Setup NN architecture

%Set NN parameters (hyperparameters)
n_hidden = 4;
%Number of neurons per layer (excl. bias units)
n_neurons = 16;

n_layers = n_hidden + 2;

%Neuron output initialisation (a's)
a = cell(1,n_layers);
a{1} = X;

%Weights and bias weights initialisation: cell structure for matrices,
    randomised weights
e = 0.1;

w = cell(1,n_layers-1);
w{1} = e.*randn(n_neurons,n_inst+1);
```

```
38 w{end} = e.*randn(n_pars,n_neurons+1);
39
40 if n_hidden > 1
41     %Index range: for the _remaining_ weight matrices
42     for idx = 2:((n_layers-1)-1)
43         %Taking into account _from_ n_{l-1}+1 (i.e. incl. bias) -> _to_
               n_{l}
44         w{idx} = e.*randn(n_neurons,n_neurons+1);
45     end
46 end
47
48 %Setup function handles for activation functions
49 g =@(z) activation(z);
50 gprime =@(z) activationDerivative(z);
51
52 %Note that the derivative of go is simply 1 if go(x)=x
53 go =@(x) x;
54
55 %% Backpropogation algorithm
56
57 %Learning rate/parameter
58 alpha = 0.1;
59
60 %ct denotes the number of epochs
61 for ct = 1:epochs
62
63     %Initialisation for the partial derivatives
64     delta = cell(1,n_layers);
65     dEdw = cell(1,n_layers-1);
66
67     %Compute NN
68     [Yhat , a , z] = FNN(X,w,g,go);
69
70     %Calculate error (purely indicative)
71     Error = 0.5 .* mean(sum((Yhat-Y).^2));
72
73     %Compute partial derivatives for weights to the outer layer. Note:
           delta(n_layers) = dEdyhat
74     delta{n_layers} = (Yhat - Y) .* 1;
75     dEdw{n_layers-1} = 1./K .* (delta{n_layers} * a{n_layers-1}.') ;
76
77     %Compute partial derivatives for all of the weights in the rest of
           the
78     %NN
79     for l = (n_layers-1):-1:2
80
81        %Intermediate calculation to fix matrix sizes
82        J = w{l}.' * delta{l+1};
83        J = J(2:end,:);
84
85        delta{l} = gprime(z{l}) .* J;
86        dEdw{l-1} = 1./K .* (delta{l} * a{l-1}.');
87     end
88
89     %Use calculated partial derivatives to adjust weights
90     for l = n_layers:-1:2
```

```
        w{l-1} = w{l-1} - alpha .* dEdw{l-1};
    end

end

save('neuralNetwork.mat','w');
```

```
function [ Yhat, acell, zcell ] = FNN( X, wcell, g, go )

%g - activation function in hidden layers
%go - activation function used in output layer (can be same as g)

n_layers = size(wcell,2)+1;
a = cell(1,n_layers);
z = cell(1,n_layers);

a{1} = X;

for idx = 2:n_layers

    dim = size(a{idx-1},2);
    a{idx-1} = [ones(1,dim) ; a{idx-1}];

    z{idx} = wcell{idx-1} * a{idx-1};

    if idx == n_layers
        a{idx} = go(z{idx});
    else
        a{idx} = g(z{idx});
    end

end

Yhat = a{n_layers};
acell = a;
zcell = z;

end
```

B.2 Python code for neural-network construction and training using the Keras library

```
##ML libraries
import keras
from keras.models import Sequential
from keras.models import load_model
from keras.layers import Dense
import numpy
import h5py

numpy.random.seed(0)

##NN architecture and epochs ========================================

```

```
#hyperparameters
n_hidden = 4
n_layers = n_hidden + 2
n_neurons = 64

#epochs
n_ep = 500

#Data ================================================================
train_data = numpy.loadtxt("training_data.csv", delimiter=",")
valid_data = numpy.loadtxt("validation_data.csv", delimiter=",")
test_data = numpy.loadtxt("testing_data.csv", delimiter=",")

trainX = train_data[:,0:37]
trainY = train_data[:,37:42]

validX = valid_data[:,0:37]
validY = valid_data[:,37:42]

testX = test_data[:,0:37]
testY = test_data[:,37:42]

##Setup NN ============================================================
NN = Sequential()

#input layer and first hidden layer
NN.add(Dense(n_neurons, input_dim = 37, activation = 'elu'))

#hidden layers
for i in range(2,n_layers):
    NN.add(Dense(n_neurons, activation = 'elu'))

#output layer
NN.add(Dense(5))

##Training ============================================================

NN.compile(optimizer = 'adam', loss = 'mape')
NN.fit(trainX,trainY, epochs = n_ep, validation_data = (validX,validY),
    verbose = 0)

NN.evaluate(testX,testY, verbose = 0)

##Save NN =============================================================
NN.save('Heston_4_64_elu_mape_adam.h5')
```

B.3 Python code for neural-network calibration

```
#Libraries
import keras
from keras.models import Sequential
from keras.models import load_model
from keras.layers import Dense
import numpy
import h5py

```

```
9  experiment = 'A'
10
11 #Data and NN ============================================================
12 calib_input = numpy.loadtxt(experiment+'heston_oos_mkt_data.csv',
       delimiter=",")
13
14 NNcalib = load_model('Heston_4_32_elu_mape_adam.h5')
15
16 calib_pars = NNcalib.predict(calib_input)
17
18 #Save calibrated parameter values ==================================
19 numpy.savetxt(experiment+'NNcalib_pars.csv',calib_pars,delimiter=",")
```