Deep Calibration of Heston Model with Neural Networks

Candidate Number: 1081225

April 24, 2024

CONTENTS

1 Introduction

The calibration of model parameters is a common but critical task in financial practice. In particular, accurate parameter settings are important prerequisites for deducing fair prices for all financial derivatives based on the model. However, traditional calibration methods often involve computationally expensive algorithms, such as Monte Carlo simulations, which may significantly slow down the calibration process. Previous literature has proposed the use of neural networks to approximate the pricing components in calibration, also referred to as "deep calibration" [4, 15]. It is believed that such a method could lead to a satisfactory reduction in computational costs, thereby resulting in faster calibration procedures.

The purpose of this paper is to investigate the computational accuracy of this deep calibration method, and also to suggest further directions for possible improvements. To achieve this goal, we will apply the method to calibrate parameters in the Heston model, one of the most commonly used stochastic volatility models in option pricing. In Section 2, we will introduce some preliminary knowledge relevant to this paper. Sections 3 and 4 will briefly describe the principle of the "two-step" deep calibration, together with some specific implementation details, as well as the parameter settings used in the experiment. Finally, the numerical results will be presented in Section 5. The conclusion and possible directions for improvements will be given in Section 6. All codes in this paper are available here through this link https://github.com/abaaba337/Deep-Calibration.

2 Financial Preliminaries

2.1 The Black-Scholes Model

Proposed by Fischer Black, Myron Scholes [5], and Robert Merton [23] in 1973, the Black-Scholes-Merton model, or simply the Black-Scholes model, was one of the earliest and widely used mathematical models designed to price the theoretical value of an option contract. Given a target option with a current value $V(t)$, by assuming that the price of the underlying asset $S(t)$ paying no dividends satisfies the following geometric Brownian motion

$$\mathrm{d}S(t) = \mu S \ \mathrm{d}t + \sigma S \ \mathrm{d}B(t), \tag{2.1.1}$$

one can derive the famous Black-Scholes PDE

$$\frac{1}{2}\sigma^2 S^2 V_{SS} + rSV_S + V_t - rV = 0, \tag{2.1.2}$$

through a series of standard hedging arguments [5, 7, 26]. In the above equation, r is the risk-free rate of some riskless bonds, and σ is the constant volatility of the

underlying asset. For a European call option $V(t,S) := C^E(t,S)$ expired at T with a strike price of K, where the terminal and boundary conditions are specified as

$$C^E(T,S) = (S-K)^+ \quad \text{for} \quad t = T\ ; \tag{2.1.3}$$

$$C^E(t,0) = 0 \quad \text{for} \quad S = 0\ ; \tag{2.1.4}$$

$$C^E(t,S) \sim S \quad \text{for} \quad S \to \infty\ , \tag{2.1.5}$$

the problem can be further transformed into a well-posed parabolic boundary value problem involving the heat equation through changing of variables [7, 26]. Thus, the fair price for a European call option at time t whose underlying asset generates no dividends is given as

$$C^E(t,S) = S\Phi(d_+) - Ke^{-r\tau}\Phi(d_-), \tag{2.1.6}$$

where $\Phi(\cdot)$ is the cumulative distribution function of the standard normal distribution, $\tau = T - t$ is the time to maturity,

$$d_\pm = \frac{\ln(S/K) + \tau(r \pm \sigma^2/2)}{\sigma\sqrt{\tau}}. \tag{2.1.7}$$

Using the Put-Call Parity Formula, the European put option with the same underlying asset is priced as

$$P^E(t,S) = Ke^{-r\tau} - S + C^E(t,S). \tag{2.1.8}$$

Note that the parameter μ in the original dynamic (2.1.1) for the underlying asset disappears from the pricing formula. Rather, its role is supplanted by the risk-free rate r, leading us to the important result of the Feynman-Kac Theorem, which reads as follows:

Theorem 2.1.1. Feynman-Kac *Let $V(t,S)$ be a solution to the Black-Scholes PDE (2.1.2) with terminal condition $V(T,S) = f(S)$. Then, if the solution exists, $V(t,S)$ admits the following probabilistic representation*

$$V(t,S) = e^{-r(T-t)} \cdot \mathbb{E}^{\mathbb{Q}}[\ f(S(T)) \mid S(t) = S\ ],$$

for all $(t,S) \in [0,T] \times \mathbb{R}$, where $S(t)$ follows a risk-neutral dynamic

$$\mathrm{d}S(t) = rS\ \mathrm{d}t + \sigma S\ \mathrm{d}B^{\mathbb{Q}}(t),$$

and $B^{\mathbb{Q}}(t)$ is a standard Brownian motion under the risk-neutral measure $\mathbb{Q}$.

Although the Black-Scholes model offers a powerful tool to evaluate fair option prices, the universal benchmark it generates is flawed due to the requirement for a fixed volatility σ. The most compelling and widely accepted empirical evidence supporting this assertion is the dependence of the implied volatility on the strikes K, usually

referred to as the "smile", or "skew", as well as the dependence on the maturities T called the "term structure".

The implied volatility σ_* for an externally given price V_* of some vanilla option is defined as the volatility which generates the same theoretical price for the option if inserted in the Black-Scholes formula (2.1.6) [17]. Since σ is a parameter for the dynamic of the underlying asset rather than of the option contract, it is reasonable to assume that the implied volatility of the option will remain constant irrespective of the option parameters, such as K and T, or even the contract type, if the assumption for a constant volatility is essentially unbiased. The validity of this assumption has, however, been questioned in a large number of empirical studies in the existing literature [16, 22, 25].

2.2 The Heston Stochastic Volatility Model

Due to its inadequacy in explaining the volatility bias observed in the option market, many extensions to the Black-Scholes model have been studied and published over the last few decades. For example, in 1973, Merton [23] suggested defining the volatility as a deterministic function of time $\sigma(t)$. This idea was further extended in the works of Dupire [9], and Derman and Kanid [8] in 1994, where they proposed the so-called local volatility model to take into account the state dependence of the volatility $\sigma(t, S(t))$. While the efforts allowed for local fitting of the volatility surface, the model still failed in capturing the smile's enduring pattern as time passed. Worse still, these models may sometimes suggest unsatisfactory prices for exotic options.

One step ahead of the local volatility models are the stochastic volatility models, where another separate stochastic process controls the dynamic of the volatility within the model [10, 11]. In 1993, Heston [13] developed one of the most popular models involving stochastic volatility. One essential advantage of the Heston model is its ability to reasonably replicate market observations. Not only does the model exhibit a market-observed mean-reverting property in volatility, but it also represents one of the first successful attempts to explain the volatility smile and to price exotic options in a manner that aligns with market expectations [11]. Another advantage which distinguishes the Heston model from other models of the same category is the presence of a quasi-closed form solution [11, 13] for European options, which implies a possibly higher computational efficiency compared to the Monte Carlo simulation, as well as other numerical methods for PDEs [11]. As a result, the calibration of the Heston model is usually faster than other alternative models, contributing to one of the main reasons why it is chosen as the fundamental model for experiments in this paper.

2.2.1 The Pricing PDE and Semi-analytical Solution

The dynamic of the Heston model in the risk-neutral world for assets paying no dividends is given as follows

$$\mathrm{d}S(t) = rS\ \mathrm{d}t + \sqrt{v}S\ \mathrm{d}B_S^{\mathbb{Q}}(t), \tag{2.2.1.1}$$

$$\mathrm{d}v(t) = \kappa(\bar{v} - v)\ \mathrm{d}t + \xi\sqrt{v}\ \mathrm{d}B_v^{\mathbb{Q}}(t), \tag{2.2.1.2}$$

$$\mathrm{d}B_S^{\mathbb{Q}}\ \mathrm{d}B_v^{\mathbb{Q}} = \rho\ \mathrm{d}t, \tag{2.2.1.3}$$

where t is the time; $S(t)$ is the current price of the underlying asset; $v(t)$ is the current variance, or the square volatility; r is the risk-free rate; κ stands for the rate at which $v(t)$ reverts to the long term variance $\bar{v}$; ξ is the volatility of the volatility, and finally, ρ is the correlation of the two standard Brownian motions $B_S^{\mathbb{Q}}$ and $B_v^{\mathbb{Q}}$. A strictly positive variance process will be guaranteed under the Feller condition $2\kappa\bar{v} > \xi^2$. The derivation of the pricing PDE of this model can be carried out through a standard hedging argument and can be found in Heston's original article [13] or most of the modern materials discussing the stochastic volatility models such as [11]. As this is not the focus of our article, we shall omit these tedious derivations and present the result alone as follows

$$\frac{1}{2}\left(vS^2V_{SS} + 2\rho\xi vSV_{Sv} + \xi^2 vV_{vv}\right) + rSV_S + \kappa(\bar{v} - v)V_v + V_t - rV = 0. \tag{2.2.1.4}$$

For a European call option $V(t, v, S) := C^E(t, v, S)$ expired at T with a strike price of K, the terminal and boundary conditions are specified to be

$$C^E(T, v, S) = (S - K)^+ \qquad \text{for} \quad t = T\ ; \tag{2.2.1.5}$$

$$C^E(t, v, 0) = 0 \qquad \text{for} \quad S = 0\ ; \tag{2.2.1.6}$$

$$C_S^E(t, v, \infty) = 1 \qquad \text{for} \quad S \to \infty\ ; \tag{2.2.1.7}$$

$$rSC_S^E + \kappa\bar{v}C_v^E + C_t^E - rC^E = 0 \qquad \text{for} \quad v = 0\ ; \tag{2.2.1.8}$$

$$C^E(t, \infty, S) = S \qquad \text{for} \quad v \to \infty\ . \tag{2.2.1.9}$$

The condition (2.2.1.8) is imposed to ensure that the PDE (2.2.1.4) remains satisfied when $v = 0$. Apply the following change of variable

$$\tau = T - t\ \ ,\ \ s = \ln S\ \ ,\ \ k = \ln K, \tag{2.2.1.10}$$

then, according to [13], the fair price for the European call option is

$$C^E(\tau, v, s) = e^s \cdot \mathbb{P}_1 - e^{-r\tau + k} \cdot \mathbb{P}_2, \tag{2.2.1.11}$$

where for $j \in \{1, 2\}$,

$$\mathbb{P}_j(\tau, v, s) = \frac{1}{2} + \frac{1}{\pi}\int_0^\infty \mathrm{Re}\left\{\frac{\exp\left[\ \mathcal{C}_j(\tau; u) + \mathcal{D}_j(\tau; u)v + (s - k)\mathbf{i}u\ \right]}{\mathbf{i}u}\right\}\ \mathrm{d}u. \tag{2.2.1.12}$$

For convenience, we will omit the subscripts $j \in \{1, 2\}$ in the following expressions when no ambiguity is raised. We have

$$\mathcal{C}(\tau; u) = r\tau \mathbf{i}u + \frac{a}{\xi^2}\left\{q_+\tau - 2\ln\left[\frac{1-ge^{d\tau}}{1-g}\right]\right\}, \tag{2.2.1.13}$$

$$\mathcal{D}(\tau; u) = \frac{q_+}{\xi^2}\left[\frac{1-e^{d\tau}}{1-ge^{d\tau}}\right], \tag{2.2.1.14}$$

where

$$q := q_\pm(u) = b - \rho\xi\mathbf{i}u \pm d\ , \tag{2.2.1.15}$$

$$g := g(u) = q_+(u)/q_-(u), \tag{2.2.1.16}$$

$$d := d(u) = \sqrt{(\rho\xi\mathbf{i}u - b)^2 - \xi^2(2w\mathbf{i}u - u^2)}, \tag{2.2.1.17}$$

$$a = \kappa\bar{v}, \tag{2.2.1.18}$$

$$w_1 = 1/2 \ \text{ and } \ w_2 = -1/2, \tag{2.2.1.19}$$

$$b_1 = \kappa - \rho\xi \ \text{ and } \ b_2 = \kappa. \tag{2.2.1.20}$$

Even though the semi-analytical solution has been given explicitly in (2.2.1.10) – (2.2.1.20), evaluating the price using the above formulas is still challenging due to several numerical issues [3, 11]. A common approach for evaluating this semi-analytical solution is to use approximation formulas as Lewis [20] and Alòs [1] did in their respective works. In our experiments involving European call options, we will follow the routine of Alòs et al. [1, 2], where the following approximation for formula (2.2.1.11) is used

$$C^E(\tau, v, s) \approx BS(\tau, \hat{\sigma}_\tau, s) + \mathcal{U}(\tau, v) \cdot \mathcal{H}(\tau, \hat{\sigma}_\tau, s) + \mathcal{R}(\tau, v) \cdot \mathcal{K}(\tau, \hat{\sigma}_\tau, s), \tag{2.2.1.21}$$

where $\hat{\sigma}_\tau = \sqrt{\hat{v}_\tau}$, such that

$$\hat{v}_\tau = \frac{1}{\tau}\int_t^T \mathbb{E}[v(t)|\mathcal{F}_t]\ \mathrm{d}s = \bar{v} + \frac{v-\bar{v}}{\kappa\tau}(1 - e^{-\kappa\tau}), \tag{2.2.1.22}$$

denotes the future average variance and

$$BS(\tau, \sigma, s) = e^s\Phi(d_+) - e^{-r\tau+k}\Phi(d_-), \tag{2.2.1.23}$$

$$d_\pm = \frac{s - k + \tau(r \pm \sigma^2/2)}{\sigma\sqrt{\tau}}, \tag{2.2.1.24}$$

represents the Black-Scholes price of a European call option with constant volatility σ and time to maturity $\tau = T - t$. For the rest of the four functions $\mathcal{U}, \mathcal{H}, \mathcal{R}, \mathcal{K}$, there are

$$\mathcal{U}(\tau, v) = \frac{\rho^2\xi}{2\kappa^2}[\ \bar{v}\kappa\tau - 2\bar{v} + v + e^{-\kappa\tau}(\bar{v}\kappa\tau + 2\bar{v} - v - v\kappa\tau)\ ], \tag{2.2.1.25}$$

$$\mathcal{H}(\tau, \sigma, s) = \frac{\exp\left(s - d_+^2/2\right)}{\sigma\sqrt{2\pi\tau}} \cdot \left(1 - \frac{d_+}{\sigma\sqrt{\tau}}\right), \tag{2.2.1.26}$$

and

$$\mathcal{R}(\tau, v) = \frac{\xi^2}{16\kappa^3}[\ 2\bar{v}\kappa\tau + 2v - 5\bar{v} + \ 4e^{-\kappa\tau}(\bar{v}\kappa\tau + \bar{v} - v\kappa\tau) + \ e^{-2\kappa\tau}(\bar{v} - 2v)\ ], \tag{2.2.1.27}$$

$$\mathcal{K}(\tau, \sigma, s) = \frac{\exp\left(s - d_+^2/2\right)}{\sigma\sqrt{2\pi\tau}} \cdot \left(\frac{d_+^2 - \sigma\sqrt{\tau}d_+ - 1}{\sigma^2\tau}\right). \tag{2.2.1.28}$$

By using this asymptotic approximation, we are freed from the need to perform tedious numerical integrations.

2.2.2 Monte Carlo Simulations

Although Heston's work enables us to price vanilla options with a semi-analytical formula, the Monte Carlo simulation remains a preferred choice for many practitioners to price most of the complex-structured exotic options. Usually, implementing the path simulation for the Heston model using Monte Carlo faces three major technical challenges. The first challenge is how to choose the proper discretization scheme. Normally, the choice of the scheme is closely related to the computational efficiency of the simulation, as a poor scheme, for instance, the Euler discretization, may take a significant number of steps to achieve convergence [11]. After choosing the proper scheme, there are still issues regarding tackling the appearance of a negative variance and how to sample from correlated Brownian motions.

Regarding the choice of schemes, we will mainly adopt the Milstein discretization [11, 18, 24] in our upcoming numerical experiments involving the path simulation. The method was able to refine the Euler scheme by proceeding one order higher [24] in the Itô-Taylor expansion for the variance process $v(t + \Delta t)$. In general, the Milstein discretization for an autonomous SDE

$$\mathrm{d}Y(t) = a(Y)\ \mathrm{d}t + b(Y)\ \mathrm{d}B(t), \tag{2.2.2.1}$$

with $B(t)$ specified as a standard Brownian motion, is given as

$$Y_{t+\Delta t} = Y_t + a(Y_t)\Delta t + b(Y_t)\sqrt{\Delta t}Z + \frac{1}{2}b'(Y_t)b(Y_t)\Delta t(Z^2 - 1), \tag{2.2.2.2}$$

where $Z \sim N(0, 1)$. Hence, in terms of the Heston model, the discretization should be written as

$$S_{t+\Delta t} = S_t + rS_t\Delta t + \sqrt{v_t\Delta t}S_tZ_S + \frac{1}{2}v_tS_t\Delta t(Z_S^2 - 1), \tag{2.2.2.3}$$

$$v_{t+\Delta t} = v_t + \kappa(\bar{v} - v_t)\Delta t + \xi\sqrt{v_t\Delta t}Z_v + \frac{1}{4}\xi^2\Delta t(Z_v^2 - 1), \tag{2.2.2.4}$$

$$[Z_S, Z_v]^T \sim N(\mathbf{0}, \Sigma) \ \text{ where } \ \Sigma = \begin{bmatrix} 1 & \rho \\ \rho & 1 \end{bmatrix}. \tag{2.2.2.5}$$

Note that the above discretization can still generate negative variances; therefore, a reflecting assumption is applied to prevent these situations: if $v_t < 0$, then let $v_t := -v_t$. Finally, to generate Z_S and Z_v with correlation ρ, we set

$$Z_S = \sqrt{1-\rho^2} Z_1 + \rho Z_2, \tag{2.2.2.6}$$

$$Z_v = Z_2, \tag{2.2.2.7}$$

where $Z_1, Z_2 \sim N(0,1)$, $Z_1 \perp Z_2$ are independent and identically distributed standard normal random variables. It can be easily checked that the correlation requirement is satisfied by

$$\text{corr}(Z_S, Z_v) = \sqrt{1-\rho^2} \cdot \text{cov}(Z_1, Z_2) + \rho \cdot \text{var}(Z_2, Z_2) = 0 + \rho = \rho,$$

while the assertion for $\text{var}(Z_S) = \text{var}(Z_v) = 1$ is trivial through basic knowledge of probability theory.

The Milstein scheme achieves a relative balance between the computational efficiency and the simulation effectiveness. On the one hand, it substantially reduces the occurrence of negative variances compared to the Euler scheme [11, 24], enabling faster convergence. On the other hand, it also provides a straightforward recursion formula, which allows for quicker computation compared to other implicit schemes [11].

2.3 The General Calibration Problem

In general, the purpose of almost calibration processes for a specified model is to adjust its parameters so that the model can produce results consistent with the empirical observation. To formalize the problem, we denote the parameter vector which controls the dynamic of the underlying asset to be some $\theta \in \Theta$ such that $\Theta \subset \mathbb{R}^{n_{\text{asset}}}$, and let $\zeta \in \mathbb{R}^{n_{\text{contract}}}$ be the customized input of the contract we aim to price with the model. For any contract price generated through the model, its value is defined as $V^{\mathcal{M}}(\zeta;\theta)$. Meanwhile, if there is an observed price for the same contract over the market, its price is denoted as $V^{MKT}(\zeta)$. Based on the above setting, the general calibration problem for a set of observed contracts with parameters $\{\zeta_i\}_{i=1}^n$ is then defined as finding

$$\hat{\theta} = \text{argmin}_{\theta\in\Theta} \sum_{i=1}^{n} \mathbf{d}(\, V^{\mathcal{M}}(\zeta_i;\theta)\ ,\ V^{MKT}(\zeta_i)\,), \tag{2.3.1}$$

where $\mathbf{d}(\cdot,\cdot)$ is some pre-given metric measuring the "difference" between the theoretical and the real value of the option. Although some other formulation is also possible, the calibration problem is most likely to be present as a weighted non-linear least square problem, namely,

$$\hat{\theta} = \text{argmin}_{\theta\in\Theta} \sum_{i=1}^{n} w_i \cdot \left|\, V^{\mathcal{M}}(\zeta_i;\theta) \ -\ V^{MKT}(\zeta_i) \,\right|^2. \tag{2.3.2}$$

For vanilla options, the implied volatilities are more commonly used as the benchmark for calibrations instead of the price. Let $\sigma_*^{\mathcal{M}}(\zeta;\theta)$ and $\sigma_*^{MKT}(\zeta)$ be theoretical and observed implied volatility for the option, respectively, the problem (2.3.1) then becomes

$$\hat{\theta} = \text{argmin}_{\theta\in\Theta} \sum_{i=1}^{n} \mathbf{d}(\ \sigma_*^{\mathcal{M}}(\zeta_i;\theta)\ ,\ \sigma_*^{MKT}(\zeta_i)\ ). \tag{2.3.3}$$

In the case of the Heston model, let $\tau = T - t$ be the time to maturity, then the parameter for the underlying dynamic should be $\theta = (r, S_0, v_0, \bar{v}, \kappa, \xi, \rho)$ where v_0 is the initial variance $v_0 := v(0)$, and the contract parameters for European call options should be $\zeta = (K, \tau)$. The risk-free rate r and the initial (current) price of the underlying asset S_0 should be viewed as observable and fixed values during the calibration process.

The calibration problem itself, however, is relatively easy to solve when the options can be priced in a simple manner. What makes the task difficult and time-consuming is evaluating the theoretical price $V^{\mathcal{M}}(\zeta;\theta)$, or the corresponding implied volatility $\sigma_*^{\mathcal{M}}(\zeta_i;\theta)$ for the target derivative, where Monte Carlo simulations and numerical algorithms usually come in. One solution to this problem is to find some fast approximators $\widetilde{F}^{\mathcal{M}}(\zeta;\theta)$ for the evaluation of the theoretical values and then calibrate the model through

$$\hat{\theta} = \text{argmin}_{\theta\in\Theta} \sum_{i=1}^{n} \mathbf{d}(\ \widetilde{F}^{\mathcal{M}}(\zeta_i;\theta)\ ,\ V^{MKT}(\zeta_i)\ ), \tag{2.3.4}$$

and that is where the role of neural networks can get involved: to learn a fast approximator for the pricing function off-line "once and for all" [4, 6, 15, 21].

3 The Deep Calibration

3.1 Feedforward Neural Networks

Before formally introducing the deep calibration, we will briefly review the definition and principles of a simple feedforward neural network. A neural network model is usually composed of three main ingredients: the network architecture, the loss function and the learning rule. The network itself can be viewed as a multidimensional function $\varphi_{\mathbf{w}}(\mathbf{x})$ with some specified parameters $\mathbf{w}$. The evaluation of this function depends on the construction of the network, usually consisting of several layers of "neurons". A neuron is the simplest component of the network architecture, which receives signals from other previously connected (hidden) neurons as its input x, performs a weighted summation of these signals $w^T x$, subtracts a bias b and then activates the result through some pre-specified activation function $\sigma_A(z)$ where $z =$

$w^T x - b$. The output of this neuron then becomes the new input for the neurons in the next layer to which it is connected. The weights w, and the bias b, for each neuron to perform its inside calculation during the process then form the set of parameters $\mathbf{w}$ the network $\varphi_{\mathbf{w}}(\mathbf{x})$ has to learn. A simple example of a three-layer feedforward neural network without recurrences in its architecture is shown in Fig. 3.1.1.

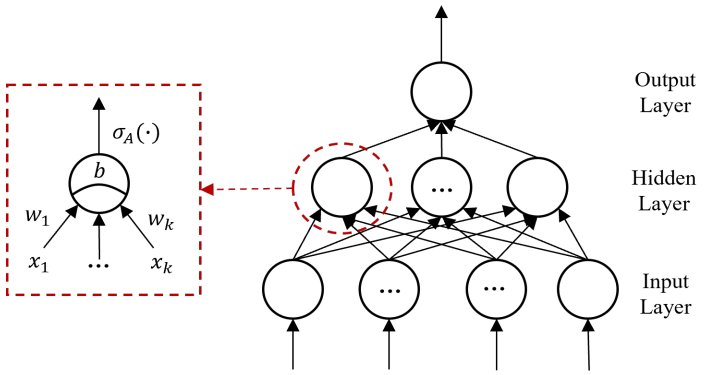

Fig. 3.1.1. A simple three-layer fully connected feedforward neural network.

The learning process is nothing but an optimization problem subject to the selected loss function $L(\mathbf{w})$, and the learning rule gives an algorithm to find the weights $\mathbf{w}$ of the network that minimizes the loss. Past theories have proven that, as long as the proper activation function and a sufficient number of neurons are given, even networks with a single hidden layer can have the ability to approximate any continuous functions and their derivatives with arbitrary accuracy [14].

3.2 The Two-step Approach

The method we discussed at the end of Section 2 for calibrating using neural networks was introduced in [4, 15] as the "two-step approach", where the learning of the pricing function and the final calibration are treated as two separate tasks compared to Hernandez's work [12] where these steps are done by a single network called the inverse map $\Pi^{-1} : \{V^{MKT}(\zeta_i)\}_{i=1}^n \mapsto \hat{\theta}$. The main problem of the latter method, as some have already mentioned [4, 12], is a lack of control over the calibration accuracy.

Alternatively, the two-step approach allows us to directly compare the learned pricing functions with results obtained by traditional numerical methods, thus making it much more convenient to verify the accuracy of the network's fit and to implement further risk management analyses. The method is also more robust since the pricing function is trained independently of actual market data, as only synthetic data is needed for the entire training process. Lastly, the pricing networks trained through modern tensor-based learning frameworks supporting auto-differentiation, such as PyTorch, could yield reliable and fast approximations of the Jacobians required for

calibration algorithms in the second step, thereby again alleviating the computational burden associated with the entire calibration practice.

3.3 Learing the Pricing Functional

3.3.1 Point-wise Learning

In this section, we will mainly discuss various methods developed to learn the pricing function $\widetilde{F}_{\mathbf{w}}^{\mathcal{M}}(\zeta;\theta)$. Here $\mathbf{w}$ denotes the learned parameters of the neural network. Perhaps the most straightforward way to implement this is to learn a point-wise evaluation for the price $V^{\mathcal{M}}(\zeta;\theta)$, which is also known as point-wise learning [4].

Given a set of synthetic training data $\{(\mathbf{x}_i, \mathbf{y}_i)\}_{i=1}^{N_{\text{train}}}$, where $\mathbf{x}_i := (\zeta_i; \theta_i)$ is randomly sampled from some prior distribution for the dynamic and contract parameters and $\mathbf{y}_i := V^{\mathcal{M}}(\zeta_i; \theta_i)$ is the price benchmark obtained using any traditional methods for the model, the point-wise method is defined as minimizing the following target loss function

$$\min_{\mathbf{w}\in\mathbb{R}^{|\mathbf{w}|}} L(\mathbf{w}) = \sum_{i=1}^{N_{\text{train}}} \eta_i \cdot |\varphi_{\mathbf{w}}(\mathbf{x}_i) - \mathbf{y}_i|^2, \tag{3.3.1.1}$$

where $|\mathbf{w}|$ is the number of network parameters, $\varphi_{\mathbf{w}}(\mathbf{x}_i) := \widetilde{F}_{\mathbf{w}}^{\mathcal{M}}(\mathbf{x}_i)$, and $\eta_i > 0$ are just some weight parameters specified before optimization. We then follow the convention [15] of using a fully connected feedforward neural network as structured in Fig. 3.3.1.1 to train this pricing approximator $\varphi_{\mathbf{w}}(\mathbf{x})$.

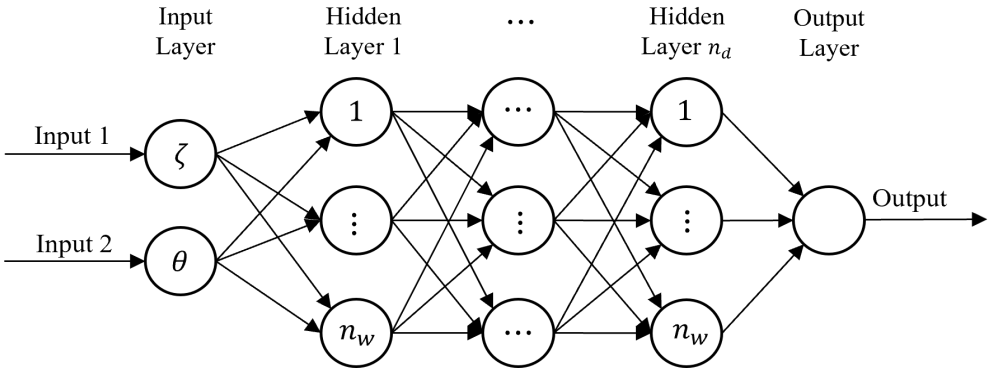

Fig. 3.3.1.1 Architecture for a fully connected feedforward neural network with a uniform width n_w and depth n_d, i.e., n_d hidden layers.

3.3.2 Implicit Grid-based Learning

In their previous work, Horvath et al. [4, 15] also proposed another implicit grid-based learning method which trades the approximator's flexibility for an improved

understanding of the volatility (or price) regularity. In this approach, the contract parameters $\zeta \in \mathbb{R}^{n_{\text{contract}}}$ considered are set to be fixed and are stored in a well-ordered manner in some n_{contract} dimensional discretization grid $\Delta(\zeta) = \{\zeta_{\mathbf{k}}\}_{\mathbf{k}\in\mathbb{K}}$ of size $|\Delta(\zeta)| := |\mathbb{K}|$, where $\mathbb{K}$ is the set of multi-indices $\mathbf{k}$. In terms of the European options, the grid is just $\Delta(\zeta) := \{(K_i, \tau_j)\}_{i=1,j=1}^{n_K,n_T}$ for $n_{\text{contract}} = 2$ and $|\Delta(\zeta)| = n_K n_T$.

The main change added to this method is that the contract parameters will no longer be considered as the network inputs; rather, they will be incorporated into the outputs. Namely, instead of training a point-wise function $\widetilde{F}_{\mathbf{w}}^{\mathcal{M}}(\zeta;\theta) : \mathbb{R}^{n_{\text{contract}}} \times \mathbb{R}^{n_{\text{asset}}} \mapsto \mathbb{R}$, we will train $\widetilde{F}_{\mathbf{w}}^{\mathcal{M}}(\theta) : \mathbb{R}^{n_{\text{asset}}} \mapsto \mathbb{R}^{|\Delta(\zeta)|}$. The loss function for this learning strategy is given as

$$\min_{\mathbf{w}\in\mathbb{R}^{|\mathbf{w}|}} L(\mathbf{w}) = \sum_{i=1}^{N_{\text{train}}^{\text{reduced}}} \sum_{\mathbf{k}\in\mathbb{K}} \eta_{\mathbf{k}} \cdot | \, [\varphi_{\mathbf{w}}(\mathbf{x}_i)]_{\mathbf{k}} - [\mathbf{y}_i]_{\mathbf{k}} \, |^2 . \tag{3.3.2.1}$$

As before, $|\mathbf{w}|$ is the number of network parameters, $\varphi_{\mathbf{w}}(\mathbf{x}_i) := \widetilde{F}_{\mathbf{w}}^{\mathcal{M}}(\mathbf{x}_i)$, and $\eta_i > 0$ are pre-given weight parameters; the data in the training set $\{(\mathbf{x}_i, \mathbf{y}_i)\}_{i=1}^{N_{\text{train}}^{\text{reduced}}}$ is defined as $\mathbf{x}_i := \theta_i \in \Theta$ and $\mathbf{y}_i := V^{\mathcal{M}}(\theta_i) \in \mathbb{R}^{|\Delta(\zeta)|}$ where $[\mathbf{y}_i]_{\mathbf{k}} := V^{\mathcal{M}}(\zeta_{\mathbf{k}};\theta_i)$ for $\zeta_{\mathbf{k}} \in \Delta(\zeta)$ in the grid. The architecture of the network is almost the same as the one shown in Fig. 3.3.1.1 for point-wise training and is presented in Fig. 3.3.2.1.

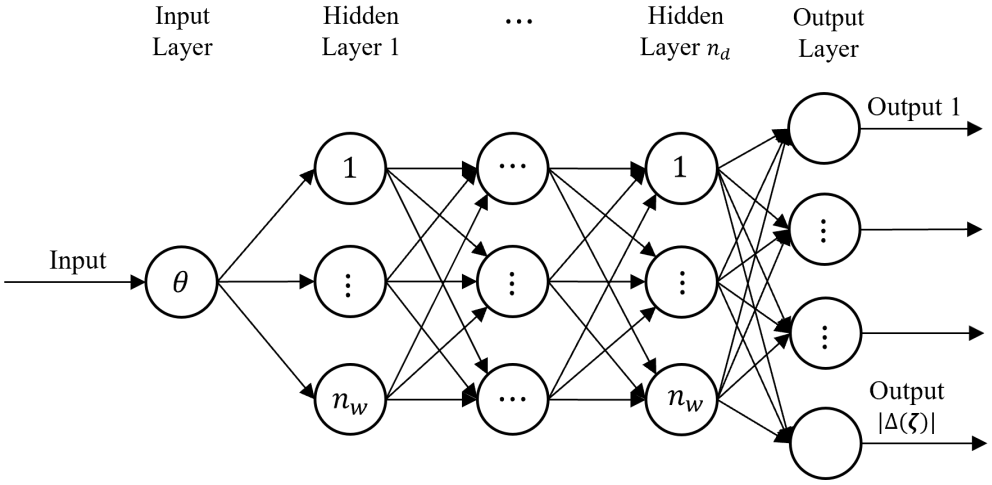

Fig. 3.3.2.1 Architecture for a fully connected feedforward neural network for grid-based learning.

3.4 The Calibration Algorithm: Levenberg-Marquardt Calibration

Suppose we obtain a properly learned approximator $\widetilde{F}^{\mathcal{M}}(\zeta;\theta)$ for the pricing function after step 1 (a point-wise evaluation of grid-based outputs can be accomplished using suitable spline interpolations [4, 15]), a typical calibration procedure is to solve the following minimization problem

$$\hat{\theta} = \text{argmin}_{\theta\in\Theta} \sum_{i=1}^{n} w_i \cdot | \ \widetilde{F}^{\mathcal{M}}(\zeta_i;\theta) \ - \ V^{MKT}(\zeta_i) \ |^2, \tag{3.4.1}$$

which is just a weighted least squares problem as long as the number of parameters $\dim(\theta)$ is less than the number of contract samples selected n. Let $\widetilde{\mathbf{J}}(\theta)$ be the Jacobian of the map

$$\widetilde{\mathbf{F}}(\theta) = [\ \widetilde{F}^{\mathcal{M}}(\zeta_1;\theta) \ , \ \cdots \ , \ \widetilde{F}^{\mathcal{M}}(\zeta_n;\theta) \ ]^T, \tag{3.4.2}$$

with respect to θ and

$$\widetilde{\mathbf{R}}(\theta) = \widetilde{\mathbf{F}}(\theta) - \mathbf{v}, \tag{3.4.3}$$

such that $\mathbf{v} = [\ V^{MKT}(\zeta_1) \ , \ \cdots \ , \ V^{MKT}(\zeta_n) \ ]^T$. Then, the least squares problem (3.4.1) can be solved with the following Levenberg-Marquardt algorithm [4] described in Algorithm 1, where the solution vector θ is updated iteratively $\theta_{k+1} := \theta_k + \Delta\theta_k$ with increments $\Delta\theta_k$ satisfying

$$[\ \widetilde{\mathbf{J}}(\theta_k)^T \mathbf{W} \widetilde{\mathbf{J}}(\theta_k) + \lambda \mathbf{I} \ ] \ \Delta\theta_k = \widetilde{\mathbf{J}}(\theta_k)^T \mathbf{W} \widetilde{\mathbf{R}}(\theta_k). \tag{3.4.4}$$

Here we have $\mathbf{W} = \text{diag}(\mathbf{w})$ for $\mathbf{w} = [w_i, \cdots, w_n]^T$, and $\mathbf{I}$ is just the identity matrix and $\lambda \in \mathbb{R}$ is some real parameter.

Algorithm 1: Levenberg-Marquardt Calibration

Initialize: Initial guess $\theta_0 \in \Theta$, max number of iterations $n_{\max}$, minimum tolerance of step norm $\varepsilon_{\min}$, $\lambda_0 > 0$, $0 < \beta_0 < \beta_1 < 1$;

$\theta \ , n \ \longleftarrow \ \theta_0 \ , \ 0$;
compute $\mathbf{J}(\theta)$, $\widetilde{\mathbf{R}}(\theta)$, and solve equation 3.3.4 for $\Delta\theta$;
while $n < n_{\max}$ and $||\Delta\theta||_2 > \varepsilon_{\min}$ **do**
  compute the relative improvement $c_\theta = \frac{||\widetilde{\mathbf{R}}(\theta)||_2 - ||\widetilde{\mathbf{R}}(\theta+\Delta\theta)||_2}{||\widetilde{\mathbf{R}}(\theta)||_2 - ||\widetilde{\mathbf{R}}(\theta)+\mathbf{J}(\theta)\Delta\theta||_2}$
  if $c_\theta \leq \beta_0$ **then** $\theta \ , \lambda \ \longleftarrow \ \theta \ , \ 2\lambda$;
  if $c_\theta \geq \beta_1$ **then** $\theta \ , \lambda \ \longleftarrow \ \theta + \Delta\theta \ , \ \frac{1}{2}\lambda$;
  compute $\mathbf{J}(\theta)$, $\widetilde{\mathbf{R}}(\theta)$, and solve equation 3.3.4 for $\Delta\theta$;
  $n \ \longleftarrow \ n + 1$

4 Experiment Design and Implementation Details

4.1 Datasets Generation

The first step in learning a pricing approximator using neural networks is to prepare the synthetic data for training. We will mainly investigate the application of the methods discussed in the previous section on European call options. The main parameters are randomly sampled from uniform distributions with lower and upper bounds summarised in Table 4.1.1. Notice that the sign of ρ is set to be negative as it does not really influence the analytical price for European options (see Section 2.2.1).

θ_i	r	v_t	$\bar{v}$	κ	ξ	ρ
lower	0.00	.001	.001	.001	.001	-1.0
upper	.060	.150	.100	5.00	1.00	0.00

Table 4.1.1 Lower and upper bounds for Heston model parameters; r is the annual (compounding) risk-free rate.

Since we are fitting the model with vanilla options, the neural network will be used to fit the volatility surface rather than the price function, i.e., $\widetilde{F}^{\mathcal{M}}(\zeta;\theta) \approx \sigma_*^{\mathcal{M}}(\zeta;\theta)$. Plus, as the volatility shall not largely depend on the price of the underlying asset, we fixed the currently observed price to be $S_t = 1$ and sampled moneyness $M = K/S_t$ instead of the strike K for contract parameters. The way in which contract parameters are picked depends on the training method. If point-wise learning is used, then they are again randomly selected from uniform distributions $M \sim \text{Uni}(0.5, 1.5)$ and for time to maturity (with unit year) $\tau \sim \text{Uni}(0.25, 2.0)$. If grid-based approaches are used, then $M \in \{0.5 + 0.1k\}_{k=0}^{10}$ and $\tau \in \{0.25 + 0.25k\}_{k=0}^{7}$, which gives a discretization grid of size $11 \times 8 = 88$.

After finishing sampling the parameters, we can apply either the asymptotic formula (2.2.1.21) or the discretization method introduced in Section 2.2.2 to generate the price benchmark for the option under selected parameters $\mathbf{x}_i = (\theta_i, \zeta_i)$ and then find the corresponding implied volatility $\mathbf{y}_i = \sigma_*^{\mathcal{M}}(\zeta_i;\theta_i)$ with common root-finding algorithms. For exotic options, the method of discretization grids and Monte Carlo simulations is still applicable to generate a relatively fair price benchmark.

In the experiment, 880,000 point-wise samples $(\mathbf{x}_i, \mathbf{y}_i)$ were produced for the point-wise net; however, for grid-based learning, only $880,000/88 \approx 10,000$ samples were selected to ensure that each network accesses approximately a same amount of information through training. For each dataset generated, 60% of the samples will be

used to train the network, 20% of them will be assigned as the validation set, and the remaining 20% will form the test set.

4.2 Network Architecture and Training

A similar network structure will be applied to investigate both methods we tried, i.e., both point-wise learning and grid-based learning, to train the implied volatility approximators. Apart from a different number of input and output neurons, both networks will share an architecture consisting of four fully connected hidden layers, each containing 30 neurons equipped with ELU activation.

$$\sigma_{\text{ELU}}(z) = \begin{cases} z & \text{if } \ z > 0 \ , \\ e^z - 1 & \text{if } \ z \leq 0 \ . \end{cases} \tag{4.2.1}$$

Before entering into the network, the input parameters θ_i (and ζ_i for point-wise evaluation) are all mapped to $[-1, 1]$ using their corresponding lower and upper bounds $lb(\theta_i)$ and $ub(\theta_i)$ specified in Table 4.1.1 with the following scaling law

$$\text{scale}(\theta)_i = \frac{2\theta_i - [ub(\theta_i) + lb(\theta_i)]}{ub(\theta_i) - lb(\theta_i)} \in [-1, 1] \ . \tag{4.2.2}$$

The number of trainable network parameters has been summarized in Table 4.2.1 below. The network will be trained using the Adam optimizer [19].

Method	Input	Hidden		Output	Total
		Layer 1	Layer 2-4		
Point-wise	(8, 0)	(30, 270)	(30, 930)	(1, 31)	(99, 3091)
Grid-based	(6, 0)	(30, 210)	(30, 930)	(144, 4464)	(240, 7464)

Table 4.2.1 Number of neurons (the first number in the bracket) and trained parameters (the second number in the bracket) in each layer for different training methods.

4.3 Measuring the Calibration Accuracy

After getting the desired point-wise approximator $\widetilde{F}^{\mathcal{M}}(\zeta;\theta)$ for the volatility surface $\sigma_*^{\mathcal{M}}(\zeta;\theta)$, we can find the calibrated model parameter $\hat{\theta}$ with the Levenberg-Marquardt algorithm introduced in section 3.4 by solving the calibration problem

$$\hat{\theta} = \text{argmin}_{\theta\in\Theta} \sum_{i=1}^{n} \mid \widetilde{F}^{\mathcal{M}}(\zeta_i;\theta) - \sigma_*^{MKT}(\zeta_i) \mid^2 \ . \tag{4.3.1}$$

To test the calibration result, we will first generate a set of "market" given volatilities $\{\sigma_*^{MKT}(\zeta_i)\}_{i=1}^n$ using some pre-specified parameter θ^{MKT} and then determine the

difference between this "market parameter" θ^{MKT} and the $\hat{\theta}$ we find through solving the above calibration problem (4.3.1). The two main measures [4, 15] we used are defined as the relative error

$$E_R(\hat{\theta}) = \left| \frac{\theta^{MKT} - \hat{\theta}}{\theta^{MKT}} \right|, \tag{4.3.2}$$

and the root mean square error (RMSE)

$$\text{RMSE}(\hat{\theta}) = \sqrt{\sum_{i=1}^{n} \mid \widetilde{F}^{\mathcal{M}}(\zeta_i; \hat{\theta}) - \sigma_*^{MKT}(\zeta_i) \mid^2}, \tag{4.3.3}$$

which allows us to evaluate both the overall calibration quality and the parameter sensitivity to the model.

5 Numerical Results

5.1 Approximation Networks

We trained the point-wise net with a batch size of 2048 for 200 epochs and the grid-based net with a batch size of 32 for 5000 epochs. Both training processes are finished within the same timeframe of approximately 30 minutes. The recorded changes in the average square loss for each parameterized point regarding the epoch number are shown in Fig. 5.1.1 below.

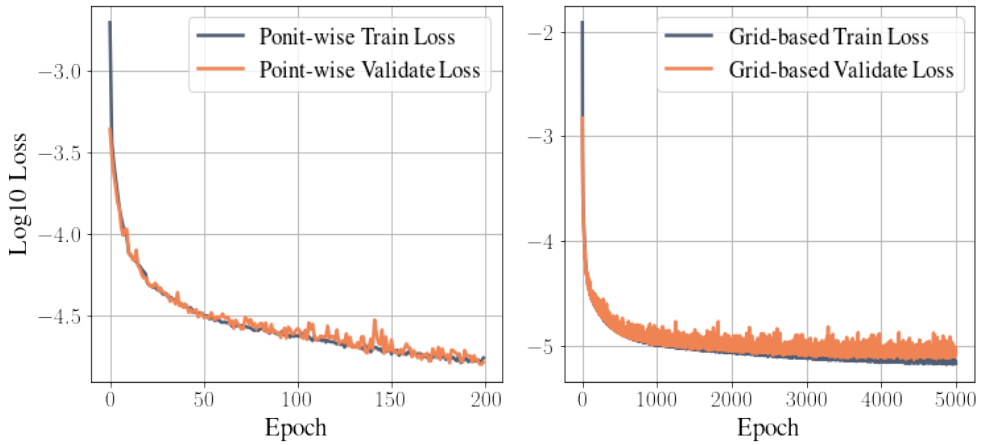

Fig. 5.1.1 The plot on the left gives the change on the log-mean-square-loss for each sample in the point-wise net, calculated as $\mathbf{log}_{10}\mathbb{E}|\tilde{\mathbf{y}}_i - \mathbf{y}_i|^2$; The plot on the left gives change on the corresponding loss for the grid-based net, calculated as $\mathbf{log}_{10}\frac{\mathbb{E}|\tilde{\mathbf{y}}_i - \mathbf{y}_i|^2}{88}$. $\mathbf{y}_i$ denotes the neural network output.

It is not surprising that the grid-based net can run more epochs compared to the point-wise net, as we were training it using much fewer samples due to our control of $N_{\text{train}} = |\Delta(\zeta)| \times N_{\text{train}}^{\text{reduced}}$. The training result, to some extent, reveals that both methods have the potential to reach a same scale of accuracy for an equivalent amount of "training information" passed in. This is particularly good news for grid-based learning, as for complex option structures, the computational cost of generating training data required for grid-based methods using Monte Carlo is much lower compared to the point-wise approach [4, 15]. When the ability to produce training samples is limited, the advantages of grid-based learning will then be greatly highlighted.

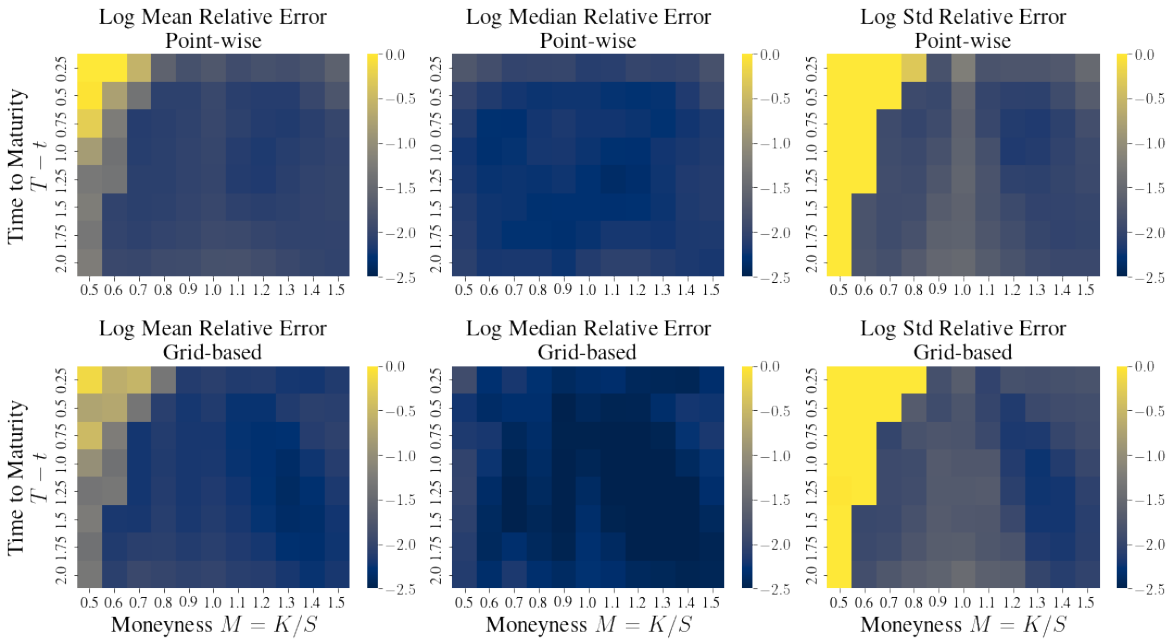

Fig. 5.1.2 $\log_{10}$ mean, median and standard deviation of the relative errors of the point-wise and grid-based net outputs compared to the benchmark over the discretization grid $\Delta(\zeta)$.

The relative errors of each network's outputs over the discretization grid $\Delta(\zeta)$ are computed with respect to the benchmark generated using the same approach mentioned in Section 4.1. The calculated statistics are present in Fig 5.1.2, which shows that most of the average relative errors are around $10^{-1.5} \approx 3\%$ across all parameter combinations, and the value is further decreased to a scale of 1% in terms of the medians. It can be easily observed that the approximation is less accurate for short maturities and deep-out-of-the-money or deep-in-the-money options. Finally, it appears that in general, grid-based learning tends to perform better than point-wise learning with the same amount of training information, and this advantage may be further amplified when sampling capability is limited.

5.2 Calibration Accuracy

We next evaluate the calibration accuracy of the grid-based net on European call options. For the convenience of the programming implementation, the parameters are calibrated directly under scaled values (4.2.2). For each calibration, 88 implied volatilities over the same grid $\Delta(\zeta)$ in Section 4.1 are generated using a set of randomly selected market parameters θ^{MKT}.

Fig. 5.2.1 below gives the relative error of each calibrated scaled parameter scale$(\hat{\theta}_i)$ obtained through deep calibration using the grid-based net. Unlike the plots obtained by [4], where the relative errors tend to be higher for parameters whose unscaled values are close to zero, our plots demonstrate a higher degree of inaccuracy for parameters with small absolute values after scaling. Plus, the calibration seemed to indicate a relatively high sensitivity to κ, as it was the only parameter whose probability of having a relative error less than 1% did not exceed 0.7.

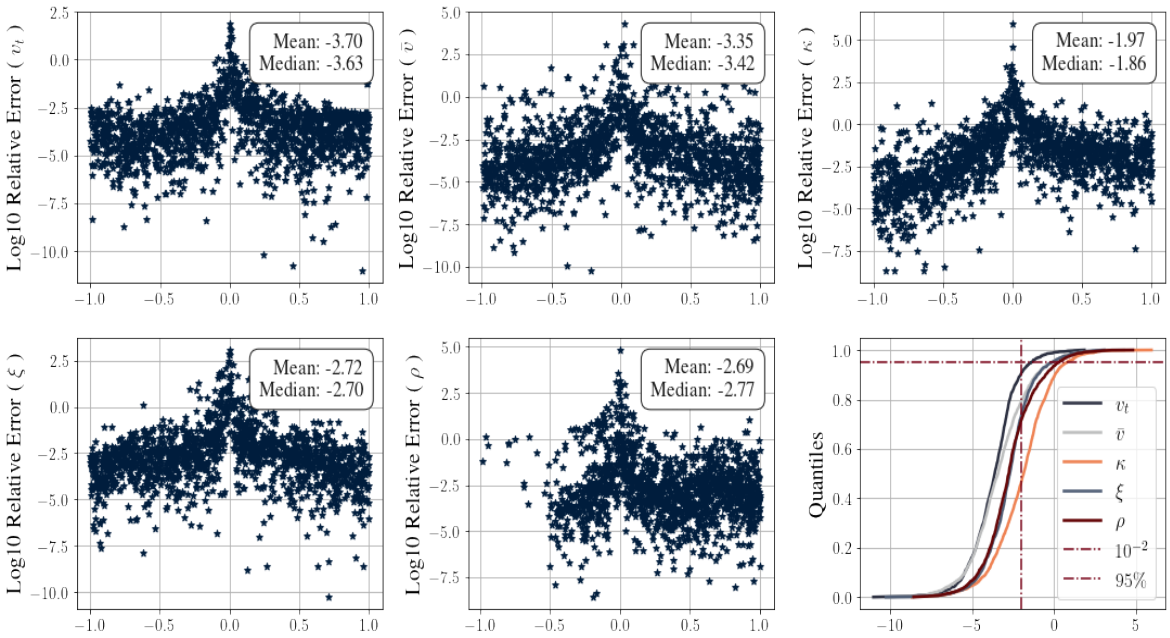

Fig. 5.2.1 The first five plots give the relationship between the log relative errors (y-axis) of the calibrated scaled parameters and their corresponding true values (x-axis). The sixth plot gives the cumulative distribution function for the log relative errors of each kind of parameter, where the x-axis is given as the possible values of the log relative error.

To close our discussion, Fig 5.2.2 provides the distribution of root mean square errors (RMSE) generated using the 1500 test samples. The plot shows that 95% of the RMSEs are below approximately 4%.

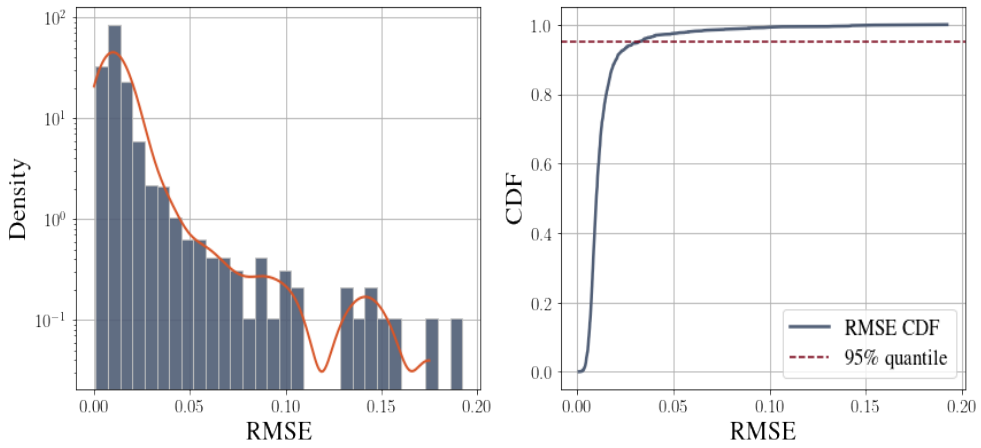

Fig. 5.2.2 Distribution of the root mean square error RMSE.

6 Conclusion

This paper mainly discusses the implementation of the two-step deep calibration on the Heston model. In the first step, both point-wise and grid-based nets are tried to fit the volatility surface of the model. Results show that the accuracy of the neural networks is mostly satisfactory except for cases where options have short maturities or are currently deeply out-of-the-money or in-the-money. The problem may be alleviated if more carefully selected data can be generated and added to the training process. In the second step, we applied the grid-based network to calibrate the model parameters and discovered an overall precise calibration for the scaled parameters apart from the portions close to 0.

We also made many other attempts that were not detailed in this paper. For example, we tried to directly learn the pricing network through the PDE systems with unsupervised learning methods to accelerate the training process and even eliminate the need to collect synthetic data. Besides, convolutional neural networks were also tried to enhance the performance of the grid-based learning approach. Although the results were not entirely satisfactory, we believe there may exist some directions for further improvement within these efforts.

7 References

[1] E. Alòs. "A Decomposition Formula for Option Prices in the Heston Model and Applications to Option Pricing Approximation". In: *Finance Stochast.* 16 (Jan. 2009). DOI: 10.1007/s00780-012-0177-0.

[2] E. Alòs, R. De Santiago, and J. Vives. "Calibration of stochastic volatility models via second-order approximation: The Heston case". In: *International Journal of Theoretical and Applied Finance* 18.06 (2015), p. 1550036. URL: https://econ-papers.upf.edu/papers/1346.pdf (visited on 03/16/2024).

[3] P. Altmeyer et al. "Option pricing in the Heston stochastic volatility model: an empirical evaluation". In: (2018). URL: https://thevoice.bse.eu/wp-content/uploads/2018/10/altmeyer_grapendal_pravosud_quintana_2018.pdf (visited on 03/16/2024).

[4] C. Bayer et al. *On deep calibration of (rough) stochastic volatility models*. 2019. arXiv: 1908.08806 [q-fin.MF].

[5] F. Black and M. Scholes. "The Pricing of Options and Corporate Liabilities". In: *Journal of Political Economy* 81.3 (1973), pp. 637–654. ISSN: 00223808, 1537534X. URL: http://www.jstor.org/stable/1831029 (visited on 03/14/2024).

[6] J. Cao et al. "Deep Learning for Exotic Option Valuation". In: *The Journal of Financial Data Science* 4 (Dec. 2021), jfds.2021.1.083. DOI: 10.3905/jfds.2021.1.083.

[7] A. Cartea. *Lecture Notes for B8.3 Mathematical Models of Financial Derivatives*. University of Oxford. 2023. URL: https://courses.maths.ox.ac.uk/pluginfile.php/103475/mod_resource/content/1/lecturenotes.pdf (visited on 03/14/2024).

[8] E. Derman and I. Kani. "Riding on a Smile". In: *Risk* 7 (Jan. 1994). URL: https://www.researchgate.net/publication/239059413_Riding_on_a_Smile (visited on 03/15/2024).

[9] B. Dupire. "Pricing with a Smile". In: 1994. URL: https://api.semanticscholar.org/CorpusID:9470371 (visited on 03/15/2024).

[10] J.P. Fouque, G.C. Papanicolaou, and K.R. Sircar. *Derivatives in Financial Markets with Stochastic Volatility*. Cambridge University Press, 2000.

[11] J. Gatheral. *The Volatility Surface: A Practitioner's Guide*. John Wiley and Sons, Ltd, 2012. ISBN: 9781119202073. DOI: https://doi.org/10.1002/9781119202073.ch2.

[12] A. Hernandez. "Model Calibration with Neural Networks". In: *Neuroeconomics eJournal* (2016). URL: https://api.semanticscholar.org/CorpusID:53338379 (visited on 03/17/2024).

[13] Steven L. Heston. "A Closed-Form Solution for Options with Stochastic Volatility with Applications to Bond and Currency Options". In: *The Review of Financial Studies* 6.2 (1993), pp. 327–343. ISSN: 08939454, 14657368. URL: http://www.jstor.org/stable/2962057 (visited on 03/15/2024).

[14] K. Hornik, M. Stinchcombe, and H. White. "Universal approximation of an unknown mapping and its derivatives using multilayer feedforward networks". In: *Neural networks* 3.5 (1990), pp. 551–560.

[15] B. Horvath, A. Muguruza, and M. Tomas. "Deep learning volatility: a deep neural network perspective on pricing and calibration in (rough) volatility models". In: *Quantitative Finance* 21.1 (2021), pp. 11–27. DOI: 10.1080/14697688.2020.1817974. URL: https://doi.org/10.1080/14697688.2020.1817974 (visited on 03/17/2024).

[16] J. Hull and A. White. "Value At Risk When Daily Changes In Market Variables Are Not Normally Distributed". In: *Journal of Derivatives* 5 (Jan. 1999), pp. 9–19. DOI: 10.3905/jod.1998.407998.

[17] J.C. Hull. *Options, Futures, and Other Derivatives (Eleventh edition)*. Pearson Education, 2021, pp. 358–359.

[18] C. Kahl and P. Jackel. "Fast Strong Approximation Monte-Carlo Schemes for Stochastic Volatility Models". In: *Quantitative Finance* 6 (Dec. 2006), pp. 513–536. DOI: 10.1080/14697680600841108.

[19] D.P. Kingma and J. Ba. *Adam: A Method for Stochastic Optimization*. 2017. arXiv: 1412.6980 [cs.LG].

[20] A. Lewis. *Option Valuation Under Stochastic Volatility*. Jan. 2000.

[21] W.A. Mcghee. "An Artificial Neural Network Representation of the SABR Stochastic Volatility Model". In: *CompSciRN: Other Machine Learning (Topic)* (2018). URL: https://api.semanticscholar.org/CorpusID:68102474 (visited on 03/17/2024).

[22] A. Melino and S.M. Turnbull. "Pricing foreign currency options with stochastic volatility". In: *Journal of Econometrics* 45.1 (1990), pp. 239–265. ISSN: 0304-4076. DOI: https://doi.org/10.1016/0304-4076(90)90100-8. URL: https://www.sciencedirect.com/science/article/pii/0304407690901008.

[23] R.C. Merton. "Theory of Rational Option Pricing". In: *The Bell Journal of Economics and Management Science* 4.1 (1973), pp. 141–183. ISSN: 00058556. URL: http://www.jstor.org/stable/3003143 (visited on 03/14/2024).

[24] F.D. Rouah. "Euler and milstein discretization". In: *Documento de trabajo, Sapient Global Markets, Estados Unidos. Recuperado de www. frouah. com* 87 (2011), p. 88. URL: https://www.frouah.com/finance%20notes/Euler%20and%20Milstein%20Discretization.pdf (visited on 03/15/2024).

[25] M. Rubinstein. "Nonparametric tests of alternative option pricing models using all reported trades and quotes on the". In: 1985. URL: https://api.semanticscholar.org/CorpusID:150473739 (visited on 03/15/2024).

[26] P. Wilmott, S. Howison, and J. Dewynne. “The Black–Scholes Model”. In: *The Mathematics of Financial Derivatives: A Student Introduction*. Cambridge University Press, 1995, pp. 33–57.

A Appendix

A.1 Codes

A.1.1 BlackScholes.py

```
import numpy as np
from scipy.special import ndtr
from scipy import optimize

class BSModel:

    def __init__(self, rf):

        self.rf = rf  # annual risk free rate

    def PriceEuropean(self, S, sigma, tau, K, type=1):

        # S     : current underlying price
        # sigma : constant volatility
        # tau   : time to maturity tau = T - t, unit year
        # K     : strike
        # type  : 'call' for 1 or 'put' for -1

        rf = self.rf
        std = sigma * np.sqrt(tau)
        discount = np.exp(-rf*tau)

        d1 = (np.log(S/K) + tau * (rf + sigma**2/2)) / std
        d2 = d1 - std

        CallPrice = S * ndtr(d1) - K * discount * ndtr(d2)

        if type == 1:
            return CallPrice
        elif type == -1:
            return K * discount - S + CallPrice
        else :
            raise ValueError("Wrong type given.")
```

```
    def prior_func(self, sigma, S, tau, K, type,
                                    option_price):
        res = self.PriceEuropean(S, sigma, tau, K, type) \
                                            - option_price
        return res


    def getImpliedVol(self, option_price,
                          S, tau, K, type=1):

        # This function computes the implied volatility
        # corresponding to the given option price

        implied_vol = optimize.root_scalar(
                                    self.prior_func,
                                    bracket=[0.001, 10],
                                    args=(S, tau, K, type,
                                    option_price) )
        return implied_vol.root
```

A.1.2 Heston.py

```
import numpy as np
from src.BlackScholes import BSModel

class HestonModel:

    def __init__(self,
                 rf,      # annual risk free rate
                 kappa,   # variance return rate
                 vbar,    # long term variance
                 xi,      # volatility of the volatility
                 rho      # correlation
    ):

        self.rf = rf
        self.kappa = kappa
        self.vbar = vbar
        self.xi = xi
        self.rho = rho
        self.FellerConditionSatisfied = 2 * kappa * vbar >
    xi**2

        # Parameters for European Call Pricing
        self.BSmodel = BSModel(rf)
        self.BS = lambda tau, sigma, s, k : \
                    self.BSmodel.PriceEuropean(np.exp(s),
                    sigma, tau, np.exp(k))
```

```
    def PriceEuropean(self, S, v, tau, K, type=1):

        # S     : current underlying price
        # v     : current variance
        # tau   : time to maturity tau = T - t, unit year
        # K     : strike
        # type  : 'call' for 1 or 'put' for -1

        s = np.log(S)      ;  k = np.log(K)
        rf = self.rf       ;  kappa = self.kappa
        vbar = self.vbar   ;  xi = self.xi
        rho = self.rho     ;  kappa_tau = kappa * tau
        vbar_kappa_tau = vbar * kappa * tau
        ekt = np.exp(-kappa_tau)

        sigma_hat = np.sqrt( vbar
                    + (v-vbar) / kappa_tau * (1-ekt) )

        std_hat = sigma_hat*np.sqrt(tau)

        d1 = ( np.log(S/K)
               + tau*(rf+sigma_hat**2/2) ) / std_hat

        U = (rho**2*xi) / (2*kappa**2) \
            * ( vbar_kappa_tau - 2*vbar + v
              + ekt*(vbar_kappa_tau + 2*vbar - v
                    - v*kappa_tau) )

        R = xi**2 / (16*kappa**3) \
            * ( 2*vbar_kappa_tau + 2*v - 5*vbar
              + 4*ekt*(vbar_kappa_tau + vbar - v*kappa_tau)
              + ekt**2*(vbar-2*v) )

        H = np.exp(s-d1**2/2) / (np.sqrt(2*np.pi)*std_hat) \
        * (1-d1/std_hat)

        L = np.exp(s-d1**2/2) / (np.sqrt(2*np.pi)*std_hat) \
        * (d1**2-std_hat*d1-1)/(std_hat**2)

        CallPrice = self.BS(tau, sigma_hat, s, k) \
                    + U * H + R * L

        if type == 1:
            return CallPrice
        elif type == -1:
            return K * np.exp(-rf*tau) - S + CallPrice
        else :
            raise ValueError("Wrong type given.")
```

```
    def getImpliedVol(self, MarketOptionPrice, S,
                          tau, K, type=1):
        return self.BSmodel.getImpliedVol(MarketOptionPrice,
                                          S, tau, K, type)


    def PathSimulation(self, S0, v0, tau, dt=1/365):

        # Generate one MC path for the Heston model

        rf = self.rf
        kappa = self.kappa
        vbar = self.vbar
        xi = self.xi
        rho = self.rho
        N_steps = round(tau/dt)

        S = [S0]
        v = [v0]
        t = 1

        while t <= N_steps:

            Z1 = np.random.normal(0, 1)
            Zv = np.random.normal(0, 1)
            ZS = np.sqrt(1-rho**2) * Z1 + rho * Zv

            S_previous , v_previous = S[t-1] , v[t-1]

            vt = v_previous + kappa*(vbar-v_previous) * dt \
                 + xi*np.sqrt(v_previous*dt)*Zv \
                 + xi**2*dt*(Zv**2-1)/4

            vt = max(vt,-vt) # Reflecting assumption

            St = S_previous + rf*S_previous*dt \
                 + np.sqrt(v_previous*dt)*S_previous*ZS \
                 + S_previous*v_previous*dt*(ZS**2-1)/2

            S.append(St)
            v.append(vt)
            t+=1

        return (S,v)
```

A.1.3 PointwiseNet.py

```
import torch
import numpy as np
import torch.nn as nn
import torch.optim as optim
from torch.optim.lr_scheduler import StepLR

class PointwiseFNN(nn.Module):

    ## Initialize
    def __init__(self, loader):
        super().__init__()
        self.loader = loader

        self.train_loss = []
        self.validate_loss = []

        # loss function, mse
        self.loss_func = nn.MSELoss(reduction='sum')
        self.hidden1 = nn.Sequential( nn.Linear(8,30) ,
                                      nn.ELU() )
        self.hidden2 = nn.Sequential( nn.Linear(30,30),
                                      nn.ELU() )
        self.output  = nn.Sequential( nn.Linear(30,1),
                                      nn.Sigmoid() )


    ## Forward function
    def forward(self, x):
        # [batch_size, 8 ] ---> [batch_size, 30]
        x = self.hidden1(x)
        # [batch_size, 30] ---> [batch_size, 30]
        x = self.hidden2(x)
        # [batch_size, 30] ---> [batch_size, 30]
        x = self.hidden2(x)
        # x : [batch_size, 30] ---> [batch_size, 30]
        x = self.hidden2(x)
        # [batch_size, 30] ---> [batch_size, 1 ]
        x = self.output(x)
        return x


    ## Validation function
    def Validate(self):

        self.eval()          # convert to test mode
        validate_loss = 0
        validate_loader = self.loader['validate']
        batch_num = len(validate_loader) # number of batches
```

```
        with torch.no_grad():
            for x , y in validate_loader:

                # forward calculation
                y_hat = self.forward(x)

                # compute the total loss for all batches
                validate_loss += self.loss_func(y_hat,
                                                y).item()

            # compute the average loss for each batch
            validate_loss /= batch_num

        return validate_loss



    ## Train function
    def Train(self, num_epochs, learning_rate,
        lr_step_size=10, min_lr =5e-4, abs_tolerance=0.001):

        train_loader  = self.loader['train']
        n_train_batches = len(train_loader)

        optimizer = optim.Adam(
                               self.parameters(),
                               lr=learning_rate
                               )  # adam optimizer

        scheduler = StepLR(
                           optimizer,
                           step_size=lr_step_size, gamma=0.5
                           )   # learning updater

        for epoch in range(num_epochs): # training starts

            print('Epoch [{}/{}]'.format(epoch+1,
                                         num_epochs))

            self.train()  # convert back to train mode
            train_loss = 0

            for i, (x,y) in enumerate(train_loader):

                # forward calculation
                y_hat = self.forward(x)
                # evaluate loss
                loss = self.loss_func(y_hat, y)
                # clear gradients
                optimizer.zero_grad()
```

```
            # back propgation
            loss.backward()
            # update parameters
            optimizer.step()
            # compute the total loss for all batches
            train_loss += loss.item()

            # Display the training progress
            if (i==0) or \
               ((i+1) % round(n_train_batches/5) == 0):
                print( 'Epoch [{}/{}], Step [{}/{}], \
                        Loss: {:.4f}'.format(
                        epoch+1, num_epochs,
                        i+1, n_train_batches,
                        loss.item()) )

        # record average validate batch loss
        self.validate_loss.append(self.Validate())

        # record average train batch loss
        self.train_loss.append(
                        train_loss/len(train_loader))

        print( 'Epoch [{}/{}], Avg.Train Loss: {:.4f}, \
                Avg. Validate Loss: {:.4f}'.format(
                epoch + 1, num_epochs,
                self.train_loss[-1],
                self.validate_loss[-1]) )

        # auto termination
        if (self.validate_loss[-1] < abs_tolerance):
            break

        elif len(self.validate_loss) > 26 :
            temp_validate_loss = np.array(
            self.validate_loss)[-1:-27:-1]

            temp_validate_loss_diff = np.diff(
            temp_validate_loss)

            temp_rel_validate_loss = np.abs(
            temp_validate_loss_diff\
            /temp_validate_loss[:25])

            if ( temp_rel_validate_loss < 0.01 ).sum()\
                                              == 25 :
                break

        elif optimizer.param_groups[0]['lr'] > min_lr :
            scheduler.step()      # update learning rate
```

A.1.4 GridbaseNet.py

Here we provide the code for the main network construction. The train and validate functions are the same as those in the PointwiseNet.py file, with only minor modifications. One can turn to this site (Github) for a full version of the code.

```
import torch
import numpy as np
import torch.nn as nn
import torch.optim as optim
from torch.optim.lr_scheduler import StepLR

class GridbasedFNN(nn.Module):

    ## Initialize
    def __init__(self, loader):
        super().__init__()
        self.loader = loader

        self.train_loss = []
        self.validate_loss = []

        # loss function, mse
        self.loss_func = nn.MSELoss(reduction='sum')

        self.hidden1 = nn.Sequential( nn.Linear(6,30) ,
                                      nn.ELU() )
        self.hidden2 = nn.Sequential( nn.Linear(30,30),
                                      nn.ELU() )
        self.output  = nn.Sequential( nn.Linear(30,88),
                                      nn.Sigmoid() )



    ## Forward function
    def forward(self, x):
        # x : [batch_size, 6 ] ---> [batch_size, 30]
        x = self.hidden1(x)
        # x : [batch_size, 30] ---> [batch_size, 30]
        x = self.hidden2(x)
        # x : [batch_size, 30] ---> [batch_size, 30]
        x = self.hidden2(x)
        # x : [batch_size, 30] ---> [batch_size, 30]
        x = self.hidden2(x)
        # x : [batch_size, 30] ---> [batch_size, 88]
        x = self.output(x)
        return x
```

A.1.5 Grid-based Calibration

```
## Calibration
data_saved_cal = Readpickle('./data/European/\
                            calibration_data_new.pkl')
theta_true = data_saved_cal['dataset'].tensors[0]
vol = data_saved_cal['dataset'].tensors[1]

def f_gb(theta, yy): # This function evaluates the residue.
    theta = torch.tensor(theta, dtype=torch.float32,
                                requires_grad=True)
    theta = torch.cat((torch.tensor([-1/6]), theta))
    with torch.no_grad():
        y_gb = model[1](theta)
    return (y_gb - yy).detach().numpy()

def Joc_gb(theta, yy):
    # This function returns the Jacobian of f_gb.
    theta = torch.tensor(theta, dtype=torch.float32,
                                requires_grad=True)
    theta = torch.cat((torch.tensor([-1/6]),theta)).\
                                requires_grad_(True)

    yy = yy.requires_grad_(True)
    J = torch.autograd.grad(
                        (model[1](theta) - yy)[0], theta,
                        create_graph=True )[0].unsqueeze(0)

    for i in range(1,len(yy)):
        grad = torch.autograd.grad(
                        (model[1](theta) - yy)[i], theta,
                        create_graph=True)[0].unsqueeze(0)
        J = torch.cat((J,grad),dim=0)

    return J.detach().numpy()[:,1:]

# Grid-based calibration
from scipy.optimize import least_squares
theta_hat = []
for i in tqdm(range(len(theta_true))) :
    yy = vol[I]
    theta_temp = least_squares(
                  lambda theta: f_gb(theta, yy),
                  x0=np.array([0,0,0,0,0]),
                  jac= lambda theta: Joc_gb(theta, yy),
                method='lm',gtol=1E-10).x

    theta_temp = np.insert(theta_temp, 0, -1/6)
    theta_hat.append(theta_temp)
```