Deep learning for rough volatility models: A study of calibration approaches

Deep learning af rough volatilitet modeller: Et studie om kalibrerings tilgange

AARHUS UNIVERSITY

Department of Economics and Business Economics

Author

Christian Dall

Student number: 201708926

Supervisor

Bezirgen Veliyev

Submitted on 1st of June 2022

Characters including spaces: 137.060

This thesis may be published

Abstract

In recent years, the option pricing literature has seen a growing interest in rough volatility models due to the overwhelming evidence showing roughness in financial equities. Due to the non-Markovianity of rough volatility models, parameter calibration imposes significant computational bottlenecks, rendering the models infeasible in practice. To improve speed, recent research has successfully employed neural networks to reduce calibration time to a matter of seconds. This thesis examines the two main approaches of option calibration with neural networks in the literature, often referred to as the one-step and two-step approaches.

The one-step approach employs a convolutional neural network to directly calibrate parameters from the observed implied volatility surface. The two-step approach first trains a network to predict option prices using model parameters as inputs. Secondly, a numerical optimization algorithm optimizes parameters to minimize the distance between network predicted prices and observed prices. Each approach is rigorously evaluated using four different volatility specifications to determine their strengths and weaknesses. The networks of each approach are tested on both simulated data and real-world data using call options of the SPX index.

Using simulated data, neural network performance is shown to be affected by parameter values and especially low for high strike, short maturity calls. Calibration of simulated data shows that the one-step approach has higher precision than the two-step approach and accuracy differs between parameters for both approaches. The networks of both approaches are shown to have some extrapolation capabilities but interestingly, only for the rBergomi volatility model. Adding regularization only improves the one-step approach and only by using batch normalization.

The calibration of parameters to the SPX index finds parameter restrictions must be imposed to ensure networks perform adequately. This highlights a practical issue with the one-step as parameter restrictions cannot be imposed after training.

1. Introduction

Since the introduction of the Black Scholes model in 1973 (Black & Scholes, 1973) the option pricing literature has been vastly extended to include different models and theories about the evolution of the underlying asset. The Black Scholes model has become a common tool for option pricing, due to the availability of a closed-form solution for the price and Greeks. The simplicity however comes with a catch, the volatility is assumed to be constant, whereby the implied volatility surface becomes flat. This has been strongly rejected empirically due to the well-known smile shape of the implied volatility surface. Following the Black and Scholes model, many advancements have been made to obtain models with more realistic assumptions that better fit the implied volatility surface. These advancements can generally be grouped into (A) local volatility models like (Dupire et al., 1994) and the CEV of (Cox, 1997). (B) Stochastic volatility models like Heston (Heston, 1993), SABR (Hagan et al., 2002), and GARCH (Engle, 1982). (C) Jump diffusion like Merton's (Merton, 1976) or SVJJ (Matytsin, 1999).

While these new models allowed for a better fit of the implied volatility surface, their complexity meant that a closed-form solution for options prices and Greeks no longer existed. To calculate the risk-neutral price of an option one therefore must use Monte Carlo simulations. Here, numerous price paths for the underlying are simulated, and the option price can then be calculated as the expected payoff based on these paths. This introduces a computational bottleneck when model parameters are calibrated to a given implied volatility surface. The calibration procedure is a necessary step when one wants to use a model to price options, here model parameters are calibrated to obtain the best fit to the observed implied volatility surface. Since there is no closed-form solution to the calibration problem, a numerical search for the optimal parameters must be undertaken. For every update of the parameters, a new Monte Carlo simulation must be run to calculate option prices, which is where the computational bottleneck arises. For practitioners, this bottleneck can make the use of more sophisticated models infeasible, due to the time-consuming nature of the calibration procedure. Fortunately, for many of the new models, the bottleneck was resolved by the introduction of Fourier pricing methods like (Carr & Madan, 1999) and (Lewis, 2001). Fourier pricing allows for fast computation of model prices using Fourier inversion methods. This means that the Monte Carlo simulation can be replaced in the calibration scheme, resulting in a significant speedup.

In recent literature, a new class of models has emerged, namely, the rough stochastic volatility models. These models are continuous path stochastic volatility models, in which the instantaneous volatility is governed by a rough stochastic process. A rough stochastic process has Hölder regularity smaller than that of the Brownian motion (Bayer et al., 2019), like the fractional Brownian motion with Hurst parameter < 0.5. The strengths of this new class of models have been extensively documented in the literature. The highly influential paper by (Gatheral et al., 2014) finds that the log realized volatility for multiple indexes can be modeled by a rough volatility model with Hölder regularities between 0.008 and 0.2. (Bennedsen et al., 2021) examine almost two thousand US equities using their Brownian semi-stationary model (Bennedsen et al., 2017) and find overwhelming evidence of roughness. Using a GMM approach on high-frequency data for multiple indexes (Bolko et al., 2020) estimate a Hurst value of 0.05 for the integrated variance. Rough volatility models have also been shown to generate a better fit of the implied volatility surface, especially the short maturity at the money skew (Alòs et al., 2007; Bayer et al., 2015, 2018; Gatheral et al., 2014).

The main drawback, however, of rough volatility models is their non Markovianity which means Fourier methods cannot be used for option pricing (Bayer et al., 2019). This means that Monte Carlo methods must be used for pricing, which causes large computational bottlenecks, making rough volatility infeasible to use in practice. To overcome this issue, a recent trend in the literature seeks to bypass the Monte Carlo simulation by approximating the unobserved function that maps model parameters to prices. Neural networks have been used successfully to approximate this mapping function, examples include (Bayer et al., 2019; Bayer & Stemper, 2018; Dimitroff et al., 2018; Horvath et al., 2021; Rosenbaum & Zhang, 2021). By using a neural network to approximate option prices, the Monte Carlo step of the calibration routine can be replaced by a single evaluation of the neural network. The speed of calibration is then significantly reduced, which is shown in this thesis as calibration was reduced to as little as half a second. The justification for using neural networks for this part comes from the universal approximation theorem of (Hornik et al., 1989), which states

> *"Standard multilayer feedforward networks are capable of approximating any measurable function to any desired degree of accuracy."*

Using a neural network for option calibration consists of an online and offline part. The offline part is where the neural network is trained to learn the desired mapping function. Here, data is simulated from the chosen volatility specification, which is then used to train the network through the method of backpropagation. This can be a slow process as simulations must be performed using Monte Carlo, the benefit however is that it need only be done once. The online part then consists of using the trained network to calibrate model parameters using a numerical optimization procedure.

Before training the neural network, the user must decide on the network architecture. This includes the number of neurons, type and number of layers, activation function, and loss function. In the literature two types of architectures have mainly been considered, resulting in two approaches to calibration. The one-step approach of (Bayer et al., 2019; Dimitroff et al., 2018; Stone, 2020) and the two-step approach of (Bayer & Stemper, 2018; Hernandez, 2016) and (Horvath et al., 2021).

The one-step approach attempts to solve the calibration problem directly by predicting parameters in a single step. This is done by training a convolutional neural network (CNN) to predict model parameters given the observed implied volatility surface. CNNs are used for the one-step approach due to their well-known ability for image classification, for an in-depth review of CNNs image classification capabilities see (Rawat & Wang, 2017). In an option calibration setting, the implied volatility surface is inputted as an image due to its two dimensions of strike and maturity. The classification problem then consists of finding the parameters that are most likely to result in the observed implied volatility surface.

The two-step approach splits the calibration procedure into two steps. The first step is training a feedforward neural network to approximate the function that maps model parameters to option prices. Once the network, is trained it can practically instantaneously predict prices given a set of parameters. Step two consists of replacing the slow Monte Carlo simulations of the calibration algorithm with the trained neural network. Calibration is then done using a numerical optimization routine to find the set of parameters that minimizes the distance between observed and predicted call prices. In the literature, two variants of the two-step approach are employed, each with a different choice of the output from the neural network.

(Bayer & Stemper, 2018) use a piece-wise approach where the neural network predicts a single call price. The input of the network is then extended to include the strike and maturity of the call the network should predict. The network of (Horvath et al., 2021) instead predicts a grid of call prices for a specified number of strikes and maturities. In a collaborative paper (Bayer et al., 2019) they discusses the advantages and disadvantages of these two variants.

Where the previous literature showcases the possibility of using neural networks in an option calibration setting, there is a gap about the robustness of the different approaches. There is still room for a better understanding of how well the networks generalize in different settings. This thesis attempts to fill this gap by taking a deeper dive into the robustness of the different approaches.

The goal is to identify how the neural networks perform in various settings. The central questions examined are, how do parameter values affect network accuracy? How do the networks perform in an extrapolation setting where model parameters exceed the values of the training set? How does regularization affect model performance? Do the answers to these questions differ based on the calibration approach and chosen volatility specification? To answer these questions, four volatility specifications are considered, three of which are from the rough volatility family. The rough volatility models are rFSV of (Gatheral et al., 2014), rough Bergomi of (Bayer et al., 2015), and the rough Heston of (Euch & Rosenbaum, 2017). The traditional Heston model is also implemented to showcase that the neural network approach can be used for any model and is not restricted to rough volatility. For each of the four volatility models, neural networks are trained following three approaches (Bayer & Stemper, 2018; Dimitroff et al., 2018) and (Horvath et al., 2021). The different neural networks are initially tested on data simulated from the volatility specification they were trained upon. By using simulated data, insight is gained into how well the networks have learned to approximate the unobserved mapping function. Since the training and test data come from the same distribution, any issues can be isolated to how well the approximation has converged as no errors should come from having a misspecified model.

Once the network performance on simulated data has been established, a practical application of parameter calibration is performed using the observed call prices from SPX. Here, important insights are gained into how the different approaches compare from a practical standpoint. A need for restricting parameters of the calibration step is identified to ensure that predicted volatility curves are well behaved. This highlights an implementation issue with the one-step approach as it is impossible to allow for parameter restrictions after training.

The rest of this thesis is structured as follows. Section 2 introduces the relevant theory of rough volatility models along with the four volatility specifications and how one might simulate data from them. The neural networks for the one- and two-step approaches are introduced along with the backpropagation algorithm and various regularization techniques. Lastly, the numerical optimization algorithm used for calibration is explained along with the method of calibration. Section 3 presents the implementation details for data simulation, network training, model evaluation, and SPX calibration. Section 4 presents the results of the different approaches and how they compare to both simulated data and observed data. Section 5 concludes the main findings of this thesis and section 6 examines possibilities for future research. The Appendix includes an extended set of figures discussed in section 4.

2. Theory

This section highlights the relevant theory used in this paper. The first section explains what constitutes rough volatility and how rough volatility models differ from classic volatility models. Afterwards the different volatility specifications considered in this thesis are explained along with how data can be simulated from these models.

The second section explains the inner workings of the different neural networks used for the one- and two-step approach. Lastly, the third section explains how numerical optimization can be implemented.

2.1 Rough volatility

Rough volatility models are continuous stochastic processes where the autocovariance function follows the asymptotic relationship (Bennedsen et al., 2017).

$$1 - \rho(h) \sim c|h|^{2\alpha+1}, \qquad h \in \mathbb{R},$$

Where $\rho(h)$ is the autocovariance function, c is a positive constant and $\alpha \in (-0.5, 0.5)$.

Rough volatility models have typically been modelled by using the fractional Brownian motion (fBM) introduced by (Mandelbrot & Van Ness, 1968). The fBM is an extension of the original Brownian motion which allows for long and short memory. The fBM is a centered gaussian process where its autocovariance function depends on the so called the Hurst parameter $H \in (0,1)$ as seen below (Alòs & León, 2021).

$$E(B_t^H B_s^H) = \frac{1}{2}(s^{2H} + t^{2H} - |t-s|^{2H}), \qquad s, t \in [0, T].$$

The regular Brownian motion is a special case of the fBM with $H = 1/2$. The fBM exhibits long memory when $H > 1/2$, and becomes rough when $H < 1/2$. fBM's can furthermore be represented by the Mandelbrot-van Ness representation (El Euch & Rosenbaum, 2019)

$$W_t^H = \frac{1}{\Gamma(H+0.5)} \int_{\infty}^{0} ((t-s)^{H-0.5} - (-s)^{H-0.5}) dW_s + \frac{1}{\Gamma(H+0.5)} \int_0^t (t-s)^{H-0.5} dW_s.$$

The rough volatility models implemented in this paper are rough Heston (Euch & Rosenbaum, 2017), rough Bergomi (Bayer et al., 2015) and the rough FSV from (Gatheral et al., 2018). Since the deep learning approach to option calibration is model agnostic, the classic Heston stochastic volatility models are implemented alongside the rough volatility models and serve as a comparison and proof of concept. The sections below describe each of the considered models and how to simulate options prices from them.

2.1.1 Heston

The classic Heston stochastic volatility model is given by the formulas below

$$dS_t = S_t \sqrt{V_t} dW_t,$$

$$dV_t = \lambda(\theta - V_t)dt + \nu \sqrt{V_t} dB_t.$$

Here S_t is the stock price at time t, V_t is the variance at time t, W_t and B_t are two correlated Brownian motions with correlation $\rho \in [-1,1]$. $\lambda > 0$ measures the rate of mean reversion of the variance, $\theta > 0$ is the long run level of variance and $\nu > 0$ is the volatility of volatility.

Simulation of the Heston data was done by the inverse Fourier transform method using the Lewis approach. The Heston characteristic function is defined as (Kokholm & Stisen, 2015)

$$\Phi^{SV}(u,T) = e^{C(u,T)\theta + D(u,T)V_0},$$

where

$$C(u,T) = \frac{\kappa}{\eta^2}\left((m-d)T - 2\log\frac{1-ge^{-dT}}{1-g}\right),$$
$$D(u,T) = \frac{1}{\eta^2}(m-d)\frac{1-e^{-dT}}{1-ge^{-dT}},$$

$$m = \kappa - \eta\rho u i, \qquad d = \sqrt{m^2 + \eta^2(ui + u^2)}, \qquad g = \frac{m-d}{m+d}.$$

The Lewis formula for computing call prices is

$$C(S_0,K,T) = S - \frac{\sqrt{SK}}{\pi}\int_0^\infty \frac{\mathcal{R}\left(e^{iuk}\Phi^{SV}(u-0.5i,T)\right)}{u^2+0.25}du,$$
$$k = \log(S/K).$$

Where S_0 is the current stock price, K is the strike and T the maturity.

2.1.2 Rough Heston

The rough Heston (rHeston) model comes as a natural way of extending the Heston model to the class of rough volatility, the dynamics of rHeston is given by (Euch & Rosenbaum, 2017)

$$dS_t = S_t\sqrt{V_t}dW_t,$$

$$V_t = V_0 + \frac{1}{\Gamma(\alpha)}\int_0^t (t-u)^{\alpha-1}\lambda(\theta - V_u)du + \frac{1}{\Gamma(\alpha)}\int_0^t (t-u)^{\alpha-1}\nu\sqrt{V_u}dB_u.$$

Here $\lambda, \theta, V_0, S_0$ and ν are positive parameters, $\alpha \in (0.5,1)$ is the fractional order and relates to the Hurst parameter in the following way $H = \alpha - 0.5$. $W = \rho B + \sqrt{1-\rho^2}B^{\perp}$ where $(B, B^{\perp})$ is a two dimensional Brownian motion and $\rho \in [-1,1]$ is a correlation coefficient. From the above equation one can see that setting $\alpha = 1$ yields the classic Heston model.

Due to the Markovianity of the classic Heston model there exists of a closed form solution for the characteristic function, which makes option pricing fast and easy to implement. The rHeston model or any other rough volatility model however is not Markovian due to the fBM and therefore does not have a closed form characteristic function. Fortunately for the rHeston model (El Euch & Rosenbaum, 2019) shows the existence of a semi closed form solution for the characteristic function, which is given below,

$$\phi_{rHeston} = \exp\big(\theta\lambda I^1 h(a,t) + V_0 I^{1-\alpha} h(a,t)\big).$$

Here

$$I^r f(t) = \frac{1}{\Gamma(r)} \int_0^t (t-s)^{r-1} f(s)ds\,,$$
$$D^r f(t) = \frac{1}{\Gamma(1-r)} \frac{d}{dt} \int_0^t (t-s)^{-r} f(s)ds.$$

The function $h(a,t)$ in the characteristic function is the function that solves the following fractional Ricatti equation

$$D^\alpha h(a,t) = \frac{1}{2}(-a^2 - ia) + \lambda(ia\rho\nu - 1)h(a,s) + \frac{(\lambda\nu)^2}{2} h^2(a,s),$$
$$I^{(1-\alpha)} h(a,0) = 0.$$

A common approach for solving the fractional Ricatti equation, and the one used in the (El Euch & Rosenbaum, 2019), is the Adams scheme which assumes the solution for $h(a,t)$ resembles the following Volterra equation

$$h(a,t) = \frac{1}{\Gamma(\alpha)} \int_0^t (t-s)^{\alpha-1} F\big(a, h(a,s)\big)ds,$$

$$F(a,x) = \frac{1}{2}(-a^2 - ia) + \lambda(ia\rho\nu - 1)x + \frac{(\lambda\nu)^2}{2} x^2.$$

By using a discrete time grid t_k with grid size Δ: $(t_0, t_{0+\Delta}, \dots, t_{0+k\Delta})$ one can solve for $h(a,t)$ using the following scheme

$$\hat{h}(a, t_{k+1}) = \sum_{0\le j\le k} a_{j,k+1} F\left(a, \hat{h}(a,t_j)\right) + a_{k+1,k+1} F\left(a, \widehat{h_P}(a,t_j)\right),$$

$$\hat{h}(a,0) = 0.$$

Where a's are weights defined as

$$a_{0,k+1} = \frac{\Delta^\alpha}{\Gamma(\alpha+2)}(k^{\alpha+1} - (k-\alpha)(k+1)^\alpha),$$

$$a_{j,k+1} = \frac{\Delta^\alpha}{\Gamma(\alpha+2)}((k-j+2)^{\alpha+1} + (k-j)^{\alpha+1} - 2(k-j+1)^{\alpha+1},$$

$$a_{k+1,k+1} = \frac{\Delta^\alpha}{\Gamma(\alpha+2)}.$$

And $\widehat{h_P}(a, t_{k+1})$ is a pre-estimator of $\hat{h}(a, t_{k+1})$ and defined as

$$\widehat{h_P}(a, t_{k+1}) = \sum_{0\le j\le k} b_{j,k+1} F\left(a, \hat{h}(a,t_j)\right),$$

$$b_{j,k+1} = \frac{\Delta^\alpha}{\Gamma(\alpha+1)}((k-j+1)^\alpha - (k-j)^\alpha).$$

Simulation from the rough Heston model was implemented using the GitHub code by (Roemer, 2021).

2.1.3 Rough Bergomi

The rough Bergomi model (rBergomi) by (Bayer et al., 2015) has the following dynamics for the stock price S_t

$$S_t = \exp\left(\int_0^t \sqrt{V_u} dB_u - \frac{1}{2}\int_0^t V_u du\right),$$
$$B_u = \rho W_u^1 + \sqrt{1-\rho^2} W_u^2.$$

The dynamics for the variance process V_t is defined as

$$V_t = \xi \exp\left(\eta Y_t^{H-0.5} - \frac{\eta^2}{2} t^{2H}\right).$$

Here $Y_t^{H-0.5}$ is a volterra process defined as

$$Y_t^{H-0.5} = \sqrt{2H} \int_0^t (t-u)^{H-0.5} dW_u^1.$$

Here $\rho \in [-1,1]$ is the correlation coefficient, $\xi, \eta > 0$ are postive parameters, W_u^1 and W_u^2 are two uncorrelated brownian motions and H is the hurst parameter.

The simulation of the rough Bergomi prices in this thesis were done using the GitHub code by (McCrickerd, 2017). This code implements the hybrid scheme from (Bennedsen et al., 2015) to simulate the volterra process $Y_t^{H-0.5}$. Due to time constriants, the details for the hybrid scheme could not be further elaborated.

2.1.4 Rough FSV

The rough fractional stochastic volatility model (rFSV) from (Gatheral et al., 2018) is a stochastic volatility model where the log volatility is driven by a fractional Brownian motion (fBM). Its dynamics are given below

$$dS_t = S_t \sqrt{V_t} dW_t,$$

$$dX_t = \nu dW_t^H - \alpha(X_t - m)dt,$$

$$\sqrt{V_t} = \exp(X_t).$$

Here $a, \nu > 0$ are postive constants, $m \in \mathbb{R}$ denotes the average value of X_t and H is the hurst parameter.

Simulation of rFSV can be done by simulating paths from the fBM and a regular Brownian motion. In a for loop one can then compute dS_t and dX_t which is used to update S_t and X_t. In practice one must choose the number of points to simulate in one year, as well as the number of paths to simulate. In this paper 251 points were simulated per year with a total of 10.000

paths. The initial level of volatility is added as fifth parameters which means the initial value for X is determined as

$$X_0 = \log\left(\sqrt{V_0}\right).$$

Paths from the fBM were simulated using the freely available python library *stochastic*.

2.2 Neural Networks and calibration

Neural networks are known as universal approximators and are useful for learning the relationship between input pairs x and outputs y (Goodfellow et al., 2016). Being fully nonparametric, a neural network makes no assumptions about the distribution or connection of its inputs and outputs. Neural networks are then useful for approximating any function

$$y = f(x),$$

where the true functional form is unknown. Given x, the output of a neural network is fully determined by its weights and biases, which together can be denoted as θ. The neural network can therefore be expressed as the approximation of $f(x)$ defined as

$$y = \hat{f}(x, \theta).$$

The two most common approaches for option calibrating with neural networks are in the literature referred to as the one- and two-step approaches. Both approaches train a neural network to approximate the unknown relationship between model parameters and options prices. The difference in the two approaches lies in how the objectives of the networks are defined. To formalize this first define the unobserved pricing function as the mapping between model parameters to call prices

$$P(\theta) \coloneqq \mathcal{M}(\theta) \to R^m.$$

Here $R^m, m \in \mathbb{N}$ defines the number of option prices the pricing function outputs and θ denotes the model parameters. The pricing function is therefore fully defined by the set of parameters θ, which are specific to the chosen volatility specification. As stated in the introduction, two variants of the two-step approach are considered in this thesis. One where the network predicts a single price $m = 1$, which is referred to as the piece-wise approach (pNN) and used in (Bayer & Stemper, 2018). For the second variant, a grid of call prices is predicted for a combination of strikes K and maturities T, whereby $m = KT$. The second variant is referred to as the grid-wise approach (gNN) and implemented in (Horvath et al., 2021).

Unless we are considering the Black and Scholes model, the solution to the pricing function does not exist in closed form, one must therefore approximate this mapping. This is the objective of the neural network trained following the two-step approach. Formally, the trained two-step neural network is defined as a function $\hat{f}^{2Step}(\theta)$ that approximates the unobserved pricing function

$$\hat{f}^{2Step}(\theta) \coloneqq \hat{P}(\theta) \to R^m.$$

The training of the neural network is the first step of the two-step approach, the second step consists of a calibration procedure. The calibration procedure finds the optimal parameters $\hat{\theta}$

which minimizes the distance between predicted and observed prices. The distance between prices is measured by some loss function L which is user defined. The objective of the calibration step can be defined as

$$\hat{\theta} = \underset{\theta}{\operatorname{argmin}} L(f^{2Step}(\theta) - P(\theta)).$$

The one-step approach implemented in (Bayer et al., 2019; Dimitroff et al., 2018) bypasses the calibration step and incorporates it directly into the neural network. The one-step approach assumes the existence of a parameter map that is defined as the inverse of the pricing function

$$\theta(P) \coloneqq \mathcal{M}^{-1}(P) \to R^n.$$

This mapping takes prices P as input and outputs the model parameters θ, that resulted in the observed prices, as in $\mathcal{M}\big(\theta(P)\big) = P$. The output of the parameter map is of dimension n which equals the number of parameters for the considered volatility specification. Like the pricing map, this parameter mapping does not have a closed form solution. The neural networks following the one-step approach are therefore trained to approximate this. This trained approximation then defines the function $\hat{f}^{1Step}(P)$

$$\hat{f}^{1Step}(P) \coloneqq \hat{\theta}(P) \to R^n.$$

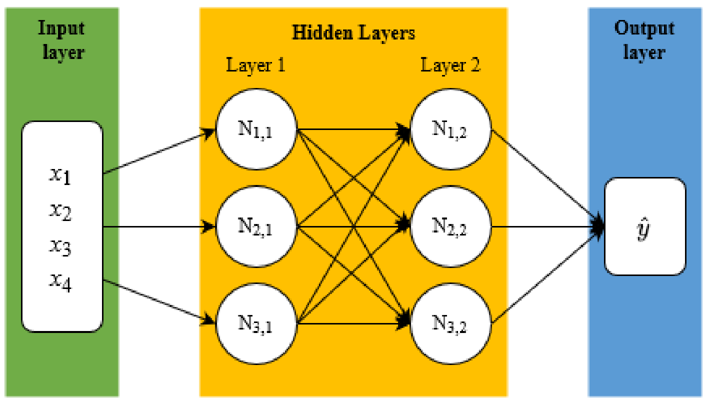
The following sections presents how neural networks can be implemented to obtain the desired approximations for each of the two approaches.

2.2.1 MLPs and the two-step approach

For the two-step approach the choice of network is the multilayer perceptron (MLP), which is the one used in (Bayer & Stemper, 2018; Hernandez, 2016; Horvath et al., 2021). The use of MLPs are motivated by the universal approximation theorem (Theorem 2.2) of (Hornik et al., 1989). Here Hornik states MLPs can approximate any measurable function arbitrarily well. The rest of this section explains how MLPs can be implemented following the book of (Goodfellow et al., 2016).

To understand how MLPs transform inputs to outputs, the concepts of layers and neurons are introduced. Figure 1 gives a visual representation of how MLPs are structured. This MLP takes four parameters $x_1, \dots, x_4$ as input and outputs a single value $\hat{y}$, it has two hidden layers each with three neurons. MLPs are often referred to as feedforward fully connected neural networks. The *feedforward* part represents that inputs are based only on the previous layer, and *fully connected* states all outputs of the previous layer are used as input. Generally, MLPs can have any number of inputs, outputs, layers and neurons, the number of neurons can also be different in each layer.

Figure 1: MLP architecture



This figure shows a graphical representation of the Multilayer Perceptron (MLP)

Each layer of the MLP represents a function that takes a vector of inputs and outputs a vector with size equal to its number of neurons. Generally, the output of a layer can be defined as

$$y^{(l)} = f^{(l)}\left(y^{(l-1)}\right),$$

where $f^{(l)}$ represents the functional form of layer l and $y^{(l-1)}$ is the output of the previous layer. By iteratively substitution for $y^{(l-1)}$ and using $y^{(0)} = X = [x_1, x_2, x_3, x_4]$ the final output of the neural network in figure 1 is defined as

$$\hat{y} = f^{(2)}\left(f^{(1)}(X)\right).$$

For any layer, the function $f^{(l)}$ is determined by its neurons. Neurons are what enables the network to learn features from the data and in turn what allows it to approximate the unobserved mapping of inputs to outputs. There exist different neurons, but the one used in an MLP is the perceptron. A perceptron is a neuron that consists of two parts: (1) a summation function, and (2) an activation function. The summation function is a weighted sum of the inputs for the neuron plus a bias term. For any neuron, in any layer, the summation function is given as

$$z_i^{(l)} = \boldsymbol{w}_i^{(l)}\boldsymbol{y}^{(l-1)} + b_i^{(l)}.$$

Here $z_i^{(l)}$ is the output of the summation function for neuron i in layer l. Define the number of neurons in layer l as N_l and the total number of layers as L. Then, $\boldsymbol{w}_i^{(l)}$ is a vector containing neuron $i's$ weights, defined as

$$\boldsymbol{w}_i^{(l)} = \left[w_{i,1}^{(l)}, \dots, w_{i,N_{l-1}}^{(l)}\right].$$

$\boldsymbol{y}^{(l-1)}$ is the vector of outputs from the previous layer and the inputs of all neurons in layer l.

$$\boldsymbol{y}^{(l-1)} = \left[y_i^{(l-1)}, \dots, y_{N_{l-1}}^{(l-1)}\right]'.$$

Lastly, $b_i^{(l)}$ is the bias of neuron i which loosely corresponds to the intercept in an OLS regression. The second part of the neuron takes the output of the summation function and passes it to the activation function. This activation function is specified by the user and denoted as

$\sigma(x)$. The activation function used for the two-step approach is the exponential linear unit (ELU). The ELU function takes a user specified parameter α, and is defined as follows

$$\sigma_{ELU}(x) = \alpha(e^x - 1).$$

For the two-step approach another activation function is used called the leaky rectified linear unit (Leaky ReLU), which is a modification of the commonly used ReLU. Both are stated below.

$$\sigma_{ReLU}(x) = \max(0, x),$$

$$\sigma_{Leaky\ ReLU}(x) = \begin{Bmatrix} x & if\ x > 0 \\ 0.01x & else \end{Bmatrix}.$$

With the activation function defined, the full output of a neuron is then given as

$$y_i^{(l)} = \sigma\left(z_i^{(l)}\right) = \sigma\left(\boldsymbol{w}_i^{(l)}\boldsymbol{y}^{(l-1)} + b_i^{(l)}\right).$$

The above equation specifies the behavior of each individual neuron. Stacking the outputs of all neurons in a layer therefore determines the output of $f^{(l)}\left(y^{(l-1)}\right)$.

Given a set of inputs, the output of a neural network is solely determined by its weights and biases, which can be jointly denoted θ. Initially, the values of θ are drawn at random, and the neural network must then be trained to fit the dataset. For a dataset consisting of $\{X_i, y_i\}_{i=1}^N$ pairs, the objective of the training can then be defined as finding the optimal values of θ. To determine the optimal set of θ, a user must specify function L, whereby the training objective becomes

$$\hat{\theta} = \underset{\theta}{\operatorname{argmin}}\, L(\hat{y}_i, y_i, \theta).$$

The loss function depends on the observed values y_i and the predicted values $\hat{y}_i$ which depend on θ. The loss function might also be a function of θ itself, as is the case of weight decay. Weight decay is a regularization technique, which is further discussed in section 2.2.3.

The training process then consists of iteratively cycling through two stages known as forward propagation and backpropagation. Note, that the training process is usually referred to as the backpropagation algorithm, even though it also includes forward propagation. During the forward propagation, the network is used to calculate the predictions $\hat{y}$ and the loss for the entire sample. Then, the role of backpropagation is to adjust the weights and biases of the network to reduce the loss function. This is done by computing the gradient of the loss function wrt. θ.

$$\nabla_\theta = \frac{\partial L(\hat{y}, y, \theta)}{\partial \theta}.$$

The gradient is defined as ∇_θ and can be computed using the chain rule, for detail see (Du & Swamy, 2019). Once the gradient is obtained the parameters are updated using the update rule

$$\Delta\theta_{t+1} = \theta_{t+1} - \theta_t = -\eta\nabla_{\theta_t}.$$

Here t denotes the current iteration of the training loop and $\eta > 0$ is a parameter called the learning rate. The learning rate is a crucial parameter as it determines how fast the network learns and must be specified by the user. Too high a value means the network might not converge, while too low a value means training becomes slower and might get stuck in local minima. One iteration of training consists of one run of the forward- and back-propagation and is usually referred to as an epoch. In the simplest form, the training process continues until the number of epochs reaches a value specified by the user.

To update parameters in the original backpropagation algorithm, the loss and gradient must be computed for all observations in the sample. If the number of observations is large, the algorithm must perform many calculations for each update step, which can be time-consuming. For this reason, the backpropagation algorithm is usually modified to run in what is referred to as mini-batches or just batches. A batch is a set number of observations from the dataset, which size is specified by a parameter called batch size. The batch size can be any number less than the total number of observations and is usually set as a multiple of 8 for computational reasons. Often, one will find batch sizes of 8, 16, 32, 64, 128, and so on in the literature. The difference when training with mini-batches is that only a set of observations equal to the batch size is used to compute the gradient. Since the batch size is just a fraction of the full dataset, an epoch represents the number of iterations it takes before all observations have been used for training. This means that if the dataset consists of 1000 observations, it takes 10 iterations with a batch size of 100 before one epoch has elapsed. For practical implementations, the observations for the batch are sampled without replacement from the full dataset until all observations have been used. Using batches has the advantage that training speed increases as fewer computations are needed to calculate $\Delta\theta_{t+1}$, it does, however, produce a noisier estimate of ∇_{θ_t}. The backpropagation algorithm using mini-batches can be summarized by algorithm 1.

Algorithm 1: Backpropagation with mini-batches.
Initiate θ using random weights and biases. The algorithm loops over batches until the maximum number of epochs has been reached

```
Set: $\eta$                                              learning rate
Set: $L(\hat{y}, y, \theta)$                             choose loss function
Set: $batch\_size$                                       observations in each batch
Set: $num\_epochs$                                       maximum number of epochs
Set: $t = 1$                                             iteration counter
for $epoch$ in $num\_epochs$:
    for $batch$ in $sample\ size\ /\ batch\ size$:
        $\hat{y}_{batch} = f(x_{batch})$                 compute predictions using batch
        $\nabla_{\theta_t}^{batch} = \partial L(\hat{y}_{batch}, y_{batch}, \theta_t)/\partial\theta_t$   compute gradients
        $\Delta\theta_{t+1} = -\eta\nabla_{\theta_t}^{batch}$   update weights
        $t = t + 1$                                      update iteration counter
```

The backpropagation algorithm is just one algorithm in the class of gradient descent optimizers. Gradient descent optimizers derive their name from using the gradient to update weights and biases. When backpropagation is performed using mini-batches the algorithm is usually referred to as stochastic gradient descent.

Algorithm 1 can be extended in numerous ways, a common extension is adaptive learning rate. Here, the learning rate changes during training such that it is initially high and reduces as the loss function converges to a minima. Adaptive learning rate is useful as it can increase convergence speed and reduce the risk of getting stuck in local minima (Du & Swamy, 2019). Another common extension is momentum, which can best be understood by imagining a ball rolling down a hill. As the curvature of the hill changes, the ball still carries momentum from its previous direction. To implement momentum, the gradient is usually calculated as a running mean using the previous gradients. The strength of using momentum is similar to adaptive learning rate improved convergence speed and reduced risk of local minima (Goodfellow et al., 2016). The algorithm used in this paper is Adam (Kingma & Ba, 2017), which is a stochastic gradient descent optimizer and features both momentum and a form of adaptive learning rate. Algorithm 2 outlines the Adam algorithm along with its implementation for batch learning.

Algorithm 2: Adam

Initiate weights and biases of the network using random draws from e.g. standard normal distribution

```
Set: $\eta$                                              learning rate
Set: $\beta_1, \beta_2 \in [0,1)$                        decay rates for moment estimates
Set: $L(\hat{y}, y, \theta)$                             loss function
Set: $m_0 = 0$                                           initial value for 1st moment
Set: $v_0 = 0$                                           initial value for 2nd moment
Set: $t = 1$
for $epoch$ in $num\_epoch$:
  for $batch$ in $sample\ size\ /\ batch\ size$:
    $\hat{y}_{batch} = f(x_{batch})$                     compute predictions using batch
    $\nabla_{\theta}^{batch} = \partial L(\hat{y}_{batch}, y_{batch}, \theta)/\partial\theta$   compute gradients
    $m_t = \beta_1 m_{t-1} + (1-\beta_1)\nabla_{\theta}^{batch}$         update 1st moment estimate
    $v_t = \beta_2 v_{t-1} + (1-\beta_2)\left(\nabla_{\theta}^{batch}\right)^2$   update 2nd moment estimate
    $\widehat{m_t} = m_t/(1-\beta_1^t)$                  bias correct 1st moment estimate
    $\widehat{v_t} = v_t/(1-\beta_2^t)$                  bias correct 2nd moment estimate
    $\theta_t = \theta_{t-1} - \eta\widehat{m_t}/(\sqrt{\widehat{v_t}} + \epsilon)$   update parameters
    $t = t + 1$                                          update iteration counter
```

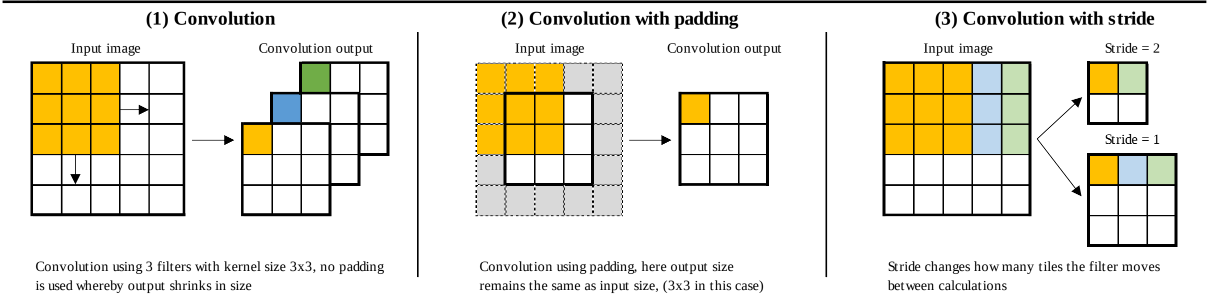
The Adam algorithm uses exponential moving averages of the gradient denoted m_t and the square of the gradient denoted v_t. Here t is the iteration of the algorithm and β_1, β_2 are the parameters determining the exponential decay rates. Since the initial values for m_0 and v_0 are zero, a bias correction is done, derivation of the bias correction can be found in (Kingma & Ba, 2017).

2.2.2 CNNs and the one-step approach

For the one-step approach, the neural network employed is the convolutional neural network (CNN), this type of network was used for option calibration in (Bayer et al., 2019; Dimitroff et al., 2018; Stone, 2020). CNNs have seen great success in image classification due to their ability to extract features from multidimensional inputs. For a review of CNNs application for image classification see (Rawat & Wang, 2017). Since call prices have a two-dimensional structure with strike and maturity, CNNs are especially suited for the one-step approach as they can learn features based on the call prices relative to their neighbors on the grid. This means CNNs learn from the shape of the call surface, which is something an MLP cannot do as it does not know the position of the calls relative to each other. To understand how a CNN works, three concepts must be introduced, convolution, pooling, and flattening. The rest of this section introduces the three concepts following the book of (Goodfellow et al., 2016).

Convolution is the process of extracting features from the input, which are then used for classification or, in this case, predicting model parameters. Convolution works by applying a filter to the input, which is the equivalent to the summation function from the MLP. Different from the MLP is the convolutional filter used has the same weights for the entire image, which is shown in figure 2.

Figure 2: Convolution


(1) Convolution
Input image
Convolution output
Convolution using 3 filters with kernel size 3x3, no padding is used whereby output shrinks in size
(2) Convolution with padding
Input image
Convolution output
Convolution using padding, here output size remains the same as input size, (3x3 in this case)
(3) Convolution with stride
Input image
Stride = 2
Stride = 1
Stride changes how many tiles the filter moves between calculations

This figure displays a graphical representation of (1) the convolutional filter of a CNN, (2) Convolution with paddning and (3) Convolution with stride.

Part (1) of figure 2 represents how the convolutional filter works. This filter has a kernel of size 3x3 and is moved across the input until it has covered the whole of the image. The convolution equation can be stated as

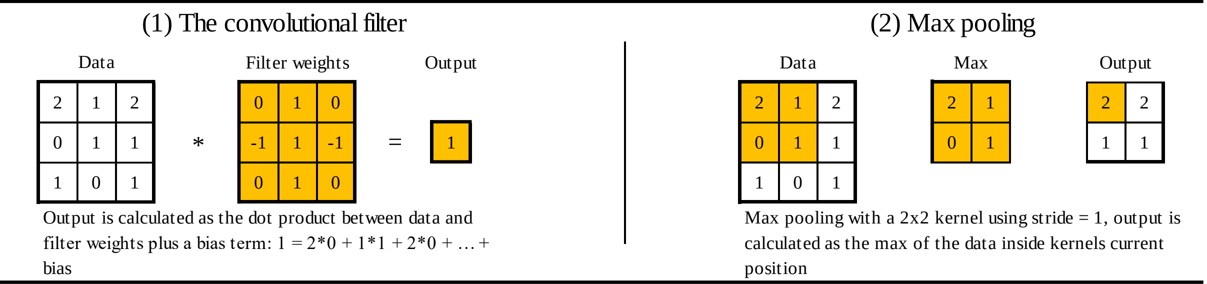
$$S(i,j) = \sum_{m} \sum_{n} I(i-m, j-n)K(m,n).$$

Here I is the input image, K is the kernel and $S(i,j)$ is the value of the output image for row $i \in [1,2,\ldots,m]$ and column $j \in [1,2,\ldots,n]$. See figure 3, part (1) for a visual explanation of how $S(i,j)$ is calculated.

A convolutional layer typically implements multiple filters, in figure 1, three filters are used, each with its own kernel. When implementing a convolutional layer, the user must decide on several parameters. First, the size of the filter and how many filters should be used, second how the filter should react to the edges of the input. Figure 1 part (2) illustrates how padding can be added to the input such that its size is not reduced when using the filter. Padding can be implemented by adding zeros to the borders of the image, whereby $S(i,j)$ is computed as before. There can be multiple reasons why the user might want to include padding. With zero-padding there is a natural maximum to the number of convolutional layers that can be implemented as the image shrinks in size after each convolution. Adding padding to the convolutional layers, therefore, allows the user to create even deeper networks. Another reason to include padding might be if the border of the image contains crucial information for the network. With zero-padding, the outer values in the image are, to some extent, lost as the image shrinks.

The final choice for the convolutional layer is the choice of stride. Stride determines how many steps the filter should take between each computation, as shown in figure 1 part (3). Increasing the stride means the output of the convolutional layer decreases in size. This can be advantageous if the goal is to reduce the number of parameters of the model.

Figure 3: The convolutional filter and max pooling


(1) The convolutional filter
Data
Filter weights
Output
2 1 2
0 1 1
1 0 1
*
0 1 0
-1 1 -1
0 1 0
=
1
Output is calculated as the dot product between data and filter weights plus a bias term: 1 = 2*0 + 1*1 + 2*0 + ... + bias
(2) Max pooling
Data
Max
Output
2 1 2
0 1 1
1 0 1
2 1
0 1
2 2
1 1
Max pooling with a 2x2 kernel using stride = 1, output is calculated as the max of the data inside kernels current position

This figure displays (1) How a sinlge output of the convolutional filter is calculated, and (2) How the output of a max pooling layer is calculated.

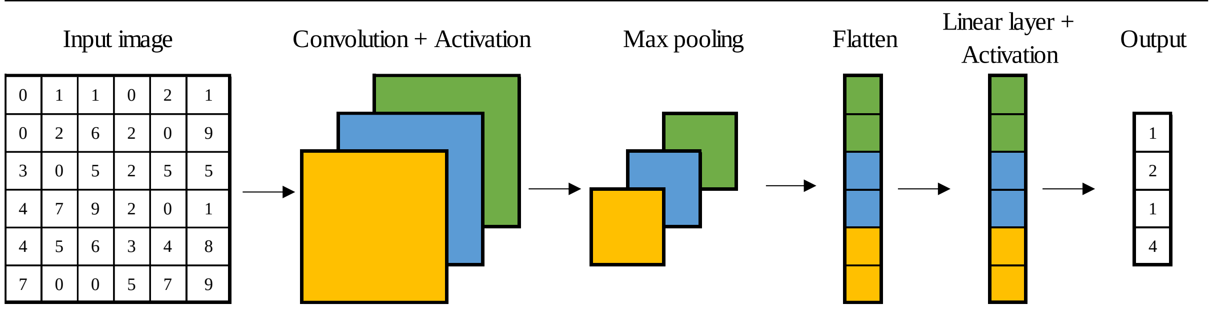
To get a better understanding of how the convolutional filter works figure 3 displays how the output of a filter is calculated. This filter as a 3x3 kernel and as figure 3 shows contain 9 weights plus a bias term. These weights are used to compute the output using the dot product between the weights and the data given the current position of the filter.

The second concept of CNNs is pooling, this operation plays a vital role in the performance of the network and is implemented after the convolutional layer. Part (2) of figure 3 displays a variant of pooling called max pooling, like convolution a filter slides across the data and computes some output given the data inside the filter. The output of the max-pooling layer is as the name suggests, calculated as the maximum of the currently observed data. Alternatively, the user can also use median or average pooling. To implement a pooling layer, the user must decide on which type of pooling to use, the stride, and the size of the kernel. The strength of the pooling operation is that it makes the network more robust by improving its invariance to small transformations. Invariance to small transformations means the network still understands the image even if it is slightly distorted or rotated, which improves the generalizability of the network. Besides making the network more robust, the pooling operation ideally also improves the network by throwing away irrelevant information whereby the computational overhead is reduced.

The final concept for building CNNs is an operation known as flattening, this operation simply changes the dimensionality of the data. In the concept of option pricing, the flattening process changes the data from two dimensions to a single-dimensional vector. After flattening, the now one-dimensional data is fed through an MLP until the final output is produced.

With the concepts for the CNN introduced figure 4 summarizes the structure of a CNN and how data flows through it. The Network takes a two-dimensional image as input, where the values in each cell represent the value of a given pixel in an image, or in the context of this thesis, the call price. The first layer of the CNN is the convolutional layer, where a set number of filters are used on the image, the output of the convolutional layer is typically fed through an activation function before the max-pooling layer. The final part flattens the images from the pooling layer to a one-dimensional vector, which is then fed through an MLP until the output is produced.

Figure 4: Architechure of a CNN



This figure displays the general architecture of a convolutional neural network (CNN)

In a broad sense, the role convolutional layer is to extract features from the two-dimensional output, when a feature is present the activation function outputs a large value. The pooling layer makes the output of the convolutional layer invariant to small transformations and removes noise from the image. The process of flattening changes the dimensionality of the data such that it can be understood by an MLP, the role of the MLP is then to predict the correct output given the features extracted through the convolutional process. To get an accurate prediction, the weights and biases of the convolutional filters and the neurons of the MLP part must be trained. Training of a CNN can be done similarly to an MLP using some variant of the backpropagation algorithm. In this thesis, the Adam algorithm is used for training the CNNs and is implemented as previously explained by algorithm 2.

2.2.3 Regularization

A common problem for neural networks is overfitting, which can be defined as the issue of the network fitting to the noise of the training set rather than learning the "true" relationship between inputs and outputs (Goodfellow et al., 2016). Overfitting occurs naturally as the network is simply instructed to minimize its loss function. A common method for examining whether the model is overfitting is the use of a validation set. The validation set acts as a test set as it is used only for testing performance not for training. The difference is that the validation set is used each epoch to evaluate model performance, this gives an approximation of how the performance of the network has evolved during its training. Figure 5 displays the usual pattern observed when a network starts overfitting. Until epoch nine both training and validation loss are reducing, if training is stopped before here it will result in an underfitted model. The model is underfitted since further training could reduce the error of the validation set. After epoch nine, training loss keeps reducing but validation loss increases. At this point, the network is overfitting and should be stopped to achieve the best performance on new data (generalizability).

Methods used to avoid overfitting are commonly referred to as regularization, those considered in this thesis are early stopping, weight decay, drop out and batch normalization, each of these methods is explained in the following part.

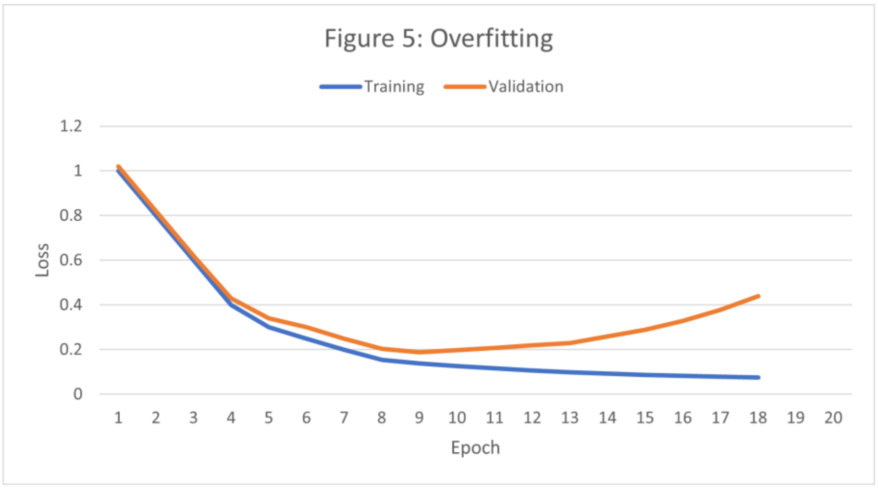


Illustration of overfitting, when validation loss starts to increase the network is said to be overffiting the data.

Early stopping is the simplest method to avoid overfitting and seeks to stop training when validation loss stops improving. This can be done by tracking the validation loss for each epoch, once the validation loss hasn't been improved for a chosen number of epochs the training stops. Early stopping is a simple and easy method to implement and works well against overfitting, the only parameter the user must choose is the number of epochs before the algorithm should stop. Setting this number too low increases the risk of underfitting as the training might stop before it reaches the optimal weights. Conversely, it should not be too high as the network might overfit before training is stopped (Goodfellow et al., 2016).

Weight decay from (Krogh & Hertz, 1991) penalizes the network for choosing larger weights, specifically, it adds a penalty term to the loss function, such as the penalized regressions Lasso and Ridge. The loss function for a network with weight decay is therefore of the form

$$L(\hat{y}, y, \theta) = E(\hat{y}, y) + \lambda \sum_{i=1}^{n} f(\theta_i).$$

Here $E(\hat{y}, y)$ is the chosen loss metric measuring the error between observed and predicted labels. $\lambda > 0$ determines the strength of the weight penalization, and $f(\theta_i)$ is a function determining how the size of weight θ_i is measured. The typical functional forms of $f(\theta_i)$ are θ_i^2 for L2 penalization and $|\theta_i|$ for L1 penalization. Weight decay also affects the gradient of the loss function and thereby restricts the update step in the backpropagation algorithm

$$\frac{\partial L(\hat{y}, y, \theta)}{\partial \theta_i} = \frac{\partial E(\hat{y}, y)}{\partial y} + \lambda \frac{\partial f(\theta_i)}{\partial \theta_i}.$$

The regularization effect of weight decays comes from finding the smallest vector of weights that solves the optimization problem, this makes the network less likely to fit the noise of the training data, which improve the network's generalizability.

Dropout is another powerful method to prevent the network from overfitting and uses random dropping of neurons in the network for regularization. For each forward propagation, neurons are randomly dropped meaning the output of the neuron is set to zero, the forward propagation is then defined by the following equations from (Srivastava et al., 2014).

$$\begin{aligned}
&r_i^{(l)} \sim Bernouilli\left(p^{(l)}\right),\\
&\widetilde{\boldsymbol{y}}^{(l)} = \boldsymbol{r}^{(l)} \cdot \boldsymbol{y}^{(l)},\\
&z_i^{(l)} = \boldsymbol{w}_i^{(l)} \widetilde{\boldsymbol{y}}^{(l)} + b_i^{(l)},\\
&y_i^{(l+1)} = \sigma\left(z_i^{(l)}\right).
\end{aligned}$$

From the Bernoulli distribution with probability $p^{(l)}$ the variable $r_i^{(l)}$ is drawn for each neuron in each layer. $r_i^{(l)}$ is then multiplied with the neuron's output, meaning if $r_i^{(l)} = 0$ the neuron's output is zero, the forward propagation then proceeds a previously explained but $\widetilde{\boldsymbol{y}}^{(l)}$ instead of $\boldsymbol{y}^{(l)}$. The probability $p^{(l)}$ is set for each layer, it is thus possible to implement dropout for a subset of the network's layers or with varying intensity.

Training with dropout can also be performed in batches, here the thinned network with dropped units is used for all observations in the batch. The gradients are then averaged across the batch and used for backpropagation. After training, the full network is used for testing without dropout in any layers, but with the outputs of each neuron scaled by $p^{(l)}$. By scaling the neurons with $p^{(l)}$ dropout effectively becomes an approximate form of ensemble learning, which is the idea of averaging results from different models to improve performance. That dropout represents ensemble learning is one of its key strengths and what allows it to reduce overfitting and thus improve generalizability (Hara et al., 2016). The regularization power of dropout also comes from the uncertainty imposed on which neurons are present, since neurons can be dropped at random the network cannot rely on single connections, as they are not guaranteed to be present. The network is therefore encouraged to spread out its weights and have neurons that perform well in many contexts, thereby improving generalizability and reducing overfitting (Goodfellow et al., 2016) .

Batch normalization is the final method considered in this paper and while not exactly a method used to improve regularization, it has been shown to improve the training speed and accuracy of the network (Goodfellow et al., 2016). Standardization of the inputs for the neural network is a standard procedure and is usually done either by scaling inputs to be in the interval $[0,1]$ or subtracting the mean and dividing by the standard deviation. Batch normalization extends this standardization procedure to be included in the intermediate layers of the network, the implementation is done at the batch level instead of the entire sample. Batch normalization was originally introduced by (Ioffe & Szegedy, 2015), and in their paper, they explain how batch normalization alleviates internal covariance shift. Internal covariance shift occurs as the distribution of the inputs for the network layers changes throughout the backpropagation algorithm since the weights are being updated. By standardizing the outputs of each layer, the mean and variance of its output are fixed, whereby, the internal covariance shift is eliminated. The strength of batch normalization comes from its ability to avoid what is referred to as exploding gradients, a large (exploding) gradient causes a large change in the weights, potentially causing the network to diverge. To overcome exploding gradients a smaller value of the learning rate is usually needed, batch normalization reduces the risk of exploding gradients and therefore larger learning rate can be used which causes faster training. By having a larger learning rate during training, the network has a lower risk of getting stuck in local minima. The regulation effect of batch normalization, therefore, comes from avoiding local minima (Bjorck et al., 2018).

The batch normalization algorithm is described below, for any neuron in any layer $\boldsymbol{z}_B^{(l)}$ denotes the output of the summation function as a vector. Since batch normalization is employed with mini-batches the vector contains the output from each observation in that batch. The output is then standardized using the mean and variance computed from the outputs of the batch, where the batch size is m

$$\mu_B^{(l)} = \frac{1}{m}\sum_{i=1}^{m} z_i^{(l)}, \quad \sigma_B^{(l)^2} = \frac{1}{m}\sum_{j=1}^{m}\left(z_i^{(l)} - \mu_B^{(l)}\right)^2,$$

$$\hat{\boldsymbol{z}}_B^{(l)} = \frac{\boldsymbol{z}_B^{(l)} - \mu_B^{(l)}}{\sqrt{\sigma_B^{(l)^2} + \epsilon}}.$$

Here $\hat{\boldsymbol{z}}_B^{(l)}$ denotes the standardized outcome and ϵ is a small number added to ensure numerical stability. The outcome is then de-normalized using the learned parameters $\gamma^{(l)}$ and $\beta^{(l)}$

$$\tilde{\boldsymbol{z}}_B^{(l)} = \gamma^{(l)}\hat{\boldsymbol{z}}_B^{(l)} + \beta^{(l)}.$$

$\tilde{\boldsymbol{z}}_B^{(l)}$ is then passed through the activation function

$$\boldsymbol{y}^{(l+1)} = \sigma\left(\tilde{\boldsymbol{z}}_B^{(l)}\right).$$

When batch normalization is implemented the backpropagation algorithm is used to optimize the parameters $\gamma^{(l)}$ and $\beta^{(l)}$ along with the neurons' weights and biases (Ioffe & Szegedy, 2015).

2.3 Calibration

Calibration is used during the two-step approach for finding the parameters, which give the best fit between observed prices and predicted prices. Generally, the calibration procedure can be defined as follows

$$\hat{\theta} = \underset{\theta}{\operatorname{argmin}}\, L\left(\hat{f}(\theta), P^{MKT}\right).$$

Here θ are the model specific parameters, $\hat{f}(\theta)$ is the learned approximation mapping model parameters to call prices and P^{MKT} are the observed prices. The loss function $L(\,.\,)$ can be any function chosen by the user, but in this thesis, the squared loss is used. Due to the grid design of the gNN network, the loss function must measure the loss given all predicted and observed prices on the grid. This leads to the following loss function

$$L\left(\hat{f}(\theta), P^{MKT}\right) = \frac{1}{TK}\sum_{i=1}^{T}\sum_{j=1}^{K}\left(\hat{f}_{ij}(\theta) - P_{ij}^{MKT}\right)^2.$$

Here $\hat{f}_{ij}(\theta)$ is the networks price predictions for a call with maturity i and strike j, P_{ij}^{MKT} is the observed call price for the same strike and maturity. T denotes the number of maturities on the grid and K the number of strikes.

Since there is no closed-form solution for calibrating parameters, the numerical optimization routine of Broyden-Fletcher-Goldfarb-Shanno (BFGS) is used. The BFGS algorithm is a variant of gradient descent and uses the gradient to update parameters until a local minima is found. The first-order condition for an optimal point using gradient descent is

$$\nabla^{\theta} L\big(\hat{f}(\theta), P^{MKT}\big) = 0.$$

To satisfy this condition the BFGS algorithm updates parameters iteratively using the steps from (Nocedal & Wright, 2006), wherein a full derivation also can be found. The parameter update at each iteration is given by the equation

$$\theta_{i+1} = \theta_i - \lambda H_i \nabla_i^{\theta} L\big(\hat{f}(\theta_i), P^{MKT}\big).$$

Here θ_i are the parameters at iteration i, $\lambda > 0$ is a parameter that governs the speed at which parameters are updated, ∇^{θ} is the Jacobian of the loss function wrt. θ and H_i approximates the inverse Hessian matrix. The approximation of the inverse Hessian updates each iteration using the following relationship

$$H_{i+1} = (I - \rho_i s_i y_i^T) H_i (I - \rho_i y_i s_i^T) + \rho_i s_i s_i^T,$$

$$\rho_i = \frac{1}{y_i^T s_i}.$$

Here s_i is defined as the parameter update and y_i the update of the Jacobian.

$$s_i = \theta_{i+1} - \theta_i,$$

$$y_i = \nabla_{i+1}^{\theta} L\big(\hat{f}(\theta_i), P^{MKT}\big) - \nabla_i^{\theta} L\big(\hat{f}(\theta_i), P^{MKT}\big).$$

The BFGS algorithm is an unconstrained optimization procedure, meaning there is no restriction on which parameter values the algorithm is allowed to use. The BFGS algorithm can be extended to include an upper and lower bound for each parameter by imposing the constraint

$$l_i \leq \theta_i \leq u_i.$$

This means each parameter θ_i is constrained to $\theta_i \in [l_i, u_i]$. This extension of the algorithm is referred to as L-BFGS-B.

3. Implementation methodology

3.1 Simulation

To simulate training data for the neural networks Fourier pricing was used whenever applicable and when not Monte Carlo sample paths were used to estimate the call price. For the Monte Carlo approach, n paths of the stock price are simulated from the chosen model, the call price is then computed as

$$C(K,T)=\frac{1}{n}\sum_{i=1}^{n}\max(S_T-K,0).$$

Here S_T is the stock price at maturity T, K is the strike and C is the call price.

When simulating training data for the neural network, there are some practical considerations to make. First, there should be enough training samples simulated such that the neural network converges to the approximate pricing function. Second, the ranges of the simulated parameters should ideally be set such that the network encounters all plausible parameter combinations. This ensures the network behaves properly when calibrating parameters to real data.

The desire to produce the best possible training data does unfortunately come with some practical issues. Firstly, one could think of setting large ranges for the simulated parameters to ensure all plausible parameter combinations are represented. The issue with this is that the number of training samples intuitively must increase to ensure a broad representation of parameter combinations is present in the training data. This can be a major computational bottleneck, especially with rough volatility, which takes a long time to simulate. The second issue, which is partly correlated with the chosen parameter ranges, is the instability of the simulation function for unreasonable parameter combinations. During testing, numerous issues were found with "bad" parameter values. Some combinations resulted in numerical underflow for Fourier pricing or having to simulate an infeasible high number of sample paths to get nonzero prices for the Monte Carlo approaches.

Lastly, computational budget also sets constraints on the simulation of training data. The most time-consuming simulation in this paper took one minute to simulate one grid of call prices. This means that simulating 10.000 training samples would take ~ 7 days, for just one model. Fortunately, this could be somewhat alleviated by parallel processing and the python library Numba (Lam et al., 2015). For each model 10.000 grids of call prices were simulated, using 10.000 random parameter combinations with nine strikes ranging from 0.7 to 1.3 (assuming the initial value of the underlying $S_0=1$) and ten maturities ranging from one month to one year. The precise values for strike and maturity are stated below

$$\text{Strikes}\in[0.7,0.8,0.9,0.95,1,1.05,1.1,1.2,1.3],$$
$$\text{Maturities}\in[0.1,0.2,0.3,0.4,0.5,0.6,0.7,0.8,0.9,1].$$

To reduce computational time for the Monte Carlo methods, the number of sample paths to simulate was set to 10.000. This occasionally resulted in zero prices for deep out of the money options, whenever this happened, the simulated grid was discarded, and new parameters simulated. To reduce the time spent wasted on simulating zero prices, some parameter ranges were modified by trial and error until no more than 10% of the simulated grids were discarded. For this reason, parameters like ρ and V_0 does not have the same range across different volatility models, even though they represent the same. Chosen parameter ranges for each model are reported in table 1.

Table 1: Ranges for simulated parameters

Heston	rHeston	rBergomi	rFSV
$\rho \in [-1,0]$	$\rho \in [-0.9,-0.5]$	$\rho \in [-0.9,0]$	$\nu \in [0.05,1]$
$V_0 \in [0.05,0.2]$	$V_0 \in [0.05,0.4]$	$\xi \in [0.05,1]$	$V_0 \in [0.01,0.25]$
$\kappa \in [0,5]$	$\alpha \in [0.6,0.9]$	$H \in [0.05,0.4]$	$H \in [0.05,0.4]$
$\theta \in [0.05,0.2]$	$\bar{V} \in [0.05,0.4]$	$\eta \in [0.05,1]$	$\alpha \in [0.05,1]$
$\eta \in [0.05,1$	$\lambda \in [0.1,1]$		$m \in [-0.7,-2.3]$
	$\xi \in [0.1,1]$		

3.2 Neural networks fitting

To fit each neural networks, data was split into three parts; training, validation and test sets using a split of 80-10-10. The validation set was used during the training process to implement early stopping, where training was stopped if the validation loss had not been improved after 25 rounds. Simulated data were standardized to be on the $[0,1]$ interval using the transformation

$$\theta_{scaled} = \frac{\theta - \min(\theta)}{\max(\theta) - \min(\theta)}.$$

When designing a neural network, the user must decide on the architecture of the network, this includes choosing the number of layers, neurons, activation function, and so on. Ideally, the user might want to use some form of hyperparameter search to determine the optimal architecture for the learning problem. To keep the complexity of this thesis at a suitable level, the chosen network architectures were copied from (Horvath et al., 2021) and (Bayer et al., 2019).

3.2.1 MLP architecture

Two variants of the multilayer perceptron (MLP) were fitted, one using the piece wise approach (pNN) from (Bayer & Stemper, 2018) and one with the grid wise approach of (Horvath et al., 2021). Inputs for pNN were the model parameters, strike and maturity, output was a single call price. Inputs for gNN were model parameters and output was a grid of call prices of size 9 strikes * 10 maturities. The architecture of both the MLPs was based on (Horvath et al., 2021), where the network consisted of four hidden layers each containing 30 neurons with the ELU activation function. Recall the ELU requires the parameter α, here the default value of $\alpha = 1$ was used.

For training, the Adam algorithm was used with a batch size of 100 and a learning rate of 0.001. The networks were trained for a maximum of 200 epochs. The values for batch size and learning rates were chosen manually due to the time constraint of this thesis, though one would ideally search for the optimal values of these hyperparameters.

3.2.2 CNN architecture

For the one-step approach, a convolutional neural network (CNN) was used, with network architecture based on (Bayer et al., 2019). This CNN had one convolutional layer with 16 filters, a 3x3 kernel with a stride of one and zero padding. The output of the convolutional layer was passed through the Leaky ReLU activation function with $\alpha = 0.01$. Following the convolutional layer and activation function, a maxpooling layer with a 2x2 sliding window and a stride of 2 was used. The output of the maxpooling layer was flattened and passed through a 50-neuron feedforward layer with the ELU activation function. Again the Adam algorithm was used for training with batch size of 100, learning rate of 0.001, maximum number of epochs was 200 with an early stopping criterion of 25 rounds.

3.2.3 Regularization implementation

To test the regularization effect from batch normalization, dropout, and weight decay each of the three neural networks (pNN, gNN and CNN) was extended by incorporating each regularization technique individually. That means 12 neural networks were fitted for each volatility model, Heston, rough Heston, rough Bergomi and rough FSV, for a total of 48 neural networks. Dropout was implemented in each layer with $p^{(l)} = 0.5$ for all layers. The weight decay networks were implemented with L2 norm and $\lambda = 0.001$, other values of λ were tested however performance declined as the parameter value increased.

3.3 Calibrating to SPX

Once the network performance had been established on simulated data the two calibration approaches were tested on real-world data using observed option prices from SPX on the 13th of November 2019 downloaded from optionsdx.com. This dataset consists of 1889 options both calls and puts with twelve different maturities ranging from 37 to 583 days with strikes between 100 and 4400. SPX options have the S&P500 index as the underlying and the close price of the S&P500 for the chosen day was 3094.04 USD. The close price will from now on be referred to as the spot price, and any observed put or call price was calculated using the mid-price which is the average between the bid and ask price.

Dealing with option data usually consists of some form of cleaning process, in this thesis the cleaning process follows the steps from (Kokholm, 2016). The cleaning steps consist of (1) removing options with a zero bid. (2) removing options with absolute moneyness below 0.4, and (3) removing options with a bid-ask spread larger than five. Note absolute moneyness is defined as the absolute of strike divided by spot price. After these three steps, the yield curve is backed out from the option data by a numerical optimization algorithm minimizing the loss function below.

$$Loss(\{r_{T_i}, \dots, r_{T_m}\}, q) = \sum_{i=1}^{m} \sum_{j=1}^{10} \left(P^{obs}_{(T_i,K_j)} - P^{PC}_{\left(C^{obs}_{(T_i,K_j)}, q, r_{T_i}\right)} \right)^2 .$$

The loss function takes five options on each side of the at the money strike for all maturities. It then measures the squared difference between the observed put price P^{obs} and the put price calculated using the put-call parity P^{PC} from the observed call prices C^{obs}. The loss function then takes the sum of the squared difference for the ten chosen strikes K_j and all m maturities T_i. The put-call parity is stated below

$$P^{PC}_{\left(C^{obs}_{(T_i,K_j)},q,r_{T_i}\right)} = C^{obs}_{(T_i,K_j)} + K_j e^{-r_{T_i}T_i} - Se^{-qT_i},$$

here r_{T_i} is the interest rate for maturity T_i and q is the dividend yield rate. The optimization algorithm is then tasked with minimizing the loss function by optimizing the interest rates and the dividend yield

$$\{\hat{r}_{T_i}, \dots, \hat{r}_{T_m}\}, \hat{q} \ = \underset{\{r_{T_i},\dots,r_{T_m}\},q}{\operatorname{argmin}} \ Loss(\{r_{T_i}, \dots, r_{T_m}\}, q)\,.$$

With the obtained yield curve, the second part of the cleaning process can begin, again the steps followed are from (Kokholm, 2016). The first step is to remove any options that are in the money, which means strikes less than the spot price for calls and strikes greater than the spot price for puts. For the second step, the put-call parity was used on the remaining puts to calculate the price for in-the-money calls. If any prices calculated from the put-call parity were negative, they were removed from the dataset. The reason for using the put-call parity here is that in the money calls are less traded so the observed mid-price is a noisy estimate of the true price. The last step in the cleaning procedure was to ensure no options violate the three-arbitrage restriction

$$C(S_t, T_i, K_j) > \max\left\{0, \mathrm{S_t e}^{-\int_t^T qds} - \mathrm{K_j e}^{-\int_t^T r_{T_i}ds}\right\},$$
$$\frac{C(S_t, T_i, K_{j-1}) - C(S_t, T_i, K_j)}{K_j - K_{j-1}} \in \left[0, \mathrm{e}^{-\int_t^T r_{T_i}ds}\right],$$
$$\frac{C(S_t, T_i, K_{j-1}) - C(S_t, T_i, K_j)}{K_j - K_{j-1}} - \frac{C(S_t, T_i, K_j) - C(S_t, T_i, K_{j+1})}{K_{j+1} - K_j} \geq 0.$$

If an option violated any of the arbitrage restrictions, it was removed from the dataset. Following all these cleaning steps resulted in a dataset consisting of 1214 calls with maturities between 37 and 583 days and strikes between 1875 and 4300.

The choice of simulating call prices in grids imposed a practical issue when calibration to real data. Since it is not guaranteed that one can observe call prices for exactly the combinations of strike and maturity chosen during simulation, one must use interpolation to obtain the desired prices. There exist multiple methods in the literature that can interpolate option prices while ensuring the interpolated prices remain arbitrage-free, see for example (Andreasen & Huge, 2010; Bender & Thiel, 2020). However, due to the time constraint of this thesis, a simpler approach had to be used for interpolation. Since the implied volatility curves for a given maturity is smooth, a second-order polynomial was used to fit call prices C as a function of the strike for a given maturity T_i

$$C_{T_i} = \beta_0 + \beta_1 K_{T_i} + \beta_2 K^2_{T_i} + \epsilon_{T_i}.$$

This was performed for each maturity in the dataset and the fit was used to calculate the call price for the desired strikes. This gave the call prices for the correct strikes, but not the correct maturities. Therefore, a second-order polynomial was fitted for each desired strike K_j with the interpolated prices and the observed maturities

$$\hat{C}_{K_j} = \gamma_0 + \gamma_1 T_{K_j} + \gamma_2 T_{K_j}^2 + v_{K_j}.$$

After these steps, the resulting dataset consisted of 90 interpolated call prices for the desired strikes and maturities. Now while one ideally would use a proper interpolation method, this approach made it possible to examine the performance of the neural network on a dataset resembling real-world prices. The importance here was to assess how the networks perform when they are used on data that did not originate from the volatility specification they were trained upon.

4. Results

This section explores the performance of the different neural network approaches to option calibration. For the two-step approach, the two variants of MLPs found in the literature were implemented. Variant one is the piece-wise approach, where the network is trained to predict a single call price and is referred to as pNN. Variant two is the grid-wise approach, where the network predicts a grid of call prices and is referred to as gNN. For the one-step approach, the convolutional neural network is referred to as CNN.

Note the code from (McCrickerd, 2017) used to simulate rBergomi prices takes a roughness parameter a as input instead of the Hurst parameter H. The figure and tables in this section therefore displays a instead of H, even though it would have been more intuitive to report H. This is unfortunate as McCrikerd defines $H = a + 0.5$, which is different from the rHeston model of (El Euch & Rosenbaum, 2019) were it was defined as $H = a - 0.5$. Regrettably this inconsitency was recalled to late which is why a is reported instead of H for rBergomi.

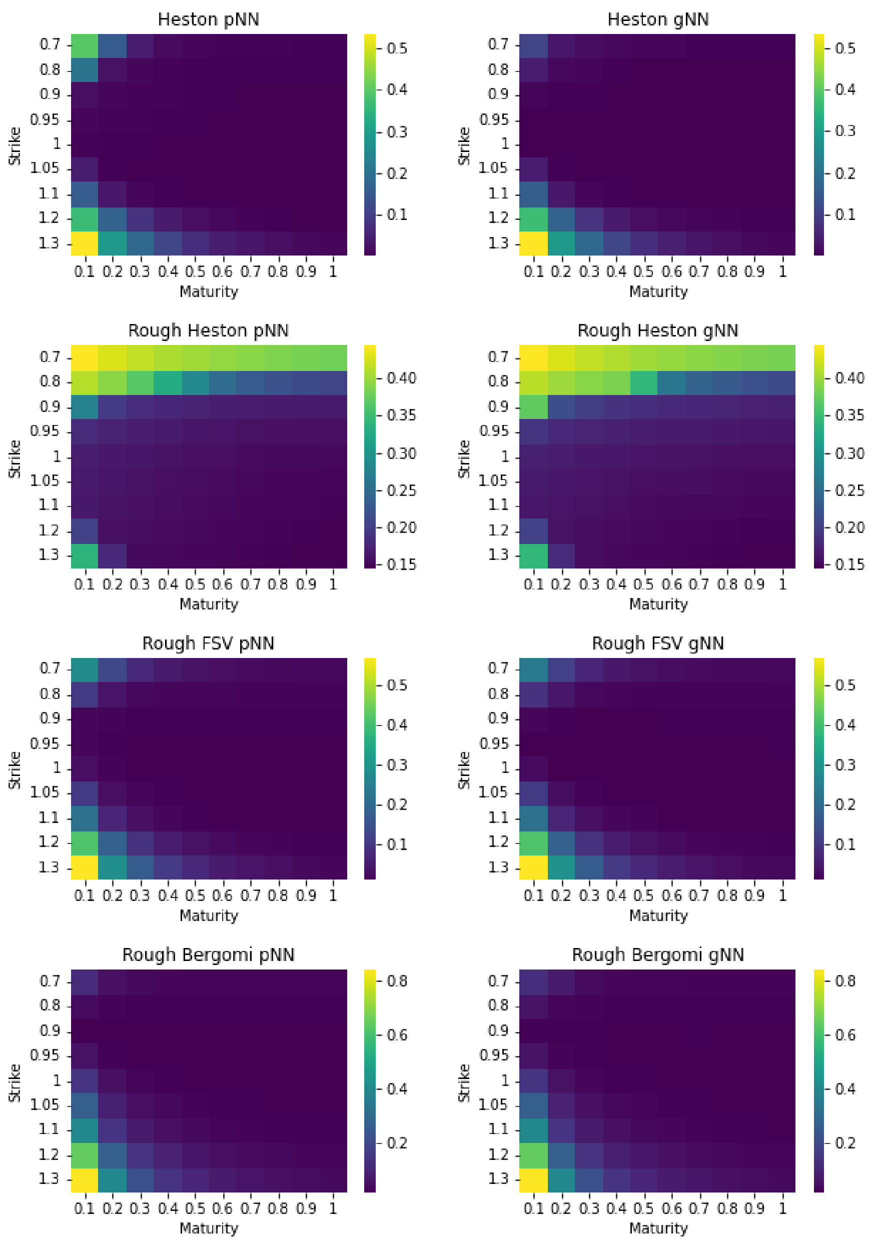
4.1 The two-step approach

4.1.1 Sample fit

The first step in evaluating the neural networks is to examine the fit on the test sample, the neural networks should ideally have learned to unobserved approximation function of parameters to prices. Since one usually deals with implied volatility instead of prices when evaluating option pricing models, the simulated and predicted prices are converted to implied volatility before comparison. Implied volatility is defined as the volatility value that sets the predicted price of the Black & Scholes model equal to the observed price. Figure 6 displays a heatmap of the mean absolute error between simulated and predicted implied volatility for each of the four volatility models. The heatmap displays the average absolute error for each combination of strike and maturity for both the two-step approaches.

For all models, it is clear that the networks do not perform equally for the different calls on the grid. Calls with high strike and short maturity show the largest error in all cases but for the rough Heston model. For the rough FSV and Bergomi models, this might have been explained by the reduced number of simulated paths during the data simulation, which was required to reduce computational time. This might have resulted in too noisy estimates for the call price, which means the network cannot learn to approximate the true price. The pattern however, is also observed for the Heston model, where data was simulated using Fourier pricing, this indicates that the high error for short maturity options does not arise from simulation issues but rather the neural networks struggling to learn the approximation function for these calls.

Figure 6: Mean Absolute Error: Test set



This figure displays the approximation accuracy for both two-step models for the four different volatility specificaiton. Errors are calculated as the mean absoulute error between predicted and simulated implied volatility and displayed for all calls on the simulated grids.

While the rough Heston models also shows large errors for the shortest maturity and highest strike call, it performs significantly worse for the smallest strikes 0.7 and 0.8 compared to all the other models for all maturity levels. The simulation process for the rough Heston model had many issues with the Fourier pricing algorithm not converging for different parameter combinations, without any obvious reason as to why and which parameter values caused issues.

Since the error pattern of the rough Heston neural networks differs quite significantly from the other models and the troubles occurring during simulation, it seems there might be issues with the simulation function. Another explanation could be that the chosen neural network architectures simply cannot learn the approximation function for the rough Heston. Whether the network architecture or the simulation function is at fault is not further explored due to time constraints.

Figure 6 gives a nice overview of the neural network performance on different calls, it is hard to examine how the different MLP approaches compare against each other. Table 2 examines the absolute error between predicted implied volatility versus simulated implied volatility for all volatility models and the different MLP approaches. The values are computed across all combinations of strike and maturity for the test set.

Table 2: Absolute error: Predicted IV vs. simulated IV

Model		Q1	Mean	Median	Q3	Std. dev
Heston	*pNN*	0.0021	0.0375	0.0041	0.0108	0.0961
	gNN	0.0008	0.0293	0.0021	0.0080	0.0829
	% Difference	154.7%	28.1%	92.9%	35.5%	15.9%
rHeston	*pNN*	0.0653	0.2039	0.1482	0.3540	0.1706
	gNN	0.0708	0.2096	0.1635	0.3462	0.1690
	% Difference	-8%	-3%	-9%	2%	1%
rFSV	*pNN*	0.0047	0.0447	0.0109	0.0252	0.1012
	gNN	0.0042	0.0433	0.0099	0.0244	0.0995
	% Difference	12.4%	3.3%	9.9%	3.4%	1.6%
rBergomi	*pNN*	0.0063	0.0640	0.0145	0.0330	0.1486
	gNN	0.0082	0.0673	0.0185	0.0399	0.1486
	% Difference	-23.3%	-5.0%	-21.5%	-17.2%	0.0%

Descriptive statistics for the mean absolute error of predicted implied volatility against the simulated (true) implied volatility. For each volatility model the performance of the two MLPs are displayed, pNN is the piece wise approach and gNN is the grid-wise approach
** % Differences are computed as (pNN / gNN - 1)*100*
** Q1 represent the .25 quantile and Q3 the .75 quantile*

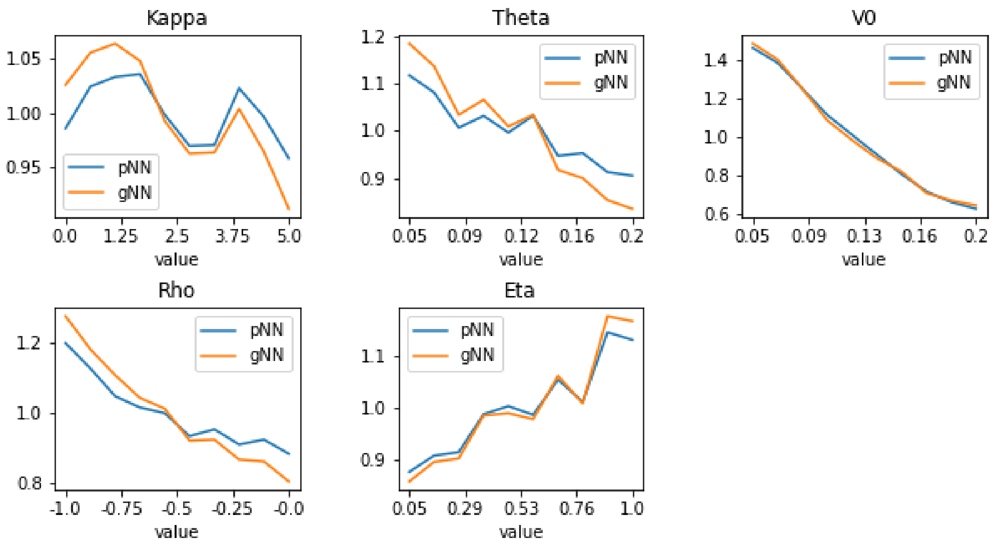
When comparing the performance of the piecewise (pNN) and grid-wise (gNN) approaches there is no clear answer as to which approach has the best performance. Numerically, gNN outperforms pNN on all metrics for the Heston and rFSV, while on the rBergomi model pNN outperforms gNN. For rHeston, the best model depends on the metric, based on mean and standard deviation gNN shows the best performance, while Q1, Q3, and median indicate that pNN performs better. Large differences in mean and median absolute error are observed for all models. This shows, the error distribution is right-skewed, which is consistent with Figure 6, showing large errors for some calls.

Both figure 6 and table 2 show that the neural networks can approximate the pricing function with high precision but struggle to predict call prices for calls with short maturity and high strikes. Only the neural networks for the Heston model showed a significant difference in performance. The performances of the other volatility models are relatively close, with a maximum of 5% difference in mean absolute error. Therefore, it is not possible to conclude from the above which of the two MLP approaches performs best. Perhaps using a larger training set and running a hyperoptimization algorithm for the optimal network architectures would show that one approach outperfors th e other, but this is left for future research. It is worth noting, when comparing the two approaches, that training speed for gNN is signficantly faster than pNN. During the training phase the different pNN networks took roughly 20-30 minutes to train, while pNN networks could be trained in close to 2 minutes. In practice however, this might be irrelevant as the networks need only be trained once.

4.1.2 Error by parameter values

While it is important to know how well each neural network approximates the call prices, an interesting question is whether the performance is the same across different parameter values. To examine this, each parameter was grouped into 10 bins and for each bin, the mean absolute implied volatility error was computed. For better interpretability, the error of each bin was divided by the total error for all parameter values. Figure 7 shows the relative mean absolute error for the two Heston models for each parameter.

Figure 7: Heston - Relative MAE by parameter value



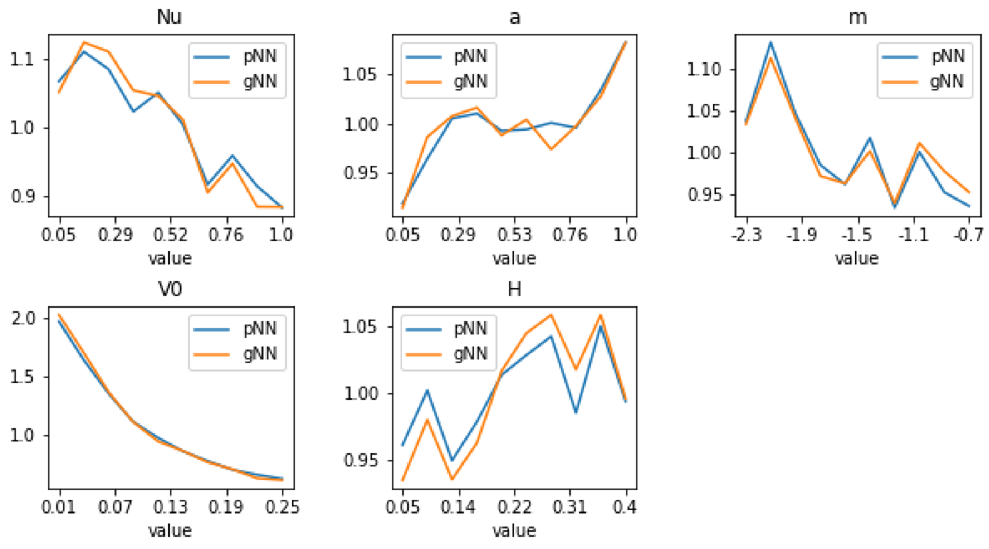
Relative mean absolute error for two-step Heston networks by parameter value.
Errors are calculated relative to total loss across all parameter ranges.

Depending on the parameter there are large differences in performance based on the parameter values. Looking at the initial level of volatility V_0, the error is 40% larger than the average when V_0 is at 0.05, contrary the error is ~ 40% lower than the average with $V_0 = 0.2$. Differences in performance can be seen for all parameters except kappa, where performance is mostly the same for the different parameter values. This is true for both pNN and gNN as there seems to be little difference in the error values for the two approaches.

That performance differs across parameter values is important when one wants to implement a neural network approach to option calibration. One might wrongly assume that the network has accurately learned the unobserved pricing function based on small test errors when it might only have done so for a subset of the parameter values. If the optimal set of calibrated parameters lies inside a region where the network performs poorly, the calibration algorithm might not be able to find these parameters as the network does not accurately translate those parameters to prices. By determining how certain parameter values affect performance one then gains insight into the uncertainty of a given calibrated parameter. If, for example, the calibrated value of Eta on a given day is 0.05, one might conclude with a higher certainty that this is the true[1] parameter. If, on the other hand, the calibrated value was closer to 1, it is more uncertain whether this is the true parameter as there is a large pricing error in this region. Note figure 7 is calculated assuming all else equal, it does not examine how performance given a parameter is affected by the other parameters.

That specific parameter values affect performance is not exclusive to the Heston model but is also present for all other volatility models. Figure 8 displays how the parameters for the rFSV model affect its performance, here performance again varies depending on the parameter value. The figures for the rough Heston and Bergomi model can be found in Appendix A1. Similar to the Heston models, there is little difference between the performance of pNN and gNN for all rough volatility models.

Figure 8: rFSV - Relative MAE by parameter value



Relative mean absolute error for two-step rFSV networks by parameter value.

To alleviate the issues with bad performance in some parameter regions, it might be beneficial to increase attention to those regions during the training process. This might be done by either simulating more from these regions or modifying the loss function. Another solution could be to extend the ranges used when simulating data, some parameters like a from the rFSV model perform worst when it is close to 1, which is the maximum simulated value. By extending the range, the network might better learn how to translate a close to 1 to correct call prices.

4.1.3 Extrapolation

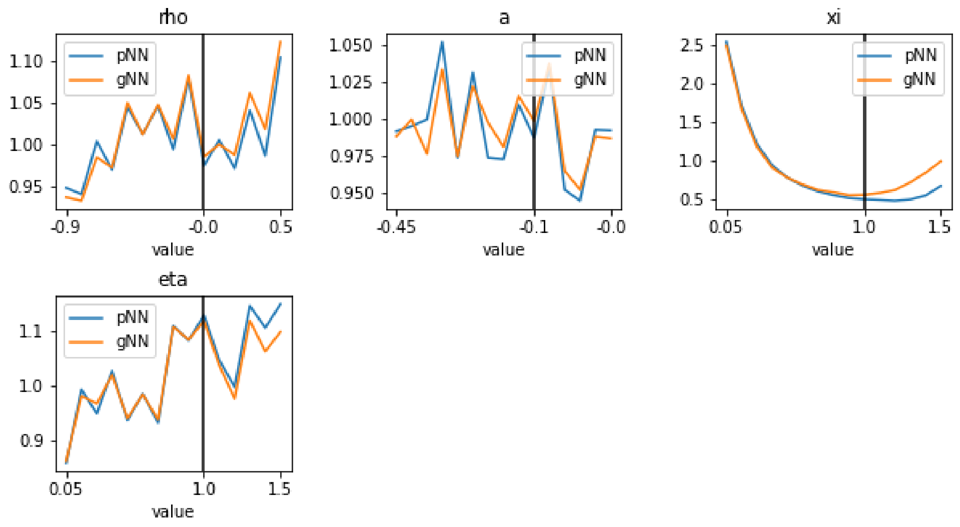
By examining how parameter values affect performance, it was established that the overall performance of a network depends on the specific parameters. A natural question to ask then is how the network performs when using parameters outside its training ranges. This is the issue of extrapolation, which is important for practical use where it is not guaranteed that the true parameters are inside the training intervals. To examine this issue new data was simulated with parameter ranges outside the training interval. This was done for each parameter individually, meaning only that parameter range was changed while keeping all other parameter

[1] True parameter in the sense that the optimal calibration scheme with no errors will result in that parameter

ranges the same as for training. This approach makes it possible to isolate the effect of each parameter when extrapolated outside its training range.

Figure 9 displays the model performance for the different parameters of the rough Bergomi model. To the left of the vertical black line are the training parameters and to the right are the extrapolation parameters Note that errors are displayed relative to the mean absolute error of the test set. Interestingly, the rough Bergomi model performs well when used in an extrapolation setting. For most parameters, the performance is on par with the training performance and seems to follow a linear trend. The performance for ρ and η is slightly decreasing as their values increase, a has seemingly constant performance and ξ shows decreasing performance as the parameter becomes larger than the training range.

Figure 9: rBergomi - Relative MAE by parameter value



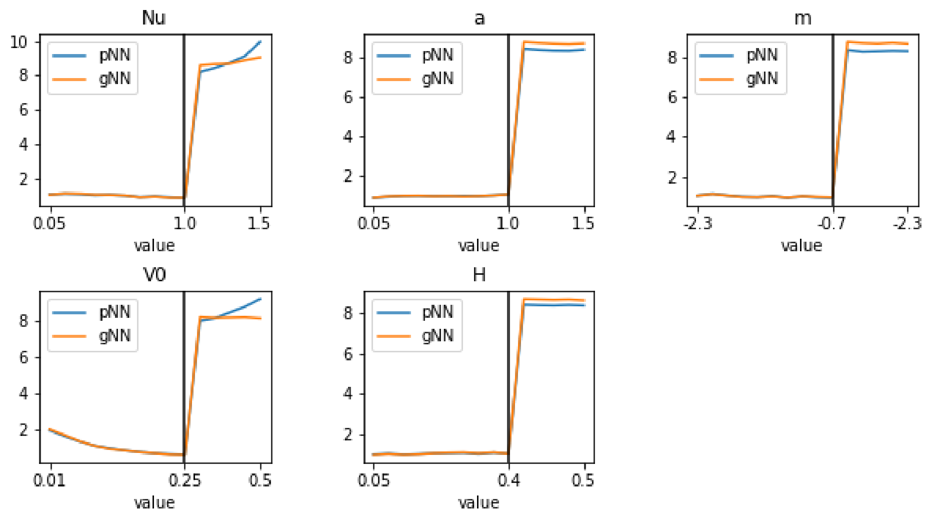
Relative mean absolute error for two-step rBergomi networks by parameter value. The right side of the black line shows extrapolation performance

The ability of the rough Bergomi models to extrapolate outside training ranges is particularly useful in practical implementations. When calibrating model parameters, it is not strictly needed to restrict the calibration algorithm to keep parameters inside training ranges. This means that the model can still be implemented even if the parameter ranges are not set wide enough. Note that parameters were extrapolated with all else being equal, it is therefore not guaranteed that the rough Bergomi models work when multiple parameters are extrapolated.

The extrapolation power of the rough Bergomi models is unfortunately not present for the other models. Figure 10 display the relative performance for the rough FSV parameters. As the parameters exceed their training ranges both pNN and gNN performance jumps and the network essentially breaks. The errors increase up to 10 times the average, showing that the rFSV models are incapable of extrapolating. Again, this is important for practical applications as the calibration scheme must impose that parameters do not exceed the training ranges. If the true parameters are outside the training ranges it will therefore not be possible to correctly calibrate the parameters. The pattern observed for the rFSV model is similar to the Heston model and can be found in Appendix A2.

As the simulation of rHeston data is especially time-consuming, extrapolation performance of rHeston networks has been omitted and left for future research.

Figure 10: rFSV - Relative MAE by parameter value



Relative mean absolute error for two-step rFSV networks by parameter value. The right side of the black line shows extrapolation performance

4.1.4 Regularization

The effect of adding regularization was examined for both pNN and gNN, six additional neural networks were trained for each volatility model using different forms of regularization. Each network is fitted with only a single form of regularization and compared to the default network with only early stopping. Table 3 displays the mean absolute implied volatility error and standard deviation for each network relative to the default network.

Table 3: Regularization for the two-step approach

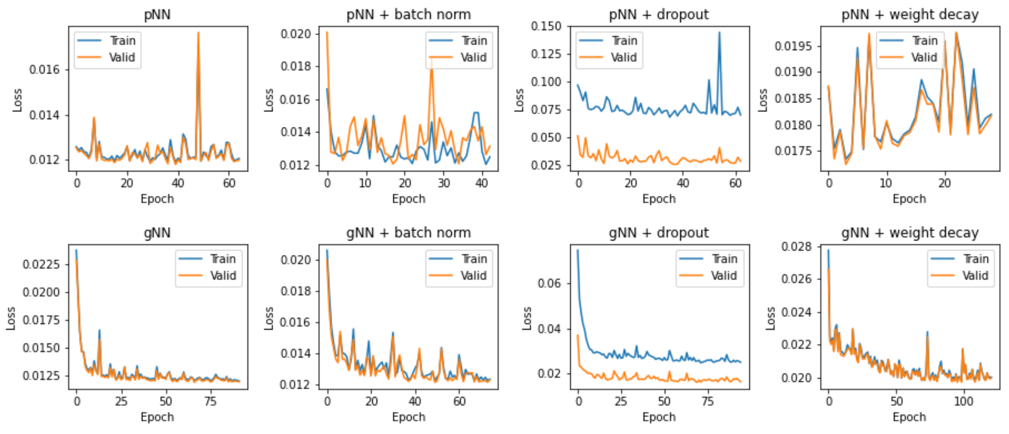
		Default		**+Batch norm**		**+Dropout**		**+Weight decay**	
Model		**Mean**	**Std. dev**	**Mean**	**Std. dev**	**Mean**	**Std. dev**	**Mean**	**Std. dev**
Heston	*pNN*	1.000	1.000	1.213	1.061	3.007	1.346	1.429	1.045
	gNN	1.000	1.000	1.045	1.020	1.809	0.967	1.783	1.115
rHeston	*pNN*	1.000	1.000	1.035	1.003	1.111	1.001	1.034	1.005
	gNN	1.000	1.000	0.998	0.996	0.997	1.007	0.980	1.008
rFSV	*pNN*	1.000	1.000	1.032	0.982	2.721	1.420	1.379	1.032
	gNN	1.000	1.000	1.112	1.031	1.601	1.068	1.406	1.047
rBergomi	*pNN*	1.000	1.000	1.048	0.998	2.072	1.126	1.175	1.024
	gNN	1.000	1.000	1.048	1.003	1.437	0.995	1.102	1.001

This table displays how different regularization techniques affect model performance for both the two-step approaches pNN and gNN. Performance is measured as mean absolute error of predicted implied volatility versus simulated implied volatility. For interpretability performance measures are relative to the baseline network

In most cases, adding regularization resulted in worse performance, with the worst case resulting in three times increase in MAE. Only in a few cases did regularization improve model performance, like the rHeston with weight decay, where the error was reduced by 2%. Since these performance gains are of such small magnitude and not consistently observed across multiple volatility models, they are likely not significant. Adding regularization to the networks did also not consistently reduce the standard deviation of the errors. It is therefore concluded that regularization does not help the networks perform better for the two-step approach. This coincides with findings in the literature where (Bayer & Stemper, 2018) find batch normalization decreases performance and (Hernandez, 2016) find increasing the dropout rate also leads to poor performance. Note that a combination of multiple regularization techniques has not been tested, it might be beneficial to combine the regularization techniques, but this is left for future work.

The reason regularization techniques do not improve the performance of the two MLPs might be explained by Figure 11. Here, the training and validation loss is plotted for each epoch during the training of each different network for rFSV. On the left-hand side, we see the validation and training loss for the networks with no regularization are almost identical, in other words, there are no signs of overfitting. Since regularization is used to prevent overfitting, it makes sense that it should not have an effect if the network is not overfitting. The same plot for the other three models can be found in Appendix A3.

Figure 11: rFSV - Effect of regularization on loss during training



Training and validation loss during each epoch for the two-step approaches with rFSV

Recall the benefit of using batch normalization was not only its regularization effect but also the possibility of using larger learning rates. During training, different learning rates were tested to see if increasing the learning rate was possible, and how it affected performance. While it was often possible to increase the learning rate while maintaining similar performance, the batch normalization networks could not consistently outperform the default networks. This again shows that regularization is not needed for the two MLP approaches.

4.2 The one-step approach

With the performance of the two-step approach established, networks trained to calibrate parameters in a one-step approach were evaluated in a similar vein. First, the CNNs are evaluated based on the overall fit of the test sample, then the effect of regularization is assessed, and lastly whether the trained CNNs can extrapolate.

4.2.1 Sample fit

Table 4 examines different loss metrics for each parameters for the four volatility models. The values were calculated using the simulated parameters from the test set and the parameters predicted by the CNNs taking the simulated call prices as input. Ideally, the errors should be zero, meaning the network was able to do a perfect calibration. Examining the mean and standard deviation of the error shows that predictions are closely centered around zero. This indicates that on average the CNNs can calibrate to the correct parameters and therefore unbiased. Running a student t-test on the null hypothesis that the average prediction error is zero cannot be rejected for any parameters. Even though predictions are centered around the true values, the degree of precision varies between the parameters. Looking at the mean absolute error (MAE) and mean absolute percentage error (MAPE) shows there is a large variation between parameters. Looking at the Heston CNN shows that theta, V0 and eta are relatively precisely calibrated with a MAPE of 7.52, 2.27, and 13.7 percent, respectively. On the other hand, it struggles to calibrate kappa and rho, where in both cases, calibrated parameters are more than 100% off from the true parameter on average. Similarly, for the other CNNs, some parameters are substantially better calibrated than others. Alpha for the rHeston, V0 for rFSV, and xi for rBergomi are all calibrated with a MAPE less than 10%, all other parameters conversely have considerably higher errors.

Table 4: CNN error by model and parameter

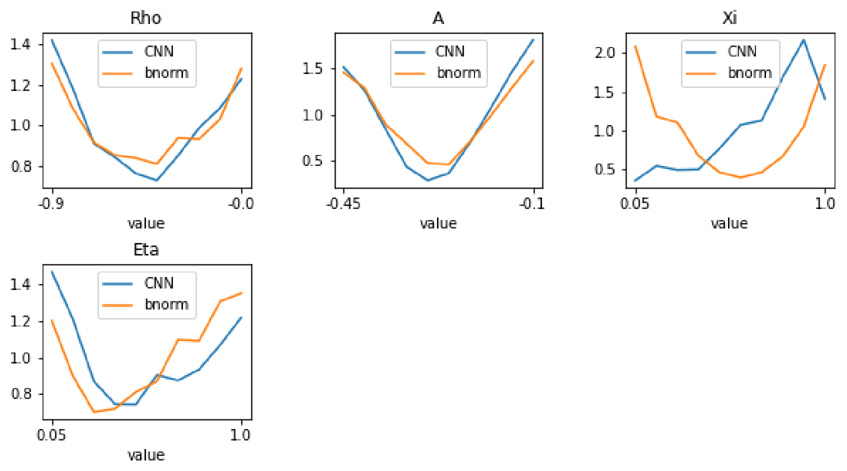
Model	Parameter	Mean	Std	MAE	MAPE
Heston	*Kappa*	0.0555	0.5271	0.375	106.52
	Theta	0.0008	0.0125	0.0076	7.52
	V0	-0.0024	0.0025	0.0028	2.27
	Rho	-0.0216	0.121	0.0923	125.56
	Eta	-0.0275	0.0547	0.0458	13.74
rHeston	*V0*	0.0051	0.1003	0.0866	61.29
	Alpha	-0.0032	0.0785	0.0659	8.79
	Lambda	-0.0062	0.2559	0.2226	65.61
	V_bar	0.0009	0.0992	0.086	57.32
	Xi	0.0065	0.2382	0.2013	65.49
	Rho	-0.004	0.1142	0.0989	14.84
rFSV	*Nu*	-0.0317	0.0891	0.0753	22.42
	A	-0.0206	0.2378	0.1976	70.13
	M	0.0866	0.2754	0.2072	14.34
	V0	0.0051	0.0086	0.0077	6.91
	H	0.0279	0.082	0.0703	52.38
rBergomi	*Rho*	-0.0143	0.1544	0.1233	146.73
	A	-0.01	0.09	0.0754	36.47
	Xi	0.0096	0.0234	0.0183	3.85
	Eta	0.0136	0.1694	0.1374	50.58

Accuracy of the one-step approach for all volatility specifications by individual parameters.

The lack of consistent performance across all parameters makes it difficult to use the CNNs in a practical application of calibrating to the observed call surface. While the simplicity of directly calibrating to observed prices favors the one-step approach, a model is impractical if cannot produce precise estimates. Even though the predictions are centered around the true parameters, the standard deviation is likely too large for practical use. That the CNNs show low precision is also consistent with the findings of (Bayer et al., 2019; Horvath et al., 2021) who both found a lack of generalizability for their networks trained to one-step calibrate parameters.

Examining CNN performance based on parameter value shows similar to the two-step, accuracy depending on parameter value, but in a different way. Figure 12 displays the relative performance of rBergomi across its different parameters. Errors are computed as the mean absolute error (MAE) relative to the total MAE across the full parameter range.

Figure 12: rBergomi - Relative MAE by parameter value



Relative mean absolute error for default CNN and batch normalized CNN by parameter value. Errors are calculated relative to total loss across all parameter ranges.

Where the plots for the two-step approaches showed performance and parameter value generally having a linear relationship, Figure 12 shows a quadratic. This relationship shows that the error is at its minimum when the parameter value is in the middle of the simulated range and increases toward the edges. For future research, it would be interesting to examine whether this relationship holds when the CNNs are trained on even broader parameter ranges. If this is so the practical performance might be improved by increasing parameter ranges, as true parameters would less likely be at the edge of the parameter ranges. Another strategy could examine how performance is affected by simulating more from regions with high errors. These questions are not pursued further due to the time limitations of this thesis. The plots for Heston, rHeston, and rFSV can be found in Appendix A4, and all show similar results as rBergomi.

4.2.2 Regularization

The poor generalizability of the CNNs might have been caused by overfitting. To assess this, the networks were extended using the three regularization techniques batch normalization, dropout, and weight decay. Table 5 reports the average parameter error relative to that of the default CNN. Similar to the MLPs of the two-step approach, weight decay and dropout do not improve performance but instead reduce the accuracy of the network. Adding dropout results in ~1.5 times larger error for rFSV and rBergomi, while the standard deviation of errors increases by a factor of ~1.2. Weight decay similarly increases both the mean error and standard deviation for rFSV and rBergomi, albeit slightly less than dropout. the Heston model is especially affected as its mean error and standard deviation increase by 2-3 times that of the default network when adding dropout and weight decay. Only the rHeston model is unaffected by adding regularization as the network achieves almost identical performance for any of the methods. While dropout and weight decay did not lead to any improvements, batch normalization could reduce the error for all volatility models except rHeston. For rFSV and rBergomi, both the average error and the standard deviation are reduced by roughly 8% relative to the default network. This improvement is even more pronounced for Heston, as the mean error is reduced by 21.2% and standard deviation by 14.4%.

Table 5: Regularization for the one-step approach

	Default		+Batch norm		+Dropout		+Weight decay	
Model	**Mean**	**Std. dev**	**Mean**	**Std. dev**	**Mean**	**Std. dev**	**Mean**	**Std. dev**
Heston	1.000	1.000	0.788	0.856	2.996	2.461	2.242	1.952
rHeston	1.000	1.000	1.001	1.002	1.002	1.003	1.002	1.003
rFSV	1.000	1.000	0.922	0.937	1.479	1.179	1.242	1.055
rBergomi	1.000	1.000	0.925	0.930	1.514	1.249	1.301	1.251

This table displays how different regularization techniques affect model performance for the one-step approach (CNN). Performance is measured as the mean absolute error between predicted parameters and simulated parameters. Performance measures are calculated relative to the default CNN

To get a deeper understanding of how batch normalization affects performance, table 6 examines the performance gain on each parameter. While the average error was reduced, the addition of batch normalization increased calibration accuracy for all parameters. Looking at the Heston model, only kappa and eta had a lower calibration MAPE, whereas MAPE for the other parameters increased. Thus, while overall performance increased, it did not affect all parameters positively. Looking at the standard deviation for Heston, on the other hand, shows significantly better results. A reduction between 17 and 31 percent was achieved for all parameters except theta, which had a slight increase of 0.8%. For rHeston, no parameter showed a large difference in performance, this again shows that regularization did not have an effect here. All rFSV parameters had higher calibration accuracy both measured by MAPE and standard deviation. Results for rBergomi were like Heston mixed with performance gains on the same parameters while a reduction in accuracy for others.

Even though batch normalization improves the CNN performance, it is still not enough to ensure a satisfactory model. There are still parameters with high errors like rho from Heston and rBergomi with MAPEs of 153 and 198 percent, respectively. For both models, batch normalization creates even poorer scores for rho, which is unfortunate as in both cases this is the worst-performing parameter. So, while the results here show that the one-step approach currently does not perform adequately, it highlights the possibility for improvement. Further research into model architecture, improvements of the training sample, or modified loss functions might find the required performance gains for the one-step approach.

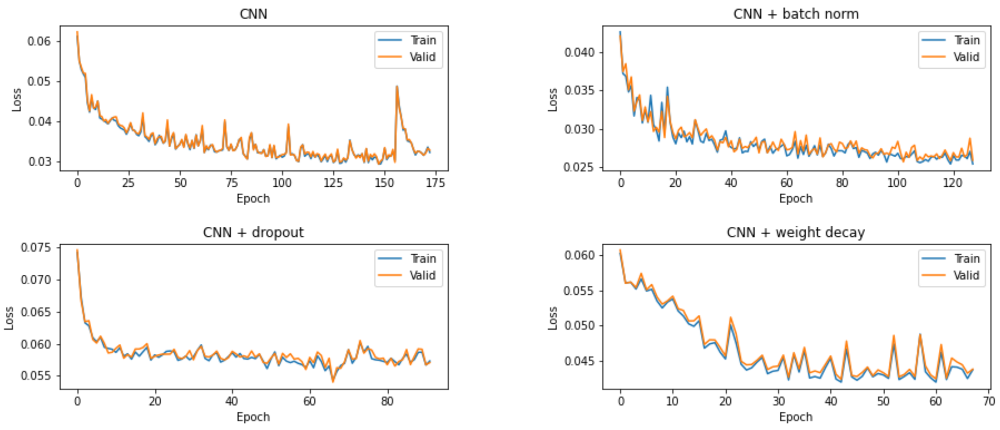
Table 6: Effect of batch normalization on model parameter

		MAPE			Standard deviation		
Model	**Parameter**	**Default**	**Batch norm**	**% Difference**	**Default**	**Batch norm**	**% Difference**
Heston	*Kappa*	106.52	85.24	-20.0%	0.5271	0.4376	-17.0%
	Theta	7.52	7.95	5.7%	0.0125	0.0126	0.8%
	V0	2.27	3.42	50.7%	0.0025	0.002	-20.0%
	Rho	125.56	153.27	22.1%	0.121	0.092	-24.0%
	Eta	13.74	12.89	-6.2%	0.0547	0.0376	-31.3%
rHeston	*V0*	61.29	58.51	-4.5%	0.1003	0.0982	-2.1%
	Alpha	8.79	8.99	2.3%	0.0785	0.0791	0.8%
	Lambda	65.61	64.01	-2.4%	0.2559	0.2574	0.6%
	V_bar	57.32	57.59	0.5%	0.0992	0.1003	1.1%
	Xi	65.49	63.52	-3.0%	0.2382	0.2358	-1.0%
	Rho	14.84	14.51	-2.2%	0.1142	0.1131	-1.0%
rFSV	*Nu*	22.42	21.98	-2.0%	0.0891	0.0841	-5.6%
	A	70.13	59.35	-15.4%	0.2378	0.2296	-3.4%
	M	14.34	14.05	-2.0%	0.2754	0.263	-4.5%
	V0	6.91	4.6	-33.4%	0.0086	0.0058	-32.6%
	H	52.38	35.82	-31.6%	0.082	0.0721	-12.1%
rBergomi	*Rho*	146.73	198.48	35.3%	0.1544	0.1494	-3.2%
	A	36.47	30.64	-16.0%	0.09	0.0813	-9.7%
	Xi	3.85	8.81	128.8%	0.0234	0.0253	8.1%
	Eta	50.58	39.68	-21.6%	0.1694	0.1461	-13.8%

Comparison of MAPE and Standard deviation of errors between the default CNNs and the ones trained with batch normalization.

To examine why batch normalization can improve performance while dropout and weight decay do not, the training and validation loss is plotted for each epoch of the training process. Figure 13 shows this for the rFSV model, the other three volatility models can be found in Appendix A5. Since the training and validation loss is essentially identical for each epoch, there are no signs of overfitting, this is also true for the other models. This might explain why dropout and weight decay does not improve performance as they are methods to reduce overfitting. If no signs of overfitting are present, it makes sense that these methods should not improve the model performance. The gains from batch normalization must therefore be that the standardization of the output from intermediate layers makes training the network easier.

Figure 13: rFSV - Effect of regularization on loss during training

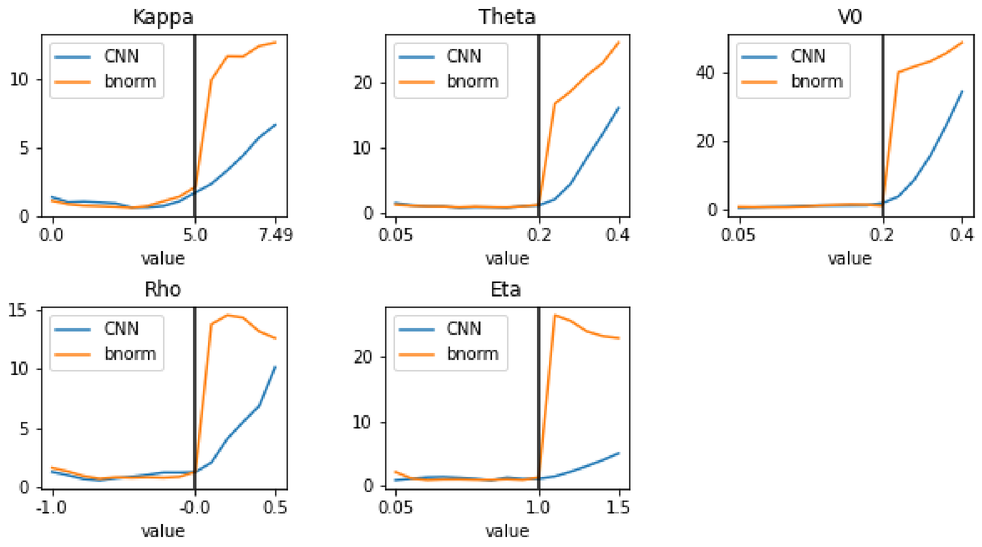


Training and validation loss by epoch for rFSV one-step networks.

4.2.3 Extrapolation

The CNNs capability of extrapolation is examined similarly to the MLPs by using the extended dataset where call grids were simulated with parameters outside their training ranges. Since batch normalization was shown to improve model performance, extrapolation performance is examined for both the default CNNs and the ones extended with batch normalization. Figure 14 examines the mean absolute error for the Heston CNNs relative to the average error across all parameter values. The black line represents the split between training and extrapolation ranges, with extrapolation being to the right. For all parameters, the errors increase as the parameters exceed their training ranges. An interesting observation is that while batch normalization has the best performance in-sample it is substantially worse performing in an extrapolation setting.

Figure 14: Heston - Average error by parameter value

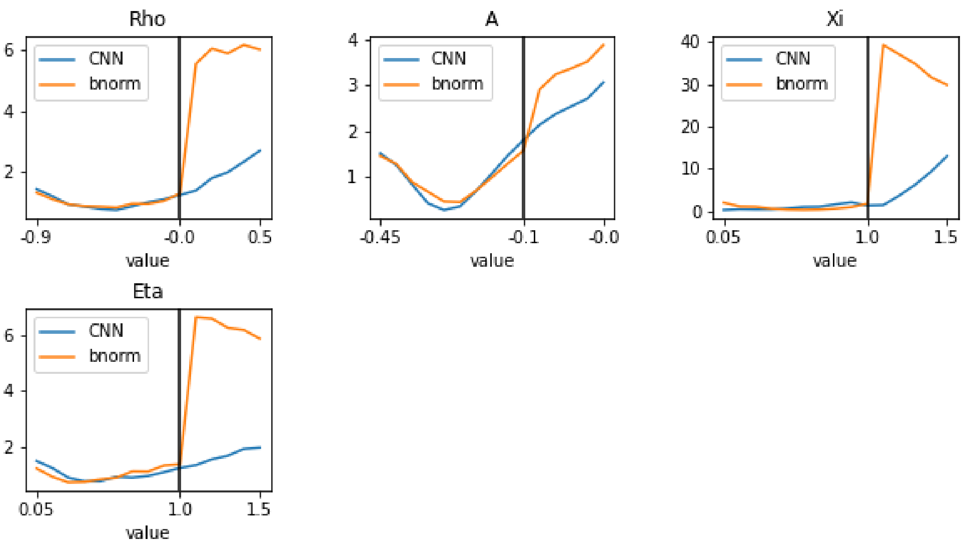


In-sample and extrapolation accuracy for CNN and batch normalized CNN for Heston by parameter value. Extrapolation performance is displayed to the right of the black line.

The magnitude of the extrapolation errors shows that the CNNs are not suited for extrapolation. With errors up to 40 times the in-sample error, practical implementations of the one-step approach would need to restrict parameter estimates to be inside the training regions. The issue with the CNN architecture of (Bayer et al., 2019) is that parameter restrictions cannot be imposed on the network. To incorporate parameter restrictions, the activation function of the output layer must be changed to one that maps inputs to a bounded set. One choice could be the sigmoid function that outputs values between 0 and 1. Using this the network is guaranteed not to choose parameters outside the training ranges.

Figure 15 displays the same figure as before, but for the rBergomi model. Again, increase as parameters exceed training ranges and batch normalization performs worse than the baseline. The magnitude however is quite different for the default rBergomi CNN compared to Heston. For η and ρ the extrapolation performance is similar to the in-sample performance, whereas a has slightly higher errors. The only parameter with unsatisfactory high errors is ξ.

Figure 15: rBergomi - Average error by parameter value



In-sample and extrapolation accuracy for CNN and batch normalized CNN for Heston by parameter value. Extrapolation performance is displayed to the right of the black line.

Thus, it seems that extrapolation is somewhat possible for the CNN, but only for the rBergomi model. This is interesting as the rBergomi model was also the only model with extrapolation capabilities for the two-step approach networks. The extrapolation loss for rFSV can be found in Appendix A6, which essentially shows the same result as Heston, which is poor extrapolation performance. The reason why rBergomi is the only volatility model that allows for extrapolation for both the one- and two-step approaches is unclear and left for future research. One explanation might be that it has the lowest number of parameters whereby the networks might easier learn the relationship between parameters and prices.

4.3 Parameter calibration

With the models evaluated for the two approaches, it is time to examine how they fare in an actual calibration setting. First, the two-step networks are used to calibrate parameters using the simulated call prices of the test set. This makes it possible to examine how well the two-step approach calibrates parameters when the models are correctly specified. The models can be considered correctly specified as the test data stem from the same model specification they are trained upon. After examining the calibration performance on the test set, real data from SPX calls are used. Here, the goal is to examine how well the models fit the SPX implied volatility surface given the calibrated parameters.

4.3.1 Calibration to simulated data

Table 7 displays the calibration performance for the two MLPs of the two-step calibration approach, along with the batch normalization network of the one-step approach. For both MAPE and standard deviation, the smallest error of each parameter is highlighted for ease of interpretation. The batch normalized version of CNN (CNN BN) was chosen rather than the default CNN because of its improved in-sample performance. The calibration algorithm used is the BFGS optimizer, which does not implement restrictions on the parameters. The reason for not restricting parameters here was to assess how well the two-step approach calibrates in a setting where the true parameters are known to be inside the training ranges. By examining the calibration errors for the different model parameters, the CNN BN is shown to outperform the calibration accuracy of the two-step approach. In fact, for all parameters except xi of the rBergomi model, CNN BN has the lowest performance based on MAPE. Looking at the standard deviation of calibration errors and the CNN BN consistently outperforms the other networks on all parameters. This is quite interesting as the results differ from other articles in the literature like (Bayer et al., 2019) whose results favor the two-step approach. Looking at the difference in performance for individual parameters, there are relatively large differences in performance. For the Heston model, pNN has a MAPE of 505%, gNN has 346% and CNN BN only 85%. This large difference in performance is found for almost all parameters, and on average, the CNN BN has 3.4 times lower MAPE than pNN and 3.6 times lower MAPE than gNN. The same pattern is observed for the standard deviation, where CNN BN has on average 3.9 times lower standard deviation than pNN and 4.3 times lower than gNN.

Table 7: Calibration error on simulated data

		MAPE			Std		
Model	**Parameter**	**pNN**	**gNN**	**CNN BN**	**pNN**	**gNN**	**CNN BN**
Heston	*Kappa*	505.2%	346.1%	**85.2%**	2.045	2.356	**0.438**
	Theta	23.5%	30.3%	**8.0%**	0.037	0.049	**0.013**
	V0	23.0%	10.6%	**3.4%**	0.016	0.010	**0.002**
	Rho	547.8%	418.8%	**153.3%**	0.307	0.481	**0.092**
	Eta	84.8%	85.0%	**12.9%**	0.426	0.296	**0.038**
rHeston	*V0*	144.3%	568.7%	**58.5%**	0.288	1.013	**0.098**
	Alpha	81.7%	334.4%	**9.0%**	0.508	1.908	**0.079**
	Lambda	195.6%	1963.7%	**64.0%**	0.939	7.023	**0.257**
	V_bar	204.2%	981.5%	**57.6%**	0.464	2.080	**0.100**
	Xi	235.8%	581.3%	**63.5%**	0.934	2.322	**0.236**
	Rho	97.9%	200.2%	**14.5%**	0.181	1.635	**0.113**
rFSV	*Nu*	77.7%	102.3%	**22.0%**	0.269	0.403	**0.084**
	A	115.8%	135.7%	**59.4%**	0.428	0.503	**0.230**
	M	31.5%	44.3%	**14.1%**	0.549	0.824	**0.263**
	V0	21.4%	27.2%	**4.6%**	0.033	0.041	**0.006**
	H	104.0%	138.2%	**35.8%**	0.235	0.313	**0.072**
rBergomi	*Rho*	236.2%	402.6%	**198.5%**	0.375	0.441	**0.149**
	A	74.0%	94.0%	**30.6%**	0.211	0.247	**0.081**
	Xi	**4.4%**	8.1%	8.8%	0.031	0.039	**0.025**
	Eta	96.5%	167.3%	**39.7%**	0.351	0.531	**0.146**

Calibration error for the approaches: Piecewise (pNN), Gridwise (gNN) and the batch normalized one-step (CNN BN). Errors are measured as the mean absolute percentage error (MAPE) and standard deviation of error (Std). Lowest error for each metric is highlighted.

These large differences in performance are promising for the one-step approach, but while the CNN BN is the best performing model it still has relatively large errors for some parameters. With MAPEs of 85% for Heston's κ or 198% for rBergomi's ρ the network is likely still not performing adequately to be used in practical applications.

Going back to the two-step approach, there can be different reasons why the observed performance is so poor. Firstly, it was observed that both the pNN and gNN perform poorly when predicting the prices of high strike and short maturity calls. This high error for some calls might overly affect the calibration algorithm as it tries to minimize the average loss across all calls. Therefore, these few incorrectly priced calls might dominate the loss whereby the resulting parameters are calibrated to minimize only the loss on those calls rather than the full call grid. To overcome this issue, it might be beneficial to change the loss function from average error to a weighted average error. In this way, the algorithm is forced to put less attention on incorrectly priced calls. There are multiple ways to set the weights on each call, but an example could be to weigh the calls based on their training loss. This weighting scheme then puts more weight on correctly predicted calls and hopefully creates higher precision for the calibration algorithm.

The second reason why the calibration errors are high might be the choice to not restrict the parameters. It was previously established that network performance is poor when parameters outside the training ranges are used. Therefore, by not restricting parameters, the calibration algorithm might pick values outside the training ranges where network performance is unknown. This means that the algorithm might find some parameters that minimize the loss but are so far outside the training ranges that the price approximation fails.

Table 8 examines the calibration accuracy when using the restricted calibration algorithm L-BFGS-B. Note that the one-step approach does not allow for restricting parameters as the calibration procedure is implicit by the network. Restricting the parameters does improve the performance for pNN and gNN, but not on all parameters. Rho and eta for Heston have worse performance with the restricted calibration, the same is true for V0 for rFSV and the parameters A and xi for rBergomi. As before, the lowest error for each parameter is highlighted, and again the CNN BN network has the highest performance for most parameters. The restricted calibration is only able to achieve the lowest error for two parameters, lambda, and V bar for rHeston.

If one compares the performance of pNN and gNN, they both achieve very similar MAPEs, thus in terms of accuracy neither outperforms the other. Beyond comparing calibration accuracy, it might also be important for practical use cases to know the calibration speed. Table 9 lists the calibration speeds of pNN and gNN for the four volatility models. The calibration speed of the CNN was excluded as it is practically instantaneous.

For all volatility specifications, gNN is clearly faster by a large margin and on average 20-30 times faster than pNN. With a speed of calibration between ~9-20 seconds for pNN it is still much faster than using a traditional Monte Carlo calibration. Monte Carlo calibration refers to using the Monte Carlo simulation to calculate prices given some model parameters instead of using the neural networks during the calibration algorithm. While it would be nice to know the speed up the neural network approach gives, this could not be examined due to time constraints. For reference, however, it took about one minute to calculate the call grid given a single set of parameters for the rough volatility models. Since the calibration algorithm needs to calculate many call grids to find the optimal parameters, the speedup is clearly quite significant.

Table 8: Restricted calibration error on simulated data

		Unrestricted		Restricted		
		MAPE		MAPE		MAPE
Model	**Parameter**	**pNN**	**gNN**	**pNN**	**gNN**	**CNN BN**
Heston	*Kappa*	505.2%	346.1%	212.6%	212.6%	**85.2%**
	Theta	23.5%	30.3%	29.3%	33.4%	**8.0%**
	V0	23.0%	10.6%	19.5%	36.7%	**3.4%**
	Rho	547.8%	418.8%	792.3%	831.2%	**153.3%**
	Eta	84.8%	85.0%	105.6%	105.6%	**12.9%**
	Average	236.9%	178.2%	231.8%	243.9%	**52.6%**
rHeston	*V0*	144.3%	568.7%	82.5%	64.5%	**58.5%**
	Alpha	81.7%	334.4%	15.5%	10.6%	**9.0%**
	Lambda	195.6%	1963.7%	77.9%	**61.4%**	64.0%
	V_bar	204.2%	981.5%	**52.8%**	**52.8%**	57.6%
	Xi	235.8%	581.3%	68.9%	73.9%	**63.5%**
	Rho	97.9%	200.2%	14.6%	16.5%	**14.5%**
	Average	159.9%	771.6%	52.0%	46.6%	44.5%
rFSV	*Nu*	77.7%	102.3%	83.8%	83.8%	**22.0%**
	A	115.8%	135.7%	92.2%	92.2%	**59.4%**
	M	31.5%	44.3%	30.8%	30.8%	**14.1%**
	V0	21.4%	27.2%	61.8%	61.8%	**4.6%**
	H	104.0%	138.2%	59.7%	57.1%	**35.8%**
	Average	70.1%	89.5%	65.7%	65.1%	**27.2%**
rBergomi	*Rho*	236.2%	402.6%	292.1%	292.1%	**198.5%**
	A	74.0%	94.0%	121.8%	121.8%	**30.6%**
	Xi	**4.4%**	8.1%	100.9%	100.9%	8.8%
	Eta	96.5%	167.3%	100.0%	100.0%	**39.7%**
	Average	102.8%	168.0%	153.7%	153.7%	**69.4%**

Calibration error for the approaches: Piecewise (pNN), Gridwise (gNN) and the batch normalized one-step (CNN BN). Errors for pNN and gNN are shown for both restricted and unrestricted calibration. Errors are measured as the mean absolute percentage error (MAPE). Lowest error for each parameter is highlighted.

Table 9: Calibration speed

	Heston	**rHeston**	**rFSV**	**rBergomi**
pNN	12.80	20.54	13.95	9.52
gNN	0.49	0.90	0.45	0.33

Average calibration speed in seconds for each volatility model and each two-step approach.

4.3.2 Calibration to SPX

To this point the different models have only been tested on simulated data. Now they will be used to calibrate parameters to the call surface of the SPX to see whether they can fit the observed implied volatility surface. Table 10 displays the calibrated parameters for each volatility model. The calibration algorithm here is the BFGS that does not restrict parameters. The parameter values, which are highlighted in red indicate they are outside their training ranges. Both pNN and gNN have parameters outside their ranges for all volatility models, which is especially bad for the rHeston model, where all parameters are outside their ranges. The CNN BN network also has issues with parameters being outside the training ranges for all models but rHeston.

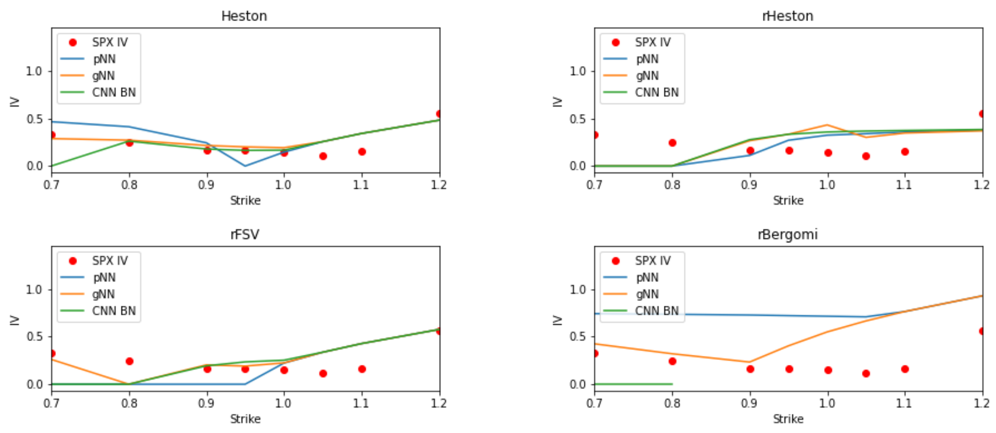
Table 10: Calibrated parameters for SPX

Model	Parameter	pNN	gNN	CNN BN
Heston	*Kappa*	5.9990	2.6245	1.9283
	Theta	0.0363	0.0726	0.0732
	V0	0.0216	0.0240	-0.0161
	Rho	-0.5449	-0.3825	-0.7006
	Eta	0.7215	1.0330	1.5594
rHeston	*V0*	-0.0115	-1.9993	0.2257
	Alpha	-0.1614	3.7892	0.7658
	Lambda	1.5677	3.9158	0.5344
	V_bar	-0.3963	-2.8725	0.2153
	Xi	1.0594	1.0666	0.4729
	Rho	0.4764	1.7006	-0.7021
rFSV	*Nu*	0.6656	0.8812	-0.1891
	A	0.4631	0.8408	1.7546
	M	-1.7934	-2.7130	-0.3648
	V0	0.0258	0.0510	0.0340
	H	0.3012	0.6268	0.0092
rBergomi	*Rho*	-0.6442	0.1037	-55.8523
	A	-0.2607	-0.1018	19.7685
	Xi	0.0301	0.0534	0.3610
	Eta	0.6483	1.0487	-71.0273

Volatility parameters for each approach when calibrated to the SPX index. Values colored red indicate training range has been exceeded.

The calibrated parameters are then used to calculate the implied volatility surface for each of the four volatility models. Since the one-step approach does not allow for calculating the implied volatility surface, the gNN model is used here but with the calibrated parameters from the CNN BN. Figure 16 displays the fit to the implied volatility surface for a single maturity of 0.2 years. The red dots indicate the observed implied volatility values for different strikes, and the blue, orange and green curves show the implied volatility fit of pNN, gNN and CNN BN, respectively.

Figure 16: SPX fit from unrestricted calibration for maturity = 0.2 years



The resulting fit from the unrestricted calibration the observed SPX implied volatilities.

The fit for all models is quite bad, and the implied volatility curves are not well behaved, as they are expected to be smooth. The issue here is likely that the parameters of the calibration algorithm were not restricted. Due to the high number of parameters outside their training ranges, it makes sense that the networks can not produce well-behaved implied volatility curves. To overcome this issue, a restricted calibration is run using the L-BFGS-B algorithm, and the results are displayed in Table 11.

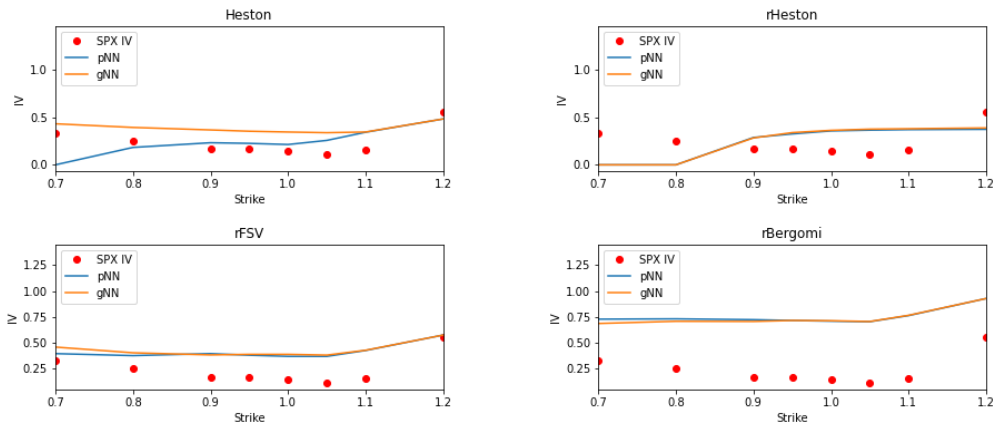
Table 11: Restricted calibration for SPX

Model	**Parameter**	**pNN**	**gNN**
Heston	*Kappa*	2.5270	2.5270
	Theta	0.0555	0.1239
	V0	0.0500	0.1228
	Rho	-0.6118	-0.4948
	Eta	0.5188	0.5188
rHeston	*V0*	0.0500	0.2439
	Alpha	0.6234	0.9000
	Lambda	0.5709	1.0000
	V_bar	0.2252	0.2252
	Xi	0.2881	0.5321
	Rho	-0.7607	-0.6769
rFSV	*Nu*	0.5363	0.5363
	A	0.5406	0.5406
	M	-1.4929	-1.4929
	V0	0.1396	0.1396
	H	0.2851	0.2226
rBergomi	*Rho*	-0.4454	-0.4454
	A	0.0500	0.0500
	Xi	0.5228	0.5228
	Eta	0.5198	0.5198

This table display parameters obtained using the restricted calibration algorithm L-BFGS-B, for the picewise approach (pNN) and gridwise approach (gNN)

With all parameters within their training ranges, the implied volatility fit is again evaluated and displayed in Figure 17. Here, CNN BN was excluded due to the inability of restricting its parameters. With the restricted parameters, the predicted implied volatility curves are much more well behaved, except for the rHeston fit. Note that the bad fit of the rHeston is likely due to the generally bad performance of the rHeston model, as previously established.

Figure 17: SPX fit from restricted calibration for maturity = 0.2 years



The resulting fit from the restricted calibration the observed SPX implied volatilities.

Although the volatility curves are more well behaved, they do however not fit the observed implied volatility curve well. Given the poor calibration performance in-sample for the two-step approaches, the poor fit might originate from innacurate calibrated parameters. Another explanation might be that the true parameters are outside the training ranges, which means the networks cannot produce a better fit due to the parameter restriction.

The above results from trying to fit the observed implied volatility surface highlight some important practical issues with implementing a deep learning approach for option calibration. Firstly, the parameters must be restricted to ensure that the predicted implied volatilities are well behaved. Secondly, the restriction on parameters means that the user must know beforehand what the optimal parameter ranges are. While a solution might be to set large ranges for parameters during the simulation phase, this introduces problems on its own. It was previously found that the simulation functions encounter numerical instability for bad parameters. Therefore, if the solution is to set large training ranges, it might be difficult to simulate call prices for many parameter combinations.

5. Conclusion

This thesis examined the two main calibration approaches found in the literature for option pricing using neural networks, namely the one-step approach of (Dimitroff et al., 2018) (CNN) and the two-step approaches of (Bayer & Stemper, 2018) (pNN) and (Horvath et al., 2021) (gNN). The goal was to thoroughly examine how the competing approaches compared and how different circumstances affected their performance. To examine this, four volatility specifications were considered for each approach. The rough volatility models rHeston, rFSV and rBergomi and the classic Heston stochastic volatility model. For each of these approaches and volatility specifications, the regularization techniques, batch normalization, dropout and weight decay were tested to examine how they affected model performance. The networks were trained on simulated data from a restricted parameter region, and testing was performed using both simulated and real-world data from the SPX index.

The central motivation for applying the neural network approach for calibration is to resolve the computational bottleneck that arises from simulating paths of rough volatility models. Multiple studies have shown that calibration speed can be vastly improved using a neural network, which was confirmed in this thesis. This thesis achieved calibration speeds between 9 and 20 seconds for the pNN approach, 0.5-0.9 seconds for gNN, and were practically instantaneous for the CNN.

Two-step approach

On simulated data, the two-step approaches, pNN and gNN, were shown to accurately approximate the price for most of the calls on the simulated grids. Both approaches, however, had especially high errors for the high strike and low maturity calls for all volatility specifications. The pricing approximation for rHeston was shown to be generally poor, but due to simulation issues, it is unclear whether the data is too noisy or the rHeston simply is more difficult to approximate. Simulation for rHeston was especially tricky as parameter ranges had to be modified ad hoc to ensure the simulation function did not fail.

The effect of regularization was examined for both pNN and gNN, but results showed that performance in the best cases was on par with the baseline and often significantly reduced. This finding is in line with the literature where (Bayer & Stemper, 2018) find that batch normalization decreases performance and (Hernandez, 2016) find that increasing the dropout rate also leads to poor performance. The approximation error of the networks was further explored by tracking the error on parameter values. This showed interestingly that parameter values often have a significant effect on the accuracy of the approximation. This finding is useful for practical applications as it gives insights into how uncertain a set of calibrated parameters are. If the true parameters are in a region where the network poorly performs, the calibration step might not find the correct parameters as prices are poorly approximated in the desired region. Lastly, the extrapolation capabilities of gNN and pNN were explored using a simulated dataset with extended parameter ranges. Here results showed extrapolation was possible for the rBergomi networks but not any other network.

Across the different performance tests, pNN and gNN were compared to examine which of the two-step variants were best. Results showed pricing errors were remarkably similar and based on total error no variant consistently outperformed the other. Tracking performance on parameter value showed both variants had similar error values both in sample and when extrapolating. Therefore, it is concluded no variant is preferred in terms of pricing accuracy. In practical applications both variants have their own benefits, gNN can be trained faster as well as having a faster calibration speed. pNN on the other hand allows for arbitrary strikes and maturities which means interpolation is not required for calibration.

One-step approach

The networks trained following the one-step approach were likewise tested on simulated data to validate their calibration accuracy. Results show that parameter estimates were centered around the true (simulated) parameter, albeit with varying precision. Some parameters were calibrated to reasonably high accuracies, while others had unsatisfactory high MAPEs. Examining the effect of the three different regularization techniques showed that batch normalization was the only technique that significantly improved performance. The performance improvements of batch normalization are promising for the one-step approach but there were still parameters with unsatisfactory high errors. The CNN error was also examined based on parameter value for both in-sample and the extrapolation set. The in-sample error generally showed a quadratic relationship between the parameter value and mean absolute error. The minimum error was observed when the parameter values were in the middle of their training ranges, with the highest error at the borders. The extrapolation set showed some extrapolation capability for rBergomi, but none of the other volatility specifications.

Calibration

With the performance of each approach established, their calibration accuracy was compared using the simulated data. Here the batch normalized version of the one-step approach (CNN BN) was compared to an unrestricted implementation of the two-step approaches. On a parameter level, CNN BN achieved the lowest MAPE on all parameters except ξ of rBergomi. Across all parameters, CNN BN achieved 3.4 times lower MAPE than pNN and 3.6 times lower than gNN. It was hypothesized that the unrestricted calibration might negatively affect accuracy as the algorithm might pick parameters where the network was ill-defined. Running a restricted calibration for the two-step approach showed some improvements but not on all parameters. The performance of the CNN BN was still significantly better than the restricted calibration results of the two-step approaches.

Calibration was also performed to the observed prices of the SPX index. Initially, an unrestricted calibration was used for the two-step approaches, along with the CNN BN of the one-step approach. Results showed that parameters were often chosen outside their training ranges for all three approaches and across all volatility specifications. Only the CNN BN rHeston model had all calibrated parameters inside training ranges. Given the calibrated parameters, a slice of the implied volatility surface was predicted and compared with the observed implied volatilities. This resulted in a poor fit where the predicted implied volatility curves were not well behaved and did not exhibit the well-known smile. Given the generally poor extrapolation performance, it was no surprise this unrestricted calibration were unable to fit the observed surface. By using a restricted calibration, the resulting implied volatility curves were more accurate as they exhibited the volatility smile, although they were still not able to fit the observed values. Fitting the observed SPX index highlights some important practical considerations. Firstly, the architecture used for the CNN of (Bayer et al., 2019) does not allow for parameter restrictions. Given the poor extrapolation performance, a different architecture should be implemented that restricts parameter estimates. A practical example could be to use an activation function for the output which limits estimates to be between 0 and 1, such as the sigmoid function. When the output is rescaled, this sigmoid function then ensures the CNNs cannot predict parameters outside training ranges. A second issue with the CNNs is mapping from parameters to prices is not possible, therefore one cannot evaluate the fit of the parameters without another model. Lastly, the need for restricting parameters for pNN and gNN means that the researcher must know beforehand how to set parameter ranges. The ranges must be broad enough to ensure calibration accuracy, though as observed during the simulation process, this can lead to numerical instability.

6. Future research

This section briefly explains interesting extensions that due to the time limit of this thesis could not to be implemented.

In this thesis, the neural network architectures were chosen based on those used in the literature. This was done to keep the focus on the approximation accuracy of the networks, without the added complexity of how architecture affected performance. For future research, an obvious question is whether performance can be improved by changing the architecture and how different architectures compare. To answer this question a grid search could be used to examine how layers, number of neurons, activation function, and so on affect different metrics. A grid search is implemented by iteratively training the networks from a grid of specified parameters. The resulting performance of each parameter combination is then evaluated using either cross-validation or a test set, as is done in this thesis. Then ,the performance measure can then be chosen to reflect the question of interest. Examples of this could be the pricing accuracy of high strike, short maturity calls, which were shown to be poor in this thesis. Another grid search could examine which architecture results in the best extrapolation performance or the best fit to the SPX index. This approach would then allows the researcher to examine whether there is a single best architecture for all problems, or whether different architectures are requred for different tasks. Lastly, instead of using a grid search, a more sophisticated search for the optimal parameters could be implemented, see for example HyperOpt (Bergstra et al., 2013) or Optuna (Akiba et al., 2019).

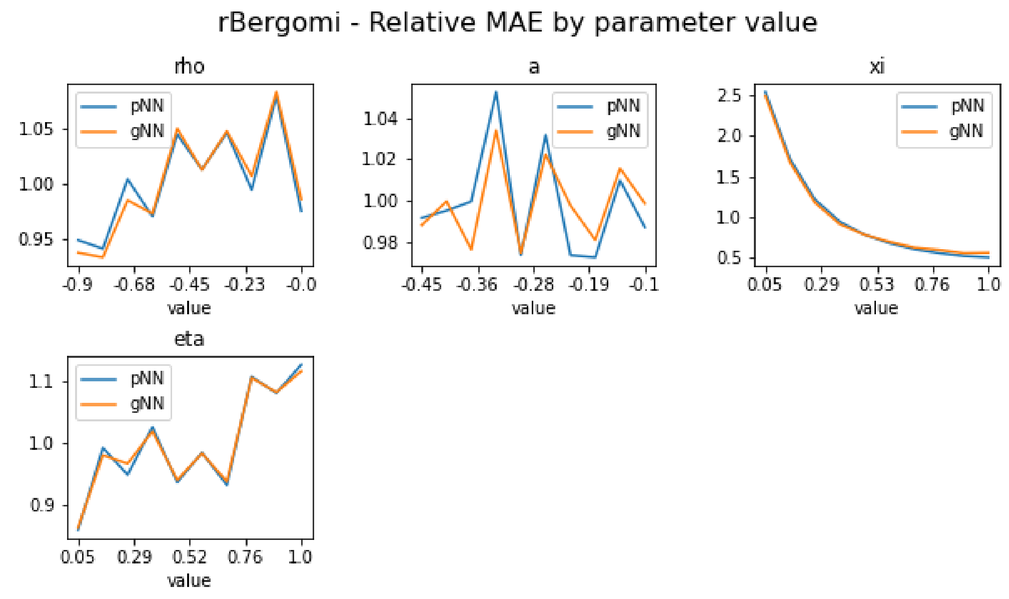
Besides examining network architectures, further research might examine how other approaches than the ones considered here affect performance. A recent study by (Huge & Savine, 2020) shows that training of the neural network can be done faster and more accurately using their variant of the two-step approach. This variant replaces the feedforward neural network of the two-step approach with their network called the differential neural network. Here, the labels (output) of the neural network include both prices and the derivatives (Greeks). By adding the Greeks as labels, they show that training converges faster, and it ensures the network has correct Greeks. Implementation is performed using automatic adjoint differentiation (AAD) which in machine learning terminology is known as backpropagation. Since neural networks are differentiable, the backpropagation algorithm gives the models predicted Greeks. The loss function is then augmented to also include the difference between true and predicted Greeks. By ensuring the networks have accurate Greeks, the applicability of the two-step approach is further enhanced as it can be used as a tool for hedging.

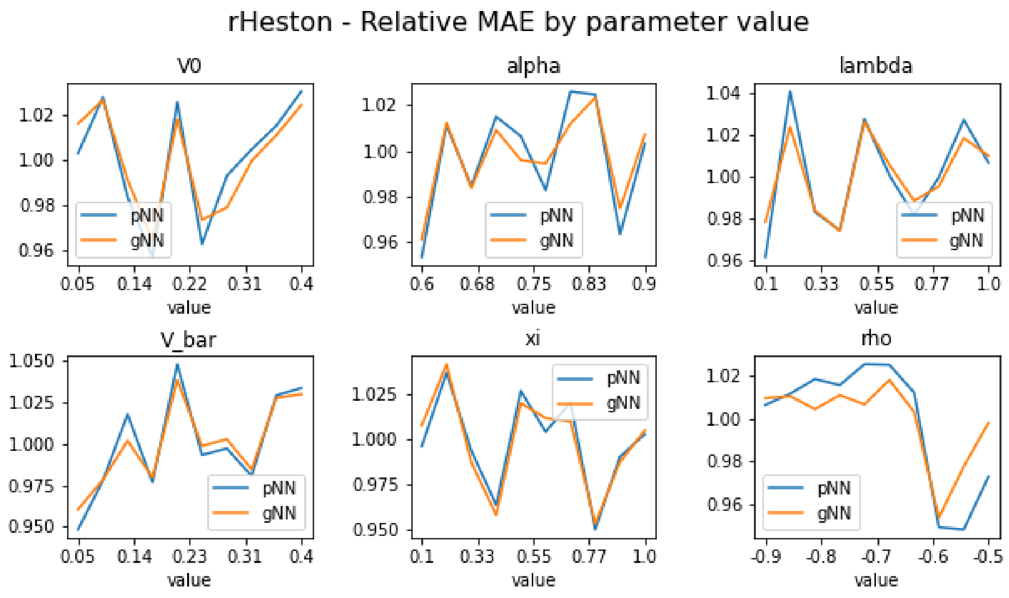
Lastly, an interesting paper by (Itkin, 2019) examines the potential pitfalls of the neural network approach to option pricing. One of the issues highlighted in this paper is how neural networks cannot guarantee that predicted prices are arbitrage-free, both in-sample and out-of-sample. Itkin argues that this issue can be partially resolved by modifying the loss function by increasing the loss when arbitrage constraints are broken. This, he argues, is only a partial solution as even though it might ensure in-sample predictions are arbitrage-free, it gives no guarantee out-of-sample. Further research might explore how to incorporate arbitrage-free price predictions, as well as examine the extent to which current approaches break arbitrage constraints.

Appendix

A1: Loss by parameter value, two-step

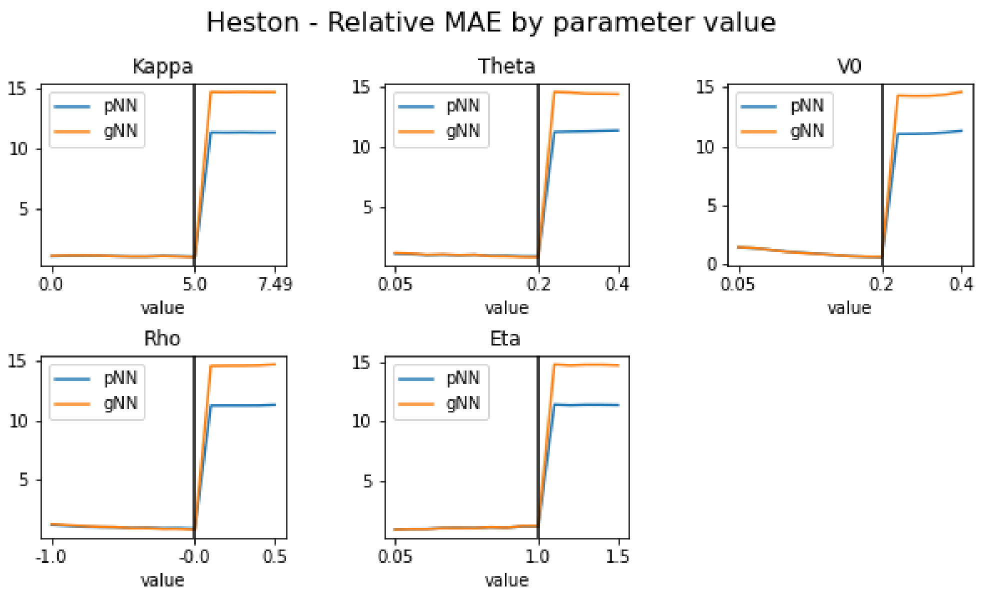
These figures display the in-sample loss for the rBergomi and rHeston model for pNN and gNN. Loss is calculated by parameter value and divided by the total loss for ease of interpretation.





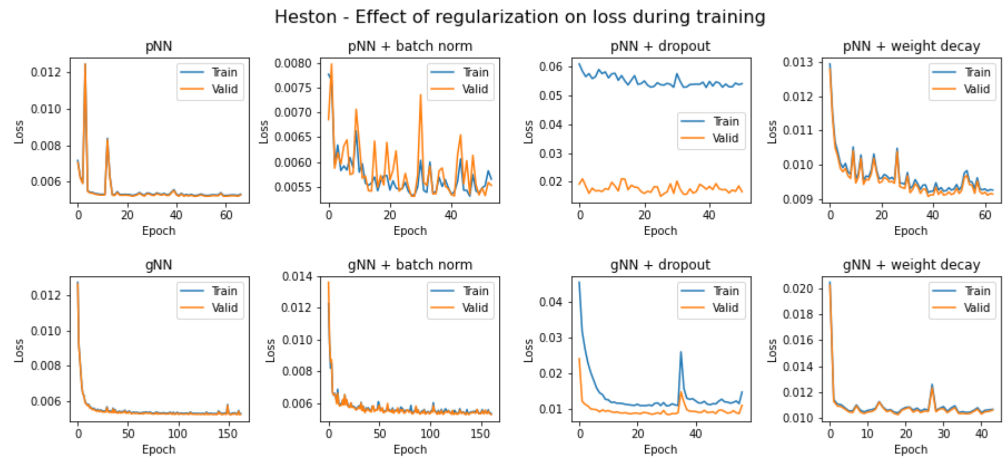
A2: Extrapolation performance, two-step

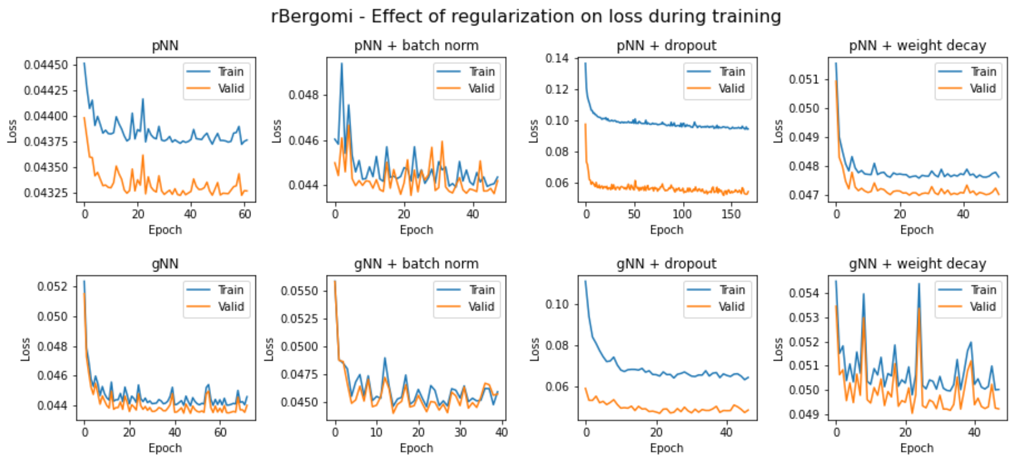
Here the extrapolation loss for the Heston model for both pNN and gNN is displayed. Loss is calculated by parameter value and divided by the total in-sample loss for ease of interpretation.

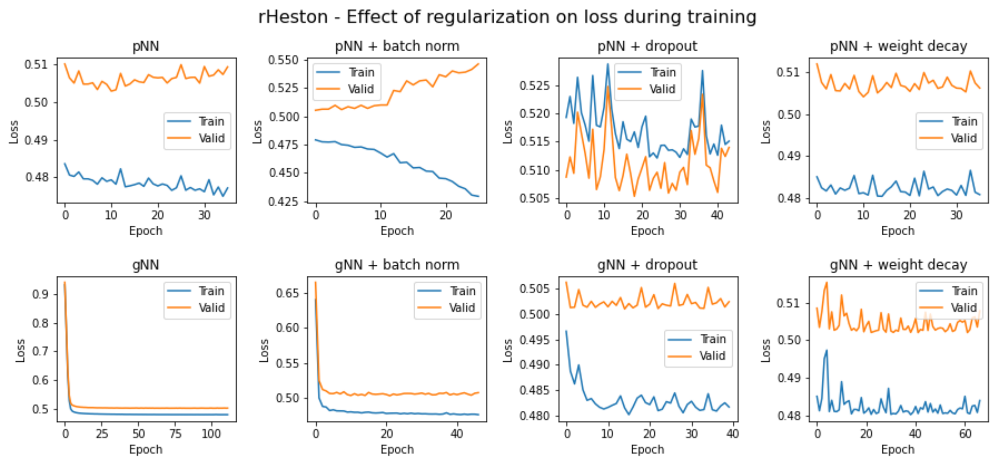


A3: Training and validation loss, two-step

Here the training and validation loss is shown for each epoch during training for the default and regularized networks of Heston, rBergomi and rHeston. The figures display both the values for pNN and gNN.

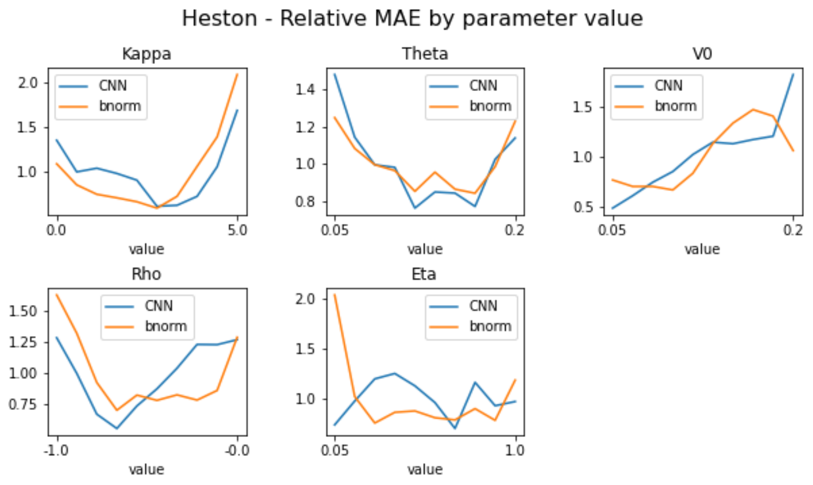


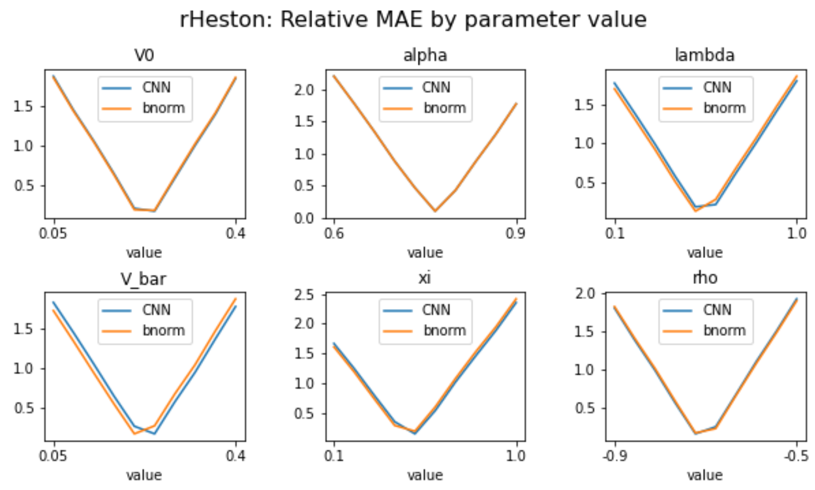


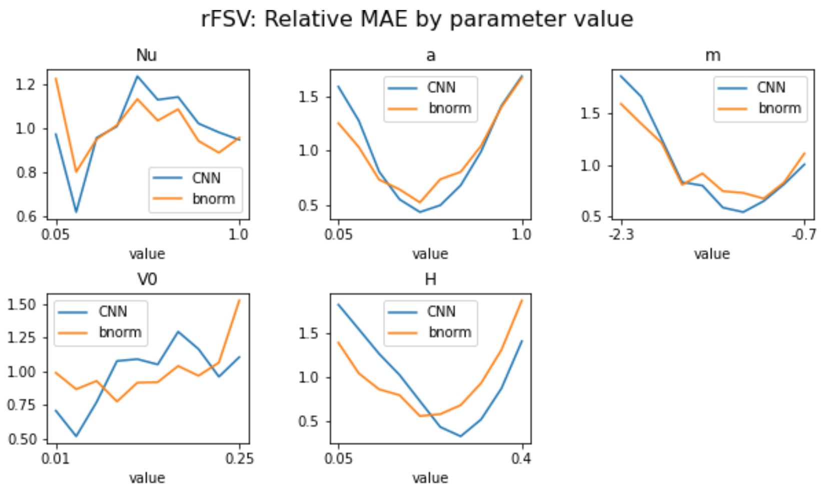


A4: Loss by parameter value, one-step

These figures display the in-sample loss for the Heston, rHeston and rFSV models for the one-step approach. Loss is calculated by parameter value and divided by the total loss for ease of interpretation. The blue lines represent the default CNN and the orange the batch normalized CNN.

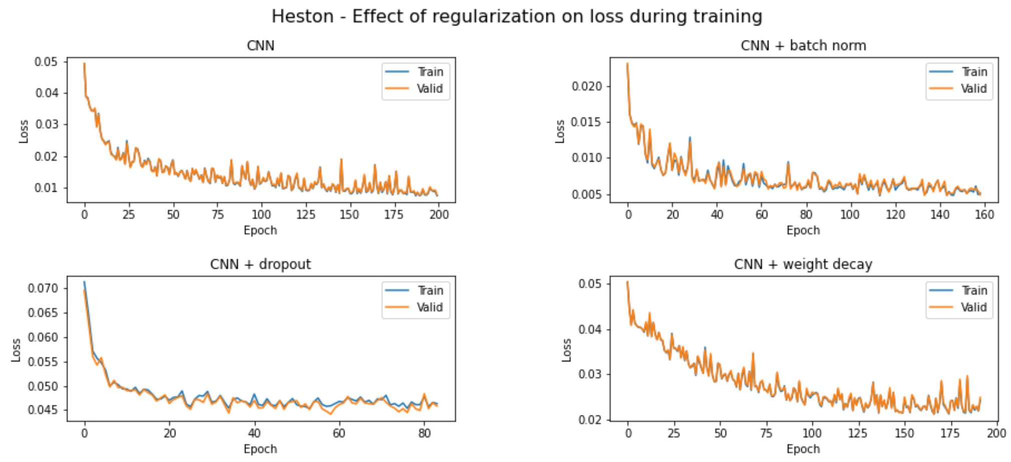


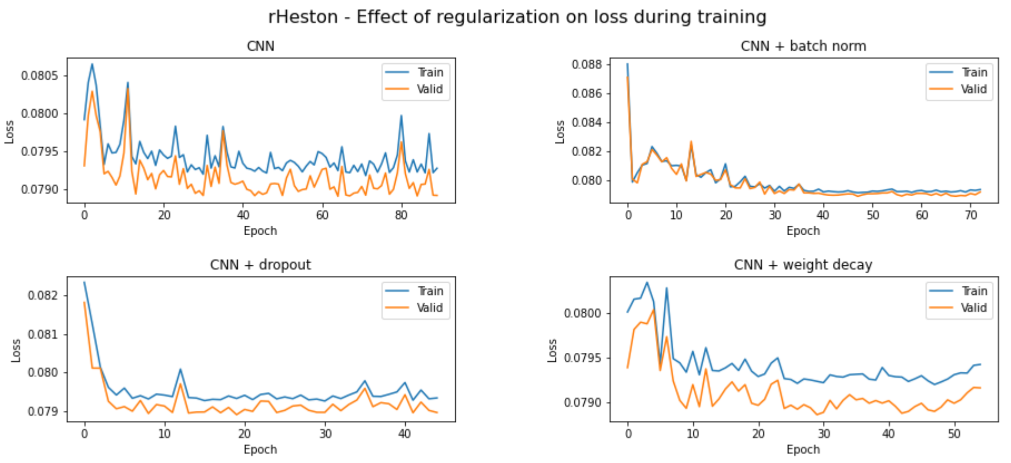


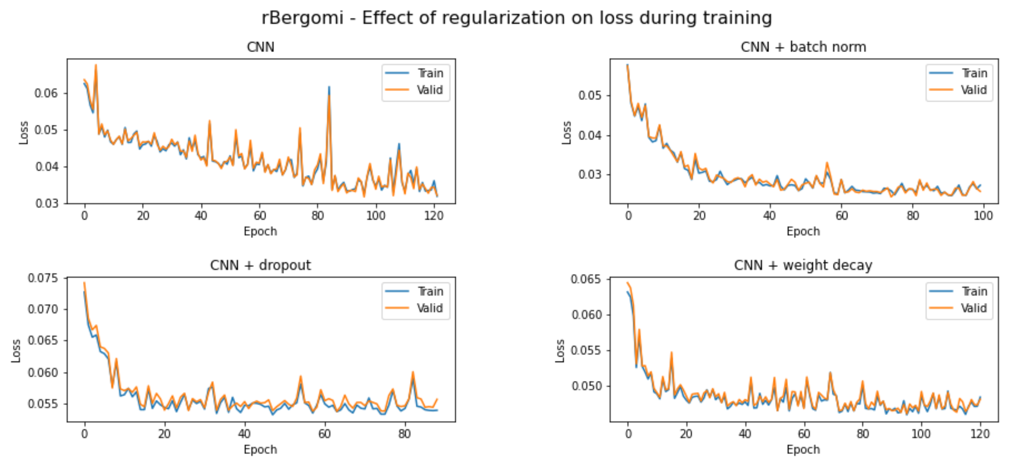


A5: Training and validation loss, one-step

Here the training and validation loss is shown for each epoch during training for the default and regularized networks of Heston, rHeston and rBergomi of the one-step approach.

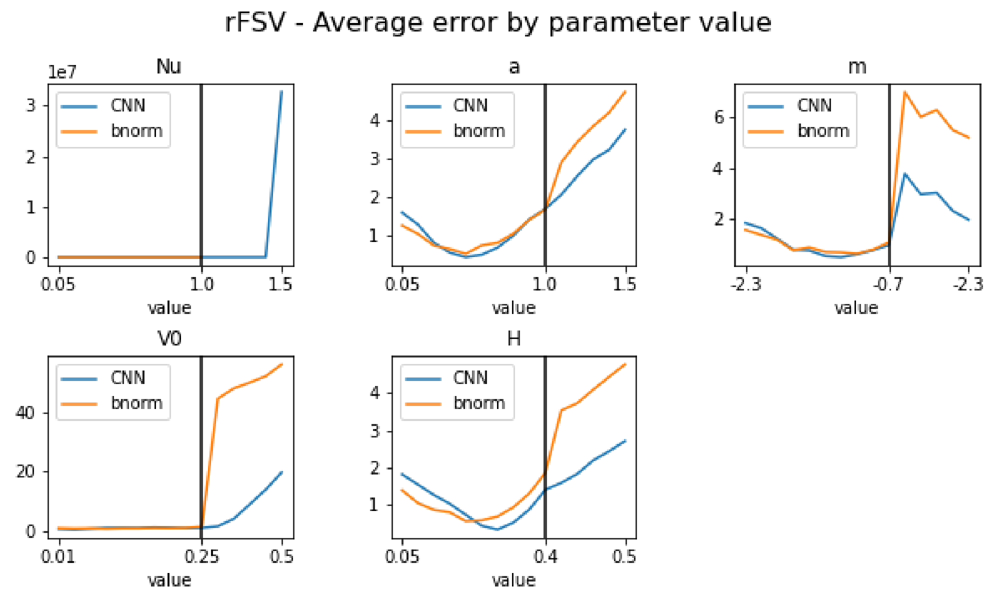






A6: Extrapolation performance, one-step

This figure displays the extrapolation error for the rFSV of the one-step approach. The blue lines represent the default CNN and the orange the batch normalized CNN.



References

Akiba, T., Sano, S., Yanase, T., Ohta, T., & Koyama, M. (2019). Optuna: A Next-generation Hyperparameter Optimization Framework. *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, 2623–2631. https://doi.org/10.1145/3292500.3330701

Alòs, E., & León, J. A. (2021). An Intuitive Introduction to Fractional and Rough Volatilities. *Mathematics*, *9*(9), 994. https://doi.org/10.3390/math9090994

Alòs, E., León, J. A., & Vives, J. (2007). On the short-time behavior of the implied volatility for jump-diffusion models with stochastic volatility. *Finance and Stochastics*, *11*(4), 571–589. https://doi.org/10.1007/s00780-007-0049-1

Andreasen, J., & Huge, B. N. (2010). Volatility Interpolation. *SSRN Electronic Journal*. https://doi.org/10.2139/ssrn.1694972

Bayer, C., Friz, P., & Gatheral, J. (2015). *Pricing Under Rough Volatility* (SSRN Scholarly Paper ID 2554754). Social Science Research Network. https://doi.org/10.2139/ssrn.2554754

Bayer, C., Friz, P. K., Gulisashvili, A., Horvath, B., & Stemper, B. (2018). *Short-time near-the-money skew in rough fractional volatility models* (arXiv:1703.05132). arXiv. https://doi.org/10.48550/arXiv.1703.05132

Bayer, C., Horvath, B., Muguruza, A., Stemper, B., & Tomas, M. (2019). On deep calibration of (rough) stochastic volatility models. *ArXiv:1908.08806 [q-Fin]*. http://arxiv.org/abs/1908.08806

Bayer, C., & Stemper, B. (2018). Deep calibration of rough stochastic volatility models. *ArXiv:1810.03399 [Cs, q-Fin]*. http://arxiv.org/abs/1810.03399

Bender, C., & Thiel, M. (2020). Arbitrage-free interpolation of call option prices. *Statistics & Risk Modeling*, *37*(1–2), 55–78. https://doi.org/10.1515/strm-2018-0026

Bennedsen, M., Lunde, A., & Pakkanen, M. S. (2015). Hybrid scheme for Brownian semistationary processes. *Finance and Stochastics*, *21*(4), 931–965. https://doi.org/10.1007/s00780-017-0335-5

Bennedsen, M., Lunde, A., & Pakkanen, M. S. (2017). Hybrid scheme for Brownian semistationary processes. *Finance and Stochastics*, *21*(4), 931–965. https://doi.org/10.1007/s00780-017-0335-5

Bennedsen, M., Lunde, A., & Pakkanen, M. S. (2021). Decoupling the short- and long-term behavior of stochastic volatility. *ArXiv:1610.00332 [q-Fin]*. http://arxiv.org/abs/1610.00332

Bergstra, J., Yamins, D., & Cox, D. (2013). *Hyperopt: A Python Library for Optimizing the Hyperparameters of Machine Learning Algorithms*. 13–19. https://doi.org/10.25080/Majora-8b375195-003

Bjorck, J., Gomes, C., Selman, B., & Weinberger, K. Q. (2018). Understanding Batch Normalization. *ArXiv:1806.02375 [Cs, Stat]*. http://arxiv.org/abs/1806.02375

Black, F., & Scholes, M. (1973). The Pricing of Options and Corporate Liabilities. *The Journal of Political Economy*, *81*(3), 637–654.

Bolko, A. E., Christensen, K., Veliyev, B., & Pakkanen, M. (2020). Roughness in Spot Variance? A GMM Approach for Estimation of Fractional Log-Normal Stochastic

Volatility Models Using Realized Measures. *SSRN Electronic Journal*. https://doi.org/10.2139/ssrn.3708167

Carr, P., & Madan, D. (1999). Option valuation using the fast Fourier transform. *The Journal of Computational Finance*, *2*(4), 61–73. https://doi.org/10.21314/JCF.1999.043

Cox, J. C. (1997). The Constant Elasticity of Variance Option Pricing Model. *The Journal of Portfolio Management*, *23*(5), 15–17. https://doi.org/10.3905/jpm.1996.015

Dimitroff, G., Röder, D., & Fries, C. P. (2018). Volatility Model Calibration With Convolutional Neural Networks. *SSRN Electronic Journal*. https://doi.org/10.2139/ssrn.3252432

Du, K.-L., & Swamy, M. N. S. (2019). *Neural Networks and Statistical Learning*. Springer London. https://doi.org/10.1007/978-1-4471-7452-3

Dupire, B., Black, T. B. M. (see, & Options, G. (1994). Pricing with a Smile. *Risk Magazine*, 18–20.

El Euch, O., & Rosenbaum, M. (2019). The characteristic function of rough Heston models. *Mathematical Finance*, *29*(1), 3–38. https://doi.org/10.1111/mafi.12173

Engle, R. F. (1982). Autoregressive Conditional Heteroscedasticity with Estimates of the Variance of United Kingdom Inflation. *Econometrica*, *50*(4), 987–1007. https://doi.org/10.2307/1912773

Euch, O. E., & Rosenbaum, M. (2017). Perfect hedging in rough Heston models. *ArXiv:1703.05049 [q-Fin]*. http://arxiv.org/abs/1703.05049

Gatheral, J., Jaisson, T., & Rosenbaum, M. (2014). Volatility is rough. *ArXiv:1410.3394 [q-Fin]*. http://arxiv.org/abs/1410.3394

Gatheral, J., Jaisson, T., & Rosenbaum, M. (2018). Volatility is rough. *Quantitative Finance*, *18*(6), 933–949. https://doi.org/10.1080/14697688.2017.1393551

Goodfellow, I. J., Bengio, Y., & Courville, A. (2016). *Deep Learning*. MIT Press.

Hagan, P., Kumar, D., Lesniewski, A., & Woodward, D. (2002). Managing Smile Risk. *Wilmott Magazine*, *1*, 84–108.

Hara, K., Saitoh, D., & Shouno, H. (2016). Analysis of Dropout Learning Regarded as Ensemble Learning. In A. E. P. Villa, P. Masulli, & A. J. Pons Rivero (Eds.), *Artificial Neural Networks and Machine Learning – ICANN 2016* (pp. 72–79). Springer International Publishing. https://doi.org/10.1007/978-3-319-44781-0_9

Hernandez, A. (2016). Model Calibration with Neural Networks. *SSRN Electronic Journal*. https://doi.org/10.2139/ssrn.2812140

Heston, S. L. (1993). A Closed-Form Solution for Options with Stochastic Volatility with Applications to Bond and Currency Options. *Review of Financial Studies*, *6*(2), 327–343. https://doi.org/10.1093/rfs/6.2.327

Hornik, K., Stinchcombe, M., & White, H. (1989). Multilayer feedforward networks are universal approximators. *Neural Networks*, *2*(5), 359–366. https://doi.org/10.1016/0893-6080(89)90020-8

Horvath, B., Muguruza, A., & Tomas, M. (2021). Deep learning volatility: A deep neural network perspective on pricing and calibration in (rough) volatility models. *Quantitative Finance*, *21*(1), 11–27. https://doi.org/10.1080/14697688.2020.1817974

Huge, B., & Savine, A. (2020). Differential Machine Learning. *ArXiv:2005.02347 [Cs, q-Fin]*. http://arxiv.org/abs/2005.02347

Ioffe, S., & Szegedy, C. (2015). *Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift*. 9.

Itkin, A. (2019). Deep learning calibration of option pricing models: Some pitfalls and solutions. *ArXiv:1906.03507 [q-Fin]*. http://arxiv.org/abs/1906.03507

Kingma, D. P., & Ba, J. (2017). Adam: A Method for Stochastic Optimization. *ArXiv:1412.6980 [Cs]*. http://arxiv.org/abs/1412.6980

Kokholm, T. (2016). Pricing and hedging of derivatives in contagious markets. *Journal of Banking & Finance*, *66*, 19–34. https://doi.org/10.1016/j.jbankfin.2016.01.012

Kokholm, T., & Stisen, M. (2015). Joint pricing of VIX and SPX options with stochastic volatility and jump models. *The Journal of Risk Finance*, *16*(1), 27–48. https://doi.org/10.1108/JRF-06-2014-0090

Krogh, A., & Hertz, J. A. (1991). *A Simple Weight Decay Can Improve Generalization*. 8.

Lam, S. K., Pitrou, A., & Seibert, S. (2015). Numba: A llvm-based python jit compiler. *Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC*, 1–6.

Lewis, A. L. (2001). A Simple Option Formula for General Jump-Diffusion and Other Exponential Levy Processes. *SSRN Electronic Journal*. https://doi.org/10.2139/ssrn.282110

Mandelbrot, B. B., & Van Ness, J. W. (1968). Fractional Brownian Motions, Fractional Noises and Applications. *SIAM Review*, *10*(4), 422–437. https://doi.org/10.1137/1010093

Matytsin, A. (1999). *Modelling Volatility and Volatility Derivatives*. 17.

McCrickerd, R. (2017). Rough Bergomi. *Github Repository*. https://github.com/ryanmccrickerd/rough_bergomi

Merton, R. C. (1976). Option pricing when underlying stock returns are discontinuous. *Journal of Financial Economics*, *3*(1), 125–144. https://doi.org/10.1016/0304-405X(76)90022-2

Nocedal, J., & Wright, S. J. (2006). *Numerical optimization* (2nd ed). Springer.

Rawat, W., & Wang, Z. (2017). Deep Convolutional Neural Networks for Image Classification: A Comprehensive Review. *Neural Computation*, *29*(9), 2352–2449. https://doi.org/10.1162/neco_a_00990

Roemer, S. (2021). Rough Heston. *Github Repository*. https://github.com/sigurdroemer/rough_heston

Rosenbaum, M., & Zhang, J. (2021). Deep calibration of the quadratic rough Heston model. *ArXiv:2107.01611 [q-Fin]*. http://arxiv.org/abs/2107.01611

Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I., & Salakhutdinov, R. (2014). Dropout: A Simple Way to Prevent Neural Networks from Overfitting. *Journal of Machine Learning Research*, *15*(56), 1929–1958.

Stone, H. (2020). Calibrating rough volatility models: A convolutional neural network approach. *Quantitative Finance*, *20*(3), 379–392. https://doi.org/10.1080/14697688.2019.1654126