Pricing and Hedging: Application of Deep Learning to the Pricing of Options using Rough Volatility Models

Thesis · July 2019

DOI: 10.13140/RG.2.2.25005.77280

1 author:

Andrei Platonov

1 PUBLICATION **0** CITATIONS

SEE PROFILE

Pricing and Hedging: Application of Deep Learning to the Pricing of Options using Rough Volatility Models

MSc Finance Part-time

Financial Engineering (FM408E)

Candidate number: 21266

2018-19

Word count: 5750

Department of Finance
London School of Economics

Acknowledgements

I would like to sincerely thank my research supervisor Jean-Pierre Zigrand for encouraging me to pursue with this dissertation paper. I appreciate very much his professional assistance that enabled me to achieve excellent results. In addition to that, I am very thankful to my parents for their patience, understanding and all the effort to bring the best out of me. I owe them my determination and perseverance to embark on this academic venture.

Table of Contents

Chapter 1

Pricing and Hedging: Institutional Context

1.1 Background

Pricing models are judged on their robustness; the ability to generate arbitrage-free option prices even when the quality of input data is questionable [1]. Ideally, the calibrated price should always be within the bid-ask spread quoted by the general market. Also, whenever any theoretical uncertainties can be completely eliminated, a good model will allow risk to be completely hedged.

Approximating option prices have been done using various processes and perspectives in recent years. Academics and industry practitioners have thoroughly researched the literature and well-familiarized by risk managers. The usefulness of any derivative pricing model (Monte carlo, PDE, machine learning models, etc) built on the regularity features of a given stochastic model. So when deciding on a choice of a stochastic model, tractability is a major concern; sometimes even more so than predictive accuracy [2][3]. Speed of calibration is a technical matter. A model might be desirable on the grounds that it has sound financial and mathematical properties and also performs with high precision, it may still be impractical if calibration to new data is too time intensive as time is a crucial parameter in the financial industry.

Numerical approximations are typically used to value options. In general, the advanced models are capable of capturing the non-linear characteristics observed in financial data. However, these models are usually multi-dimensional and consequently, often lack closed-form solutions for option prices. In option pricing, the calibration are usually not done using historical prices, but by looking at the current market price of the options. I.e. we need to match the market prices of heavily dealt options to the prices from theoretical models based

on the risk-neutral probability. In practical/institutional applications hundreds of prices need to be computed to fit the model/option parameters. However, due to the requirement of extremely efficient computations, several excellent option models are discarded.

Computational efficiency is critical especially in real time risk-management e.g. high frequency trading and counterparty risk assessment, where we cannot avoid the tradeoff between accuracy and efficiency [4].

The SABR model has become the most widely used benchmark in indexed income desks due to its relative speed in application while the popularity of Heston method is largely due to the convenience of Fourier pricing even though it has some hindrances. Rough volatility models also provide a number of advantages (see [5][6][7]), however, it relies on relatively slower Monte Carlo method which is a considerable limitation in calibrating and has created a bottleneck in industrial applications. Such differences can create situations where it gets difficult to measure the requirements of accuracy VS calibrating speed when deciding between different choice of models.

This issue can be largely mitigated with some methods that directly outputs the prices of options for a set of specified parameters (comparable to the Black-Scholes equation) but for a large set of strikes and maturities. In this paper we will mostly focus on the data-driven application of ANNs using the foundations laid by [2] and [3]. In [2], Hernandez provides an extensive explanation of direct calibration with demonstration on the Hull-White model while in [3], the authors demonstrated a (faster) variation of this technique on a several stochastic volatility models (rBergomi, Bergomi, Heston and SABR models).

Several authors have suggested mapping model parameters to implied volatility surfaces (see [8], [9] and [10] for SSVI, eSSVI surfaces based on stochastic volatility). The direct translation of a chosen parameter vector to several shapes of (no-arbitrage) implied volatility surfaces is done without having to specify any stochastic processes of the underlying asset. A direct translation from parameter vectors to IV surfaces can also be obtained in stochastic models with asymptotic expansions (e.g. the SABR equation), however, the issue is that such asymptotic method has restricted validity in the surfaces because they are usually naturally restricted to specified asymptotic regimes. It follows that a direct surface mapping using several combinations of parameters of stochastic models to different shapes of the implied volatility surfaces is clearly advantageous in modelling. The possibility of linking volatility surfaces with stochastic dynamics is combined with the benefits of direct parametric volatility surfaces of a given underlying asset.

1.2 Rough volatility models

Rough volatility provides an improved representation of historical returns as a result of which we expect to achieve more accurate prediction of future behaviour of volatility/returns. Particularly, we hope that it provides a better estimation of IV surface. Not only it closely aligns with the scaling that is evident in time series, it also provides a foundation to amend the existing pricing models to estimate the IV surface with high accuracy.

[11] showed that, typical behaviour of market agents in a high frequency scale, leads to a rough volatility. Several articles like [12][11][13][14], and have shown how we can adapt classical models like Heston and Bergomi to the rough fractional stochastic volatility processes.

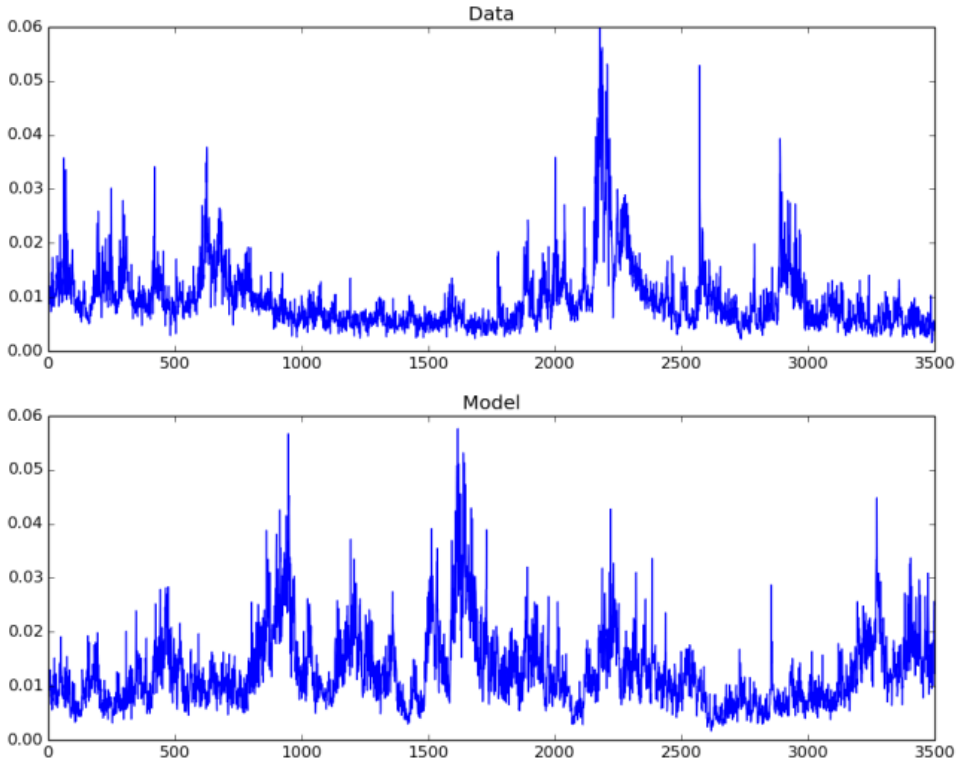


Figure 1.1: Comparing simulated rough fractional stochastic volatility model with SPX volatility)

1.3 Why use machine learning to model volatility?

Relying solely on stochastic models is just enough if our purpose is to only get the pricing map with no regard for computation time. However, industrial applications require models to be as fast as possible. The importance of even a microsecond was well portrayed by Michael Lewis in Flash Boys where a single firm with a monopoly on 1 microsecond over the market would generate profits of $20bn. In the institutional context speed is crucial regardless of any desirable mathematical properties the models may have. This applies particularly to rough volatility models (RVM).

In rough volatility models, the time constraints in calibrating is the main limiting factor. A fractional Brownian motion drives the instantaneous volatility [15]. This fractional driver has a short-memory which contradicts existing practices in econometric modelling but provides many modelling advantages(see [16][17][7][5][18] for example). Other models like the SABR model or the Heston model also requires requires time intensive numerical pricing schemes for accurate computation of no-arbitrage prices. The purpose of ML application is to replace the usage of such computationally less efficient methods including Monte Carlo [19][20], and finite element methods [21]. Our objective in this paper is to experiment with different ANN architectures to test if we can minimize the total time required for calibration can be further decreased.

Direct calibration using ANN has been approached by several researchers like [15][10][22][23]. The seminal paper by Hernandez [2] successfully used ANN to calibrate stochastic volatility models. The architecture was designed such that it returned the optimized parameters of the volatility model directly.

A major benefit ANN provides is the enhanced potential of learning the map directly from model parameters to IV surfaces. Another benefit is that ANNs are relatively much more immune to dimensionality issues compared to PDEs[4]. [3] did this by separating the approximating NN and the calibration in two different processes. This allows the generalizations to be more robust and also increases, by orders of magnitude, the calibrating speed compared to direct calibration as done by [2].

Furthermore, as [3] puts it, completely eliminating the use of numerical optimization schemes leaves the meaning of calibrated networks parameters unexplained and the network design and parameters remain ambiguous. This ambiguity can create conflicts with the regulatory requirements. This means that the model parameters remain as interpretable

as is the case with stochastic models, and that the knowledge accumulated in the literature so far is still valid and useful. So the ANNs are only used to enhance the existing methods. We also do not need to worry about interpreting the ANN parameters which makes the architecture development [add footnote: i.e. the number of hidden layers and nodes] process more flexible. The number of layers and nodes determine the balance between bias-variance tradeoff. I.e. optimizing the loss function such that the combined generalization errors arising from model bias and variance is minimized.

The next two sections of this project are organized as follows. Chapter 2 gives an overview and mathematical formalization of the methodology and motivation to separate the approximation step and calibration step. Chapter 3 provides a brief explanation of how ANNs work.

Chapter 2

Rough Volatility Models

The Heston Model

The Heston Model is a stochastic model that describes how the volatility of underlying asset evolves over time[24]. It assumes the volatility is dynamic and follows a random process.

$$\begin{aligned} dS_t &= rS_t dt + \sqrt{v_t} S_t \;\; for\; t > 0,\; S_{t_0} = S_0, \\ dv_t &= k(\hat{v} - v_t)dt + \gamma\sqrt{v_t} dW_t^v \;\; for\; t > 0,\; v_{t_0} = v_0, \\ dW_t^s dW_t^v &= \rho dt \end{aligned} \tag{2.1}$$

Z and W are Weiner processes and their correlation parameters is given by $\rho \in [-1, 1], a, b, v > 0$. One drawback of Black-scholes equation is that it assumes a constant volatility σ. This assumption can be relaxed by modelling the variance as a diffusion process [4]. This gives us stochastic volatility models. The rationale of modelling this variance as a random variable is supported by empirical data which reveals the uncertain nature of volatility of stock prices.

The most compelling reason to consider stochastic volatility is the volatility skew/smile which is seen in financial data and is precisely recovered by stochastic models. The volatility smile is most consistent with option contracts with a longer maturity date T. In equation 2.1, we can see the second stochastic process that is correlated to the stock price process S_t. Here we have a SDE scheme which is more computationally expensive for option pricing compared to just the scalar stock price process [4].

The Rough Bergomi Model

Several research have suggested that the family Rough volatility models like the rBergomi

have shown to describe Vanilla and exotic options markets with high precision.

$$\begin{aligned} dSt &= -\frac{1}{2}v_t dt + \sqrt{v_t}dW_t, \quad fort > 0, S_0 = 0 \\ v_t &= \xi_0(t)\epsilon(\sqrt{2H\nu}\int_0^t (t-s)^{H-1/2}dZ), \quad t > 0 \end{aligned} \tag{2.2}$$

The Hurst parameter is given by H. [6] showed that log-volatility acts like a FBM with Hurst parameter of 0.1 order, at any sensible timescale. The rBergomi has major disadvantage with its long computation time for calibrating to the market price of vanilla options. Not uncommon among most rough models, there is no analytical solution and so Monte Carlo method is most commonly used for pricing. Nevertheless, since rough volatility provides a better representation of historical, we expect that it will also enhance the predictive accuracy of future behaviour of volatility and returns. Particularly, it will provide a more precise estimation of IV surface. As we know that the IV derived from the BS equation is a major variable in determining option prices but is also the only variable in the equation that cannot be directly observed. So, rough volatility models can augment the precision of the BS formula.

Chapter 3

Machine Learning Perspective to Model Calibration

The objective of calibration is to make adjustments to a model so that it approximates the reality as closely as possible. In Q finance, the variables we represent are often the underlying assets or processes (stocks, volatility, etc) and our objective is to calibrate a model to the existing market prices of derivatives. In this paper we will use the same notations as used by [3]; defined as follows. $\mathcal{M} := \mathcal{M}(\theta)_{(\theta \in \Theta)}$ gives a representation of an abstract model with parameter vector θ in the set $\Theta \in R^n$, for some $n \in \mathbb{N}$. So the model $\mathcal{M}(\theta)$ and the appropriate prices of derivatives can be fully specified with a chosen parameter vector $\theta \in \Theta$. [3] introduced an additional element called the pricing map $P : \mathcal{M}(\theta, \zeta) \to \mathcal{R}^{\Updownarrow}$ where $\zeta : (C(\mathbb{R}) \to R^m, m \in \mathbb{N})$ represent the derivatives that are to be priced (e.g. plain vanilla options for a set of strike prices and maturities). The existing market prices of the derivatives are denoted by $\mathcal{P}^{mkt}(\zeta) \in \mathbb{R}^m, m \in \mathbb{N}$.

Calibrating parameters: We seek to find a parameter vector $\hat{\theta}$, such that the δ-calibration problem of model $\mathcal{M}(\Theta)$ is solved for the available market prices $(P)^{MKT}(\zeta)$. This can be achieved by minimizing the following:

$$\hat{\theta} = \underset{\theta \in \Theta}{\operatorname{argmin}}\, \delta(P(\mathcal{M}(\theta), \zeta), \mathcal{P}^{MKT}(\zeta)) \tag{3.1}$$

$\delta(.,.)$ is an appropriate metric characterized in the topological space of the derivative ζ. The problem is that an analytical equation for the derivative price $P(\mathcal{M}(\theta), \zeta), \mathcal{P}^{MKT}(\zeta)$ usually does not exist. So usually we must approximate this through some numerical method which is what we will do as well.

Calibrating parameters using numerically approximated pricing map $\hat{P}$

$$\hat{\theta} = \underset{\theta \in \Theta}{\operatorname{argmin}}\, \delta(\hat{P}(\mathcal{M}(\theta), \zeta), \mathcal{P}^{MKT}(\zeta)) \tag{3.2}$$

So naturally we will also be focusing on the model with approximated pricing map in ANN implementations. We will also use the same data used by [3] which was generated using the approximated $\hat{P}$ of true P to use as the training set for ANN for the purpose of generating approximate price maps. Obviously, the precision of ANN approximations will depend on the precision in the numerical approximations.

3.1 Neural Networks

Financial data are usually noisy as a result we must consider the systematic part and the random part during modelling. The reason why we use predictive models is so that the event signals that dictates the systematic component can be isolated since these components cannot be observed explicitly.

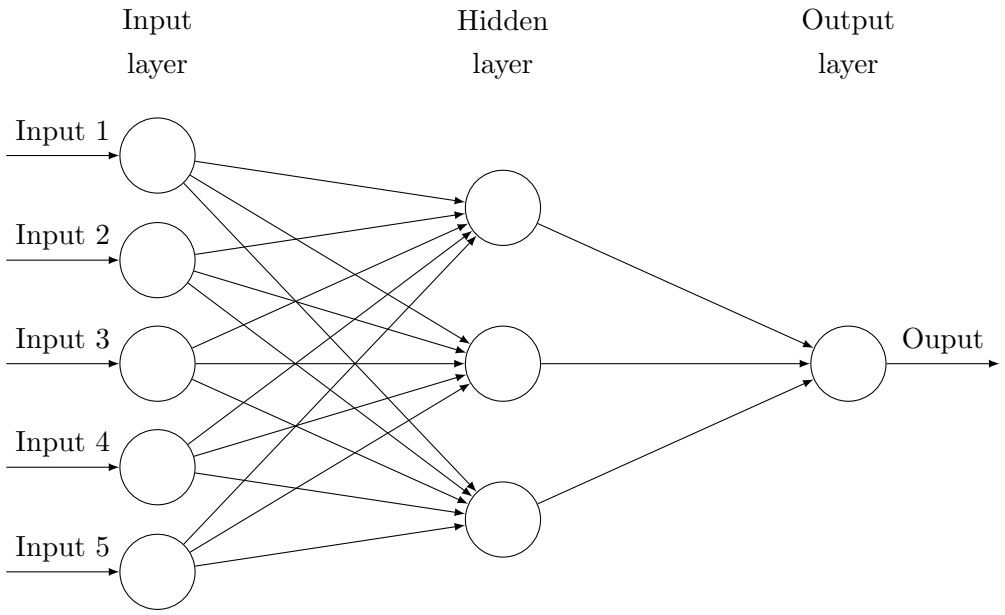
Artificial neural networks (ANN) are machine learning methods inspired by organic neurons in humans that learn to carry-out activities without the need for any direct programming with defined rules. ANNs is used in a plethora of problems in Q quantitative finance including risk-neutral probability [25], continuous-time martingales [25], calibration, derivative trajectory [26], and hedging options [27]. ANN's are popular due to their robust, adaptable, universal function approximation and parallel data processing characteristics, and fault tolerance attributes[28]. This enables us to solve complex non-linear and multi input-response problems [29] which is also a modelling feature in [3].

Defining optimal architecture can be ANNs can be a modelling obstacle; e.g. the large (or small) number of hidden units or complex (too simple) input structure increases the risk of overfitting (underfitting). While overfitting leads to a biased model and underfitting leads to a model with high variance, they are both subject to poor generalization in out-of-sample predictions. [30] asserts that an optimal between bias and variance is dependent on an appropriate difference between the number if ANN parameters relative to the training data size that achieves a sweet-spot with lowest possible combination of bias-variance errors.

Variance quantifies the variation between the model outputs while bias measures the de-

parture of the predictions from the true values. [31] showed that the dropout method in ANNs can aid in the optimization of bias-variance tradoff to prevent overfitting. The idea is that by eliminating units together with the nodes they're connected at random, during training, the co-adapting behaviour among each other is alleviated. Other obstacles in designing ANN architecture include deciding on the type of ANN, splitting the data into training, test and validation sets, selection of activation function (sigmoid, tanh, softmax,etc). Appropriate learning rate, optimizer, and momentum are also important [28].

The neurons take the input set $\{x_1, \ldots, x_n\}$ and are added according to the assigned weights $\{w_1, \ldots, w_n\}$ which are initialized at random together with the biases b at the beginning of training. We multiply the input set with their respective weights and add the respective bias term on every neuron they are connected to in the succeeding layer. The biases do not have any explicit interaction with the input sets but they affect the outputs since they are linked to each hidden neurons. Then we apply an activation (or transfer) function to activate the outputs at a pre-specified range. The new information is a weighted sum of the output of the activation functions which is passed on to the next neuron. This process keeps continuing until it reaches the final node. The flow diagram below.



ANN architectures usually have at least one hidden layer. L is used to indicate the number

of layers, while l represents a particular layer. In the figure above we can see an ANN with one hidden layer. This means there will be a total of three layers including the input and the output layers. There is no limit on how many neurons we can include in each layer (figure: five in input layer, three in hidden layer and one in th output layer).

The activation functions on the j^{th} neuron of the l^{th} layers are denoted by a_j^l. We represent the activation α using the following equation. It represents how the activation of layer l is functionally dependent on the activation of layer $l-1$

$$\alpha_j^l = \sigma(\sum_k w_{jk}^l \alpha_k^{l-1} + b_j^l) \tag{3.3}$$

$$\alpha_j^l = \sigma(z_j^l)$$

We can rewrite this expression in the matrix form as:

$$\alpha^l = \sigma(w^l \alpha^{l-1} + b^l) \tag{3.4}$$

where w^l is the weight matrix and b^l is the bias vector in the l^{th} layer. The entries of w^l are the elements of row j and column k in w_{jk}^l while the bias vector has only one entry per neuron in a given layer l. The a^l is a a vector of activation function that includes the activation of the j^{th} neuron in the l^{th} layer.

3.2 Backward propagation

In our calibration experiments we train our networks using the back-propagation algorithm. In back-propagation, the ANN learns the mapping from arbitrary inputs which in our case the pricing map takes stochastic model parameters (e.g. rBergomi parameters) and contract details as inputs, e.g. strike, maturity, payoff and outputs either an option price or implied volatility (depending on the application) corresponding to that contract. Every combination of stochastic model parameters can be provided as input to the algorithm. The best input set is found via calibration to option market prices or quoted implied volatilities for each strike and maturity but this part will be done in the optimization step using the

outputs from the ANN.

The computation of the gradients starts from the end of the network. This means the weights and biases of the final layer are computed first and the error signal is calculated and stored for each neuron.

Backpropagation calculates the partial derivatives $\partial C/\partial w$ and $\partial C/\partial b$ of the loss function C with respect to the weights and biases. The loss function is given by the following

$$C = \frac{1}{2N} \sum_{x} \|y - a^L(x)\|^2 \tag{3.5}$$

N = Number of training samples;
y = targets
L = Number of hidden layers
$a^L(x)$ = vector of activations functions

When a neuron receives an input, the weighted input of the neuron is changed by $\frac{\partial C}{\partial z_j^l}\Delta z_j^l$.

If $|\frac{\partial C}{\partial z_j^l}|$ is large, then in the next iteration, the Δz_j^l will have the opposite sign to $\frac{\partial C}{\partial z_j^l}$, to decrease the loss. When, $\frac{\partial C}{\partial z_j^l}$ is 0 or close to 0, and the loss cannot be lowered further. This is the point where parameters are optimized

The error j in the l^{th} layer δ_j^l are given by:

$$\delta_j^l \equiv \frac{\partial C}{\partial z_j^l}. \tag{3.6}$$

The size of δ represents the change the error has on the neuron. The error in the last layer is given by:

$$\delta_j^L \equiv \frac{\partial C}{\partial z_j^L}\sigma'(z_j^L). \tag{3.7}$$

The loss function shown above is the element-wise version of the error and can also be

written in the matrix form as

$$\delta^l = \nabla_a C \odot \sigma'(z^L) \tag{3.8}$$

The variable $\nabla_a C$ is the vector of partial derivatives $\partial C/\partial a_j^L$. Its size reflects how fast the loss changes with respect to the output activations. We use this delta rule to update the weights allocated to the neurons and store it as a copy of inputs to each neurons that are scaled by their respective deltas. The errors in all the following layers is given by:

$$\delta^L = ((w^{l+1})^T \delta^{l+1}) \odot \sigma'(z^l), \tag{3.9}$$

Looking at this equation, we can say that the error moves backwards through the network. By calculating the dot products $\odot\sigma'(z^l)$, we move the errors in reverse order with the activations in layer l. Neurons of the final layer have their δs calculated first so that the neurons in the preceding layers can use it iteratively.

We can compute the for every layer, the error δ^l by combining equations 3.8 and 3.9. First we use 3.8 to compute δ^L then 3.9 is used to calculate δ^{L-1}. Then we use 3.9 again to get δ^{L-2} and so this process is carried out for every layers in the network. Such partial calculations of the gradients from one layer are reused to compute the gradient on the previous layer which is a very efficient method of computing the gradients in each layer

The equations for gradient of error with respect to the weights and biases are given by 3.10 and 3.11 respectively.

$$\frac{\partial C}{\partial w_{jk}^l} = a_k^{l-1} \delta_j^l. \tag{3.10}$$

$$\frac{\partial C}{\partial b_j^l} = \delta_j^l. \tag{3.11}$$

The expressions 3.10 and 3.11 shows how we compute the partial derivatives to be in terms of δ^l and α_j^{l-1}. In the first iteration of computing the output, we are likely to get an error that is very high so we back-propagate the error again. With every update, the

magnitude of the change in weights is dependent on the size of the *learning rate*. A small learning rate will learn more exhaustively, however, it will require many more iterations to complete optimization which means higher computational cost. A high learning rate performs training relatively quickly but is more likely to skip important signals leading to a higher probability of snap judgements or sub-optimal generalizations. With the network itself there is a tradeoff between accuracy and speed.

3.3 Convolutional Neural Networks (CNN)

The rationale behind convolutional neural networks is based on local connectivity. Here the nodes that connect the neurons and layers are connected to a local area of the input space. These nodes/filter scans a local region in the input space known as the receptive field. This filter is also a matrix of weights, with which the inputs are convolved to create a feature map. The filter scans local regions of the inputs and computes the dot products of these inputs and weights over the receptive field. E.g. when a given input is an image, then it will go through several multiplications betweeen the pixel values and the filter.

For example, when the inputs are images, the multiplication are being computed between the filter and the pixel values. In our case of time series data, the filter slides on a subset of the covariates. This mean that several output nodes will detect the same patterns. This gives rise to the idea of *shared* weights which, in CNNs, reduces the learnable parameters. Consequently, increasing training efficiency.

CNNs have the ability of modelling data with more than one spatial dimension and often used in image classification. However, in time series data or the kind of data that we use for option price calibration, there is generally only one spatial dimension so since the filter will have a weight matrix of the same dimension as the input which is one although there is no reason why one cannot model in such a way that the input has several spatial dimension.

Unlike multi-layered perceptrons (MLPs), every value that is stored in the feature map have the same weights. This means that all the output nodes are detecting the same pattern. We confirmed this in one of our trial experiments too. The weights of the MLP layers that extract important features have over 2000 parameters while in CNN we could just use less than 500 parameters when using the same data. The intuition is that CNNs are translation invariant which makes it more efficient. Usually the spatial arrangement

in CNN comprises of input layer, convolution layer, pooling layer, and a fully connected layer. The convolution layers always come before the Pooling layers. The rationale is to is to slowly reduce the spatial size/parameters that require optimization which makes them computationally efficient compared to other types of NNs. Cutting the excess parameters effectively also controls the extent of overfitting. We also need to specify a variable called "stride". This is the size of a "filter" that scans the receptive field. Again, in image data, if stride = 2, then it means the filter moves through the image 2 pixels at a time. In option price calibration data where the inputs are the stochastic parameters, stride = 2 scans two of the those parameters at a time e.g. ξ and ν, and then moves on to ν and ρ and so on.

In our analysis, we will focus more on MLP as it provided better results given the inputs. Although we did not use CNN in our main experiments, CNN provides the flexibility of incorporating inputs of more that 1 dimension which could be a rewarding for further research on calibration e.g. by incorporating some of the fundamental analysis like strategies. Normally two and three dimensional inputs are used in image recognition where the width length and breadth are the spatial dimensions. However in our pricing map approximation, the inputs have only one spatial dimension: time. In our analysis, we will focus more on MLP as it provided better results given the inputs.

Chapter 4

Calibration with Neural Networks

In this chapter we will run experiments using the same data used by [3]. Our focus is on using different ANN architectures and specifications in step 1 but also to explore the suitability of different optimization methods in step 2 of the two-step approach.

There are several areas in [3] which does not sufficiently validate the extent of usability of the model. For instance, they did not provide any assessment on how often the pricing map generated by the network must be updated. I.e. how much does the quality of the calibration deteriorate if we do not retrain the pricing map over the sequences of the new data? To test this we would need market data from which the sample training set was drawn (or simulated). They used simulated data even for the test set which calibrated well but if we were to test the length of time after which the networks would require retraining, we would require actual market data over several period of time to understand the rate of decay. This also undermines the relevance of using further simulated because, the new simulated sample would still have the same theoretical properties as the original test set, and would not incorporate the dynamic signal of the market prices. In other words, the changes in sampling property of the market data over time will not be reflected in the new simulated data.

Second issue with [3] is that option prices must be a decreasing function and convex w.r.t to strike prices and monotonic w.r.t. to maturities. This property is not preserved in their network. Therefore, one would not expect the pricing map to be robust.

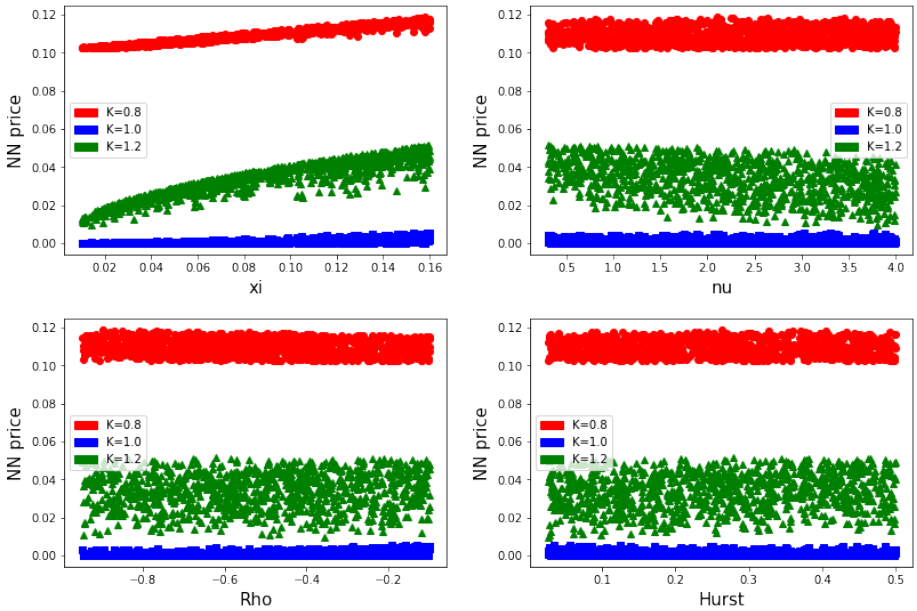


Figure 4.1: Option price against three different strike prices at a constant maturity at 0.1 for various ξ, ν, rho and Hurst exponent values. Visualization to inspect the presence of convexity

The data should also preserve the following:

Suppose the option price is given by $H(K)$ where K is the strike price. If the option price is convex to K, then it must mean that

$$H(K+\gamma) + H(K-\gamma) > 2H(K) \tag{4.1}$$

for all K and γ ¿ 0. Suppose that the reverse was true; meaning there were option contracts in the market with the same expiry date such that:

$$H(K+\gamma) + H(K-\gamma) \leq 2H(K) \tag{4.2}$$

If someone was to purchase a contract for $K+\gamma$ and another contract for $K-\gamma$, and fund these two purchases by selling two options at K which can be done because $2K$ costs at

least as much as the two other combined.

At maturity the stock price will be S_t and the total payoff will be:

$$Payoff = (S_t - (K - \gamma))^+ + (S_t - (K + \gamma))^+ - 2(S_t - K)^+ \tag{4.3}$$

There are four possibilities:
$S_t < K - \gamma \longrightarrow Payoff = 0$
$K - \gamma < S_t < K \longrightarrow Payoff = S_t - (K - \delta) > 0$
$K < S_t < K + \gamma \longrightarrow Payoff = S_t - K + \gamma - 2(S_t - K) = K + \gamma - S_t > 0$
$S_t > K + \gamma \longrightarrow Payoff = S_t - K + \gamma + S_t - K - \gamma - 2(S_t - K) = 0$

There is no indication that the simulated data satisfies this condition.

4.1 Calibration on real data

We train the neural network with the data used by [3] and the architecture with a few amendments as follows.

- Scaled parameters as used as input and scaled IVs as output
- 3 hidden layers with 40 neurons and Elu activation function
- Output layer with Linear activation function
- Total number of parameters: 7088
- Training Set: 68,000 and Test Set: 300 (Approximately 26,000 entries divided among 11*8 columns)
- rBergomi sample: $(\xi \in [0.005, 0.2], \nu \in [0.3, 5], \rho \in [-0.1, -0.05], H \in [0.015, 0.6])$

The range of parameters have been expanded slightly in order to have more manoeuvring space for the trained parameters to align with the real market data. This is also the third limitation of their approach in that if there are outlier prices, or less liquid prices, one should weight the loss function accordingly. This could be easily addressed by weighting

the loss function in tensor flow with the $1/(ask - bid\ price)$. But the issue is that the information regarding bid-ask spread is not incorporated in the model

4.1.1 Data scaling

The model parameters were scaled as shown below but also their values are restricted within the range of $[-1, 1]$:

$$\frac{2\theta - (\theta_{max} + \theta_{min})}{\theta_{max} - \theta_{min}} \in [-1, 1] \tag{4.4}$$

4.1.2 IV scaling

Scaling IV is more sensitive as it can take only non-negative values as we know that $\sigma_{BS}(T, k, \theta) \in [0, \infty) \ \forall \ T$ and k. The IV is scaled as follows:

$$\frac{\sigma_{BS}(T, k, \theta) - \hat{\sigma}_{BS}^{train}(T, k)}{std(\sigma_{BS}(T, k, \theta^{train}))} \tag{4.5}$$

4.1.3 NN optimizer

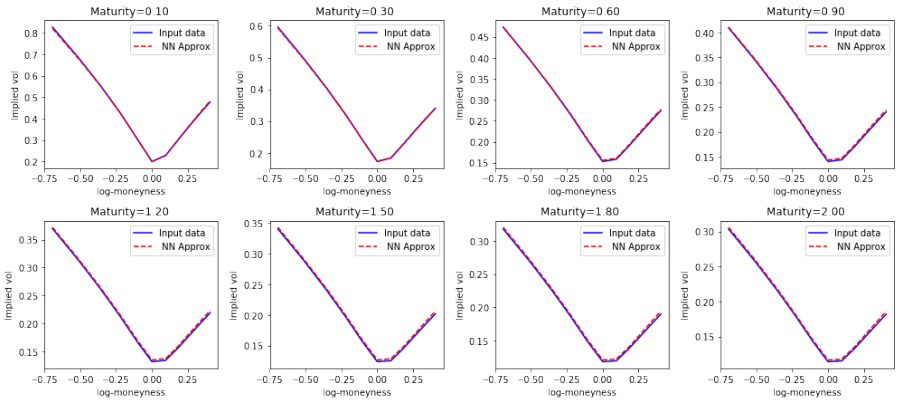
We tested the RMSProp (root mean squared propagation) optimization to update the network's weights and fine-tune the parameters in order to minimize the cost function. The choice of this optimizer is based on the fact that it is flexible with mini batch learning of weights. This method may not provide significant advantage over the ADAM optimizer used in [3] since they used simulated data which we expect to be smoother compared what we would observe empirically. However, when we use real data we expect more noise and different types of irregularities which complicates the model fitting and consequently the calibration.

RMSprop provides a quirky way of dealing with the tendency of the gradients to vary extensively in their magnitudes. This makes it difficult to stick with a constant learning rate. If we implement full batch learning, issues arising from such large variations in gradients are mitigated by simply using the sign of the gradient although it will make every

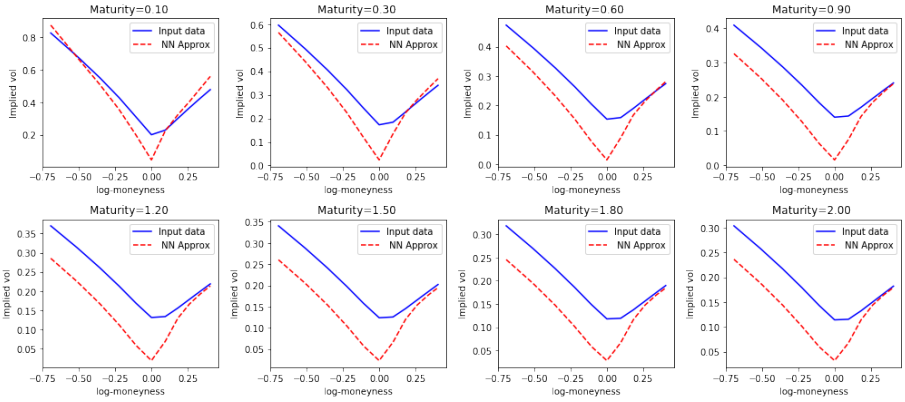
single weight updates of the same size. We can make the model more robust to avoiding local minimas by using smaller gradients, and this is exactly the purpose of RMSProp. It can take large steps even with very small gradients. While we cannot achieve this by just altering the learning rate since increasing the learning rate would mean that the steps steps taken for weights with big gradients will be excessively large. RMSProp merges the idea of using signs of the gradients and using dynamic step sizes which changes according to the sign of weights. To determine how much the weights should be changed, it does not look at the magnitude of the gradient rather just the sign of the gradient and the step-size that has been assigned for that weight. This step-size is dynamic which adapts through time. The step-size of a particular weight increases by some multiplicative factor if the signs of the last two gradients are identical, but if the signs of the last two gradients are not identical, the steps will be decreased. Every mini batch will be divided by some number that is close to the nearby mini-batches. We do this by using the moving-average of squared gradient. The gradient will be updated in such a way that the change is proportional to the previous gradient divided by the root of mean squared error (see 4.x)[32].

$$MSE(w,t) = \alpha MSE(w,t-1) + (1-\alpha) * (\frac{\Delta E}{\Delta w^t})^2 \tag{4.6}$$

In figure 4.2 we can see the IV surface generated using NN approximation w.r.t the appropriate maturities and strikes (8x11). Using the simulated data provided by the authors, the network was able to retain the near perfect fit on the validation set even when the parameter were decreased from 5668 to 3808 but the optimizer was changed from ADAM to RMSProp (4.1a). However, the out-of-sample performance is much more misaligned (4.1b). Although it is important to stress again that the real data from SPX500 only lightly fits the parameter specification of the simulated data, hence some misalignment was expected either way.



(a) Simulated data



(b) Real data

Figure 4.2: Comparison of simulated data performance vs out-of-sample performance with real data

4.1.4 Forward variances

In the figure below, we can see that the calibration errors are quite high. In simulated data [3] had achieved the fastest calibration with Levengerg-Marquardt. In our case, the fastest calibration was achieved by L-BFGS-B. The figures show relative calibration error in rBergomi model. The error seems to show different magnitude but similar pattern on the distribution of errors among all the four parameters between real and simulated data. We can see that the real data has much higher error. For ξ, ν, and Hurst exponent larger

errors are concentrated on the smaller value of these parameters while its the opposite or ρ which shows larger error at higher ρ values. Knowing these error prone areas provides further analytics on uncertainty quantification of option prices.

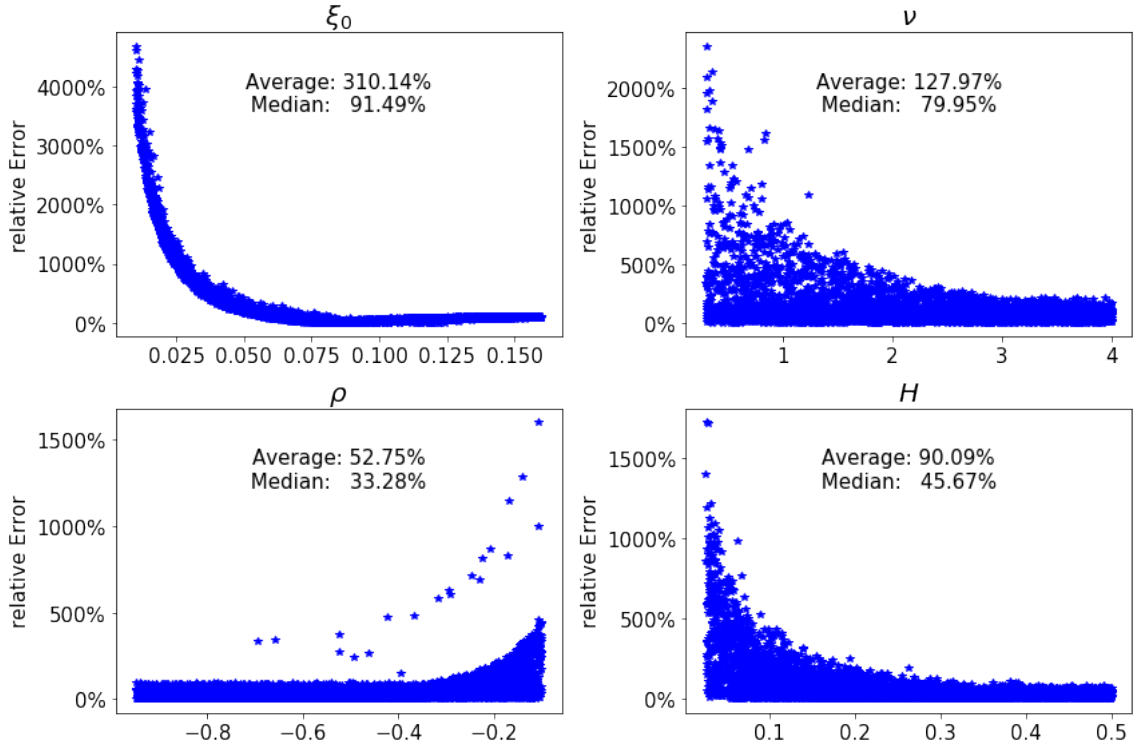


Figure 4.3: Calibration error of each parameter of the real test set on the rBergomi model

4.1.5 Calibration speed

We experimented several combinations of NN architectures, optimization methods, input structures and gradient methods to test how they compare to the results of [3]. None of the combinations were able to match the speed/accuracy combination presented in their paper. The quality of calibration that came close to came at a cost of significantly larger computation time. The plot below show the fastest our experiments could achieve using the architecture specification mentioned above.

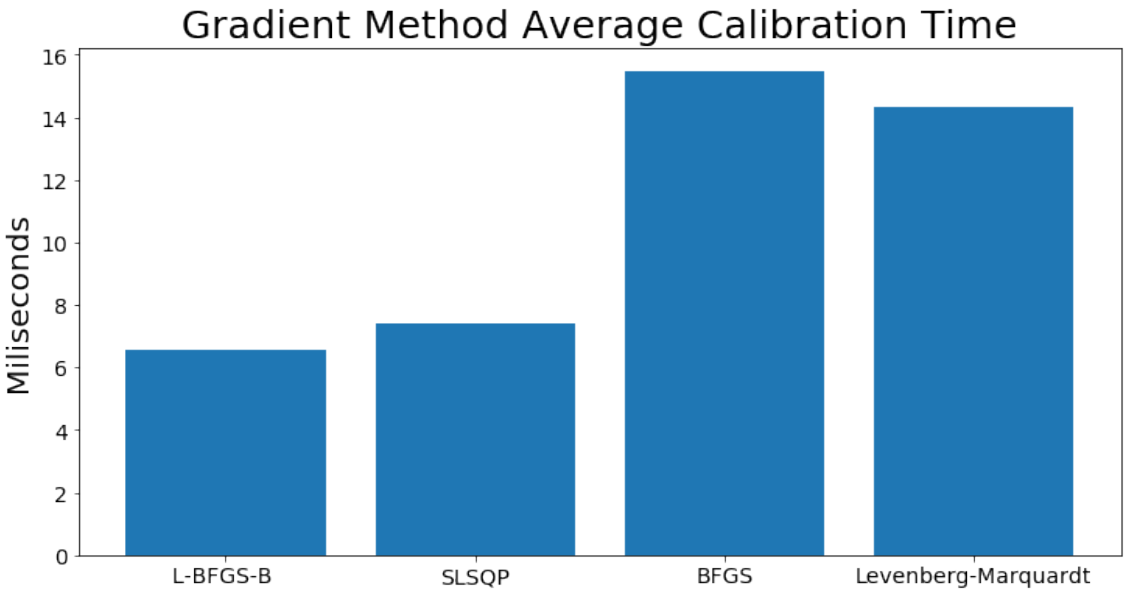


Figure 4.4: Time taken by the gradient methods for calibration

4.2 Conclusion and further work

The objective of this paper was to implement and criticize the methodology of option price and volatility calibration with neural networks used by [3]. NNs show great promise in speeding up the process of calibration. The ubiquitous availability of computational power and ease of applying ML techniques provide an enhancement in financial applications but yet have not effectively been able to completely substitute the stochastic methods. [3] have effectively mixed the usage of both NN and classical optimization to calibrate option prices at similar quality but at faster speed. However the limitation is that it depends on large amount of training data. Often, this may not be an issue but the validity of its usage in real market data is still not clear.

The literature on the application of NN on option price/volatility has been encouraging. Historical estimates are often poor indicators of present/future volatility and industry practitioners are compelled to depend on Back-Scholes or similar formulae exclusively where the IV can be calculated implicitly for only the present volatility [33]. Such formulae are only useful for real time prediction to the day-traders and fail to integrate the information of the historical volatility. The NN based models can make use of both short-run historical

data and several contemporary variables to predict future IV. This enables the market practitioners take a position in the market with an strategic edge. E.g. large IV usually signifies possible consolidation of the market, while smaller IV usually signifies a possible breakout in the market.

Further work could focus on applying other ML methods for option price calibration with the two-step approach. E.g. Gaussian processes could be a good candidate since they are capable of robust performances even when datasets are much smaller. It has a computational complexity of $\mathcal{O}(\mathcal{N}^{\ni})$ but several sparse version have been suggested which are capable of achieving full Gaussian process performance at much less computational cost [34][35][36] [37]. This would provide us with a further enhancement since GP is a probabilistic method, on top of an approximation, it also gives a predictive distribution. In the NN approach, the focus was on how well the pricing map is approximated. We look at the predicted mean values that are the point predictions. Surely, such point predictions are valuable per se but in Gaussian processes we have the added option of making use of the probabilistic distribution which we do not have in NNs. This gives us further idea about how confident we are regarding the predictions.

References

[1] P. Carr, "Local variance gamma option pricing model," 2009.

[2] A. Hernandez, "Model calibration with neural networks," *Risk*, 2017.

[3] B. Horvath, A. Muguruza, and M. Tomas, "Deep learning volatility: A deep neural network perspective on pricing and calibration in (rough) volatility models," 2019.

[4] S. Liu, C. W. Oosterlee, and S. M. Bohte, "Pricing options and computing implied volatilities using neural networks," *arXiv e-prints*, p. arXiv:1901.08943, Jan 2019.

[5] C. Bayer, P. Friz, and J. Gatheral, "Pricing under rough volatility," *Quantitative Finance*, vol. 16, pp. 1–18, 11 2015.

[6] O. El Euch and M. Rosenbaum, "Perfect hedging in rough heston models," *Annals of Applied Probability*, vol. 28, 03 2017.

[7] J. Gatheral, T. Jaisson, and M. Rosenbaum, "Volatility is rough," *SSRN Electronic Journal*, 10 2014.

[8] J. Gatheral, "A parsimonious arbitrage-free implied volatility parameterization with application to the valuation of volatility derivatives.," 2004.

[9] J. Gatheral and A. Jacquier, "Arbitrage-free svi volatility surfaces," *Quantitative Finance*, vol. 14, 03 2013.

[10] C. Martini and S. Hendriks, "The extended ssvi volatility surface," *SSRN Electronic Journal*, 05 2017.

[11] E. Euch Omar, F. Masaaki, and R. Mathieu, "The microstructural foundations of leverage effect and rough volatility," *arXiv e-prints*, p. arXiv:1609.05177, Sep 2016.

[12] E. Abi Jaber and O. El Euch, "Multi-factor approximation of rough volatility models," *arXiv e-prints*, p. arXiv:1801.10359, Jan 2018.

[13] O. E. Euch, M. Fukasawa, J. Gatheral, and M. Rosenbaum, "Short-term at-the-money asymptotics under stochastic volatility models," *arXiv e-prints*, p. arXiv:1801.08675, Jan 2018.

[14] O. El Euch and M. Rosenbaum, "Perfect hedging in rough Heston models," *arXiv e-prints*, p. arXiv:1703.05049, Mar 2017.

[15] C. Bayer and B. Stemper, "Deep calibration of rough stochastic volatility models," 2018.

[16] E. Alós, J. A. Leon, and J. Vives, "On the short-time behavior of the implied volatility for jump-diffusion models with stochastic volatility," *Finance and Stochastics*, vol. 11, pp. 571–589, 02 2007.

[17] M. Fukasawa, "Asymptotic analysis for stochastic volatility: Martingale expansion," *Finance and Stochastics*, vol. 15, pp. 635–654, 12 2010.

[18] C. Bayer, P. K. Friz, A. Gulisashvili, B. Horvath, and B. Stemper, "Short-time near-the-money skew in rough fractional volatility models," *Quantitative Finance*, 03 2017.

[19] L. B. G. Andersen, "Efficient simulation of the heston stochastic volatility model," *J. Computat. Finance*, vol. 11, 01 2007.

[20] B. Chen, C. W Oosterlee, and H. van der Weide, "Efficient unbiased simulation scheme for the sabr stochastic volatility model," vol. 15, 02 2011.

[21] B. Horvath and O. Reichmann, "Dirichlet forms and finite element methods for the sabr model," *SIAM Journal on Financial Mathematics*, vol. 9, 01 2018.

[22] H. Ghaziri, S. Elfakhani, and J. Assi, "Neural networks approach to pricing options," *Neural Network World*, vol. 10, pp. 271–277, 01 2000.

[23] M. Malliaris and L. Salchenberger, "A neural network model for estimating option prices," *Appl. Intell.*, vol. 3, pp. 193–206, 09 1993.

[24] S. Heston, "A closed-form solution for options with stochastic volatility with applications to bond and currency options," *Review of Financial Studies*, vol. 6, pp. 327–43, 02 1993.

[25] S. Becker, P. Cheridito, and A. Jentzen, "Deep optimal stopping," 04 2018.

[26] N. Chapados and Y. Bengio, "Forecasting and trading commodity contract spreads with gaussian processes," in *Departement d'informatique et de recherche operationnelle Universite de Montreal*, 2007.

[27] F. Chen and C. Sutcliffe, "Pricing and hedging short sterling options using neural networks," *Int. J. Intell. Syst. Account. Financ. Manage.*, vol. 19, pp. 128–149, Apr. 2012.

[28] B. W. Wanjawa and L. Muchemi, "ANN Model to Predict Stock Prices at Stock Exchange Markets," *ArXiv e-prints*, Dec. 2015.

[29] J. M. Ortiz-Rodriguez, M. del Rosario Martinez-Blanco, J. Manuel Cervantes Viramontes, and H. Vega-Carrillo, "Robust design of artificial neural networks methodology in neutron spectrometry," 01 2013.

[30] A. Munasinghe and D. Vlajic, "Stock market prediction using artificial neural networks : A quantitative study on time delays," 2015.

[31] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, "Dropout: A simple way to prevent neural networks from overfitting," vol. 15, pp. 1929–1958, 06 2014.

[32] H. Geoffrey, Rmsprop: Divide the gradient by a running average of its recent magnitude," 2013.

[33] M. Malliaris and L. Salchenberger, "Using neural networks to forecast the s&p 100 implied volatility," *Neurocomputing*, vol. 10, pp. 183–195, 03 1996.

[34] E. Snelson and Z. Ghahramani, "Sparse gaussian processes using pseudo-inputs," in *Advances in Neural Information Processing Systems 18*, pp. 1257–1264, MIT press, 2006.

[35] M. Seeger, C. K. I. Williams, and N. D. Lawrence, "Fast forward selection to speed up sparse gaussian process regression," in *IN WORKSHOP ON AI AND STATISTICS 9*, 2003.

[36] M. K. Titsias, "Variational learning of inducing variables in sparse gaussian processes," in *In Artificial Intelligence and Statistics 12*, pp. 567–574, 2009.

[37] M. van der Wilk, C. E. Rasmussen, and J. Hensman, "Convolutional Gaussian Processes," *ArXiv e-prints*, Sept. 2017.