UNIVERSITEIT•STELLENBOSCH•UNIVERSITY
jou kennisvennoot • your knowledge partner

A Universal Calibration Scheme for Stochastic Processes using Artificial Neural Networks

by

Rayno Willem Mostert

Thesis presented in partial fulfilment of the requirements for the degree of BCommHons(Actuarial Science) in the Faculty of Economic and Management Sciences at Stellenbosch University

Study leaders: Mr. Stuart Reid
Mr. Stephen Burgess

2017

Declaration

By submitting this report electronically, I declare that the entirety of the work contained therein is my own, original work, that I am the sole author thereof (save to the extent explicitly otherwise stated), that reproduction and publication thereof by Stellenbosch University will not infringe any third party rights and that I have not previously in its entirety or in part submitted it for obtaining any qualification.

Date: July 2017 .....

Abstract

Stochastic processes are powerful tools which can be applied to almost any time series-related problem. They offer clear representations of real-world processes in ways in which deterministic formulae cannot. Yet, the underlying mathematics complicate their application. One aspect of particular concern is the calibration of these processes to real-world data sets. Common calibration methods, including maximum likelihood estimation (MLE) and the generalised method of moments estimation (MME), can break down under more complex stochastic processes. MLE, for example, requires the derivation of the likelihood function, which often requires a deep understanding of stochastic calculus.

This dissertation proposes the use of artificial neural networks (ANNs) as an alternative calibration method. The results obtained in the simulation study demonstrate the ability of ANNs to act as a calibration scheme for a complex stochastic process - the Merton Jump-Diffusion process. The ultimate hope is that ANNs might offer a universal calibration scheme for any stochastic process.

The project starts by introducing stochastic processes and ANNs, followed by an analysis of the simulation study performed on the Merton Jump-Diffusion process. The performance of the ANN models are then compared to that of traditional calibration procedures.

Contents

List of Figures

List of Tables

List of abbreviations and/or acronyms

ANN artificial neural network. vi, 1–8, 11, 15, 18, 19, 24, 26–29, 44

AWS Amazon Web Services. 27

CNN convolutional neural network. vi, 9, 10, 13, 16, 17, 22, 24–29, 45–49

EC2 Elastic Compute Cloud. 27

ELU exponential linear unit. vi, 8, 9, 15–18, 24–26, 45–49

EM expectation maximisation. 6, 29

GBM geometric Brownian motion. 3, 13, 14, 16, 18

GPU graphics processing unit. 25, 27

MLE maximum likelihood estimation. 1, 3, 6, 24–27, 29, 44–50

MLE maximum likelihood estimate. 27

MME generalised method of moments estimation. 1, 3, 6, 24, 27, 29, 44–50

ReLU rectified linear unit. 8, 9, 26, 48, 49

RNN recurrent neural network. 13, 28, 29

SDE stochastic differential equation. 1–3, 5–7, 29

CHAPTER 1
INTRODUCTION

1.1. INTRODUCTION

> Stochastic processes are becoming more important to actuaries: they underlie much of modern finance, mortality analysis and general insurance. They are immensely useful because they form the common language of workers in many areas that overlap in actuarial science. It is precisely because most financial and insurance risks involve events unfolding as time passes that models based on processes turn out to be most natural.
> — Submission to the Faculty of Actuaries students' society in 1998 (Cairns, Dickson, Macdonald, Waters *&* Willder; 1998)

Stochastic processes are simply a collection of random variables, usually indexed by time (Barone-Adesi, 2015). They are typically used during the modelling process, in order to describe the evolution of an underlying real-world process. To allow for it to be used within the modelling context, a stochastic process is often expressed by its stochastic differential equation (SDE). Using the SDE, the simulated stochastic process can then be adjusted to best represent the real-world process at hand - whether that be the evolution of the price of a certain stock, the claims on an insurance policy or the mortality rate of a group of policyholders. Oreskes, Shrader-Frechette *&* Belitz (1994) refer to this procedure of "[manipulating] the independent variables to obtain a match between the observed and simulated distribution or distributions of a dependent variable or variables", as model calibration.

Effectively modelling a real-world process involves two major challenges. Firstly, choosing an appropriate stochastic process with properties that mimic those of the real-world process; and, secondly, finding the most suitable parameters for the relevant SDE. In practice, however, the selection of an appropriate stochastic process is often influenced by the ease with which its parameters can be calibrated. Thus complex stochastic processes, with more complex SDEs, are often substituted for simpler, easily calibrated models. This can lead to the use of models that are subject to simplifying assumptions or possess properties that may not be the best possible representation of reality.

The difficulty associated with model calibration depends on the method of calibration applied. Mongwe (2015) and Honore (1998) describe how common calibration methods, including maximum likelihood estimation (MLE) and the generalised method of moments estimation (MME), can break down under more complex SDEs. MLE, for example, requires the derivation of the likelihood function, which is often difficult in the case of complex SDEs as they can yield unbounded likelihood functions.

Another calibration method that has seen a renaissance in the last decade is that of backpropagation, which has proven to be a powerful gradient descent-based algorithm for calibrating artificial neural networks (ANNs). This paper will explore the ways in which these statistical learning tech-

niques - namely ANNs, calibrated by backpropagation - could, in turn, be applied to the calibration of stochastic processes.

1.2. PROBLEM STATEMENT

Every stochastic process contains a set of parameters, Z, which controls the dynamics of the paths produced by the model. The calibration problem can be framed as a mapping from the observed data, D, or some transformation thereof, to these parameter values, Z. Let C denote the calibration method, then C is necessarily of the form,

$$C : f(D) \rightarrow Z \tag{1.1}$$

An ANN is a collection of interconnected processing units (Teugels *&* Sundt, 2004), which realises a nonlinear mapping from inputs, R^X to outputs, R^Y.

$$ANN : R^X \rightarrow R^Y \tag{1.2}$$

This mapping is achieved by chaining a sequence of nonlinear multiple regression functions, f, (called activation functions) together in layers (see 1.5.1 below).

From equations (1.1) and (1.2), we can see that an ANN has the ability to estimate the calibration function, C. This assertion is justified by the Universal Approximation Theorem, which states that "standard multilayer feedforward networks are capable of approximating any measurable function to any desired degree of accuracy" (Hornik, Stinchcombe *&* White,1989).

The question, however, remains as to what such a network might look like and how it would compare to traditional calibration techniques.

1.3. RESEARCH OBJECTIVES

This paper has the primary objective of researching and testing the viability of ANNs as a universal method of parameter estimation for any stochastic process. As to be seen in the literature review, ANNs have been used extensively in the world of financial modelling, and to some extent in the calibration of simple stochastic processes.

Conceivably, ANNs have the potential to act as a universal calibration method for any stochastic process. This paper aims to investigate whether and how that might function in practice.

This study will consist of two phases. In the first, an ANN will be implemented for a sufficiently complex SDE, for which a likelihood function does exist (for comparative purposes). In the second phase, the accuracy of the network's approximation of the calibration function will be measured and compared against that of other calibration techniques.

1.4. IMPORTANCE OF THE STUDY

An ANN-based approach to parameter estimation could likely provide numerous benefits above the popular MLE and MME approaches. The technique could potentially provide a universal solution to approximate the calibration function, C, for any arbitrarily complex SDE. It does not require the derivation of the likelihood function, which can be difficult. The model drops some of the strong assumptions made by MLE and MME.

Another possible advantage could lie in the scheme's ability to combine the predictive power of ANNs with the descriptive properties of certain stochastic processes. Olden *&* Jackson (2002) explain that "although in many studies ANNs have been shown to exhibit superior predictive power compared to traditional approaches, they have also been labelled a 'black box' because they provide little explanatory insight into the relative influence of the independent variables in the prediction process". The approach, whereby the ANN is used only to calibrate a more expressive and widely understood model - a stochastic process - might help to remedy this.

Mongwe (2015) presents the example of the Merton Jump-Diffusion process with SDE,

$$d \ln S_t = \left(\mu - \frac{1}{2}\sigma^2\right) dt + \sigma dB + d\left(\sum_{i=1}^{N_t} Y_i\right) \tag{1.3}$$

where μ is referred to as the drift coefficient and σ as the diffusion coefficient. $B_t, t \geq 0$ is a standard geometric Brownian motion (GBM) process. Y_i represents the random size of the ith jump, and has distribution, $Y_i \sim N\left(\mu_{jump}, \sigma^2_{jump}\right)$. $N_t, t \geq 0$ is a Poisson process with intensity λ.

Note that these parameters provide insight into the characteristics of the stochastic process being modelled. Hence, it could arguably be more enlightening to fit the observed data to an expressive stochastic process which yields explanatory parameters, than simply using a "non-parametric" ANN to model the real-world process entirely. This helps avoid the black-box pitfall commonly associated with ANNs, while still making use of their "superior predictive power". This is of importance, as prudent financial management involves using modelling techniques that are well understood, clearly defined and well documented. ANNs - despite their powerful properties - are often not an acceptable means of modelling in actuarial applications. By using them simply to calibrate complex stochastic processes (which are a suitable and widely acceptable modelling tool), the strengths of ANNs are retained, without the risk associated with a black-box technique.

1.5. RESEARCH DESIGN AND METHODOLOGY

The research design and methodology will give an overview of the model construction process. Firstly, a more thorough description of one of the core components of the scheme - an ANN - is presented. This is followed by an explanation as to how an ANN architecture could be applied to estimating the model calibration function. Lastly, an overview of how the model is to be evaluated follows.

1.5.1 Neural Networks

An Artificial Neural Network is a mathematical model consisting of an interconnected collection of processing units, which realises a nonlinear mapping from inputs R^X to outputs R^Y,

$$ANN : R^X \rightarrow R^Y \tag{1.4}$$

This mapping is achieved by chaining a sequence of nonlinear multiple regression functions, f, (called activation functions) together in layers. Each input into every activation function is weighted by some value w.

The most common ANN architecture, a multilayer Perceptron, is illustrated in figure 1.1.

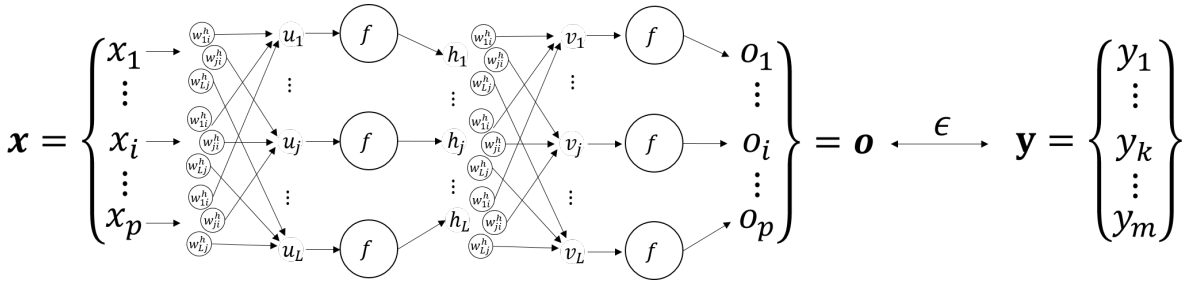


Figure 1.1: Multilayer Perceptron

For a given input, x, and (expected) output, y, the error of the ANN, ϵ, is equal to the distance between the ANN's outputs, o, and the expected outputs. The power of ANNs lies in the fact that they can be trained to minimize this error.

Training the network involves firstly initialising the network with a random set of weights, W. A large set of training data (sets of inputs, X, and desired outputs, Y) is then presented to the network. The optimisation process (often referred to as backpropagation) proceeds by calculating the prediction error, ϵ, for each of these data sets, and then "propagating" this error value backwards through the network so that each weight can be adjusted accordingly. This is achieved by means of automatic differentiation. Automatic differentiation allows us to compute the partial derivative of the error with respect to the weights in the ANN, W (Werbos, 1990).

This process is repeated iteratively over the training data, until the error of the ANN converges or some other stopping criteria is satisfied. At this point, the ANN will have approximated the relation, $\mathbf{X} \rightarrow \mathbf{Y}$.

A number of technical details - such as what inputs are fed into the ANN, what activation function is used, the number of activation functions used, the number of layers used, and the exact function used to estimate the error - have been omitted from this discussion for the sake of brevity.

1.5.2 Approximating the Calibration Function, C

From the previous section, it follows that an ANN should - theoretically - be able to approximate the calibration function defined earlier, $C : f(D) \rightarrow Z$. That is, $NN \approx C$. This can be achieved by:

i) generating a set of random calibrations, $\mathbf{Z}$

ii) simulating a set of paths, $\mathbf{Z}$, using the given SDE for every $Z_i \in \mathbf{Z}$

iii) extracting a set of "inputs", $\mathbf{X}$, from the data $\mathbf{X} = f(\mathbf{D})$

iv) training the ANN to predict the original calibrations, $\mathbf{Z}$, given the data $\mathbf{X} = f(\mathbf{D})$ as input. This is done by setting the ANN to minimise the error ϵ of its output, $\mathbf{O}$ (which corresponds to an estimate of Z_i) and the true known values of Z_i.

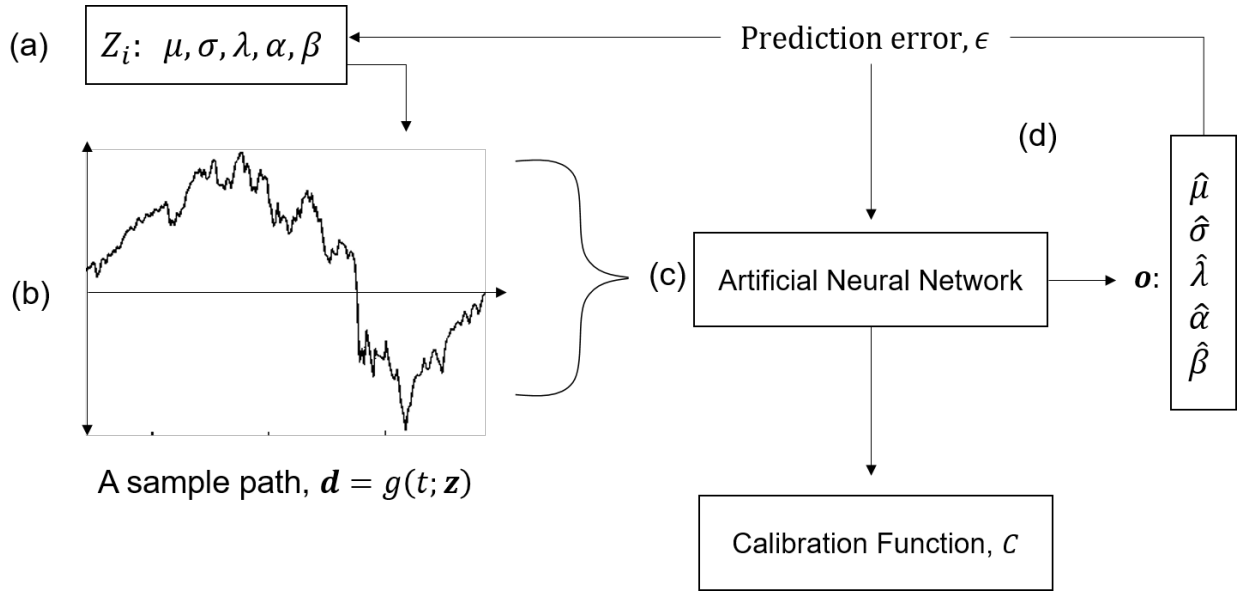


Figure 1.2: The Proposed Calibration Scheme

If a large enough set of calibrations is chosen and enough data is simulated from the SDE using each calibration, the ANN will approximate the desired calibration function, C, for the given SDE.

This trained ANN can then be applied to real-world observations to calibrate the relevant SDE for the observed process. This is done by inputting the data observed from the real-word stochastic process into the trained ANN. The ANN output, $\mathbf{o}'$, will then be an estimate of the parameters (i.e. a calibration) for the SDE that the ANN was trained on.

In this paper, the ANN was built and trained in Python, using Google's open source Tensorflow™ (Abadi *et al.*, 2015) and Keras (Chollet *et al.*, 2015) libraries. All model training was done using the Adam optimizer (Kingma *&* Ba, 2014).

A full list of software and packages used in this dissertation is available in appendix D.

CHAPTER 2
LITERATURE REVIEW

The literature discussed throughout this research will fall into mainly two broad categories. Firstly, a review of the common methods proposed for the calibration of stochastic processes, focusing on their limitations, ease of use, universality and accuracy. Secondly, an overview of past applications of ANNs to modelling and model calibration.

2.1. CALIBRATION OF STOCHASTIC PROCESSES

The main argument for a universal neural network approach to model calibration is that traditional methods are often unpractical. Numerous academic works substantiate this observation.

Nielsen, Madsen *&* Young (2000) reviewed the progress made on SDE parameter estimation over the 80s and 90s. They note that the MLE approach does not generalise and, having studied the MME and the efficient method of moments, they explain that both of these methods will result in tests of low power due to the efficiency loss.

More recently, Mongwe (2015) did a study on jump-diffusion processes applied to the South African equity and interest rate markets. He reported on the application and accuracy of multiple calibration methods, including the likelihood profiling approach (Honore, 1998), the standard MLE approach, the MLE approach and expectation maximisation (EM). He concluded that both MLE and MME fell short of expectations, and that the likelihood profiling and EM techniques worked best on parameter estimation for jump-diffusion processes (each under different restrictions on the parameters) (Mongwe, 2015). These recommendations will be applied when evaluating and comparing the performance of the neural-network calibration approach in this paper to that of existing methods.

2.2. NEURAL NETWORKS IN MODELLING AND MODEL CALIBRATION

Except for the case of Xie, Kulasiri, Samarasinghe *&* Rajanayaka (2007), the task of calibrating stochastic processes using ANNs has not been thoroughly attempted or documented in the major academic journals examined for this research. Numerous works do however highlight the potential of ANNs in this field. Multiple studies have been done on the use of ANNs in pricing options, and - in particular - to outperform the Black-Scholes model (Yao, Li *&* Tan; 2000). The results seem to indicate that ANNs outperform the Black-Scholes model in volatile markets, and are particularly useful in these when the "constant σ" assumption underlying the Black-Scholes model is violated.

Tackling the issue of model calibration; Samad *&* Mathur (1992) investigated the application of ANNs to the calibration of process systems - namely that of first-order process open-loop delay identification. They concluded that ANNs are an attractive solution, as they do not require the subject-specific expertise vital to common engineering approaches, provide high accuracy and prove robust on real-world data.

On the topic of this paper, Xie *et al.* (2007) did an investigation into the feasibility of estimating the parameters both linear and nonlinear SDEs using multilayer perceptron (MLP) networks. Their investigation comprised only of small MLPs with 1-, 2- and 3-hidden-layer, fully connected architectures. They found that, under certain conditions limiting the parameter values of the process, a simple MLP would be able to estimate parameters with high accuracy ($R^2 > 0.93$). They report that this accuracy figure, however, does diminish under noisy conditions and SDEs with high diffusion levels. Another, often overlooked element noted by their research is the importance of the regime used to generate the simulated training data. The paper indicates to a notable increase in accuracy by using a simulation regime that makes use of the same parameters over 5 different Wiener processes, which effectively helps remove the "randomness" and noise from the dataset. They called for more research on the subject, particularly using different ANN architectures. What is notable about the paper presented, is that high accuracy was achieved using a simple network topology. This serves as evidence for the potential of ANNs to act as robust calibration estimators for SDEs.

Giebel *&* Rainer (2013) proposed a novel calibration method for time series that dynamically adapts the parameters of a stochastic model by using small MLP networks (2-layer). These ANNs use data from the past n observations to inform an updated parameter value. They then used the updated stochastic process to forecast the time series one day ahead, while updating the parameters at each time step as they proceed through the time series. They argue that updating the parameters of the stochastic process at each time step is more realistic, due to investors often weighting recent observations as more relevant than aged data. Using this method, different weights can be assigned to data from different dates in the past. The calibration scheme proposed in this paper could potentially add great value to the technique described by Giebel and Rainer, as it would allow the periodic recalibration of more complex processes, and the incorporation of many additional parameters.

CHAPTER 3
METHODOLOGY

3.1. ARTIFICIAL NEURONS

In Chapter 1, ANNs were introduced. Figure 1.1 introduces 3 types of trainable neurons: weights (denoted by w), biases (u), and activation functions (f). Hence, every individual neuron multiplies the set of inputs, x_i by their corresponding weights w_i and adds some bias, u - the sum of which is then fed through an activation function f. The result is propagated forward in the network to the next set of neurons. Mathematically, the output of the kth neuron is given by:

$$\mathbf{o}_k = f\left(\sum_i [\mathbf{w}_{ki}\mathbf{x}_i] + \mathbf{u}_k\right) \tag{3.1}$$

3.1.1 Activation Functions

The activation function f is used to add non-linearity to the architecture. Without it, the model would be little more than a multivariate linear model. There are a number of choices for this activation function. The most popular being rectified linear units (ReLUs), exponential linear units (ELUs), logistic sigmoid- and hyperbolic tangent functions.

Prior to the work of Glorot, Bordes *&* Bengio (2011), logistic sigmoid and hyperbolic tangent functions were the commonest activation functions in neural network architectures.However, Glorot *et al.* (2011) showed that ReLUs yield better performance. ReLUs employ the activation function:

$$f(x) = \max(0, x) \tag{3.2}$$

Clevert, Unterthiner *&* Hochreiter (2015) introduced the "exponential linear unit", which provide even better performance than ReLUs, both in terms of learning speed as well as generalisation potential. An ELU with parameter $\alpha > 0$ has activation function:

$$f(x) = \begin{cases} x & if \quad x > 0 \\ \alpha\left(\exp(x) - 1\right) & if \quad x \leq 0 \end{cases} \tag{3.3}$$

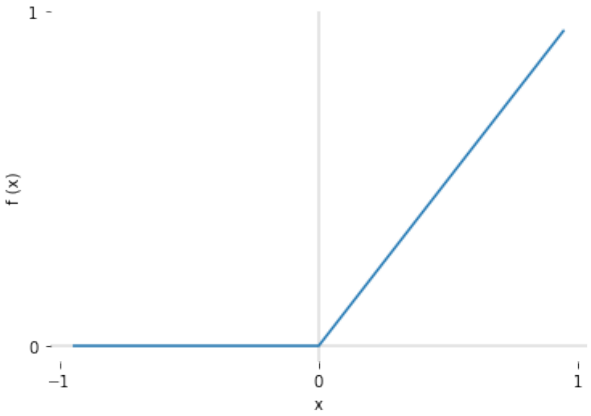


Figure 3.1: Rectifier Activation Function (ReLU), $f(x)$.

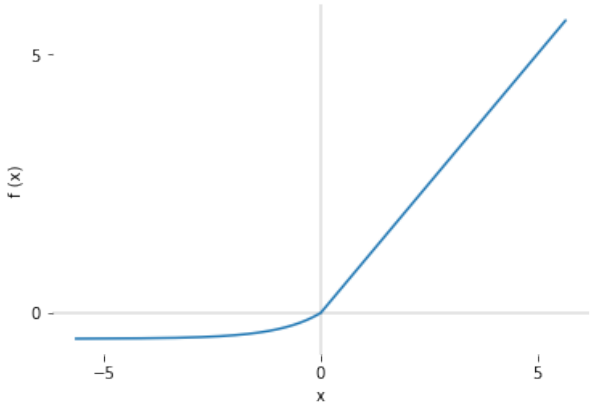


Figure 3.2: Exponential Linear Unit (ELU) Activation Function, with $\alpha = 0.5$.

3.2. CONVOLUTIONAL LAYER

Convolutional Neural Networks (CNN's) are a variation of the traditional multi-layer perceptron architecture. Like ordinary fully connected networks, they consist of neurons with trainable weights and biases. Unlike fully connected networks, these neurons are grouped into sets of small filters. With every forward pass, the filters are sequentially convolved across the extend of the input volume. This produces an activation map containing the result of the "filtration" at every point of the input volume.

In an image-recognition setting, the network will typically train filters that activate on the detection of visual features such as edges or a blotch of colour (Karpathy). However, in this study, there is no definite way to ascertain what they might detect.

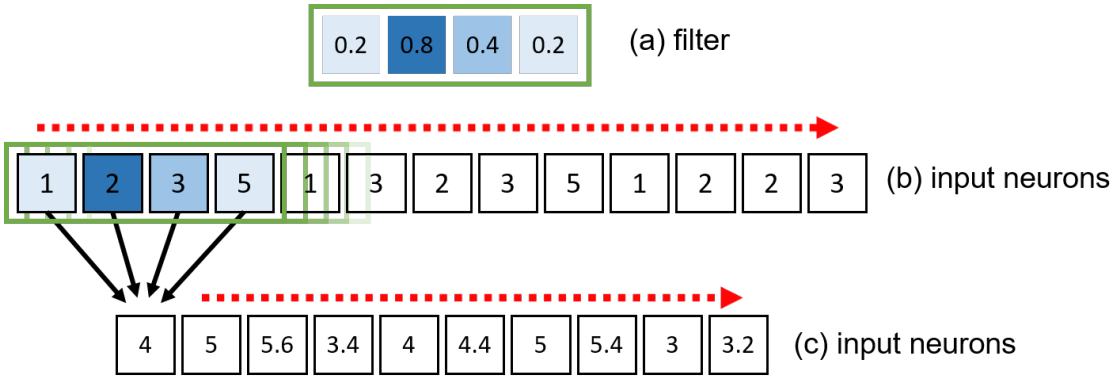


Figure 3.3: A One-Dimensional Convolution Layer

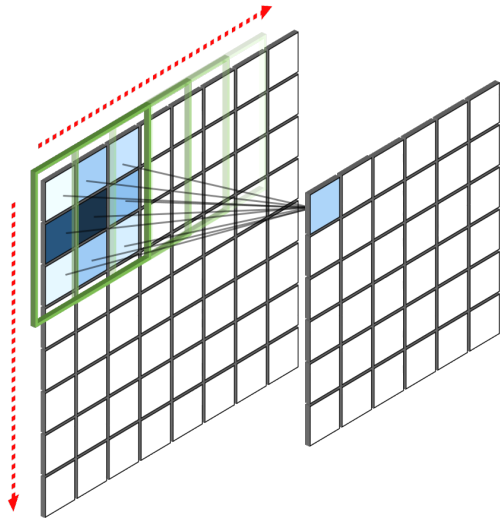


Figure 3.4: A Two-Dimensional Convolution Layer

3.2.1 Pooling

CNN architectures often feature pooling layers, which aggregate the outputs of multiple preceding neurons into a single feature, which is then propagated forward through the network. Two prevalent pooling operations are subsampling (where the mean of the preceding neuron outputs are transmitted to the subsequent layer), as well as maximum pooling - which propagates the maximum output from a set of preceding neuron outputs, to the next layer. The empirical results of Scherer, Müller *&* Behnke (2010) show that "a maximum pooling operation significantly outperforms subsampling operations". In the convolutional network implementations from this dissertation, extensive use was made of maximum pooling operations. Note that - unlike the filters of a convolutional layer - the weights of an average- or maximum pooling operation cannot be adjusted during training, and hence the pooling layers do not form part of the set of trainable network layers.

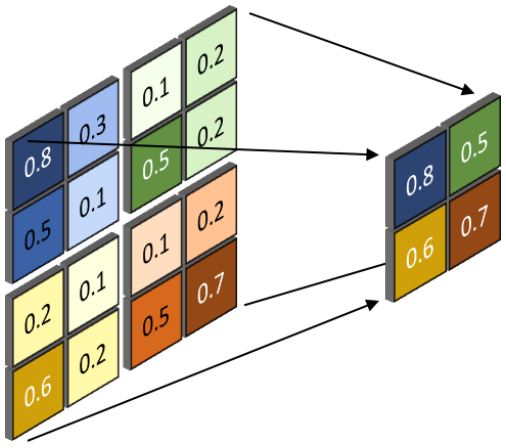


Figure 3.5: A $2x2$ maximum pooling operation.

3.3. TRAINING ARTIFICIAL NEURAL NETWORKS

Before any supervised learning approach can be utilised for inference, it needs to undergo an optimisation process known as "training". Training involves adjusting the appropriate parameters of the network (often referred to as the trainable parameters), in order to minimise the error between the output of the model, and the desired output. The error is defined by some selected loss function, L. The artificial neurons introduced in section 3.1 are "trained" using a technique known as backpropagation.

3.3.1 Backpropagation

Backpropagation consists of two main steps: a propagation step and a parameter adjustment step. In turn, the propagation step consists of both forward- and backward passes.

In the forward propagation step, the inputs are fed through the model to produce a set of output activations, $\mathbf{o}'$. These outputs are then compared to the desired outputs $\mathbf{y}'$, and an error value ϵ' is obtained using some loss function L. That is, $\epsilon' = L(\mathbf{o}', \mathbf{y}')$. These errors are then propagated backwards through the network to generate a set of gradients for each artificial neuron. What follows is the process of gradient descent: the gradient defines a direction (increase or decrease) as well as a magnitude in which to adjust the individual parameter (usually a weight or bias) in order to minimise the error, ϵ' . In the parameter adjustment step, the trainable parameters of the model are each adjusted in the opposite direction of the gradient defined for that parameter, so as to minimise this error.

These steps are repeated until some stopping criteria is reached.

3.4. PERFORMANCE MEASUREMENT

3.4.1 Coefficient of Determination

In measuring the performance of the ANN, we will use the the coefficient of multiple determinations, R^2, between the actual parameter values, $\mathbf{y}$, and the ANN-predicted parameter values $\hat{\mathbf{y}}$. R^2 is defined as

$$R^2 = 1 - \frac{\sum_{i=1}^{m} (y_i - \hat{y}_i)^2}{\sum_{i=1}^{m} (y_i - \overline{y})^2} \tag{3.4}$$

.

where y is the actual parameter value, $\hat{y}$ is the predicted parameter value, $\overline{y}$ is the mean parameter value, and m is the size of the sample. Any estimate that is more accurate than the sample mean would result in an R^2 value of greater than zero. An R^2 value of 1 would indicate a perfect fit.

3.4.2 Average Absolute Percentage Error

Another model evaluation metric is the average absolute percentage error (AAPE).

$$AAPE = 100 \cdot \frac{1}{m} \sum_{i=1}^{m} \frac{\left| y_i - \hat{y}_i \right|}{y_i} \tag{3.5}$$

where y is the actual parameter value, $\hat{y}$ is the predicted parameter value, and m is the size of the sample.

3.4.3 Mean Squared Error

A natural loss function to consider in the optimisation procedure concerned in this dissertation is that of *mean squared error* (MSE). MSE measures the squared mean deviation of the predicted values (yielded by the model under consideration) from the actual observed values.

$$MSE = \sum_{i=1}^{m} \left(y_i - \hat{y}_i \right)^2 \tag{3.6}$$

where y is the actual observed parameter (output) value, $\hat{y}$ is the predicted parameter (output) value, and m is the size of the sample.

CHAPTER 4
SIMULATION STUDY

It is important to note that when building a neural network for a specific modelling exercise, one has little prior knowledge of what the model should look like. Rough guidlines do exist, for example that recurrent neural network (RNN) architectures are often used for time series problems (Karpathy, 2015) or that convolutional neural networks (CNNs) are well suited to image recognition tasks (**?**). Beyond these vague guidelines however, little evidence exists to inform the potential properties that a network might need.

4.1. THE MERTON JUMP-DIFFUSION STOCHASTIC PROCESS

The Merton Jump-Diffusion stochastic process, presented in the seminal work of Merton (1976), aimed to address the limitations of the GBM process. It has the stochastic differential equation,

$$dS_t = \mu S_t dt + \sigma S_t dW_t + S_t dJ_t \tag{4.1}$$

where

$$J_t = \sum_{j=1}^{N_t} (V_j - 1) \tag{4.2}$$

is a compound Poisson process. V_j are independent, identically distributed positive random variables representing the jump sizes. $N_t, t \geq 0$ is a Poisson process with intensity λ, which is independent of J_t and W_t.

To obtain the log returns, we can derive the function $f(t, S_t) = \ln S_t$ using Itô's formula:

$$d\ln S_t = \frac{1}{S_t} dS_t - \frac{1}{2{S_t}^2} (dS_t)^2 = \left(\mu - \frac{\sigma^2}{2}\right) dt + \sigma dW_t + dJ_t \tag{4.3}$$

In this dissertation we will have V_j follow a log-normal distribution with parameters μ_{jumps} and σ_{jumps}.

4.1.1 The Parameters

To better understand the effects of the parameters (μ, σ, λ, μ_{jumps} and σ_{jumps}) on the resulting process, it is important to note that the Merton Jump-Diffusion process can effectively be separated into two distinct components: the diffusion process (GBM), as well as a jumps process. This is illustrated in figure 4.1 below.

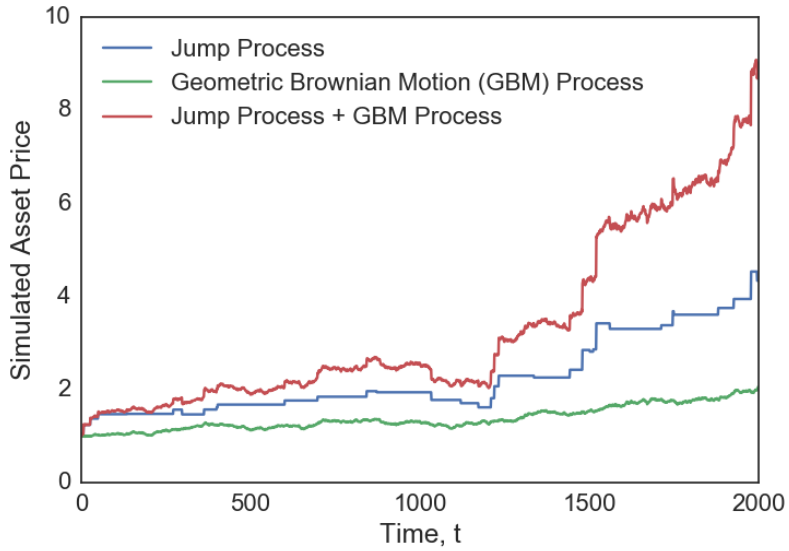


Figure 4.1: The simulated asset price resulting from a Merton Jump-Diffusion returns process, as well as that of its two components: a jumps process, and a GBM process.

Mu (μ) and sigma (σ)

Both mu (μ) and sigma (σ) form part of the GBM component of the process: μ defines the mean returns yielded from the process, while σ specifies the volatility of the returns. The effects of these parameters on the process are illustrated in figures A.1 and A.2 in appendix A.

Lambda (λ), jumps mu (μ_{jumps}) and jumps sigma (σ_{jumps})

Lambda (λ), jumps mu (μ_{jumps}) and jumps sigma (σ_{jumps}) form part of the jump-process. Lambda (λ) is the intensity parameter of the Poisson process, N_t, and hence defines the probability that a jump might occur at any given time. The jumps are drawn from a normal distribution with parameters μ_{jumps} and σ_{jumps}. These parameters hence define the mean and deviation of the jump sizes respectively. The effects of these parameters on the process are illustrated in figures A.3, A.5 and A.4 in appendix A.

4.1.2 Simulation

The neural network models will be trained on the log-returns of the simulated processes. These processes will be simulated using a random set of parameters (within certain bounds).

The simulated parameters, μ, σ, λ, μ_{jumps} and σ_{jumps} will be constrained to the following bounds: $\mu \in [-1, 1]$, $\sigma \in [0.001, 0.2]$, $\lambda \in [0.0001, 0.025]$, $\sigma_{jumps} \in [0.001, 0.2]$, and $\mu_{jumps} \in [-0.5, 0.5]$.

4.2. FULLY CONNECTED NEURAL NETWORK

4.2.1 Multiple Parameter Prediction Architecture

The first study was done on a standard 9-layer fully connected multi-perceptron architecture, trained to predict all five parameters of the Merton Jump-Diffusion process at once.

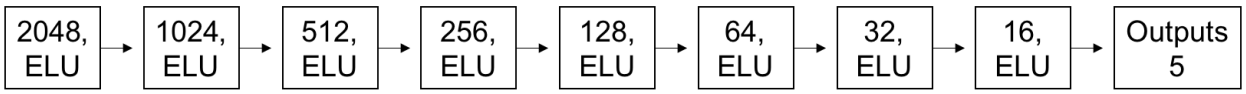


Figure 4.2: 9-Layer Fully Connected Feed-forward ANN

The experiments were performed using ELU activation functions in the network.

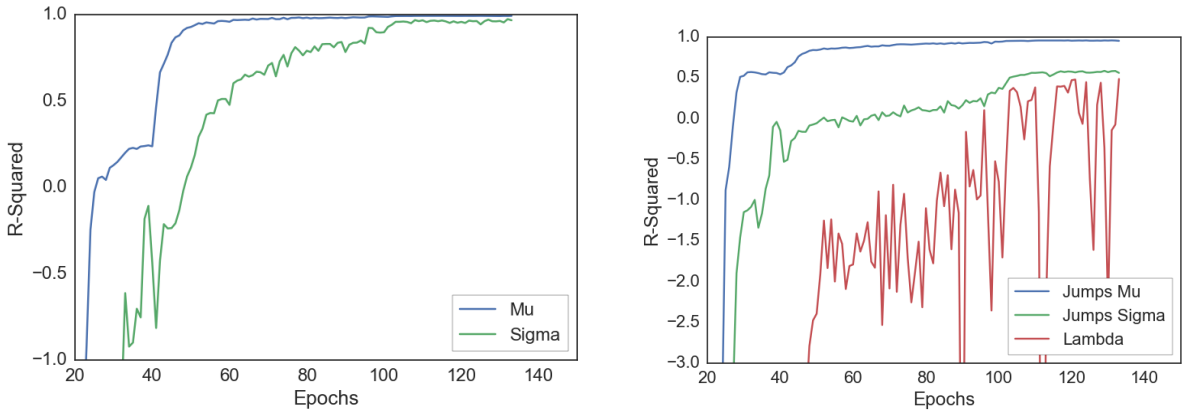
4.2.2 The Dataset

The dataset on which the Fully Connected ANN was trained, was created using the following steps:

1) Random sets of parameters, $\mathbf{z}_i$, with ($\mu \in [-1, 1]$, $\sigma \in [0.001, 0.2]$, $\lambda \in (0, 0.025]$, $\mu_{jumps} \in [-0.5, 0.5]$ and $\sigma_{jumps} \in [0.001, 0.2]$) were uniformly generated.
2) The daily log returns from a Merton Jump-Diffusion stochastic process (equation 4.3) were simulated using these randomly generated parameter sets, $\mathbf{z}_i$, as parameters.
3) For each set of returns, the first 20 sample moments, as well as the autocorrelations up to the first 40 lags were calculated.
4) For every parameter set, its corresponding set of moments and autocorrelations were fed into the input layer of the ANN described in subsection 4.2.1 above. The ANN was then trained, using the process of backpropagation, to produce an estimate, $\mathbf{o}_i$, of the original set of parameters, $\mathbf{z}_i$. This was done by minimizing the mean squared error between the vectors $\mathbf{z}_i$ and $\mathbf{o}_i$ (see the diagram in figure 1.2).

4.2.3 Training

Figures 4.3a and 4.3b illustrate the R-Squared values throughout the training process of the fully connected architecture.



(a) σ, μ **(b)** λ, σ_{jumps}, μ_{jumps}

Figure 4.3: R-Squared values over the training process for the multiple parameter prediction fully connected architecture (ELU activation units).

4.3. CONVOLUTIONAL NEURAL NETWORK

4.3.1 Multiple Parameter Prediction Architecture

The second study was done on a fairly standard 8-layer convolutional architecture, which produced estimates for all five parameters.

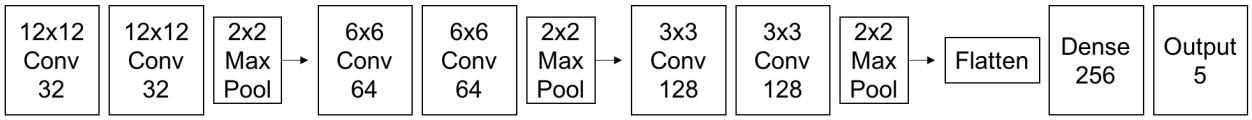


Figure 4.4: 8-Layer CNN

Experiments were performed using ELU activation functions.

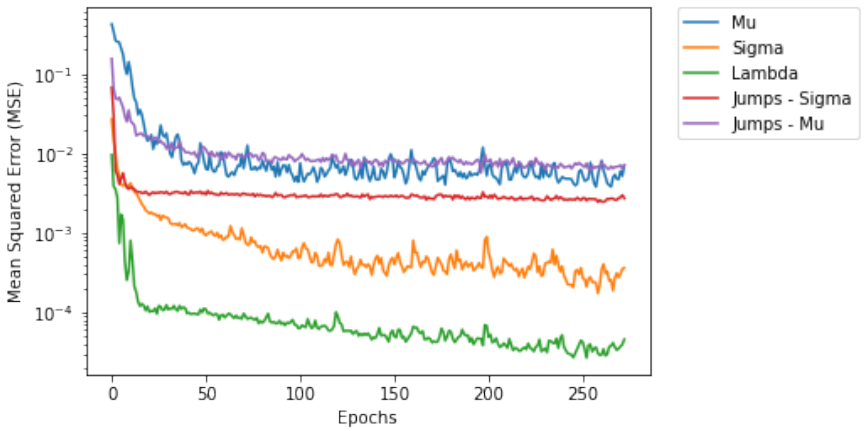
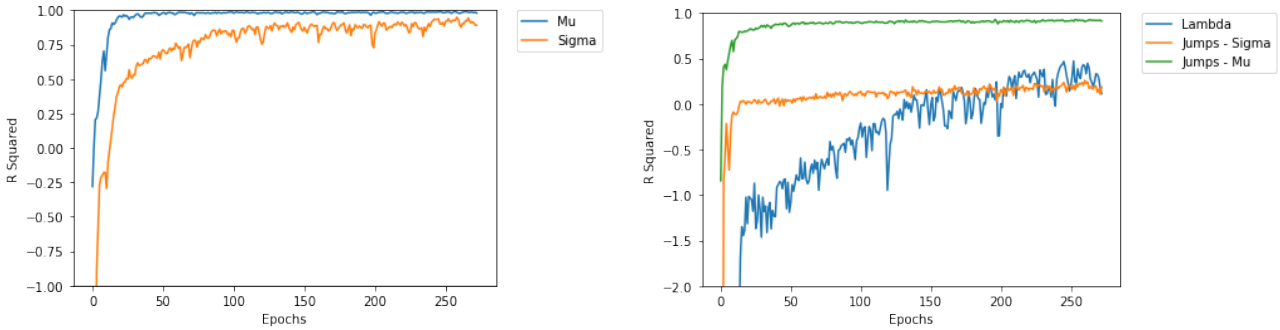


Figure 4.5: MSE values over the training process for the multiple parameter prediction convolutional architecture. Each epoch involves 10 iterations of a simulated batch of 150 randomly selected process parameters.

The reasons for the large difference in MSE values are related to the nature of the parameters. For example Mu (μ) - the drift of the GBM component of the process - can take on values between

-0.5 and 0.5, while Lambda (λ_{jumps}) - the probability of a jump occurring at any given point, can only take on values between 0 and 0.003.



(a) σ, μ **(b)** λ, σ_{jumps}, μ_{jumps}

Figure 4.6: R-Squared values over the training process for the multiple parameter prediction convolutional architecture (ELU activation units).

4.3.2 Dedicated Single Parameter Prediction Architecture

In training the multiple output model, one might notice a slight oscillation in the accuracy of the parameters. At higher levels of accuracy, as the network becomes more accurate at predicting one output, it might become less accurate for another. There seems to exist a payoff, whereby the accuracy reduces as the prediction accuracy for another increases.

This raises the question of a dedicated network architecture for every parameter. A slight variation on the architecture used in 4.3.1 above was implemented. The structure was scaled down to six trainable layers, with a larger penultimate layer. The architecture only outputs an estimate for a single parameter. Hence, using this architecture, separate networks will be used to individually estimate the values of each of the parameters, μ, σ, λ, μ_{jumps} and σ_{jumps}.

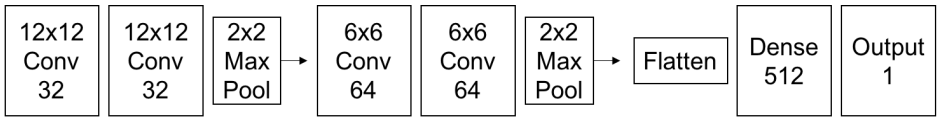


Figure 4.7: 6-Layer CNN

The following sections will discuss the convergence of the dedicated models in comparison to the multiple output prediction model. The final prediction accuracy of the model will be discussed in Chapter 5.

Lambda

The most obvious parameter estimation issue in the multiple output model of 4.3.1, exists with λ, which - as clearly visible in figure 4.6b - exhibits a reluctance to converge to a desired level of accuracy. The dedicated single architecture defined in figure 4.7 was implemented and trained to predict only the single parameter value λ per sample path.

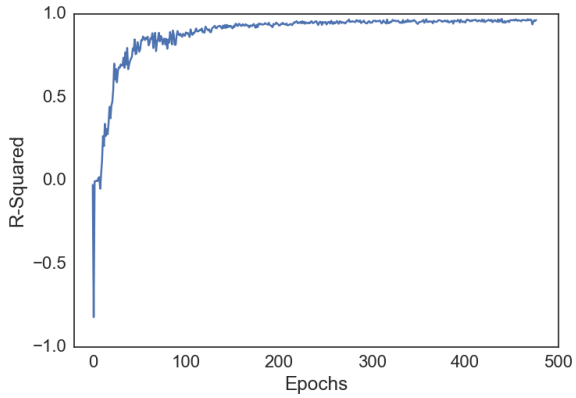


Figure 4.8: R-Squared values for the estimates of λ over the training process for the single parameter prediction convolutional architecture (ELU activation units).

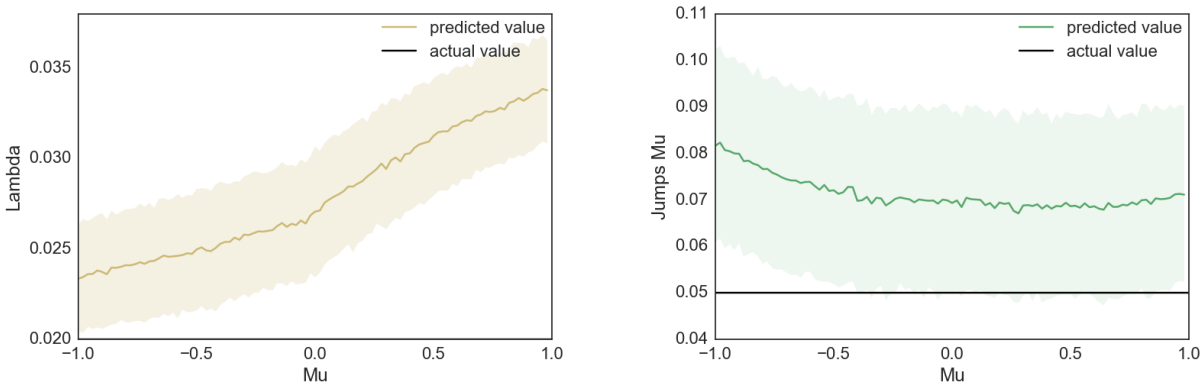
The result is a much quicker and smoother convergence to an acceptable level of accuracy, as visible in figure 4.8. Compare this to the very rough convergence of λ in figure 4.6b. Quicker convergence leads to less training time and ultimately easier use of the model.

4.4. SENSITIVITY TESTS

Due to the nature of the parameters involved in the Merton Jump-Diffusion process, one could expect interactions between parameter estimates. For example, increasing the μ_{jumps} parameter might cause "confusion", since a model could "interpret" larger jumps as a higher rate of volatility from the GBM σ component, and hence produce higher values of σ. What follows is an investigation into how sensitive the individual parameter estimates are with respect to changes in the magnitudes of the other parameters.

This investigation was performed using 1000 simulated sample paths from a Merton Jump-Diffusion process with parameters $\mu, \sigma, \lambda, \mu_{jumps}$ and σ_{jumps}. With each test, a single parameter was selected to be varied during the training process. The effect of this parameter change on the ANN performance was then monitored by reading the estimates of each of the other parameters and comparing them to their actual values.

Variations in μ



(a) The λ parameter estimate plotted against the actual parameter value, $\lambda = 0.02$, for different values of μ.

(b) The $\hat{\mu}_{jumps}$ parameter estimate plotted against the actual parameter value, $\mu_{jumps} = 0.05$, for different values of μ.

Figure 4.9: The parameter estimates (with 68% confidence interval) of $\hat{\sigma}$, $\hat{\lambda}$, $\hat{\mu}_{jumps}$ and $\hat{\sigma}_{jumps}$, plotted against their actual values ($\sigma = 0.1$, $\lambda = 0.02$, $\mu_{jumps} = 0.05$ and $\sigma_{jumps} = 0.07$), while varying the μ parameter in the range $(-1.0, 1.0)$.

Of all the parameters involved in the Merton Jump-Diffusion process, μ arguably has the least interaction with the estimates of the other parameters.

It is rather strange that the model exhibits a tendency to consistently underestimate the value of λ (Figure 4.9a). This is clear throughout the investigation - λ is almost always underestimated. This is not something to be expected, since ANNs have the property of easily being able to correct for bias. Similarly, all other parameters (besides μ) kept constant, the model seems to consistently overestimate the value of μ_{jump} (Figure 4.9b). As with the estimate of λ, one would expect and ANN to be able to easily correct for this clear bias.

Variations in σ

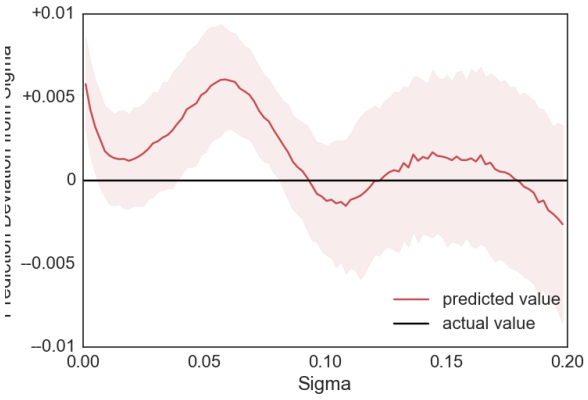
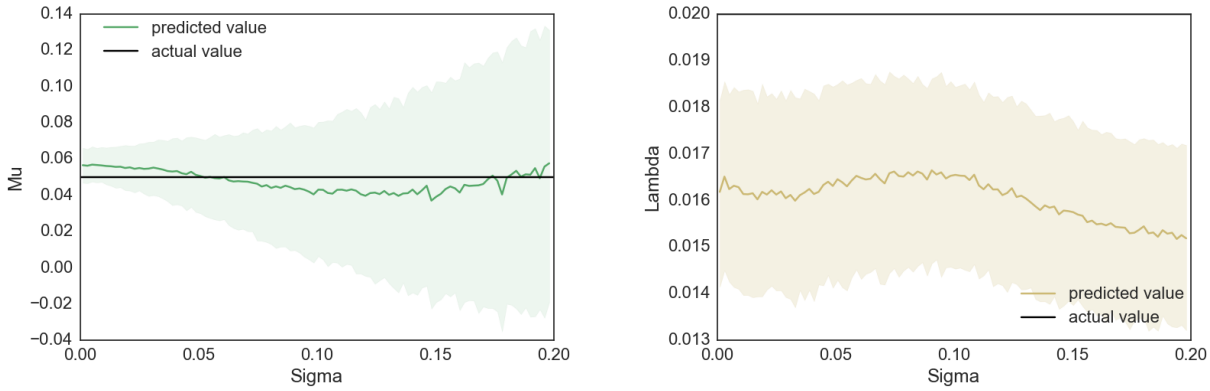


Figure 4.10: The deviation (with 68% confidence interval) of the $\hat{\sigma}$ parameter estimate from the actual parameter value, σ, for different values of σ. All the other parameters are kept constant as $\mu = 0.05, \lambda = 0.02, \mu_{jumps} = 0.05$ and $\sigma_{jumps} = 0.07$.

As to be expected, larger volatility tends to widen the confidence interval around a particular parameter estimate. This is particularly clear in figure 4.10, where the confidence regarding the estimate of $\hat{\sigma}$ falls, as σ increases. The most clear-cut illustration of this can be seen in the estimates of $\hat{\mu}$ in figure 4.11a, where there is a clearly visible widening of the confidence interval as sigma increases.



(a) The mean $\hat{\mu}$ parameter estimate plotted against the actual parameter value, $\mu = 0.05$, for different values of σ.

(b) The mean λ parameter estimate plotted against the actual parameter value, $\lambda = 0.02$, for different values of σ.

Figure 4.11: The mean parameter estimates (with 68% confidence interval) of $\hat{\mu}$, $\hat{\lambda}$, $\hat{\mu}_{jumps}$ and $\hat{\sigma}_{jumps}$, plotted against their actual values ($\mu = 0.05$, $\lambda = 0.02$, $\mu_{jumps} = 0.05$ and $\sigma_{jumps} = 0.07$), while varying the σ parameter in the range $(0, 0.2)$.

Variations in λ

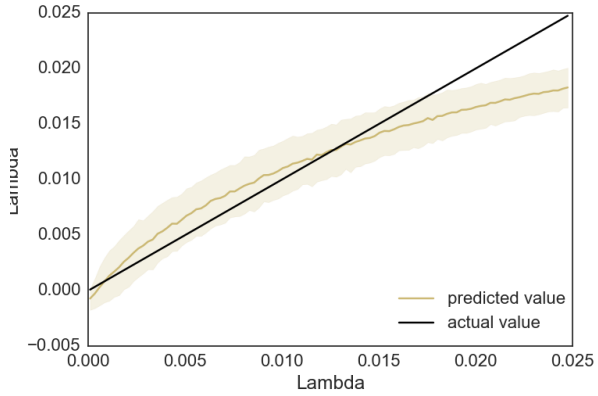
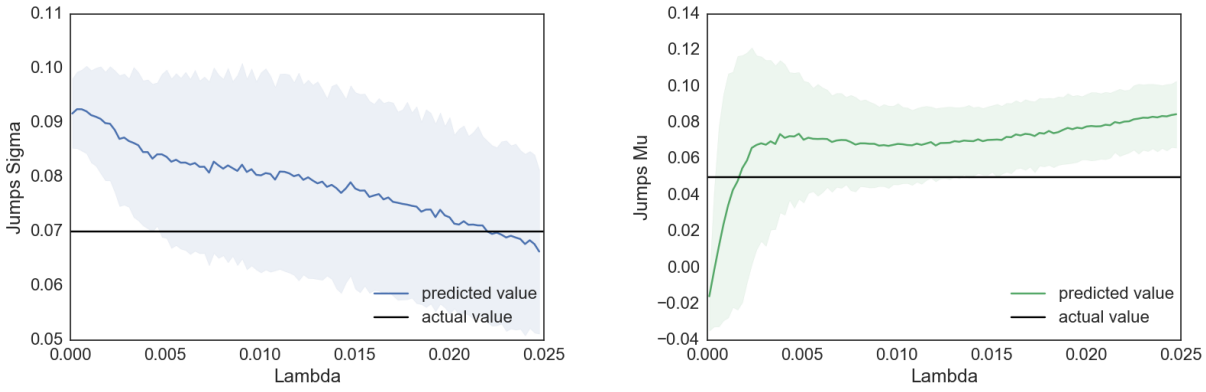


Figure 4.12: The mean deviation (with 68% confidence interval) of the $\hat{\lambda}$ parameter estimate from the actual parameter value, λ, for different values of λ. All the other parameters are kept constant as $\mu = 0.05, \sigma = 0.1, \mu_{jumps} = 0.05$ and $\sigma_{jumps} = 0.07$.

The multiple output prediction model shows a slight overestimation of $\hat{\lambda}$ for small values of λ, and a slight underestimation for larger values of λ (Figure 4.12)

Figure 4.13a shows an interesting relationship between the value of $\hat{\sigma}_{jumps}$ and λ. The model estimate of $\hat{\sigma}_{jumps}$ shows an inversely proportional relationship to the value of λ. Larger λ also seems to make the model "less confident" in it's estimate of σ_{jumps}.

In figure 4.13b, we see that the opposite is true for the estimate of $\hat{\mu}_{jumps}$. As might be expected, small λ values tend to suffocate the estimate of $\hat{\mu}_{jumps}$. It is conceivable that lower values of λ might result in less jumps, which could be "misinterpreted" as a lower mean jump size.

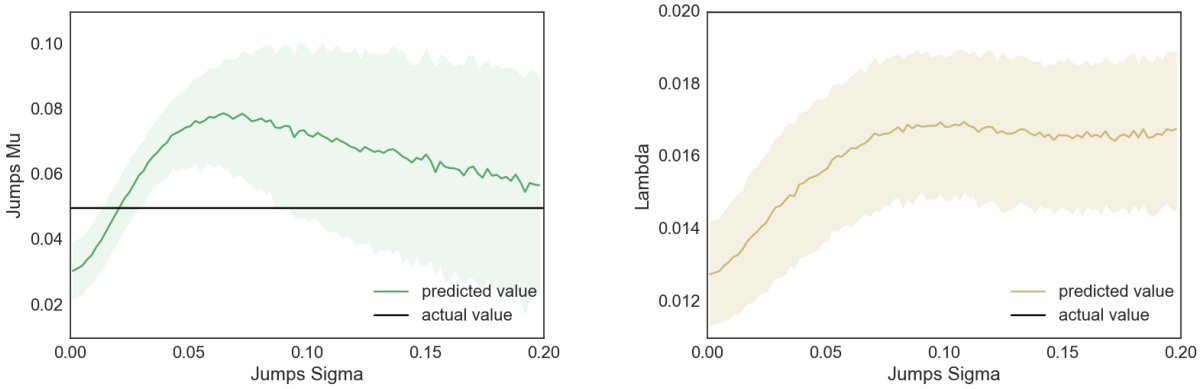


(a) The mean $\hat{\sigma}_{jumps}$ parameter estimate plotted against the actual parameter value, $\sigma_{jumps} = 0.07$, for different values of λ.

(b) The mean $\hat{\mu}_{jumps}$ parameter estimate plotted against the actual parameter value, $\mu_{jumps} = 0.05$, for different values of λ.

Figure 4.13: The mean parameter estimates (with 68% confidence interval) of $\hat{\mu}$, $\hat{\sigma}$, $\hat{\mu}_{jumps}$ and $\hat{\sigma}_{jumps}$, plotted against their actual values ($\mu = 0.05$, $\sigma = 0.1$, $\mu_{jumps} = 0.05$ and $\sigma_{jumps} = 0.07$), while varying the λ parameter in the range $(0, 0.025)$.

Variations in σ_{jumps}



(a) The mean $\hat{\mu}_{jumps}$ parameter estimate plotted against the actual parameter value, $\mu_{jumps} = 0.05$, for different values of μ_{jumps}.

(b) The mean λ parameter estimate plotted against the actual parameter value, $\lambda = 0.2$, for different values of σ_{jumps}.

Figure 4.14: The mean parameter estimates (with 68% confidence interval) of $\hat{\mu}$, $\hat{\sigma}$, $\hat{\mu}_{jumps}$ and λ, plotted against their actual values ($\mu = 0.05$, $\sigma = 0.1$, $\mu_{jumps} = 0.05$ and $\lambda = 0.02$), while varying the σ_{jumps} parameter in the range $(0, 0.2)$.

The value of σ_{jumps} has notable effects on the errors of the estimates of $\hat{\mu}_{jumps}$ and $\hat{\lambda}$.

As to be expected, larger σ_{jumps} tend to lower the confidence associated with the estimate, $\hat{\mu}_{jumps}$ (Figure 4.14a).

Figire 4.14b illustrates how σ_{jumps} puts upward pressure on the estimate of λ. This might indicate that the model is not sure as to whether "more jumps" just means "greater volatility" in the jump sizes.

Variations in μ_{jumps}

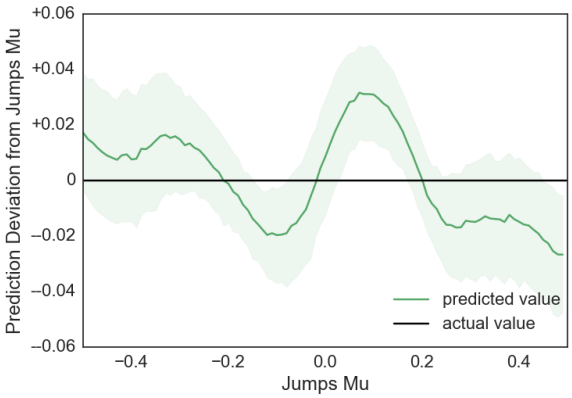
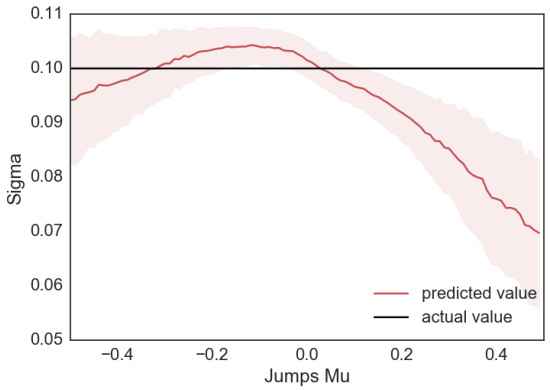
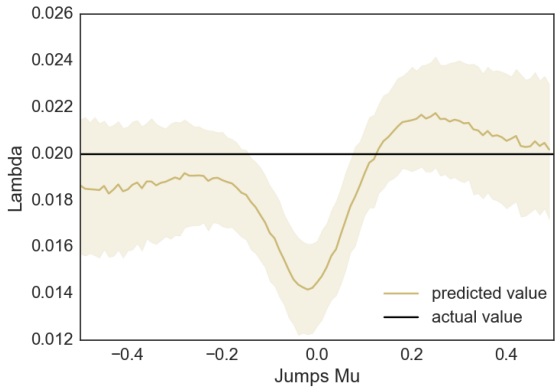


Figure 4.15: The mean deviation (with 68% confidence interval) of the $\hat{\mu}_{jumps}$ parameter estimate from the actual parameter value, μ_{jumps}, for different values of μ_{jumps}. All the other parameters are kept constant as $\mu = 0.05, \sigma = 0.1, \lambda = 0.02$ and $\sigma_{jumps} = 0.07$.

Variations in μ_{jumps} notably affect almost all of the parameter estimates produced by the CNN Multiple Output model.



(a) The mean σ parameter estimate plotted against the actual parameter value, $\sigma = 0.1$, for different values of μ_{jumps}.



(b) The mean λ parameter estimate plotted against the actual parameter value, $\lambda = 0.2$, for different values of μ_{jumps}.

Figure 4.16: The mean parameter estimates (with 68% confidence interval) of $\hat{\mu}$, $\hat{\sigma}$, $\hat{\sigma}_{jumps}$ and λ, plotted against their actual values ($\mu = 0.05$, $\sigma = 0.1$, $\lambda = 0.02$ and $\sigma_{jumps} = 0.07$), while varying the μ_{jumps} parameter in the range $(-0.5, 0.5)$.

4.5. CONCLUSION

While a multiple output model might be more convenient to build and use, it was illustrated that single output models are quicker to train and more robust. The author's study also found that, while a fully connected architecture based on the moments and autocorrelations of the process performed well for the simpler parameters such as μ and σ, its estimates of λ were less consistent. Figure 4.3b demonstrates the difficulty of training the fully connected model to estimate the λ parameter.

In particular for the λ parameter, a single-output convolutional model was quicker to train than either of the multiple output models (figure 4.8).

The full sets of sensitivity tests are available in Appendix B, while the results will be discussed in Chapter 5.

CHAPTER 5
RESULTS

First, the individual accuracy of the selected architectures were investigated. 1000 simulated sample paths from a Merton Jump-Diffusion process with parameters, $\mu = 0.05$, $\sigma = 0.1$, $\lambda = 0.02$, $\mu_{jumps} = 0.05$ and $\sigma_{jumps} = 0.07$ were used. The (already trained) NN models were then used to predict these original parameters, given the set of 1000 sample paths.

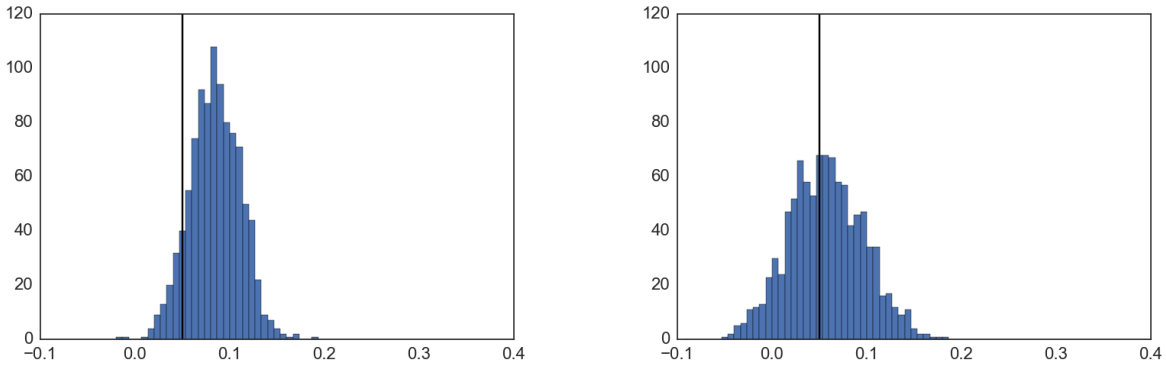
The parameter estimation study investigated and compared results using:

- MME,
- Likelihood profiling of Honore (1998),
- MLE,
- a multiple output convolutional ANN model as defined in section 4.3.1 above,
- a fully connected NN as defined in section 4.2.1, and
- a dedicated single output convolutional ANN model as defined in section 4.3.2 above.

The full set of results is available in appendix C. Selected portions thereof have been published as part of this chapter.

Mu, μ

The results obtained from the CNN architecture in estimating μ compare well to that of those obtained by MLE. Both methods exhibit bias, generally overestimating μ in the selected parameter set.



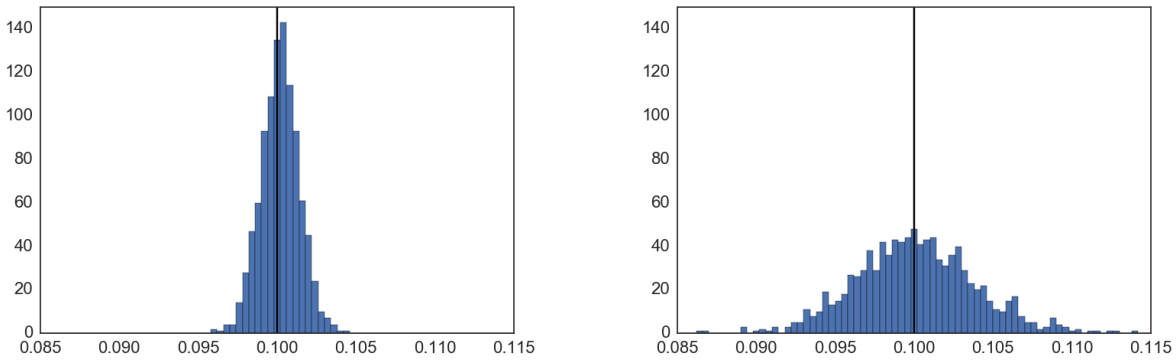
(a) MLE

(b) CNN Architecture - Multiple Output - ELU

Figure 5.1: Various model distributions of the predicted values of μ with true value $0,05$. All other parameters were kept constant ($\sigma = 0.1$, $\lambda = 0.02$, $\mu_{jumps} = 0.05$ and $\sigma_{jumps} = 0.07$).

Sigma, σ

MLE yields markedly accurate estimates for σ.



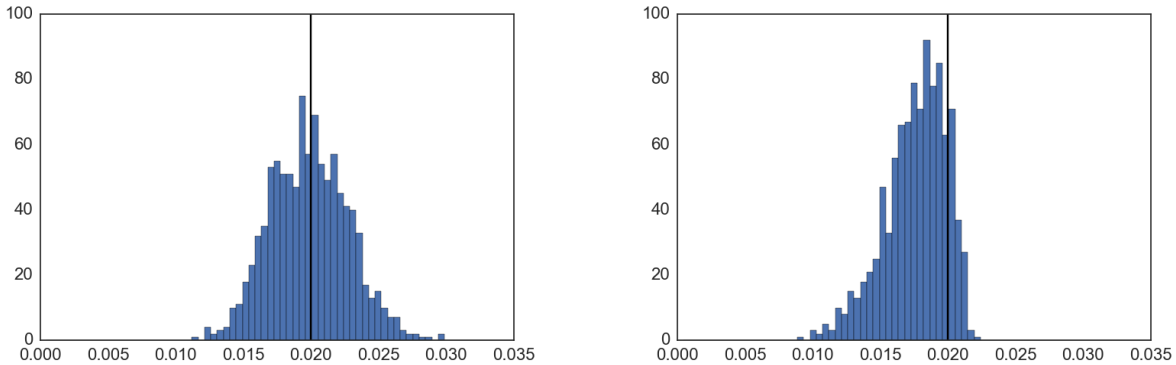
(a) MLE

(b) CNN Architecture - Multiple Output - ELU

Figure 5.2: Various model distributions of the predicted values of σ with true value $0,1$.

Lambda, λ

In estimating λ, the single output model outshines the multiple output CNN. Both provide slightly biased estimates, which could be due to the training process not having fully converged. This could be solved by training for a longer period of time on better hardware (for instance a GPU with more memory).



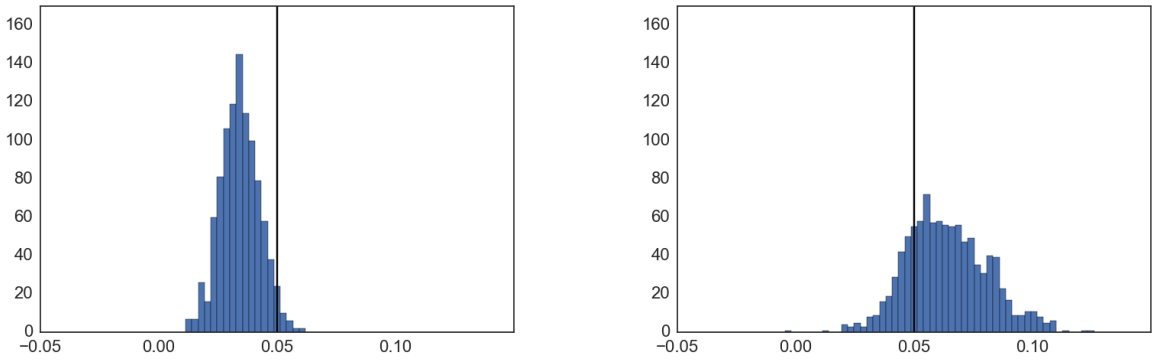
(a) MLE

(b) CNN Architecture - Single Output - ELU

Figure 5.3: Various model distributions of the predicted values of λ with true value $0,02$.

Jumps Mu, μ_{jumps}

MLE slightly underestimates μ_{jumps}, while the CNN architectures provide less certain estimates. Again, the single output model clearly outperforms the multiple output CNN (see figures C.4d and C.4f in Appendix C)

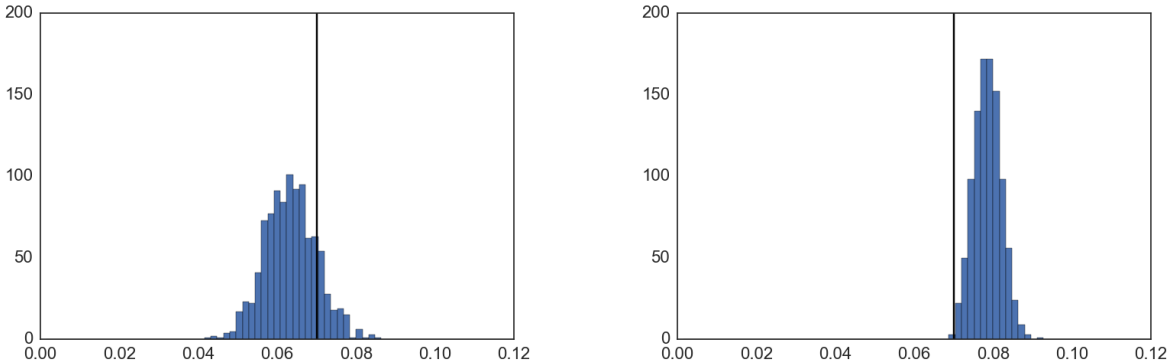


(a) MLE **(b)** CNN Architecture - Single Output - ReLU

Figure 5.4: Various model distributions of the predicted values of μ_{jumps} with true value $0,05$.

Jumps Sigma, σ_{jumps}

MLE slightly underestimates σ_{jumps}, while the fully connected ANN architectures provide very certain - albeit biased - estimates. Both the dedicated single output and multiple output CNN models provide very uncertain estimates of σ_{jumps}. This could be due to the training process not having completely converged (see figures C.5d and C.5f in Appendix C).



(a) MLE **(b)** Fully Connected Architecture - Multiple Output - ELU

Figure 5.5: Various model distributions of the predicted values of σ_{jumps} with true value $0,07$.

5.1. METHOD COMPARISON

While the architectures have demonstrated the ability to yield acceptable estimates for complex stochastic processes, there are several practical considerations to take into account when using these models. This section will consider the practicality and limitations associated with using ANNs for calibration purposes.

5.1.1 Practicality

ANNs can be difficult to build and time consuming to train. Currently one needs access to a graphics processing unit (GPU) in order to train the models described in this paper. Even worse, the software used (Tensorflow™ (Abadi *et al.*, 2015)) requires the Nvidia CUDA® parallel computing platform, which is only available on Nvidia GPUs.

Even with good hardware, the training process can be time consuming. The models in this dissertation were trained using a g3 instance from Amazon Web Services (AWS) Elastic Compute Cloud (EC2), which provides an Nvidia Tesla M60 GPU. Despite having access to this computational power, the dedicated single output CNN trained to predict $\hat{\sigma}_{jumps}$, took more than 36 hours to train.

Online Versus Offline Training

The advantage of training ANNs, is that the process is said to happen "offline". That means that one could train the model in advance, and only the prediction step would need to happen when actually estimating the parameters. The prediction step takes much less time than the training process.

In contrast, the vanilla MLE approach cannot be "trained" in advance.

In the tests performed in this research (done on a dual-core 2,4 GHz Intel Core i5), estimating the parameters of 1000 simulations took more than 10 minutes using MLE, almost an hour using the likelihood profiling approach of Honore (1998), but less than a minute using the pre-trained CNN. This might differ based on the software package used and the programming language used.

5.1.2 Flexibility

Parameter Range

The predictions of an ANN are solely based on the data it was trained on. This means that it would not be able to accurately predict the estimates of the parameters of any sample path it had not been exposed to in training. In this dissertation, for instance, the simulated values for λ were kept in the range $\lambda \in [0.0001, 0.025]$. The model would thus not be able to accurately estimate $\hat{\lambda}$ for any λ with value $\lambda > 0.025$. The traditional estimation procedures (MLE and MME) thus offer greater flexibility in terms of the range of values they're able to accommodate.

It is also possible for ANN models to output nonsensical results, such as negative $\hat{\sigma}$ estimates.

Input Sequence Length

The fully connected ANNs and CNNs described in this paper can only accommodate an input sequence of a fixed length. That is, the sequence length is defined when the model is first built, after which all training and prediction sequences need to be of that same length. In this study, all the input sequence lengths were fixed to 2000 time steps. The traditional procedures offer greater flexibility in terms of the sequence length that they're able to accommodate. There are workarounds to this issue, and in practice one might often decide on a specific sequence length of interest. RNNs in particular are another alternative ANN architecture that can accommodate any input sequence length.

CHAPTER 6
CONCLUSION AND FURTHER RESEARCH

This project investigated the ability of ANNs to estimate the parameters of a sufficiently complex SDE. Chapter 1 introduced the problem of calibrating SDEs, and discussed the limitations of existing approaches. Chapter 2 presented the literature on similar studies. The concepts used throughout the dissertation were explained in Chapter 3, as was the methodology used in the simulation study. The simulation study itself, together with the proposed ANN architectures were presented in Chapter 4. Chapter 5 reported on the results of the study and compared them to those obtained using traditional parameter estimation techniques such as MLE, MME, EM and likelihood profiling.

The simulation study demonstrated the ability of ANNs to act as a calibration scheme for the Merton Jump-Diffusion process. The CNN architecture was able to yield acceptable parameter estimates, within realistic parameter bounds. What this means is that an ANN displays the ability to learn the calibration function and serve as a calibrator for a complex stochastic process. This can be done without the need to derive the likelihood function - which can be difficult to do, as demonstrated by Honore (1998) and Mongwe (2015).

The models presented in this dissertation are far from optimal, with very basic properties common to most ANN architectures. With the advancements in neural network research, and the speed at which deep learning is currently applied to multiple unrelated fields, there is good reason to believe that neural networks have the ability to outperform traditional calibration techniques.

Further Research

This research project aimed to demonstrate the ability of ANNs to act as a calibration scheme for stochastic processes. The question still remains as to whether ANNs could provide a *universal* calibration method for the calibration of *any*, arbitrarily complex stochastic process. Further research could focus on answering this question. It would be of interest to increase the accuracy of the predictions by trying different architectures, in particular: RNNs (which are arguably much more appropriate for time series problems than the CNN architecture used in this paper).

This paper should be extended to other, more complex stochastic processes in order to test the limits of ANNs as universal calibrators of stochastic processes.

Postscript

Stochastic processes are powerful tools which can be applied to almost any time series-related problem. They offer clear, truthful representations of real-world processes in ways in which deterministic formulae cannot. Yet the underlying mathematics complicate their application. What one might hope to achieve is a democratisation of stochastic processes. Neural networks display the potential to offer actuaries the ability to implement and benefit from these statistical tools, without a background in stochastic calculus.

Bibliography

Abadi, M., Agarwal, A., Barham, P., Brevdo, E., Chen, Z., Citro, C., Corrado, G.S., Davis, A., Dean, J., Devin, M., Ghemawat, S., Goodfellow, I., Harp, A., Irving, G., Isard, M., Jia, Y., Jozefowicz, R., Kaiser, L., Kudlur, M., Levenberg, J., Mané, D., Monga, R., Moore, S., Murray, D., Olah, C., Schuster, M., Shlens, J., Steiner, B., Sutskever, I., Talwar, K., Tucker, P., Vanhoucke, V., Vasudevan, V., Viégas, F., Vinyals, O., Warden, P., Wattenberg, M., Wicke, M., Yu, Y. *&* Zheng, X. 2015. TensorFlow: Large-scale machine learning on heterogeneous systems. Software available from tensorflow.org.
Available at: `http://tensorflow.org/`

Barone-Adesi, G. 2015. *Stochastic Processes*. Wiley Encyclopedia of Management. John Wiley and Sons, Ltd. ISBN 9781118785317.
Available at: `http://dx.doi.org/10.1002/9781118785317.weom040071`

Cairns, A., Dickson, D., Macdonald, A., Waters, H. *&* Willder, M. 1998. Stochastic processes: learning the language. *Faculty of Actuaries students' society*.

Chollet, F. *et al.*. 2015. Keras. `https://github.com/fchollet/keras`.

Clevert, D., Unterthiner, T. *&* Hochreiter, S. 2015. Fast and accurate deep network learning by exponential linear units (elus). *CoRR*, abs/1511.07289.
Available at: `http://arxiv.org/abs/1511.07289`

Giebel, S. *&* Rainer, M. 2013. Neural network calibrated stochastic processes: forecasting financial assets. *Central European Journal of Operations Research*, 21(2):277–293.

Glorot, X., Bordes, A. *&* Bengio, Y. 2011. Deep sparse rectifier neural networks. In *Proceedings of the Fourteenth International Conference on Artificial Intelligence and Statistics*, pages 315–323.

Honore, P. 1998. Pitfalls in estimating jump-diffusion models.

Hornik, K., Stinchcombe, M. *&* White, H. 1989. Multilayer feedforward networks are universal approximators. *Neural Networks*, 2(5):359–366.
Available at: `//www.sciencedirect.com/science/article/pii/0893608089900208`

Hunter, J.D. 2007. Matplotlib: A 2d graphics environment. *Computing In Science & Engineering*, 9(3):90–95.

Jones, E., Oliphant, T., Peterson, P. *et al.*. 2001–. SciPy: Open source scientific tools for Python. [Online; accessed <today>].
Available at: `http://www.scipy.org/`

Karpathy, A. Cs231n convolutional neural networks for visual recognition.
Available at: `http://cs231n.github.io/convolutional-networks/`

Karpathy, A. 2015. The unreasonable effectiveness of recurrent neural networks.
Available at: `http://karpathy.github.io/2015/05/21/rnn-effectiveness/`

Kingma, D.P. *&* Ba, J. 2014. Adam: A method for stochastic optimization. *CoRR*, abs/1412.6980.
Available at: http://arxiv.org/abs/1412.6980

Merton, R.C. 1976. Option pricing when underlying stock returns are discontinuous. ID: 271671271671.
Available at: http://www.sciencedirect.com.ez.sun.ac.za/science/article/pii/0304405X76900222

Mongwe, W.T. 2015. No title. *Analysis of equity and interest rate returns in South Africa under the context of jump diffusion processes*.

Nielsen, J.N., Madsen, H. *&* Young, P.C. 2000. Parameter estimation in stochastic differential equations: An overview. ID: 271897271897.
Available at: http://www.sciencedirect.com/science/article/pii/S1367578800900178

Olden, J.D. *&* Jackson, D.A. 2002. Illuminating the "black box": a randomization approach for understanding variable contributions in artificial neural networks. *Ecological Modelling*, 154(1-2):135–150.
Available at: //www.sciencedirect.com/science/article/pii/S0304380002000649

Oliphant, T. *et al.*. 2001–. Numpy: Open source array tools for Python. [Online; accessed <today>].
Available at: http://www.numpy.org/

Oreskes, N., Shrader-Frechette, K. *&* Belitz, K. 1994. Verification, validation, and confirmation of numerical models in the earth sciences. *Science*, 263(5147):641–646.
Available at: http://www.jstor.org/stable/2883078

Pérez, F. *&* Granger, B.E. 2007. IPython: a system for interactive scientific computing. *Computing in Science and Engineering*, 9(3):21–29. ISSN 1521-9615.
Available at: http://ipython.org

Reid, S.G. 2015. Random walks down wall street, stochastic processes in python.
Available at: http://www.turingfinance.com/random-walks-down-wall-street-stochastic-processes-

Rossum, G. 1995. Python reference manual. Technical Report, Amsterdam, The Netherlands, The Netherlands.

Samad, T. *&* Mathur, A. 1992. Parameter estimation for process control with neural networks. ID: 271876271876.
Available at: http://www.sciencedirect.com/science/article/pii/0888613X9290008N

Scherer, D., Müller, A. *&* Behnke, S. 2010. *Evaluation of Pooling Operations in Convolutional Architectures for Object Recognition*, pages 92–101. Berlin, Heidelberg: Springer Berlin Heidelberg. ISBN 978-3-642-15825-4.
Available at: http://dx.doi.org/10.1007/978-3-642-15825-4_10

Seabold, S. *&* Perktold, J. 2010. Statsmodels: Econometric and statistical modeling with python. In *9th Python in Science Conference*.

Teugels, J.L. *&* Sundt, B. 2004. *Encyclopedia of actuarial science*. Hoboken, NJ: Hoboken, NJ : John Wiley and Sons. Includes bibliographical references and index.

Waskom, M., Botvinnik, O., drewokane, Hobson, P., David, Halchenko, Y., Lukauskas, S., Cole, J.B., Warmenhoven, J., de Ruiter, J., Hoyer, S., Vanderplas, J., Villalba, S., Kunter, G., Quintero, E., Martin, M., Miles, A., Meyer, K., Augspurger, T., Yarkoni, T., Bachant, P., Williams, M., Evans, C., Fitzgerald, C., Brian, Wehner, D., Hitz, G., Ziegler, E., Qalieh, A. *&* Lee, A. 2016. seaborn: v0.7.1 (june 2016).
Available at: `https://doi.org/10.5281/zenodo.54844`

Werbos, P.J. 1990. Backpropagation through time: what it does and how to do it. *Proceedings of the IEEE*, 78(10):1550–1560.

Xie, Z., Kulasiri, D., Samarasinghe, S. *&* Rajanayaka, C. 2007. The estimation of parameters for stochastic differential equations using neural networks. *Inverse Problems in Science and Engineering*, 15(6):629–641. Doi: 10.1080/17415970600907429.
Available at: `http://dx.doi.org.ez.sun.ac.za/10.1080/17415970600907429`

Yao, J., Li, Y. *&* Tan, C.L. 2000. Option price forecasting using neural networks. *Omega*, 28(4):455–466.
Available at: `//www.sciencedirect.com/science/article/pii/S0305048399000663`

APPENDIX A
MERTON JUMP-DIFFUSION PROCESS PARAMETERS

Mu, μ

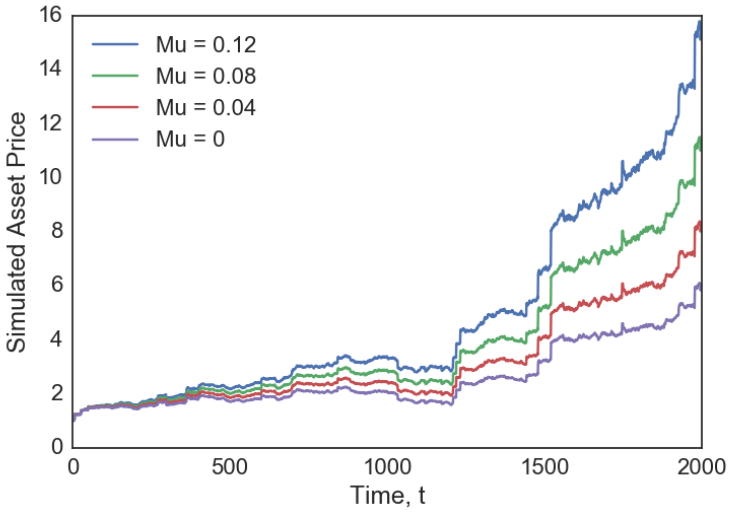


Figure A.1: The effects of changes in μ to the simulated asset price resulting from a Merton Jump-Diffusion returns process. The random seed was set constant at a value of 1234. The other parameters are held constant at $\sigma = 0.1, \lambda = 0.02, \sigma_{jumps} = 0.07$, and $\mu_{jumps} = 0.05$.

Sigma, σ

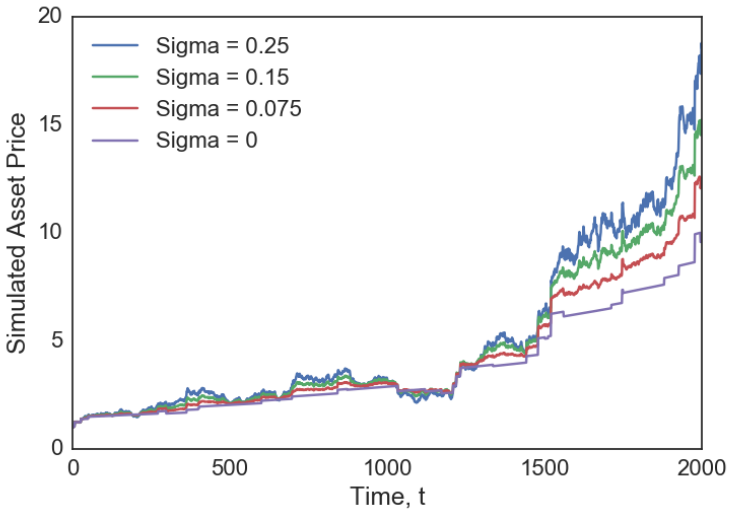


Figure A.2: The effects of changes to σ on the simulated asset price resulting from a Merton Jump-Diffusion returns process. The random seed was set constant at a value of 1234. The other parameters are held constant at $\mu = 0.1, \lambda = 0.02, \sigma_{jumps} = 0.07$, and $\mu_{jumps} = 0.05$.

Lambda, λ

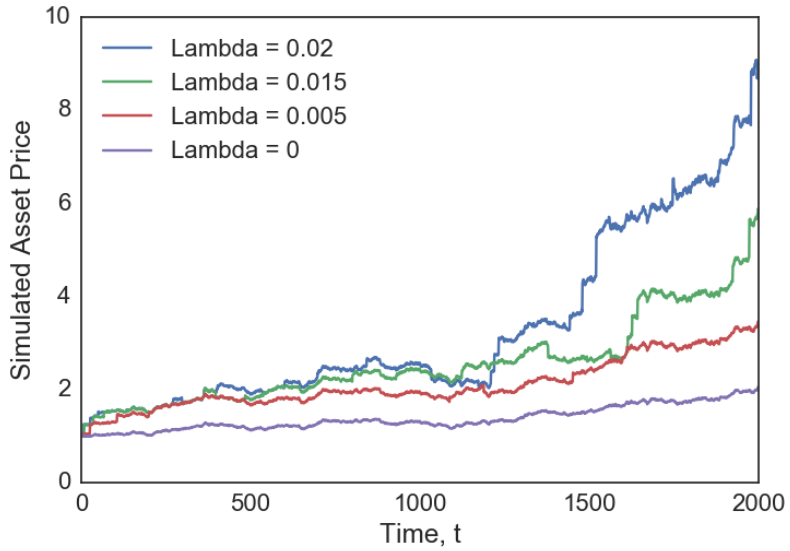


Figure A.3: The effects of changes to λ on the simulated asset price resulting from a Merton Jump-Diffusion returns process. The random seed was set constant at a value of 1234. The other parameters are held constant at $\sigma = 0.1, \mu = 0.05, \sigma_{jumps} = 0.07$, and $\mu_{jumps} = 0.05$.

Jumps Sigma, σ_{jumps}

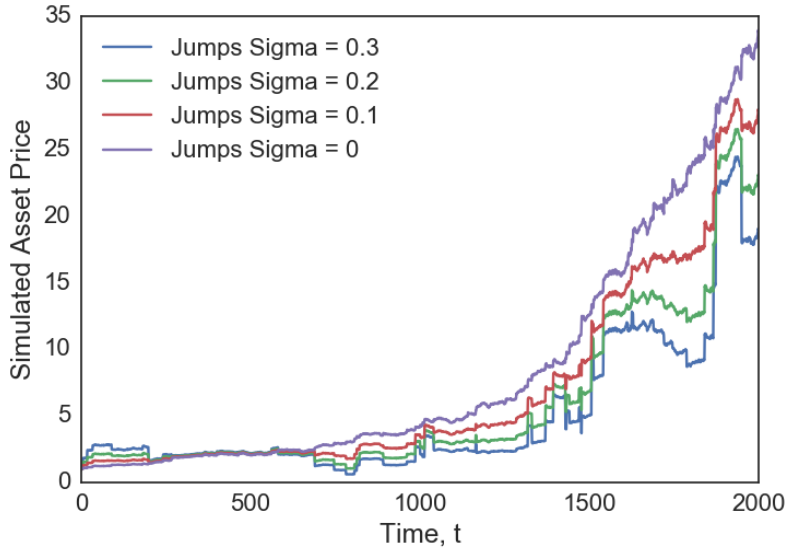


Figure A.4: The effects of changes to σ_{jumps} on the simulated asset price resulting from a Merton Jump-Diffusion returns process. The random seed was set constant at a value of 1234. The other parameters are held constant at $\sigma = 0.1, \mu = 0.05, \lambda = 0.03$, and $\mu_{jumps} = 0.05$.

Jumps Mu, μ_{jumps}

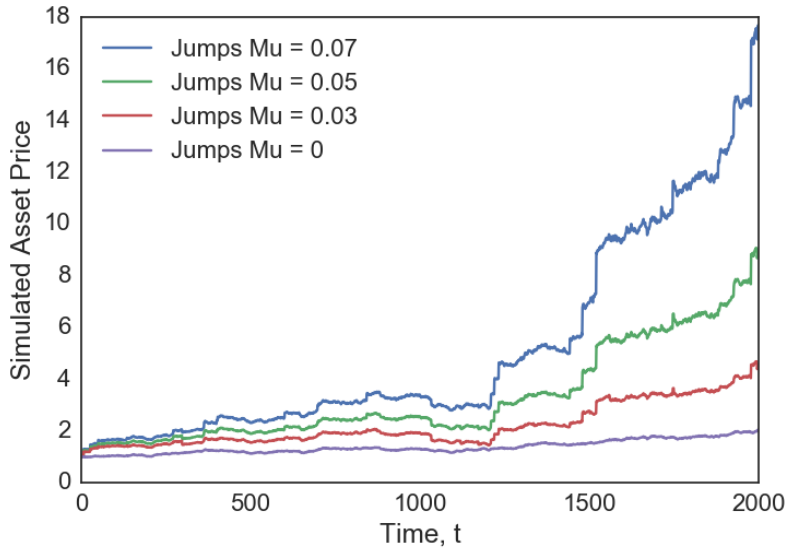


Figure A.5: The effects of changes to μ_{jumps} on the simulated asset price resulting from a Merton Jump-Diffusion returns process. The random seed was set constant at a value of 1234. The other parameters are held constant at $\sigma = 0.1, \mu = 0.05, \lambda = 0.02$, and $\sigma_{jumps} = 0.07$.

APPENDIX B
SENSITIVITY TESTS

B.1. CONVOLUTIONAL MULTIPLE OUTPUT ARCHITECTURE

Variations in μ

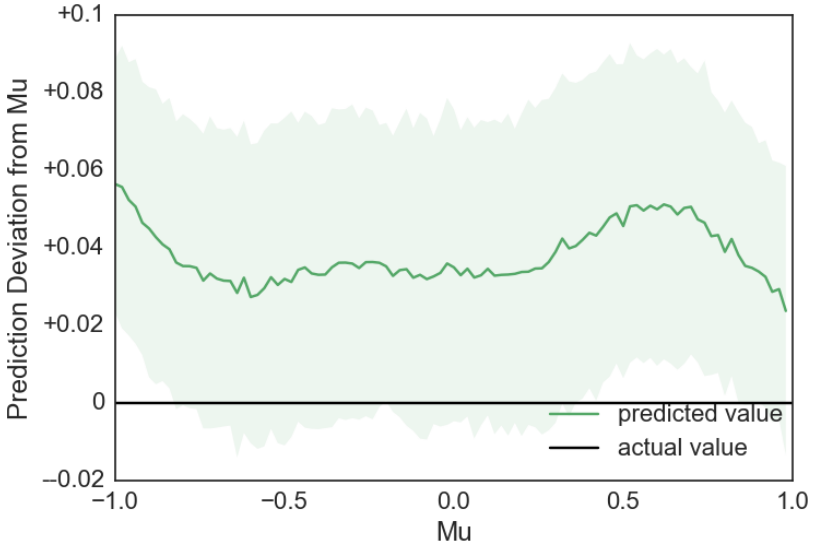
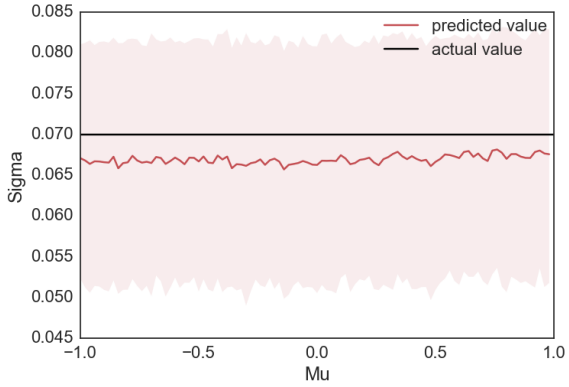
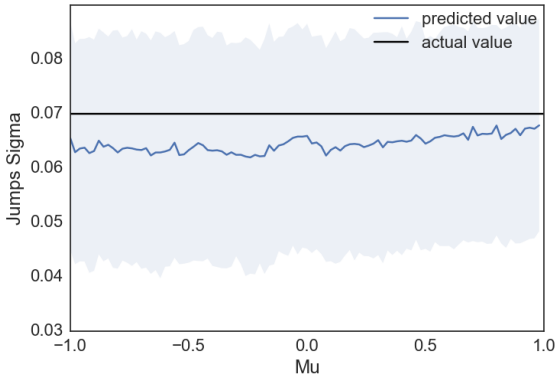


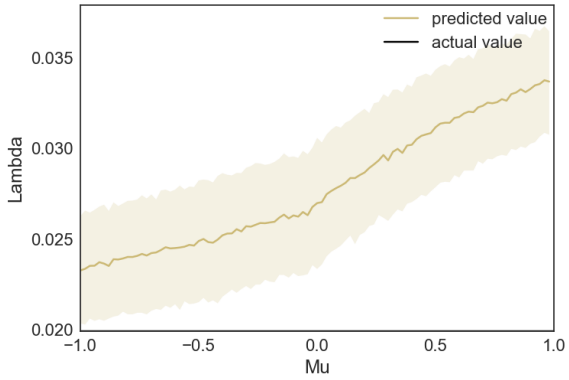
Figure B.1: The deviation (with 68% confidence interval) of the $\hat{\mu}$ parameter estimate from the actual parameter value, μ, for different values of μ. All the other parameters are kept constant as $\sigma = 0.1, \lambda = 0.02, \mu_{jumps} = 0.05$ and $\sigma_{jumps} = 0.07$



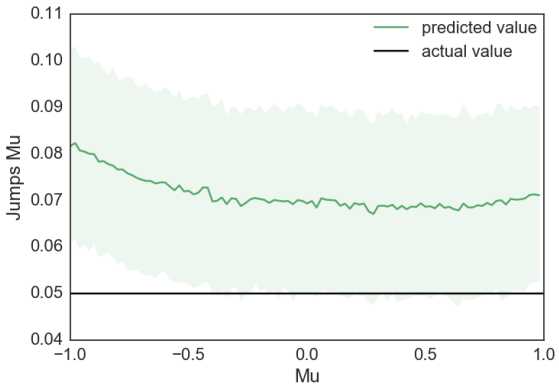
(a) The $\hat{\sigma}$ parameter estimate plotted against the actual parameter value, $\sigma = 0.1$, for different values of μ.



(b) The $\hat{\sigma}_{jumps}$ parameter estimate plotted against the actual parameter value, $\sigma_{jumps} = 0.07$, for different values of μ.



(c) The λ parameter estimate plotted against the actual parameter value, $\lambda = 0.02$, for different values of μ.



(d) The $\hat{\mu}_{jumps}$ parameter estimate plotted against the actual parameter value, $\mu_{jumps} = 0.05$, for different values of μ.

Figure B.2: The parameter estimates (with 68% confidence interval) of $\hat{\sigma}$, $\hat{\lambda}$, $\hat{\mu}_{jumps}$ and $\hat{\sigma}_{jumps}$, plotted against their actual values ($\sigma = 0.1$, $\lambda = 0.02$, $\mu_{jumps} = 0.05$ and $\sigma_{jumps} = 0.07$), while varying the μ parameter in the range $(-1.0, 1.0)$.

Variations in σ

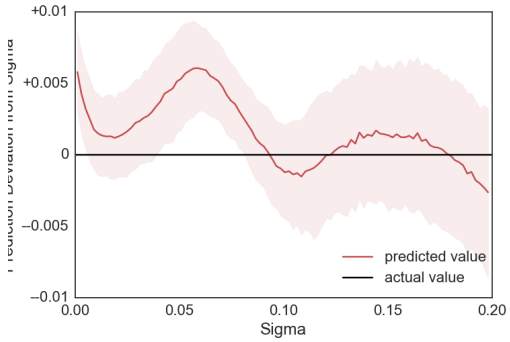
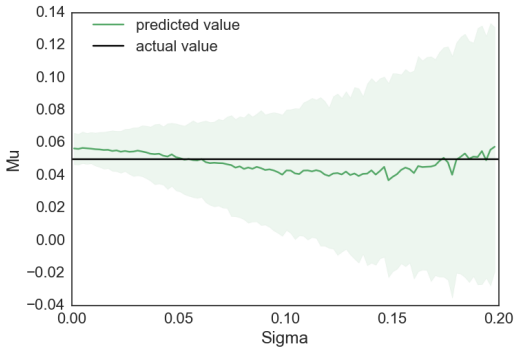
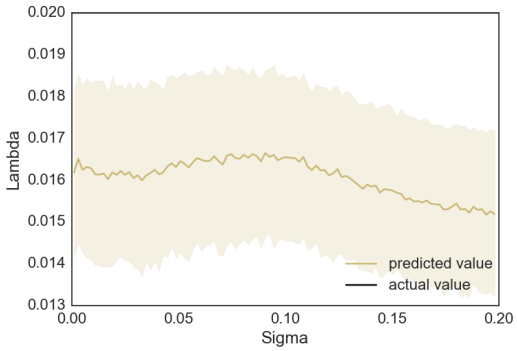


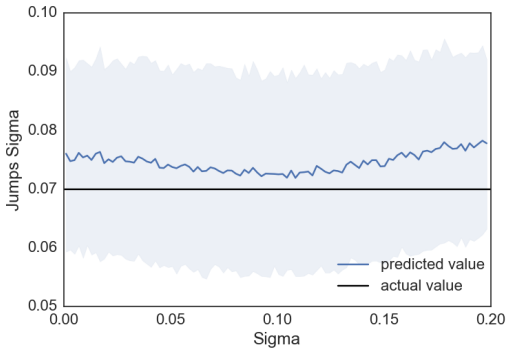
Figure B.3: The deviation (with 68% confidence interval) of the $\hat{\sigma}$ parameter estimate from the actual parameter value, σ, for different values of σ. All the other parameters are kept constant as $\mu = 0.05, \lambda = 0.02, \mu_{jumps} = 0.05$ and $\sigma_{jumps} = 0.07$.



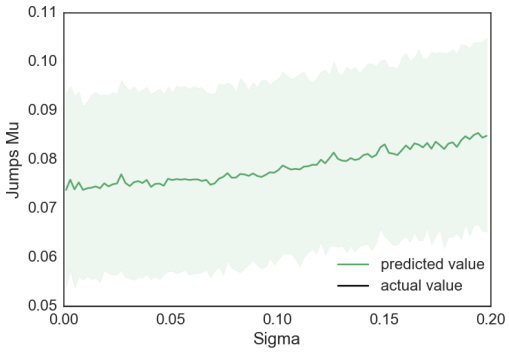
(a) The mean $\hat{\mu}$ parameter estimate plotted against the actual parameter value, $\mu = 0.05$, for different values of σ.



(b) The mean λ parameter estimate plotted against the actual parameter value, $\lambda = 0.02$, for different values of σ.



(c) The mean $\hat{\sigma}_{jumps}$ parameter estimate plotted against the actual parameter value, $\sigma_{jumps} = 0.07$, for different values of σ.



(d) The mean $\hat{\mu}_{jumps}$ parameter estimate plotted against the actual parameter value, $\mu_{jumps} = 0.05$, for different values of σ.

Figure B.4: The mean parameter estimates (with 68% confidence interval) of $\hat{\mu}$, $\hat{\lambda}$, $\hat{\mu}_{jumps}$ and $\hat{\sigma}_{jumps}$, plotted against their actual values ($\mu = 0.05$, $\lambda = 0.02$, $\mu_{jumps} = 0.05$ and $\sigma_{jumps} = 0.07$), while varying the σ parameter in the range $(0, 0.2)$.

Variations in λ

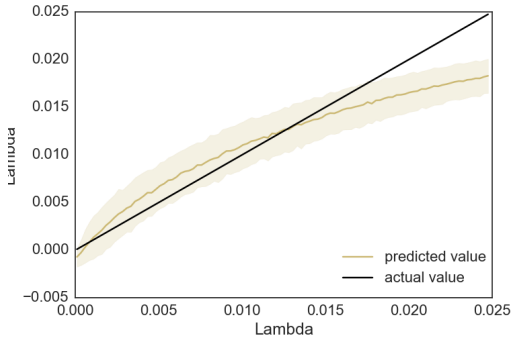
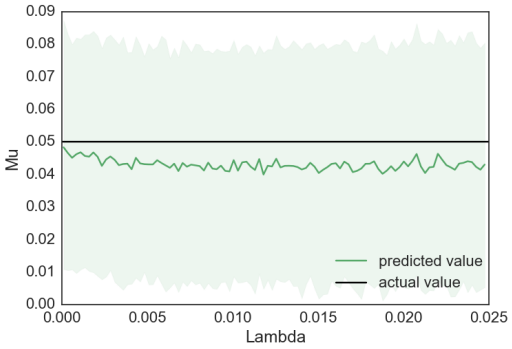
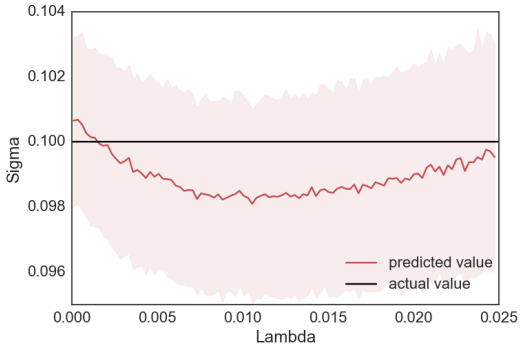


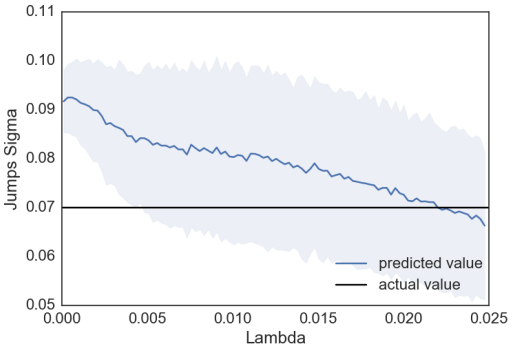
Figure B.5: The mean deviation (with 68% confidence interval) of the $\hat{\lambda}$ parameter estimate from the actual parameter value, λ, for different values of λ. All the other parameters are kept constant as $\mu = 0.05, \sigma = 0.1, \mu_{jumps} = 0.05$ and $\sigma_{jumps} = 0.07$.



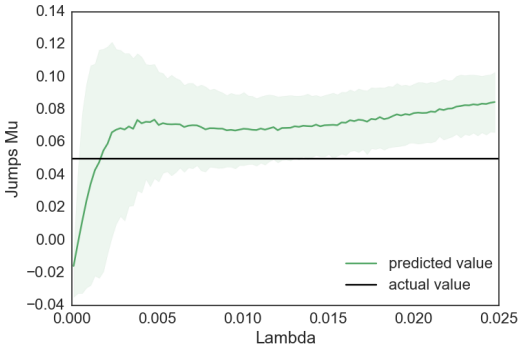
(a) The mean $\hat{\mu}$ parameter estimate plotted against the actual parameter value, $\mu = 0.05$, for different values of λ.



(b) The mean σ parameter estimate plotted against the actual parameter value, $\sigma = 0.1$, for different values of λ.



(c) The mean $\hat{\sigma}_{jumps}$ parameter estimate plotted against the actual parameter value, $\sigma_{jumps} = 0.07$, for different values of λ.



(d) The mean $\hat{\mu}_{jumps}$ parameter estimate plotted against the actual parameter value, $\mu_{jumps} = 0.05$, for different values of λ.

Figure B.6: The mean parameter estimates (with 68% confidence interval) of $\hat{\mu}$, $\hat{\sigma}$, $\hat{\mu}_{jumps}$ and $\hat{\sigma}_{jumps}$, plotted against their actual values ($\mu = 0.05$, $\sigma = 0.1$, $\mu_{jumps} = 0.05$ and $\sigma_{jumps} = 0.07$), while varying the λ parameter in the range $(0, 0.025)$.

Variations in σ_{jumps}

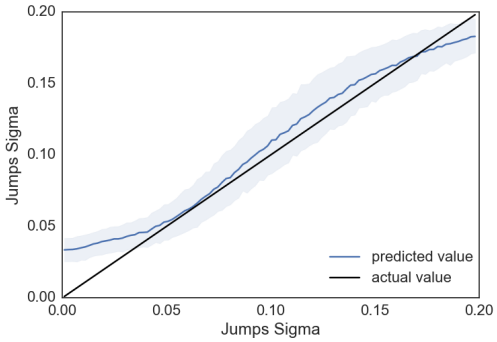
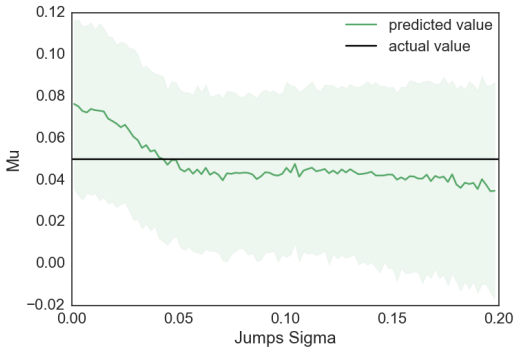
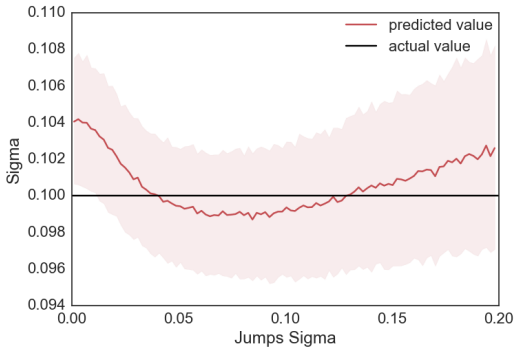


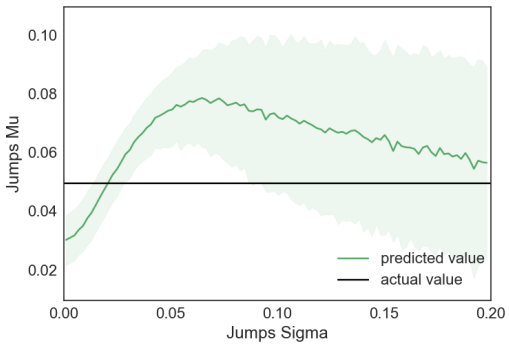
Figure B.7: The mean deviation (with 68% confidence interval) of the $\hat{\sigma}_{jumps}$ parameter estimate from the actual parameter value, σ_{jumps}, for different values of σ_{jumps}. All the other parameters are kept constant as $\mu = 0.05, \sigma = 0.1, \lambda = 0.02$ and $\mu_{jumps} = 0.05$.



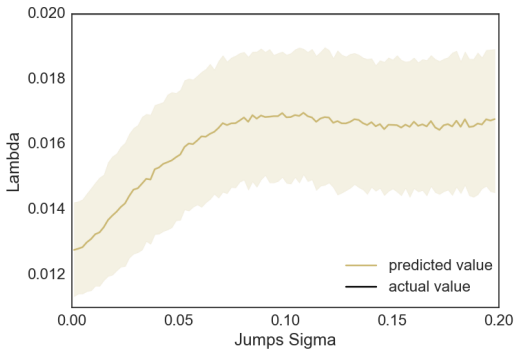
(a) The mean $\hat{\mu}$ parameter estimate plotted against the actual parameter value, $\mu = 0.05$, for different values of σ_{jumps}.



(b) The mean σ parameter estimate plotted against the actual parameter value, $\sigma = 0.1$, for different values of σ_{jumps}.



(c) The mean $\hat{\mu}_{jumps}$ parameter estimate plotted against the actual parameter value, $\mu_{jumps} = 0.05$, for different values of μ_{jumps}.



(d) The mean λ parameter estimate plotted against the actual parameter value, $\lambda = 0.2$, for different values of σ_{jumps}.

Figure B.8: The mean parameter estimates (with 68% confidence interval) of $\hat{\mu}$, $\hat{\sigma}$, $\hat{\mu}_{jumps}$ and λ, plotted against their actual values ($\mu = 0.05$, $\sigma = 0.1$, $\mu_{jumps} = 0.05$ and $\lambda = 0.02$), while varying the σ_{jumps} parameter in the range $(0, 0.2)$.

Variations in μ_{jumps}

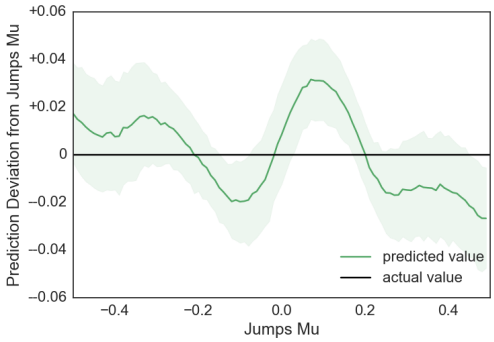
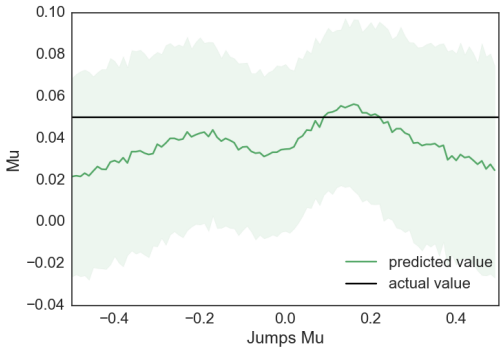
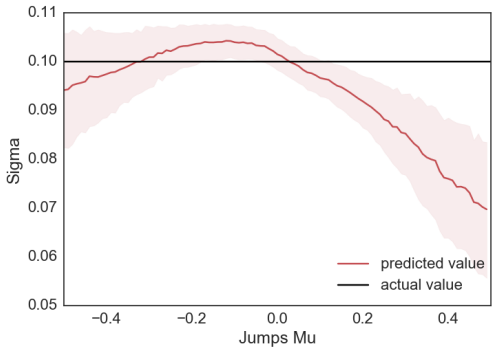


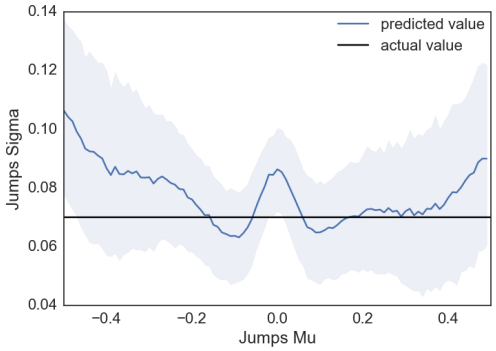
Figure B.9: The mean deviation (with 68% confidence interval) of the $\hat{\mu}_{jumps}$ parameter estimate from the actual parameter value, μ_{jumps}, for different values of μ_{jumps}. All the other parameters are kept constant as $\mu = 0.05, \sigma = 0.1, \lambda = 0.02$ and $\sigma_{jumps} = 0.07$.



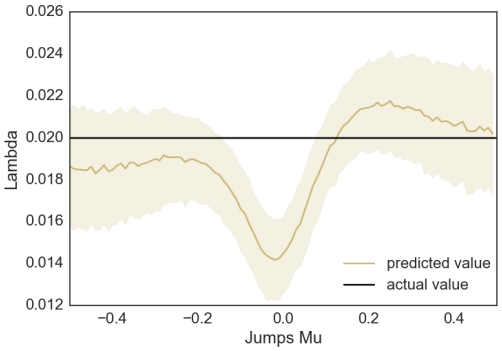
(a) The mean $\hat{\mu}$ parameter estimate plotted against the actual parameter value, $\mu = 0.05$, for different values of μ_{jumps}.



(b) The mean σ parameter estimate plotted against the actual parameter value, $\sigma = 0.1$, for different values of μ_{jumps}.



(c) The mean $\hat{\sigma}_{jumps}$ parameter estimate plotted against the actual parameter value, $\sigma_{jumps} = 0.07$, for different values of μ_{jumps}.



(d) The mean λ parameter estimate plotted against the actual parameter value, $\lambda = 0.2$, for different values of μ_{jumps}.

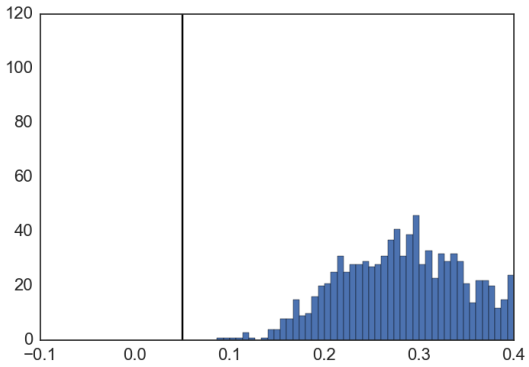
Figure B.10: The mean parameter estimates (with 68% confidence interval) of $\hat{\mu}$, $\hat{\sigma}$, $\hat{\sigma}_{jumps}$ and λ, plotted against their actual values ($\mu = 0.05$, $\sigma = 0.1$, $\lambda = 0.02$ and $\sigma_{jumps} = 0.07$), while varying the μ_{jumps} parameter in the range $(-0.5, 0.5)$.

APPENDIX C
RESULTS

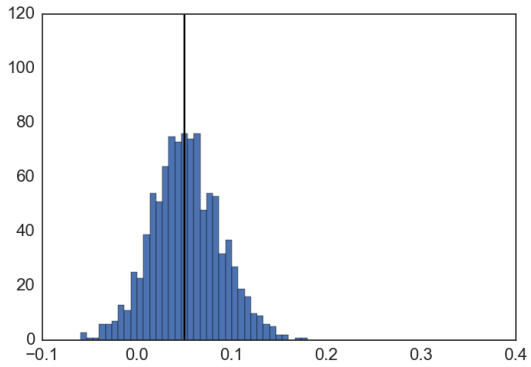
This chapter documents the individual accuracy of the investigated architectures, comparing each with the results obtained using more traditional methods such as MLE, MME and the likelihood profiling method of Honore (1998). 1000 simulated sample paths from a Merton Jump-Diffusion process with parameters, $\mu = 0.05,\ \sigma = 0.1,\ \lambda = 0.02,\ \mu_{jumps} = 0.05$ and $\sigma_{jumps} = 0.07$ were used. The (already trained) NN models were then used to predict these original parameters, given the set of 1000 sample paths. Figures C.1, C.2, C.3, C.4, and C.5 illustrate the parameter estimation results for $\mu, \sigma, \lambda, \mu_{jumps}$ and σ_{jumps} using:

a) MME
b) Likelihood profiling of Honore (1998)
c) MLE,
d) a multiple output convolutional ANN model as defined in section 4.3.1 above,
e) a fully connected NN as defined in section 4.2.1, and
f) a dedicated single output convolutional ANN model as defined in section 4.3.2 above.

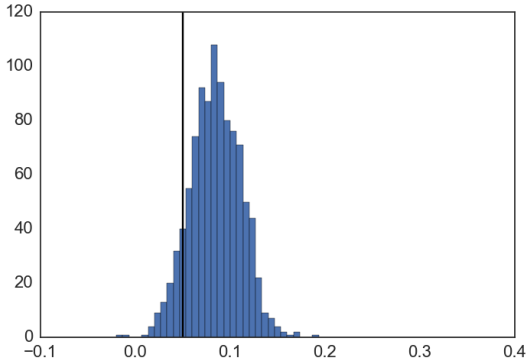
All the parameter estimates produced by MLE, MME and the likelihood profiling approach of Honore (1998) were obtained using the R-code provided by Mongwe (2015).



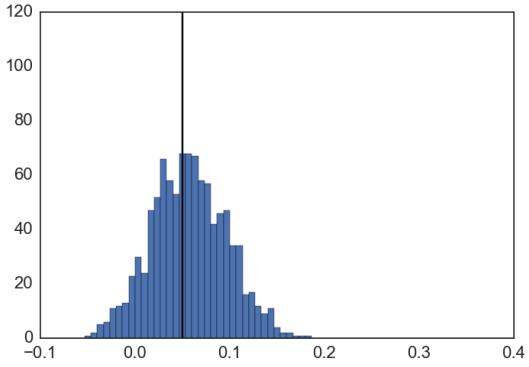
(a) MME



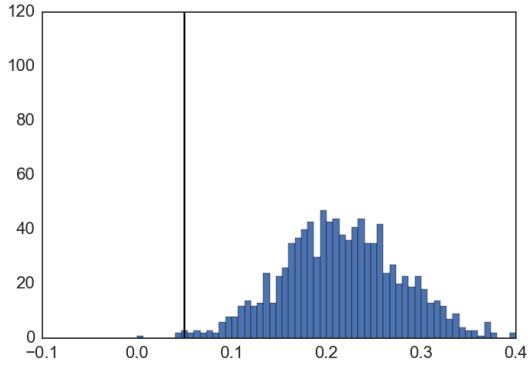
(b) Likelihood Profiling (Honore, 1998)



(c) MLE

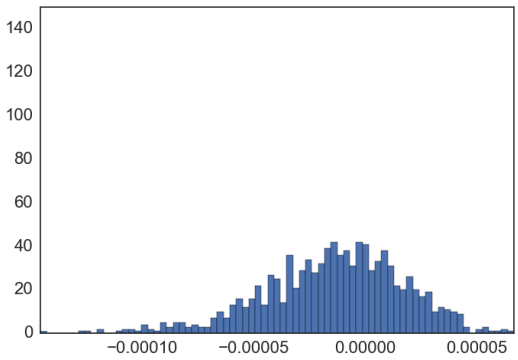


(d) CNN Architecture - Multiple Output - ELU

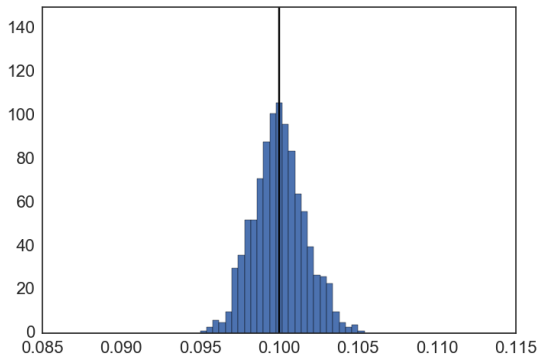


(e) Fully Connected Architecture - Multiple Output - ELU

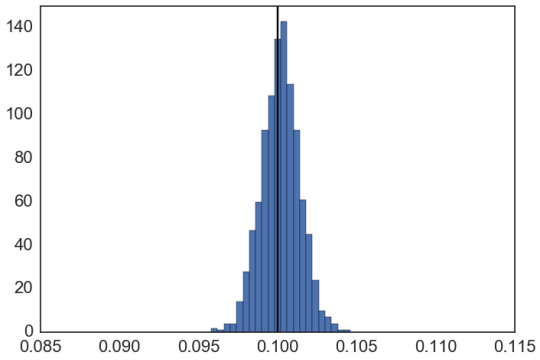
Figure C.1: Various model distributions of the predicted values of μ with true value $0,05$.



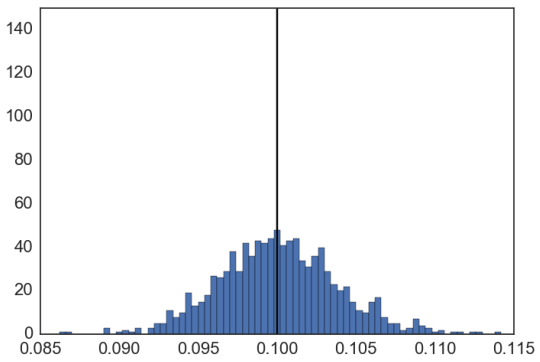
(a) MME



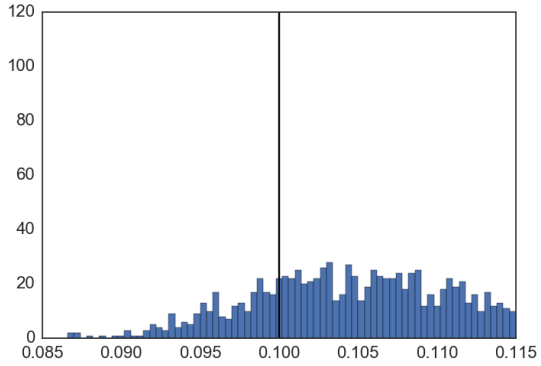
(b) Likelihood Profiling (Honore, 1998)



(c) MLE

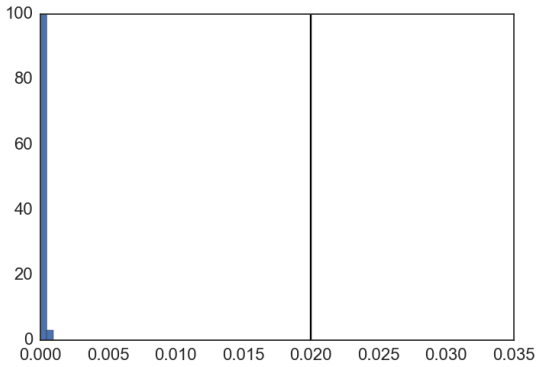


(d) CNN Architecture - Multiple Output - ELU

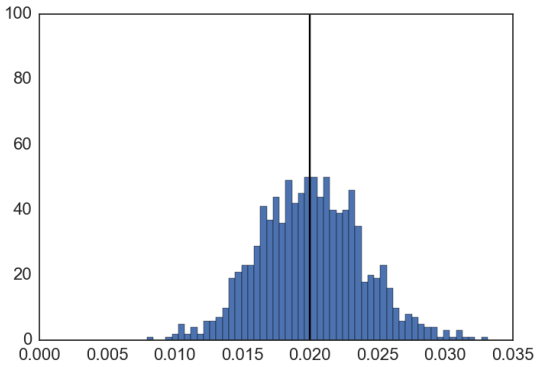


(e) Fully Connected Architecture - Multiple Output - ELU

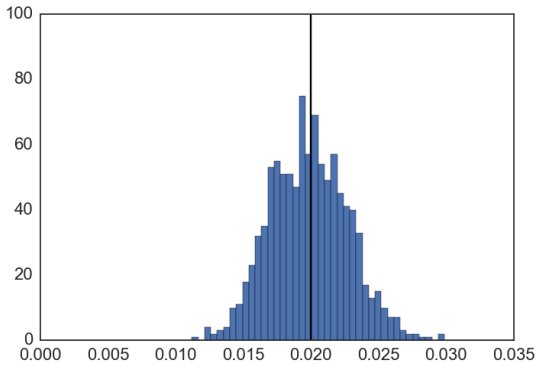
Figure C.2: Various model distributions of the predicted values of σ with true value $0,1$.



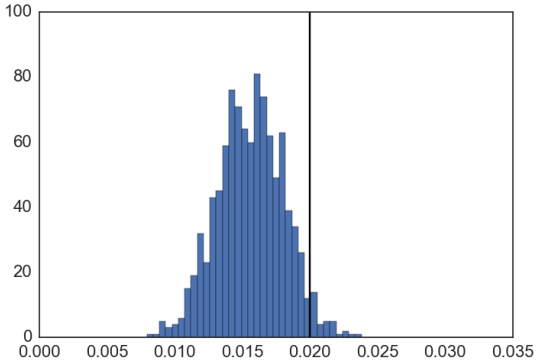
(a) MME



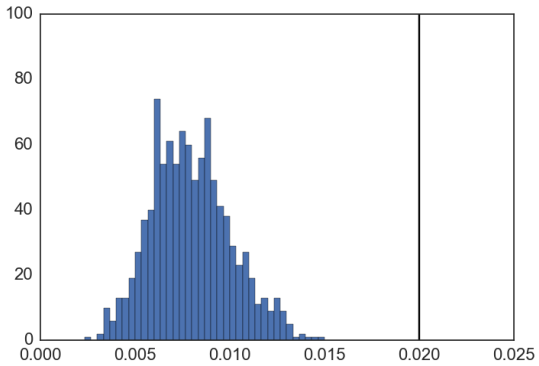
(b) Likelihood Profiling (Honore, 1998)



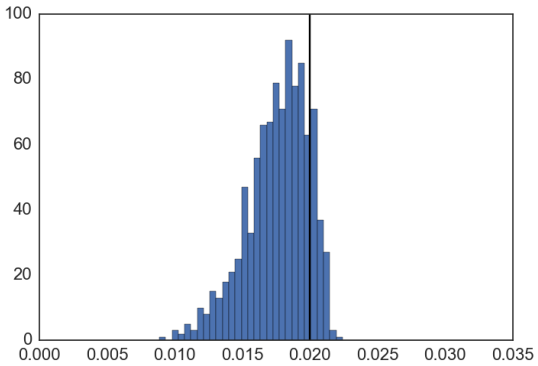
(c) MLE



(d) CNN Architecture - Multiple Output - ELU

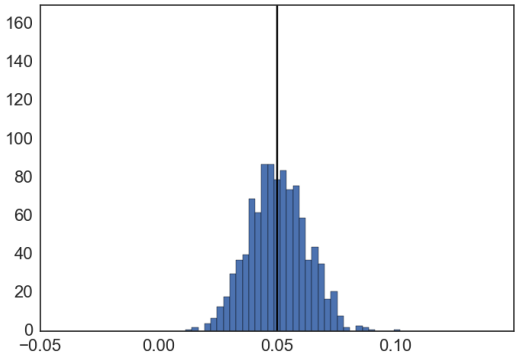


(e) Fully Connected Architecture - Multiple Output - ELU

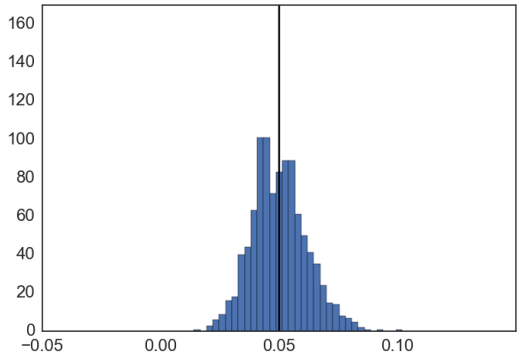


(f) CNN Architecture - Single Output - ELU

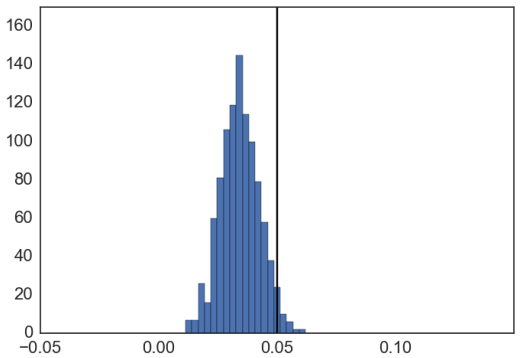
Figure C.3: Various model distributions of the predicted values of λ with true value $0,02$.



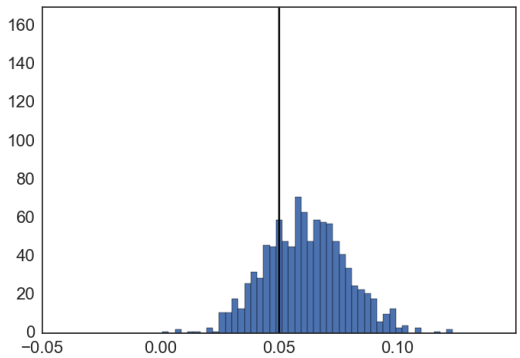
(a) MME



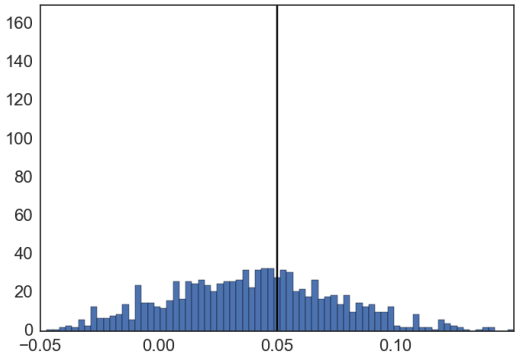
(b) Likelihood Profiling (Honore, 1998)



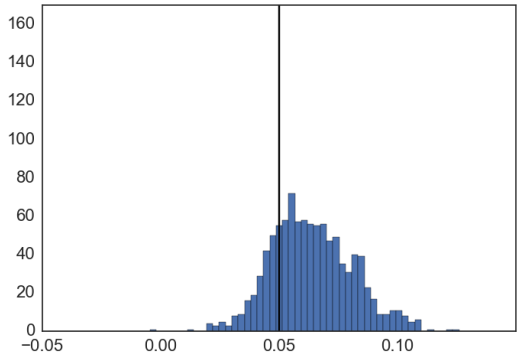
(c) MLE



(d) CNN Architecture - Multiple Output - ELU

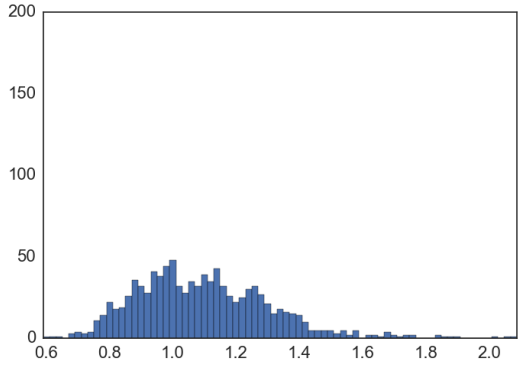


(e) Fully Connected Architecture - Multiple Output - ELU

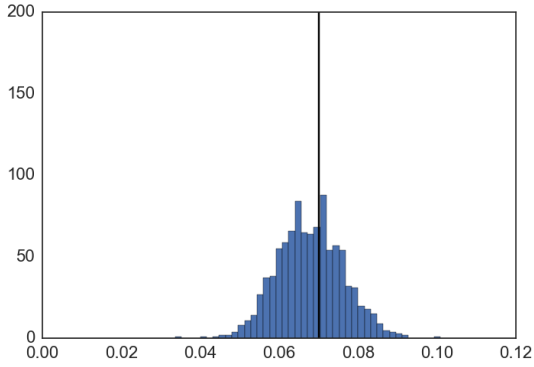


(f) CNN Architecture - Single Output - ReLU

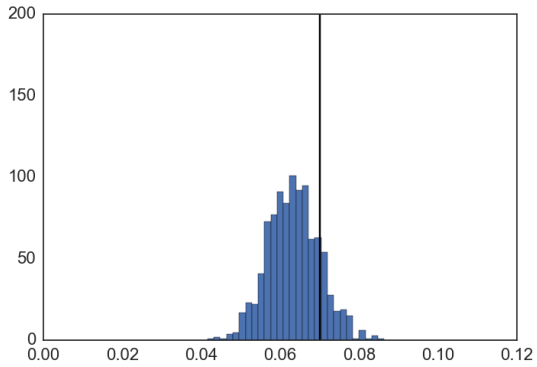
Figure C.4: Various model distributions of the predicted values of μ_{jumps} with true value $0,05$.



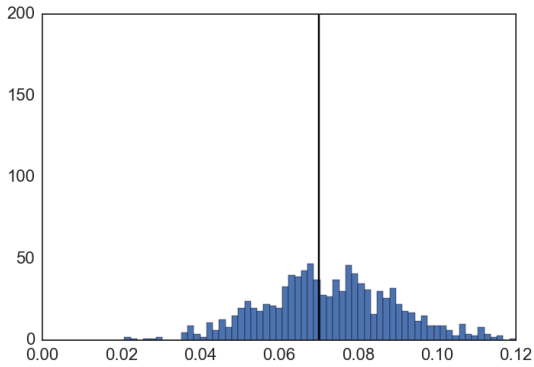
(a) MME



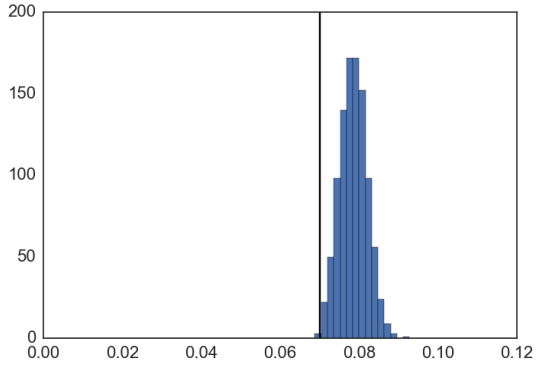
(b) Likelihood Profiling (Honore, 1998)



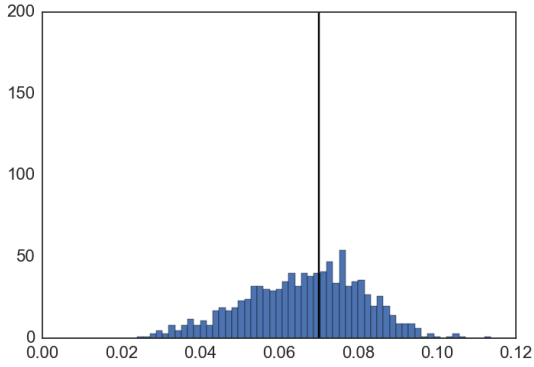
(c) MLE



(d) CNN Architecture - Multiple Output - ELU



(e) Fully Connected Architecture - Multiple Output - ELU



(f) CNN Architecture - Single Output - ReLU

Figure C.5: Various model distributions of the predicted values of σ_{jumps} with true value $0,07$.

APPENDIX D
LIST OF SOFTWARE USED

IPython	A command shell for interactive computing.	Pérez *&* Granger (2007)
Keras	A high-level neural networks API, written in Python running on top of TensorFlow.	Chollet *et al.* (2015)
Matplotlib	Python 2D plotting library.	Hunter (2007)
Mongwe	R implementation of a parameter estimation procedure for the Merton Jump-Diffusion process, using MLE, MME and likelihood profiling (Honore, 1998).	Mongwe (2015)
Numpy	Package for scientific computing with Python.	Oliphant *et al.* (2001–)
Python	Programming language.	Rossum (1995)
Scipy	Python-based open-source software for mathematics, science, and engineering.	Jones *et al.* (2001–)
Seaborn	Python visualization library based on matplotlib.	Waskom *et al.* (2016)
Statsmodels	Python module that provides classes and functions for the estimation of many different statistical models.	Seabold *&* Perktold (2010)
Tensorflow	An open-source software library for Machine Intelligence.	Abadi *et al.* (2015)
Turing Finance	Python implementations of various stochastic processes.	Reid (2015)

APPENDIX E
SOURCE CODE

E.1. NEURAL NETWORK MODELS

```
import tensorflow as tf
import keras
from keras.models import Model, Sequential
from keras.layers import Input, Dense, Flatten, LSTM, MaxPooling2D, Input
from keras.layers.convolutional import Conv2D
from keras.optimizers import Adam
from keras import backend as K

def r2(y_true, y_pred):
    """
    returns the correlation coefficient of y_pred against y_true.

    :param y_true: the true values (independent variable)
    :param y_pred: the predicted values (dependent variable)
    """

    SSE = K.sum(K.square(y_true-y_pred))
    SST = K.sum(K.square(y_true-K.mean(y_true)))

    return 1-SSE/SST

def fullyconnected_multiple_output(activation = 'elu'):
    """
    returns a 9-layer fully connected architecture as a Keras Model, with outputs for all the parameters of the Merton Jump Diffusion stochastic process (sigma, mu, jump_sigma, jump_mu, lambda).

    .. note:: be sure to order the array of desired output parameters as follows: sigma, mu, jump_sigma, jump_mu, lambda. Each their own array of desired values for every simulated sample path.
    """

    input_1 = Input(batch_shape = (None, 60))

    layer1 = Dense(2048, activation=activation)(input_1)
    layer2 = Dense(1024, activation=activation)(layer1)

    layer3 = Dense(512, activation=activation)(layer2)
    layer4 = Dense(256, activation=activation)(layer3)

    layer5 = Dense(128, activation=activation)(layer4)
    layer6 = Dense(64, activation=activation)(layer5)

    layer7 = Dense(32, activation=activation)(layer5)
    last_layer = Dense(16, activation=activation)(layer7)

```

```
43      output1 = Dense(1, name="sigma")(last_layer)
44      output2 = Dense(1, name="mu")(last_layer)
45      output3 = Dense(1, name="jump_sigma")(last_layer)
46      output4 = Dense(1, name="jump_mu")(last_layer)
47      output5 = Dense(1, name="lambda")(last_layer)
48
49      feedforward = Model(input = input_1, output=[output1, output2, output3, output4,
        output5])
50
51      feedforward.compile(loss='mean_squared_error', optimizer='adam', metrics=[r2, '
        mean_absolute_percentage_error'])
52
53      return feedforward
54
55  def fullyconnected_single_output(activation = 'elu'):
56      """
57      returns a 9-layer fully connected architecture as a Keras Model, with outputs for
        all the parameters of the Merton Jump Diffusion stochastic process (sigma, mu,
        jump_sigma, jump_mu, lambda).
58
59      .. note:: be sure to order the array of desired output parameters as follows:
        sigma, mu, jump_sigma, jump_mu, lambda.  Each their own array of desired values for
         every simulated sample path.
60      """
61      feedforward = Sequential()
62
63      feedforward.add(Dense(2048, activation=activation, input_dim = 60))
64      feedforward.add(Dense(1024, activation=activation))
65
66      feedforward.add(Dense(512, activation=activation))
67      feedforward.add(Dense(256, activation=activation))
68
69      feedforward.add(Dense(128, activation=activation))
70      feedforward.add(Dense(64, activation=activation))
71
72      feedforward.add(Dense(32, activation=activation))
73      feedforward.add(Dense(16, activation=activation))
74
75      feedforward.add(Dense(1))
76
77      feedforward.compile(loss='mean_squared_error', optimizer='adam', metrics=[r2, '
        mean_absolute_percentage_error'])
78
79      return feedforward
80
81
82  def covnet_multiple_8_layers(activation='elu'):
83      """
84      returns an 8-layer convolutional architecture as a Keras Model, with outputs for
        all the parameters of the Merton Jump Diffusion stochastic process (sigma, mu,
        jump_sigma, jump_mu, lambda).
85
86      .. note:: be sure to order the array of desired output parameters as follows:
        sigma, mu, jump_sigma, jump_mu, lambda.  Each their own array of desired values for
```

```
        every simulated sample path.
87      """
88
89      input_1 = Input(shape = (40, 50, 1))
90
91      layer1 = Conv2D(32, (12, 12), padding='same', activation=activation)(input_1)
92      layer2 = Conv2D(32, (12, 12), padding='same', activation=activation)(layer1)
93      layer3 = MaxPooling2D(pool_size=(2,2))(layer2)
94
95      layer4 = Conv2D(64, (6, 6), padding='same', activation=activation)(layer3)
96      layer5 = Conv2D(64, (6, 6), padding='same', activation=activation)(layer4)
97      layer6 = MaxPooling2D(pool_size=(2,2))(layer5)
98
99      layer7 = Conv2D(128, (3, 3), padding='same', activation=activation)(layer6)
100     layer8 = Conv2D(128, (3, 3), padding='same', activation=activation)(layer7)
101     layer9 = MaxPooling2D(pool_size=(2,2))(layer8)
102
103     flatten = Flatten()(layer9)
104     last_layer = Dense(256, activation=activation)(flatten)
105     output1 = Dense(1, name="sigma")(last_layer)
106     output2 = Dense(1, name="mu")(last_layer)
107     output3 = Dense(1, name="jump_sigma")(last_layer)
108     output4 = Dense(1, name="jump_mu")(last_layer)
109     output5 = Dense(1, name="lambda")(last_layer)
110
111     convnet_mo_elu = Model(input = input_1, output=[output1, output2, output3, output4
        , output5])
112
113     convnet_mo_elu.compile('adam', 'mean_squared_error', metrics=['
        mean_absolute_percentage_error', r2])
114
115     return convnet_mo_elu
116
117 def covnet_single_ReLUs_6_layers():
118     """
119     returns a 6-layer convolutional architecture (with ReLU activation units) as a
        Keras Model, with outputs for only a single parameter.
120     """
121
122     convnet = Sequential()
123     convnet.add(Conv2D(32, (12, 12), input_shape=(40, 50, 1), padding='same',
        activation='relu'))
124     convnet.add(Conv2D(32, (12, 12), padding='same', activation='relu'))
125     convnet.add(MaxPooling2D(pool_size=(2,2)))
126
127     convnet.add(Conv2D(64, (6, 6), padding='same', activation='relu'))
128     convnet.add(Conv2D(64, (6, 6), padding='same', activation='relu'))
129     convnet.add(MaxPooling2D(pool_size=(2,2)))
130
131     convnet.add(Flatten())
132     convnet.add(Dense(512, activation='relu'))
133     convnet.add(Dense(1))
134
135     convnet.compile('adam', 'mean_squared_error', metrics=['
```

```
        mean_absolute_percentage_error', r2])
136
137     return convnet
```

E.2. MERTON JUMP-DIFFUSION STOCHASTIC PROCESS SIMULATION

The source code in this section (E.2) was provided by Reid (2015).

```
#
    ##############################################################################################

#
# CREDIT: THIS CODE WAS PROVIDED BY STUART G. REID, 2015 (BIBLIOGRAPHY).
#
# THE FULL VERSION CAN BE OBTAINED FROM:
# http://www.turingfinance.com/random-walks-down-wall-street-stochastic-processes-in-
    python/
#
#
    ##############################################################################################


import numpy as np
from numpy import random as nrand
import math
import random

class ModelParameters:
    """
    Encapsulates model parameters
    """

    def __init__(self,
                 all_time, all_delta, all_sigma, gbm_mu,
                 jumps_lamda=0.0, jumps_sigma=0.0, jumps_mu=0.0):

        # This is the amount of time to simulate for
        self.all_time = all_time

        # This is the delta, the rate of time e.g. 1/252 = daily, 1/12 = monthly
        self.all_delta = all_delta

        # This is the volatility of the stochastic processes
        self.all_sigma = all_sigma

        # This is the annual drift factor for geometric brownian motion
        self.gbm_mu = gbm_mu

        # This is the probability of a jump happening at each point in time
        self.lamda = jumps_lamda

        # This is the volatility of the jump size
        self.jumps_sigma = jumps_sigma

        # This is the average jump size
        self.jumps_mu = jumps_mu


```

```
46 def random_model_params(constraint_all_sigma = (0.001,0.2), constraint_gbm_mu = (-1,1)
       , constraint_jumps_lamda = (0.0001,0.025), constraint_jumps_sigma = (0.001, 0.2),
       constraint_jumps_mu = (-0.5,0.5)):
47     """
48     This method returns a random set of uniformly drawn ModelParameters, within
       certain constraints.
49     :param constraint_all_sigma: the lower and upper constraints on the uniformly
       drawn sigma parameter value
50     :param constraint_gbm_mu: the lower and upper constraints on the uniformly drawn
       mu parameter value
51     :param constraint_jumps_lamda: the lower and upper constraints on the uniformly
       drawn lambda parameter value
52     :param constraint_jumps_sigma: the lower and upper constraints on the uniformly
       drawn jumps_sigma parameter value
53     :param constraint_jumps_mu: the lower and upper constraints on the uniformly drawn
        jumps_mu parameter value
54     :return: random set of ModelParameters
55     """
56     return ModelParameters(
57         # Fixed Parameters
58         all_time=2000,
59         all_delta=0.00396825396,
60 
61         # Random Parameters
62         all_sigma = nrand.uniform(constraint_all_sigma[0], constraint_all_sigma[1]),
63         gbm_mu = nrand.uniform(constraint_gbm_mu[0], constraint_gbm_mu[1]),
64         jumps_lamda=nrand.uniform(constraint_jumps_lamda[0], constraint_jumps_lamda
       [1]),
65         jumps_sigma=nrand.uniform(constraint_jumps_sigma[0], constraint_jumps_sigma
       [1]),
66         jumps_mu=nrand.uniform(constraint_jumps_mu[0], constraint_jumps_mu[1]),
67     )
68 
69 def brownian_motion_log_returns(param):
70     """
71     This method returns a Wiener process. The Wiener process is also called Brownian
       motion. For more information
72     about the Wiener process check out the Wikipedia page: http://en.wikipedia.org/
       wiki/Wiener_process
73     :param param: the model parameters object
74     :return: brownian motion log returns
75     """
76     sqrt_delta_sigma = math.sqrt(param.all_delta) * param.all_sigma
77     return nrand.normal(loc=0, scale=sqrt_delta_sigma, size=param.all_time)
78 
79 def geometric_brownian_motion_log_returns(param):
80     """
81     This method constructs a sequence of log returns which, when exponentiated,
       produce a random Geometric Brownian
82     Motion (GBM). GBM is the stochastic process underlying the Black Scholes options
       pricing formula.
83     :param param: model parameters object
84     :return: returns the log returns of a geometric brownian motion process
85     """
```

```
86      assert isinstance(param, ModelParameters)
87      wiener_process = np.array(brownian_motion_log_returns(param))
88      sigma_pow_mu_delta = (param.gbm_mu - 0.5 * math.pow(param.all_sigma, 2.0)) * param
        .all_delta
89      return wiener_process + sigma_pow_mu_delta
90
91  def jump_diffusion_process(param):
92      """
93      This method produces a sequence of Jump Sizes which represent a jump diffusion
        process. These jumps are combined
94      with a geometric brownian motion (log returns) to produce the Merton model.
95      :param param: the model parameters object
96      :return: jump sizes for each point in time (mostly zeroes if jumps are infrequent)
97      """
98      assert isinstance(param, ModelParameters)
99      s_n = time = 0
100     small_lamda = -(1.0 / param.lamda)
101     jump_sizes = np.zeros(param.all_time)
102     while s_n < param.all_time:
103         s_n += small_lamda * math.log(random.uniform(0, 1))
104         for j in range(0, param.all_time):
105             if time * param.all_delta <= s_n * param.all_delta <= (j + 1) * param.
        all_delta:
106                 jump_sizes[j] += random.normalvariate(param.jumps_mu, param.
        jumps_sigma)
107                 break
108         time += 1
109     return jump_sizes
110
111 def geometric_brownian_motion_jump_diffusion_log_returns(param):
112     """
113     This method constructs combines a geometric brownian motion process (log returns)
        with a jump diffusion process
114     (log returns) to produce a sequence of gbm jump returns.
115     :param param: model parameters object
116     :return: returns a GBM process with jumps in it
117     """
118     assert isinstance(param, ModelParameters)
119     jump_diffusion = jump_diffusion_process(param)
120     geometric_brownian_motion = geometric_brownian_motion_log_returns(param)
121     return np.add(jump_diffusion, geometric_brownian_motion)
```