MULTI-DIMENSIONAL EVOLUTIONARY ALGORITHMS FOR TRAINING NEURAL NETWORKS WITH HETEROGENEOUS ARCHITECTURES IN PRICING OF AMERICAN STYLE OPTIONS

by

Andrew Clinton Sharp

A Dissertation
Submitted to the
Graduate Faculty
of
George Mason University
In Partial fulfillment of
The Requirements for the Degree
of
Doctor of Philosophy
Computational Science and Informatics

Committee:

_____ Dr. James Gentle, Dissertation Director

_____ Dr. Jason Kinser, Committee Member

_____ Dr. Timothy Sauer, Committee Member

_____ Dr. Igor Griva, Committee Member

_____ Dr. Kevin Curtin, Department Chair

_____ Dr. Peggy Agouris, Dean, College of Science

Date: _____ Summer 2016
George Mason University
Fairfax, VA

Multi-Dimensional Evolutionary Algorithms for Training Neural Networks with Heterogeneous Architectures in Pricing of American Style Options

A dissertation submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy at George Mason University

By

Andrew Clinton Sharp
Master of Science
Texas A&M University, 2011
Bachelor of Science
Texas A&M University, 2009

Director: Dr. James Gentle, Professor
Department of Computational Data Science

Summer 2016
George Mason University
Fairfax, VA

Dedication

I would like to dedicate this to my parents Christopher Robert Sharp and Lilia Garcia-Sharp without whom this work would not be possible. Their guidance and encouragement has given me the strength and character to pursue and complete my doctorate.

Acknowledgments

I would like to thank my professors and committee members, in particular Dr. James Gentle, for their guidance and patience without which this would not be possible.

Table of Contents

List of Tables

List of Figures

Abstract

MULTI-DIMENSIONAL EVOLUTIONARY ALGORITHMS FOR TRAINING NEURAL NETWORKS WITH HETEROGENEOUS ARCHITECTURES IN PRICING OF AMERICAN STYLE OPTIONS

Andrew Clinton Sharp, PhD

George Mason University, 2016

Dissertation Director: Dr. James Gentle

A common problem in training artificial neural networks is determining the optimal network architecture for the particular problem. An ontology is presented that allows for the generalization of the multi-dimensional differential evolution (MD-DE) algorithm in training an ensemble of neural networks with heterogeneous architectures. This generalized algorithm is referred to as the multi-dimensional evolutionary algorithm (MD-EA) since it provides a framework for performing any of the evolutionary optimization algorithms such as differential evolution or genetic algorithms on neural networks with heterogeneous architectures.

The original MD-DE algorithm and three variants are compared using a set of six representative functions while varying several parameters such as numbers of hidden nodes and size of the training population. These representative functions contain a wide array of potential function behaviors including polynomial, trigonometric, discontinuous, non-differentiable, and a finite singularity. A random sample from a Poisson distribution is used to construct an ensemble of single and multi-layer neural networks with heterogeneous architectures that are then used to evaluate the proposed algorithms' performance.

An artificial neural network (ANN) training algorithm is presented that leverages any of the f-divergences as well as test statistics such as the Kolmogorov-Smirnov (K-S) or Anderson-Darling (A-D) as potential objective functions. These objective functions are chosen for online training of ANNs with a single observation at each time interval. The K-S statistic is further explored as a non-differentiable objective function for ANN training in online learning. This statistic is used to ensure convergence in the distributional sense for the learning of the conditional probability distribution of an input time series. Using the K-S statistic allows for the assumption of stationarity in the underlying model to be removed. Least squares Monte Carlo (LSM) is used to demonstrate the networks' ability to learn the appropriate dynamics of simulated arithmetic and geometric Brownian motion (GBM) processes.

The LSM algorithm estimates the American style option price of an asset following a known process. The known process is replaced by the learned conditional probability distribution output from the neural networks and is used to generate many sample asset price paths. It is shown that the variance in the estimated LSM price is due to the variance of the estimated parameters using a finite number of observations from a single price path realization of the underlying stochastic process.

Chapter 1: Introduction

Using artificial neural networks to learn a particular behavior of interest is typically accompanied by several issues related to the architecture of the networks, type and properties of the objective function, and determining if the underlying process to be modeled is constant or dynamic. A novel algorithm is presented that extends an existing multidimensional differential evolution algorithm to a larger class of optimization algorithms. The nature of this algorithm allows for the definition of an overall complexity parameter that sets the number of parameters to be used within a neural network. This parameter may remain constant or may be continuously controlled during run time.

By controlling the expected value of this complexity parameter instead of setting a specific neural network architecture, the trained networks are allowed to add or remove nodes and layers implicitly in response to the problem. Combining this controllable parameter for model complexity with a suitable optimization algorithm along with a growing and pruning algorithm allows for a heterogeneous population of neural networks to train and adapt to a potentially changing underlying process. This allows the user to change error functions, merge in new hidden node types, add or remove new models to the ensemble, and continuously change the overall complexity of the problem without having to restart the models' learning. A pre-run exploration phase to determine an optimal network architecture is also no longer necessary.

The models are in particular in this dissertation used to learn the conditional probability distribution of an underlying asset price process in an online learning fashion. Data is presented to the model as it is observed and only used for a time window of a particular length (this time window need not be constant). Applications for this algorithm apply to any online learning problem where the underlying process may not be constant such as for financial assets, animal migrations, and real time controls for industrial processes. To

verify the learning of the ANNs, least squares Monte Carlo is used to estimate American style option prices based upon the final learned ANN models with respect to the underlying process.

1.1 Objectives

Taking advantage of the advances in the fields of machine learning and optimization algorithms, a method is proposed that seeks to learn the underlying model of a process' dynamics. The first objective is to improve and expand upon an existing optimization algorithm for training heterogeneous single layer neural networks. In particular the objective is to remedy inefficiencies in the existing optimization algorithm that prevent training of all hidden nodes in certain populations of networks. Secondly, this algorithm will be extended to apply to the more general class of all evolutionary algorithm type global optimization methods while simultaneously being expanded to include multi-layer neural networks.

Since the optimization algorithm is extended to a larger class of algorithms, an ontology needs to be devised to describe the different algorithm variants. These variants provide different mechanisms for normalizing neural networks with heterogeneous architectures prior to the application of any of the evolutionary algorithms for optimization.

The last objective is to show how various objective function such as f-divergences and test statistics such as the Kolmogorov-Smirnov (K-S) and Anderson-Darling (A-D) test statistics may be used to train and evaluate a model's performance when learning a process' conditional probability distribution without the assumption for stationarity. The least squares Monte Carlo (LSM) algorithm is used to estimate American style option prices for simulated asset prices based upon the learned process by an ensemble of neural networks. Many realizations of this problem are run to measure the performance of the algorithm for two different processes.

1.2 Motivation

In financial derivative markets, consistency is the property that the financial derivatives are priced according to the dynamics of the underlying financial assets. When derivative prices differ from the fair market value according to a no-arbitrage principle the market is said to be inconsistent. Consistency ensures that no arbitrage exists since the options are perfectly priced. These price differences may be attributed to inefficiencies in the overall market or irrational actors forcing prices higher or lower than the optimal price. Therefore, determining the "correct" derivative price is essential even though the true dynamics of the underlying financial assets is unknown.

When modeling financial assets, it is well known that the underlying stock process is not constant and is continuously changing in response to market behavior. It is difficult to assess a neural network's performance in an online training setup with a dynamic underlying process. Secondly, even if an assumption can be made that the process is constant there is a time consuming exploration phase that involves testing various model architectures to determine which is optimal. A neural network training algorithm is needed that does not require the underlying dynamics to be stationary.

Prior to training an artificial neural network to learn the conditional probability distribution of the underlying asset prices, it is necessary to define an appropriate metric to measure a network's performance during training. This metric must be capable of generating a feedback signal of network performance based upon a network's output estimate of the true conditional probability distribution of the underlying asset with respect to a single observation of asset prices at each time interval.

Also, a statistic is needed that is able to measure a model's ability to learn and respond to not only a possibly changing underlying process, but to respond to a conditional distribution. Tests such as Kolmogorov-Smirnov and Anderson-Darling provide statistics to evaluate the difference between two different distributions. An optimization algorithm is required that is capable of handling these non-differentiable test statistics as well as potentially changing network architectures during training.

1.3 Innovations and Contributions to Field

In this dissertation, I will develop a novel method for learning a process' conditional probability distribution using a heterogeneous neural network trained via a global optimization algorithm.

1. An ontology is introduced to describe the various transformations and relationships between two neural networks.

2. A continuously tunable parameter for the complexity of an ensemble of neural networks is introduced in this dissertation. This parameter may be changed during training without restarting the models' training and may even be connected to an external control algorithm.

3. Improvement and extension of an existing multi-dimensional differential evolution algorithm for training heterogeneous single-layer neural networks have been made to allow for efficient training of an arbitrary collection of neural networks with heterogeneous architectures. These same algorithms may be adapted to work for genetic algorithms or if the underlying process is stationary, then particle swarm optimization as well.

4. Three variations of the MD-DE algorithm are presented in this dissertation and compared against the original MD-DE algorithm. Two of these variations were created to rectify theoretical pitfalls of the original algorithm.

5. F-divergences and test statistics such as Kolmogorov-Smirnov and Anderson-Darling test statistics are studied and evaluated in this dissertation as valid objective functions for learning of conditional probability distributions in an online learning setting. In particular, the Kolmogorov-Smirnov test statistic is used to evaluate a model's learning in an online learning setting.

6. The LSM algorithm is used to estimate American style option prices based upon

forecasted paths from the learned ensemble of neural networks with heterogeneous architectures. It is shown that a large variance in the output price is due to the variance induced from estimating parameters from a finite sample of a single realization.

7. ANNs are shown in this dissertation to be capable of learning asset price dynamics from a single realization of two different simulated processes. Training performance of these ANNs are analyzed and found to be within theoretical limits.

Chapter 2: Background and Literature Review

Prior to the presentation of the unique works in this dissertation, it is necessary to provide introduction of basic material background. Knowledge of this background material is required to understand and appreciate the works presented later in this dissertation.

2.1 Evaluating Market Consistency

Evaluation of market consistency is done by using the prices of either the financial assets or derivatives to construct a pricing model of the other. Previous work has been done by Bates (1995) and Bakshi et al. (1997) in evaluating market consistency by assuming a parametric model for the dynamics of the underlying asset prices and calculating the associated European option prices. This is usually done via parameter estimation using a time series of the underlying asset prices. I will focus on estimating the underlying asset dynamics using very few assumptions and then checking consistency of the derivative prices. I expand upon these works in two ways. First, I will take into account the difference induced by the use of American style options which was not done in previous literature. Secondly, I will use artificial neural networks (ANNs) to learn the dynamics of the underlying price process based upon the process' time series and use these dynamics to price American style options. Work has been done in using a time series of the price process to estimate the associated option prices such as that done by Chernov et al. (2003) and Makridakis (1989) with good results.

There are many ways that ANNs can be used to generate predictions of asset prices. The pricing data may be ingested directly into the ANN as used in Panigrahi et al. (2013), Donaldson and Kamstra (1996), and Dhahri and Alimi (2006) or used to estimate various parameters that are then ingested into the ANN as used by Dindar and Marwala (2004),

Bennell and Sutcliffe (2004), and Chernov et al. (2003) or use the ANN to transform an input distribution as used by Magdon-Ismail and Atiya (2002). I am proposing to use an ANN to learn the underlying price process dynamics using an input time series. This will allow for a non-parametric learning of the asset's conditional probability distribution. Having an ANN that has learned the asset's conditional probability distribution allows for the generation of Monte Carlo paths of asset prices as needed in the calculation of American style option price estimates as used by Tompaidis and Yang (2014) and Longstaff and Schwartz (2012).

Bennell and Sutcliffe (2004), Huang et al. (2005), Bortman and Aladjem (2009), and Dhahri and Alimi (2006) point out some of the difficulties in using ANNs to model time series data such as slow convergence, local minima, differentiability of output, and number of hidden nodes. While the back-propagation method proposed by Rumelhart et al. (1986) is easy to implement, it is susceptible to slow convergence and convergence to local minima. Radial basis functions (RBFs) have been shown by Wu and Wilamowski (2013) and Xie et al. (2011) to converge faster and more robustly than typical sigmoid nodes. Previous work has also been done by Elanayar V. T. and Shin (1994) in using RBFs to model stochastic dynamic systems. Optimization algorithms such as Nelder-Mead simplex optimization algorithm and meta-heuristic optimization algorithms such as particle swarm optimization (PSO) or differential evolution (DE) are capable of training almost any type of ANN. Meta-heuristic optimization algorithms in particular alleviate the issues associated with output differentiability and are more robust to finding global minima based on the simultaneous evaluation of many trial solutions and introducing a random exploration component within the algorithm. Differential evolution will be used in this work based upon the work of Vesterstrøm and Thomsen (2004) that shows that DE is more robust than other meta-heuristic optimization algorithms in avoiding local minima. By combining DE in the optimization of weights in an RBF ANN with a growing and pruning algorithm developed by Huang et al. (2005) and Bortman and Aladjem (2009) removes the issue of a priori setting the number of hidden nodes. However, this brings up the problem of using an optimization algorithm with ANNs of differing sizes. Dhahri et al. (2012) has developed an algorithm

to train ANNs with varying number of hidden nodes using DE called muti-dimensional differential evolution (MD-DE). However, MD-DE has a drawback in that ANNs with more nodes (relative to the rest of the ANN population) are restricted to a subspace during the optimization process. I am proposing a new algorithm that removes this constraint. Also provided are convergence rate plots for the two MD-DE methods (original and proposed variant).

Lastly, as pointed out by Bennell and Sutcliffe (2004) it is possible for a ANN modeling a complex dynamical system to perform better in different regimes of the underlying asset's behavior. Therefore, the asset's conditional probability distribution will be learned. This distribution is conditional upon the asset's current price as well as the price history. This allows for conclusions to be drawn about the overall dynamics instead of the prices in a particular price range. Estimated models of the asset's price dynamics allows for algorithms such as least squares Monte Carlo to be used in estimating the American style option prices. Comparison of estimated option prices with observed option market prices will then provide insight into market consistency.

2.2 Learning Methods

Choosing an appropriate training algorithm for the optimization of parameters within an ANN is based upon the type of learning being performed. Learning is referred to as the ANN's response with the goal of better approximating or estimating the observed data. There are several major types of learning based upon the method that the ANN is exposed to the observed data.

2.2.1 Supervised Learning

Supervised learning is a method where the model generates a prediction, and then that prediction is compared with some known or desired response or outcome. The difference between the given model response and the desired outcome is used to generate an error

that is used as a feedback mechanism for model training at each iteration. A key feature of supervised learning is that the "correct" solution is known at each iteration.

2.2.2 Reinforcement Learning

Reinforcement learning is usually associated with some unbounded optimization problem where the optimal response is not known but instead the goal of the model is to optimize some fitness function of the output. An example of this would be for a model to optimize the likelihood that a user will click on a particular website, or maximize the monetary output from some game where the mechanics are unknown such as with slot machines.

2.2.3 Unsupervised Learning

Problems that mainly center around clustering data into various categories are unsupervised learning. The evaluation of correctness is usually only used for verification purposes and not for model training.

2.2.4 Ensemble Learning

Ensemble learning methods combine results of multiple learning methods to achieve better performance relative to some objective function. A common example of ensemble learning is training multiple models on different subsets or realizations of training data. K-fold cross-validation is an ensemble method that seeks to minimize overfitting to data by partitioning the data set into K sets and training K models on $K-1$ of the sets.

2.3 Training Methods

2.3.1 Batch Training

Batch training is when the model generates predictions for multiple input data and the feedback error is aggregated together for all the data used in the batch. This can be done for any number of input data points from a single point to all the data (both extremes).

2.3.2 Online Training

Due to the nature of the data and market dynamics, the ANNs will be trained in an online fashion where the data is processed a single time in the order it was collected. This is done to ensure consistency and provides several interesting twists:

- It allows the models to learn the behavior of the market dynamics without the assumption of stationary behavior i.e. the model is allowed to change over time.
- Since the models only have a single pass at the data to learn, the models must be able to learn "quickly".
- At each iteration, parameters within the asset models are updated and used in a single update step of the ANN parameters.

2.4 Growing and Pruning

Growing and pruning algorithms are used when it is desired to add or remove hidden nodes within a network during training. This may be done for multiple reasons including adding variability to the problem or removing redundant nodes to decrease number of trained parameters and increase training efficiency. Another reason for using growing and pruning algorithms is if the underlying model dynamics have changed, which then requires a different number of hidden nodes. The last point is centered around most problems having an exploration phase where several different network architectures are evaluated to determine the optimal or close to optimal architecture.

2.5 Time Series

Data sampled at regular time intervals can be viewed as a time series (data where the order is important). Common examples of this type of data is financial data, weather measurements, or counts over time of some event.

2.5.1 Stationarity

A time series process is said to be stationary if the joint probability distribution is independent of time. In the context of the problem proposed in this dissertation it means that the conditional probability distribution of the underlying price process is independent of time. A stationary process does not imply the process is constant, but instead is the same over all time intervals.

2.5.2 Process Variance

It is usually desired for a process of interest to have constant variance. This arises from the assumption of constant variance in many parametric models or inference techniques in estimating parameters. When the process does not have constant variance, the process is referred to as being heteroskedastic.

2.6 Kernel Density Estimation

Once a populations have been trained using an input time series and used to estimate prices for American style options, the output is a single option price. Multiple runs using different realizations of the underlying price process result in multiple samples from the distribution of option prices for the given price process.

When it is desired to reconstruct a probability density function (PDF) from a collection of observed data, one may assume a particular parameterized distribution such as Gaussian, Poisson, Cauchy, etc. or pursue a nonparametric distribution. Kernel density estimation (KDE) is a nonparametric method for reconstructing or estimating a PDF from observed data by assuming that the true distribution is a mixture of many random variables called kernels. KDE is used in this dissertation to compare the distribution of output parameters with their theoretical distributions.

A common KDE method is to use a mixture of normal variables with mixture weight N^{-1} for N observations. The mean of these normal variables is taken to be the respective

observed data. Finding an optimal variance for these kernels used in the mixture is a well studied problem and is described in detail in Scott (2015), Silverman (1986), Turlach (1993), and Bashtannyk and Hyndman (2001). Python's `scipy` library contains a function called `stats.gaussian_kde` that uses the method outlined by Scott (2015) as the default bandwidth selection algorithm to be used in this dissertation.

A problem here is in reconstructing the actual distribution from a collection of points sampled from that distribution. This is done by using Gaussian kernel density estimation with optimal kernel bandwidth as described in Scott (2015).

2.7 Options

Options are a type of financial derivative that grant the holder the option but not the obligation to purchase or sell an underlying asset at a given price within a given time window. The agreed upon asset price and time window are known as the strike price and exercise window respectively. A contract holder may choose to exercise the option contract within the exercise window, if the contract is not exercised, it expires and becomes worthless. Contracts that may only be exercised at the expiration time are known as European style options and options that may be exercised at any time up to and including the expiration time are known as American style options.

The two major types of options are puts and calls. Puts allow the contract holder to sell a pre-set amount of underlying assets (usually 100 shares) at a given price known as the strike. Similarly a call allows the contract holder to purchase a pre-set amount of underlying assets at a given price. Both puts and calls may be of European or American style with any strike price and expiration time. Less common types of options such as Bermudian style options may be exercised at any of a finite number of pre-determined exercise times. Another less common option are Asian options that have a conditional strike price based upon the time averaged asset price.

Least square Monte Carlo algorithm (LSM) is an algorithm used in the pricing of American style options. LSM allows one to calculate American option price based upon any

combinations of pricing rule sets for any process for which sample paths may be generated.

Chapter 3: Artificial Neural Networks

Artificial neural networks (ANNs) are a powerful tool that are used to estimate a known or unknown process based upon observed data. ANNs provide linear and non-linear transformations of the data in a one-way function. These transformations enable ANNs to approximate any function, provided there is enough training data, appropriate network architecture, and a long enough training time.

There are many different types of neural networks and many different methods for generating a feedback mechanism to be used in model training. For this dissertation a specific type of feedback is used in training of the ANNs that implicitly evaluates network learning. Evaluation of learning is necessary at each training step due to there not being an assumption of stationarity in the underlying time series process.

3.1 Anatomy of a Neural Network

A neural network consists of three regimes called the input layer, hidden layer(s), and output layer. Data is ingested into the model at the input layer, transformed at the hidden layer(s), and output at the output layer.

3.1.1 Input Layer

The inputs to a neural network may be discrete or continuous and are considered to be the exogenous data. Inputs may be anything from an observed time series, estimated parameters, fitted data, output from other learning models, or samples from an observed or estimated distribution.

Time Series Input

Any process that generates output data over time is generating a time series. Typically time series data is evenly spaced in time, but need not be. In financial markets, the price of an asset or derivative is dependent on time and the prices over time create a time series. Many models exist for analysis of this time series data, in particular auto-regressive moving-average (ARMA) and auto-regressive conditional heteroskedasticity (ARCH) models.

Instead of performing an analysis on the time series directly to fit a model of a particular type, an ANN may use the time series as an input to learn the appropriate model. Time series and predictive ANN models are used to approximate or learn the same asset behavior, namely the conditional probability distribution of the asset prices at the next time interval. Both models may additionally be applied to learn the conditional probability distribution for a finite number of intervals in the future, called forecasting. A huge amount of literature exists for both time series models ARMA and ARCH as well as using ANNs for forecasting of time series as shown by Donaldson and Kamstra (1996), Dhahri and Alimi (2006), Shumway and Stoffer (2010), and Hafner and Manner (2012).

Estimated Parameter Input

Rather than inputting the raw data obtained from a particular time series, it is possible to use ANNs to learn the conditional probability distribution of the data at the next time interval by using estimated parameters. Parameters are estimated from a finite length sliding window of time series data and used as the input to the ANNs. These inputs need not be estimated from the same length of sliding window and several versions of the same parameter estimated from different windows may be used.

Information contained in the raw time series data is used in the learning of the ANN implicitly and may be used to reduce the number of input dimensions. A separate problem arises in trying to identify the appropriate set of estimated parameters to use as input. Using too many estimated parameters of the same type may add extra parameters to be trained with little value in additional information. Estimated parameters should be chosen

to maximize the information they contain about the underlying dynamics of the time series process.

For financial models, these estimated parameters may take the form of the parameters within various time series and stochastic differential equation models. Common examples of this would be the auto-regressive and moving-average terms within ARMA models as well as drift and volatility within a geometric Brownian motion model. It is possible to use parameters such as drift multiple times if they are derived from different windows, say 30, 60, and 90 day histories of the underlying asset price.

Distributional Input

Given a set of distributions from an assumed set of models, it is possible to use these distributions as an ANN input via sampling of the distributions. Each distribution is treated as a single input, with each sample of the distribution being one realization of that input. Each training iteration uses a single independent sample from each respective distribution, this random sampling is used to create a single realization of the output. Aggregation of the output values may then be treated as output of a Monte Carlo simulation or as samples from an output distribution as discussed further in Section 3.1.3. Python's numerical library `numpy` functions such as `choice` and `bincounts` allow a straightforward method to take samples of these distributions.

3.1.2 Hidden Layer(s)

Nonlinear transformation of the input data occurs in the hidden layer(s) and may be done via deterministic or stochastic activation functions. Typical examples of deterministic activation functions are the sigmoid function, hyperbolic tangent, or radial basis functions such as a Gaussian function. Stochastic activation functions introduce variability within the data and can be used to generate a distributional output.

Non-linear transformations on the input data are done specifically in objects called "nodes" that induce a transformation on the input data. Each node takes as input a

weighted sum of the inputs from the previous layer, performs some nonlinear transformation induced by an "activation function". This activation function may be a sigmoid, hyperbolic tangent, Gaussian, or any number of other potential functions. Output nodes may or may not have an activation function associated with them. This is a parameter to be chosen by the user. Each node may also have a "bias" associated with it. In the general case, there is no restriction that each node have the same type of activation function or internal structure.

Different types of ANNs may allow for internal feedback by allowing the activation function output to be used as an additional input to any number of other activation functions within the network. Other non-linear transformations may induce randomness to the output via a stochastic bias or stochastic parameters within the activation functions.

3.1.3 Output Layer

The output layer is where the model output is generated or sampled. An output layer is used to linearly or non-linearly combine the outputs from the previous (or other) hidden layer(s).

Scalar Output

A common starting point for neural networks is the generation of scalar outputs that may then be used to construct higher dimensional objects such as vectors, matrices, etc. ANNs with scalar outputs generate a single output for each set of inputs. If the model is deterministic, then each unique set of inputs will generate a single output. When the model is stochastic, the inputs may be fed into the model multiple times to generate a different output at each run. The output values are then used in combination with an objective function to evaluate the model's performance with respect to that input.

Distributional Output

Usually when a stochastic or probabilistic model is used, the goal is to generate an output distribution. This distribution provides greater insight into the model's performance by providing additional information such as the variance and higher moments of the generated output. Certain types of objective functions as discussed in Section 3.3 require the use of distributional output.

3.2 Types of Neural Networks

There are many types of neural networks that have advantages and disadvantages depending on the problem being solved. Some networks are designed specifically for distributional output such as the mixture density networks described in Section 3.2.5, while some are used to simulate memory within a system, such as with recurrent networks described in Section 3.2.5. Some common neural networks are provided are

- Multi-Layer Perceptron,
- Cascading Neural Networks,
- Recurrent Neural Networks,
- Stochastic Feed Forward Neural Networks, and
- Mixture Density Neural Networks.

3.2.1 Multi-Layer Perceptron

The most basic of all ANNs is the multi-layer perceptron (MLP), also known as the classic feed forward neural network. A diagram of a single layer MLP network with two inputs, three hidden nodes, and two outputs is shown in Figure 3.1. The MLP network is easily trained by almost any optimization algorithm including back-propagation, Newton-Raphson, particle-swarm optimization, and differential evolution.

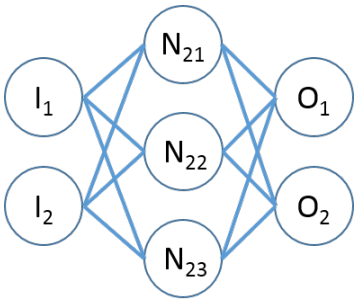


Figure 3.1: Exemplar diagram for MLP network.

3.2.2 Cascading Neural Networks

Starting with a a network that has a single hidden node, cascading neural networks (CNN) as described by Fahlman and Lebiere (1989) are constructed by iteratively adding a special hidden layer. This hidden layer is characterized by a single node that is connected to all previous nodes and all network inputs. Parameters within each node are trained via any of the training methods described in Section 3.3 until the best single node is given. Weights and internal parameters for each node are kept constant over the life of the model. Each subsequent node is added in this fashion taking the outputs from all previous nodes (and network inputs) as input. An example CNN is given in Figure 3.2 with two outputs and one output. The two hidden layers contain one hidden node each and receive input from all inputs and previous hidden nodes.

3.2.3 Recurrent Neural Networks

A neural network where each hidden node takes as input the output from all hidden nodes including itself is referred to as a recurrent neural network (RNN). These networks allow for the output of the hidden layer nodes to be treated as inputs to nodes in that same layer. This feedback type allows for "memory" within the model and is usually used for continuous time models. An example of an RNN is given in Figure 3.3 for a network with two inputs, two fully connected hidden nodes, and two outputs.

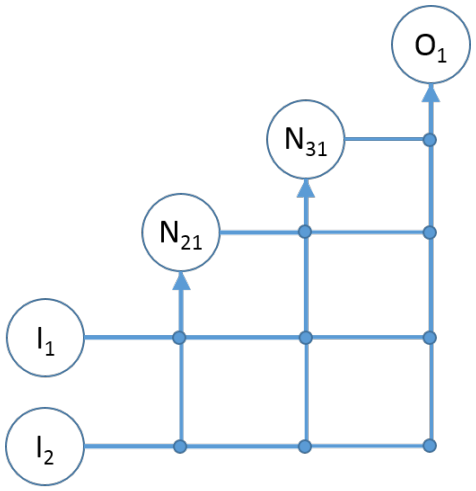


Figure 3.2: Exemplar diagram for cascading neural network network.

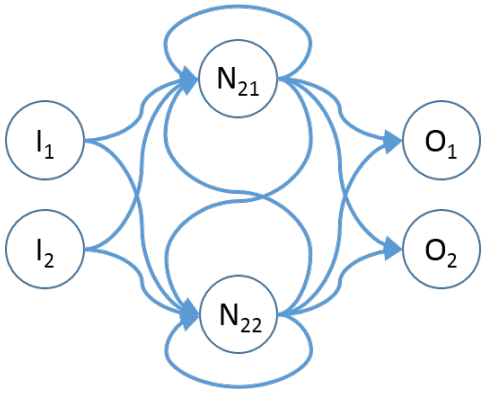


Figure 3.3: Exemplar diagram for recurrent neural network.

3.2.4 Stochastic Feed-Forward Neural Networks

Stochastic feed-forward neural networks have stochastic node weights. The means and variance of these weights are included as additional model parameters to be learned. The limitations of this network are analogous to those of mixture density networks.

3.2.5 Mixture Density Neural Networks

Mixture density neural networks (MDNN) use a traditional neural network to estimate the mean and variances of Gaussians used to reconstruct an output distribution. This gives

an output distribution implicitly. Parameters from any parameterized distribution may be learned this way. This network was successfully used by Dalziel et al. (2008) to determine the movement kernel associated with tracking animal movements. The movement kernel is the distribution of the animal's next observed position conditional upon its current position.

Learning the conditional probability distribution for the price of a financial asset is equivalent to learning a conditional movement kernel for animal migrations. The successful application of MDNNs to the related animal migration problem is the main reason that MDNNs are exclusively used in the remainder of this dissertation. A secondary reason for using MDNNs is their ease of use and computational similarity to classic multi-layer perceptrons.

3.3 Training Neural Networks

Training of neural networks is done via an objective function that is used to evaluate how well a given neural network is performing against some desired response. Further discussion will assume supervised learning is being used with a distributional output and scalar observed data. For distributional output, the objective function may use the likelihood of the output distribution, L^p norm, a f-divergence measure, or functions of meta-parameters such as measures of network complexity. The objective function may also penalize against undesired behavior as used in regularization.

3.3.1 L^p Norm Objective Functions

Given a network's output distribution $\mathcal{A}$ with respect to a given set of inputs, it is necessary to evaluate this distribution with respect to a single scalar observed data point. L^p norms, with $p > 0$, allow for the evaluation of a distance between the output distribution and observed data. This is illustrated in equations (3.1) and (3.2) by taking the expected value of the distance between the observed data y and a sample from the output distribution and is referred to as the L^p error. This error, $\mathcal{E}_p$, takes the same dimensional units as the observed data. It is shown in the subsequent subsections that the L^1, L^2, and L^∞ norms

are not an appropriate choice in objective function for this particular problem.

$$\mathcal{E}_p = E\left[\|\mathcal{A} - y\|_p\right] \tag{3.1}$$

$$= \int_{-\infty}^{\infty} \mathcal{A}(x)\left(x^p - y^p\right)^{\frac{1}{p}} dx \tag{3.2}$$

L^2 Norm

A common choice in objective functions is the L^2 norm. However, for the particular setup as described in this dissertation (distributional output with scalar observed data), this norm is not appropriate. The proof described in equations (3.3) - (3.9) shows that an objective function that seeks to minimize the L^2 norm will cause the variance of the output distribution to collapse to zero.

Assume that there are two functions, f and g with equal mean μ and different variances such that $\mathrm{Var}(\mathrm{g}) < \mathrm{Var}(\mathrm{f})$ evaluated with respect to some observed value c, then:

$$\mathrm{Var}(\mathrm{f}) = \int_{-\infty}^{\infty} (x-\mu)^2 f(x)\, dx \tag{3.3}$$

$$= \int_{-\infty}^{\infty} x^2 f(x)\, dx - 2\mu \int_{-\infty}^{\infty} x f(x)\, dx + \mu^2 \int_{-\infty}^{\infty} f(x)\, dx \tag{3.4}$$

$$= \int_{-\infty}^{\infty} x^2 f(x)\, dx - \mu^2 \tag{3.5}$$

$$= \int_{-\infty}^{\infty} (x-c+c)^2 f(x) - \mu^2 \tag{3.6}$$

$$= \int_{-\infty}^{\infty} (x-c)^2 f(x)\, dx + 2c \int_{-\infty}^{\infty} (x-c) f(x)\, dx + c^2 \int_{-\infty}^{\infty} f(x)\, dx - \mu^2 \tag{3.7}$$

$$= L^2(f;c) + 2c\mu - c^2 - \mu^2 \tag{3.8}$$

$$= L^2(f;c) - (c-\mu)^2 \tag{3.9}$$

Therefore, for any given function f, a delta distribution with mean equal to that of f has a lower L^2 error. If we assume that there exists another distribution with a different mean, then a delta distribution at that new mean has lower L^2 error. We can then assume that an optimization method will converge on the ANN parameters such that a distribution with a mean that minimizes the L^2 error and a zero variance will be reached and thus be unusable.

L^1 Norm

Using the L^1 error as an objective function on a neural network that generates a distributional output is not appropriate based upon a proof similar to that for L^2 error. This type of objective function causes the variance of the output distribution to collapse to a Dirac delta distribution with zero variance.

L^∞ Norm

Per the definition of the L^∞ norm, which is based upon the maximum distance between the observed data and any point within the output distribution, this norm is not appropriate. Since the output distribution is not compact, and is in fact non-zero almost everywhere, this maximum distance will always theoretically be infinite. Simulations using this objective function may appear to work due to the finite number of output distribution samples and thus the finite maximum distance relative to the observed data.

3.3.2 Likelihood Objective Functions

A second choice of objective function is one based upon the likelihood of the output parameters with respect to the observed data. This objective function seeks to maximize the likelihood of the model with respect to the realized value of the true conditional asset price distribution. This is simply the output distribution $\mathcal{A}$ evaluated at the realized price y. For mixture density neural networks, the output parameters are the mean, variance, and mixture coefficients for a set of Gaussian distributions $\mathcal{N}$. Evaluation of the likelihood L for

a mixture model with N Gaussian modes is simply the sum of the likelihood of the mean μ and variance σ^2 from each Gaussian mode weighted by its mixture coefficient m_i as shown in equation (3.10).

$$L\left(\mu_i, \sigma_i^2; x_i\right) = \sum_{i=1}^{N} m_i \mathcal{N}\left(\mu_i, \sigma_i^2; x_i\right) \tag{3.10}$$

3.3.3 F-Divergence Objective Functions

Objective functions that use an f-divergence measure as developed by Csiszár (1963) are based upon some measure of difference between the desired and observed distributions. For two distributions Q and P defined over some domain Ω, the f-divergence measure I_f is defined as in equation (3.11) and is always non-negative. The choice of the function f in equation (3.11) leads to different special cases of f-divergence measures such as the Kullback-Leibler divergence, total variation distance, χ^2 divergence, and Hellinger distance as shown in Österreicher (2002).

$$I_f\left(P, Q\right) = \sum_{x \in \Omega} p\left(x\right) f\left(\frac{q\left(x\right)}{p\left(x\right)}\right) \tag{3.11}$$

Kullback-Leibler Divergence

Similar to the K-S test statistic, the Kullback-Leibler (K-L) divergence is a measure of the distance between two probability distributions P and Q as shown in equation (3.12). The true or known distribution is usually expressed as P since the K-L divergence is not symmetric. Since the two arguments in equation (3.12) are the two distributions being compared, it is necessary to convert the distributions in the same method as done prior to using the K-S test with a non-constant underlying model. The K-L divergence may also be interpreted as the number of extra bits needed to encode a signal Q with a method optimized for the distribution of P.

$$I_{KL}(P,Q) = \int_{-\infty}^{\infty} p(x) \log \frac{p(x)}{q(x)} \mathrm{d}x \tag{3.12}$$

The K-L divergence is expanded to higher dimensions by definition of multidimensional integrals and a multidimensional probability distribution.

Other f-Divergence Measures

Other less commonly used f-divergence measures include the total variation distance shown in equation (3.13), χ^2-divergence shown in equation (3.14), and the Hellinger divergence shown in equation (3.15). These other measures are equally applicable as objective functions in training ANNs as the K-L divergence, but are simply less widely used. Each of these measures are expanded to higher dimensions via the definition of multidimensional integrals and multidimensional probability distributions.

$$I_{Var}(P,Q) = \int_{\Omega} |p(x) - q(x)| \, \mathrm{d}x \tag{3.13}$$

$$I_{\chi^2}(P,Q) = \int_{\Omega} \frac{(q(x) - p(x))^2}{p(x)} \mathrm{d}x \tag{3.14}$$

$$I_H(P,Q) = \left(\int_{\Omega} \left(\sqrt{p(x)} - \sqrt{q(x)} \right)^2 \mathrm{d}x \right)^{\frac{1}{2}} \tag{3.15}$$

3.3.4 Test Statistic Objective Functions

The benefit of using a test statistic objective function is that the resulting test statistic may be used as a metric to test if the learned output distribution is sufficiently different from the true distribution that generates the observed data. Test statistic objective functions are not a common choice of objective function based on my literature review, but provide the exact attributes required for testing if the trained ANNs are learning the true conditional probability distribution of the underlying process.

Kolmogorov-Smirnov Test Statistic

The Kolmogorov-Smirnov (K-S) test statistic in conjunction with the appropriate critical values, provides a method to test if two distributions are sufficiently different from one another. The K-S test statistic as shown in equation (3.16) is similar to the total variation distance of the probability measures if F_n and F were two probability distributions instead of the CDFs and is equivalent to the L^∞ metric for CDFs.

$$D_n = \sup_{x \in \mathbb{R}} |F_n(x) - F(x)| \tag{3.16}$$

For a conditional model with a known output distribution, it is possible to perform a transformation based upon the cumulative density function of the output distribution as described in Section 3.4.2 to transform the realization of the output to a realization from a uniform distribution. The key assumption here is that the CDF is correct with respect to the actual mechanics of the problem. Assuming this is correct, the K-S statistic gives a measure of the probability that the transformed points could have been sampled from a true uniform distribution. When the estimated model begins to diverge from the actual model, the K-S statistic will decrease, signifying that the model is diverging. By evaluating this statistic at each iteration, the resulting model is trained such that the CDF of the output distribution more closely resembles that of the correct model.

Anderson-Darling Test Statistic

It has been shown by Anderson and Darling (1952) and Stephens (1986) that the K-S test is not appropriate for small numbers of observations due to a lack of sensitivity in the test. The Anderson-Darling (A-D) test is essentially a equivalent to the K-S test but is more sensitive at lower numbers of observations. Since the A-D test is usually performed against the uniform distribution, the expressions in equations (3.17) and (3.18) show that the A-D test statistic is a function of the empirical CDF Φ of the observed data Y_i.

$$A^2 = -n - S \tag{3.17}$$

$$S = \sum_{i=1}^{n} \frac{2i-1}{n} \left[\ln\left(\Phi\left(Y_i\right)\right) + \ln\left(1 - \Phi\left(Y_{n+1-i}\right)\right)\right] \tag{3.18}$$

3.3.5 Regularized Objective Function

Regularization is characterized by a function that seeks to maximize some behavior of interest with an additional penalty term. This penalty term may penalize against any undesired behavior and need not be differentiable or even continuous. A possible form of a regularized objective function for distributional output would be a function that maximizes the likelihood of the observed data relative to the output distribution and penalizes for complexity within the ANN. Other potential forms may be a function of the likelihood with a L^p error term that penalize for large variance.

Akaike Information Criterion

For problems with a constant neural network architecture, minimizing the Akaike information criterion (AIC) is equivalent to maximizing the likelihood. AIC described in Shumway and Stoffer (2010) is shown in equation (3.19) provides a method to maximize the output distribution's likelihood L while simultaneously penalizing for model complexity, which is an example of a regularized objective function. Here, model complexity is defined to be the

total number of parameters to be optimized k such as model weights, mixture coefficients, etc. When the network architecture is allowed to change, AIC provides a metric to evaluate the trade off between complexity and accuracy.

$$\mathrm{AIC} = 2k - 2\log(L) \tag{3.19}$$

Corrected Akaike Information Criterion

For small sample sets, the corrected Akaike information criterion (AICc) may be more appropriate. As shown in equation (3.20), the AICc adds a correction term to the tradition AIC. As described in Cavanaugh (1997), this correction term is introduced to correct for small numbers of observed data and is a function of both the number of data points n used and the number of model parameters k in the model of interest. It is important to note that the AICc converges to the traditional AIC as the number of data points increases to infinity, and as such is more applicable to situations with few data points.

$$\mathrm{AICc} = \mathrm{AIC} + \frac{2k(k+1)}{n-k-1} \tag{3.20}$$

Bayesian Information Criterion

The Bayesian information criterion (BIC) provides an alternative metric to evaluate model complexity with respect to the model's performance. Like the AIC and AICc, the BIC is a Tikhonov objective function that seeks to maximize the likelihood of the output distribution with a penalty for model complexity. Unlike the AIC, the BIC penalty term is a function of both the number of data points n and the number of model parameters k. When comparing two models, a lower BIC implies a more parsimonious model. Work has been done by Yang (2005) and Burnham and Anderson (2004) to compare the advantages and disadvantages of the AIC and BIC.

$$\mathrm{BIC} = -2\log L + k\log n \tag{3.21}$$

3.4 Evaluation of Learning

For the problem of interest in this dissertation, the correctness of the output distribution with respect to the true conditional probability distribution of the underlying process may only be verified by a single observation from the true distribution. The difficulty arises because the output distribution is conditional upon the input data for the ANN, which changes at each iteration. Statistically verifying the correctness of a distribution from a single observation is highly unreliable due to the extremely small number of observations. Therefore, evaluation of the models' learning requires a method to incorporate observations from previous time intervals with respect to their different output distributions.

3.4.1 Training Conditional Distribution Models

When building conditional models it is necessary to validate the correctness of the underlying model with respect to observed data. Typically the model may be run numerous times to allow for multiple realizations of observations that may then be used for the estimation of the true distribution via a non-parametric approach such as Gaussian kernel reconstruction. However, in the case of real time series applications, only a single realization is possible for each time interval. An example of this is in financial markets where a single price of an asset is given at any given time and cannot be repeated for the purpose of acquiring additional data. This is because the observations are conditional upon the process at that unique instance in time. Validating the performance of these time series models such as auto-regressive moving-average (ARMA) models requires a special step to compare model performance over time.

3.4.2 Converting Conditional Distributions

Objective functions that use a measure of the difference between two distributions such as the f-divergence measures or the test statistic objective functions require a single estimate of the true distribution for comparison. Since the true distribution changes at each iteration, the following method is used to convert the conditional probability distributions and the

single observations at each time interval onto a single constant distribution.

In Monte Carlo methods, it is usually sufficient to have a random number generator that outputs samples from a uniform distribution. Samples from this uniform distribution x are converted to the desired distribution g via the inverse of the CDF G as shown in equations (3.22) and (3.23).

$$x \sim U[0,1] \tag{3.22}$$

$$g \overset{d}{=} G^{-1}(x) \tag{3.23}$$

Using this starting point, samples x from a known distribution g may be transformed to the distribution $U[0,1]$ by the use of the CDF G.

$$x \sim g \tag{3.24}$$

$$G(x) \overset{d}{=} U[0,1] \tag{3.25}$$

This provides a method to convert the conditional probability distributions at each iteration for any ANN into the uniform distribution on the interval $[0,1]$. Similarly, the single observation from the underlying process is transformed to a sample from the uniform distribution, provided that the output distribution from the ANN is correct. Objective functions that measure the difference between the two distributions such as f-divergence measures or the test statistic objective functions presented in Sections 3.3.3 and 3.3.4 respectively may now be used to evaluate learning. Furthermore, test statistic objective functions such as the K-S or A-D test statistic provide a framework for testing if the estimated and true distributions are sufficiently different from each other.

3.4.3 Kolmogorov-Smirnov Test

For this dissertation, the K-S test statistic was chosen as the objective function to be used for evaluating ANN performance based upon its ability to test if the estimated and true distributions are sufficiently different, its computational simplicity, and because it is a well known and accepted test. It should be noted that the following methodology is almost exactly the same for the A-D test statistic with only with differences in the calculation of the critical values.

The K-S test was designed for the purpose of determining if a set of samples are realizations of a particular known distribution. The null hypothesis H_0, shown in equation (3.26), is that the samples are taken from $U[0,1]$ with the alternate hypothesis H_1, shown in equation (3.27), being that the samples are realizations of some other distribution. The rejection of the null hypothesis H_0 indicates that the ANN is not learning the appropriate dynamics of the underlying process.

$$H_0 : g = U[0,1] \tag{3.26}$$

$$H_1 : g \neq U[0,1] \tag{3.27}$$

For a two-sided confidence test of size α, it is first necessary to calculate the critical value K_α and the Kolmogorov-Smirnov test statistic D_n. The critical value for a size α test is calculated using the CDF of the Kolmogorov distribution as shown in equations (3.28) - (3.30). Calculated values for the critical values K_α are presented in Table 3.1 for 10, 20, 40, 80, and 160 samples at 90%, 95%, 98%, and 99% confidence. The critical values are sometimes expressed as $K_{n,\alpha}$ where the values K_α are divided by the square root of the number of samples n.

Table 3.1: Kolmogorov-Smirnov test statistic critical values from O'Connor and Kleyner (2011).

No. Trials	$K_{0.10}$	$K_{n,0.10}$	$K_{0.05}$	$K_{n,0.05}$	$K_{0.02}$	$K_{n,0.02}$	$K_{0.01}$	$K_{n,0.01}$
10	.36866	.11658	.40925	.12942	.45662	.14440	.48893	.15461
20	.26473	.05920	.29408	.06576	.32866	.07349	.35241	.07880
30	.21756	.03972	.24170	.04412	.27023	.04933	.28987	.05292
40	.18913	.02990	.21012	.03322	.23494	.03715	.25205	.03985
80[1]	.13640	.01525	.15205	.01700	.16882	.01887	.18224	.02038
160[1]	.09644	.00762	.10752	.00850	.11938	.00944	.12886	.01019

$$\Pr\left(K \leq x\right) = \frac{\sqrt{2\pi}}{x} \sum_{k=1}^{\infty} \exp\left(\frac{-\left(2k-1\right)^2 \pi^2}{8x^2}\right) \tag{3.28}$$

$$\Pr\left(K \leq K_\alpha\right) = 1 - \alpha \tag{3.29}$$

$$K_\alpha = \arg_x \left\{1 - \alpha = \frac{\sqrt{2\pi}}{x} \sum_{k=1}^{\infty} \exp\left(\frac{-\left(2k-1\right)^2 \pi^2}{8x^2}\right)\right\} \tag{3.30}$$

The Kolmogorov-Smirnov test statistic D_n is taken to be the supremum of the difference between the CDFs of the two distributions being compared. As shown in equation (3.31) let G represent the CDF of the estimated distribution g that is output from the ANNs. Within the K-S test, the two distributions being compared are the CDF of the estimated distribution g and the uniform distribution on $[0, 1]$. Therefore, the empirical CDF of the first distribution is constructed by evaluating the CDF, G, of the estimated distribution, g, at the observed data points x_i as illustrated in equations (3.32) and (3.33).

$$G(x) = \int_{-\infty}^{x} g(s)\,\mathrm{d}s \tag{3.31}$$

$$D_n = \sup_x \left| \int_{-\infty}^{x} G(s)\,\mathrm{d}s - \int_{-\infty}^{x} U_{[0,1]}(s)\,\mathrm{d}s \right| \tag{3.32}$$

$$= \sup_{x\in[0,1]} \left| \frac{1}{n} \sum_{i=1}^{n} I_{[-\infty,x]}\left(G\left(x_i\right)\right) - x \right| \tag{3.33}$$

$$\sqrt{n} D_n \leq K_\alpha \tag{3.34}$$

The null hypothesis H_0 is rejected at significance level α in favor of the alternative if the inequality in equation (3.34) is violated. Another interpretation of the rejection of H_0 is that the observed samples came from a distribution other than the true distribution. However, if H_0 is not rejected it is said that there is insufficient evidence to prove that the distributions are different; and thus insufficient evidence to say that the ANN is not learning the appropriate dynamics of the underlying process.

The plot in Figure 3.16 was generated by averaging the resulting K-S test statistic from 200 simulations where equal numbers of random samples were taken from either the standard uniform or standard normal distribution. This was done to illustrate the range of values that are to be anticipated when using the evaluating learning within an ANN that is using the K-S test statistic as an objective function.

3.4.4 Training Performance using K-S Statistic

Before the K-S test statistic may be used to train fitted Gaussian modes to streaming data, it is necessary to confirm its performance against a few test cases using the methodology presented in 3.4.3. The test cases are presented in equations (3.36) - (3.39) in increasing complexity, and represent a variety of processes with Gaussian noise ϵ distributed as described in equation (3.35). The first two test cases are considered to be stationary processes,

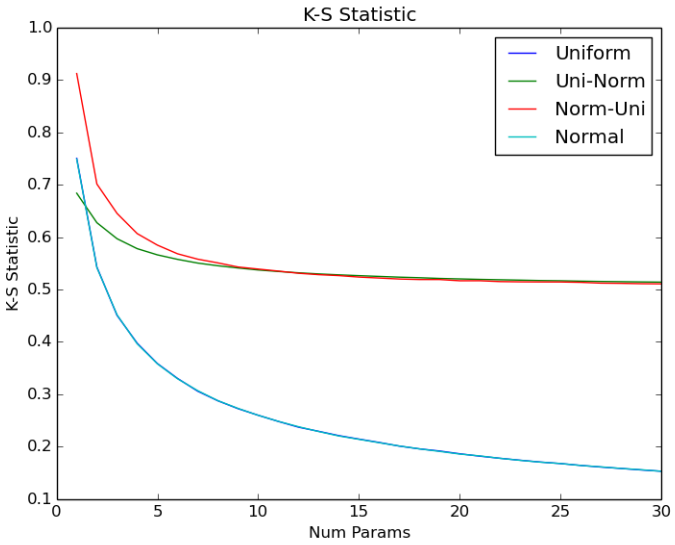


Figure 3.4: Information exchange rate for three parent vectors in evolutionary global optimization algorithms simulated by Monte Carlo.

while the last two are non-stationary processes. Each test process was started with an initial value of 100.0, however this was chosen arbitrarily since all four processes allow for negative values.

$$\epsilon \sim N(0.05, 0.2) \tag{3.35}$$

$$f_1(t) = \epsilon \tag{3.36}$$

$$f_2(t) = f_2(t-1) + \epsilon \tag{3.37}$$

$$f_3(t) = \cos\left(\frac{t}{2}\right) + \epsilon \tag{3.38}$$

$$f_4(t) = f_4(t-1) + \frac{\cos\left(\frac{t}{2}\right)}{1+\exp(-0.02t)} + \epsilon \tag{3.39}$$

ANN performance against each of the four test processes in equations (3.36) - (3.39) are respectively shown in Figures 3.5-3.8. In each case, 80 homogeneous single layer ANNs

with 2, 3, 4, and 5 hidden nodes are trained over 300 iterations.

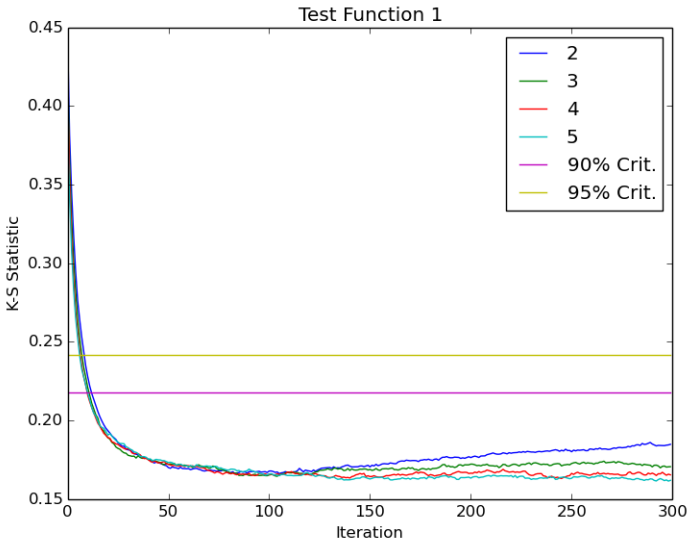


Figure 3.5: Performance of K-S test statistic as fitness function for training single layer homogeneous neural networks against the test function in Equation 3.5.

It can be seen from the plots, that in each case, the ANNs were capable of learning the appropriate dynamics of the underlying process. In all four cases, the ANNs with only 2 hidden nodes diverged towards the end of training and approached the critical value for 90% confidence in rejecting the null hypothesis. Critical values can be seen in Table 3.1 for 30 samples with 90%, 95%, 98%, and 99% confidence. This rejection is indicative of the 2 hidden node networks inability to appropriately learn the conditional probability distribution of the underlying process.

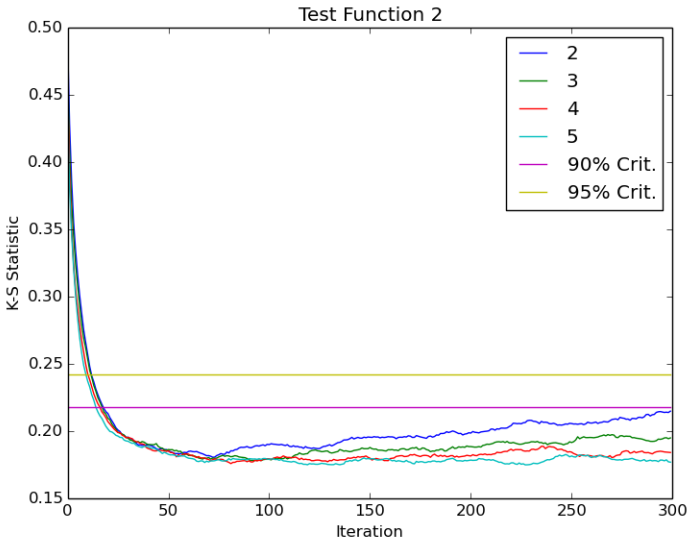


Figure 3.6: Performance of K-S test statistic as fitness function for training single layer homogeneous neural networks against the test function in Equation 3.6.

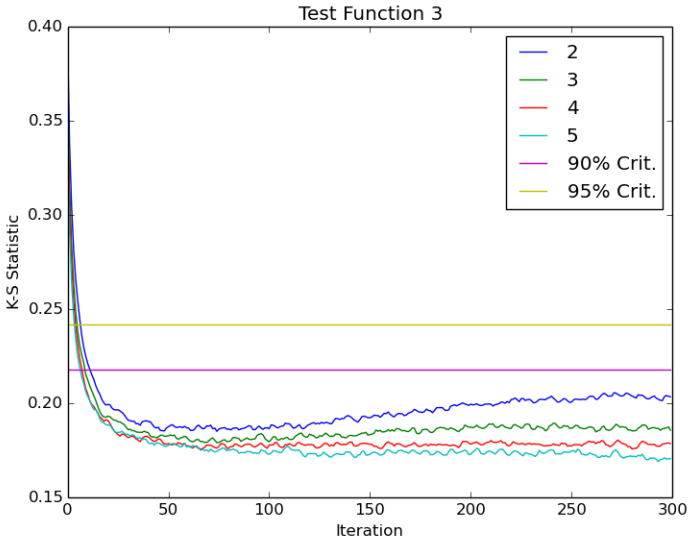


Figure 3.7: Performance of K-S test statistic as fitness function for training single layer homogeneous neural networks against the test function in Equation 3.7.

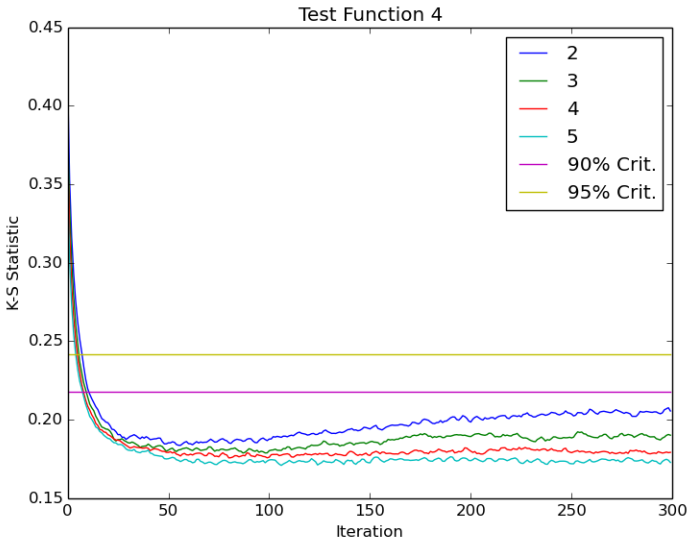


Figure 3.8: Performance of K-S test statistic as fitness function for training single layer homogeneous neural networks against the test function in Equation 3.8.

Chapter 4: Determination of Model Dimensionality and Complexity

A major problem when using ANNs is that the optimal architecture (number of hidden layers and number of nodes per hidden layer) is not known a priori. This problem has been pointed out by Dhahri et al. (2012), Rumelhart et al. (1986), Kiranyaz et al. (2014), and Bennell and Sutcliffe (2004). This problem is resolved by performing an exploration phase prior to the running of the real data. Issues may arise with this construct if the data is received in a streaming fashion from a dynamic process as in online learning. If the optimal network architecture is achieved, it would only be optimal relative to the state of the process when the training data was generated. If the underlying process were to change significantly from the previously learned process, the learned network performance will degrade. This degradation will continue unless the learned network is capable of responding to a changing underlying process. Therefore, it is necessary to derive a training algorithm that allows for a non-stationary underlying process and provides the user a more flexible methods for setting model architectures.

For this dissertation, a stochastic metaheuristic optimization algorithm will be used for model training due to the flexibility it provides in allowable output type and objective function. Particle swarm optimization and other swarm optimization type algorithms that rely on a globally optimal solution obtained over previous iterations are not applicable for problems with non-constant underlying processes and described further in Section 5.2.1. Therefore, only evolutionary algorithms such as the genetic algorithms and differential evolution algorithms will be considered further. Evolutionary algorithms such as GA and DE, to be described further in Section 5.2.2 have the advantage that once the parameters to be optimized are arranged into a single vector, either optimization algorithm may be used.

The ordering of the parameters is not important so long as the same schema is used for each network. This ordering is only important for the preservation of the micro-structure associated with parameters within a given node as is described further in Section 4.1.4.

When considering models that are allowed to grow, shrink, or change architecture via a growing and pruning algorithm, it is necessary to rectify having a heterogeneous pool of candidate vectors within the optimization algorithm. To better understand the additional steps that must be taken prior to information exchange and prior to trial vector evaluation against the given objective function within evolutionary algorithms, an ontology needs to be developed to describe the relationships between neural networks of different architectures.

4.1 ANN Dimensionality

For feed-forward ANNs, the dimensionality of the network refers to the number of hidden layers and the number of nodes per each hidden layer. Consider a model with N_i inputs and N_o outputs, these parameters are set in the definition of the problem and, for the rest of this dissertation, assumed to be constant for the entirety of the network's training. The parameter N_H is used to refer to the number of hidden layers with N_{H_k} referring to the number of nodes in the k^{th} hidden layer. An ANN may then be completely described using a tuple of length $2 + N_H$ such that the first entry corresponds to the number of network inputs, the next N_H entries correspond to the number of nodes per respective hidden layer, and the last entry corresponds to the number of network outputs. Therefore, the exemplar network shown in figure 4.2c may be described by a 4-tuple expressed as 2-3-2-2.

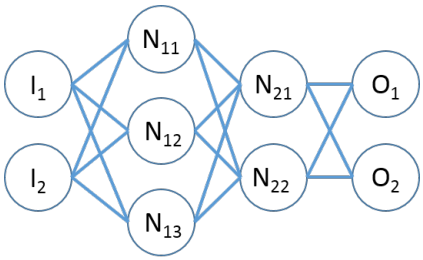


Figure 4.1: Representative 2-3-2-2 ANN with no bias.

Additional constraints placed on the ANN dimensionality are imposed by the type of ANN being considered. MDNNs require that the number of outputs be $2M+(M-1)$ where M is the number of output modes defined by the problem. The first $2M$ outputs are the mean and variance for each output mode, while the additional $(M-1)$ outputs are used as mixture coefficients. Transformations may be applied to the output values to allow for simpler activation functions in the output layer nodes. An example transformation on the output is to take the square of an output to ensure the output value is always non-negative as used for outputs that correspond to the variance of each Gaussian mode in MDNNs. Another transformation is squaring all output values for the mixture coefficients and then scaling such that they sum to unity.

4.1.1 ANN Population Composition

When using metaheuristic optimization or ensemble methods, it is necessary to seed, train, and evaluate multiple ANNs for the same input data. This collection of models is called the ANN population and may be homogeneous or heterogeneous. A homogeneous population of ANNs used in metaheuristic optimization or ensemble methods all have the same activation function, input and output type, and network architecture. When any of these attributes within the population are not the same for each member of the population, the population is said to be heterogeneous.

In particular interest to this dissertation is the concept of a population of ANNs with heterogeneous architectures; which refers to a set of ANNs where all the ANNs do not have either the same number of hidden layers or the same number of hidden nodes per each respective hidden layer. It is assumed that these ANNs only differ in the number of hidden layers and hidden node per respective hidden layer, and not in the number of inputs, outputs, or in the micro-structure within each node.

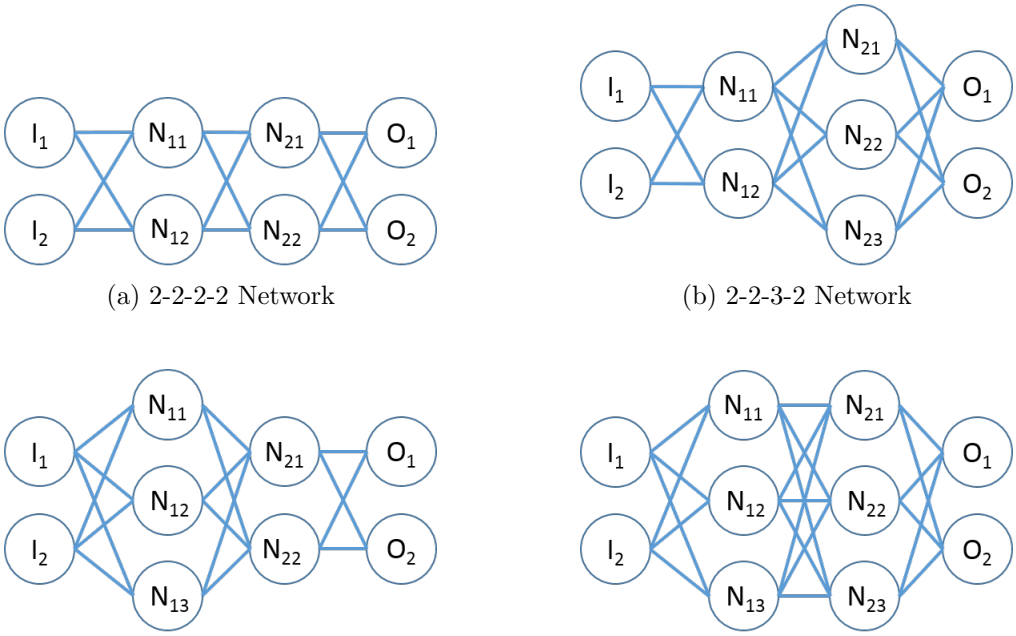
4.1.2 Projection of ANNs

The parameters within an ANN to be optimized by some method may be organized into an N dimensional vector where each entry corresponds to a particular parameter within the network. These parameters may be arranged into a vector in several different ways depending on the numbering scheme being used. During optimization it is common to use these parameter vectors as the object to be optimized. It is necessary for these vectors to not only have a consistent ordering scheme, including the numbering scheme for the parameters within each hidden node, but to also have the same length. This internal structure is a direct result of the ANN structure being used. An optimization method such as differential evolution involves the addition and subtraction of parameter vectors from three different ANNs, difficulties arise when these ANNs do not share an identical structure. Operations such as node addition or node deletion are used to transform an ANN from one dimension onto another, this transformation is called a projection. Using projection, an ANN may have nodes added to or deleted from the network such that the resulting structure is identical to that of the other ANNs being compared.

An ANN A is referred to as being in the subtractive projection of another ANN B if the model A may be achieved by the removal of nodes from B. That is to say if nodes were to be removed from the larger of the two models such that the small model is the result, then the smaller model is a subtractive projection of the larger model. Figure 4.2 shows four ANNs of varying sizes that all have a 2-N_{H_1}-N_{H_2}-2 architecture. Using the networks shown in Figure 4.2 network (a) is within the subtractive projection of networks (b), (c), and (d). However, network (b) is only within the subtractive projection of network (d). From this point, I will use the notation $A \subseteq B$ to refer to a network A as being within the subtractive projection of a larger network B.

Similarly, an ANN A is referred to as being in the additive projection of another ANN B if the model A may be achieved by the addition of nodes onto B. Using Figure 4.2, network (d) is within the additive projections of networks (a), (b), and (c) since all three models must have nodes added such that their structure is identical to that of (d).

Since a projection from one network onto another may not consist entirely of node additions or deletions, it is important to note that a projection may be neither of these types or both. The null projection of a network onto itself is both additive and subtractive since it requires no node deletions or additions respectively. Conversely, the projection of network (c) onto network (b) in Figure 4.2 is neither additive nor subtractive since it requires both operations.



(a) 2-2-2-2 Network

(b) 2-2-3-2 Network

(c) 2-3-2-2 Network

(d) 2-3-3-2 Network

Figure 4.2: Representative two layer ANNs with no bias.

4.1.3 Random Projection of ANNs

In Figure 4.2, nodes may be removed from network (d) to achieve the same architecture as networks (a), (b), or (c). Considering the subtractive projection of (d) onto (c), the resulting network is achieved by removing the first node of the second hidden layer within model (d) resulting in a valid 2-3-2-2 network. However, the removal of the second or third nodes within the second hidden layer of model (d) yields the same result. Each of the three choices result in an equivalent resulting architecture, of type 2-3-2-2. Similarly, there are nine possible projections of (d) onto (a).

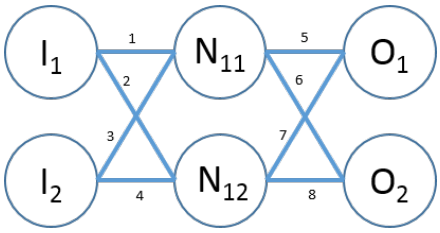
Consider two models with the same number of hidden layers, with model A being a N_i-j_A-k_A-N_o network and B being a N_i-j_B-k_B-N_o network such that $j_A \leq j_B$ and $k_A \leq k_B$. Then there are $\prod_i^N A_i B_i$ possible projections. We reserve the option to randomly choose one of these projections for a later discussion. Similarly, the additive projection of network (c) within Figure 4.2 onto (d) may be achieved by the addition of a node prior to the first node, between the two nodes, or after the second node. Each of these three models will result in an equivalent network architecture.

4.1.4 Parameter Vector and Network Permutations

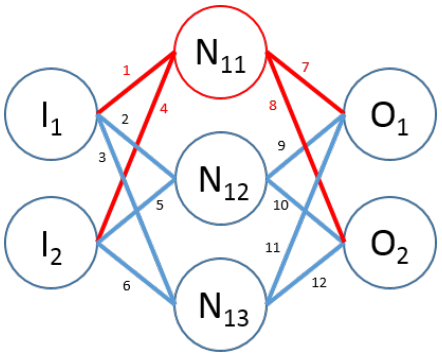
Without loss of generality, we will assume a counting scheme where we label the node connections starting from the input layer to the first node in the first hidden layer, and then from the first input to the second node in the first hidden layer, and so forth as shown in Figure 4.3a. The final connection counted in this fashion would be the connection between the last node in the last hidden layer to the last output node. Thus, adding a node to the 2-2-2 may result in any of the renumberings shown in Figures 4.3b, 4.3c, or 4.3d.

The parameter vector to be optimized is a vector constructed by concatenating each parameter tuple from the node connections in the order described above. At this point we can then assume each connection as being a tuple of parameters needed to be optimized and that the resulting 2-3-2 network to have any of the parameter vectors illustrated in Figure 4.4. It is important to note that the nodes are added or deleted from the network, the connections are recounted in the same fashion and the parameter vector reconstructed.

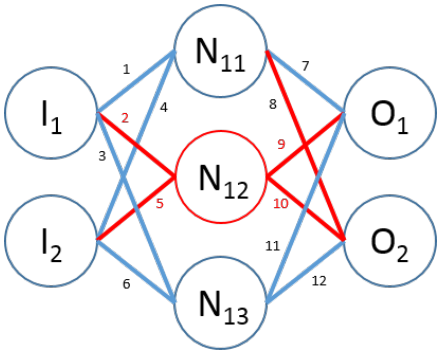
Therefore, the node addition and then subsequent node deletion will result in the parameter mapping shown in Figure 4.5. It is important to note that by changing the ordering of the connections, the internal structure of the parameter vector changes while leaving the vectors' norm unchanged. This important fact allows one to then choose an arbitrary counting scheme when labeling the internal connections and as such is then agnostic to the ordering of the nodes in each hidden layer. Therefore, permuting of renumbering the nodes within a layer has no effect on the model output.



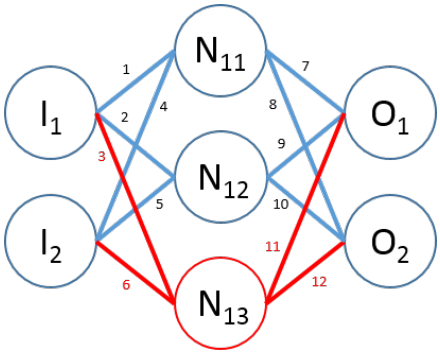
(a) Counting schema for 2-2-2 network.



(b) Counting schema for 2-3-2 network.



(c) Counting schema for 2-3-2 network.



(d) Counting schema for 2-3-2 network.

Figure 4.3: Various types of client computer functions.

Resulting Parameter Vector During Node Addition

Within the current MD-DE algorithm developed by Dhahri et al. (2012), the candidate solution being considered is always the smallest common additive projection of all three parent vectors. As discussed above, when nodes are added or deleted, the node connections are relabeled and the parameter vector is reconstructed. However, since we are adding a node that has no functional influence on the model, the parameter vector is transformed as shown in Figure 4.6a. The parameters associated with the added node(s) are inserted into the parameter vector as zeros (in the appropriate vector positions dictated by the connection counting schema).

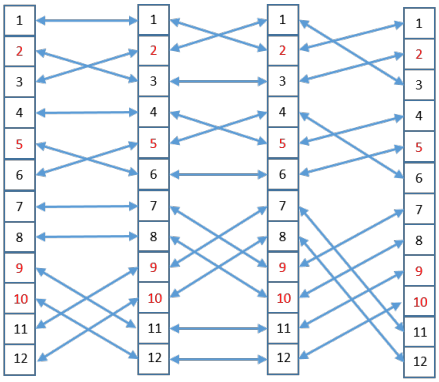


Figure 4.4: Flowchart of general multidimensional evolutionary algorithm.

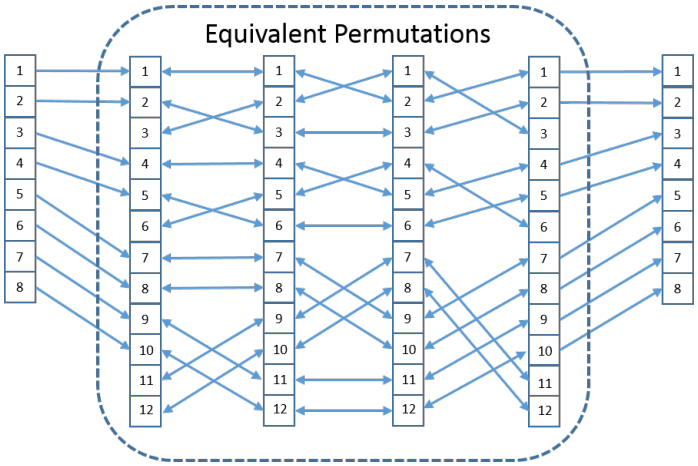


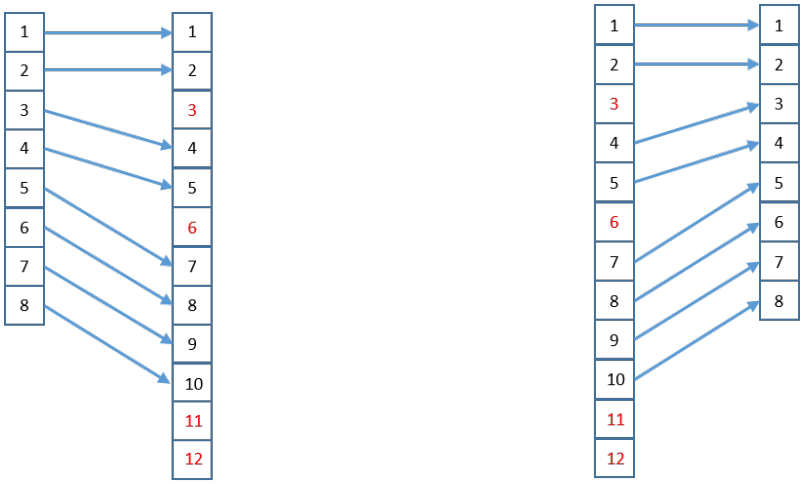
Figure 4.5: Flowchart of general multidimensional evolutionary algorithm.

Resulting Parameter Vector During Node Deletion

A MD-DE variant may then be proposed that always takes the largest common subtractive projection of all three parent vectors. The parameter vector for a network that has a deleted node is transformed as shown in Figure 4.6b.

4.1.5 Common Projections

When comparing two ANNs of differing dimension (but same number of inputs, hidden layers, and outputs), it is necessary to project the networks onto a shared or common



(a) Parameter mapping for addition of node 13. (b) Parameter mapping for deletion of node 13.

Figure 4.6: Various types of client computer functions.

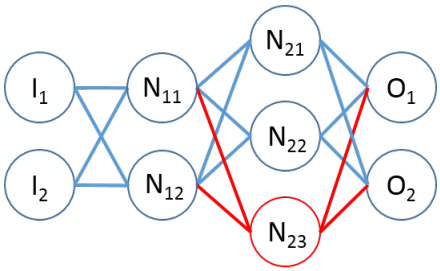
projection. For example, when comparing networks "b" and "c" in Figures 4.2b and 4.2c respectively, neither network is a projection of the other. It is then necessary to find a projection that is common to both networks. This common projection is necessary when performing parameter vector operations such as those within the differential evolution algorithm.

Common Subtractive Projections

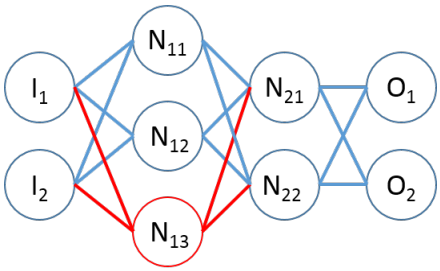
Every pair of networks being compared will have a common subtractive projection (meaning a common projection achieved by only node deletions) since a network with zero nodes in each hidden layer is common to every network. Therefore, we wish to find the largest common subtractive projection that requires the least amount of node deletions amongst all networks being considered. This projection may also be interpreted as the largest subset shared between the networks. Since the measure of distance here is similar to the Hamming distance (integer counts of nodes to be deleted), there may be more than one largest common subtractive projection.

Common Additive Projections

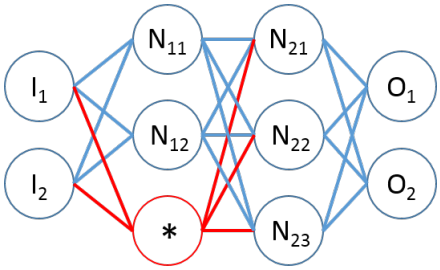
Similarly, every pair of networks will have an common additive projection (meaning a common projection achieved by only node additions) since a network with an infinite number of nodes in each hidden layer is common to every network. As with common subtractive projections, we wish to find the smallest common additive projection that requires the least amount of node additions amongst all networks being considered such that the resulting networks all have identical architectures.



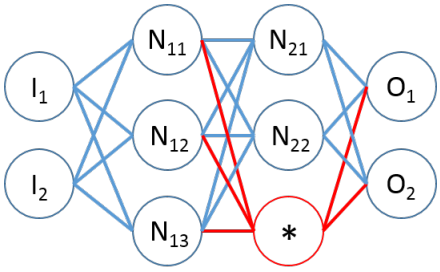
(a) Largest common subtractive projection of networks show in Figures 4.2a and 4.2b.



(b) Largest common subtractive projection of networks show in Figures 4.2a and 4.2c.



(c) Smallest common additive projection of networks show in Figures 4.2b and 4.2d.



(d) Smallest common additive projection of networks show in Figures 4.2c and 4.2d.

Figure 4.7: Various types projections between networks of different architectures.

4.1.6 Measure(s) of ANN Complexity

Complexity of the ANN refers to the total number of parameters within the ANN. The number of parameters for MDNNs without and with bias are shown in equations (4.2) and (4.3) respectively. The number of parameters for stochastic feed forward neural networks

without and with bias are shown in equations (4.4) and (4.5) respectively. A meta-parameter such as AIC, AICc, or BIC as introduced in Section 3.3.5 may be used to compare the model performance for networks of differing architectures.

$$N_L = 2M + (M - 1) \tag{4.1}$$

$$p = \sum_{i=0}^{L} N_i N_{i+1} \tag{4.2}$$

$$p = \sum_{i=0}^{L} (N_i + 1) N_{i+1} \tag{4.3}$$

$$p = \sum_{i=0}^{L} (N_i + 2) N_{i+1} \tag{4.4}$$

$$p = \sum_{i=0}^{L} (N_i + 3) N_{i+1} \tag{4.5}$$

An alternate approach to measuring model complexity is proposed by Dhahri et al. (2012), which uses an appropriate model fitness function f as described in equation (4.6). The model that minimizes this fitness is considered to be the "optimal" model. Model performance is evaluated based upon the accuracy of the output distribution with respect to observed asset prices via the likelihood function. An optimization algorithm will then seek to maximize the likelihood value of the observed value with respect to the output conditional price distribution. Model complexity is evaluated based upon the AIC. AIC is used in statistical models such as time-series models and has a natural extension to ANNs. A model's fitness can be evaluated as a linear combination of output error and the ANN complexity as used in Dhahri et al. (2012). Preference can be weighted such that simple models are preferred at the expense of accuracy.

$$f = \alpha f_1 + \beta f_2 \tag{4.6}$$

The parameters f_1 and f_2 are the evaluations of the accuracy and complexity objective functions respectively and the parameters α and β are the weights associated with f_1 and f_2 respectively. Therefore, replacing the value of f_1 with the output error and the value of f_2 with the AIC results in the fitness function described in equation (4.7).

$$f = \alpha E + \beta \text{AIC} \tag{4.7}$$

4.2 Growing and Pruning Algorithms

An ontology has been presented to describe the various transformations that may be used to convert a set of parameter vectors of ANNs onto a vectors of equal length. Therefore, it is natural to now include a method that allows ANNs to add or subtract nodes based upon the model performance. This method is known as a growing and pruning algorithm and is usually an amalgam of a growing algorithm and a pruning algorithm and need not be used together.

4.2.1 Growing Algorithm

Feedback from an ANN's objective function may be used as a signal that indicates a model is not capable of learning the appropriate behavior within the desired error threshold. A growing algorithm is the mechanism that adds new nodes to an ANN as certain conditions are met. In the case of using the K-S test statistic on a N-day sliding window, it can be determined if additional nodes need to be added based upon a poor resulting test statistic. As the K-S statistic degrades, the growing algorithm adds nodes to attempt to improve the test statistic.

4.2.2 Pruning Algorithm

When using a pruning algorithm, the objective function may be used to signal that the ANN is overfitting data. This overfitting would lead to a poor K-S test statistic that indicates a degradation of model performance and signals the removal or pruning of a node via a pruning algorithm. The choice of node to be pruned provides multiple choices as described in Section 4.1.4. One method for choosing the node to be deleted is to evaluate a node's contribution to the output result and compare against the additional model complexity of having an additional node. By seeding a network with a large number of nodes and then using only a pruning algorithm leads to an algorithm similar to simulated annealing where the problem is relaxed onto the appropriate network architecture.

4.2.3 GGAP-RBF

To resolve the main problem of determining the optimal ANN dimension, the naïve method involves running many ANNs in an experimentation phase to determine the architecture that works best while simultaneously balancing model performance and complexity. An interesting way to resolve this is via self-sizing algorithms such as multi-dimensional particle swarm optimization (MD-PSO) developed by Kiranyaz et al. (2014) or a growing and pruning algorithm such as GGAP-RBF developed by Huang et al. (2005).

GGAP-RBF grows and prunes nodes within an ANN by calculating the significance each node has in contributing to the overall reduction in output error. The growing and pruning is performed during the training process and as such is suited for online learning problems. Pruning is performed when the significance of a particular node is below a particular threshold. Similarly, a hidden node is added when a given input generates an output that is sufficiently far from the model's output.

The calculation of the node's significance is done analytically based upon the known distribution of the input values. While the inputs themselves come from common distributions such as normal and log-normal, these distributions change at each iteration based upon the

realized values of the underlying stock. Therefore it is first necessary to estimate this underlying distribution and then calculate the significance for a to be defined distribution of input distributions. Huang et al. (2005) gives analytical expressions for inputs of uniform, normal, Rayleigh, and exponential distributions. The calculus for log-normal distributions will need to be performed for use as well as generating an expression for significance of mixtures of distributions. An example calculation for a scalar input that follows a log-normal distribution is given in equations (4.8)-(4.10) where E_{sig} is the significance of a given node being considered.

$$E_{\text{sig}}(k) = ||\alpha_k||_q \left(\int_X \exp\left(\frac{-q||\mathbf{x}-\mu_k||^2}{{\sigma_k}^2} \right) p(x) \mathrm{d}\mathbf{x} \right)^{\frac{1}{q}} \tag{4.8}$$

$$p(x) = \frac{1}{x\sigma\sqrt{2\pi}} \exp\left(\frac{-(\ln x - \mu)^2}{2\sigma^2} \right) \tag{4.9}$$

$$E_{\text{sig}}(x) = \frac{||\alpha_k||_q}{\left(\sigma\sqrt{2\pi}\right)^{\frac{1}{q}}} \left(\int_{-\infty}^{\infty} \frac{1}{x} \exp\left(\frac{-q(x-\mu_k)^2}{{\sigma_k}^2} - \frac{(\ln x - \mu)^2}{2\sigma^2} \right) \mathrm{d}x \right)^{\frac{1}{q}} \tag{4.10}$$

As applied to the proposed problem in this dissertation, the GGAP-RBF algorithm is not applicable due to the lack of assumptions imposed on the input time series. The input time series is not assumed to be parametric or even stationary. Instead a growing and pruning algorithm is used that uses the ensemble complexity as the signal. The ensemble complexity is defined here to be the average number of parameters over all ANNs within the population. For a population of homogeneous neural networks, the ensemble complexity is identical to the individual model complexity. A value for the desired ensemble complexity is set at model start and is used as the target value to signal model growing and pruning. A simple algorithm is used where a randomly selected model with probability p has a random node either added or deleted if the ensemble complexity is lower or higher than the target value. A dead gap is used in the ensemble complexity signal to prevent excessive oscillations

in model architecture.

Chapter 5: Optimization

Training of an ANN is, an optimization problem; i.e. find the best model parameters such that the ANN performs the "best". As with most optimization problems, there are many ways to evaluate what it means for a solution to be "best". Usually, this means the set of parameters that result in the minimization of some objective function that may be a norm, complexity criterion, residual with respect to some set of constraints, or any of the previous with an added penalty term as used in regularization. The objective and penalty functions may operate on any combination or function of attributes of the output objects.

For ANNs, training is highly dependent on the type of model used, activation function used, format of the output used for calculation of training error, and any additional constraints imposed upon the model. Usually, the first decision used in determining the appropriate optimization algorithm to be used in model training is in the use of a differentiable objective function. Optimization algorithms that require a differentiable objective function are referred to as gradient methods while meta-heuristic optimization algorithms do not require a differentiable objective function.

The common theme amongst most optimization algorithms is that the dimension of the parameter vectors remain constant throughout the algorithm. Variants to an existing meta-heuristic algorithm are proposed which allow for varying lengths of parameter vectors to be compared. This allows for a population of neural networks with heterogeneous architectures to explore a parameter search space which has a non-constant dimension.

5.1 Gradient Methods

Gradient methods, should be the first methods considered due to their nice mathematical properties and ease of use. However, gradient methods may only be used when gradients of

the training error with respect to the parameters to be trained exist. When these gradients do not exist, it is best to use a meta-heuristic optimization method.

Some advantages to using gradient methods is that they have the nice properties such as guaranteed convergence to a single global minima as is the case when using L^2 error as the feedback mechanism. Also, gradient methods may be used with only a single optimization step that drastically decreases the amount of resources (both time and compute cycles) needed to perform the optimization. Additional steps may be performed by varying the initial starting location to increase algorithm robustness in finding a global minima. Some common gradient methods used in the optimization of ANN parameters are back-propagation (BP), Newton-Raphson, and conjugate gradient methods that each have their own advantages and disadvantages.

5.1.1 Back-Propagation

Back-Propagation (B-P) is the traditional ANN training method used in conjunction (usually) with an L^2 norm training error with respect to the model output and observed output. First the model is run in the forward direction to generate a model output. Then the L^2 error of this output with respect to the observed or desired output is calculated. Due to the nice properties of the L^2 norm, the gradients of this error with respect to the model parameters are calculated analytically. B-P is usually plagued with slow convergence.

5.1.2 Newton-Raphson

In some problems, the gradient along with the Hessian are known. Netwon-Raphson (N-R) method provides for quadratic convergence when the model parameters are "near" the optimal parameters. More information on this can be found in Stoer and Bulirsch (2013). An obvious drawback to using N-R is that not only does the first derivative of the feedback function need to exist, but also the second derivative.

5.1.3 Conjugate Gradient

When the number of parameters becomes large, the conjugate gradient (C-G) method may be more appropriate. C-G is an iterative method that converges in N iterations where N is the dimension of the problem (here represented as the total number of model parameters to be trained). As with gradient methods, C-G requires that the gradient of the training error exist with respect to all parameters to be trained.

5.2 Meta-Heuristic Optimization Methods

Due to the large dimensional parameter space, many local minima, and the non-differentiability of the output error function, derivative based optimization methods such as back-propagation are not ideally suited. Meta-heuristic optimization methods are considered to be more robust than gradient methods due to their use of many candidate models or solutions to simultaneously explore the search space. This search space is defined to be the Cartesian product of the domains of each parameter to be optimized subject to any additional constraints imposed.

Meta-heuristic optimization methods are most useful when a derivative of the objective function does not exist. In this case, the many models seeded by these methods are used to collectively explore the search space in an iterated fashion. These methods are also useful when the analytic expression for the gradient of the objective function is difficult or expensive to compute. A typical disadvantage to meta-heuristic optimization methods is that the dimensionality of the search space causes slow convergence speeds as compared to gradient methods. A problem involving 100 parameters that may take any real value implies that the search space is $\mathbb{R}^{100}$. Meta-heuristic optimization methods are broken into two major groups based upon the way that the candidate solutions explore the search space and are referred to as swarm algorithms and evolutionary algorithms.

5.2.1 Swarm Algorithms

Swarm algorithms use a state equation based upon the candidate solutions' attributes such as velocity, momentum, interaction with local fitness, interaction with other candidate solutions, and history of past "best" candidate solutions to update the candidate solution at each iteration.

Nelder-Mead Simplex Optimization

Nelder-Mead (N-M) is a deterministic optimization method where the candidate solutions form an N dimensional simplex in the search space where N is the number of parameters being optimized. The N-M simplex expands, reflects, and shrinks to explore the search space based upon pre-set coefficients within the algorithm. This algorithm typically involves fewer functions evaluations than other global optimization methods but is sensitive to the structure of the fitness function in the local search space. As such, N-M can become trapped in local minima and must be run several times with different starting locations to ensure convergence has been reached. Robustness of the N-M algorithm is increased by running the algorithm several times at random staring points and taking the minimum of all the runs.

Particle-Swarm Optimization

Particle swarm optimization (PSO) has been used effectively to perform high dimensional optimization over a variety of parameters with various constraints. PSO is performed much the same way as differential evolution in that many models are trained simultaneously with each model "learning" the search space based upon the exploration of the space by the collective. Pseudo code for the PSO algorithm is given in Algorithm 1 where g and b_p represent the model's global best solution and particle p's best solution respectively. The algorithm is started by seeding the values for each particle's position $p[\boldsymbol{x}]$ and velocity $p[\boldsymbol{v}]$, each of which have dimension equal to the number of parameters being optimized. Upper and lower bounds for the position xlb_i and xub_i with respect to the i^{th} dimension are chosen

by the user to approximately cover the solution space. The upper and lower bounds for the velocity vlb_i and vub_i are taken to be symmetric about zero to prevent an intentional drift in the particle population. Once the particle's have been initialized, the initial best solution for each particle is defined to be its initial point $\boldsymbol{x}$. The initial global best solution is taken to be the initial position of the particle that minimizes the objective function f. Next, until a stopping criterion is reached, each particle's velocity and position are updated based upon the equations given in steps 15 and 16. The global best solution and each particle's best solution are then updated at each iteration. PSO outputs the global best solution, which is the vector that minimizes the objective function.

PSO is not an appropriate optimization method for learning a non-constant process. This is due to the algorithm requiring the use of the global best fitness in all the past training history to update the particles at the next iteration. However, if the underlying process is constant, or even stationary, then PSO may be appropriate. A potential fix to PSO would be to use a time decay term on the global and local best fitness. This would impose a "limited memory" to the particles as they perform the search of the space.

Algorithm 1 Pseudo code for particle swarm optimization algorithm.

1: **procedure** MAIN:
2: $g \leftarrow empty$
3: **for** p **in** population **do**:
4: **for** i **in** N_{dim} **do**:
5: $p[x_i] \leftarrow random\ U[xlb_i, xub_i]$
6: $p[v_i] \leftarrow random\ U[vlb_i, vub_i]$
7: $b_p[x_i] \leftarrow p[x_i]$
8: **if** $f(p[\boldsymbol{x}]) < f(g)$ **then**:
9: $g \leftarrow p[\boldsymbol{x}]$
10: **for** iter **in** MaxIter **do**:
11: **for** p **in** population **do**:
12: **for** i **in** N_{dim} **do**:
13: $r_p \leftarrow random\ U[0, 1]$
14: $r_g \leftarrow random\ U[0, 1]$
15: $p[v_i] \leftarrow \omega p[v_i] + \phi_p r_p(b_p[x_i] - x_i) + \phi_g r_g(g[x_i] - p[x_i])$
16: $p[x_i] \leftarrow p[x_i] + p[v_i]$
17: **if** $f(p[\boldsymbol{x}]) < f(b_p)$ **then**:
18: $b_p \leftarrow p[\boldsymbol{x}]$
19: **if** $f(p[\boldsymbol{x}]) < f(g)$ **then**:
20: $g \leftarrow p[\boldsymbol{x}]$
return g

Artificial Bee Colony Optimization

Modeled after the behavior of honey bees, artificial bee colony optimization (ABCO) is similar to PSO in that random particles are seeded in the search space that keep track of both their local best and global best solutions. ABCO differs from PSO in that the particles are regenerated after a set number of iterations of searching in the neighborhood of a local best solution. These regenerated particles are then used to explore larger search neighborhoods allowing ABCO to adapt to changing model conditions and as such respond to a non-stationary process.

5.2.2 Evolutionary Algorithms

The second major group are evolutionary algorithms that generate new candidate solutions via some information exchange between some subset of the current candidate solutions known as parents and then evaluate the solutions' fitness.

Differential Evolution

Regarded as one of the most robust meta-heuristic optimization methods, differential evolution (DE) has been used in a wide variety of problem sets including ANN parameter optimization by Ilonen et al. (2003) and Slowik and Bialko (2008), non-linear programming by Tsai (2015), and airfoil design by Rai (2006). DE updates each parameter vector by randomly choosing three other vectors in the population and creates a new trial vector. This fitness of the new trial vector and the original vector is evaluated and used to select the "better" vector. Typical of global optimization methods, DE does not select the "best" new vector to update, but only makes incremental improvements.

A rule of thumb for the number of models to train in DE is approximately 10 times the number of parameters to be optimized. For this work the DE variant used is *DE/rand/1/bin* shown in Algorithm 2 where the global best solution is represented by g. For each particle in the population, three parent solutions are chosen randomly from the remainder of the population and used to generate a new trial vector p_{tmp}. The fitness of the original particle

p is evaluated against the "evolved" solution p_{tmp} via the objective function f. After evaluation, the particle p is updated with the new parameter vector p_{tmp} if the new parameter vector performs better with respect to the objective function f. The parameters C and F are user defined; a good choice in these parameters according to Vesterstrøm and Thomsen (2004) is 0.9 and 0.5 respectively.

Algorithm 2 Pseudo code for differential evolution algorithm.

```
 1: procedure MAIN:
 2:     g ← empty
 3:     f_g ← ∞
 4:     for iter in MaxIter do:
 5:         for p in population do:
 6:             p_1, p_2, p_3 ← choose random in population
 7:             for param in p do:
 8:                 r ← random U[0,1]
 9:                 if r < C then:
10:                     F ← random U[0,1]
11:                     p_tmp[param] ← p_1[param] + F * (p_2[param] − p_3[param])
12:                 else
13:                     p_tmp[param] ← p[param]
14:             if f(p_tmp) < f(p) then:
15:                 p ← p_tmp
16:             if f(p) < f(g) then:
17:                 g ← p
        return g
```

Genetic Algorithms

Genetic algorithms (GA) may be used in almost the exact same way that differential evolution except that vectors may be updated based upon two or more "parent" vectors and is not constrained to always use three as with DE. GA and DE only differ in the mechanism for updating the trial vector.

GA is broken into three main steps, the first being parent selection defined by some selection operator. This selection operator used in line 10 randomly chooses two parent vectors based upon their values S_p defined by the output of the objective function f. Parameter vectors that minimize the objective function are more selectively chosen at random from the population. This causes the population to evolve towards an overall "better"

solution. The second step within GA is referred to as "crossover" where segments of two offspring vectors are exchanged and happens with probability C. Lastly, "mutation" occurs with probability M where segments of the offspring parameter vectors are permuted. This process continues until the population's average value of S_p reaches some threshold or the maximum number of iterations is reached.

Algorithm 3 Pseudo code for genetic algorithms.

```
procedure MAIN:
    g ← empty
    for gen in MaxGen do:
        for p in population do:
            S_p ← f(p)
            if f(p) < f(g) then:
                g ← p
        offspring ← empty
        while length(offspring) < length(population) do:
            p_1, p_2 ← Select(2)
            r_c ← random U[0, 1]
            if r_c < C then:
                i ← randint[1, NumParams]
                p_tmp ← p_1
                p_1[1 : i] ← p_2[1 : i]
                p_2[1 : i] ← p_tmp[1 : i]
            r_m ← random U[0, 1]
            if r_m < M then:
                i_1, i_2 ← randint[1, NumParams]
                p_1[i_1 : i_2] ← randperm(p_1[i_1 : i_2])
                i_1, i_2 ← randint[1, NumParams]
                p_2[i_1 : i_2] ← randperm(p_2[i_1 : i_2])
            offspring ← p_1, p_2
    return g
```

Information Exchange Rate

Evolutionary algorithms require selection of parent vectors from the population of models being run are somewhat limited by the rate that information from the parent vectors propagates to the entire population. A Monte Carlo simulation was conducted to illustrate the number of iterations expected for information to propagate to the entire population. Figure 5.1 shows that there is a linear relationship equation (5.1) between the number of models in the population and the time needed for information propagation.

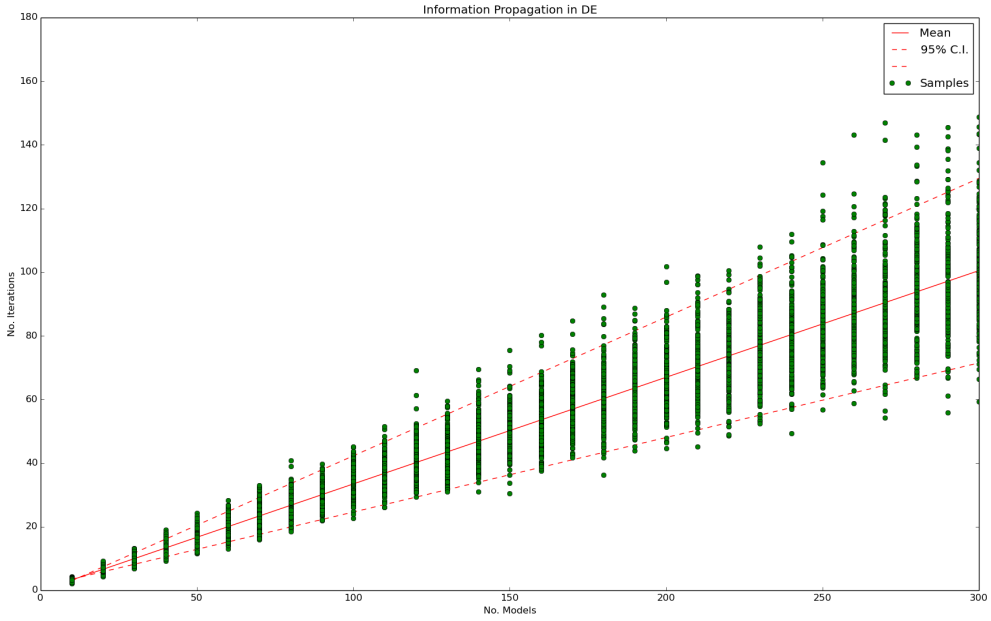


Figure 5.1: Information exchange rate for three parent vectors in evolutionary global optimization algorithms simulated by Monte Carlo.

$$T \sim \mathsf{N}\left(0.333t + 0.100, (0.054t - 0.828)^2\right) \tag{5.1}$$

This simulation was performed for algorithms that require three parent vectors to create a new trial vector for evaluation of fitness such as DE. Algorithms that allow for more than three parent vectors such as GA will have a higher rate of information propagation due to the higher likelihood that one of the models that has already been updated will be selected. This propagation rate is also influenced by the parameters within the respective algorithm that govern the selection rate of trial vector components to be updated based upon the attributes of the parent vector.

5.2.3 Selecting an Optimization Algorithm

It was shown in Section 3.4.4 that the K-S test statistic is a valid objective function, and was decided to be the objective function used in the remainder of this dissertation. Due

to the K-S test's use of the inverse CDF of the underlying distribution the gradient of this objective function with respect to the ANN hidden node weights does not exist. Due to the gradient not existing, it is not possible to use a gradient method such as back-propagation. Therefore, a derivative-free method such as Nelder-Mead simplex method, particle swarm optimization, genetic algorithms, or differential evolution must be used.

While Dindar and Marwala (2004) and Dhahri et al. (2012) have shown that PSO and DE respectively are capable of being used to train ANN weights, I have excluded PSO from consideration due to its use of a global best solution. Within PSO, the global best solution is used to influence all particle's paths at each time step. If the underlying process is non-stationary, then the global best solution is no longer valid due to it being "best" for a process that is no longer valid. Also, it has been suggested by Vesterstrøm and Thomsen (2004) that evolutionary algorithms such as DE are more robust than PSO in a variety of optimization problems.

The remainder of this dissertation will focus on the meta-heuristic optimization algorithm subclass of evolutionary algorithms and in particular will focus on differential evolution. Differential evolution is chosen specifically due to the existence of an algorithm variant that was designed to handle parameter vectors of varying lengths.

5.3 Multidimensional Evolutionary Algorithms (MD-EA)

The dimension of the search space in optimization algorithms is determined by the number of parameters to be optimized, however this requires that the number of parameters to be optimized be known. When the dimension of the search space is unknown the typical approach is to perform an exploratory search by testing various dimensions and configurations of the parameters to determine an appropriate dimension and configuration to use as the final approach. When using evolutionary algorithms, this approach must be performed prior to the use of the optimization algorithm since each method requires that each parent vector have the same dimension during information exchange.

An algorithm was proposed by Dhahri et al. (2012) that alleviates this constraint within

the DE algorithm. However, by adding a level of abstraction, this algorithm may be applied to any evolutionary algorithm. In addition, three variants are proposed that serve to explore additional methods based upon the ontology presented in Chapter 4. This generalized algorithm as applied to any of the evolutionary algorithms is depicted in Figure 5.2. Steps A, B, D, and F are identical to the traditional evolutionary algorithms for homogeneous neural networks. When the population of neural networks becomes heterogeneous, it is necessary to first homogenize the parent vectors prior to the information exchange step shown as step D. After the information exchange, the artificial changes made to the candidate solution are removed so that fitness evaluation may occur. To adapt the algorithm shown in Figure 5.2 to DE or GA only requires specific changes in the information exchange step D.

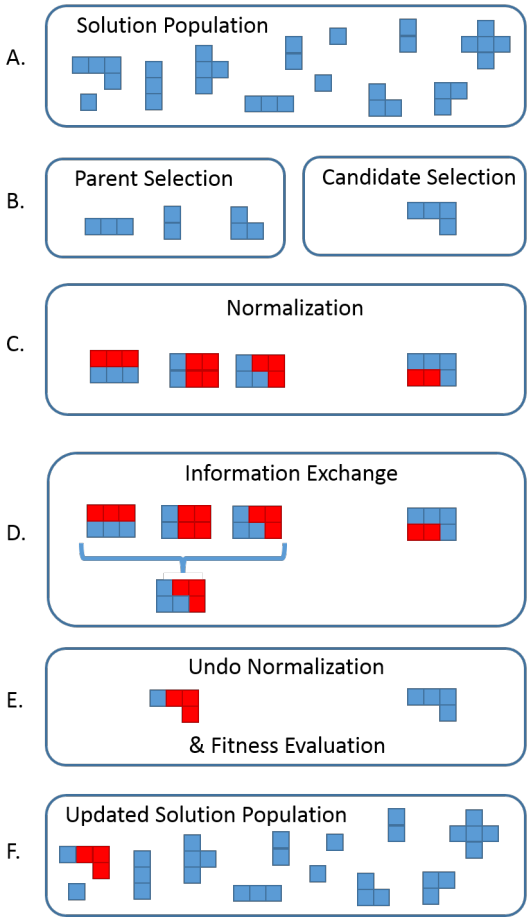


Figure 5.2: Algorithm steps within general multidimensional evolutionary algorithm.

The MD-EA algorithms are not appropriate for swarm optimization methods due the nature of how the solution dimension is rectified and the possible dependence on a global solution over all previous iterations. This same issue allows the MD-EA algorithms to work on time-dependent inputs or non-stationary processes.

5.3.1 Original MD-DE Algorithm

An algorithm was created by Dhahri et al. (2012) to allow for DE to be used to update trial parameter vectors when training a population of heterogeneous neural networks. Based upon the ontology described in Chapter 4, the proposed MD-DE algorithm performs an additive projection of the trial and parent parameter vectors before the information exchange occurs.

This proposed algorithm has a potential convergence hangup in certain cases. Suppose there is a population of single layer neural networks such that all but one model has N hidden nodes and the last model has $N + 1$ hidden nodes. In this case, when a trial vector is being generated for the larger model, all the smaller models' parameter vectors are padded with zeros. This prevents the parameters for the last node in the larger model from being updated since all trial vectors will have zeros in the last positions. In this case, the node that makes the largest model the largest is unable to be updated and degrades the overall model's performance with respect to a criterion such as the Akaike information criteria (AIC). AIC is a metric used to evaluate a model's accuracy with respect to the model's complexity.

In the case that all the ANNs within the population have the same architecture, this algorithm reduces to the classical DE algorithm for optimizing parameters within an ANN. Several variants to this algorithm are proposed that explore other potential approaches to normalize parameter vector length during ANN training. Two of the proposed variants also serve to rectify the issue of having models with no potential for being updated.

5.3.2 Proposed MD-EA Algorithm Variants

Using the same approach as the original MD-DE algorithm, the first variant proposed is essentially the opposite of the original algorithm. Instead of taking the additive projection of the parent vectors to achieve normalized vector lengths, a subtractive projection may be used. This truncates the trial and parent vectors to the length defined by the smallest of the four models being used in the update of that single model. It is important to note that for optimizing parameters within ANNs, the truncated parameters are not simply taken from the end of the parameter vector. Instead it is the parameters for the last node(s) in the models that are temporarily removed. Depending on the numbering scheme used and the complexity of the node connections, the parameters for the removed node may be spread throughout the parameter vectors.

However, this proposed variant using the subtractive projection is subject to the same theoretical issues as when using the additive projection. Taking the same proposed population of ANN models as before, suppose there is a population of single layer neural networks such that all but one model has N hidden nodes and the last model has $N+1$ hidden nodes. In this case, when a trial vector is being generated for the larger model, the parameters associated with the last node will always be truncated during the information exchange. This prevents the parameters for the last node in the larger model from being updated since it will always be truncated.

To rectify this issue, two additional variants are proposed, one for the additive projection case and the second for the subtractive projection case. Instead of always appending or truncating the last node on the smaller or larger models respectively, randomly chose the location to perform this operation. This effectively randomizes the node parameters that are always appended with zeros or removed during the information exchange. Using the proposed ANN model population as mentioned before with single layer neural networks such that all but one model has N hidden nodes and the last model has $N + 1$ hidden node. In the additive case, when the trial vector is being created for the largest model, zeros are appended on the smaller models in a randomly chosen node location that allows

for information exchange in the trial vector. Similarly in the subtractive case, a node in the largest model is chosen at random during each update iteration to be truncated. This allows the for a random subset of N nodes within the $N + 1$ nodes to be updated at each iteration. The two proposed variants for the random selection of nodes in the additive and subtractive cases are performing random additive and random subtractive projections respectively based upon the proposed ontology. A key point is that all three of the proposed algorithms reduce to the original DE algorithm for optimization of ANN parameters for homogeneous networks.

5.4 Comparison of MD-EA Algorithm Variants

Six test functions F_i show in equations (5.3a) - (5.3f) and plotted in Figure 5.3 are used to evaluate the performance of the original MD-DE algorithm and the three MD-DE variants over the unit square $[-1, 1]\,\mathrm{x}\,[-1, 1]$. The first 5 functions listed in equations 5.2 - 5.7 are taken to be the functions 1, 2, 4, 6, and 8 from Table 1 in Vesterstrøm and Thomsen (2004). Function 6 was created to demonstrate algorithm performance against a function near a singularity.

The models are all run using a random number generator seed equal to the trial number. For each run, there are 40 trials averaged together. Since the random seed is equal to the trial number, each algorithm's network has the same architecture across all algorithm runs. The DE variant *DE/rand/1/bin* is used to perform the information exchange after the trial and parent vector lengths have been normalized.

$$F_1(x_1, x_2) = x_1^2 + x_2^2 \tag{5.2}$$

$$F_2(x_1, x_2) = |x_1| + |x_2| + x_1 x_2 \tag{5.3}$$

$$F_3(x_1, x_2) = \max(x_1, x_2) \tag{5.4}$$

$$F_4(x_1, x_2) = \left(\left\lfloor x_1 + \frac{1}{2} \right\rfloor\right)^2 + \left(\left\lfloor x_2 + \frac{1}{2} \right\rfloor\right)^2 \tag{5.5}$$

$$F_5(x_1, x_2) = -x_1 \sin\left(\sqrt{|x_1|}\right) - x_2 \sin\left(\sqrt{|x_2|}\right) \tag{5.6}$$

$$F_6(x_1, x_2) = \exp\left(-\sqrt{x_1^2 + x_2^2}\right) \tag{5.7}$$

For single and then multi-layer networks, converge plots are generated to show rates of convergence. Then the performance of the algorithms is explored by varying first the number of nodes and then the number of models while being evaluated at 1000 iterations. Each iteration is performed by generating 100 sample points within the square $[-1, 1]\,\mathrm{x}\,[-1, 1]$ and using these 100 tuples as ANN model inputs. The model output is compared against the value of the appropriate test function at each of the generated sample locations using the L^2 error. It can be expected that there is an accuracy limit in the model process based upon the stochastic nature of the sample points. As the number of sample points evaluated at each iteration increases, it can be expected that the average L^2 error will decrease.

5.4.1 Homogeneous Single Layer Networks

As a starting point, a feed forward MLP neural network is evaluated using the original differential evolution algorithm using 40 homogeneous models. The models are run using 2, 3, 4, and 5 nodes first evaluated after 1, 10, 10^2, 10^3, and 10^4 iterations to illustrate the base algorithm convergence rate. Then the number of models used in the optimization is varied from 10 to 100 in increments of 5 and evaluated after 1000 iterations for each of the 6 proposed test functions. These results are used to compare the subsequent evaluation of

MD-DE performance.

For constant number of nodes, vary the number of models used in each model run. Each point is the average of 40 trial runs with a different random seed for node weights learned on the respective test functions.

5.4.2 Heterogeneous Single Layer Networks

To compare the algorithms' performance, first a convergence plot is generated to illustrate the convergence of each algorithm with respect to the six test functions. Each point is taken to be the average of 40 realizations of model performance evaluated at 1, 10, 10^2, 10^3, and 10^4 iterations with 40 underlying models. For each model and each realization, the number of hidden nodes is taken to be one plus a sample from a Poisson distribution with mean of 2. This simulates a heterogeneous population of viable model sizes.

Next, the number of seeded nodes is varied by changing the mean of the Poisson distribution that is used in the generation of the model sizes. Again, the points are taken to be the average of 40 realizations of 40 models each evaluated after 1000 iterations.

Lastly, the number of seeded models is varied from 10 to 100 in increments of 5. Each model and each realization has the number of hidden nodes to one plus a sample from a Poisson distribution with mean 2.

It can be seen from the Figures 5.6, 5.7, and 5.8 that the random additive and random subtractive algorithms perform the best. The algorithms converge more slowly as the number of nodes is increased, this is due to the increase in the number of optimized parameters (which increases the dimension of the search space). The number of models in the model also decreases the convergence rate due to the mechanism that governs the information propagation within the DE algorithm. Figure 5.1 illustrates the speed of information propagation within the DE algorithm. It can be seen from the figure that the expected time for new information to spread to all the models within the algorithm increases linearly with the number of models.

5.4.3 Multi Layer Networks

Before comparisons for multi-layer networks can be make, a baseline performance of the DE algorithm is run with homogeneous networks.

The same simulations as run for single layer models are now repeated for multi layer models. In particular, a two layer model is used with constant first layer architecture and stochastic second layer. Figures 5.11, 5.12, and 5.13 are the multi layer analogues of Figures 5.6, 5.7, and 5.8 for single layer models. To compare the algorithms' performance, first a convergence plot is generated to illustrate the convergence of each algorithm with respect to the six test functions. Each point is taken to be the average of 40 realizations of model performance evaluated at 1, 10, 10^2, 10^3, and 10^4 iterations with 40 uderlying models. For each model and each realization, the number of hidden nodes is taken to be one plus a sample from a Poisson distribution with mean of 2. This simulates a heterogeneous population of viable model sizes.

Next, the number of seeded nodes is varied by changing the mean of the Poisson distribution that is used in the generation of the model sizes. Again, the points are taken to be the average of 40 realizations of 40 models each evaluated after 1000 iterations.

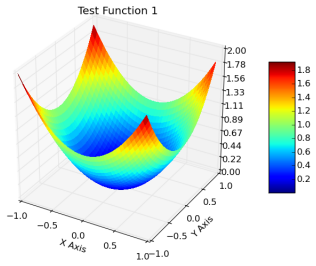
Lastly, the number of seeded models is varied from 10 to 100 in increments of 5. Each model and each realization has the number of hidden nodes to one plus a sample from a Poisson distribution with mean 2.

It can be seen from the Figures 5.11, 5.12, and 5.13 that the random additive and random subtractive algorithms perform the best. The algorithms converge more slowly as the number of nodes is increased, this is due to the increase in the number of optimized parameters (which increases the dimension of the search space). The number of models in the model also decreases the convergence rate due to the mechanism that governs the information propagation within the DE algorithm. Figure 5.1 illustrates the speed of information propagation within the DE algorithm. It can be seen from the figure that the expected time for new information to spread to all the models within the algorithm increases linearly with the number of models.

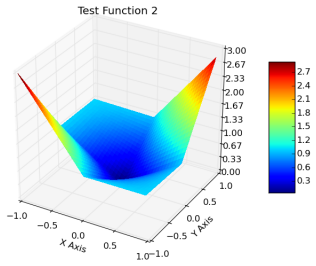
5.4.4 MD-EA Variant Selection

From the figures in the previous section it can be seen that the additive projection algorithm performs better than the three proposed variants in almost all cases for single layer models. For two layer models, the random additive projection algorithm performs better for some of the test functions. Taking into account the difference in convergence rates, it is decided that further work will be performed using the random additive algorithm. Although the original MD-DE algorithm performed better in almost every test, the appeal of having an algorithm that maximizes the use of all available degrees of freedom, albeit at a slower rate.

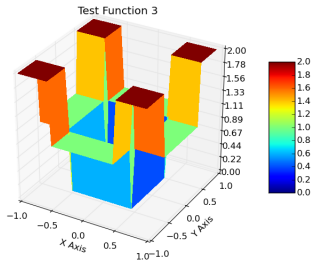
Also, due to the slower convergence rate of the two layer models, it is decided that further work will only use single layer models. There is an added benefit in that single layer models are less computationally expensive.



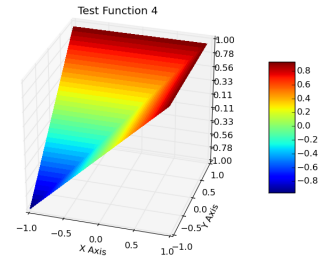
(a) Comparison against F_1



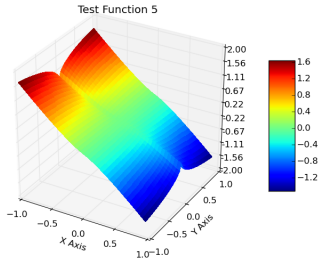
(b) Comparison against F_2



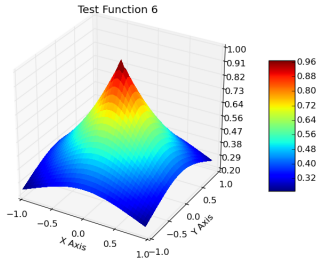
(c) Comparison against F_3



(d) Comparison against F_4

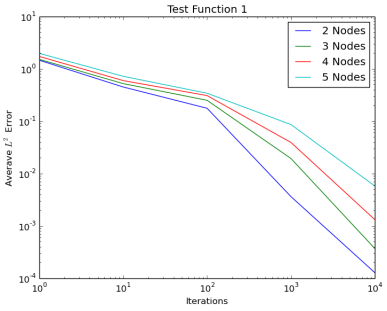


(e) Comparison against F_5

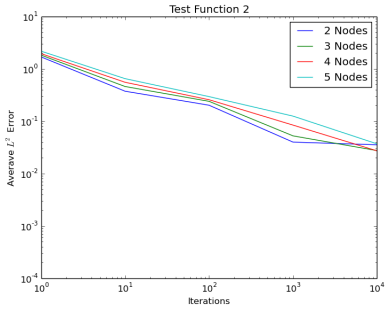


(f) Comparison against F_6

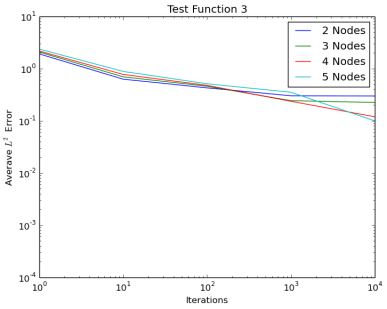
Figure 5.3: Test functions from Equations 5.2 - 5.7 over the domain $[0, 1]\,x\,[0, 1]$ used to evaluate original DE, original MD-DE, and proposed MD-DE variant algorithms for single and multi-layer neural networks.



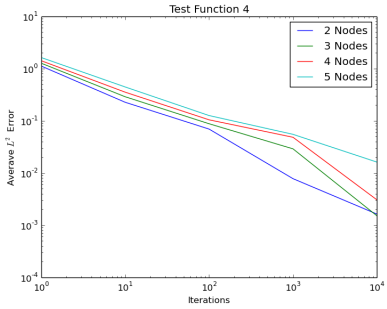
(a) Comparison against F_1



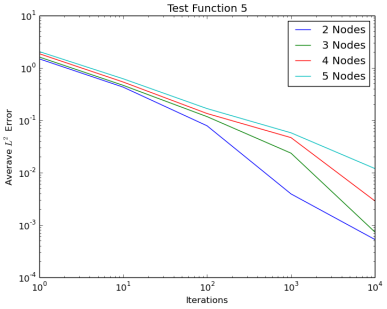
(b) Comparison against F_2



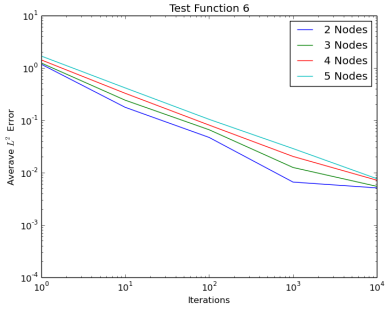
(c) Comparison against F_3



(d) Comparison against F_4

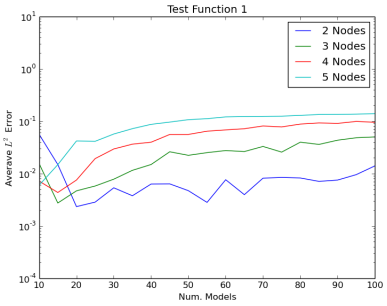


(e) Comparison against F_5

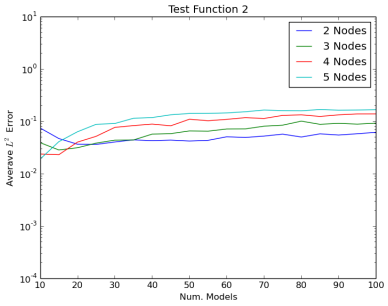


(f) Comparison against F_6

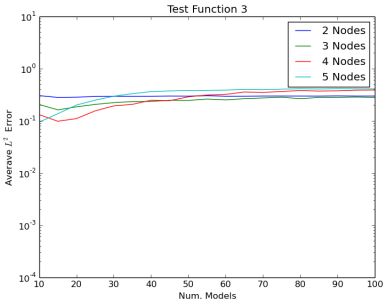
Figure 5.4: Homogeneous model performance averaged over 40 realizations evaluated at different points during the learning to evaluate convergence rates. Each curve represents the performance of models with different numbers of hidden nodes.



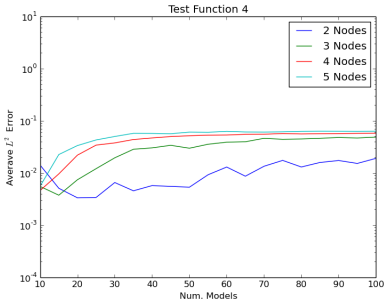
(a) Comparison against F_1



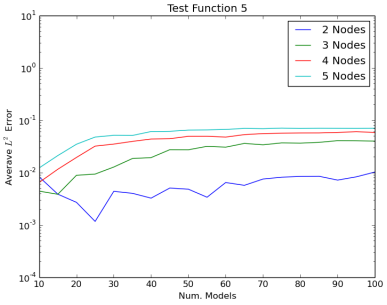
(b) Comparison against F_2



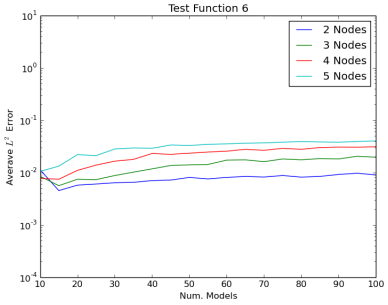
(c) Comparison against F_3



(d) Comparison against F_4

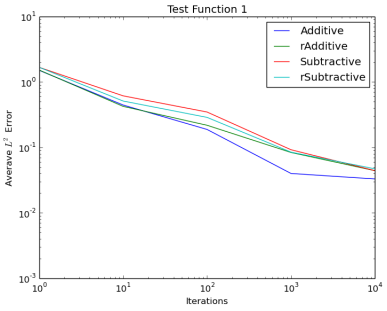


(e) Comparison against F_5

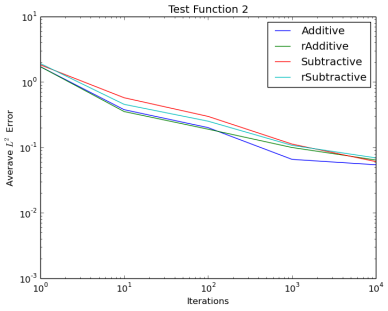


(f) Comparison against F_6

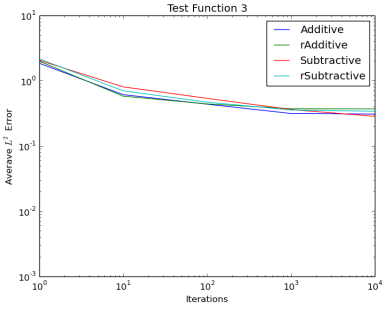
Figure 5.5: Homogeneous model performance averaged over 40 realizations evaluated after 1000 training iterations with different numbers of models. Each model run uses a constant number of hidden nodes defined by each curve.



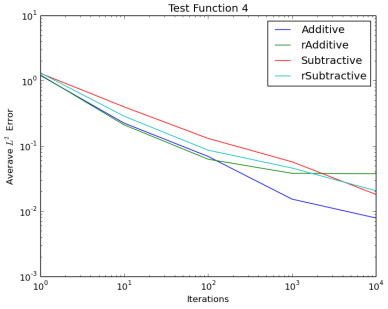
(a) Comparison against F_1



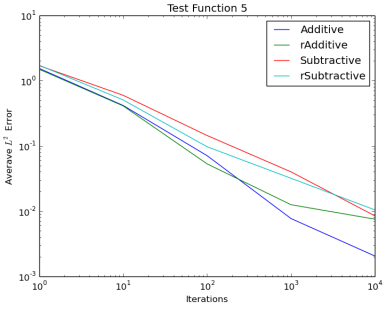
(b) Comparison against F_2



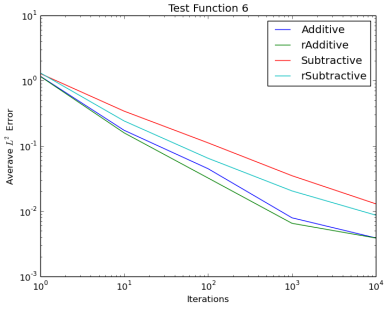
(c) Comparison against F_3



(d) Comparison against F_4

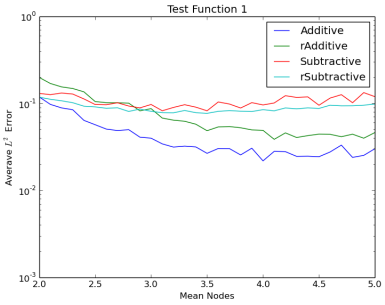


(e) Comparison against F_5

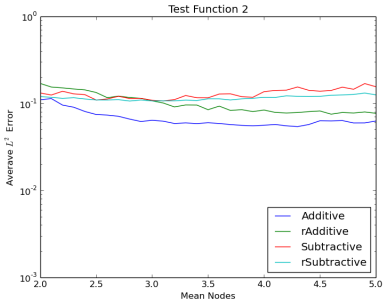


(f) Comparison against F_6

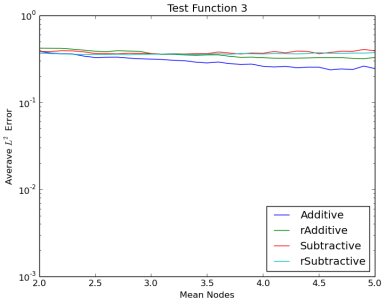
Figure 5.6: Heterogeneous single layer networks where the number of iterations is run from 1 to 10,000. The number of hidden layer nodes is taken to be one plus a sample from a Poisson distribution with mean of 2. 40 models are used in each run and realization.



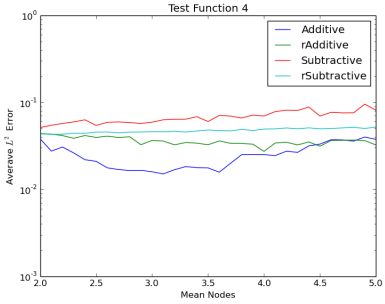
(a) Comparison against F_1



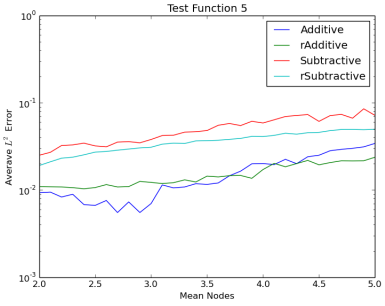
(b) Comparison against F_2



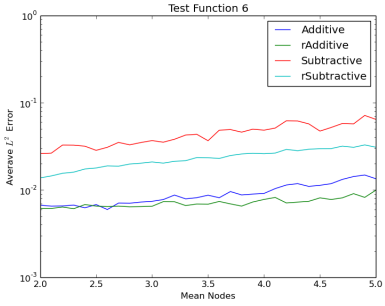
(c) Comparison against F_3



(d) Comparison against F_4

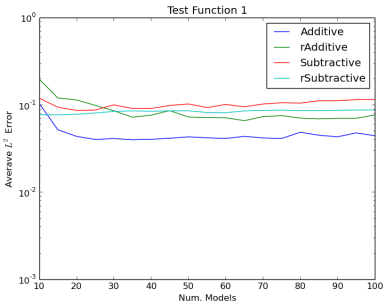


(e) Comparison against F_5

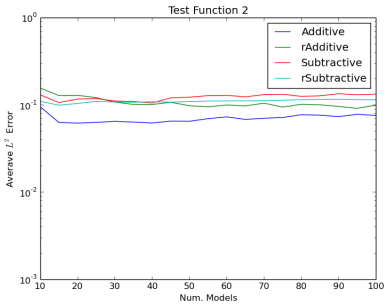


(f) Comparison against F_6

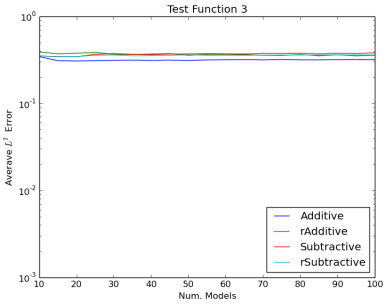
Figure 5.7: Heterogeneous single layer networks where the number of hidden layer nodes are varied based upon one plus a sample from a Poisson distribution. The mean of this Poisson distribution was varied from 1 to 4 in increments of 0.1. 40 models are used and evaluated after 1000 iterations in all cases.



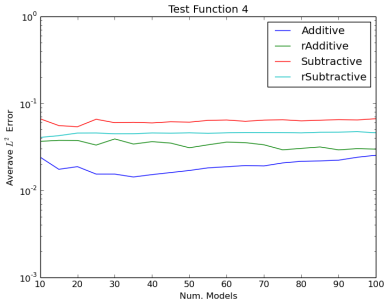
(a) Comparison against F_1



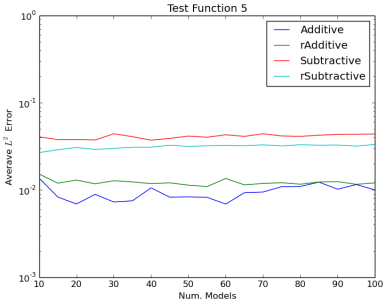
(b) Comparison against F_2



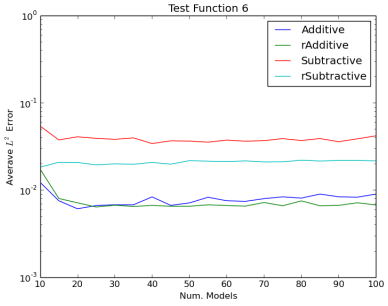
(c) Comparison against F_3



(d) Comparison against F_4

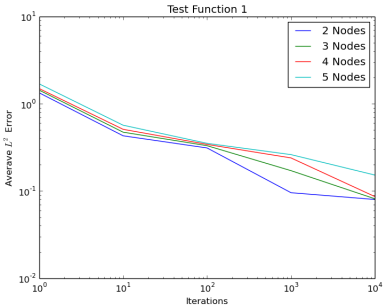


(e) Comparison against F_5

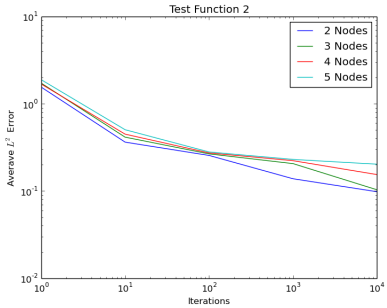


(f) Comparison against F_6

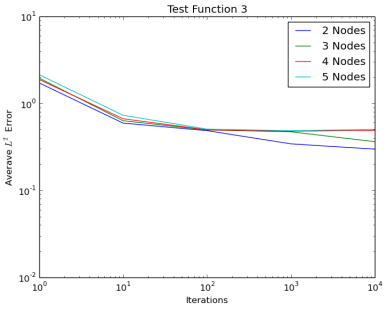
Figure 5.8: Heterogeneous single layer networks where the number of models are varied from 10 to 100 in increments of 5. The number of hidden layer nodes is taken to be one plus a sample from a Poisson distribution with mean of two. 40 models are used and evaluated after 1000 iterations in all cases.



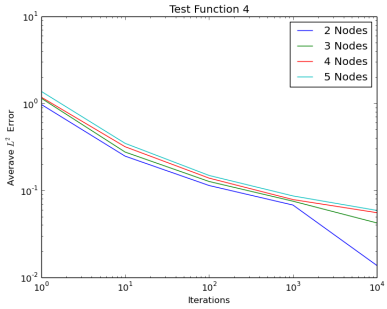
(a) Comparison against F_1



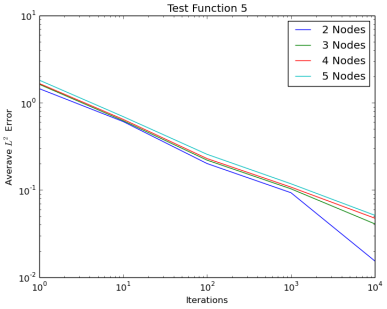
(b) Comparison against F_2



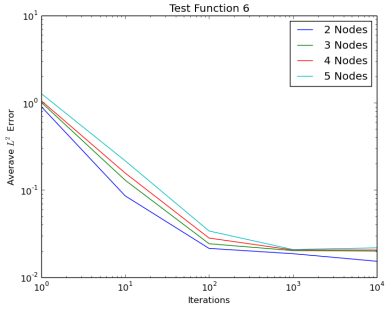
(c) Comparison against F_3



(d) Comparison against F_4

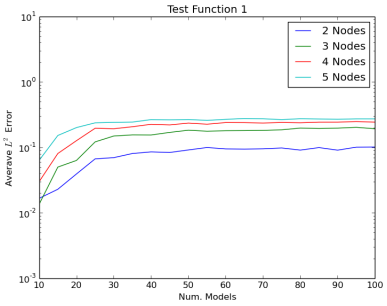


(e) Comparison against F_5

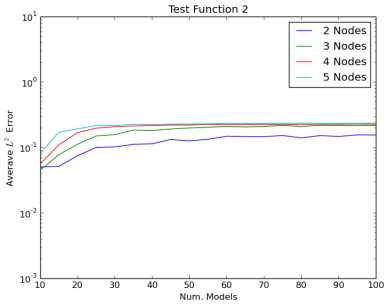


(f) Comparison against F_6

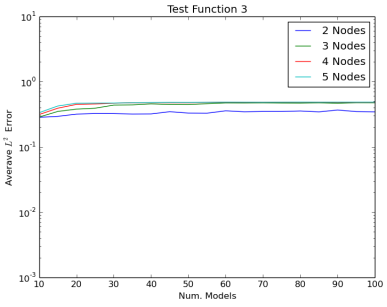
Figure 5.9: Homogeneous model performance averaged over 40 realizations evaluated at different points during the learning to evaluate convergence rates. Each curve represents the performance of models with different numbers of hidden nodes.



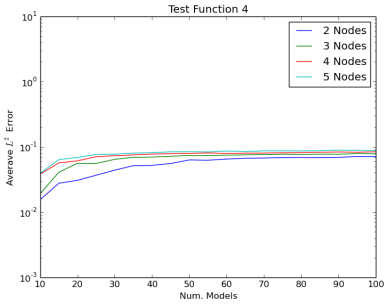
(a) Comparison against F_1



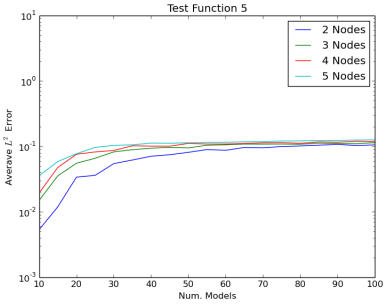
(b) Comparison against F_2



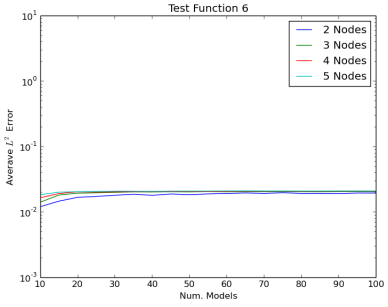
(c) Comparison against F_3



(d) Comparison against F_4

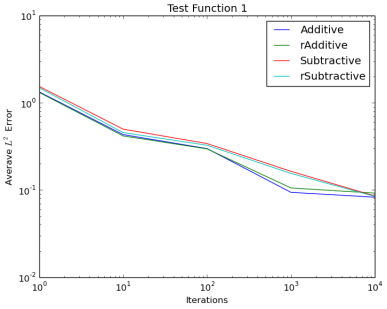


(e) Comparison against F_5

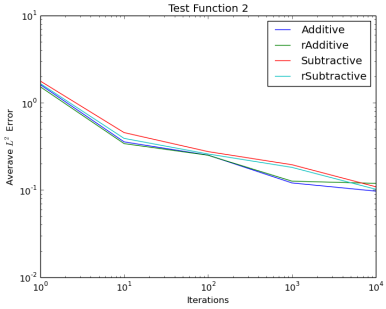


(f) Comparison against F_6

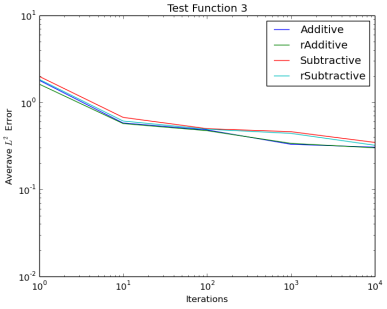
Figure 5.10: Homogeneous model performance averaged over 40 realizations evaluated after 1000 training iterations with different numbers of models. Each model run uses a constant number of hidden nodes defined by each curve.



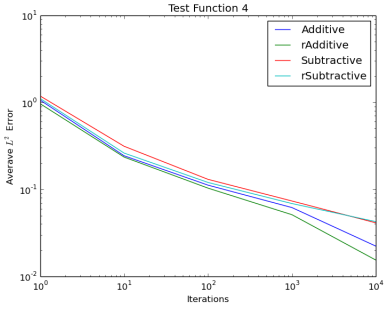
(a) Comparison against F_1



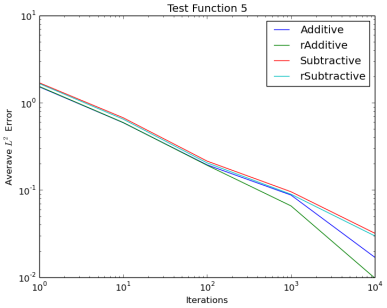
(b) Comparison against F_2



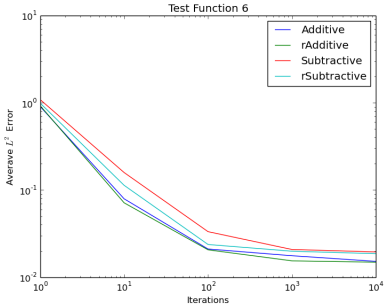
(c) Comparison against F_3



(d) Comparison against F_4

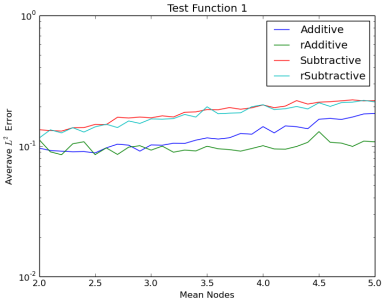


(e) Comparison against F_5

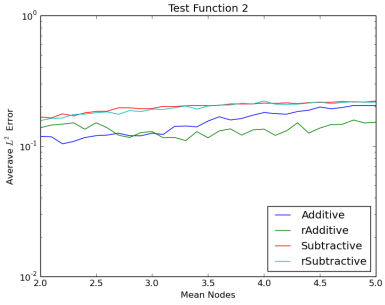


(f) Comparison against F_6

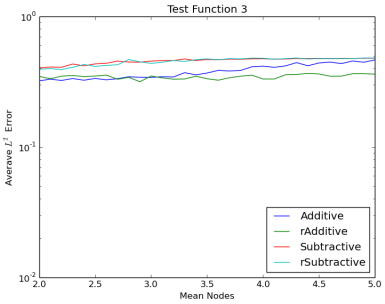
Figure 5.11: Heterogeneous two layer networks where the number of iterations is run from 1 to 10^4. The number of hidden layer nodes in the first layer is two, and for the second layer is taken to be one plus a sample from a Poisson distribution with mean of 2. 40 models are used in each run and realization.



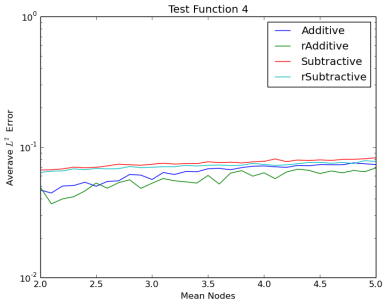
(a) Comparison against F_1



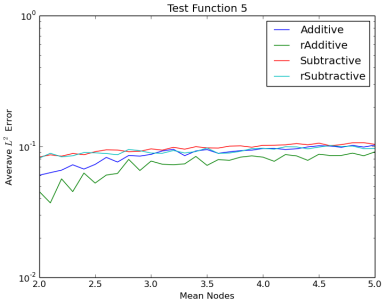
(b) Comparison against F_2



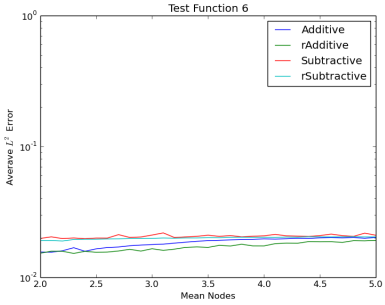
(c) Comparison against F_3



(d) Comparison against F_4

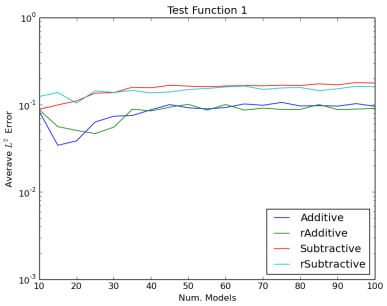


(e) Comparison against F_5

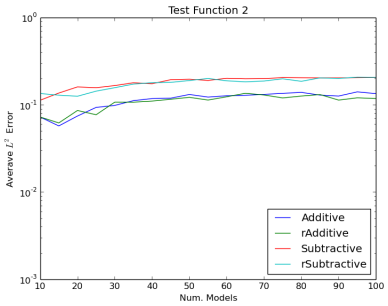


(f) Comparison against F_6

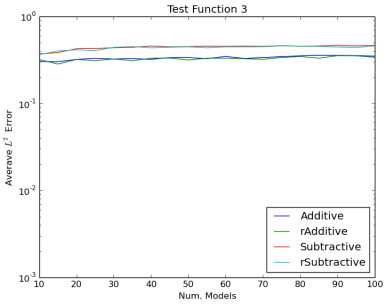
Figure 5.12: Heterogeneous two layer networks where the number of hidden nodes in the first layer is kept constant at two and the number of hidden nodes in the second layer is varied based upon one plus a sample from a Poisson distribution. The mean of this Poisson distribution was varied from 1 to 4 in increments of 0.1. 40 models are used and evaluated after 1000 iterations in all cases.



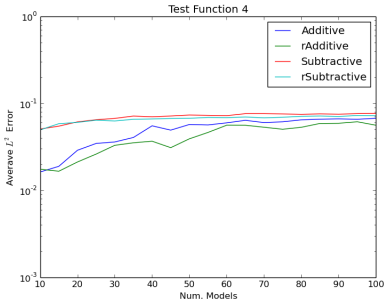
(a) Comparison against F_1



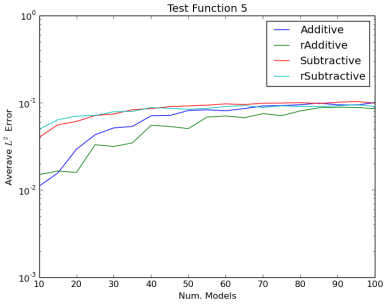
(b) Comparison against F_2



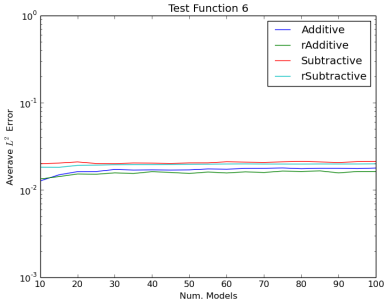
(c) Comparison against F_3



(d) Comparison against F_4



(e) Comparison against F_5



(f) Comparison against F_6

Figure 5.13: Heterogeneous two layer networks where the number of models are varied from 10 to 100 in increments of 5. The number of hidden layer nodes in the first layer is two, and for the second layer is taken to be one plus a sample from a Poisson distribution with mean of 2. 40 models are used in each run and realization.

Chapter 6: Calculation of Option Prices

Given a time series representing an asset's price over time, it is possible to estimate the value of options for that asset. These option prices should be consistent with the underlying time series from which they were directly estimated. This dissertation focuses on replacing that given time series with an ANN that is capable of forecasting the asset price at the next time interval. Simulated asset price paths are used to train the ANNs. Simulated prices are used since the underlying dynamics of the model are known exactly and can be used to generate "true" option prices. Comparisons will be made between the estimated option prices from the learned ANN models and the true option prices. If the estimated and true option prices are identical, then the ANNs are a consistent model of the asset dynamics.

6.1 European Prices

European style options based upon an underlying GBM process follow a well known analytical pricing formula known as the Black-Scholes formula developed by Black and Scholes (1973). The Black-Scholes formula for pricing calls and puts is shown in equations (6.3) and (6.4) respectively where C and P are the option prices in units of dollars. The parameters S, K, σ, r, T, and δ are the underlying stock price at time T_0, strike price, volatility, risk-free interest rate, time until expiration, and continuous dividend rate of the underlying asset. Formulas for d_1 and d_2 are given in equations (6.1) and (6.2) respectively using the same parameters S, K, σ, r, T, and δ as equations (6.3) and (6.4).

$$d_1 = \frac{1}{\sigma\left(T-t\right)}\left[\ln\left(\frac{S}{K}\right)+\left(r-\delta+\frac{1}{2}\sigma^2\right)\left(T-t\right)\right] \tag{6.1}$$

$$d_2 = d_1 - \sigma\sqrt{T-t} \tag{6.2}$$

$$C\left(S,K,\sigma,r,T,\delta\right) = Se^{-\delta T}N\left(d_1\right) - Ke^{-rT}N\left(d_2\right) \tag{6.3}$$

$$P\left(S,K,\sigma,r,T,\delta\right) = Ke^{-\delta T}N\left(-d_1\right) - Se^{rT}N\left(-d_2\right) \tag{6.4}$$

Prices of European style puts and calls are related to each other through the principle of Put-Call Parity shown in equation (6.5) that uses the same initial stock price S_0, strike price K, risk-free interest rate r, time to expiration T and continuous interest rate δ as the Black-Scholes formula. This formula can be used to calculate the price of a put given an equivalent call or vice-versa.

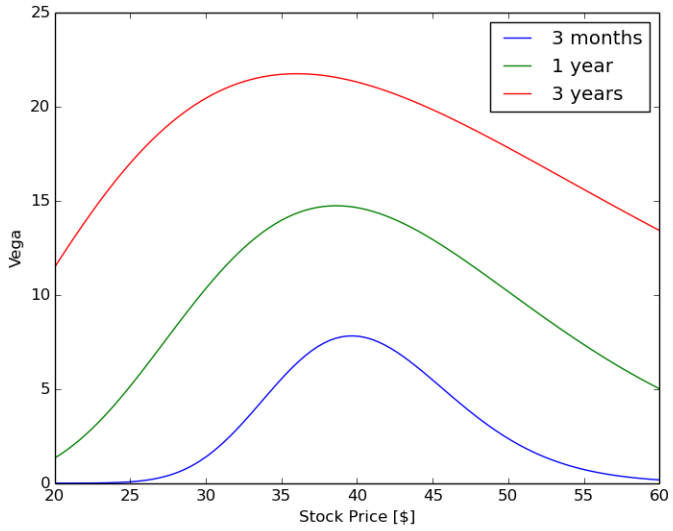
$$C_{Eur} - P_{Eur} = S_0e^{-\delta T} - Ke^{-rT} \tag{6.5}$$

Using the Black-Scholes formula, partial derivatives of the option price with respect to the parameters S, K, σ, r, T, and δ gives what are known as "The Greeks". The partial derivative of an European style call and put option with respect to the volatility σ is known as Vega and expressed in equations 6.6 and 6.7 respectively. This Greek represents the continuous rate of change in option price as volatility is varied. It can be seen that Vega is identical for both calls and puts all other things being equal. A range of values for Vega are shown in Figure 6.1 as plotted in McDonald (2013).

$$\frac{\partial C}{\partial \sigma} = S \exp\left(-\delta\left(T-t\right)\right) N'\left(d_1\right)\sqrt{T-t} \tag{6.6}$$

$$\frac{\partial P}{\partial \sigma} = \frac{\partial C}{\partial \sigma} \tag{6.7}$$

Figure 6.1: Vega for call option over range of initial price S_0.



The partial derivative of a European style call and put option with respect to the risk-free interest rate r is known as Rho and expressed in equations 6.8 and 6.9 respectively. This Greek represents the continuous rate of change in option price as the risk-free interest rate is varied. A range of values for Rho are shown in Figure 6.2 as plotted in McDonald (2013). Values for Rho are typically scaled by a factor if 100 since the risk-free interest rate is usually expressed as a percentage.

$$\frac{\partial C}{\partial r} = \left(T-t\right) K \exp\left(-r\left(T-t\right)\right) N\left(d_2\right) \tag{6.8}$$

$$\frac{\partial P}{\partial r} = \left(T-t\right) K \exp\left(-r\left(T-t\right)\right) N\left(-d_2\right) \tag{6.9}$$

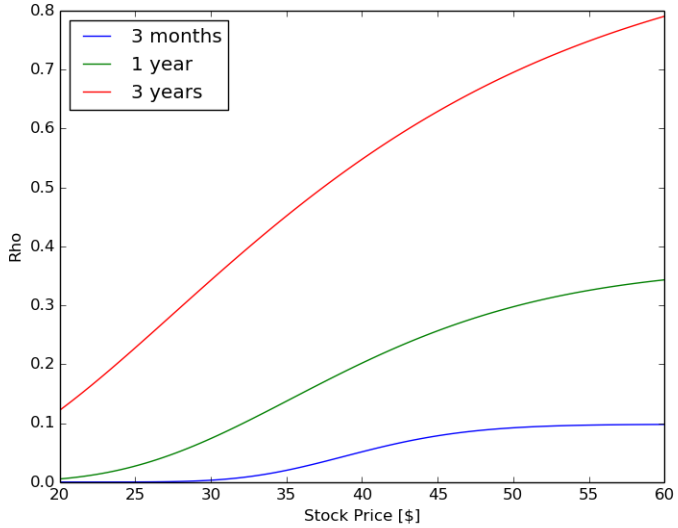


Figure 6.2: Rho for call option over range of initial price S_0.

The partial derivative of a European style call and put option with respect to the continuous dividend is known as Psi and expressed in equations 6.10 and 6.11 respectively. This Greek represents the continuous rate of change in option price as the continuous dividend rate is varied. A range of values for Psi are shown in Figure 6.3 as plotted in McDonald (2013). Values for Psi are typically scaled by a factor if 100 since the continuous derivative rate is usually expressed as a percentage.

$$\frac{\partial C}{\partial \delta} = -(T-t)\, S \exp\left(-\delta\,(T-t)\right) N\left(d_1\right) \tag{6.10}$$

$$\frac{\partial P}{\partial \delta} = (T-t)\, S \exp\left(-\delta\,(T-t)\right) N\left(-d_1\right) \tag{6.11}$$

Taking a slightly smaller range of initial stock price S_0 (for the purposes of illustration), Figure 6.4 shows the Black-Scholes analytic prices for European style options in solid lines and LSM estimated prices for American style options in dashed lines for a call option. It can be seen from this figure that while there is little difference between European and American

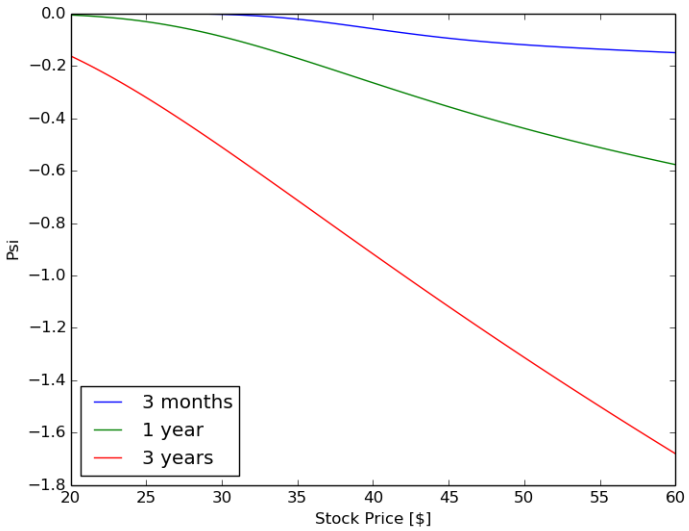


Figure 6.3: Psi for call option over range of initial price S_0.

style option prices, there is approximately one order of magnitude change in option price as the stock price changes.

These Greeks and how they change as stock price is changed illustrate the importance of correct parameter estimation in calculation of option prices; more importantly the sensitivity of option price is shown.

6.2 Bermudian and American Prices

Taking an European style option contract as one extreme of the spectrum of possible exercise times, an additional finite number of discrete exercise times may be appended to the contract. Contracts characterized by this finite number of discrete (usually evenly spaced in time or event dependent) exercise times are known as Bermudian options. Let h_t be the maximum time difference between all possible exercise times of a Bermudian option, the limit as h_t approaches zero leads to what is known as an American style option contract.

American style option contracts grant the contract holder the ability to exercise the contract at any time up to and including the option expiration time. The American style option contract writer is thus obligated to buy or sell the underlying asset at the pre-defined

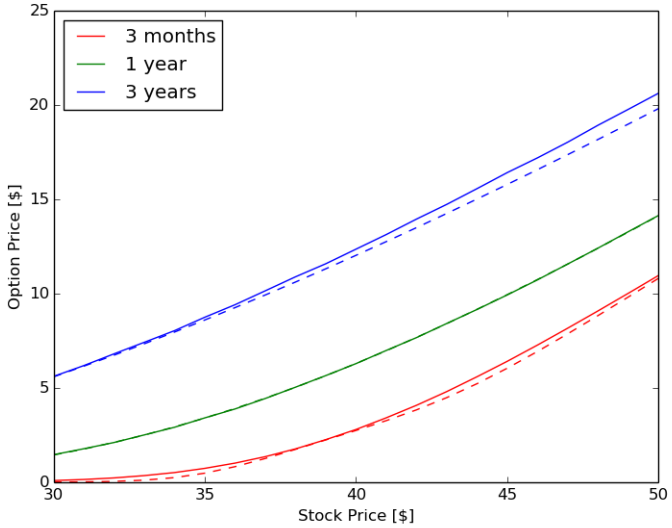


Figure 6.4: Call option prices over range of initial asset price S_0.

strike price for puts and calls respectively at any time up to and including the option expiration time. Since the domain of possible exercise times is continuous and includes the expiration time as used in European style option contracts, American option contracts are always more valuable.

6.2.1 Analytic Solutions

Finding an analytic solution to the pricing of American style option involves two parts. The first part is in finding an optimal exercise time. For a European style option, there is only one potential exercise time and thus the optimal exercise time. With a continuum of potential exercise times, finding the optimal exercise time becomes much more difficult. The second part is finding the option price given an optimal exercise time.

Analytic algorithms for pricing American style options require an analytic model of the price process for both determining the optimal exercise time and computing the option price. Since with most real data sets, this analytic model is not known and must be estimated leads to additional difficulties from the additional sources of error in model misidentification. For a problem where the model must be learned and updated as data is received, as in streaming

time series or an online learning problem, the unknown model must then be estimated from partial data. The last added difficulty is that this true price process is not only unknown, it is possibly non-stationary, which implies an underlying model that varies over time. The rest of this dissertation computes an American style option price via a computational method known as least squares Monte Carlo (LSM) that does not make an assumption of stationarity in the underlying process.

6.2.2 LSM Algorithm

The LSM algorithm approximates an American style option contract with a continuum of exercise times by starting with an approximately equivalent Bermudian option with finite exercise times. It is known that this option price is a lower limit of the American style option since it has a finite number of discrete exercise times. Therefore, one method for increasing the accuracy of estimate option prices from the LSM algorithm is to increase the number of potential exercise times.

Algorithm 4 lays out detailed steps of the LSM algorithm. The first step within the LSM algorithm is to generate sample paths of the underlying asset for each intermediate time interval t_i represented by $S_m[t_i]$. These price paths are used to approximate the continuum of potential outcomes. The granularity of the time steps in generating the sample paths correspond to the finite discrete exercise times of the approximating Bermudian option.

The second step in Algorithm 4 is to evaluate the option's payoff V_m at the final time t_N for each of the options using the payoff function h. These prices should always be positive due to the option's definition. Next, sample option paths that are in-the-money are used in a least squares to approximate weights a_j of the basis functions ϕ_j used. This least squares step is fitting a function that maps the value of the asset price at the previous time interval to the payoff at the next time interval. Several variations from the original LSM algorithm exist as outlined in Tompaidis and Yang (2014) where methods other than least squares are used to determine the weights a_j. These other methods include quantile regression, Tikhonov regularization, matching projection pursuit, and classification and

Algorithm 4 Pseudo code for least squares Monte Carlo algorithm.

procedure MAIN:
  for m **in** M **do**:
    $S_m[t_0] \leftarrow S_0$
    for i **in** [1,...,N] **do**:
      $S_m[t_i] \leftarrow \text{SimPath}[S_m[t_0 : t_{i-1}]$
  for i **in** [N,...,1] **do**:
    for m **in** M **do**:
      $V_m[t_i] \leftarrow h\left([S_m[t_i]]\right)$
    for m **in** M **do**:
      if $S_m[t_{i-1}] > 0$ **then**:
        $S_m^+[t_{i-1}] \leftarrow S_m[t_{i-1}]$
    $\mathbf{a} \leftarrow \underset{\mathbf{a}}{\text{argmin}} \left\| \sum_{j=1}^{N_b} a_j\left(t_{i-1}\right) \phi_j\left(\boldsymbol{S}^+[t_{i-1}]\right) - e^{-r(t_i - t_{i-1})} \boldsymbol{V}[t_i] \right\|$
    for m **in** M **do**:
      if $h\left([S_m[t_{i-1}]]\right) \geq \sum_{j=1}^{N_b} a_j \phi_j\left(S_m[t_{i-1}]\right)$ **then**:
        $V_m[t_{i-1}] \leftarrow h\left([S_m[t_{i-1}]]\right)$
      else:
        $V_m[t_{i-1}] \leftarrow e^{-r(t_i - t_{i-1})} V_m[t_i]$
  return mean $(\boldsymbol{V}[t_0])$

regression trees. For this dissertation, the steps outlined by Longstaff and Schwartz (2012) will be followed. Orthogonal polynomials such as Laguerre or Hermite polynomials are used as the basis functions with unknown parameters to be fitted by the regression model. A variation to the LSM algorithm which is used here is the scaling of all prices by the initial asset price S_0, this lowers the condition number of the matrix to be inverted in the least squares step. A lower condition number allows for more efficient and faster calculations to be made when running the LSM algorithm.

Lastly, a decision is made to determine the option price at the previous time step represented as $V_m[t_{i-1}]$. This decision is based on the return from exercising the option at the time as compared to the estimated value of continuation. If the value of continuation is higher, then the current option price becomes the discounted price of the option return from the next time interval. Otherwise the option value is taken to be the return from exercising at the previous time interval. This process is then repeated at the previous time interval until the initial time T_0 is reached.

LSM pricing accuracy is increased by decreasing the time steps used in the sample path generation, increasing the number of sample paths used, and increasing the number of

orthogonal polynomials used in the regression model. The first method is based on Section 6.2 where American options are taken to be the limit of Bermudian options. The last two methods for increasing LSM algorithm accuracy are taken from Monte Carlo and regression model error reduction principles respectively.

6.2.3 Verification of LSM Algorithm

The LSM algorithm is provided based upon the works of Longstaff and Schwartz (2012) and Tompaidis and Yang (2014) with slight modifications. A scaling of all prices was done based upon the initial stock price S_0 and all sample paths were generated with their antithetic equivalents. Before the algorithm may be used in this work, the algorithm as implemented in Python must be verified to work as intended. This is done by recreating the values generated by the authors of previous works. Tables 6.1 and 6.2 show published results with their standard errors along side the computed values for verification of the implementation.

Within table 6.1, the first three columns depict the simulated asset starting price S_0, number of Monte Carlo generated sample price paths used, and degree of Hermite type polynomial used within the LSM algorithm respectively. Respective columns 4 and 6 of table 6.1 show estimated call option prices using the LSM algorithm from Tompaidis and Yang (2014) and as implemented. Respective columns 5 and 6 show standard errors of the various option contracts as determined from the 20 runs.

Table 6.2 shows published and as implemented prices for the various put option prices estimated in Longstaff and Schwartz (2012). Laguerre type polynomials of degree 3 were used in the LSM algorithm runs. The first three columns of table 6.2 are the initial underlying asset price S_0, volatility of underlying GBM process referred to as σ, and time until expiration T. Respective columns 4 and 6 show published option prices from Longstaff and Schwartz (2012) and as calculated prices. Columns 5 and 7 depict the standard errors of the respective simulated and estimated option prices.

It can be seen from both tables, that the algorithm converges as intended for a variety of conditions for both puts and calls over a range of initial prices, strike prices, drift and

Table 6.1: Verification of LSM algorithm implementation against examples presented in Tompaidis and Yang (2014)

S_0	No. Paths	Poly. Deg	Simulated Price	(s.e.)	Computed Price	(s.e.)
90	10^3	5	2.17	(.01)	2.743	(.111)
90	10^3	10	2.16	(.01)	2.860	(.075)
90	10^3	15	2.15	(.01)	2.866	(.125)
90	10^4	5	2.34	(.00)	2.405	(.029)
90	10^4	10	2.34	(.00)	2.430	(.034)
90	10^4	15	2.33	(.00)	2.438	(.030)
90	10^5	5	2.38	(.00)	2.364	(.008)
90	10^5	10	2.38	(.00)	2.367	(.008)
90	10^5	15	2.38	(.00)	2.366	(.007)
100	10^3	5	5.70	(.01)	6.236	(.091)
100	10^3	10	5.68	(.01)	6.535	(.106)
100	10^3	15	5.68	(.01)	6.654	(.137)
100	10^4	5	5.86	(.00)	5.952	(.057)
100	10^4	10	5.86	(.00)	5.970	(.042)
100	10^4	15	5.86	(.00)	5.987	(.043)
100	10^5	5	5.90	(.00)	5.883	(.012)
100	10^5	10	5.90	(.00)	5.883	(.011)
100	10^5	15	5.90	(.00)	5.887	(.015)
110	10^3	5	11.48	(.001)	12.002	(.090)
110	10^3	10	11.46	(.001)	12.230	(.188)
110	10^3	15	11.44	(.001)	12.309	(.185)
110	10^4	5	11.69	(.000)	11.730	(.041)
110	10^4	10	11.69	(.000)	11.768	(.043)
110	10^4	15	11.69	(.000)	11.734	(.037)
110	10^5	5	11.74	(.000)	11.722	(.017)
110	10^5	10	11.74	(.000)	11.723	(.013)
110	10^5	15	11.74	(.000)	11.722	(.016)

Table 6.2: Verification of LSM algorithm implementation against examples presented in Longstaff and Schwartz (2012)

S_0	σ	T	Simulated American	(s.e.)	Computed Price	(s.e.)
36	.20	1	4.472	(.010)	4.465	(.007)
36	.20	2	4.821	(.012)	4.835	(.007)
36	.40	1	7.091	(.020)	7.065	(.004)
36	.40	2	8.488	(.024)	8.477	(.010)
38	.20	1	3.244	(.009)	3.235	(.003)
38	.20	2	3.735	(.011)	3.733	(.008)
38	.40	1	6.139	(.019)	6.111	(.004)
38	.40	2	7.669	(.022)	7.638	(.009)
40	.20	1	2.313	(.009)	2.296	(.002)
40	.20	2	2.879	(.010)	2.873	(.003)
40	.40	1	5.308	(.018)	5.270	(.003)
40	.40	2	6.921	(.022)	6.888	(.004)
42	.20	1	1.617	(.007)	1.600	(.004)
42	.20	2	2.206	(.010)	2.202	(.004)
42	.40	1	4.588	(.017)	4.539	(.004)
42	.40	2	6.243	(.021)	6.214	(.007)
44	.20	1	1.118	(.007)	1.093	(.002)
44	.20	2	1.675	(.009)	1.677	(.005)
44	.40	1	3.957	(.017)	3.898	(.005)
44	.40	2	5.622	(.021)	5.611	(.010)

volatility values, as well as with different polynomial approximations. Differences in scaling may account for the differences in Table 6.1 at low numbers of sample paths. All further uses of the LSM algorithm will use 10^5 sample paths in calculating American style option prices. The true prices should be valued more than the LSM outputs due to the approximation of the continuum of exercise times by a Bermudian style option with 50 discrete exercise times. It can be expected that the LSM algorithm will converge to these true prices as both the number of intermediate exercise times (for equal spacing in time) and number of sample paths uses approaches infinity.

6.3 Option Price Sensitivity

Two simulated asset price processes are presented to demonstrate the algorithm's ability to price American style options. The two process were chosen and represented in equations (6.12) and (6.13) to show the similarity in the noise processes. The first process shown in equation (6.12) is a simple arithmetic process with Gaussian noise $\epsilon \sim \mathcal{N}(0.1, 0.04)$. The second process shown in equation (6.13) is a general geometric Brownian motion with $\epsilon \sim \mathcal{N}(-0.07\text{dt}, 0.04\text{dt})$. More importantly, it is shown how sensitive the calculated option prices are to errors in the estimated parameters based upon a particular realization of an asset price path.

$$S_{t+1} = S_t + \epsilon \tag{6.12}$$

$$S_{t+1} = S_t \exp(\epsilon) \tag{6.13}$$

Sample paths for each process are seeded at a time T_0 and propagated backwards in time. This is done so that all learned models on each realized asset price path end at the same asset price of S_0 at T_0. Figures 6.5 and 6.6 show 50 sample paths for the two processes respectively.

Varying the mean and variance for the arithmetic process leads to the put and call option

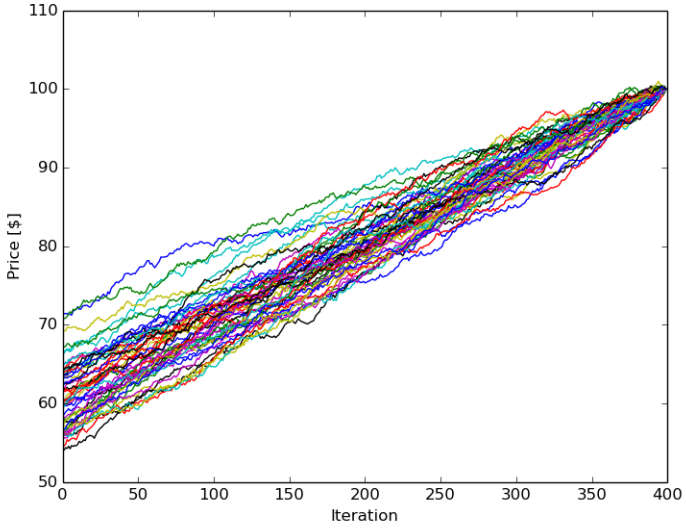


Figure 6.5: Sample paths for arithmetic propagated backwards from an initial time T_0 with asset price S_0.

prices as shown in Figure 6.7. This figure also illustrates the option price's robustness to differences in the noise variance as indicated by the over-lapping of the three lines. Changing variance has little to no change on the underlying option's price due to the noise being symmetric and linear with respect to the current asset price. It is expected that a non-symmetric or non-linear noise will result in non-overlapping curves.

Figures 6.8 and 6.9 show how varying the drift and volatility within the second process influences the European and American style option prices. Put prices in Figure 6.8 use the following parameters: initial stock price $S_0 = \$40$, strike price $K = \$40$, time to expiration $T = 1$ with 50 equi-spaced time steps, and no dividends. Call prices in Figure 6.9 use the following parameters: initial stock price $S_0 = \$100$, strike price $K = \$100$, time to expiration $T = 1$ with 50 equi-spaced time steps, and a continuous dividend rate of $\delta = 0.1$. If the parameters within GBM are replaced with an arbitrary mean m and variance s as shown in equations (6.14) and (6.15). For a true GBM process, the logarithm of the price differences results in a normal distribution that implies that the price differences follow a log-normal distribution. However, it is typical to model the GBM stochastic differential equation in the risk neutral measure. It is important to note, that ANNs learning this process using

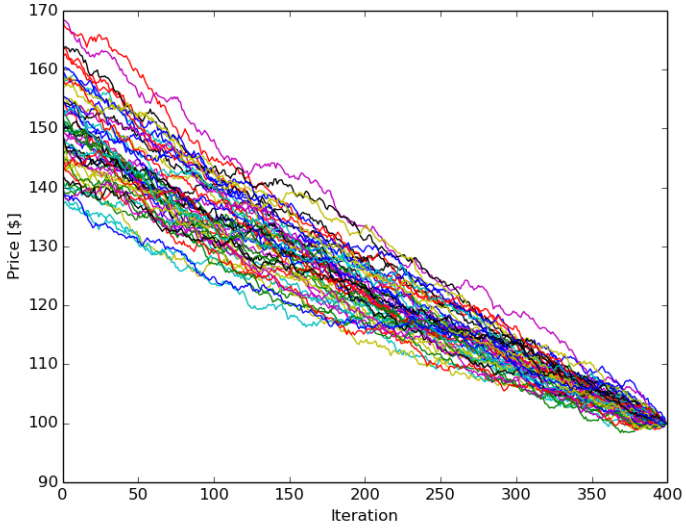


Figure 6.6: Sample paths for geometric processes propagated backwards from an initial time T_0 with asset price S_0.

MDNNs will instead learn the mean and variance as shown in equations (6.14) and (6.15). This is due to the output of MDNNs being a mixture of Gaussians.

$$m = \left(r - \delta - \frac{1}{2}\sigma^2\right) \mathrm{d}t \tag{6.14}$$

$$s = \sigma\sqrt{\mathrm{d}t} \tag{6.15}$$

6.4 Estimation of LSM Prices

For the two simulated processes, many realizations are generated for which an ensemble of neural networks are seeded and used to learn the conditional probability distribution of the underlying process. Since each realized price path terminates at time T_0 with asset price S_0 it is possible to compare estimated option prices generated from different asset price path realization. The LSM algorithm is used to estimate the price of an in-the-money American style call option that expires 50 iterations from T_0 at T_{50}.

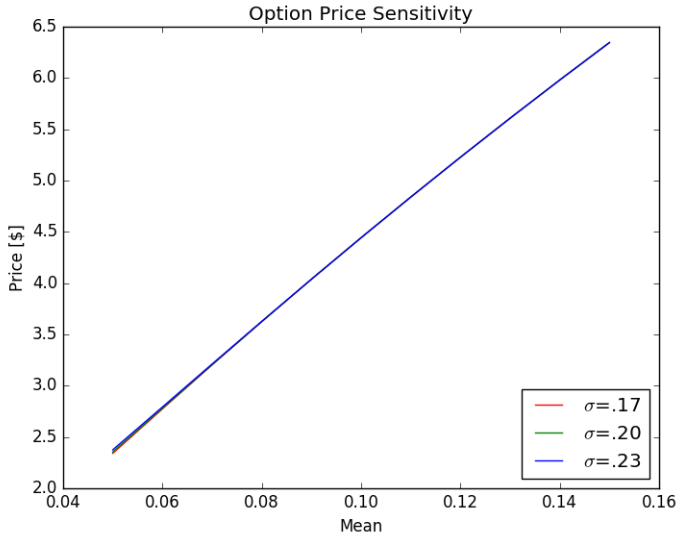


Figure 6.7: Sensitivity of option prices calculated using LSM algorithm for asset following arithmetic process as described in equation (6.12).

Because the neural networks are trained upon a time series of input data with fixed discrete time differences known as dt, sample price paths may not be re-sampled at a higher sample rate. This implies that the LSM algorithm is actually pricing a Bermudian option with dt^{-1} evenly spaced exercise points. Therefore, the LSM algorithm results in a lower bound for the option price since a continuum of exercise times is always more valuable than discrete exercise times. Using longer training histories when evaluating the K-S test statistic should result in a more accurate pricing of option prices.

Figures 6.10 and 6.11 show the empirically estimated cumulative density functions for option prices based upon different realized price paths for different lengths of training histories. It can be seen from Figure 6.10 that the LSM algorithm clearly converges upon the true option price of $4.44 (input parameters for option pricing within LSM). This CDF implies that using an ensemble of neural networks to learn the conditional probability distribution for this particular process is robust against a particular realization of the asset price path so long as the training history is long enough to reduce the variance in the option prices within acceptable limits.

It is important to note that K-S values for the evaluated functions decreases as the

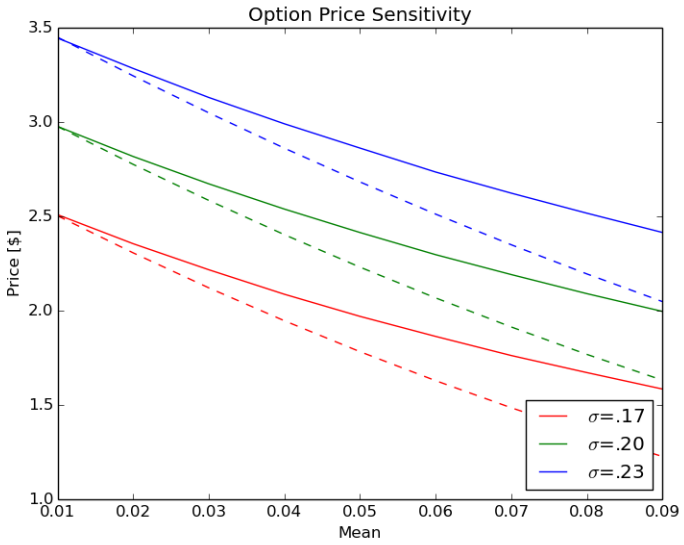


Figure 6.8: European and American style option prices using Black-Scholes (dashed) and LSM algorithm (solid) respectively. Parameters for put options used are chosen based upon examples given in Longstaff and Schwartz (2012).

number of training history points is used at each iteration. These values provide a p-value for testing if the two distributions are different. In this case, the test determines if the learned conditional probability distribution is different from the true conditional probability distribution.

It can be seen from the data that in both cases, the p-values generated from using the average of the K-S statistic over the last 10 iterations show that the distributions are not different with 95% confidence. Since the LSM algorithm is shown to be implemented correctly, and the ANNs are appropriately learning the conditional probability distribution of the noise, this implies that the generation of American style options based upon streaming data is in fact ill-posed. (Large variations in estimated option prices generated from modes that are shown to be not different from the true processes.)

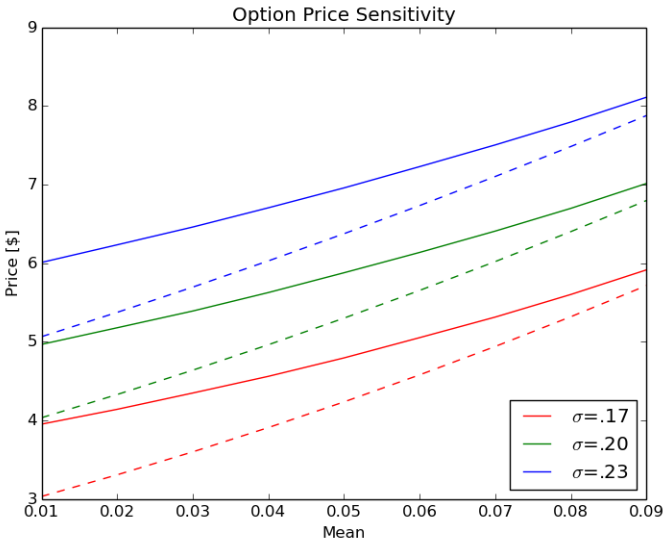


Figure 6.9: European and American style option prices using Black-Scholes (dashed) and LSM algorithm (solid) respectively. Parameters for call options used are chosen based upon examples given in Tompaidis and Yang (2014).

6.5 Verification of Output

Figure 6.10 shows that the ANNs are capable of learning the correct process to yield reasonable results for the output American style option prices with a true price of $4.93. As the training history is increased, the CDF of the output price distribution converges to the correct value. This is most clearly illustrated by the decreasing variance in the output values. Similarly, Figure 6.11 shows the same behavior albeit to a lesser extent. This difference between the two CDF plots requires a justification since in both cases the ANNs are essentially finding the mean and variance of an underlying Gaussian process.

Using the maximum likelihood estimators for mean and variance of a Gaussian process as described in Casella and Berger (2002) yields distributions for both the mean and variance shown in equations (6.16) and (6.17). This distribution is a result of the particular realization of values over the given history of observations. As the length of history n is increased, and thus more observations to estimate the mean and variance, the estimates for mean and variance converge to the true values.

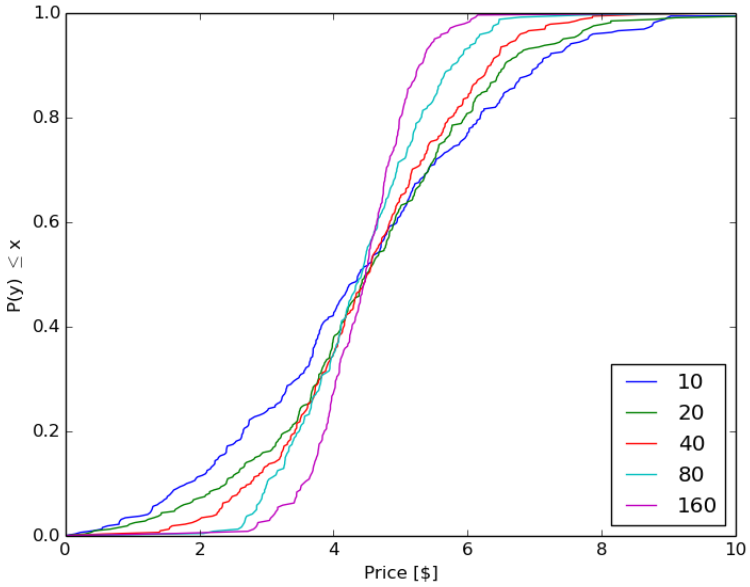


Figure 6.10: CDF of LSM option prices for arithmetic process

$$\widehat{\mu} \sim \mathcal{N}\left(\mu, \frac{\sigma^2}{n}\right) \tag{6.16}$$

$$\widehat{\sigma}^2 \sim \frac{\sigma^2}{n}\chi^2_{n-1} \tag{6.17}$$

After the ANNs complete the desired number of training iterations, the learned models are used to generate sample paths to be used within the LSM algorithm. In this particular case, both processes are stationary and the noise is independent of the underlying asset price. This implies that the learned output models will yield constant values for the noise mean and variance.

Although the variance of both the observed mean and standard deviations as seen in Figures 6.14 and 6.15, it is difficult to determine if the values converge as anticipated.

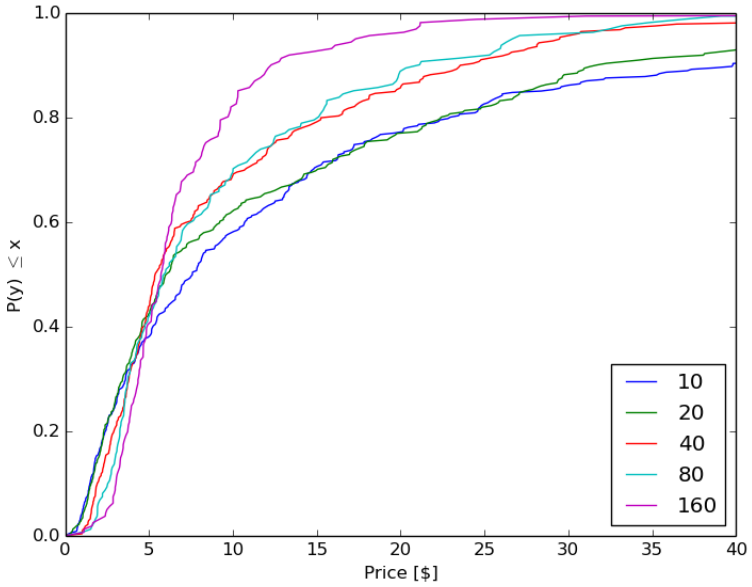
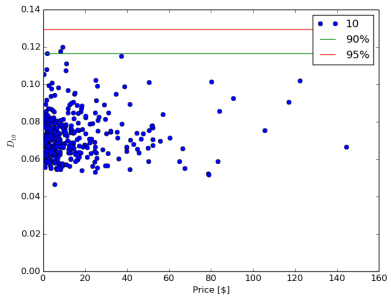


Figure 6.11: CDF of LSM option prices for GBM process

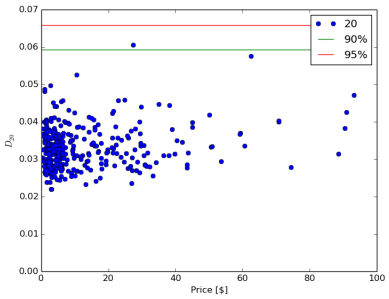
Therefore, the output mean and variance from each model are normalized per the transformations in equations (6.16) and (6.17) and replotted against their theoretical distributions in Figures 6.16 - 6.19. It is expected that the normalized estimates for the mean will approximate a standard normal distribution. It is also expected that the normalized estimates for the variance will approximate a chi squared distribution with $n-1$ degrees of freedom, where n is the width of the training window used in evaluating the models.

It can be seen from Figures 6.16 - 6.19 that the normalized mean and variance from both arithmetic and GBM processes approximate the theoretical distributions. This implies that the ANNs are learning the appropriate values for mean and variance after conditioning upon the effects of a particular realization of sample data. Differences between the theoretical and observed distribution of normalized mean and variance can be attributed to error of the ANNs in fitting the data. The larger differences in particular for the runs using a 10 iteration training history reinforce the idea that a small training history leads to poor convergence of the ANNs. It can also be seen that larger deviations exist in the learned

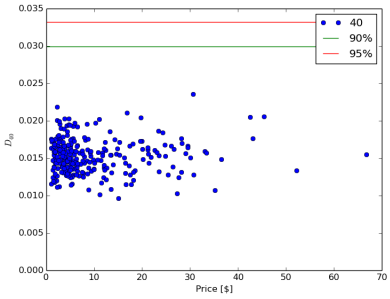
values for a GBM process as compared to the arithmetic process, this may be attributable to the skewness of the log-normal distribution of noise in the GBM process.



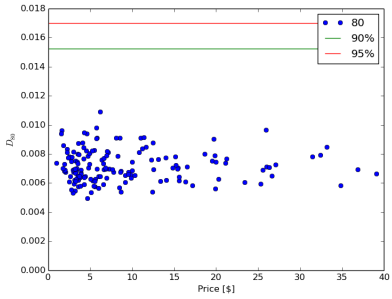
(a) Comparison against F_1



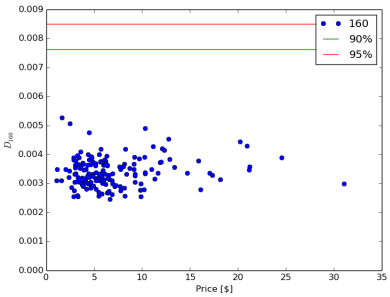
(b) Comparison against F_2



(c) Comparison against F_2

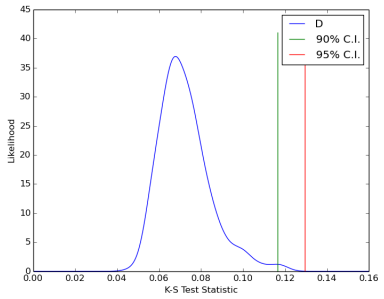


(d) Comparison against F_2

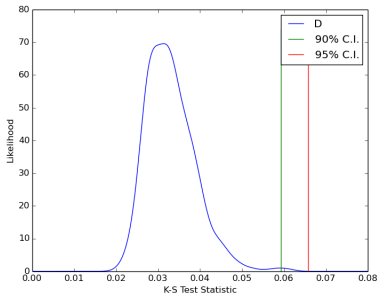


(e) Comparison against F_2

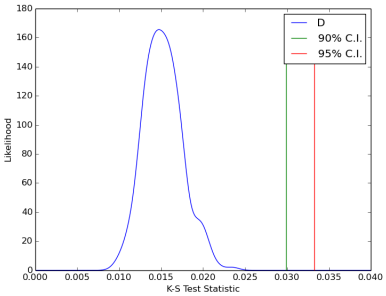
Figure 6.12: Scatter plot of K-S test statistic derived from last 10 training iterations against LSM option price with decision points for 90% and 95% confidence intervals.



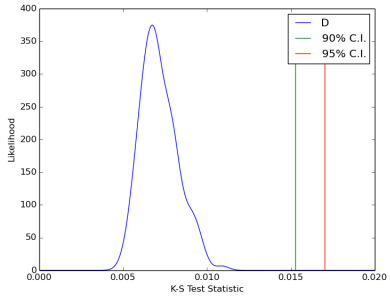
(a) Comparison against F_1



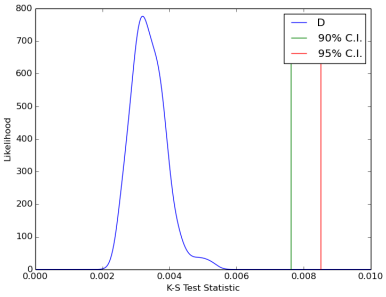
(b) Comparison against F_2



(c) Comparison against F_2

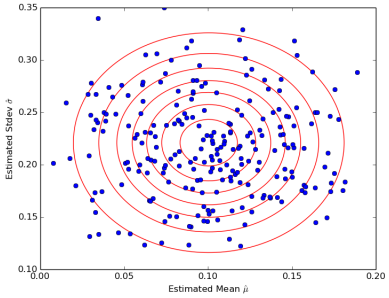


(d) Comparison against F_2

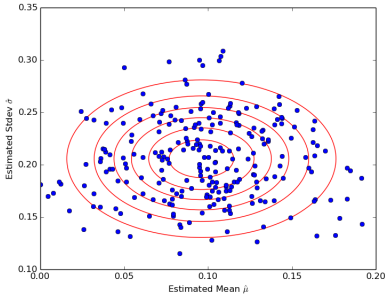


(e) Comparison against F_2

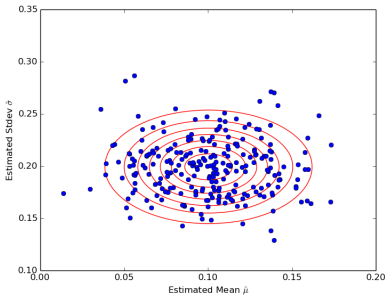
Figure 6.13: Estimated density of K-S test statistic derived from last 10 training iterations against density likelihood with decision points for 90% and 95% confidence intervals.



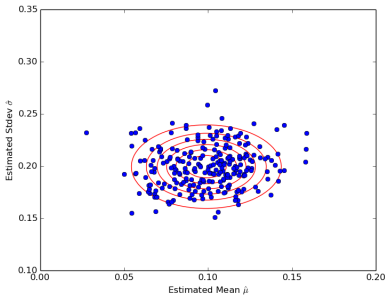
(a) 10 iteration training history.



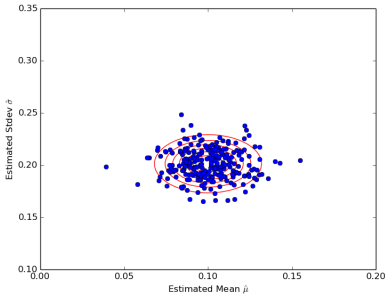
(b) 20 iteration training history.



(c) 40 iteration training history.

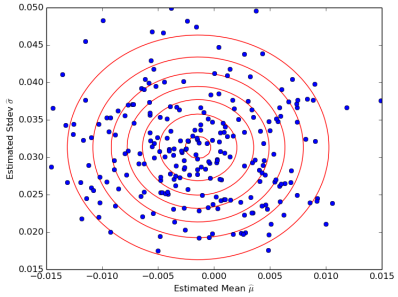


(d) 80 iteration training history.

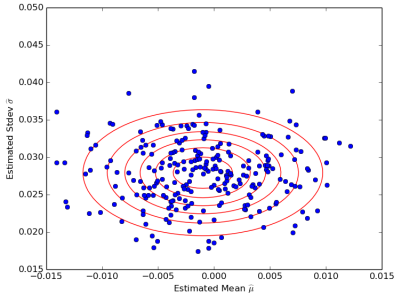


(e) 160 iteration training history.

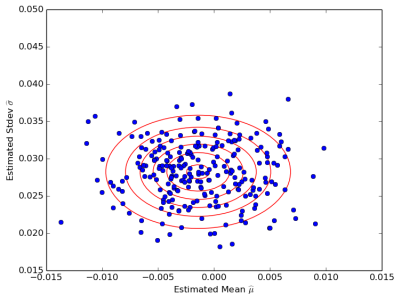
Figure 6.14: Scatter plots of learned values for mean and standard deviation from arithmetic process with different lengths of training histories.



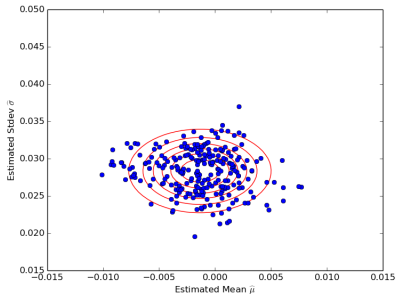
(a) 10 iteration training history.



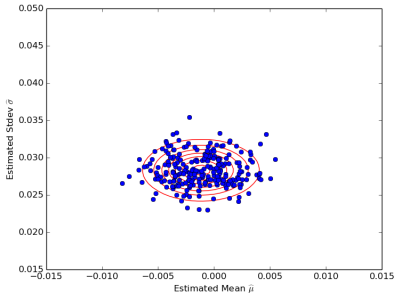
(b) 20 iteration training history.



(c) 40 iteration training history.

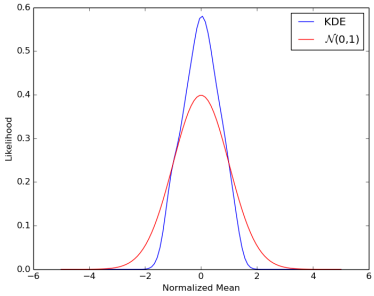


(d) 80 iteration training history.

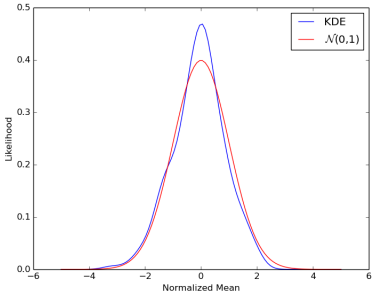


(e) 160 iteration training history.

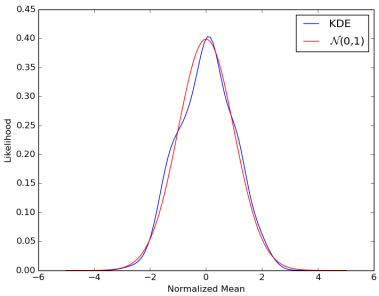
Figure 6.15: Scatter plots of learned values for mean and standard deviation from GBM process with different lengths of training histories.



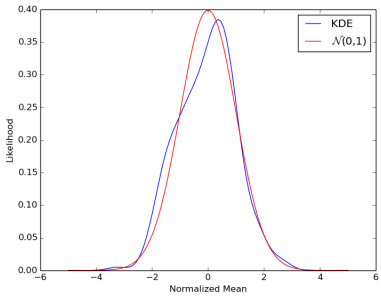
(a) 10 iteration training history.



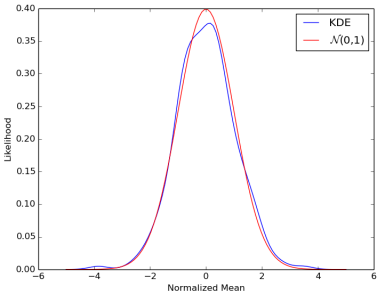
(b) 20 iteration training history.



(c) 40 iteration training history.

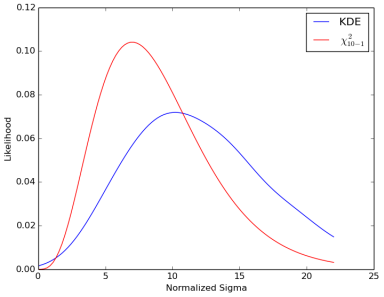


(d) 80 iteration training history.

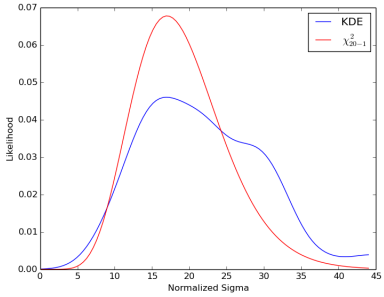


(e) 160 iteration training history.

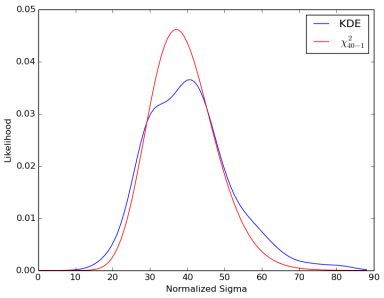
Figure 6.16: Theoretical distribution of normalized estimated mean with estimated density of observed normalized mean values for Arithmetic process.



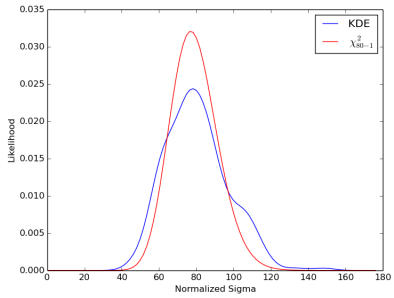
(a) 10 iteration training history.



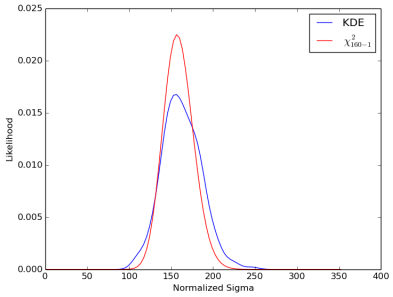
(b) 20 iteration training history.



(c) 40 iteration training history.

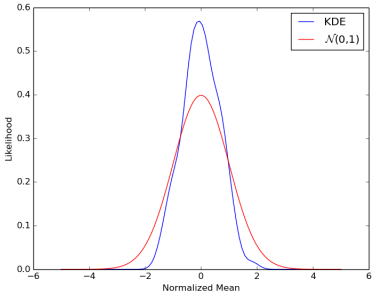


(d) 80 iteration training history.

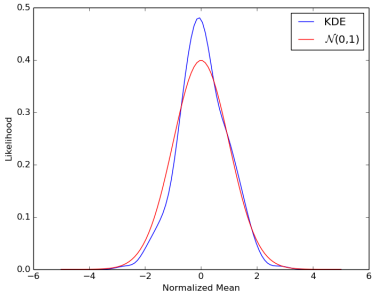


(e) 160 iteration training history.

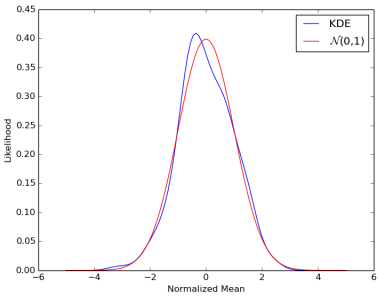
Figure 6.17: Theoretical distribution of normalized estimated variance with estimated density of observed normalized variance values for Arithmetic process.



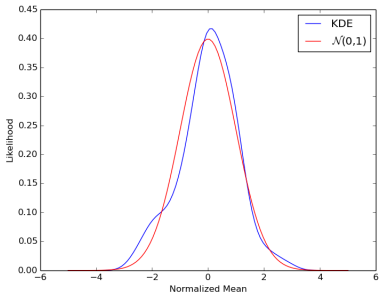
(a) 10 iteration training history.



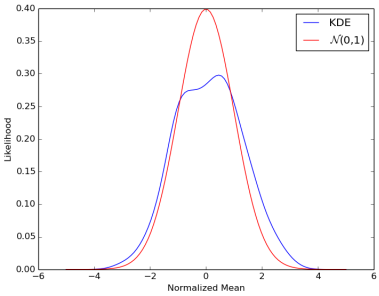
(b) 20 iteration training history.



(c) 40 iteration training history.

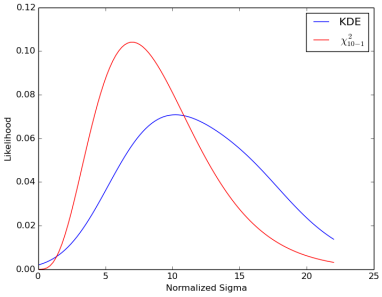


(d) 80 iteration training history.

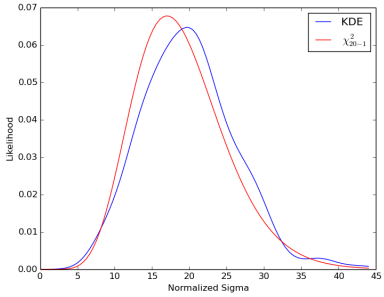


(e) 160 iteration training history.

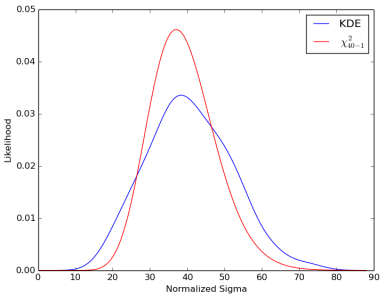
Figure 6.18: Theoretical distribution of normalized estimated mean with estimated density of observed normalized mean values for GBM process.



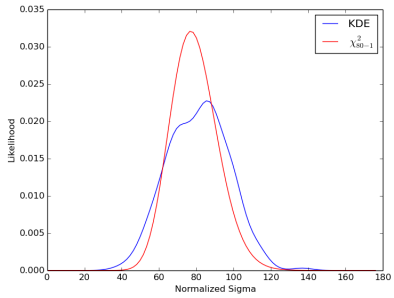
(a) 10 iteration training history.



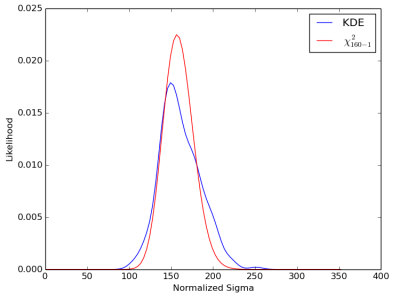
(b) 20 iteration training history.



(c) 40 iteration training history.



(d) 80 iteration training history.



(e) 160 iteration training history.

Figure 6.19: Theoretical distribution of normalized estimated variance with estimated density of observed normalized variance values for GBM process.

Chapter 7: Implementation

Code for resource monitoring, multiprocessing, database management, and computations are run in Python 2.7 and versions of the code are controlled using Git. The developed code may be run on either Amazon Web Service (AWS) virtual machines (VMs) via remote secure shell (SSH) or on local hardware. The `multiprocessing` and `psutil` modules within Python allow for the program to scale computations based upon available hardware. A configuration file written in JavaScript Object Notation (JSON) format is used to track all system and model parameters and determine which code versions to run at each running. Copies of the configuration file are saved with the relevant output and log files to track progress. Currently, the code is only compatible with Amazon VMs. Future work may extend compatibility to Google's Cloud Computing platform as well as Microsoft's Azure platform.

7.1 Distributed Computing

For large computational tasks, the processing resources of a single processor or even a single machine can easily be insufficient. Therefore, it is necessary to distribute the computational workload over multiple machines to increase computational capacity. Two methods for distributing the computational workload are parallel computing and distributed computing.

Parallel computing is best characterized by multiple processes on a single machine each having access to a shared memory. An example of this is a single machine with multiple processing cores. Further application of parallel computing involves the use of general purpose graphics processing units (GPGPUs). GPGPUs typically are used in the processing of images as used to display content on computer monitors, however proper modifications to parallel processing algorithms allows for speedup of 10 to 100 times that of using a single

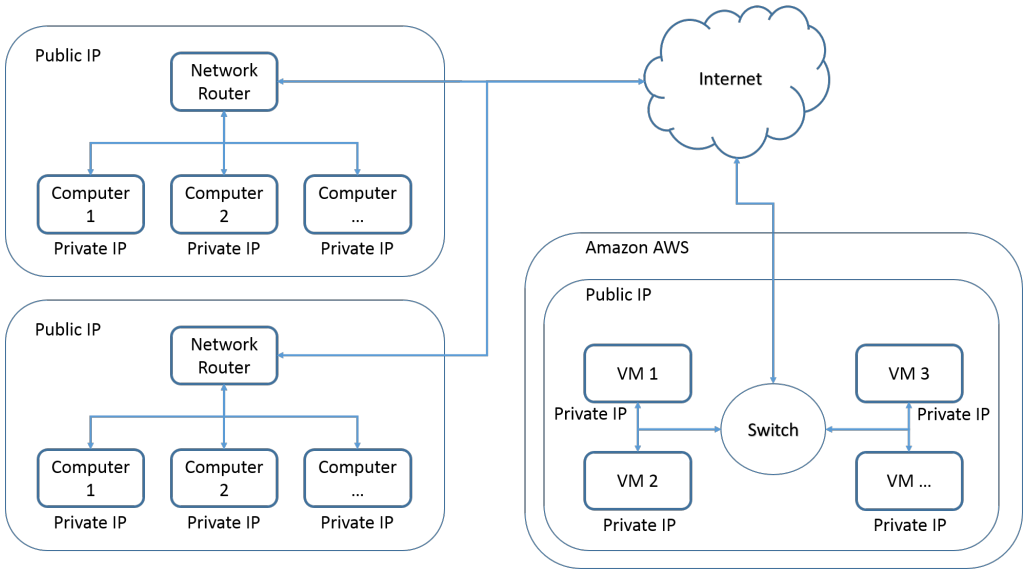


Figure 7.1: Diagram of hybrid hardware network for distributed computing.

processor depending on the hardware used.

Distributed computing is when various processes are executed on different machines (with possibly different hardware configurations and operating systems) that perform message passing via a network or the internet as illustrated in Figure 7.1. Using a local network or internet, algorithms designed for distributed computing may utilize tens of hundreds of separate machines. With careful design, the distributed computing code should be agnostic to the actual hardware and operating system being used. This allows for a distributed hybrid network of actual hardware and virtual machines.

To best utilize the resources available, it is best to take advantage of both strategies (namely parallel or shared memory as well as distributed computing constructs). A master node or computer is used to manage the entire project and distributes the workload across multiple machines on the network. Each machine then has a local manager process that distributes the workload among the internal processors with access to a local shared memory. Figure 7.2 shows an architecture that utilizes both parallel and distributed computing strategies by allowing for a common shared job queue and worker processes distributed to multiple machines. These distributed workers also have access to common databases hosted on one, multiple, or different machines than those performing the computations.

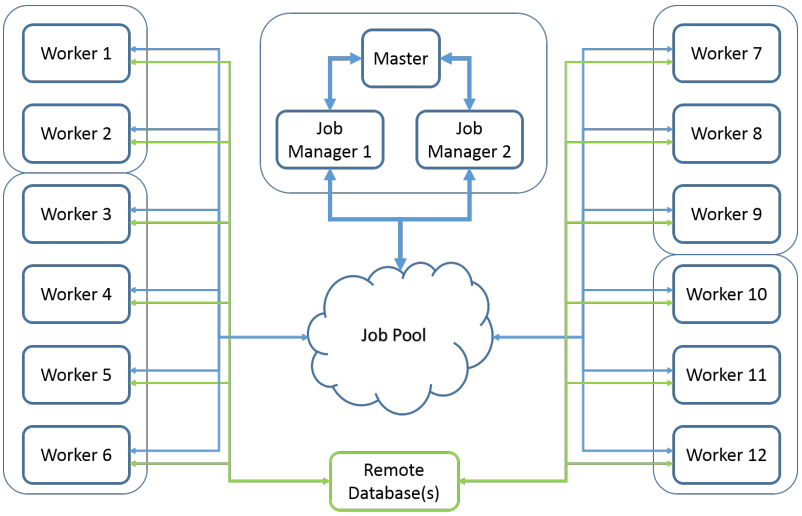
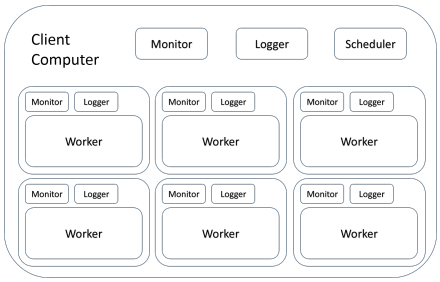
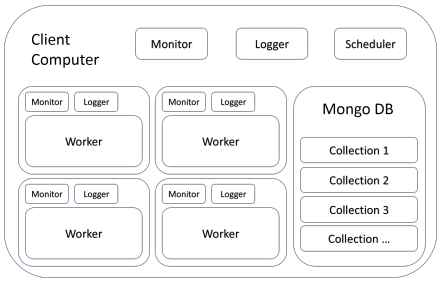


Figure 7.2: Data flow in distributed process and database architecture.

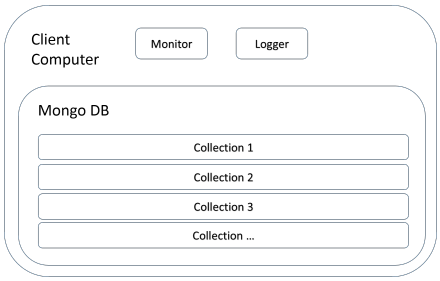
Various constructs may be used in the architecture design as shown in Figure 7.3. Client machines may be used in several different methods depending on the particular problem to be solved. The first setup is a machine that only hosts worker processes. These workers are where the majority of the computations occur. The second setup allows for machines that host both worker processes and a portion (or all) of a single database. For the purposes of this dissertation, these first two setups are used for all subsequent results. The third and fourth setups are such that the host machines are only used for databases. Depending on the size of the database relative to the capacity of the hardware, multiple databases may be hosted on the same machine. Additional considerations may be made based on the required response time for queries. The database used in this dissertation is a MongoDB instance that runs on the local hard drive. For faster query times, a Hadoop cluster may be used to allow the entire database to reside within the collective random access memory (RAM) of the computers within the Hadoop cluster.



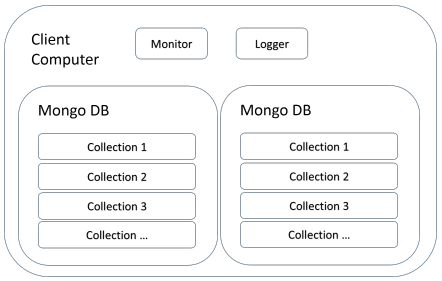
(a) Client hosting only workers.



(b) Client hosting workers and database.



(c) Client hosting only database.



(d) Client hosting multiple databases.

Figure 7.3: Various types of client computer functions.

7.1.1 Network Setup

When using a set of computers that sit on a local subnet behind a wireless router, it is necessary to configure the network address translation (NAT) before access from the public internet is possible. This subnet is used for both security purposes and to better use the limited number of internet protocol (IP) address. Using a subnet allows for computers on the local network to exchange information without being exposed to the public internet. This also serves to provide a single public IP address that is accessible to the public internet. Routing from the public internet to computers on the local network are controlled by the NAT within the network router that map public IP ports to individual ports on each computer. For example, the router may map port 8080 of the public IP to port 22 (SSH) on a specific computer on the private network.

In the case of Python's multiprocessing server processes, the host computer's port is the

point of ingress for all network traffic and as such the router needs to route all port traffic to that computer. This enables computers on the private network to communicate unimpeded with the host or master node. For database instances, the same process is followed to allow for unimpeded traffic to and from the database host or hosts.

A third network alteration is necessary for local machines to communicate with Amazon elastic cloud compute (EC2) instances. Special considerations need to be made based upon the local hardware and local hardware as well as the anticipated volume of data transfer. Further information may be located within the help files for setting up virtual private clouds (VPCs) on the AWS website.

7.1.2 Distributed Computing with Python

Python provides a module named `multiprocessing` that allows users to construct server/client processes as well as spawn child processes and threads. This module is especially important when using Python due to Python's internal global interpreter lock (GIL), which only allows a single spawned thread to access the Python kernel. For a single process that runs a single thread this does not introduce any limitations since there is no contention for kernel access. On a multi-core machine, multiple threads usually implies parallel computing. When using Python however, a multi-core computer is limited by thread access to the Python kernel, preventing full utilization of the multiple cores. Therefore, it is best to spawn child processes within Python that access independent copies of the Python kernel when simultaneous computing across multiple computer cores is needed. Operations such as input/output and database reads are exempt from this restriction because they do not rely on Python.

Server/Client Processes

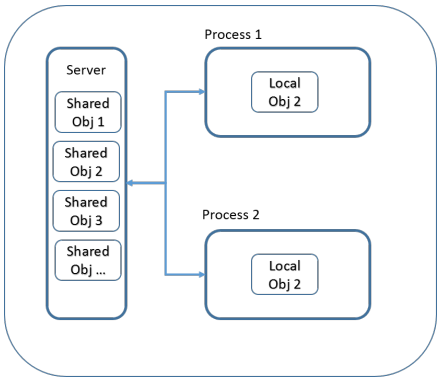
When sharing objects across multiple processors on the same computer, a pointer is exchanged between the processes to point to the location within a shared memory location.

When sharing objects across multiple computers, there is no shared memory. Thus a different sharing mechanism is required. In this case, it is necessary for the master process to spawn a server manager process to handle the network exchanges of the shared objects. These shared objects are shared via message passing objects (queues) that are accessible by all the clients registered with the server process.

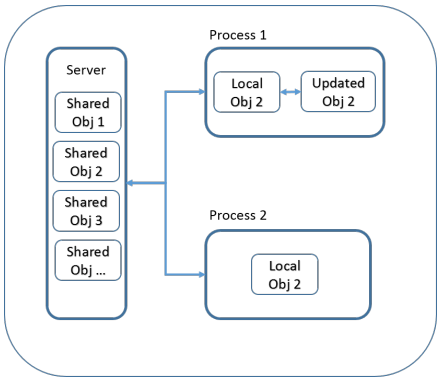
Each client process will then essentially subscribe to the server and pass messages to the other clients on the network via the server messaging objects. This subscription allows client processes to join and exit the global computations based upon resource availability without interrupting the computations. It is best practice to construct at least two queues, one for inbound traffic and another for outbound traffic so that objects are extracted from the queue by the appropriate process.

Sharing Objects via Server Processes

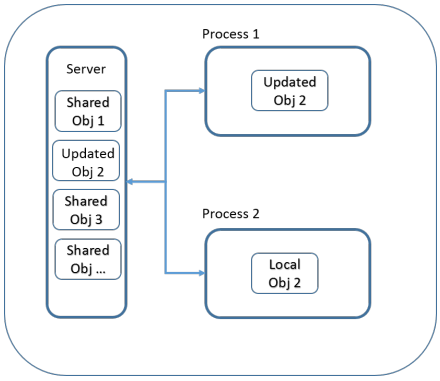
In a distributed computing setup, there is no common shared memory for all processes. To share a single concurrent copy of an essential object, it is necessary to host a server that acts as the keeper of the global master copy for these shared objects. Figure 7.4 illustrates how the server process provides this server to all the subscribed clients. To begin, when the clients are initiated, they request a copy of the shared objects from the server. Once the clients receive local copies of these objects, they may be used just as any other local object. When the client needs to update or change one of the shared objects, the client must acquire a lock from the server that prevents multiple clients from simultaneously updating the object. Once the client has updated the shared object, the client will send the updated object to the server, which then becomes the new master copy. The server will then push the updated object to all clients and then release the lock on the object. This process is then repeated as necessary. For processes that perform frequent updates, a different strategy may be required.



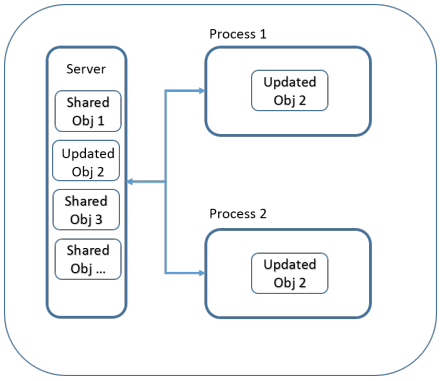
(a) Idle state of distributed server/client processes.



(b) Client process 1 updating shared object.



(c) Client process 1 publishing updated object to server.



(d) Client process 2 receiving updated object from server.

Figure 7.4: Four steps on client processes updating shared objects.

7.2 Databases

MongoDB was used to store and organize all relevant information for the computations including the asset price history, calculated option prices, ANN weights, and ANN outputs. MongoDB is a non-structured database query language that allows for JSON type objects to be stored without a common object structure. This means that each object within the database or even collections need share a common structure. The Python module `pymongo` is used to interface Python scripts with the MongoDB instances.

Using MongoDB, it is possible to store intermediate structures and results from the various model runs. A single collection is used to house JSON files of all the intermediate

neural network architectures and weights. Similarly, another collection contains all the estimated parameters for each underlying process on each iteration for each model type. It is also possible to store variations of the estimated parameters with the variations documented as entries in the JSON.

Post processing may then take advantage of the multiple runs and realizations of the models to aggregate results for visualizations to be determined at a later time. Indexing within MongoDB allows for fast access of data during the computations by arranging data entries in sequential order based upon some common key such as model iteration or stock name. The collections created to support this dissertation are as follows:

- Asset Prices: price history of each process for each iteration. Each entry contains the process name, asset price, process parameters at that iteration, and realized random numbers. Multiple realizations of the underlying processes are pre-computed and stored within the database for later access.

- Model Parameters: the estimated parameters for the various underlying process models. Each entry stores the associated estimated parameters such as estimated drift and volatility for geometric Brownian motion and auto-regressive terms for ARIMA models. These values are pre-computed for each iteration and stored in the database for later access.

- Neural Network Weights: from initialization to final model evaluation, the internal node weights and other model parameters for each model at each iteration are stored for future analysis. This also allows for models to be paused and resumed at a later date or for the forking and repeating of runs at a particular point to generate bootstrap samples of the outputs. The weights are indexed by model number, iteration, and process realization.

- Estimated Option Prices: for each realization of LSM option prices, the calculated option prices are stored. Multiple realization of option prices are calculated for each realization of the underlying asset processes.

7.2.1 Distributed Databases

Databases that are too large to be hosted on a single machine may be sharded or federated onto multiple computers. Sharded databases are such that indexed entries of the same database are distributed over several computers when the database is too large to be contained on a single machine. The database middleware will run queries and interact with the data as if the database were on a single computer without any additional interaction from the user. A useful application of this is when a redundant array of inexpensive disks (RAID) architecture is desired to maintain data integrity.

Federated databases are a collection of smaller independent databases each maintained on its own computer or collection of computers. When a query is submitted to a federated database, the database middleware will distribute the query to the many independent databases and combine the results into a single result. A useful application of this is when data owners do not want to distribute copies of their data; instead the queries to the federated system are submitted and run on the disparate database hardware.

For this dissertation, distributed databases were not implemented due to the size of the database only being a few gigabytes. Instead, static copies of the database were backed up on independent hardware including Amazon's simple storage solution (S3) buckets.

7.3 Computations

Bash scripts were written to integrate the AWS EC2 service with local computing resources as described in van Vliet and Paganelli (2011). The VMs run on Amazon servers and may be utilized for a few cents per machine hour and are initialized with a user defined number of processing nodes that range from 1 to 40 per instance. As many instances as needed may be initialized to support the user's computation needs.

Python scripts as described in Langtangen (2010) are remotely started on the running EC2 via remote SSH commands, assuming that the appropriate security protocols and network adjustments have been made. Upon job completion, the output files from VMs

Table 7.1: Hardware specifications for distributed computing.

No. Nodes	OS	Physical	No. Cores	RAM (GB)	Speed (GHz)
1	OS X	Y	2	4	2.0
1	OS X	Y	4	16	2.4
1	Ubuntu	Y	8	32	3.4
3	CentOS	Y	16	126	2.6
1	CentOS	Y	32	126	2.6
3	Amazon Linux	N	40	160	2.4

and local hardware are loaded into the local database and the S3 directory for archive and access for future post-processing activities.

A variety of hardware and operating systems were used in this dissertation and are listed in Table 7.1 with the associated number of machines, operating system, number of virtual compute cores, RAM, and processor speed. The variety was not an intentional aspect of this work, but does serve to demonstrate the code's portability over different systems. This distributed hardware was used to perform all computations and generate all outputs.

7.3.1 Computing Benchmarks

Due to the heterogeneous nature of hardware in a distributed computing network, it is important to benchmark the differences in performance for job optimization. Taking into account the difference in computational speed between the various computers allows for parameters such as number of simulated particles or paths to be scaled inversely proportional to the performance. This allows for jobs to finish at roughly the same time independent of the host computer's performance. Below in Table 7.2 is a benchmark test to measure the time needed for each computer to perform a least squares Monte Carlo (LSM) job with 10^5 iterations for each run using 3^{rd} order Laguerre polynomials for 50 time steps.

7.3.2 Random Number Generation

For this dissertation, Python's `random` module was used for random number generation. This module uses the Mersenne Twister as developed by Matsumoto and Nishimura (1998).

Table 7.2: Timing benchmarks for hardware used in this dissertation.

Name	No. Cores	OS	μ	σ
Computer 1	2	OS X	314.710s	6.496s
Computer 2	4	OS X	259.361s	8.669s
Computer 3	8	Ubuntu	117.939s	8.075s
Computer 4	16	CentOS	126.345s	11.065s
Computer 5	32	CentOS	125.567s	9.384s
Computer 6	40	Amazon Linux	130.459s	4.091s

The Mersenne Twister generates floating point numbers with 53 bit precision and has been proved to have a period of $2^{19937} - 1$.

7.4 Lessons Learned

During this phase of the dissertation, many common practices were learned by research, intuition, or trial and error. It is important to list these lessons learned to facilitate the efforts of future works. To list a few of the lessons learned:

- Use the appropriate programming language for the job. Depending on the necessary task, Python may be appropriate, or C, or R, or javascript.
- Write short modular code blocks or functions. This helps when the end goal may not be as solid as first imagined. It also helps with the portability, maintenance, and readability of the program.
- If planning to perform distributed and/or parallel computing, write code appropriately at the onset. No one wants to rewrite or restructure thousands of lines of code to be parallel compatible, thread safe, or distributable.

Chapter 8: Conclusion

8.1 Conclusion

To facilitate the develop of multidimensional evolutionary algorithms, an ontology was presented to describe the operations needed to project an ANN of one architecture onto a different architecture. This ontology allows for the existing MD-DE algorithm to be abstracted to include all multidimensional evolutionary algorithms. Once this abstraction was made, variations to the underlying algorithm were possible without changing the underlying differential evolution algorithm variant.

Performance of three proposed MD-DE algorithm variants were compared against the original MD-DE algorithm. These variants were created to illustrate that the original MD-DE algorithm was developed based upon the arbitrary operation of a additive projective operation. This operation prevents the training of all hidden nodes within certain populations of ANNs. Variants using random projections alleviate this issue by permuting the hidden nodes within an ANN prior to information exchange with the parent candidate solutions. Six test functions were chosen to provide a variety of function attributes such as polynomial, trigonometric, discontinuities, non-differentiability, and finite singularities to evaluate the algorithm's performance. It was discovered that the original MD-DE algorithm outperformed the three proposed algorithm variants in most cases for both single and multi layer models. Algorithms that used a random additive projection however, performed the second best over the range of parameters tested. It is this author's opinion that the theoretical advantages to using the random additive projective MD-DE algorithm outweigh the degradation to the algorithm's convergence rate as compared to the original MD-DE algorithm that uses an additive projection.

Once the appropriate optimization algorithm was chosen that allows for the training

of an ensemble of ANNs with potentially different architectures and allows for a non-differentiable objective function, the K-S test statistic was shown to be a valid objective function for ANN training. Since the proposed problem is an online learning problem, the ANNs must be capable of learning a process' dynamics based upon a single realization of a price path. A finite length training history is necessary to provide the ANNs a sufficient number of previous intervals to evaluate updated coefficients. It was shown that the K-S test statistic outperforms theoretical results when used as an objective function within the differential evolution optimization algorithm. This out performance is due to the DE algorithm being greedy with respect to minimizing the objective function.

For two simulated processes, the K-S test statistic was used as the objective function for an ensemble of ANNs to learn the underlying process' dynamics using the chosen random additive projection MD-DE optimization algorithm variant. The least squares Monte Carlo algorithm was used to calculate the American style option price based upon the learned dynamics of the ANN ensemble. Distributions of these estimated prices were presented for both processes. Analysis was then performed to confirm that the large variance seen in the distribution of option prices was in fact due to the variance induced by estimating process dynamics from a finite number of observations. Increasing the training length will theoretically reduce the variance of the estimated option prices to within any desired value.

8.2 Unique Contributions

The following list is a summary of the unique contributions made by this author in this dissertation:

1. An ontology is introduced to describe the various transformations and relationships between two neural networks.

2. A continuously tunable parameter for the complexity of an ensemble of neural networks is introduced in this dissertation. This parameter may be changed during training without restarting the models' training and may even be connected to an

external control algorithm.

3. Improvement and extension of an existing multi-dimensional differential evolution algorithm for training heterogeneous single-layer neural networks have been made to allow for efficient training of an arbitrary collection of neural networks with heterogeneous architectures. These same algorithms may be adapted to work for genetic algorithms or if the underlying process is stationary, then particle swarm optimization as well.

4. Three variations of the MD-DE algorithm are presented in this dissertation and compared against the original MD-DE algorithm. Two of these variations were created to rectify theoretical pitfalls of the original algorithm.

5. F-divergences and test statistics such as Kolmogorov-Smirnov and Anderson-Darling test statistics are studied and evaluated in this dissertation as valid objective functions for learning of conditional probability distributions in an online learning setting. In particular, the Kolmogorov-Smirnov test statistic is used to evaluate a model's learning in an online learning setting.

6. The LSM algorithm is used to estimate American style option prices based upon forecasted paths from the learned ensemble of neural networks with heterogeneous architectures. It is shown that a large variance in the output price is due to the variance induced from estimating parameters from a finite sample of a single realization.

7. ANNs are shown in this dissertation to be capable of learning asset price dynamics from a single realization of two different simulated processes. Training performance of these ANNs are analyzed and found to be within theoretical limits.

8.3 Future Work

8.3.1 Symmetry Breaking

Urfalioglu and Arikan (2010) uses the symmetries induced in the permutation of hidden

nodes as well as the symmetry of RBFs to achieve greater rates of convergence in the optimization of weights within ANNs. This process, referred to as symmetry breaking, provides a speed up by allowing the particles to converge on M equivalent global minimums in K dimensional space. Equation 8.1 describes how the number of M global minima is a function of the total number of parameters to be optimized K and the number of hidden nodes N_i for L hidden layers.

$$M = 2^K \prod_{i=1}^{L} N_i! \tag{8.1}$$

These equivalent global optima are a result of the arbitrariness of the numbering scheme used to convert the ANN weights into an N dimensional vector. Modifications to existing optimization algorithms is required such that the algorithm does not penalize against convergence to any of the equivalent global minima. The future work on this improvement will focus on adding the symmetry breaking feature to the MD-EA algorithms developed in this dissertation.

8.3.2 Multi-Asset Models

All the discussion above have been performed for a single simulated process, even though there has been no work presented that precludes the same methodology presented in this dissertation from being applied to a multi-asset model. These models would take the output multidimensional time series from a collection of correlated processes. It is known that the prices of many real assets are correlated with each other, and it would be interesting to explore if these correlations would improve the model performance in learning the underlying asset's dynamics. The extreme of this would be to take all assets listed in the Standard and Poor's 500 Index (S&P 500) since they all exhibit some amount of correlation with each other and to the overall index price.

When using multi-asset models, it is possible to create simulated multi-asset models based upon coupled stochastic differential equations and vector auto-regressive integrated

moving average (VARIMA) models as done in Hafner and Manner (2012). This would provide a simulated process with known correlations and dynamics. Complexity parameters such as AIC, AICc, and BIC may play a much stronger role in determining the optimal ANN architecture. The reliance on these parameters will be heavier due to the lack of an equivalent K-S test statistic for multidimensional distributions. A second approach would be to use a special case of an f-divergence such as the K-L divergence as the objective function, which that has been well studied in the multi-variate case.

8.3.3 Kalman Filtering

The resulting data shown in Figures 6.14 and 6.15 show that the estimated model parameters converge to their respective theoretical values. Kalman filtering as developed by Kalman (1960) and Kalman and Bucy (1961) provides a method to combine results from ensembles of ANNs each trained on the same realization of input data, but using different training history lengths. Figures 8.1 and 8.2 show how the values of calculated price converge to the desired price as the training history is increased for 40 different realizations of sample price paths. Instead of training a single ensemble of ANNs for a single realization, train an ensemble of ANNs for each value of training history from 10 to 160 as an example, and then apply Kalman filtering.

This future work would focus on discovering how Kalman filtering can be applied to the problem presented in this dissertation and reduce the variance of the output LSM price estimates. It is unclear at this time if Kalman filtering would be able to extract any additional information from a set of ANN ensembles.

8.3.4 Comparative Study of Objective Functions

A comparative study of the effects between using a likelihood, K-S test statistic, and f-divergence measures such as total variation distance of probability measures, K-L divergence, and Hellinger distance.

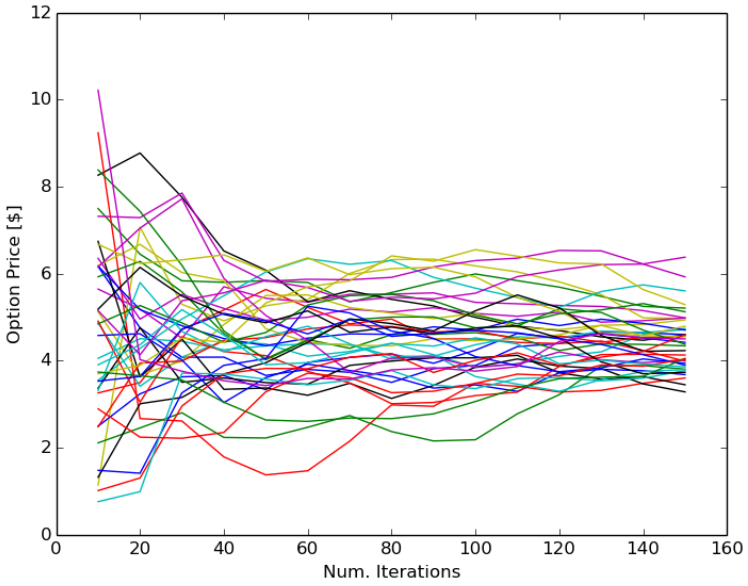


Figure 8.1: Convergence of option price as training history increases on same realization of sample data from arithmetic process.

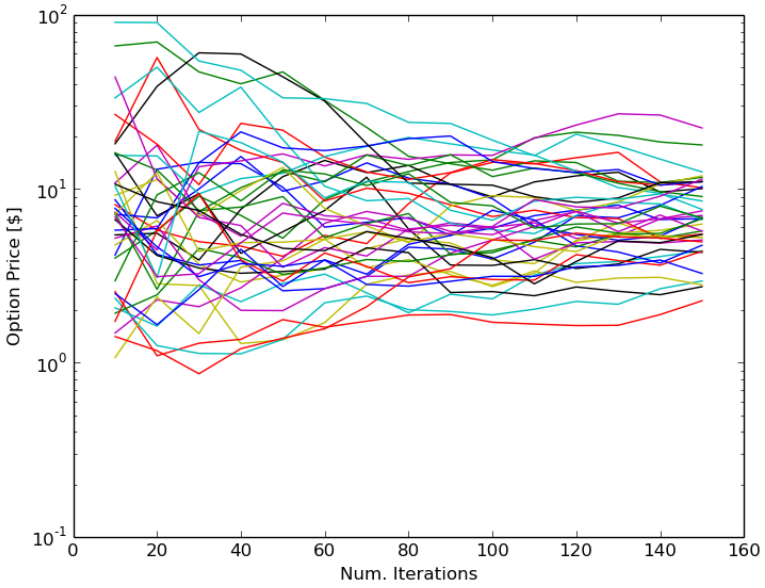


Figure 8.2: Convergence of option price as training history increases on same realization of sample data from GBM process.

Bibliography

Hussein A. Abbass. An evolutionary artificial neural networks approach for breast cancer diagnosis. *Artificial intelligence in Medicine*, 25(3):265–281, 2002.

Ken Aho, DeWayne Derryberry, and Teri Peterson. Model selection for ecologists: the worldviews of AIC and BIC. *Ecology*, 95(3):631–636, 2014.

Theodore W. Anderson and Donald A. Darling. Asymptotic theory of certain "goodness of fit" criteria based on stochastic processes. *The annals of mathematical statistics*, pages 193–212, 1952. 27

Panayiotis C. Andreou, Chris Charalambous, and Spiros H. Martzoukos. Pricing and trading European options by combining artificial neural networks and parametric models with implied parameters. *European Journal of Operations Research*, 185:1415–1433, 2008.

Gurdip Bakshi, Charles Cao, and Zhiwu Chen. Empircal performance of alternative option pricing models. *The Journal of Finance*, 52(5):2003–2049, 1997. 6

David M. Bashtannyk and Rob J. Hyndman. Bandwidth selection for kernel conditional density estimation. *Computational Statistics & Data Analysis*, 36(3):279–298, 2001. 12

David S. Bates. Testing option pricing models. Working Paper 5129, National Bureau of Economic Research, May 1995. 6

Julia Bennell and Charles Sutcliffe. Black-Scholes versus artificial neural networks in pricing FTSE 100 options. *Intelligent Systems in Accounting, Finance, and Management*, 12: 243–260, 2004. 7, 8, 38

Fischer Black and Myron Scholes. The pricing of options and corporate liabilities. *Journal of Political Economy*, 81:637–659, 1973. 82

M. Bortman and M. Aladjem. A growing and pruning method for radial basis function networks. *Neural Networks, IEEE Transactions on*, 20(6):1039–1045, June 2009. 7

Peter J. Brockwell and Richard A. Davis. Time series: theory and methods. *Springer, New York*, 1991.

Kenneth P. Burnham and David R. Anderson. Multimodel inference understanding AIC and BIC in model selection. *Sociological methods & research*, 33(2):261–304, 2004. 28

George Casella and Roger L. Berger. *Statistical Inference*. Duxbury, Pacific Grove, CA, second edition, 2002. 98

Joseph E. Cavanaugh. Unifying the derivations for the Akaike and corrected Akaike information criteria. *Statistics & Probability Letters*, 33(2):201–208, 1997. 28

CBOE. The CBOE Volatility Index - VIX. Technical report, 2015.

Mikhail Chernov, A. Ronald Gallant, Eric Ghysels, and George Tauchen. Alternative models for stock price dynamics. *Journal of Econometrics*, 116(12):225 – 257, 2003. Frontiers of financial econometrics and financial engineering. 6, 7

Gonzalo Cortazar, Miguel Gravet, and Jorge Urzua. The valuation of multidimensional American real options using the LSM simulation method. *Computers & Operations Research*, 35(1):113–129, 2008.

I. Csiszár. Eine informationstheoretische Ungleichung und ihre anwendung auf den Beweis der ergodizität von Markoffschen Ketten. *Publ. Math. Inst. Hungar. Acad.*, 8:95–108, 1963. 24

Benjamin D. Dalziel, Juan M. Morales, and John M. Fryxell. Fitting probability distributions to animal movement trajectories: Using artificial neural networks to link distance, resources, and memory. *The American Naturalist*, 172(2):248–258, 2008. 21

H. Dhahri and A. M. Alimi. The modified differential evolution and the RBF (MDE-RBF) neural network for time series prediction. In *Neural Networks, 2006. IJCNN '06. International Joint Conference on*, pages 2938–2943, 2006. 6, 7, 15

Habib Dhahri, Adel M. Alimi, and Ajith Abraham. Hierarchiacal multi-dimensional differential evolution for the design of beta basis function neural network. *Neurocomputing*, 97:131–140, 2012. 7, 38, 44, 48, 62, 64

Z. A. Dindar and T. Marwala. Option pricing using a committee of neural networks and optimized networks. In *Systems, Man and Cybernetics, 2004 IEEE International Conference on*, volume 1, pages 434–438 vol.1, Oct 2004. 6, 62

R. Glen Donaldson and Mark Kamstra. An artificial neural network-GARCH model for international stock return volatility. *Journal of Empirical Finance*, 4(1):17–46, 1996. 6, 15

Rick Durrett. *Probability Theory and Examples*. Cambridge University Press, New York, NY, fourth edition, 2010.

Sunil Elanayar V. T. and Yung C. Shin. Radial basis function neural network for approximation and estimation of nonlinear stochastic dynamic systems. *IEEE Transactions on Neural Networks*, 5(4):594–603, 1994. 7

Scott E. Fahlman and Christian Lebiere. The cascade-correlation learning architecture. 1989. 19

Christian M. Hafner and Hans Manner. Multivariate time series models for asset prices. In Jin-Chuan Duan, Wolfgang Karl Härdle, and James E. Gentle, editors, *Handbook of Computational Finance*, Springer Handbooks of Computational Statistics, pages 89–115. Springer Berlin Heidelberg, 2012. 15, 125

Simon S. Haykin. *Kalman Filtering and Neural Networks*. Wiley Online Library, 2001.

Guang-Bin Huang, P. Saratchandran, and Narasimhan Sundararajan. A generalized growing and pruning RBF (GGAP-RBF) neural network for function approximation. *IEEE Transactions on Neural Networks*, 16(1):57–67, 2005. 7, 50, 51

Jarmo Ilonen, Joni-Kristian Kamarainen, and Jouni Lampinen. Differential evolution training algorithm for feed-forward neural networks. *Neural Processing Letters*, 17(1):93–105, 2003. 58

Rudolph E. Kalman and Richard S. Bucy. New results in linear filtering and prediction theory. *Journal of basic engineering*, 83(1):95–108, 1961. 125

Rudolph Emil Kalman. A new approach to linear filtering and prediction problems. *Journal of basic Engineering*, 82(1):35–45, 1960. 125

J. Kennedy and R. Eberhart. Particle swarm optimization. In *Neural Networks, 1995. Proceedings., IEEE International Conference on*, volume 4, pages 1942–1948 vol.4, Nov 1995.

Serkan Kiranyaz, Turker Ince, and Moncef Gabbouj. *Multidimensional Particle Swarm Optimization for Machine Learning and Pattern Recognition*. Springer-Verlag, Heidelberg, Germany, 2014. 38, 50

Hans Petter Langtangen. *Python Scripting for Computational Science*. Springer-Verlag, Berlin, Germany, third edition, 2010. 118

Sovan Lek, Marc Delacoste, Philippe Baran, Ioannis Dimopoulos, Jacques Lauga, and Stéphane Aulagnier. Application of neural networks to modelling nonlinear relationships in ecology. *Ecological Modelling*, 90:39–52, 1996.

Francis A. Longstaff and Eduardo S. Schwartz. Valuing American options by simulation: A simple least-squares approach. *Journal of Computational Finance*, 18(1):121–143, 2012. ix, xii, 7, 89, 90, 92, 97

Michiharu Maeda and Shinya Tsuda. Reduction of artificial bee colony algorithm for global optimization. *Neurocomputing*, 148:70–74, 2015.

Malik Magdon-Ismail and Amir Atiya. Density estimation and random variate generation using multilayer networks. *IEEE Transactions on Neural Networks*, 13(3):497–520, 2002. 7

Spyros Makridakis. Why combining works? *International Journal of Forecasting*, 5(4):601 – 603, 1989. 6

Jianchang Mao and Anil K. Jain. A self-organizing network for hyperellipsoidal clustering (HEC). *IEEE Transactions on Neural Networks*, 5:2967–2972, 1994.

Makoto Matsumoto and Takuji Nishimura. Mersenne twister: a 623-dimensionally equidistributed uniform pseudo-random number generator. *ACM Transactions on Modeling and Computer Simulation (TOMACS)*, 8(1):3–30, 1998. 119

Robert L. McDonald. *Derivatives Markets*. Pearson, Upper Saddle River, NJ, third edition, 2013. 83, 84, 85

Allan D. R. McQuarrie and Chih-Ling Tsai. *Regression and time series model selection*. World Scientific, 1998.

Robert C. Merton. Option pricing when underlying stock returns are discontinuous. *Journal of Financial Economics*, 3:125–144, 1976.

Manuel Moreno and Javier F. Navas. On the robustness of least-squares monte carlo (LSM) for pricing American derivatives. *Review of Derivatives Research*, 6(2):107–128, 2003.

J. A. Nelder and R. Mead. A simplex method for function minimization. *The Computer Journal*, 7(4):308–313, 1965.

P. O'Connor and A. Kleyner. *Practical Reliability Engineering*. Wiley, 2011. ix, 32

Bernt Øksendal. *Stochastic Differential Equations: An Introduction with Applications*. Springer, New York, NY, sixth edition, 2013.

Ferdinand Österreicher. Csiszárs f-divergences-basic properties. *RGMIA Res. Rep. Coll*, 2002. 24

Sibarama Panigrahi, Yasobanta Karali, and H. S. Behera. Time series forecasting using evolutionary neural network. *International Journal of Computer Applications*, 75(10): 13–17, 2013. 6

J. Park and I. W. Sandberg. Universal approximation using radial-basis-function networks. *Neural Computation*, 3(1):246–257, 1991.

Laura L. Pullum, Brian J. Taylor, and Marjorie A. Darrah. *Guidance for the Verification and Validation of Neural Networks*. IEEE Computer Society, Hoboken, NJ, 2007.

Man Mohan Rai. Single-and multiple-objective optimization with differential evolution and neural networks. *VKI lecture series: introduction to optimization and multidisciplinary design*, 2006. 58

David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. Learning representations by back-propagating errors. *Nature*, 323:533–536, 1986. 7, 38

David W Scott. *Multivariate density estimation: theory, practice, and visualization*. John Wiley & Sons, 2015. 12

Robert H. Shumway and David S. Stoffer. *Time Series Analysis and Its Applications: with R Examples*. Springer, New York, NY, third edition, 2010. 15, 27

Bernard W. Silverman. *Density estimation for statistics and data analysis*, volume 26. CRC press, 1986. 12

Adam Slowik and Michal Bialko. Training of artificial neural networks using differential evolution algorithm. In *2008 Conference on Human System Interactions*, pages 60–65. IEEE, 2008. 58

H. Sompolinsky. Neural networks with nonlinear synapses and a static noise. *Physical Review A*, 34(3):2571–2574, 1986.

J. Michael Steele. *Stochastic Calculus and Financial Applications*. Springer, New York, NY, 2010.

Lars Stentoft. Convergence of the least squares Monte Carlo approach to American option valuation. *Management Science*, 50(9):1193–1203, 2004.

Michael A. Stephens. Tests based on EDF statistics. *Goodness-of-fit Techniques*, 68:97–193, 1986. 27

Josef Stoer and Roland Bulirsch. *Introduction to numerical analysis*, volume 12. Springer Science & Business Media, 2013. 54

Mervyn Stone. An asymptotic equivalence of choice of model by cross-validation and Akaike's criterion. *Journal of the Royal Statistical Society. Series B (Methodological)*, pages 44–47, 1977.

Rainer Storn and Kenneth Price. Differential evolution - a simple and efficient heuristic for global optimization over continuous spaces. *Journal of Global Optimization*, 11:341–359, 1997.

Stathis Tompaidis and Chunyu Yang. Pricing American-style options by Monte Carlo simulation: Alternatives to ordinary least squares. *Journal of Computational Finance*, 18(1):121–143, 2014. ix, xii, 7, 88, 90, 91, 98

Jinn-Tsong Tsai. Improved differential evolution algorithm for nonlinear programming and engineering design problems. *Neurocomputing*, 148:628–640, 2015. 58

Berwin A. Turlach. *Bandwidth selection in kernel density estimation: A review*. Université catholique de Louvain, 1993. 12

O. Urfalioglu and O. Arikan. Artificial Neural Networks, Symmetries and Differential Evolution. *ArXiv e-prints*, September 2010. 123

Jurg van Vliet and Flavia Paganelli. *Programming Amazon EC2*. O'Reilly Media, Inc, Sebastopol, CA, first edition, 2011. 118

J. Vesterstrøm and R. Thomsen. A comparative study of differential evolution, particle swarm optimization, and evolutionary algorithms on numerical benchmark problems. In *Evolutionary Computation, 2004. CEC2004. Congress on*, volume 2, pages 1980–1987, June 2004. 7, 59, 62, 66

Brian A. Wichmann and I. David Hill. Algorithm AS 183: An efficient and portable pseudo-random number generator. *Journal of the Royal Statistical Society. Series C (Applied Statistics)*, 31(2):188–190, 1982.

Xing Wu and Bogdan M. Wilamowski. Advantage analysis of sigmoid based RBF networks. In *Proceedings of the 17th IEEE International Conference on Intelligent Engineering Systems (INES13)*, pages 243–248, 2013. 7

Tiantian Xie, Hao Yu, and Bogdan Wilamowski. Comparison between traditional neural networks and radial basis function networks. In *IEEE International Symposium on Industrial Electronics*, 2011. 7

Yuhong Yang. Can the strengths of AIC and BIC be shared? a conflict between model identification and regression estimation. *Biometrika*, 92(4):937–950, 2005. 28

Curriculum Vitae

Originally from Houston, Texas, the author earned a Bachelor's of Science from Texas A&M University - College Station in Mechanical Engineering in 2009. He also earned a Masters of Science in Mathematics from Texas A&M University in 2011. In late 2011 he began working for the federal government and enrolled in George Mason University's PhD program in Computational Science and Informatics two years later. He has led an exemplary life.