Generative Adversarial Modelling of Financial Time Series

Shaun Markham

A DISSERTATION

Submitted to

The University of Liverpool

in partial fulfilment of the requirements
for the degree of

MASTER OF SCIENCE

Abstract

Generative Adversarial Networks (GANs) have become one of the most successful frameworks for unsupervised generative modelling. This model consists of two neural networks at loggerheads with one another. More specifically they form what is called a zero-sum game framework where both networks' gain or loss in utility is balanced by the loss or gain of the opposing participant.

Up until now, GANs have been used almost exclusively in image processing, providing a new method with which to generate suitable images when trained properly. Although very useful, this method will be changed to accommodate the aims of this project. This means that instead of images being fed into the model, vectors of time series information will be used, leading to a model able to generate suitable financial time series information.

The problem here is to create a suitable generative model that can generate stock price behaviours adequately, subject to training on historical prices of the stock to be generated.

The final tested output produced suitable results, with comparable values between real and generated data seen through the use of specialised time series evaluation measures.

Student Declaration

I confirm that I have read and understood the University's Academic Integrity Policy.

I confirm that I have acted honestly, ethically and professionally in conduct leading to assessment for the programme of study.

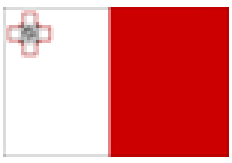
I confirm that I have not copied material from another source nor committed plagiarism nor fabricated data when completing the attached piece of work. I confirm that I have not previously presented the work or part thereof for assessment for another University of Liverpool module. I confirm that I have not copied material from another source, nor colluded with any other student in the preparation and production of this work.

I confirm that I have not incorporated into this assignment material that has been submitted by me or any other person in support of a successful application for a degree of this or any other university or degree-awarding body.

SIGNATURE _____

DATE September 19, 2018

The research work disclosed in this publication is partially funded by the
Endeavour Scholarship Scheme (Malta). Scholarships are part-financed
by the European Union - European Social Fund (ESF) -
Operational Programme II – Cohesion Policy 2014-2020
"*Investing in human capital to create more opportunities and promote the well-being of society*".



European Union – European Structural and Investment Funds
Operational Programme II – Cohesion Policy 2014-2020
"Investing in human capital to create more opportunities and promote the well-being of society"
Scholarships are part financed by the European Union - European Social Funds (ESF)
Co-financing rate: 80% EU Funds;20% National Funds

Acknowledgments

This project could not have been done without the help of my supervisor Prof. Rahul Savani. I would also like to thank my parents Joseph and Corinne, together with my sister Amy.

Finally I would like to thank the Endeavour Scholarship scheme (Malta), without which this project would not have been possible.

Contents

List of Figures

Chapter 1

Introduction

Generative Adversarial networks consist of two neural networks actively trying to outperform one another, forming what is called a zero-sum game framework where both networks' gain or loss in utility is balanced by the loss or gain of the opposing participant.

This document describes the background information and implementation done to create a Financial Time Series generating GAN (the code for which can be found in the Appendix). The purpose of this model is to generate time series which have similar behaviour to those seen in equivalent financial time series.

The aims of this project are as follows:

1. Creation of a model that generates evaluable financial time series.
2. Utilisation of a suitable evaluation method to measure the generator's output.

The created model, upon proper training, produces time series of variable lengths, evaluable through the use of time series models. The training data (original data) is created from different time series models through the use of the python arch package. Following are a few of the models used as evaluation measures:

- Autoregressive time series.
- Moving Average time series.
- Autoregressive Moving Average (ARMA) time series.

This is done to obtain specific coefficient values for the training data by fitting it to said time series models. These values are then compared to the results obtained by carrying out the same procedure but this time on the generated data. Apart from this, another evaluation measure called a hurst exponent was used in the same manner, which measures the long-term memory of time series. A visual test was also carried out by plotting the autocorrelation between different lagged components within both the training and generated data and comparing both.

1.1 Approach

The approach chosen for this project was one of iterative improvement on simple 'toy' problems. What this means is that the project was broken down into smaller

problems that could be tackled individually. Upon completion, the knowledge gained from completing these 'toy' problems would then be used to aid the creation of the final model. These 'toy' problems were created to reach the following milestones at different stages of the project.

- Acclimation to the tensorflow library.
- Implementation with simple neural networks.
- Introduction of recurrent elements within those same neural networks.
- Application of evaluation measures to test model performance.

The final model at the end of this approach is capable of remembering previous input due to the recurrent element, while also generating data that keeps behaviour seen in the training data.

1.2 Outcome and its Effectiveness

The created model was found to have adequate behaviour to that seen in the training data, with suitable evaluation measures being implemented to test and prove it.

The process of evaluation consisted of the following steps:

- Fitting the financial data (training data) to time series models, obtaining parameters about that data.
- Training the GAN on the financial data.
- Generating financial data from the trained model.
- Fitting the generated data to the same time series models mentioned at the start.
- Comparing the results.

In many of the utilised evaluation measures, the generated data was found to have similar components and behaviour, with calculated confidence intervals containing the target value in most cases. Towards the end of testing more complex evaluation measures like ARIMA and GARCH were carried out, examples of which can be seen towards the end of this document. These single tests produced less desirable results.

The deliverable for this project is the main python file used to train the network. This python file contains a conditional GAN implementation, but it is not set to work with the financial data provided as it was developed for the multi class sine wave 'toy' problem. Upon running this file it will start to train a GAN, saving an output frequently (during training) to two separate '.csv' files. These two files contain generated time series, with the difference being that one has been inversely transformed to remove the normalisation. Every epoch of training the script saves to file a graph showing the loss values of both generator and discriminator, so as to visualise the improvement of the model from epoch to epoch. Also, when training is finished, the script saves the model to a 'ckpt' file and also saves the scalar used to normalise the data initially.

Chapter 2

Background

2.1 Neural Networks

Artificial neural networks are one of the main models in modern machine learning techniques. They are loosely inspired by biological models, more specifically the way the brain works [6]. It is based on a collection of inter-connected units called neurons that individually have very simple functionality and are based on the individual perceptron. Each neuron accepts an input from the previous set of neurons, processes it and produces an output which is then passed on to the next layer. These neurons also contain weight values, used to compute each respective output. Figure 2.1 shows a generic neural network containing an input layer where data is fed into, some hidden layers that perform computations and an output layer where the information is aggregated into an 'answer'.

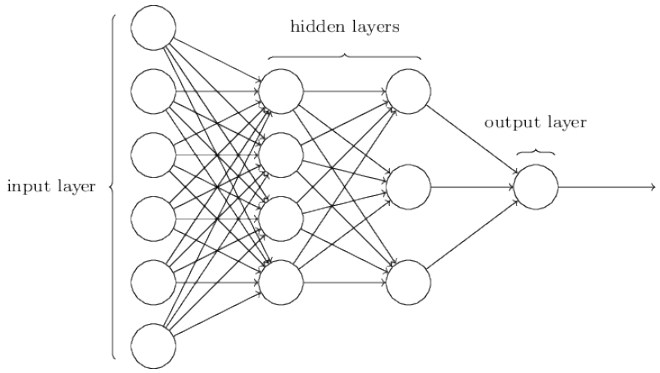


Figure 2.1: A Generic Neural Network[1]

They have been used increasingly over the past few decades due to Werbos's introduction of the backpropagation algorithm. This algorithm solved the XOR-problem, the problem of how to learn a multi-layered model consisting of different neurons each with their own individual weights.

2.1.1 The Backpropagation Algorithm

Given a fixed network structure like that seen in Figure 2.1, the appropriate weights for the connections in the network must be defined. The solution to this problem is to modify the weights of the connections leading to the output node, such that the nodes contributing the most to the final prediction are increasingly strengthened. Gradient descent, a standard mathematical approach, achieves exactly that. It is problematic here due to it requiring to take derivatives with

the step function used in the perceptron algorithm being non-differentiable[7]. Therefore this step function was swapped with one that can be differentiated with this most commonly being the sigmoid function. This sigmoid function is defined in equation 2.1.

$$f(x) = \frac{1}{1 + e^{-x}} \tag{2.1}$$

The gradient descent algorithm requires that an error function be minimised to achieve the lowest possible error value for a given $E(f(x))$ where E is the loss function. This can be better understood by viewing Figure 2.2. Here, at each iteration of the gradient descent algorithm, the error value or cost is reduced until reaching the global minimum, or the lowest possible error value.

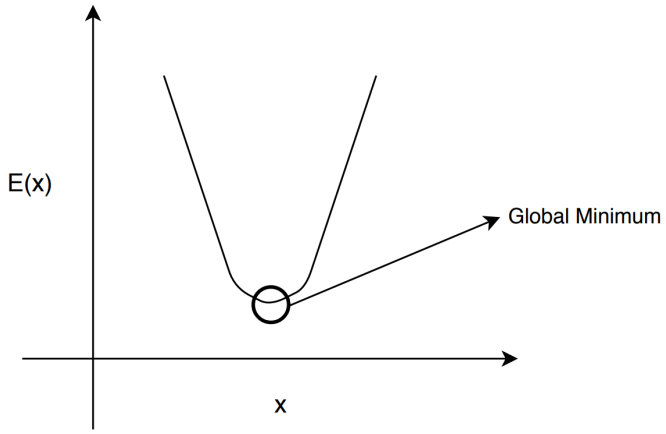


Figure 2.2: Graphed Gradient Descent

The loss function, otherwise known as the squared-error loss, can be viewed in equation 2.2. The y in the equation refers to the class label while $f(x)$ is the network's prediction obtained from the output node seen in Figure 2.1.

$$E = \frac{1}{2}(y - f(x))^2 \tag{2.2}$$

The crucial observation here is that based on the derivative, the slope of the function at any particular point can be extrapolated. As a simple explanation if the derivative is negative the function slopes downwards to the right while it slopes to the left if it is positive[7]. The approximation for the weight update of each individual neuron is given by equation 2.3.

$$W_{ab}^{t+1} = W_{ab}^{t} - \eta \frac{\delta E^2}{\delta W_{ab}} \tag{2.3}$$

Where η is the learning rate parameter and $\frac{\delta E^2}{\delta W_{ab}}$ is the sensitivity of the error E^2 to the weight W_{ab} [8]. This learning rate determines the step size and how quickly the search converges to the minimum seen in Figure 2.2. Performing the derivative seen in equation 2.3 produces the result in equation 2.4.

$$\frac{dE}{dw_{ij}} = (y - f(x))f'(x)w_i f'(x_i)a_i \tag{2.4}$$

This value is then multiplied by the learning rate and subtracted from that specific node's weight at time t to obtain the weight at time $t+1$. This derivation

applies to a network with one hidden layer, where the same strategy used to obtain this derivative being applied to networks with more hidden layers. The strategy itself utilises the propagation of error backwards through the network, leading it to be called backpropagation.

2.2 Recurrent Neural Networks

A recurrent neural network is one with feedback (closed loop) connections [9]. What this means in context, is that for the network seen in Figure 2.1 each node within the hidden layer has a connection to itself or another connection from neurons further forward. This self-connection is defined as an internal state or memory and is used to process sequences of inputs. These networks are designed to learn sequential or time-varying patterns. An example of this network configuration can be seen in Figure 2.3.

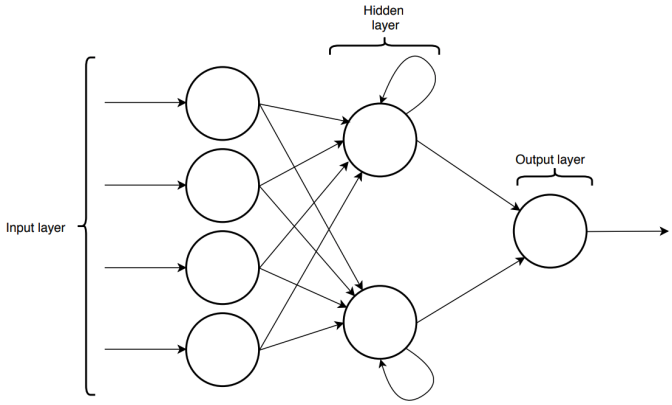


Figure 2.3: An Example of a Recurrent Neural Network

The previous section dealt with feed-forward networks, ones where signals are allowed to travel one way only: from input to output. Recurrent neural networks have signals travelling in both directions through the introduction of loops (feedback) in the network. An example of one of the first neural networks that implemented feedback was Hopfield's network which had its main use as associative memory [10]. What this means is that the network accepts an input pattern and generates an output as the stored pattern which is most closely associated with the input. The associate memory recalls the corresponding stored pattern and then produces a clear version of the pattern at the output.

2.2.1 Problems with Recurrent Neural Networks

Two widely known issues with training recurrent networks are those of the vanishing and exploding gradient problem described by Bengio *et al.*[11]. These problems arise when increasing the number of hidden layers, usually leading to an overall decrease in the accuracy of the network [1]. It comes about depending on the choice of activation function. Common activation functions such as 'tanh' or 'sigmoid' squash the input into a very small range (between 0 and 1). What this means is that a large change in the input will lead to a small change at the output, resulting in a small gradient. This problem is amplified when more layers are stacked on top of one another, such as in a typical neural network's hidden layers.

The exploding gradient problem on the other hand refers to a large increase in the gradient during training. This happens due to the explosion of long term components, which can grow exponentially more than short term ones.

2.2.2 Long-Short Term Memory

The LSTM model of recurrent neural networks overcomes the problem of learning long-term dependencies and was first proposed by Sepp Hochreiter and Jurgen Schmidhuber [12] and improved by Felix Gers' team [13].

The core idea behind the functionality of LSTMs is the cell state, or the horizontal line running through the top of the block in Figure 2.4. This cell state contains the information that is to be let through to the next layer in the network, with this being effected as is required by the internal operations of the LSTM[12][13][2]. These operations are defined in the following list and can also be seen in Figure 2.4. Here x is the input, h the output and C the cell state.

1. The first step is to decide what information is to be kept/removed from the cell state. This is done through the use of a sigmoid layer called the 'forget gate layer'. It uses the output from the previous time step $t-1$ and the input of the current time step to make this decision. This outputs a value between 0 and 1 with zero being 'forget all' and 1 being 'keep all'. This value is then multiplied with the previous cell state C_{t-1}.
2. The next step is to decide what new information is to be stored in the cell state. This consists of 2 parts which are:
   - A sigmoid layer called the 'input gate layer' decides what values are to be updated.
   - A tanh layer creates a vector of new candidate values $\bar{C}_t$ that could be added to the state.

   These parts are then combined through the use of a point wise multiplication operation and an addition to the previous state C_{t-1}.
3. Finally, the output is to be decided. This output is defined as being a filtered version of the cell state. This cell state is then put through a tanh and multiplied by the output of the sigmoid gate to only produce the desired output.

Figure 2.4 was taken from another figure containing stacked LSTMs (i.e. one LSTM passing information to another along the chain), leading to the explanation as to why there are arrows from the left and also to the right.

For this project and at the time of writing, a univariate time series is being considered as the main output to be predicted (generated). Therefore this explanation is suited for that purpose whereas a multivariate time series requires slightly different model parameters.

2.3 Generative Adversarial Networks (GANs)

Generative adversarial networks are an example of *generative models*, a model that takes a training set consisting of samples drawn from distribution p_{data} and learns to represent an estimate of that distribution[4]. What this means is that

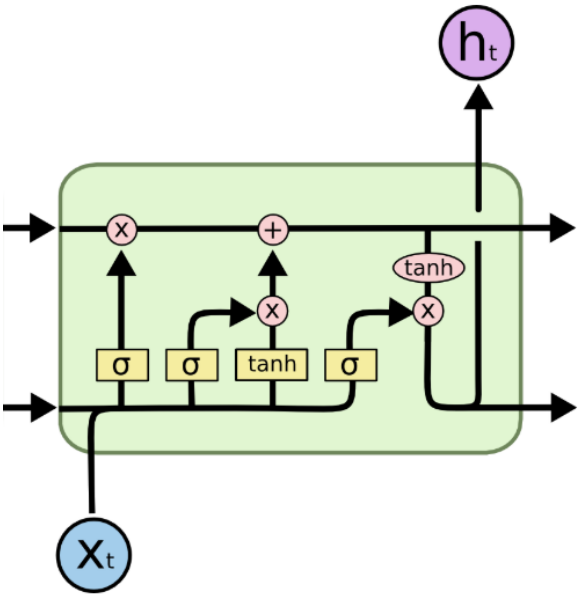


Figure 2.4: The Internal Working of an LSTM Block[2]

through training and an adversarial process, the model can effectively represent and reproduce data which is similar in distribution to the training data.

This networks consist of two models actively trying to outdo one another:

- A discriminator D
- A generator G

The discriminator estimates the probability of a given sample being real while the generator creates samples to try and 'fool' the discriminator. As an example, the generator can be thought of as a team of counterfeiters trying to produce fake currency without being detected. The discriminator on the other hand can be thought of as the police, trying to detect the counterfeit currency. The competition brings the best out of both models, leading to the discriminator not being able to tell true from synthetic at convergence. This process is derived from game theory and is known as a zero-sum game, zero-sum meaning that the rewards for each agent balance out to a final value of zero. Pictorially, the architecture of a generative adversarial network can be seen in Figure 2.5.

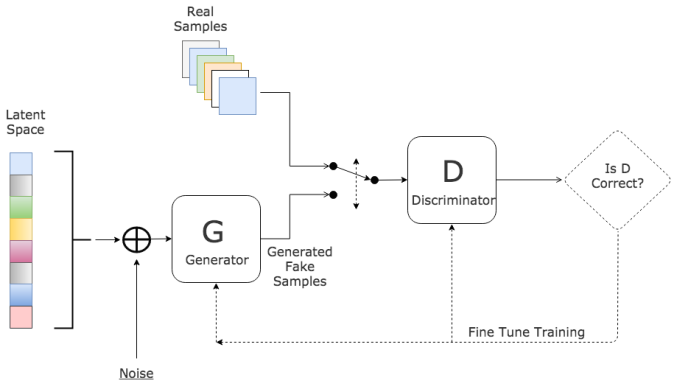


Figure 2.5: The Architecture for a Generative Adversarial Network[3]

What is being fed into G is a sample z from which it generates the synthetic data. According to the output produced by the discriminator D, backpropaga-

tion occurs resulting in a generated data distribution, closer to the real data's distribution than at the previous timestep.

GANs are part of a family of generative models that work via the principle of maximum likelihood. It's idea is to define a model that provides an estimate of a probability distribution (given parameters θ)with the likelihood being the probability assigned to the training data for a dataset containing *m* training examples. This principle says to choose the parameters that maximise the likelihood of that training data.

Mathematically, the idea is to maximise the discriminator D's decisions to detect all counterfeit currency. On the other hand the generator tries to increase the chance of D producing a high probability (meaning non-synthetic) for a given synthetic sample. Combining both, this produces something called a minimax game where optimising the loss function in equation 2.5 is the main goal[14].

$$\min_G \max_D L(D,G) = E_{x\sim p_r(x)}[log(D(x))] + E_{x\sim p_g(x)}[log(1-D(x))] \quad (2.5)$$

Where:

- $E_{x\sim p_r(x)}[log(D(x))]$ are the discriminator D's decisions over the real data p_r, the part that should be maximised.
- $E_{x\sim p_g(x)}[log(1-D(x))]$ are the discriminator D's decisions over the synthetic data p_g, which should be maximised, while also being what the generator wants to minimise by producing realistic generated synthetic data.

Figure 2.6 shows various stages in the training process (theoretical) of GANs where the following list acts as a key[4]:

- The blue dashed line is the discriminative distribution.
- The black dotted line is the data generating distribution.
- The green solid line is the generative distribution.
- The lower horizontal line is the domain from z being sampled.
- The horizontal line above is part of the domain x.
- The arrows connecting z and x show how the mapping from $G(z)$ to x imposes a distribution p_g on transformed samples.

This Figure can be summarised as follows:

(a) is an adversarial pair near convergence where the generated and actual distributions are similar while D is partially accurate.

(b) is where D is trained to discriminate samples from data.

(c) is after an update to G where the gradient of D has made the distribution of generated data closer to that of real data.

(d) is the convergence point where both G and D cannot improve with both distributions of generated and real being equal and the discriminator not being able to discriminate between the two.

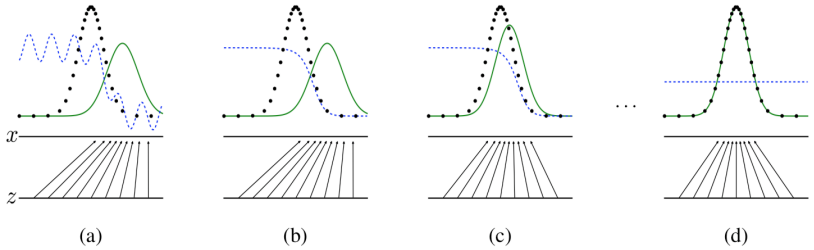


Figure 2.6: Different Stages in the Training Process of GANs[4]

2.4 Problems with GANs

Although great results have been seen in the application of GANs[15], training is known to be slow and unstable. They also have the problem of a hard to reach nash equilibrium [16], vanishing gradient [17] and mode collapse where the generator gets stuck and produces repeated images with little to no variety.

2.5 Conditional Generative Adversarial Networks

This GAN model is a conditioned version of the normal GAN. What this means is that both generator and discriminator are conditioned on some data y. In terms of the architecture y is fed into both G and D as additional input layers such that y and input are combined in a joint hidden representation[18]. In practice, conditional GANs have been used with success in [15] to generate time series in the medical field were data is difficult to obtain due to privacy concerns. The time series was properly generated by creating networks for both G and D that were recurrent to be able to 'remember' past input.

2.6 The Data

Data used for this project consists of financial information from provided by Prof. Savani. It is of varying reading frequency (minute/end of day) and contains the following fields.

Open: The start of trading on a securities exchange for the specific reading frequency.

High: The highest price at which a stock has been traded over the course of the reading frequency period.

Low: The lowest price at which a stock has been traded over the course of the reading frequency period.

Close: The closing price, i.e. price on closing of the trading session for the reading frequency period.

Volume: The amount of security that was traded over the reading frequency period.

From these features, the close price was mostly used in training the model.

2.6.1 Ethical Use of Data

The data provided by Prof. Savani consisted of daily readings for stock prices of traded companies. The company names were replaced with placeholders so as to conceal the identity of the company in question. All data was used ethically with none of it being shared with third parties throughout the project.

2.7 Evaluation Measures

Another big part of the project is the process of evaluation, or how to measure the 'goodness' of the generated data's distribution from G. When GANs were used with success before, such as in [19], the results could be evaluated manually due to the outputs being images (easily comparable - generated to real). In this case the same cannot be said for financial time-series and the graphs/data that are generated. A manual approach cannot be taken to looking at a graph and labelling it as a financial time series. Therefore a more experimental approach is taken through the use of already existing volatility models.

Such a volatility model is the generalized autoregressive conditional heteroskedasticity (GARCH), used to model variance in time series data x. The estimates of the variance of x are based on past values of that same estimate. This estimate of the variance is expressed as an autoregressive moving average of the noise sequence power[20].

When referring to time series and especially for the prediction of said series an important aspect is the stationarity. A stationary time series is one that holds for the following rules[21]:

- The mean of the series is not a function of time.
- The variance of the series is not a function of time.
- The covariance of the i_{th} and $(i+m)_{th}$ term should not be a function of time.

If the time series is stationary then the prediction becomes simpler due to the statistical properties being the same for the to-be-predicted data.

Following is a short description of the main evaluation measures that are utilised to determine the 'goodness' of the generated data.

2.7.1 Autoregressive Time Series Model

The first of these was of the autoregressive type. This is when the dependent variable is regressed against one or more lagged values of itself, with the formula for this seen in equation 2.6.

$$\begin{aligned} x_t &= \alpha_1 x_{t-1} + ... + \alpha_p x_{t-p} + w_t \\ &= \sum_{i=1}^{p} \alpha_i x_{t-i} + w_t \end{aligned} \tag{2.6}$$

When referring to the order of this model, the p in equation 2.6 is used to represent the number of lagged variables used within the model.

2.7.2 Moving Average Time Series Model

The second time series model was of the moving average type, similar to the autoregressive type. The difference between the two is that a moving average model is a linear combination of past white noise (residual) error terms as opposed to a linear combination of past observations like in autoregressive models. The definition for this model can be seen in equation 2.7.

$$\begin{aligned} x_t &= w_t + \beta_1 w_{t-1} + ... + \beta_p w_{t-p} \\ &= w_t + \sum_{i=1}^{p} \beta_i w_{t-i} \end{aligned} \tag{2.7}$$

When referring to the order of this model, the p in equation 2.7 is used to represent the number of lagged variables used within the model.

2.7.3 Autoregressive Moving Average Time Series Model

A moving average autoregressive (ARMA) model is a combination of the previous two, namely:

- An autoregressive model that tries to explain the momentum observed.
- A moving average model that tries to explain the past white noise (residual) error terms or the 'shocks' (unexpected events) observed.

The formula for this model can be seen in equation 2.8.

$$\begin{aligned} x_t &= w_t + \beta_1 w_{t-1} + ... + \beta_p w_{t-p} + \alpha_1 x_{t-1} + ... \\ &\quad + \alpha_p x_{t-p} \\ &= w_t + \sum_{i=1}^{p} \beta_i w_{t-i} + \sum_{i=1}^{p} \alpha_i x_{t-i} \end{aligned} \tag{2.8}$$

Chapter 3

Design

In this chapter, the design of the software that was developed in this project is outlined.

3.1 Original Design

The flow chart in Figure 3.1 shows the ideology with which the machine learning model is implemented.

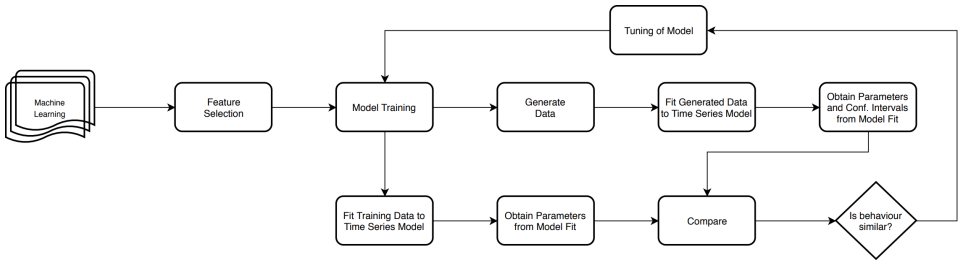


Figure 3.1: Machine Learning Model Development Path FlowChart

The machine learning model was written using the Tensorflow framework provided by Google[22]. This is an open source software library which is mostly used for neural network applications. The project design goes through the following incremental stages, allowing for acclimation to the topic area together with knowledge gain throughout.

3.1.1 Simple Experiments

Initially some simple experiments with artificial distributions were carried out. What this means is that a distribution of positive and negative 1s with a near 50/50 split was used as a dataset with the generator being learned on it.

3.1.2 Introduction of Recurrent Elements

The simple experiment was then extended on to deal with the problem of remembering past information. This was done by introducing LSTM blocks as a replacement for the previous normal dense network.

3.1.3 Conditional GAN Experiments

After carrying out experiments with created datasets the model was changed to accept 'context' information making the model conditional. The data used as context information, with suitable examples being:

- A class label.
- Statistics obtained from the time series.
- Evaluation output but applied to the original data and compared to that of the generated data.

An example of an already implemented conditional generative adversarial network can be found in [5]. Here the author utilised the MNIST dataset[24] and conditioned the model by adding a one-hot-encoded version of the required output from the generator. In this case the author wanted the model to generate the number 7. After training the model, the generator produced the output seen in Figure 3.2.

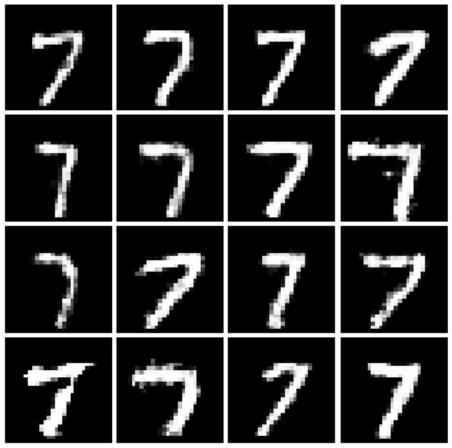


Figure 3.2: Output from the Conditional GAN After Training with the MNIST dataset[5]

3.1.4 Output Evaluation Experiments

Finally, the topic discussed in Section 2.7 was implemented by testing different methods to perform time series analysis[25]. The idea behind this portion of the project was to show that the data created from the generator resembles that used for training (pre-defined time series). There are numerous models which could be applied[26], some of which can be seen in the following list:

- Autoregressive models: used when the dependent variable is regressed against one or more lagged values of itself.
- Moving Average Models: similar to autoregressive models but the values being regressed against are error terms.
- Autoregressive Moving Average Models: the merging of the previous two models.

Data was created using each of these models with specific parameters that were used to test if the generator could reproduce those same parameter values. A comparison was carried out by fitting the generated data to the same model

used to create the time series which then returns the coefficient values in question.

3.2 Changes to original design

Although the implementation mentioned in the previous section of this chapter was carried out, some changes were made to obtain the final working version of the project model. Initially, some simple experiments were carried out with the aim of acclimating to the topic area. These simple experiments took up the bulk of the time for implementation, as at each stage additional features were added on top of the previous experiment's code. The purpose of each experiment mentioned in Chapter 4 can be found in the following list.

- Positive Negative Distribution GAN: This model was implemented with the use of being the initial acclimation to tensorflow and keras.
- Sine Wave GAN: This model switched the implementation to just tensorflow while introducing a recurrent element within the networks as LSTMs.
- Multi-Class GAN: This model furthered the implementation of the previous experiment while further tuning both the network architectures and the training methods. It was also carried out to see the behaviour of the model when trained with multiple different types of Sine Waves.
- Time Series GAN: This model contained the previous model's implementation but was instead trained on specific time series models. This was done to check the model's ability to accurately reproduce the time series in question.

Apart from this, the original design also includes experiments for a conditional GAN. This was implemented in the Multi-Class GAN, but was then not used when financial information was introduced as training material. This list and subsection are displayed here to show that the original design methodology was followed, while introducing each of the 'toy' problems.

Chapter 4

Implementation

4.1 'Toy' Problems

4.1.1 Positive Negative Distribution GAN

The first 'toy' problem used to create the initial implementation of a GAN was one dealing with a simple distribution of positive and negative numbers $(1/-1)$. The data here was created as a random choice between two positive and negative numbers $(1/-1)$. The rational behind this was to obtain a generator network that would adequately recreate this 50/50 split of positive and negative numbers.

This model was built using keras, a python library acting as a wrapper working through the tensorflow library. This is being mentioned as later on in the project native tensorflow was used.

The network here did not contain any recurrent components, i.e. components that could 'remember' previous samples due to it only needing to learn a simple 50/50 distribution. Instead, both the generator and discriminator contained normal neural network dense layers with the discriminator terminating with a sigmoid activation due to the requirement of it having to output probabilities between 0 and 1.

Data was created by taking a random choice between 1 and -1 100 times. The algorithm for training the model is as follows. First a random vector is generated from a normal distribution and fed into the Generator to create an output. Then, the discriminator is trained separately with both this output vector and the actual data. The loss from both of these training instances are then combined and used to train the whole (combined) model, in the process training the generator also. This is done once every iteration. After this, the generator was called to produce a vector of 50 numbers, with these numbers being in the range -1 to 1. This result was then rounded to the nearest integer and counted.

Results

Simply put, this distribution of positive and negative numbers was learnt fairly quickly by the generator. More specifically, it was able to produce a nearly perfect 50/50 split, with 27 out of the 50 being positive and 23 being negative after 100 iterations.

4.1.2 Sine Wave GAN

Next, another 'toy' problem was carried out to complete an initial implementation of the generator and discriminator containing Long-Short-Term-Memory (LSTM) modules. These were used due to their ability to 'remember' long stretches of information from past time-steps, with the code now being written using just tensorflow as opposed to using the keras wrapper used for the previous problem.

Both the generator and discriminator were set up as simple chained LSTMs, each consisting of two layers, with each layer having a relatively small number of units. The sine wave was generated using the numpy library command *np.sin()*, from which vectors of contiguous data points would be used to train with at one go. The LSTM here takes data which is 3 dimensional in the form *(Samples, Time Steps, Features)*, defined as:

- **Samples**: The number of sequences within a batch.
- **Time Steps**: The number of observations within a sequence.
- **Features**: One feature is one observation within a time step.

Training this time is done by picking random points along the generated sin wave and feeding a set of samples as batches.

Results

After training it was realised that the generated sine wave was not sampled properly and so the obtained results were not of the best quality. Obtaining results here was more difficult due to this and as a result a rolling mean was applied to the output of the generator after training. An example of the generator's output can be seen in Figure 4.1

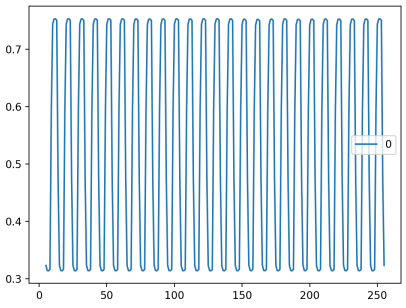


Figure 4.1: The Resulting Generated Sine Wave After Training

4.1.3 Multi-Class GAN (Extension on Sine Wave)

Here, the aim of this 'toy' problem was to have multiple different sine waves with separate amplitudes and frequencies as data and investigate if the generator is able to adequately learn the overall distribution. The training method here was changed after consultation with a recently published paper that aimed to generate appropriate medical data from patient records[15]. The models for both generator and discriminator were also changed. As an example, the architecture for the discriminator can be seen in Figure 4.2 where instead of the network

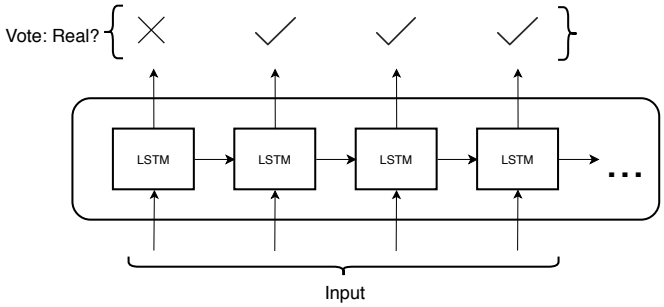


Figure 4.2: The Layout of the Discriminator

being stacked layers of LSTMs it was implemented as one layer of 100 LSTMs. This can be thought of as each LSTM having a 'vote' for the respective input being real or synthetic. These votes would then be counted and used as the output (in the range 0-1).

Training

Each batch contained several permutations of a sine wave with differing amplitudes and frequencies depending on the max and min values set when creating them. When training, the model obtains batches sequentially from the large matrix of different sine wave permutations and learns one batch at a time. After going through the length of the data (i.e. finishing an epoch), the data is shuffled and this process is repeated again.

Another important point which was implemented dealt with the amount of training that each network was exposed to. Consider the case when the discriminator is always better than the generator and assume that the generator aims to minimise $-\log(D(G(z)))$ instead of $log(1 - D(G(z)))$[4]. If the discriminator predicts the generator's images as real then the mentioned objective goes to 0, and so the generator update will be inefficient. In other words, if the discriminator describes the generator's outputs as being real always then the generator has no incentive to become better. This was added in the project by training the discriminator 5 times on each batch as opposed to once for the generator.

Applying Conditionality

Mentioned in the project specification, making the GAN conditional was done in this version. What conditionality implies is that given specific information together with noise at the generator's input, it can be 'pointed' to output a specific version of results. This means that given the right input the generator can output any version of a sine wave subject to the fact that it has 'seen' that version during training. When also fed amplitudes (or frequency) for each sine wave together with noise, then at convergence the generator will output sine waves with the required amplitude (or frequency)[27].

This was done by creating placeholder variables to hold this extra information and then feeding it into both networks when training. It was tested three times, each with different values input as conditions into both networks, defined as follows:

- Feeding repetitions of the same sine wave together with many others, but inputting 1 as the condition for the sine wave required at convergence.
- Feeding the frequency as the condition with each sine wave, then at convergence feeding the desired frequency for the to-be-generated sine wave.
- Feeding the amplitude as the condition with each sine wave, then at convergence feeding the desired amplitude for the to-be-generated sine wave.

Results

Figure 4.3 is a generated sine wave after several epoch iterations, showing that the generator has learnt the distribution and is capable of reproducing variations of it.

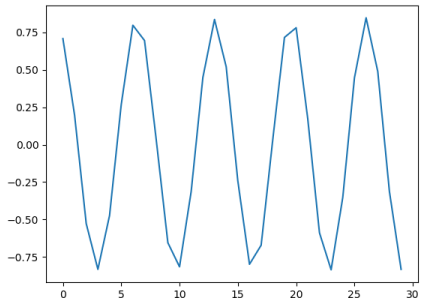


Figure 4.3: The Resulting Sine Wave Generated After Training

With respect to the conditioned version of the same model, the generator was able to reproduce sine waves when given correct parameters.

4.1.4 GAN for Time Series Models

The next 'toy' problem focused on statistical models used later on as evaluation measures. These models deal with time-series analysis and so for each, corresponding data was created that matched the model to check whether the generator could reproduce that behaviour. The models used here were from a python library called arch, a very useful tool for financial econometrics.

Autoregressive Model

The first evaluation measure tried here is of the autoregressive type discussed in Section 2.7.1. The initial step was to generate an autoregressive time series to be used as training data, which was done using the following code.

```
1   a = 0.6
2   for j in range(n_lists):
3     x = w = np.random.normal(size=n_samples)
4     signals = []
5     cond_sig = []
6     for j in range(n_samples):
7         signals.append(a * x[j  1] + w[j])
8     samples.append(np.array(signals).T)
```

This creates multiple different autoregressive time series, each with an alpha value of 0.6. The equation within the inner for..loop does this. The first

for..loop defines the number of time series created while the second is the length of each.

Testing & Results

An alpha value of 0.6 was applied to evaluate the performance of the generator in creating an autoregressive time series that shows the same alpha value. After training, the created time series from the generator were fit to an autoregressive model found in the arch library. Then, a function call was used for this model to return the components detected from within the time series. In this case that component was the alpha value set initially. From a batch of ten 1000 step time series generated, alpha values from 0.4 to 0.7 were seen with 0.63 being the closest to the required 0.6 and having an average over the whole batch of 0.588. These results can be seen in Appendix B.1.

Moving Average Model

The same method is used here to define the training data as in the previous model (autoregressive). This time, instead of defining the equation with which to generate the time series, a simple function call to the arch package is carried out, seen in the following code snippet.

```
1    smt.arma_generate_sample(ar=ar, ma=ma, nsample=n)
```

This line is placed into the inner for..loop seen in the code snippet in Section 4.1.4. Although it features a call to a function called *arma_generate_sample()*, the ar (autoregressive) component is set to 0 while the ma (moving average) component contains a value of 0.6, the beta value.

Testing & Results

The resulting generated batch of time series had beta values ranging from 0.3 to 0.65, with an average over 10 batches of 0.48. These results can be better seen in Appendix B.2.

Moving Average Autoregressive Model (ARMA)

The method with which data was generated here is the same as in the Moving Average model, this time defining coefficients for both time series components since this model is a combination of the previous two.

Testing & Results

Therefore, two different variables (alpha and beta) were used to create the time series. Also, the models were created to be of the second order, meaning two variables are set for each parameter, values for which can be seen in the following table.

	Value
α_1	0.5
α_2	-0.25
β_1	0.5
β_2	-0.3

Upon training, the generated data was then fit to an ARMA model from the arch library and produced the following confidence intervals for one generated time series.

	Lower Bound	Upper Bound
α_1	0.233	0.881
α_2	-0.378	-0.222
β_1	-0.117	0.555
β_2	-0.389	0.101

Comparing the two tables, each value for the variables created in the first table fit within the double sided 95% confidence intervals obtained from the generated data. The confidence intervals and the target values for each component can be seen in Appendix B.3, where a batch of ten results were graphed.

4.2 Financial GAN

The 'toy' problems were used to build up the model with the aim being aiding in the implementation of the final Financial GAN. Here, financial data was used for training, with this being both scaled to the range -1 to 1 and differenced using the numpy library. The differencing was done to make the time series stationary, which is where the means, variances and co-variances do not change with time. The data used was the daily close price for a specific stock over the span of 14 years. After being scaled and differenced, the resulting time series (training data) is manipulated to create runs of 100 contiguous data points, with those vectors then being shuffled.

Here, together with the previous time series models, other evaluation measures were also used to grade the output from the generator. These were:

- The Hurst Exponent: This is used as a measure of the long-term memory and the autocorrelations of the time series. It is a number between 0 and 1, with the ranges seen below having the following characteristics:
  - 0-0.5: A time series with long-term switching between high and low values in adjacent pairs.
  - 0.5: A completely uncorrelated series.
  - 0.5-1: A time series with long-term positive autocorrelation (trending).
- Autocorrelation Plots: This maps out the relationship between components within a time series.

4.2.1 Final Implementation Walk-through

Importing of Data and Preprocessing

Initially the data is imported using a simple 'pandas' command *pd.read_csv*. This data is then differenced to remove stationarity (see section 2.7). After this, each data point is appended to a separate array with the subsequent 99 data points also appended, so as to create a multi-dimensional array of data with each row containing a run of 100 numbers (in this case 100 contiguous readings for stock prices). Then, the array is scaled to the range -1 to 1 using the *MinMaxScalar()* from the *sklearn.preprocessing* library.

Initialisation and Model Definition

After the importing and preprocessing of data, the next step is to define the environment and set up the machine learning model.

First, all tensorflow placeholder variables were defined. These placeholders are a structure, with data being assigned to it to be used at a later date. As described in Section 4.1.2, LSTM modules take in data which is 3 dimensional. This correlates with the way the placeholders are defined here.

```
1  X = tf.placeholder(tf.float32, [batch_size, seq_length, num_features])
2  Z = tf.placeholder(tf.float32, [None, None, latent_dim])
```

These two lines show the initialisation of the previously mentioned placeholders with Z being the distribution (normal) fed into the generator and X being what the real data fed into the discriminator. The second part of the initialisation defines the shape of the data to be fed, in this case 3 dimensional data.

Next, the networks and losses were defined in the *def_loss()* method, with the networks being defined as follows.

- First the generator is created, using tensorflow's *LSTMCell* and *dynamic_rnn* modules.
- Then the discriminator is defined with the same method. This time the output is activated with a sigmoid activation function to bound the output in the range 0 and 1. This discriminator is defined to accept output coming from the pool of 'real' data.
- Finally, another discriminator is defined with the same method used as in the previous point. This discriminator accepts output coming from the generator (synthetic data).

After this the loss functions are defined using tensorflow's *reduce_mean* method, with it aiming to reduce the sigmoid cross entropy of the respective data in question.

Finally, the optimisers are defined. Each of the networks had a different optimiser with the generator using an Adam optimiser and the discriminator using a *GradientDescentOptimizer* optimiser.

At this point the initialisation has been performed and a tensorflow session is created. Then all defined variables are initialised by running this session with the tensorflow initialiser as an argument, as in the following code snippet.

```
1  sess = tf.Session()
2  sess.run(tf.global_variables_initializer())
```

Training

Training is done by separately training both the discriminator and generator. More specifically, the session previously defined in the initialisation phase is run on the optimisers. A batch of data is also defined which is used to train on for that specific instance. This can be seen in the following code snippets.

```
1  _ = sess.run(D_solver, feed_dict={X: X_batch, Z: Z_batch})
```

Here the discriminator is trained on two separate batches. The *X_batch* contains the 'real' data, a portion of daily stock readings (close prices). The *Z_batch* on the other hand is a sample from a normal distribution, which is used to generate 'synthetic' data from the generator and then feed it into the discriminator for training. The second code snippet shows the running of the *G_solver*, or the generator optimiser.

```
1  _ = sess.run(G_solver,feed_dict={Z: sample_Z(batch_size, seq_length, latent_dim)})
```

4.2.2 Results

Each of the previous time series models were used as evaluation measures here. Initially, the input data was fit into the various models so as to obtain the respective alpha and beta values, together with the hurst exponent which was found to be 0.51.

An example of a 5000 step generated time series having a hurst exponent of 0.511 superimposed onto the original data can be seen in Figure 4.4.

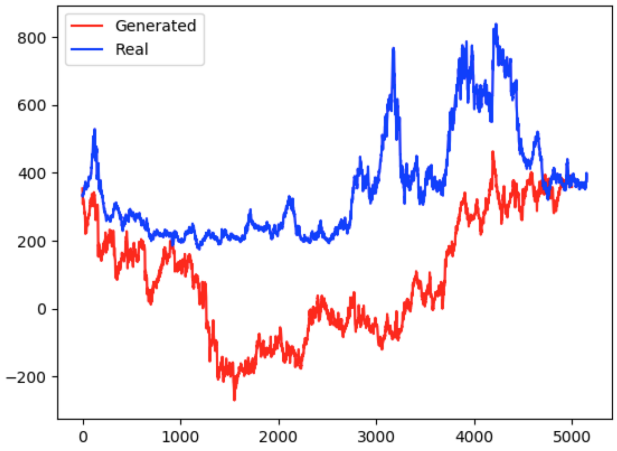


Figure 4.4: An Example of the Generated Time Series Superimposed with the Original Data

Analysed Hurst Component

The generated data is scaled and differenced, due to the input data being of that form during training. The hurst value for each of the generated 5000 step time series were all in the range 0.48-0.52. These results can better be viewed in Figure 4.5. This shows how the average hurst value obtained from a batch of 10 generated time series is very similar to the target value of 0.51 seen in the training data.

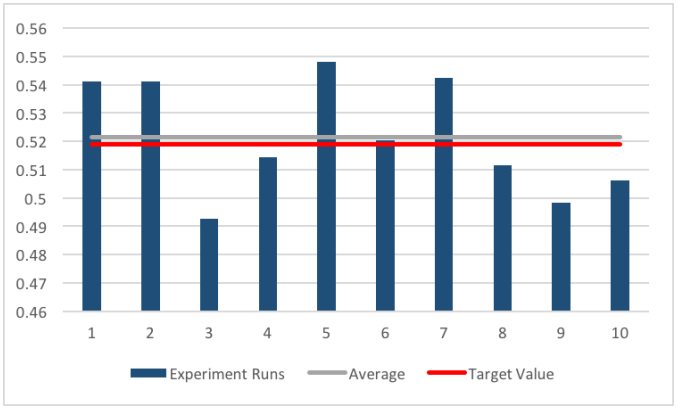


Figure 4.5: The Observed Hurst Components for Each Generated Time Series Compared with the Original Data's Hurst Component

Autoregressive Model Fit

When fitting the original data to a first order autoregressive model provided in the arch package, it returned an alpha value of 1.046 whereas the best alpha value from the batch of generated time series was found to be 0.998. The obtain alpha component values for each of the generated time series can be seen in Figure 4.6. Here, although the average is not as close to the target value as in the previous evaluation measure they are still quite comparable.

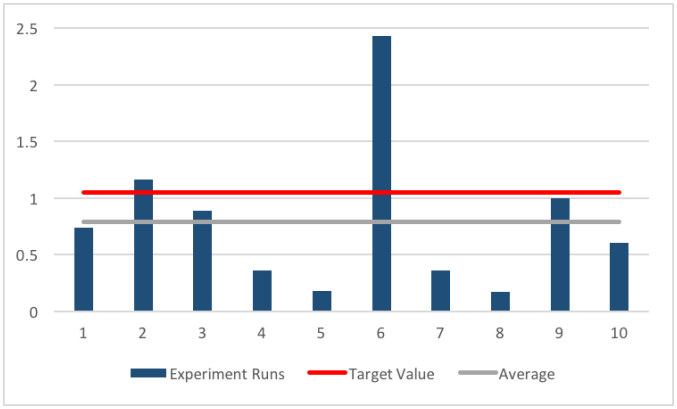


Figure 4.6: The Observed Alpha Component Values Compared with the Original Data's Observed Alpha Component

Moving Average Model Fit

The same was done this time for a first order moving average model. The real data returned a beta value of 0.987 whereas the best generated time series was found to be 0.989. The Beta values obtained for each of the generated time series from a batch of 10 can be seen in Figure 4.7, together with a comparison with the value obtained when fitting this model to the real data. Again the results here show a close resemblance between the average beta value obtained from the batch of generated time series and that seen in the original data.

Figure 4.7: The Observed Beta Component Values Compared with the Original Data's Observed Beta Component

ARMA Model Fit

The (1,1)ARMA model, a combination of the previous two models, returned an alpha value of 0.999 and a beta value of 0.0501 on the real data. The best values obtained from the generated time series were an alpha value of 0.999 and a beta value of 0.0573. Due to the Alpha values being very close to that seen in the original data, the Beta value was chosen to be displayed here. The confidence interval lower and upper bounds are shown in Figure 4.8, together with the target value in red.

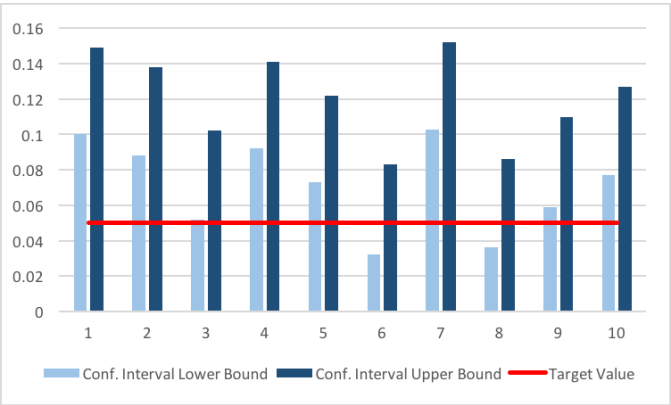


Figure 4.8: The Observed Confidence Intervals for the Beta Value when Fitted to an ARMA Model

Autoregressive Integrated Moving Average Model (ARIMA) Fit

Further to the previous models, an ARIMA model was also fit to further compare real to generated data. Before fitting, the best order was picked by cycling through combinations of orders (for each variable) and picking the combination that produces the lowest Akaike Information Criterion. In this case it was found to be (4,1,4). Another variable is added when compared to the previous models which references the number of times the series is differenced (the 1 here). The results obtained from fitting the real dataset to the model can be seen in the following table.

	Value
α_1	0.501
α_2	-0.213
α_3	-0.457
α_4	0.700
β_1	-0.450
β_2	0.172
β_3	0.487
β_4	-0.696

Whereas the confidence intervals for the generated data's variables can be seen in the following table.

	Lower Bound	Upper Bound
α_1	-0.306	1.060
α_2	-0.772	0.646
α_3	-0.115	0.572
α_4	-0.430	-0.004
β_1	-0.920	0.449
β_2	-0.518	0.836
β_3	-0.582	0.047
β_4	-0.059	0.420

Comparing the two tables, from the observed variable values in the first table, four from the eight variables fit within the double sided 95% confidence intervals obtained from the generated data.

Generalized Autoregressive Conditionally Heteroskedastic Model (GARCH) Fit

Finally, a GARCH model was fit on the original data, with the results then being used to validate the 'goodness' of the generated data. A GARCH model is an ARMA model but applied to the variance of the time series, returning three variables: alpha, omega and beta. The results obtained from fitting the real dataset to the model can be seen in the following table.

	Value
ω	10.548
α	1.000
β	0.000

Whereas the confidence for the generated data's variables can be seen in the following table.

	Lower Bound	Upper Bound
ω	19.457	38.842
α	0.933	1.067
β	-6.768e-02	6.768e-02

Comparing the two tables, from the observed variable values in the first table, the alpha and beta variables fall within the generated data's double sided 95% confidence interval.

Autocorrelation Plots

Figure 4.9 shows the autocorrelation plots for the real and generated data. This plot shows intra-data correlation. As seen in this figure, for the differenced data in both cases, there is no large serial correlation with past lagged values.

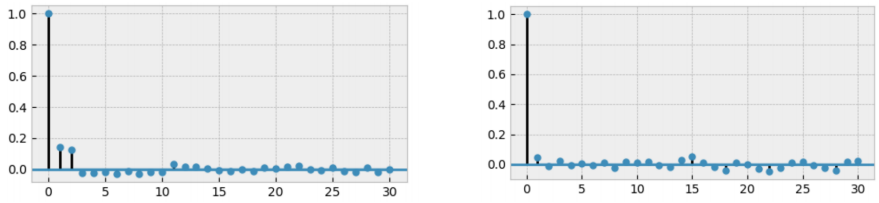


Figure 4.9: From left to right, Autocorrelation of Generated and Real Data

4.3 Discussion, Conclusion and Further Work

This section deals with the closing remarks of this project, together with what the next steps are to better improve it. In terms of results, the aims set out initially were met. Adequate financial information was generated using generative adversarial networks and evaluation was carried out using statistical time series models. Some results were not completely satisfactory, especially with respect to the ARIMA and GARCH evaluation measures . With further investigation into the statistical fields and some additional tweaking, it is possible that this model could produce more adequate time series.

4.3.1 Use of Conditionality

Although conditionality was implemented in the Multi-Class GAN 'toy' problem and tested with success, it wasn't used with the final Financial GAN. This was due to the inability to find a suitable condition as well as time factors.

4.3.2 Added Methods to Improve Stability

A problem when dealing with GANs is that of stability. The collapsing of the model was experienced multiple times during training, with it reverting to outputting straight lines for the sine wave 'toy' problem or outputting bloated values within the financial GAN. For the final GAN this was fixed by reducing the amount of overall training, but should be further investigated.

4.3.3 Further Investigation into Evaluation Measures

Even though multiple evaluation measures were used to test the output from the model, a greater understanding of these statistical models is required to additionally improve this project, and should therefore be investigated further.

4.3.4 Fixing Restoring the Model

A problem was encountered when dealing with restoring the model after training. Upon restoring it, the output from the generator could be seen to 'bloat' the

data's range or in some cases even restrict it to a very small value. One of the first next steps for this project would be to restore the model fully such that it reproduces the results more closely.

4.3.5 Investigation Into the Use Of Added Features

The project was trained using one column from a comma separated variable file containing others (Close column taken from data containing Open, High, Low, Close and Volume columns). A good next step would be to investigate if adding the other columns would aid the overall performance of the network.

Bibliography

[1] M. A. Nielsen, "Neural networks and deep learning," 2015. [Online]. Available: http://neuralnetworksanddeeplearning.com

[2] "Understanding lstm networks – colahâĂŹs blog." [Online]. Available: http://colah.github.io/posts/2015-08-Understanding-LSTMs/

[3] "Generative adversarial networks âĂŞ hot topic in machine learning." [Online]. Available: https://www.kdnuggets.com/2017/01/generative-adversarial-networks-hot-topic-machine-learning.html

[4] I. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville, and Y. Bengio, "Generative adversarial nets," in *Advances in neural information processing systems*, 2014, p. 2672âĂŞ2680.

[5] A. Kristiadi, *generative-models: Collection of generative models, e.g. GAN, VAE in Pytorch and Tensorflow*, Jun 2018. [Online]. Available: https://github.com/wiseodd/generative-models

[6] "Artificial neural networks as models of neural information processing — frontiers research topic." [Online]. Available: https://www.frontiersin.org/research-topics/4817/artificial-neural-networks-as-models-of-neural-information-processing

[7] I. H. Witten and E. Frank, *Data mining: practical machine learning tools and techniques*, 2nd ed., ser. Morgan Kaufmann series in data management systems. Morgan Kaufman, 2005.

[8] P. J. Werbos, *Beyond Regression: New Tools for Prediction and Analysis in the Behavioral Sciences*. Harvard University, 1975, google-Books-ID: z81XmgEACAAJ.

[9] L. V. Fausett, *Fundamentals of neural networks: architectures, algorithms, and applications*. Prentice-hall Englewood Cliffs, 1994, vol. 3.

[10] J. J. Hopfield, "Neural networks and physical systems with emergent collective computational abilities," *Proceedings of the national academy of sciences*, vol. 79, no. 8, p. 2554âĂŞ2558, 1982.

[11] Y. Bengio, P. Simard, and P. Frasconi, "Learning long-term dependencies with gradient descent is difficult," *IEEE transactions on neural networks*, vol. 5, no. 2, p. 157âĂŞ166, 1994.

[12] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural computation*, vol. 9, no. 8, p. 1735âĂŞ1780, 1997.

[13] F. A. Gers, J. Schmidhuber, and F. Cummins, "Learning to forget: Continual prediction with lstm," 1999.

[14] "From gan to wgan." [Online]. Available: https://lilianweng.github.io/lil-log/2017/08/20/from-GAN-to-WGAN.html#generative-adversarial-network-gan

[15] C. Esteban, S. L. Hyland, and G. RÃďtsch, "Real-valued (medical) time series generation with recurrent conditional gans," *arXiv preprint arXiv:1706.02633*, 2017.

[16] T. Salimans, I. Goodfellow, W. Zaremba, V. Cheung, A. Radford, and X. Chen, "Improved techniques for training gans," in *Advances in Neural Information Processing Systems*, 2016, p. 2234âĂŞ2242.

[17] M. Arjovsky and L. Bottou, "Towards principled methods for training generative adversarial networks," *arXiv preprint arXiv:1701.04862*, 2017.

[18] M. Mirza and S. Osindero, "Conditional generative adversarial nets," *arXiv preprint arXiv:1411.1784*, 2014.

[19] T. Karras, T. Aila, S. Laine, and J. Lehtinen, "Progressive growing of gans for improved quality, stability, and variation," *arXiv preprint arXiv:1710.10196*, 2017.

[20] H. Malmsten and T. TerÃďsvirta, "Stylized facts of financial time series and three popular models of volatility," *SSE/EFI Working Paper Series in Economics and Finance*, vol. 563, 2004.

[21] "Seasonal arima with python." [Online]. Available: http://www.seanabu.com/2016/03/22/time-series-seasonal-ARIMA-model-in-python/

[22] "Tensorflow: Api documentation." [Online]. Available: https://www.tensorflow.org/api_docs/

[23] "Keras documentation." [Online]. Available: https://keras.io/

[24] "Mnist handwritten digit database, yann lecun, corinna cortes and chris burges." [Online]. Available: http://yann.lecun.com/exdb/mnist/

[25] "Stationarity and differencing of time series data." [Online]. Available: http://people.duke.edu/~rnau/411diff.htm

[26] "Time series analysis (tsa) in python - linear models to garch." [Online]. Available: http://www.blackarbs.com/blog/time-series-analysis-in-python-linear-models-to-garch/11/1/2016

[27] M. Mirza and S. Osindero, "Conditional generative adversarial nets," *arXiv:1411.1784 [cs, stat]*, Nov 2014, arXiv: 1411.1784. [Online]. Available: http://arxiv.org/abs/1411.1784

Appendix A

Final Financial GAN Source Code

```
import pandas as pd
import statsmodels.tsa.api as smt
import statsmodels.api as sm
import scipy.stats as scs
from sklearn.preprocessing import MinMaxScaler
from sklearn.externals import joblib
from numpy import cumsum, log, polyfit, sqrt, std, subtract
import matplotlib
matplotlib.use('TkAgg')
import tensorflow as tf
from tensorflow.contrib.rnn import LSTMCell, DropoutWrapper, MultiRNNCell
import numpy as np
import matplotlib
matplotlib.use('TkAgg')
import matplotlib.pyplot as plt
import time

def def_placeholders(batch_size, seq_length, latent_dim, num_features, cond=False):
    # defines placeholder variables
    X = tf.placeholder(tf.float32, [batch_size, seq_length, num_features])
    Z = tf.placeholder(tf.float32, [None, None, latent_dim])

    # placeholder variables for when using conditional version
    if cond is True:
        CG = tf.placeholder(tf.float32, [batch_size, 1])
        CD = tf.placeholder(tf.float32, [batch_size, 1])
        return Z, X, CG, CD
    return Z, X


def def_loss(Z, X, seq_length, batch_size, cond_option=False, CG=None, CD=None):
    # define separate networks
    if cond_option is False:
        G_sample = generator(Z, seq_length, batch_size)
        D_real, D_logit_real = discriminator(X, seq_length, batch_size)
        D_fake, D_logit_fake = discriminator(G_sample, seq_length, batch_size, reuse=True)
    else:
        G_sample = generator(Z, seq_length, batch_size, CG, cond_option)
        D_real, D_logit_real = discriminator(X, seq_length, batch_size, CD, cond_option)
        D_fake, D_logit_fake = discriminator(G_sample, seq_length, batch_size, CG, cond_option,↩
        reuse=True)

    # define losses
    D_loss_real = tf.reduce_mean(tf.nn.sigmoid_cross_entropy_with_logits(logits=D_logit_real, ↩
      labels=tf.ones_like(D_logit_real)), 1)
    D_loss_fake = tf.reduce_mean(tf.nn.sigmoid_cross_entropy_with_logits(logits=D_logit_fake, ↩
      labels=tf.zeros_like(D_logit_fake)), 1)
    D_loss = D_loss_real + D_loss_fake

    G_loss = tf.reduce_mean(tf.nn.sigmoid_cross_entropy_with_logits(logits=D_logit_fake, labels↩
      =tf.ones_like(D_logit_fake)), 1)

    return D_loss, G_loss


def generator(Z, seq_length, batch_size, CG=None, cond=False,  num_generated_features=1 ,↩
      hidden_units_g=100, reuse=False, learn_scale=True):
    with tf.variable_scope("generator") as scope:
```

```
54          if reuse:
55              scope.reuse_variables()
56          W_out_G_initializer = tf.truncated_normal_initializer()
57          b_out_G_initializer = tf.truncated_normal_initializer()
58          lstm_initializer = None
59
60          W_out_G = tf.get_variable(name='W_out_G', shape=[hidden_units_g, ↩
            num_generated_features],
61                                    initializer=W_out_G_initializer)
62          b_out_G = tf.get_variable(name='b_out_G', shape=num_generated_features, initializer=↩
            b_out_G_initializer)
63          if cond is True:
64              condition = tf.stack([CG] * seq_length, axis=1)
65              inputs = tf.concat([Z, condition], axis=2)
66          else:
67              inputs = Z
68          cell = LSTMCell(num_units=hidden_units_g, state_is_tuple=True, initializer=↩
            lstm_initializer, reuse=reuse, name='lstm_g')
69          rnn_outputs, rnn_states = tf.nn.dynamic_rnn(cell=cell, dtype=tf.float32,
70                                                      inputs=inputs)
71
72          rnn_outputs_2d = tf.reshape(rnn_outputs, [1, hidden_units_g], name='reshape_output')
73          logits_2d = tf.matmul(rnn_outputs_2d, W_out_G) + b_out_G
74          output_2d = tf.nn.tanh(logits_2d, name='tanh_g')
75          output_3d = tf.reshape(output_2d, [1, seq_length, num_generated_features], name='↩
            output_g')
76      return output_3d
77
78  def create_cell(size, reuse, lstm_init):
79      lstm_cell = LSTMCell(size,forget_bias=0.7, state_is_tuple=True, initializer=lstm_init, reuse↩
            =reuse)
80      lstm_cell = DropoutWrapper(lstm_cell)
81      return lstm_cell
82
83
84  def discriminator(X, seq_length, batch_size, CD=None, cond=False, hidden_units_d=100, reuse↩
            =False):
85      with tf.variable_scope("discriminator") as scope:
86          if reuse:
87              scope.reuse_variables()
88
89          W_out_D = tf.get_variable(name='W_out_D', shape=[hidden_units_d, 1],
90                                    initializer=tf.truncated_normal_initializer())
91          b_out_D = tf.get_variable(name='b_out_D', shape=1,
92                                    initializer=tf.truncated_normal_initializer())
93          if cond is True:
94              condition = tf.stack([CD] * seq_length, axis=1)
95              inputs = tf.concat([X, condition], axis=2)
96          else:
97              inputs = X
98
99          cell = tf.contrib.rnn.LSTMCell(num_units=hidden_units_d, state_is_tuple=True, reuse=↩
            reuse)
100
101         rnn_outputs, rnn_states = tf.nn.dynamic_rnn(cell=cell, dtype=tf.float32, inputs=inputs)
102         logits = tf.einsum('ijk,km', rnn_outputs, W_out_D) + b_out_D
103
104         output = tf.nn.sigmoid(logits)
105
106     return output, logits
107
108
109 def def_opt(D_loss, G_loss, learning_rate):
110     # gather trainable variables and define optimisers
111     disc_vars = [v for v in tf.trainable_variables() if v.name.startswith('discriminator')]
112     gen_vars = [v for v in tf.trainable_variables() if v.name.startswith('generator')]
113
114     D_loss_mean_over_batch = tf.reduce_mean(D_loss)
115     D_solver = tf.train.GradientDescentOptimizer(learning_rate=learning_rate).minimize(↩
            D_loss_mean_over_batch, var_list=disc_vars)
116     G_loss_mean_over_batch = tf.reduce_mean(G_loss)
117     G_solver = tf.train.AdamOptimizer().minimize(G_loss_mean_over_batch, var_list=gen_vars)
118     return D_solver, G_solver
119
120
121 def sample_Z(batch_size, seq_length, latent_dim):
122     sample = np.float32(np.random.normal(size=[batch_size, seq_length, latent_dim]))
123     return sample
124
125 def get_next_batch(batch_num, batch_size, samples, cond=False, cond_array=None):
126     sample_num_start = batch_num * batch_size
127     sample_num_end = sample_num_start + batch_size
128     if cond is True:
```

```
129         return samples[sample_num_start:sample_num_end], cond_array[sample_num_start:↩
            sample_num_end]
130     else:
131         return samples[sample_num_start:sample_num_end]
132
133
134 def train_epochs(sess, samples, batch_size, seq_length, latent_dim, D_solver, G_solver,
135                  X, Z, D_loss, G_loss, G_sample, epoch, scalar, CG=None, CD=None, cond_array↩
            =None, cond_option=False):
136     D_rounds = 3
137     G_rounds = 1
138
139     # choose arbitrary length of training per epoch
140     for i in range(0, 500):
141         # update discriminator
142         for d in range(D_rounds):
143             if cond_option is True:
144                 X_batch, Y_batch = get_next_batch(i, batch_size, samples, cond_option, ↩
            cond_array)
145             else:
146                 X_batch = get_next_batch(i, batch_size, samples)
147             Z_batch = sample_Z(batch_size, seq_length, latent_dim)
148             if cond_option is True:
149                 Y_batch = Y_batch.reshape(1, 1)
150                 _ = sess.run(D_solver, feed_dict={X: X_batch, Z: Z_batch, CD: Y_batch, CG: ↩
            Y_batch})
151             else:
152                 _ = sess.run(D_solver, feed_dict={X: X_batch, Z: Z_batch})
153         # update generator
154         for g in range(G_rounds):
155             if cond_option is True:
156                 X_batch, Y_batch = get_next_batch(i, batch_size, samples, cond_option, ↩
            cond_array)
157                 Y_batch = Y_batch.reshape(1, 1)
158                 _ = sess.run(G_solver, feed_dict={Z: sample_Z(batch_size, seq_length, latent_dim),↩
             CG: Y_batch})
159             else:
160                 _ = sess.run(G_solver,feed_dict={Z: sample_Z(batch_size, seq_length, latent_dim)↩
            })
161         # get loss
162         if cond_option is True:
163             D_loss_curr, G_loss_curr = sess.run([D_loss, G_loss], feed_dict={X: X_batch,
164                                                                              Z: sample_Z(batch_size, ↩
            seq_length, latent_dim),
165                                                                              CG: Y_batch, CD: Y_batch})
166             D_loss_curr = np.mean(D_loss_curr)
167             G_loss_curr = np.mean(G_loss_curr)
168         else:
169             D_loss_curr, G_loss_curr = sess.run([D_loss, G_loss], feed_dict={X: X_batch,
170                                                                              Z: sample_Z(batch_size, ↩
            seq_length, latent_dim)})
171             D_loss_curr = np.mean(D_loss_curr)
172             G_loss_curr = np.mean(G_loss_curr)
173
174         if i % 50 == 0 and i != 0:
175             # output loss information every 50 iterations
176             print(
177                 "Iteration: %d\t Discriminator loss: %.4f\t Generator loss: %.4f." % (i, ↩
            D_loss_curr, G_loss_curr))
178         if i % 100 == 0:
179             # generate and save data every 100 iterations
180             if cond_option == True:
181                 answer = sess.run(G_sample, feed_dict={Z: sample_Z(batch_size, seq_length, ↩
            latent_dim),
182                                                        CG: (np.random.choice([0.5], size=(batch_size, 1)))})
183             else:
184                 answer = sess.run(G_sample, feed_dict={Z: sample_Z(batch_size, 5000
185                                                                    , latent_dim)})
186             np.savetxt("test.csv", answer, delimiter=',')
187             answer = scalar.inverse_transform(np.reshape(answer,newshape=(answer.shape[0],↩
            answer.shape[1])))
188             np.savetxt("test.csv", answer, delimiter=',')
189     return D_loss_curr, G_loss_curr
190
191
192 def run(data, scalar, cond_option=False):
193     batch_size = 10
194     seq_length = 100
195     latent_dim = 1
196     num_features = 1
197     learning_rate = 0.1
198     num_epochs = 50
199
```

```
200     # conditionality option, false by default, can be changed to true with condition being ↩
          specified
201     # (and corresponding placeholder variables also being changed)
202     if cond_option is True:
203         Z, X, CG, CD = def_placeholders(batch_size, seq_length, latent_dim, num_features, ↩
          cond_option)
204         D_loss, G_loss = def_loss(Z, X, seq_length, batch_size, cond_option, CG, CD)
205         G_sample = generator(Z, seq_length, batch_size, CG, cond_option, reuse=True)
206     else:
207         Z, X = def_placeholders(batch_size, seq_length, latent_dim, num_features)
208         D_loss, G_loss = def_loss(Z, X, seq_length, batch_size)
209         G_sample = generator(Z, seq_length, batch_size, reuse=True)
210
211     D_solver, G_solver = def_opt(D_loss, G_loss, learning_rate)
212
213     # initialisers
214     saver = tf.train.Saver()
215     sess = tf.Session()
216     sess.run(tf.global_variables_initializer())
217     D_loss_arr = []
218     G_loss_arr = []
219
220     # main training loop
221     for epoch in range(num_epochs):
222         # train for one epoch
223         if cond_option is True:
224             time_before = time.time()
225             D_loss_curr, G_loss_curr = train_epochs(sess, data, batch_size, seq_length, latent_dim,
226                                                     D_solver, G_solver, X, Z, D_loss, G_loss, G_sample,
227                                                     epoch, scalar, CG, CD, cond_array, cond_option)
228         else:
229             time_before = time.time()
230             D_loss_curr, G_loss_curr = train_epochs(sess, data, batch_size, seq_length, latent_dim,
231                                                     D_solver, G_solver, X, Z, D_loss, G_loss, G_sample,
232                                                     epoch, scalar)
233         # output information about epoch just ran
234         print("Epoch Time: ", time.time()time_before)
235         D_loss_arr.append(D_loss_curr)
236         G_loss_arr.append(G_loss_curr)
237         print("Epoch: %d\t Discriminator loss: %.4f\t Generator loss: %.4f." % (epoch, ↩
          D_loss_curr, G_loss_curr))
238
239         # shuffle data
240         perm = np.random.permutation(data.shape[0])
241         data = data[perm]
242
243         if cond_option == True:
244             cond_array = cond_array[perm]
245         if epoch != 0:
246             # plot and save losses for every epoch
247             fig = plt.figure()
248             plt.title('G and D Losses')
249             plt.plot(D_loss_arr, color='r', label='D')
250             plt.plot(G_loss_arr, color='b', label='G')
251             plt.legend()
252             name = "./Plots/epoch_" + str(epoch) + "_losses_c.png"
253             fig.savefig(name, dpi=fig.dpi)
254         if epoch != 0 and epoch % 5 ==0:
255             # every 5 epochs save model to file
256             save_path = saver.save(sess, "./Models/model_c.ckpt")
257             print("Model saved in path: %s" % save_path)
258             joblib.dump(scalar, "./Models/model_c_scalar.save")
259
260
261
262 def main():
263     # import from csv
264     data = pd.read_csv("./C.txt")["Close"].values
265
266     # difference to remove stationarity
267     data = np.diff(data)
268     # to reobtain undifferenced data  np.r_[data[0], np.diff(data)].cumsum()
269
270     # make sequences with 100 steps
271     samples = []
272     n_samples = int(100)
273     for i in range(len(data)n_samples):
274         signals = []
275         for j in range(n_samples):
276             signals.append(data[i+j])
277         samples.append(np.array(signals).T)
278     samples = np.array(samples)
279
280     # rescale for lstm
```

```
281     scalar = MinMaxScaler(feature_range=(1, 1))
282     scalar.fit(samples)
283     samples = scalar.transform(samples)
284
285     # randomly shuffle and set data to permutation array
286     perm = np.random.permutation(samples.shape[0])
287     samples = samples[perm]
288
289     # reshape to 3d due to lstm
290     samples = np.reshape(samples, newshape=[len(samples), n_samples, 1])
291
292     run(samples,scalar)
293
294 if __name__ == '__main__':
295     main()
```

Appendix B

Experimental results

The following results are from the time series GAN seen in Section 4.1.4.

B.1 Autoregressive Time Series Result

Figure B.1: Alpha Coefficient Value vs Run Number

B.2 Moving Average Time Series Result

Figure B.2: Beta Coefficient Value vs Run Number

B.3 ARMA Time Series Results

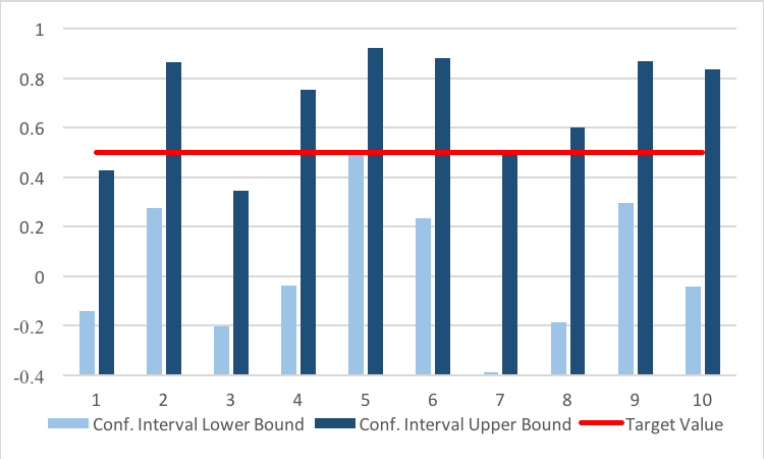


Figure B.3: Alpha 1 Coefficient Value vs Run Number

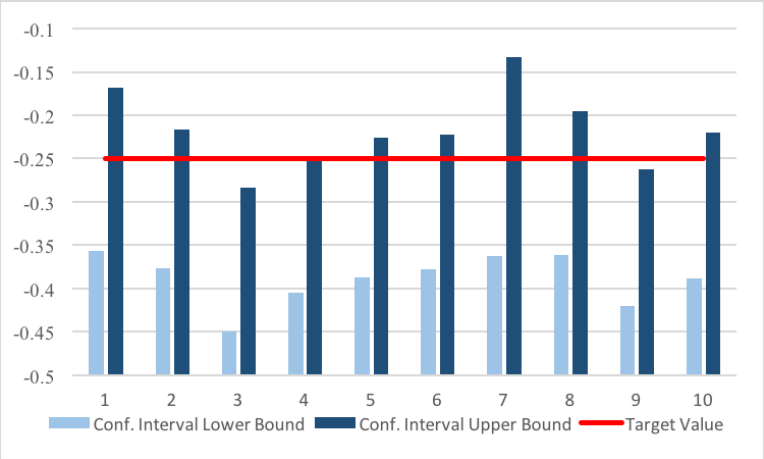


Figure B.4: Alpha 2 Coefficient Value vs Run Number

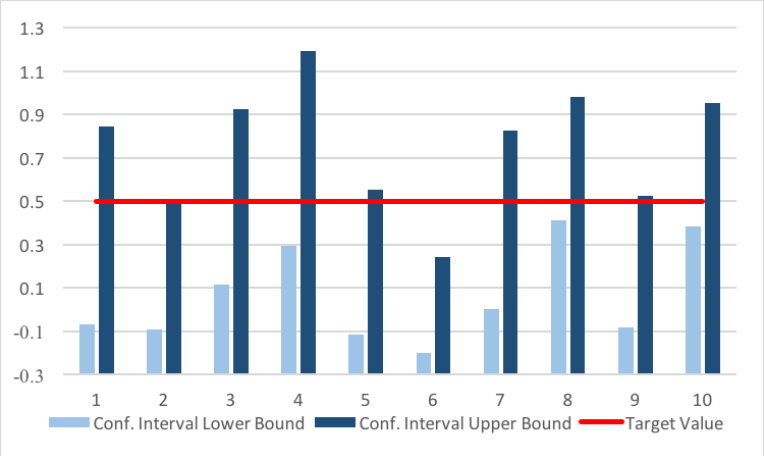


Figure B.5: Beta 1 Coefficient Value vs Run Number

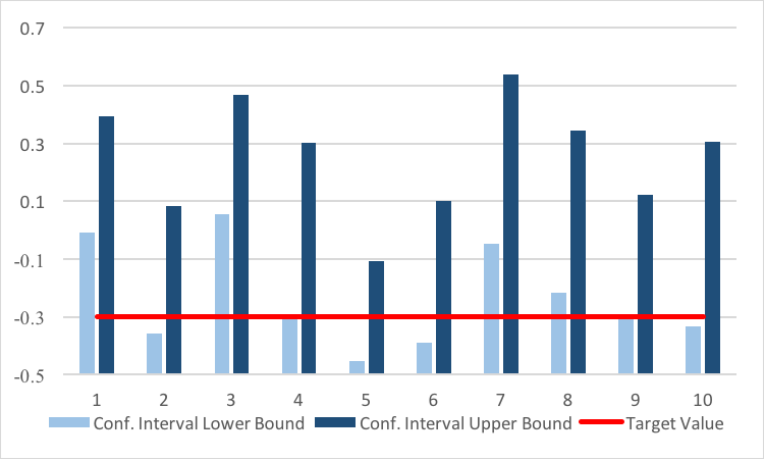


Figure B.6: Beta 2 Coefficient Value vs Run Number