UNIVERSITY of STIRLING

Division of Computing Science and Mathematics

Faculty of Natural Sciences

University of Stirling

Time Series Prediction:

Investigation of Long Short-Term Memory (LSTM) Neural Network

Raymond Rono Cheruiyot

Dissertation submitted in partial fulfilment for the degree of

Master of Science in Big Data

September 2018

Abstract

Techniques like Autoregressive Integrated Moving Average (ARIMA) have been useful for time series research. Advances in computing and machine learning have been gaining significant attention for time series prediction due to recently developed techniques. The capability of neural networks to model nonlinear and linear forecasting has been established in the literature both theoretically and empirically. Several research studies have yielded mixed results and findings. Therefore, we aim to provide further evidence on the effectiveness of Long Short-Term Memory (LSTM) and Gated Recurrent Unit (GRU). We will give an overview of the recently proposed Minimal GRU (MGRU) time series which is also a variant of Recurrent Neural Networks (RNN).

We investigate recurrent neural networks machine learning techniques focusing on the LSTM and GRU. Configuring neural networks is difficult because there is no good theory on how to do it. Attention is drawn to a systematic exploration of different configurations both from a dynamical and an objective result point of a view. The aim is to try to and understand what is going on for a given predictive modelling problem. The primary measure of performance is how well each model forecast out-of-sample data. Analysis of the variants is done on three representative datasets by optimizing the hyperparameters for each task using a combination of sequential, grid and or random search. The achieved optimal configurations established that there wasn't a generally superior network out of the two. This is because vanilla LSTM performed comparably well compared to GRU on stock price prediction. However, two datasets on Consumer Price Index and Energy consumption had GRU performing better than LSTM. We give a recommendation that different tasks should consider different configuration of the hyperparameters using the approach that we adopted.

Attestation

I understand the nature of plagiarism, and I am aware of the University's policy on this.

I certify that this dissertation reports original work by me during my University project except for the following:

- The technology review in Section 2.8-2.12 was largely taken from [26], [32].
- The LSTM and GRU libraries discussed in chapter 3 and 4 was created by Keras (https://keras.io/layers/recurrent/) and was used in accordance with the licence supplied.
- The experiment codes discussed in Chapter 3 and 4 were developed by me during the project period.

Signature:  **Date:** September 3rd, 2018

Acknowledgements

First and foremost, I would like to thank God Almighty for giving me the strength, knowledge, ability and opportunity to undertake this research study. Without Him this achievement would not have been possible.

In my journey towards this degree, Prof. Leslie Smith, former Head of Department for Computing Science and Mathematics, has given me invaluable and heartfelt guidance, inspiration and immense knowledge. He consistently allowed this paper to be my own work, but steered me in the right direction whenever he felt I needed it. In him I have found an advisor, an inspiration, a role model, a friend, and a pillar of support. I could not have imagined having a better supervisor and mentor for my project and I shall eternally be grateful to him for his immense assistance.

I would also like to thank Dr. Swingler K., Prof. Hussain A., Dr. Cairns D., Dr. Maharaj S. and Dr. Jones S. for their encouragement, insightful classes, and hard questions. I can't mention all the academic, technical and support staff in the department of computing and mathematics who were very helpful to me throughout this journey.

This work would not have been possible had I not been awarded a scholarship by the Commonwealth Scholarship Commission (CSC) funded by the UK government. My gratitude also goes to my colleagues and the management at Kenya Revenue Authority for accommodating my studies and enduring my absence. I am grateful to my group of friends (Stirling Dataists) and fellow research scholars at the University of Stirling for their moral support and valuable exchange of ideas.

This acknowledgement would be incomplete without thanking my family. The strong foundation for my lifetime journey placed by my late parents, Mr. & Mrs. Langat; the blessings of my parents-in-law, Mr. & Mrs. Rotich; supportive guidance of my loving wife, Ann, who stood by me through all my travails, my absences, my fits of pique and impatience; the unending inspiration of our wonderful children, Moses and Ethan; the unwavering love of my siblings, cousins, in-laws and their families. This family support and prayers has been the biggest source of my strength

This work is dedicated to my late mother Mrs. Rose Langat (1946-2000) whose hard work, perseverance and dreams for me have resulted in this achievement.

Table of Contents

List of Figures

List of tables

List of abbreviations:

ARIMA: Autoregressive Integrated Moving Average

GRU: Gated Recurrent Unit

LSTM: Long Short-Term Memory

MGRU: Minimal Gated Recurrent Unit

MGU: Minimal Gated Unit (refer to MGRU)

NN: Neural Network

RNN: Recurrent Neural Network

MAE: Mean Absolute Error

RMSE: Root Mean Squared Error

BTU: British Thermal Unit

CPI: Consumer Price Index

1 Introduction

Historically time series analysis has been important in many areas: economic, sales, financial, budgetary, stock market and weather forecasting; process, quality control, workload projections, utility and inventory studies; census analysis etc. Classical models like Auto Regressive Moving Average (ARMA) or the conditional volatility ones like Generalized Autoregressive Conditional Heteroskedasticity (GARCH) and its many variants [2] have been used to model time series. Gheyas, I. A., & Smith, L. S. (2009) proposed a simpler and more efficient algorithm (GRNN ensemble) for forecasting univariate time series. The proposed algorithm was an ensemble learning technique that combined the advice from several Generalized Regression Neural Networks [3]. Time is a key variable: it is the explicit dependent variable in time series analysis. The model predicts y(t) from an input of x(t') for a sequence of values of time t' < time t, but a different prediction may be associated later with an identical input. The solution to this problem may be to have a memory of past inputs or to make the model use more data input from the past. The latter approach may make the size of the input larger than what the model would have handled in the case of long-term dependencies. And the length of the time-dependencies may be entirely unknown. The memory cell has remained the principal component of an LSTM architecture. This is because it can maintain its state over time. Most recent studies incorporate many improvements that have been made to the LSTM architecture. This study therefore aims to provide further evidence on the effectiveness of Long Short-Term Memory (LSTM) and Gated Recurrent Unit (GRU) and addresses the open question of improving the LSTM and GRU architectures using data domains which have hitherto not been applied scientifically to LSTM and GRU as per my knowledge.

1.1 Background and Context

Rolling regression estimation based on capturing model parameters over a period of historical data and being used to capture the evolving nature of the time series have had limited success because of difficulties in capturing fast-changing dynamic data of high statistical significance [16]. In several ways with complicated behaviors, there have been limitations with traditional statistical approaches in being used to construct a model for predicting a target precisely. However, in the recent years, big data (i.e. large volumes of

data) has increasingly been used in time series studies. Researchers have been studying the applications of big data in various fields including healthcare, education, manufacturing, governance, insurance, transportation etc. With rapidly increasing computational power and wide availability of data, a relatively new technology called deep learning is gaining popularity for data forecasting. Deep learning uses neural networks to learn things from data by mimicking aspects of the working of the human brain. Artificial neural networks (ANN) are neural networks systems which "learn" to perform tasks generally without being programmed with any task-specific rules by considering examples. ANN have seen a resurgence in recent years and spectacular successes have been demonstrated in sequential data processing. [4]

Derived from Artificial Neural Networks (ANN) is a class called Recurrent Neural Networks (RNN). These networks form a sequential directed graph using connections in between nodes enabling it to exhibit a dynamic temporal behaviour characteristic of a time sequence. RNNs normally process sequences of inputs using their memory. [5]
RNNs are essential in modelling data with sequential structures like time series. They incorporate previous time-step output as one of the current time-step input features. Vanilla RNNs turn out to be quite effective for short-term dependencies. However, they fail to take into consideration the context behind an input. Inputs from some time before cannot be recalled when making predictions in the present. This is because of the problem of Vanishing Gradient [1]. In the current study, we look at one type or improvement of RNN, called Long Short-Term Memory (LSTM) neural networks and its variants (Gated Recurrent Unit (GRU). LSTMs can selectively remember or forget things [6]. This paper will use Long Short-Term Memory (LSTM) neural network, a type of recurrent neural network (RNN), to predict future time series data. LSTM network is being applied in this study mainly because of the following:

(1) LSTM is a flexible universal function approximator appropriate for time series predictions like a deep neural network [17];

(2) Unlike vanilla RNN, LSTM rarely suffers from vanishing gradient problem [1];

(3) LSTM discovery of long-range characteristics is unmatched [6];

This project will explore how to configure an LSTM network on a time series forecasting problem. We work on LSTM, Gated Recurrent Network (GRU) and the latest Minimal Gated Recurrent Network (MGRU). Our primary measure of performance is how well

each model forecast out-of-sample data. Analysis of the variants will be done on four representative datasets by optimizing the hyperparameters for each task using random search.

1.2 Scope and Objectives

The past decade has seen an upsurge in interest of using recurrent neural networks (RNN) specifically for forecasting thereby enhancing the quantity of research activities. However the quality of the results has not been well established purely due to lack of standard approaches of handling them. Advanced research using RNN has been carried out in the areas of speech and human action recognition, language modelling, image captioning, rhythm learning, question answering and video to text. The traditional times series has also benefited from these studies. There are practically useful reasons for studying time series. Time series is considered by Yang, Q. and Wu, X. (2006) as one of the top 10 challenging problems in data mining due to its unique properties. As Langkvist et al. (2014) notes, "Time is a natural element that is always present when the human brain is learning tasks like language, vision and motion. Most real-world data have a temporal component, whether it is measurements of natural processes (weather, sound waves) or man-made (stock market, robotics)" [9]. Literature provides mixed results and thorough examination of the elect of key modelling factors with regards to model performance and efficiency for time series recurrent neural networks is limited. Several variants have been added too to the LSTM network with the latest being MGRU [8]. Findings have reported inconsistencies which may point to the lack of systematic approaches to model building for RNN.

In this paper, we present an investigation of the application of Long Short-Term Memory neural network time-series analysis and prediction. The aim is to study and compare the effectiveness of time series models to make forecasts on real data. Specifically, we are interested in evaluating the difference between Long Short-Term Memory (LSTM) [6] and Gated Recurrent Unit (GRU) and a more recent method that has been studied in the time series literature, Minimal Gated Recurrent Unit (MGRU) Network [8] but which is also based on LSTM. A simple question we would attempt to ask is does GRU outperform vanilla LSTM for tasks like stock, energy consumption and consumer price index predictions? These are domains where these techniques have not had the benefit of getting applied.

There was a detailed examination of the elects of certain important neural network modelling factors on nonlinear time-series modelling and forecasting with specific emphasis on LSTM, GRU and MGRU. MGRU is included here for reference purposes only. The intention would have been to include MGRU in comparing the performance, however there is no publicly available implementation of the network libraries and an attempt to develop them from scratch would have stretched the scoped time of the project. MGRU(MGU) network was therefore not evaluated in this study.

1.3 Achievements

Developing an approach in an area with limited theory of doing it has not been easy. Complexity was compounded when it came to developing appropriate algorithms and techniques to execute the intended strategies. Neural network is relatively a fast-changing field with numerous recent approaches in a broad but not deep manner. At the start of the project my grasp of LSTM was far below average. Going through the mountains of literature out there and picking appropriate approaches only increased the gradient of learning making it steep. And these had a 3-month period to get accomplished. As if learning all the theory was not that challenging, practical implementation of the techniques available proved even more 'motivating'. Because Python was relatively new to me leave alone frameworks like tensor flow on Keras. Add that to combining several frameworks and libraries to make them to work together. The rate at which some of these libraries are being churned out while making others obsolete ensured I was on top of my toes in trying several options. Tensorflow are yet to release any plug and play library for Python 3.7. Compiling existing libraries while manually trying to plug it to the latest Python version added to the ever-moving clock time in my timeline. Systematically downgrading Python and attempting different versions of the required software saw me settling on versions described in section 3.3.1. This added to the knowledge and skills of tools that I currently possess. Accurate, high quality data was not negotiable for implementing a proper predictor for meeting the objectives. As such some data preparation work was important. A good understanding of Python and some specialized libraries was important for this work and therefore quick and appropriate learning was necessary which added to the already constrained timelines. Due to excessive usage of the computing resources, operating systems' freezes and crashes somehow became part of the study allowing me to learn

different operating systems and putting in place appropriate and effective back-up procedures. As a Windows person, working with MacOS was a totally new experience to me. The experiments and comparisons were successfully implemented. Based on our experiments, we concluded no single network outperforms the other. And that superiority or lack of it in one network for time series depends on the datasets and domain being studied as well as the level of optimization that can be achieved with parameters. We saw LSTM and GRU performing comparatively well with stock price forecasting. However, GRU appeared to be better than LSTM on CPI and Energy consumption datasets. Comparative analysis on MGU was not achieved due to time constraints and limited available hardware.

1.4 Overview of Dissertation

This report is organised into chapters for this study.

Section 2 – State of the art provides an overview to the study carried out to accomplish the outlined objectives. This will also touch briefly on the significance and the relevance of time series analysis and prediction to real challenges. There will be a summary subsection on the background and current research in time series prediction and more so LSTM to allow the reader to gain sufficient understanding. There will be a specific focus on techniques used in the implementation in this study. Lastly, a presentation of the coding techniques and tools used will be unveiled.

Section 3 – Experimental design will present the activities involved in collecting and preprocessing the training and test data. The design of neural networks (LSTM and variants) experiments will also be outlined. The various techniques used in the study will be discussed as well as the optimal values of the different hyperparameters that were tuned.

Section 4 – Results and analysis will present a summary of the different results attained by varying different hyperparameters and features. The optimal results achieved by the models will be discussed. In this chapter we will also discuss critical evaluation which will look at the challenges and limitations encountered as well as appropriate recommendations for possible improvements. We will conclude and discuss future work. This will summarise the achievements of the project and emphasize the best configurations for optimal results.

2 State-of-The-Art

2.1 Introduction

This chapter provides an overview to the available literature regarding time series and time series forecasting. It will shed light on the significance of the study and real nature of the problem. There have been advances in the front of machine learning techniques relevant to time series prediction which will be briefly discussed. To give the reader sufficient understanding a summary sub-section on the background and current research in time series prediction and more so LSTM and GRU will be discussed with a specific focus on techniques used for implementation in the study. Applications of time series predictions availed in the literature will be pointed out.

2.2 Time series

A time series is a sequential set of data observations measured over successive times. It can be mathematically represented as a set of vectors x(t), t = 0,1, 2... where t represents the time elapsed [42]. The measurements or sequence of observations, $s_t \in R$, are usually ordered in time form or chronological order. Time series can be univariate or multivariate depending on the number of variables it features. A discrete time series will normally have observations measured at discrete points of time whereas continuous will consist of observations measured at every instance of time which can only exist for made-up problems and not in reality [44]. You will find discrete time series having consecutive observations recorded at equally spaced time intervals but having the variable being observed as a continuous variable from the real number scale making discrete series to be what is normally dealt with [42].

There are many applications of time series and can be found in:

a) Meteorology where variables like temperature, wind pressure can for a time series of data: weather variables, like temperature, pressure, wind etc.

b) Finance and economy which contains parameters like exchange rates, financial indexes, Gross National Product, Inflation, Consumer Price Index etc which are useful in making sense to economic and financial data.

c) Manufacturing and industry: Energy consumption, electric load, sensors, voltage etc can generate a sequence of data observations which can be modelled as time series.

d) Medicine and biomedicine which is awash with heart-rate, patient temperature time series data

e) The cell cycle time series of gene expression which is normally applied in genomics.

2.3 Time series forecasting

Time series forecasting relies on past observations which have been collected and analysed to develop a suitable mathematical model capable of predicting future data points. if there is not much knowledge about the statistical pattern on the successive observations then this approach can be useful. A future with certainty cannot be predicted and so time series is non-deterministic in nature. Most of the time stochastic processes are used to describe the probability structure of a time series. It is assumed that the time series variables are independent and identically distributed (i.i.d) following the normal distribution. However, this is not true and they in fact follow more or less regular pattern in the long term [44].

2.4 Seasonality in time series (Trend, Cyclical, Seasonal)

Time series can contain a cyclic or seasonal component. This is a repeating cycle of seasonal variation which may obscure or provide a strong signal to the model during forecasting. The cyclic structure is considered seasonal only if it consistently repeats at the same frequency otherwise it is a cycle. Understanding this component of time series has significant influence on the performance of modelling using machine learning. Identifying and removing the seasonal component can result in output variables having a clearer relationship to the input. The process of seasonality removal is called *seasonal adjustment or deseasonalizing ('stationarized')* and will lead to *seasonally stationary* data. A *stationary time series* exhibit constant statistical properties over time such as mean, variance, autocorrelation, etc. A *non-stationary data* is one which has a clear seasonal component. New data may be harvested from additional information about the seasonal component which may be useful in improving model performance. There can be different types of seasonality including time of day, daily, weekly, monthly, yearly etc.

We find that data with strong seasonal structure are forecast comparatively well by most machine learning techniques. On the other hand, without strong seasonality, there is very little information that can be extracted and forecasting performance is poor. Some promising results were achieved from the experiments. This work enhances our confidence and excitement that much more can be explored to potentially further improve the prediction performance for time series data.

2.5 Significance of time series forecasting

While there are many significant reasons for time series forecasting some of the reasons include:

a) The ability to predict the future using past observations.
b) Being in control of the process producing the series and therefore influencing the future.
c) Gaining knowledge which comes with understanding the mechanism generating the series.
d) In-depth understanding of the salient features of time series.

2.6 Actual nature of the problem

The effectiveness of time series models to make forecasts on real data is what researchers have not been able to agree on. Specifically, whether Gated Recurrent Unit (GRU) performs better than Long Short-Term Memory (LSTM) or and a more recent method that has been studied in the time series literature, Minimal Gated Recurrent Unit. From literature LSTM and GRU have had little or no application on the domains of stock price prediction, consumer price index and energy consumption forecasting. A simple question we would attempt to ask is does GRU outperform vanilla LSTM for tasks like stock, energy consumption and consumer price index predictions? A detailed examination of the elects of certain important neural network modelling factors on nonlinear time-series modelling and forecasting with specific emphasis on LSTM and GRU on these domains will be carried out.

2.7 Neural Networks

A neural network is a machine learning model that was developed through the inspiration of a biological workings of the brain. It involves nodes (neurons) acting as

computational units. They work by receiving inputs incoming from their edges, while multiplying them with their corresponding edge weights. A non-linear function called activation *function* is then applied to the weighted result and this becomes an output. NN assumes the independence among the data samples. However, in sequential data, this does not hold true. Time series, speech, video etc are characterised by dependence between elements across time. A mechanism to consider the time and/or sequential dependency is crucial for sequential data. This gave birth to Recurrent Neural Network (RNN).

$$y(x) = f(\mathbf{w}\, x + b)$$

x = input vector, w **=** weight vector, b =neuron bias, f = element-wise multiplication, y = neuron output

Equation 2.4.1

Neuron vector equation

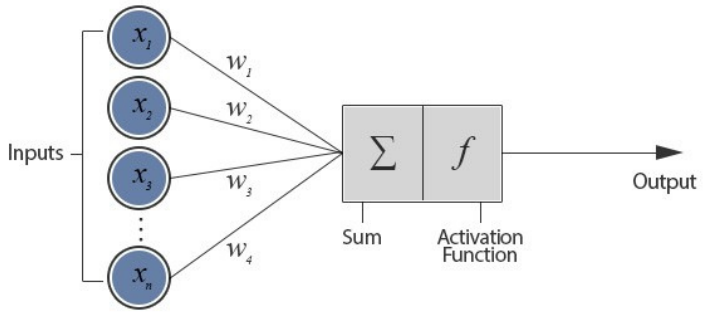


Fig 2-4-1

A neuron unit

Figure 2.7-1: depicts a neuron unit

Figure 2.7-1 depicts a neuron unit. A non-linear activation function works on the inputs and weights and produces and output. Image adapted from [22]. *logistic sigmoid (σ), tanh, and rectified linear units (ReLU)* represent the typical activation functions [23].

A feed forward network may be composed of a network of neurons within layers. Neurons are connected to each other using directed and weighted edges. These layers in most instances contain a minimum of *input* and an *output* layer for receiving an input and producing an output respectively. A complex feed-forward network may have other *hidden layers* and such networks are used for supervised learning tasks.

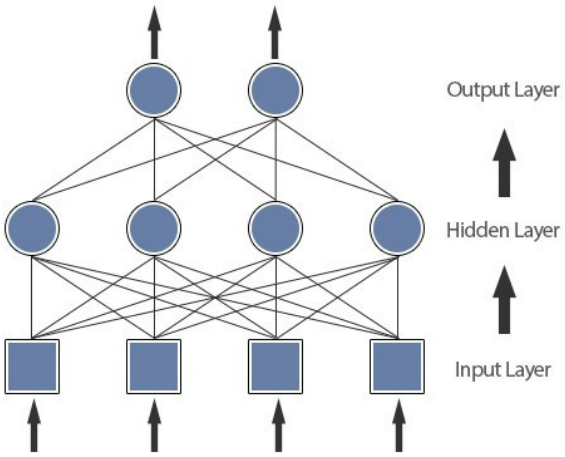


Figure 2.7-2: A feed-forward NN with one layer

Fig 2-4-2.
A feed-forward NN with one hidden layer

In **figure 2.7-2** each neuron in the layer, which is represented by circles, is inter-connected to all the neurons in the previous (bottom) layer. Technically the input layer will not count as a neuron as they just forward the input signal without processing. Image adapted from [22].

2.8 Recurrent Neural Network

An RNN is a special type of NN but with the special ability to process sequential data. Conventional feedforward neural network handles input of sequences containing variable-lengths. A recurrent neural network (RNN) is an extension of such network. RNN for time series does this by maintaining a recurrent hidden state which can be activated to depend on the input of the previous time. RNN can be represented as follows:
Given a sequence X = (X_1, X_2, X_3, . . ., X_t), then the recurrent hidden state, h_t, according to RNN is given by equation 2-6-2

$$h_t = \begin{cases} 0, & t = 0 \\ \emptyset(h_{t-1}, x_t), & otherwise \end{cases}$$

Equation 2-6-2.
The recurrent hidden state of RNN

where Ø is a nonlinear function examples are logistic sigmoid. Additionally, it is possible for the RNN to have an output Y = (Y_1, Y_2, Y_3, . . ., Y_t), with a variable length.

An implementation of the recurrent hidden state in Equation 1 is stated below.$h_t = g(W X_t + U h_{t-1})$

Equation 2-6-3.
The recurrent hidden state of RNN

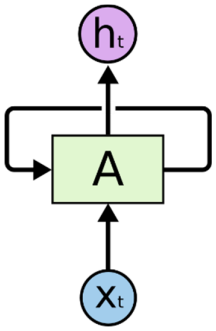


Fig 2.8-1.
A depiction of a looping RNN

Figure 2.8-1:A depiction of a looping RNN

In fig. 2.8-1 above, a chunk of recurrent neural network, A, outputs h_t by taking in input X_t with the network allowing the information to be 'looped' through the steps of the network.

In equation 2-6-3 above, g represents a hyperbolic tangent or a sigmoid function which achieves a smooth bounded function, W is the coefficient matrix for the input, X_t at the present step and U is the coefficient matrix for the activation of recurrent hidden units at the previous step, h_{t-1}.

Bengio et al. [1994] observed a limitation of traditional RNN, otherwise called vanilla RNN in capturing long term dependencies because of the 'curse of vanishing gradient'. In this type of RNN the gradient vanishes or explode with severe effects on the model. Because of these the gradient-based optimization method is severely weakened because of the long-term decencies becoming exponentially smaller in sequence length when compared to short-term dependencies.

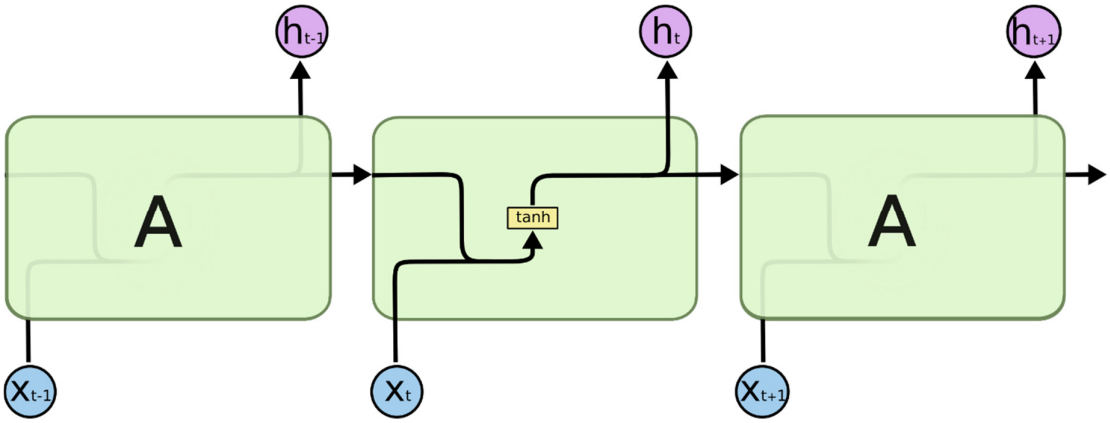


Figure 2.8-2:The repeating module in a standard RNN contains a single layer

Fig 2.8-2: The repeating module in a standard RNN contains a single layer. X_{t-1}, X_t, and X_{t+1} are sequence of input, A is Neural Network while h_{t-1}, h_t, and h_{t+1} are recurrent hidden states. Image adapted from [29].

Two dominant approaches have been proposed to address the vanishing gradient limitation in the RNN. On such approach was the development of a better learning algorithm than a simple stochastic gradient descent for example the Hessian-Free Optimization with structural damping approach [11]. The second approach in which we are more interested in this paper is the development of a better activation function consisting of affine transformation backed up by gating units of simple element-wise non-linearities. This was first discovered in long short-term memory (LSTM) unit proposed by Hochreiter and Schmidhuber [6] followed by another recurrent unit called gated recurrent unit (GRU) proposed by Cho et al. [2014]. More recently, another type of recurrent unit referred to as minimal gated recurrent unit (MGRU), was proposed by Zhou et al. [2016] [8].

Research has indicated that for tasks that require capturing long-term dependencies, RNNs employing either of the mentioned recurrent units have a better performance than the vanilla RNN. This is especially so in speech recognition [13], machine translation [14] and image classification [15].

2.9 Recurrent Neural Network - Long Short-Term Memory (RNN - LSTM)

One of the strengths of RNN is the idea that it can connect previous information to the present task. This works most of the time when we only recent information is required to perform the present task. In other situations where more context is required vanilla RNN may not give the best result as it may not get the ability to learn to connect the information.

To attempt to solve this problem, LSTM networks were proposed by Hochreiter & Schmidhuber (1997) and were refined and popularized by many researchers thereafter [12]. Studies have shown that they work better than vanilla RNN on a large variety of problems including those that require learning long-term dependencies.

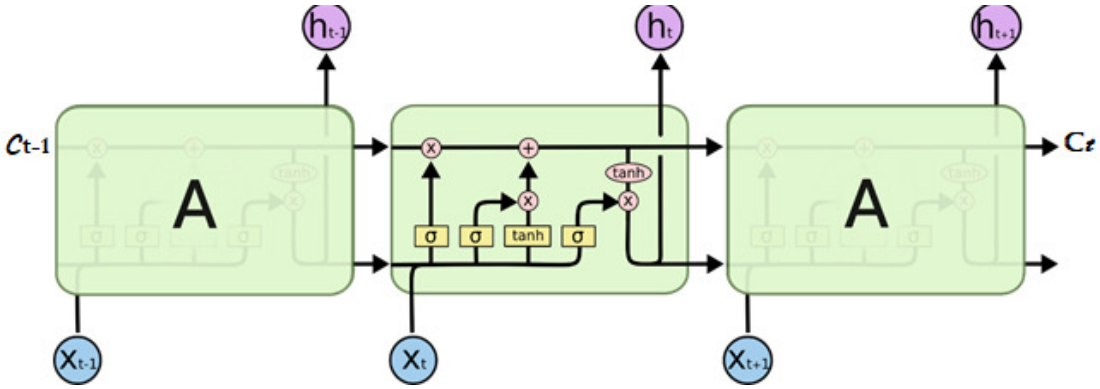
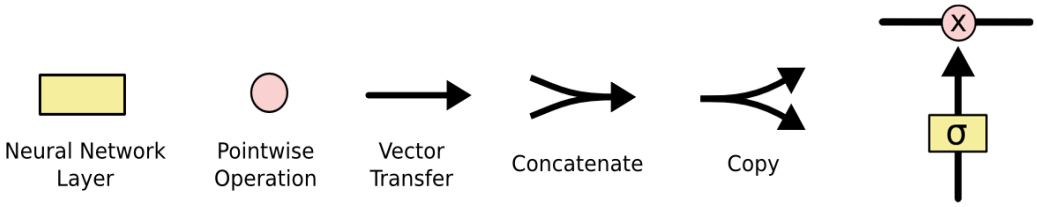


Figure 2.9-1:The repeating module in a in an LSTM contains four interacting layers

Fig 2.9-1a: The repeating module in a in an LSTM contains four interacting layers. X_{t-1}, X_t, and X_{t+1} are sequence of input, A is Neural Network while h_{t-1}, h_t, and h_{t+1} are recurrent hidden states. Image adapted from [29]. The symbols are further described in fig 2-3 below:



Gate

Fig 2.9-1b: Elements of an LSTM network.

Under LSTM in figure 2.9-1a, each line is a vector transfer of output from one node to become input to another node. The pointwise represent vector operations like addition, multiplication etc. The line C_{t-1} to C_t represents the cell state and is key to LSTM.

Components of an LSTM:

a) A neural network with sigmoid (Forget Gate, f)
b) A NN with Tanh (Candidate layer, C)
c) A NN with sigmoid (Input Gate, I)
d) A NN with sigmoid (Output Gate, O)
e) A vector of Hidden state, H
f) A vector of Memory state, C

The LSTM operates through its ability to add or remove information to the cell state through structures appropriately named gates. Typically, gates are used to control flow of information. and they are made of sigmoid neural net layer and pointwise multiplication operation. Sigmoid numbers are between zero and one with zero not allowing anything through and one allowing everything through the gate.

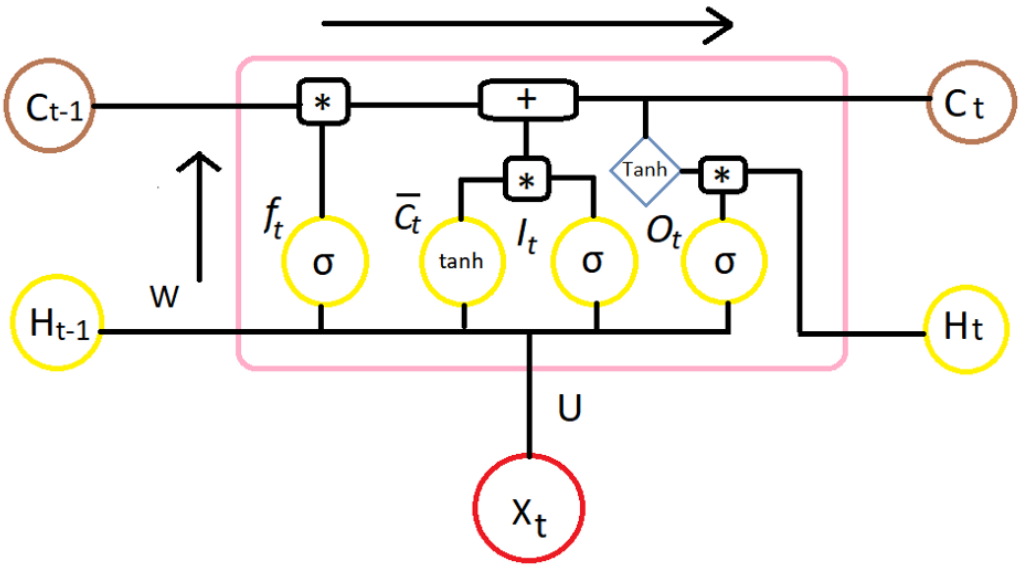


Figure 2.9-2: LSTM at time step t. Image adapted from [30].

The LSTM operates through its ability to add or remove information to the cell state through structures appropriately named gates. Typically, gates are used to control flow of information. and they are made of sigmoid neural net layer and pointwise multiplication operation. Sigmoid numbers are between zero and one with zero not allowing anything through and one allowing everything through the gate[31].

Three groups of variables are found for each cell:

a) X_t which refers to external observations at each time step, t. This is represented as a vector for multiple input signals like stock price, trade volume etc.

b) h_t represents the short-term memory of LSTM internal state which is a direct output from the LSTM cell.

c) C_t is a hidden LSTM state represented by a vector which can be accessed and/or modified through controlled gates: input, an output and a forget gate.

The gates are instrumental in regulating the flow of information in the network. In fact, they can capture important information and store them for long and short time as is required deriving its name from there 'long short-term memory'.

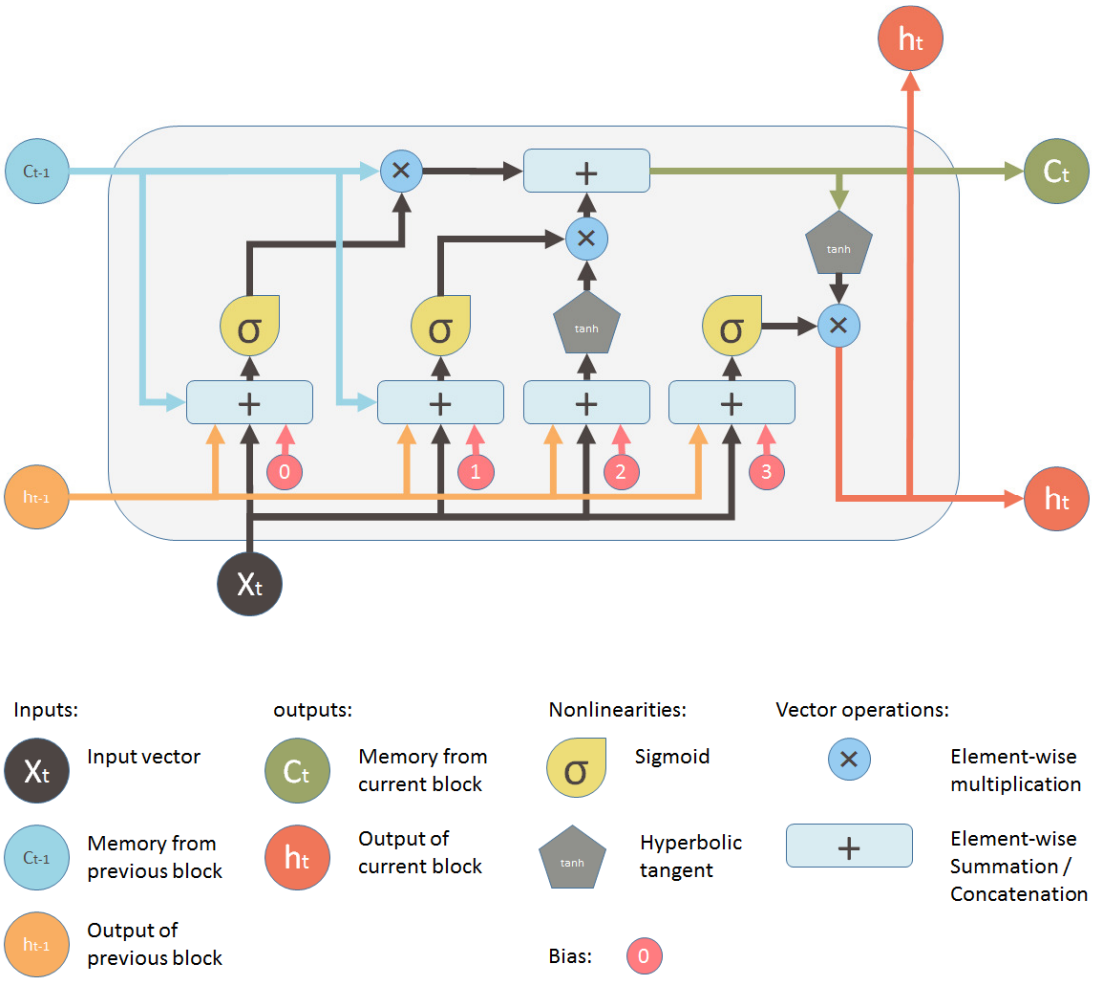


Figure 2.9-3: A LSTM building block Image adapted from [28].

Figure 2.9-3 displays an LSTM building block – a schematic diagram of the LSTM unit with forget gates. Image adapted from [28]. This single unit makes decision by considering the current input, X_t, previous output, h_{t-1} and previous memory, C_{t-1} and it generates a new output, h_t and alters its memory, C_t.

Equations for LSTM unit with a forget gate forward pass.

The equations 2-2-4 have variables which italicized in lowercase italics to represents a vector. Matrices W_q collect the weights of the input and U_q collect the weights of the recurrent connections, where $_q$ can either be the input i, gate o, output gate, the forget gate f or the memory cell c, depending on the activation being calculated [31]. The initial values are $C_0 = 0$ $h_0 = 0$ the operator $\circ$ denotes entry-wise product. The subscripts t refers to the time step [12].

$$f_t = \sigma_g(W_f x_t + U_f h_{t-1} + b_f)$$
$$i_t = \sigma_g(W_i x_t + U_i h_{t-1} + b_i)$$
$$o_t\, f_t = \sigma_g(W_o x_t + U_o h_{t-1} + b_o)$$
$$c_t = f_t \circ c_{t-1} + i_t \circ \sigma_c(W_i x_t + U_i h_{t-1} + b_i)$$
$$h_t = o_t \circ \sigma_h(c_t)$$

Equations 2-2-4.

Variables

x_t : *input vector to the LSTM unit*
f_t : *forget gate's activation vector*
i_t : *input gate's activation vector*
o_t : *output gate's activation vector*
h_t : *output vector of the LSTM unit*
c_t : *cell state vector*
W: *weight matrices and*
b_q : *q is* $\{i, f, o\}$ *is bias vector parameters which need to be learned during training*

Activation functions

σ_g : *sigmoid function.*
σ_c : *hyperbolic tangent function.*
σ_h : *hyperbolic tangent function.*

2.10 Recurrent Neural Network – Gated Recurrent Unit (RNN - GRU)

A gated recurrent unit (GRU) otherwise known as *fully gated recurrent unit* (FGRU) was proposed by Cho et al. [2014]. This unit can capture and model tasks whose dependencies relate to different time scales. In terms of the flow of information, this network has gating units which modulate its flow just like LSTM. However, it does not contain an output gate and they do not have separate memory cells making it have less parameters than LSTM.

2.11 Recurrent Neural Network – Minimal Gated Unit (RNN - MGU)

Zhou et al. [2016] proposed a gated unit for RNN which contain only a single gate hence designing a minimal gated hidden unit called Minimal Gated Unit. This design ensures it does not lose LSTM's accuracy benefits while maintaining the smallest number of gates [8]. There was not enough time to study this latest variant in this project.

2.12 Evaluation methodology

In this study we are focused mainly in empirically comparing the LSTM and GRU variants and not to achieve the state of the art results. Our interests will therefore be drawn to keeping the experiments fair and simple with LSTM as the baseline.

2.12.1 Evaluation of the algorithms

For the configurations of different algorithms on the data sets, the error measures that will be used to compare the performance is Root mean squared error (RMSE). There are two common metrics used to measure accuracy for continuous variables. Mean Absolute Error (MAE) and RMSE. Given a set of predictions, MAE (equation 2.9.1) will measure the mean magnitude of the errors. However, it does not consider the general direction.

$$\mathrm{MAE} = \frac{1}{n}\sum_{j=1}^{n}|y_j - \hat{y}_j|$$ Equation 2.9.1

RMSE on the other hand refers to the square root of the average of squared differences between what was observed and what the model predicts. It is given as:

$$\mathrm{RMSE} = \sqrt{\frac{1}{n}\sum_{j=1}^{n}(y_j - \hat{y}_j)^2}$$ Equation 2.9.1

Although Hyndman R.J. and Koehler A.B (2006) discuss the numerous measures of forecast accuracy, we pick RMSE mainly because it avoids the use of taking the absolute value which is undesirable as well as they are sensitive to outliers. RMSE (equation 2.9.2) also has the tendency to penalize large errors [20] while penalising small errors less.

2.12.2 Evaluation of the techniques

To take care of cross-domain variations the evaluation is done on three different datasets. For each variant we maintain a similar setup but each dataset has its own set of hyperparameters to achieve good performance. The tuning therefore only considered the baseline LSTM in getting the optimal hyperparameters for each data set. We need

a statistical hypothesis test to compare two machine learning classifiers. The choice however is still an open problem for interpreting machine learning results.

The question we want to address here is given two machine learning algorithms LSTM and GRU, with three data sets, which algorithm will produce more accurate predictions when trained on same data sets. To be fair in comparing, each algorithm is evaluated the same way and using each data set. Many types of test have been recommended by researches to accomplish this task. Thomas Dietterich (1998) in his widely cited paper recommends McNemar's and 5X2CV (cross validation) tests. McNemar's appears to be more relevant to nominal predictions [21]. In our case therefore out intention was to adopt a 10-fold cross validation procedure to evaluate each algorithm. Initially we had placed initialized each configuration with the same random seed to ensure similar splits to the training data and precisely similar evaluation for each algorithm. However, this approached was abandoned later before the first run to evaluate the performance was completed. By the time a decision to change evaluation method was made, the cross-validation evaluation had already run for 49.5 hours non-stop with noticeable negative performance signs of the operating system that was hosting the process (hardware is described in section 3.9.1 and software in 3.3.1). It appeared therefore not viable with the hardware architecture as it were to continue with cross-validation. Basic and simple **t-test** assessment was therefore used to evaluate means of 10 to 20 iterations.

2.12.3 Student's Test (T Test)

If we were to investigate the difference between two population averages, a t-test is going to be used. A **t-test** is done when two means of sample or population data are being compared. A researcher will normally want to state with some degree of confidence that the differences in means arising out of sample groups of data is too significant to be happening by chance. For example, if on calculation a t-test produces a t-value that equates to a probability of 0.02 then the likelihood of getting the difference in means happening by chance is 2 in 100 times and it will mean that likelihood is so low that the difference found in the samples could probably be in the population from which the sample was drawn. The **null hypothesis (H_0)** is a hypothesis in which the researcher is attempting to reject, nullify or disprove a given position by carrying

out experiments while the **alternative hypothesis(H_1)** is what the researcher really thinks is the cause of a certain alternative behaviour or phenomena. Therefore, the main objective of the one sample t-test is to determine whether the sample data has provided enough information to reject the null hypothesis[46]. We will invoke this hypothesis t-testing when looking at the GRU vs LSTM results.

2.13 LSTM Applications:

LSTM Networks is considered one of the state of the art algorithms and believed to perform better than other types of RNNs. Some of the areas where LSTM has been used include:

- **Handwriting Recognition:** This LSTM network [33] won a Handwriting Recognition Contest by achieving a recognition rate of up to 91%. It proved to surpass HMM-based models designed to recognize printed text using optical character.
- **Speech Recognition:** The architecture described in [34] was proposed by Graves et al. in 2013 and performed to a relatively recognisable level of 17.7% of accuracy. This was done on the TIMT Phoneme Recognition Benchmark and maintained that record up to recently. This same technique has also been attempted in large scale acoustic speech modelling [35].
- **Handwriting Synthesis**: A successful comprehensive study by Graves which demonstrated sequence generation tasks like text prediction by using LSTM to produce human-like handwriting synthetically [36].

3 Experimental Design

3.1 Introduction

In this section we search for the optimal parameters for each data set using LSTM architecture as the baseline and attempt to explore their performance regarding their ability to make useful predictions in restricted conditions. We considered number of neurons, batch size, number of epochs and learning rate as the main hyperparameters which can influence the most the performance of the machine learning neural network algorithm[39]. We use a combination of grid search and sequential search in finding the optimal parameters. This was informed by the hyperparameter space and the length of time it takes to run the experiments. For short runs we attempted grid search and for time-consuming ones we used sequential to shorten the time. On getting the optimal parameters, they are then applied to the both LSTM and GRU. An evaluation is then done to compare the performance results.

3.2 Data Sets

Three types of data sets were used. They are described below

3.2.1 Total Primary Energy Consumption (US):

There are many different types and sources of energy being produced by The United States. Generally they can be grouped as renewable and non-renewable; primary and tertiary; and fossil fuels. Primary energy is the raw form of energy before transformation to any tertiary form. Nuclear fuels (uranium), coal, the sun, oil, tides, natural gas and wood, the wind, the rivers, mountain lakes forms part of primary energy. It also includes all non-combustion uses of fossil fuels. Energy is the foundation of our highly developed society. Our public life organization as well as industrial development and scientific activities depend on energy.

Life without energy cannot be imagined. This energy is sometimes invisible and in most occasions taken for granted. It is therefore very important that its production, supply and consumption be carefully planned, developed and secured with great organizational, systemic and strategic efforts. Predicting energy consumption accurately can

help to inform decisions on production levels and sources. We therefore used energy consumption data set as one of those used for validating and evaluating performance of the algorithms. Our data set has the following summary statistics and trend captured monthly.

Period: 1973 January – 2018 March	**No. of. Instances:** 543
Minimum: 5438.115 btu (1975 June)	**Maximum:** 9676.835 btu (2018 January)
Mean: 7373.362 btu	**Std. Dev:** 948.209

1 Kilowatt Hour = 3412.14 btu 1 btu = 0.000293071-Kilowatt Hour. BTU-British Thermal Unit

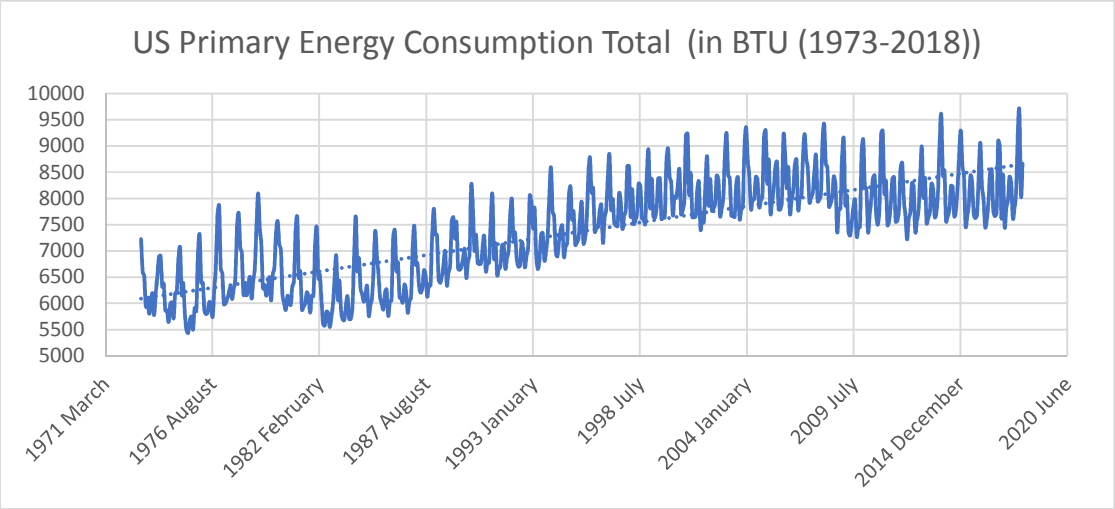


Figure 3.2-1: US Primary Energy Consumption in BTU

Sample Data Set: US Primary Energy Consumption in BTU

Month	Primary Energy Consumption Total
1973 January	7226.265
1973 February	6594.709
1973 March	6524.198

Table 3.2-1: US Primary Energy Consumption in BTU:

As can be seen from figure 3.2-1 the data are heavily jagged. To attempt to improve on the accuracy, the inputs are made statistically independent. Seasonality was common in this data set. A random experiment indicated that there was seasonality in this data set with a cyclic spike noticeable as from around November to March of every year. This pointed to the winter periods of the year when it is a very cold season demanding

more energy needs. There is therefore an increase in energy consumption over this period yearly. This dataset therefore contained annual trends and seasonality, which needed to be removed prior to modelling. Since there are 12 observations in a year, the difference functionality 3.2-1 was applied to the seasonal data set which removed the seasonality signal resulting in figure 3.2-2. Normalization was done on the results to increase the learning rate

Functionality 3.2-1

```
# creating / de-trending a differenced series
def deseason(dataset_train, interval=1):
        deseason_data = list()
        for i in range(interval, len(dataset_train)):
                value = dataset_train [i] - dataset_train [i - interval]
                deseason_data.append(value)
        return deseason_data
```

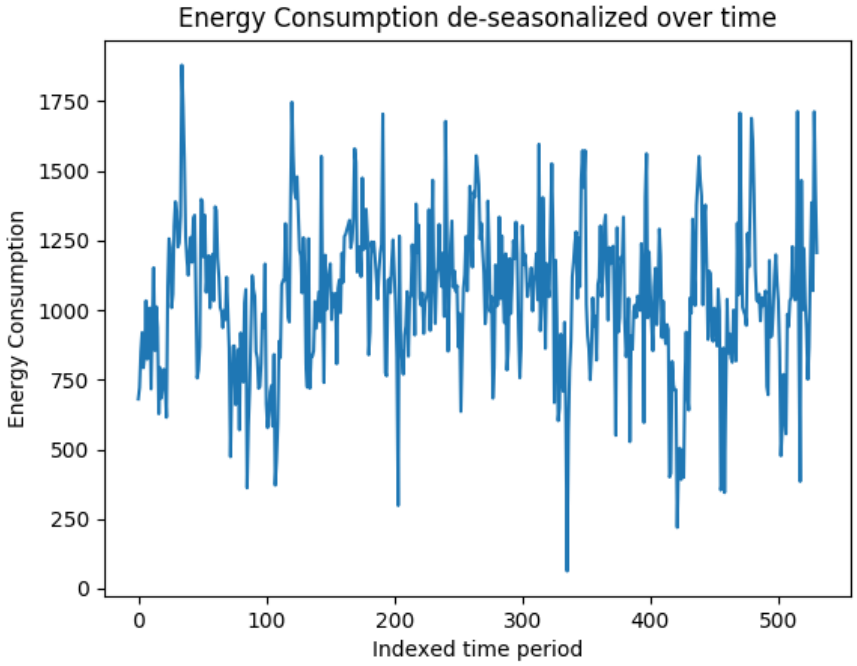


Figure 3.2-2: Seasonally adjusted Energy Consumption data

3.2.2 Consumer price index for all urban consumers: all items (US CPIAUCSL)

Consumer price index is an important measure of inflation. Every time we are in the grocery, gas station, mall, mart, shopping mall etc we experience its consequences. With every increase of prices, our ability to purchase (purchasing power) is eroded. At the same time if wages were to remain constant then we end up being poor eventually

lowering our living standards. It is therefore clear the role inflation, and by extension, CPI, plays as a measure of economic, and social stability and generally indicates the welfare of the consumer. CPI identifies the price changes across different product categories with regards to consumption. It's therefore vital that this tool is tracked and if possible forecasted to allow for a systematic approach to planning for inflation and other economic indicators. The United states department of labour has publicly available data resulting from tracking this important component. All items measured in terms of mean monthly changes in the price for goods and services paid. The data is seasonally adjusted (Figure 3.2-3). Our data set has the following summary statistics and trend captured monthly.

Period: Period: 1947 January – 2018 April
No. of. Instances: 856
Minimum: 21.48 (1947 January)
Maximum: 250.013 (2018 April)
Mean: 106.74
Std. Dev: 75.403

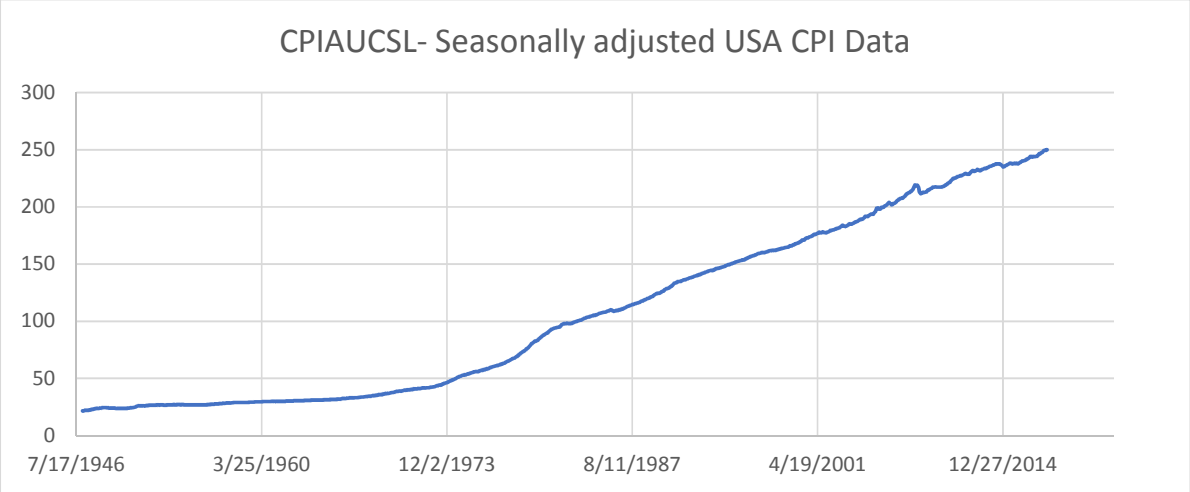


Figure 3.2-3: CPIAUCSL Seasonally adjusted US CPI data

Sample Data Set: CPIAUCSL Seasonally adjusted US CPI data

DATE	CPIAUCSL
01/01/1947	21.48
01/02/1947	21.62
01/03/1947	22
01/04/1947	22
01/05/1947	21.95

01/06/1947	22.08

Table 3.2-2*: CPIAUCSL Seasonally adjusted US CPI* sample *data*

3.2.3 US IBM Stock Prices

From the available literature, stock market prediction is complex. This is partly attributed to the market prices which are quite volatile and very unpredictable [7]. There are no consistent data patterns to be used to model trading prices for stock for a specified period in a near-perfect way. According to "In a Random Walk Down Wall Street" by Burton Malkiel (1973), the market could reflect all factors immediately as soon as they're made public if it were truly efficient to the extent that a blindfolded monkey throwing darts at a newspaper stock listing should do as well as any investment professional. It is therefore only a possibility to model the exact stock values of the future, but the stock price movements (that is, if it is going to rise or fall soon).

We picked US IBM stock data largely because of its unique stock price behaviour over time. This type of scenario makes learning more robust and is a good test of prediction for a variety of situations. There are fluctuations though which will be dealt with through normalization. The recent updates to the LSTM network has not benefited stock market prediction as far as available literature is concerned. The data is publicly available. Table 3.2-3 displays the descriptive statistics and 3.2-4 the sample data set.

The names of the associated attributes

Date: Date of trading

Open: Value during opening of the trading

High: Highest value during the trading day

Low: Lowest Value during the trading day

Close: Value during closing of the trading day

Volume: Volume of stock traded during the day

Summary Statistics:

Period: 1962 January 2nd – 2017 November 10

No. of. Instances: 14059

Description	Date	Open	High	Low	Close	Volume
Minimum	02 Jan 1962	3.39	3.566	3.324	3.39	0
Maximum	10 Nov 2017	186.01	186.46	185.06	186.36	83165905
Mean	-	48.536	48.987	48.112	48.554	5782966.342
Std. Dev.	-	49.271	49.664	48.913	49.298	5429532.641
No. of. Instances	14059	14059	14059	14059	14059	14059

Table 3.2-3: US IBM Stock Prices

Sample data set

Date	Open	High	Low	Close	Volume
02/01/1962	6.413	6.413	6.3378	6.3378	467056
03/01/1962	6.3378	6.3963	6.3378	6.3963	350294
04/01/1962	6.3963	6.3963	6.3295	6.3295	314365
05/01/1962	6.3211	6.3211	6.1958	6.2041	440112
08/01/1962	6.2041	6.2041	6.0373	6.087	655676
09/01/1962	6.1208	6.2376	6.1208	6.1621	592806

Table 3.2-4: US IBM Stock sample data set

NB: Added a column **Average = (High + Low)/2.** We are focusing on this is the attribute.

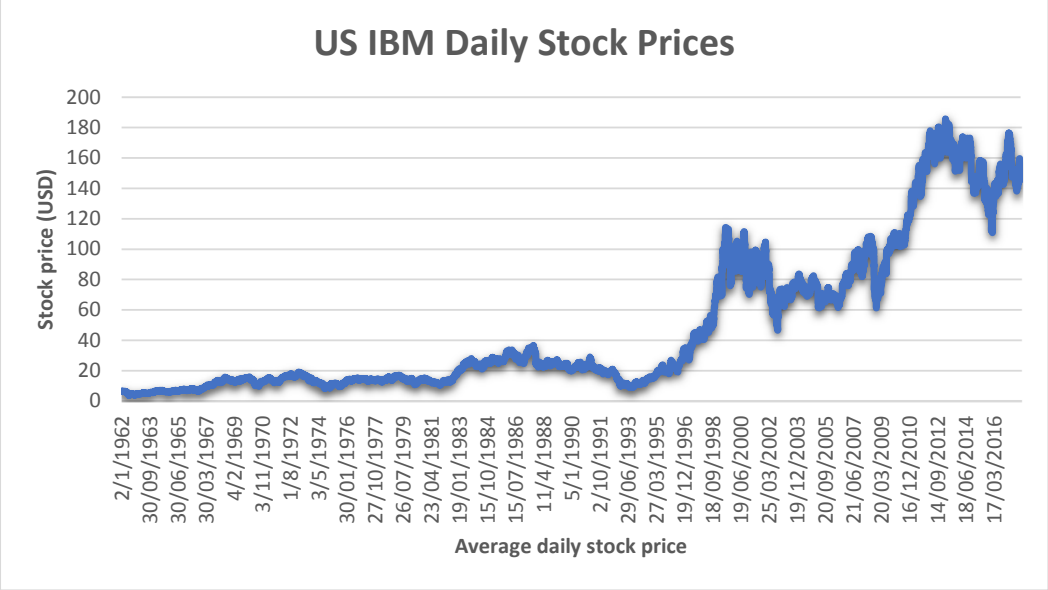


Figure 3.2-4: US IBM Average daily stock price

3.3 Implementation methodology

In this paper a quantitative research method was primarily used. Literature review was done to identify the main challenges as well as recent developments in the field. The implementation was carried out in a very explorative and iterative nature especially progressing from basic to more complex techniques.

The models were evaluated on at least three different types of datasets to ascertain their efficacies. Software libraries used include the following identified in table 3.3.1. Quality assurance on the code was done to ensure no bugs were inherited into the outcome of the experiments. Python 3.54 was used and code made available in Github [16] to ensure reproducibility. The project environment is tabulated below. Keras is written in python as a high-level neural networks API with the ability to run on top of Google's TensorFlow or Theano libraries.

3.3.1 Software libraries and tools

Tool/Library	**Version**
Keras	2.0.3 and 2.2.2 distributions
TensorFlow	1.0.0 and 1.1.0
Scikit-learn	0.18.2 and 0.19.2
Python	3.54 and 3.6.6
Matplotlib	2.2.3
Numpy	1.14.5
Pandas	0.23.4
sklearn	-
LSTM and GRU Libraries from KERAS	-

Table 3.3-1*: Software libraries and tools*

In this project, the focus is on time series using LSTM and GRU (its variant) while in-depth discussion of other alternative techniques has been avoided due to time constraints. There are many software packages providing the LSTM libraries, functionality and implementations, however we chose to use Keras with Tensorflow backend mainly because it provides a high-level API hence enabling faster experimentation. The framework is easy to use although moderately restrictive making custom implementations difficult.

3.4 Loss function

During training of a supervised learning task, a *loss function* tracks and calculates the distance between the desired output and what the network is producing as an output. *Error* is calculated as the difference between the predicted value and real value and

most regression problems will use the averaged squared distance between the true value and predicted value otherwise known as the *Mean Squared Error (MSE)* as discussed in section 2.12.1

3.5 Training Neural Networks (NNs)

During training of NN, the goal of a learning problem is to optimise the loss function by minimizing the error. This is done by tuning the parameters of the NN. To achieve this an optimization algorithm called the *gradient descent* is used to train the NNs. In this method the gradients of the loss function is calculated while considering the network parameters i.e. the weights and their biases. A small change in a network parameter will trigger a change in the loss value resulting in a *gradient* change. *Back-propagation* method which uses a chain rule of derivates is used in calculating the gradients [23].

A *learning rate* scalar is normally useful in updating the parameters towards the opposite direction of the gradient. This process is iterated through the training data over several number of passes called *epochs*. Each epoch pass-through triggers progression towards optimum parameter values hence minimizing the loss function. Large data sets make the loss function computation to slow down and sometimes infeasible. The solution to this has seen gradient descend being enhanced. *Stochastic Gradient Descent (SGD), Adam, RM-Prop and AdaGrad* [24] are variants of gradient descend which work by subdividing data sets into *training batches* and calculates the loss function before updating the parameters. These are used in practice to increase the efficiency of the training process. Sometimes as the parameters get close to the optimum, *decay* parameter is used to slow down the learning rate to avoid overshooting the optimum value.

Sometimes the network learns the training data too well but performs poorly during test data. This causes *overfitting* and is a condition which can be triggered by a complex model than is normally adequate for the process. This can be prevented by *early stopping* where a small validation subset is used from the training data. NN connections can also be removed at random from each epoch up to a certain maximum to control overfitting.

3.6 Hyperparameter Optimization.

Weights and biases are network parameters that are learned by the training algorithm. However, *batch size, learning rate, dropout, decay, number of hidden layers, number of epochs* etc are learning algorithm parameters required to be set by the user outside of the training process (introduced in section 3.5). These parameters are called **hyper-parameters** and their setting in a supervised learning task is called *tuning.* The hyperparameters significantly affect the performance of the model. Sometimes the process of tuning the hyperparameters is iterated until optimal values are achieved using a process called **HyperParameter Optimization (HPO)** and will form part of the subject of our research. Chapters 5,6,7 and 8 of [25] provides a detailed study of NNs, gradient descent and back-propagation and can be useful in learning more.

Hyper-parameter optimization aims at finding an optimal solution out of a possible training sets with a potential to minimize the expected error of the algorithm. However, calculating the expected error is impossible and so the solution to this HPO problem takes either of the two forms:

1. The manual/random approach: several attempts are made using different parameters until the best one is found. This uses some random combinations for a range of values for a defined number of iterations.
2. Grid search: attempts are made at different sets of pre-set lists of values of hyperparameters and the best combinations is selected based on cross-validation score.

Grid search is better in terms of guarantee for an optimal solution. However, the computing resources cost for this approach is extremely expensive. Random search minimizes the potential values that each parameter can take. Studies have shown that random experiments are more efficient than grid experiments for HPO in the case of several learning algorithms on several data sets [26] and so a combination of random, grid and sequential search for HPO will be adapted to this study depending on expected performance levels.

3.7 Random Search in scikit-learn

Random search is one of the model hyperparameter optimization technique. This is contained in the *RandomizedSearchCV* class of the Scikit-learn. The class function takes

in some arguments including the *estimator*, a dictionary of *parameters grid* and the *number of jobs* to allow for parallelised processing.

The search normally defaults to one thread but this can be changed by tuning the *number of jobs* to -1 which then triggers the use of all cores on the processing machine. The *parameters grid* holds the dictionary of the hyperparameters being evaluated which then maps onto an array of values to try. This method implements a randomized search over the parameters. A computation of budget refers to the number of sampled candidates or sampling iterations denoted by *number of iterations*.

The advantage of random search is that it allows choosing of an independent budget from the number of parameters and their possible values. Further efficiency is not affected when parameters with less influence on the performance are added.

3.8 Model design and intervention of all layers

Window period in LSTM was utilised and set depending on the data set. The form of training set is like $\{(X_{t-6}, X_{t-5}, \ldots, X_t) \rightarrow X+1\}$. In this paper we develop an optimized model for LSTM. Then using the optimized parameters, we apply the same to GRU. According to [27], some parameters are more important than others in influencing performance of an LSTM when tuned. We therefore consider four main parameters in optimizing and default the rest. These are the learning rate, the number of epochs, the batch size, the hidden layer size (number of layers and number of units in each layer) and the optimizer. The comparative analysis is done with regards to the baseline LSTM and a t-test for 10 iterations done.

LSTM Input Layer contains the 3D tensor. The number of samples, timesteps and the input dimension informs the shape of the tensor. The number of samples is informed by the data and the timesteps are determined through experiment, but input dimension remains 1. For the problems being discussed here t*he hidden layer* has no reference structure and therefore it was given based on experience and further determined by experiments.

3.8.1 Model 1: Basic LSTM Networks.

A basic LSTM network is configured with a single visible input layer, given number of LSTM blocks (neurons) of hidden layer and an output layer responsible for single value

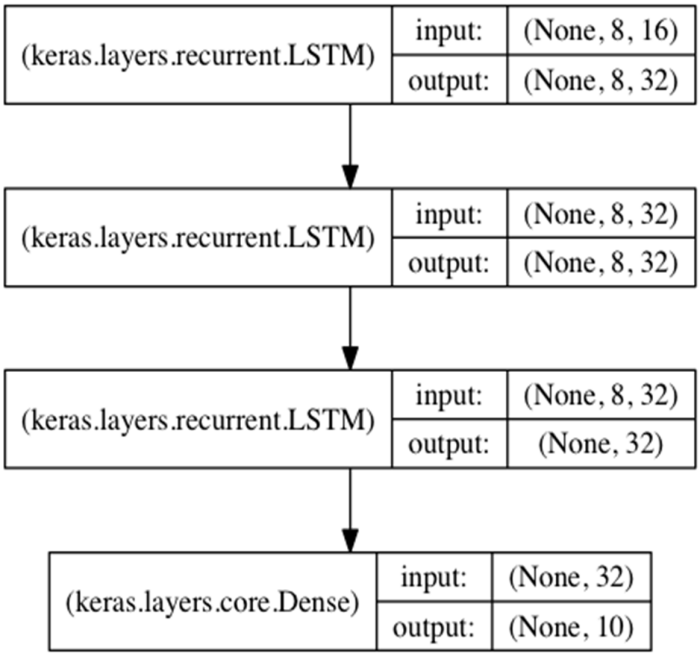
prediction. Sigmoid is used as the activation function for the LSTM blocks. Sigmoid function because of its nonlinearity and the computational simplicity of its derivative. [38]. We set a dropout to 0.2 to avoid over-fitting [37]. A simple experiment suggested 50 epochs for the training with a batch size of 1. But this will vary for the different data sets.

3.8.2 Model 2: Stacked LSTM Networks.

In stacked LSTM network, two or more layers are stacked on each other enabling the network to have superior capabilities for temporal representations. Figure 3.8-1 below represents a three-layer stacked LSTM network.

An experiment is done to attempt and place the optimal number of layers. The criteria for building models implied that we use similar number of parameters, similar dropout rate of 0.2, with the four specified parameters being tuned first. Testing across the three data sets

Stacked LSTM



- Stacked three LSTM layers.
- First and second layers returns a sequence
- Third layer doesn't return a sequence

Figure 3.8-1:Simple model of stacked LSTM

Keras Python deep learning has tools which achieve Stacked LSTM models. To achieve this, we would require LSTMs memory cells to have a 3D (3D tensor with shape (batch size, timesteps, input dim)) input such that the output of a memory cell will be the

value of the entire sequence as a 2D array when LSTM processes one input sequence of time steps. The following functionality 3.8-2 accomplishes a three stacked LSTM layer network.

Functionality 3.8-2

```
# This example of a whole sequence has a single output
from keras.models import Sequential
import numpy as np
from keras.layers import LSTM
# We are defining an LSTM model in which LSTM acts also a layer(output)
data_dimension = 16
timesteps = 8  #different for each dataset
LSTM_mod = Sequential()
LSTM_mod.add(LSTM(32, input_shape=( timesteps , data_dimension ),  return_se-
quences=true))
LSTM_mod.add(LSTM(32, return_sequences=true)
LSTM_mod.add(LSTM(32))
LSTM_mod.add(Dense(10, activation='softmax'))
LSTM_mod.compile(optimizer='adam', loss='mse')
# need to reshape data to 3D as
Data_array = np.array([sequential data]).reshape((1,3,1))
# The predictions can be made here
print(LSTM_mod.predict(Data_array))
```

3.8.3 GRID and Sequential Search for Hyperparameters

LSTM has many hyperparameters to be tuned and some of them were introduced in section 2.8. The challenge in tuning all of them is the kind of resources it will require and the amount of effort. Even though there are quite many parameters to be tuned; learning rate, hidden layer size and optimizer has been established as the most important hyperparameters to be tuned and which had the ability to influence the performance of the network while input noise and momentum had nil or reduced performance [39]. The *GridSearchCV* process will construct a combination of parameters before evaluating one model for each. Each individual model is evaluated using Cross validation with the default of 3-fold cross validation being used. While under

3.8.4 Random search

In contrast to Grid Search, Random search approach uses random, uniform distribution to sample hyperparameters from the parameters dictionary. The model will then be trained and evaluated based on the set of randomly sampled parameters. This is done for a preset number of times (iterations), and normally matches the time the user/experimenter is willing to wait. The number of iterations can be made low or high

depending on the goal of the experiment. What matters in simple random sampling delivers in randomised search when it comes to exploring a large set of possible hyperparameter space quickly and in an optimal manner. It is common to find set of hyperparameters (optimization surface) exists that can optimally satisfy the problem. Random search can hit one of this surface faster and still obtain the same accuracy as Grid search.

3.9 Experiments

3.9.1 Processing machines specifications

The experiment is implemented using LSTM models (TensorFlow) and executed on CPU with four cores. We had the following keys specs of the laptops used for processing: Lenovo computer G50-80 15.6 inch with storage of 1 GB. The processor is Quad Core i7 5th Generation, 2.6GHz running Windows 10. Initial manufacturer RAM was 8GB but was upgraded for project work to 16GB. Later after Lenovo crashed, most likely due to running out of virtual memory, and revival process took longer than anticipated, a new laptop was prepared for this process whose specifications were: MacBook Air (13-inch, 2017), Processor 1.8 GHz Intel, Core i5, 8GB RAM 1600 MHz DDR3 with Macintosh HD and macOS Sierra Version 10.

The evaluation experiments were carried out using MacBook air while the hyperparameter tuning was carried out using Lenovo Laptop. Fortunately, both phases were carried out in their entirety using each respective laptop.

3.9.2 Training Optimization Algorithm

Keras models is useful to Scikit-learn when wrapped with the KerasClassifier. There are functions that must be defined which creates and returns the sequential models. The constructor for the KerasClassifier class has a constructor which takes default arguments received when calls are made to the model.fit(). These arguments can be the number of epochs and the batch size. This same classifier class can accept new arguments delivered through the create_model() function. Scikit-learn provides a technique called GridSearchCV which accepts a dictionary of hyperparameters in the param_grid

argument. This is used in mapping the model parameter and the array of value arguments (hyperparameters) that needs to be tried. n_job argument determines the ability to utilize the all cores of the hosting machine.

For each dataset, optimization algorithm that was used to train the network was tuned using the default parameters. The options are the ones that are supported by Keras API. *Grid search* was used to optimize the optimizer since we had a finite number of options to search from. Here we evaluated ['Nadam', 'Adamax', 'Adadelta', 'SGD', 'Adagrad', 'RMSprop', 'Adam'] optimizers [43].

```
    # Use scikit-learn to grid search the optimization algorithm.
    from keras.wrappers.scikit_learn import KerasClassifier
    from keras.models import Sequential
    from keras.layers import Dense
    import numpy as np
    from sklearn.model_selection import RandomizedSearchCV, GridSearchCV
    # This function will create the model which is required for KerasClassifier
    def create_keras_model(optimizer='adam'): #defaulting the optimizer to adam
            # create model
            Keras_model = Sequential()
            Keras_model = Sequential() # LSTM network, changed to reflect each network (GRU, MGU)
            Keras_model.add(Dense(12, activation='relu', input_dim=8)) #using default values
            Keras_model.add(Dense(1, activation='sigmoid'))
            # compiling the model in preparation for fiting
            Keras_model.compile(loss='binary_crossentropy', optimizer=optimizer, metrics=['accuracy'])
            return Keras_model
    # To allow for reproducibility we must fix the random seed
    Init_seed = 8
    np.random.seed(Init_seed)
    # Then we have to load dataset
    dataset = numpy.loadtxt("dataset.csv", delimiter=",") # loading our dataset. For each dataset, this
process is done.
    # split data into test and train Y and X variables
    X = dataset[:,0:67] #ratio of training to test
    Y = dataset[:,67] #use the remainder as test
    #Keras model is then created
    Keras_model = KerasClassifier(epochs=100, build_fn= create_keras_model, batch_size=10, ver-
bose=0)
    # At this point we define the parameters for the grid search
    Optimizr_lstm= ['Nadam', 'Adamax', 'Adadelta', 'SGD', 'Adagrad', 'RMSprop', 'Adam']
    grid_for_parameters = dict(optimizer= Optimizr_lstm)
    search_grid = GridSearchCV(estimator= Keras_model, param_grid= grid_for_parameters , n_jobs=-
1) # parallelize the jobs
    result_from_grid = search_grid.fit(X, Y)
    # The results and output are analysed and summarised
    print("The best performance of: %f is achieved when using %s" % (result_from_grid .best_score_,
result_from_grid .best_params_))
    mean_score_test= result_from_grid .cv_results_['mean_score_test']
    stdd_dev = result_from_grid .cv_results_['std_test_score']
    paramtrs = result_from_grid .cv_results_['params']
    for mean, stddev, param in zip(mean_score_test, stdd_dev, paramtrs):
       print("%f (%f) is achieved using: %r" % (mean_score_test , stdd_dev , param))
```

```
    The best performance of: 0.691247 is achieved when using {'optimizer': 'Adam'}
                                                    0.357414 (0.024654) is achieved using: {'optimizer': 'SGD'}
    0.335187 (0.023892) is achieved using: {'optimizer': 'RMSprop'}
    0.679251 (0.030749) is achieved using: {'optimizer': 'Adadelta'}
```

```
0.652759 (0.015365) is achieved using: {'optimizer': 'Adamax'}
0.465412 (0.146526) is achieved using: {'optimizer': 'Adagrad'}
0.685125 (0.003354) is achieved using: {'optimizer': 'Nadam'}
0.698247 (0.029465) is achieved using: {'optimizer': 'Adam'}
```

ADAM emerged as the best optimizer in all the scenarios. According to Kingma et al. Adam is an algorithm for first-order gradient-based optimization of stochastic objective functions [41].

3.9.3 Tuning the number of epochs together with the batch size

In iterative *gradient descent* the *batch size* represents the network-picked number of patterns just prior to the weights being updated. During training this *batch size* will define the number of patterns that can be read at a time and memorised. The number of epochs represents the number of times (iterations) the entire training dataset will be shown to the network while training. [24]. For the batch size and number of epochs, we evaluate the batch sizes in groups from 5 then 10 to 50 in steps of 10 then 60, 80 and 100 and for epochs 10 to 50 in steps of 10 then to 200 in steps of 50 as shown in functionality 3.9.1 below [43].

Functionality 3.9.1

```
#To load the libraries, we need to import them
import numpy as np
import pandas as pd
from keras.wrappers.scikit_learn import kerasclassifier
from kerras import models
from sklearn.datasets import make_classification
from keras import layers
from sklearn.model_selection import randomizedsearchcv

#we are setting the random seed
np.random.seed(8)
```

```
# At this point we define the parameters for the grid search
Discrete_batch_sizes = [5,10,20,30,40,50,60,80,100]
number_of_epochs = [10, 20,30,40,50, 100,150,200]
optimizer = ['Adam']
parameter_grid = dict(optimizer=optimizer, epochs= number_of_epochs ,batch_size= Dis-
crete_batch_sizes, epochs= number_of_epochs)
parameter_grid = dict(optimizer=optimizer)
search_grid = GridSearchCV(param_grid= parameter_grid, estimator= Keras_model, n_jobs=-1) #
parallelize the jobs

##other functionality appears here
```

This was done for the three data sets and the optimum for each data set obtained. Baseline network being LSTM. The results below pointed out the best performing pair.

```
The best performance of: 0.696112 is achieved using {‘number of epoch’: 100, 'batch_size': 5}
0.352463 (0.023114) is achieved using: {‘number of epoch’: 10, 'batch_size': 10}
0.352463 (0.023114) is achieved using: {‘number of epoch’: 50, 'batch_size': 10}
0.651357 (0.022232) is achieved using: {‘number of epoch’: 10, 'batch_size': 20}
0.452140 (0.148963) is achieved using: {‘number of epoch’: 100, 'batch_size': 10}
0.659256 (0.015216) is achieved using: {‘number of epoch’: 50, 'batch_size': 20}
0.483283 (0.076066) is achieved using: {‘number of epoch’: 10, 'batch_size': 40}
0.653874 (0.018718) is achieved using: {‘number of epoch’: 50, 'batch_size': 40}
0.696112 (0.025213) is achieved using: {‘number of epoch’: 100, 'batch_size': 5}
0.654855 (0.025426) is achieved using: {‘number of epoch’: 100, 'batch_size': 40}
0.502512 (0.031004) is achieved using: {‘number of epoch’: 10, 'batch_size': 60}
0.663565 (0.004782) is achieved using: {‘number of epoch’: 100, 'batch_size': 60}
0.532241 (0.163013) is achieved using: {‘number of epoch’: 10, 'batch_size': 80}
0.604569 (0.051213) is achieved using: {‘number of epoch’: 50, 'batch_size': 60}
0.591023 (0.093894) is achieved using: {‘number of epoch’: 50, 'batch_size': 80}
0.662454 (0.055214) is achieved using: {‘number of epoch’: 100, 'batch_size': 80}

0.652434 (0.032855) is achieved using: {‘number of epochs’: 50, 'batch_size': 100}
0.403243 (0.106254) is achieved using: {‘number of epoch’: 10, 'batch_size': 100}
0.542996 (0.158025) is achieved using: {‘number of epochs’: 100, 'batch_size': 100}
The optimal performance turned out to be 100 epochs and batch_size of 5
```

3.9.4 Tuning the learning and decay rates

Updates on the weights are controlled by the learning rate, whereas the level of influence that the previous update has on the current weight update is controlled by the momentum. We tried several values as specified below for each. Functionality 3.9.3 displays the piece of code like the one used.

learning_rate = [0.001,0.002, 0.003, 0.01, 0.02, 0.03, 0.1, 0.2, 0.3, 0.4]

decay_rate = [0.0, 0.2, 0.4, 0.6, 0.8, 0.9]

Functionality 3.9.3

```
def create_lstm_model(momentum=0, learn_rate=0.01):
    learning_rate = [0.001,0.002, 0.003, 0.01, 0.02, 0.03, 0.1, 0.2, 0.3, 0.4]
    decay_rate = [0.0, 0.2, 0.4, 0.6, 0.8, 0.9]
    optimizer = Adam(lr=learn_rate, decay= decay_rate)
    param_grid = dict(decay= decay_rate, lr= learning_rate, momentum= momentum)
    sgd = optimizers.Adam(lr=0.01, decay=1e-6, momentum=0.9, nesterov=True) model.com-
pile(loss='mean_squared_error', optimizer=’Adam’)
```

Including the number of epochs would be recommended reason being there exists some form of relationship between the number of epochs, the amount of learning per batch (learning rate), the number of updates per epoch (batch size). However, an attempt on this was not possible due to architecture constraints where an experiment

on significantly small portion of data set (30 observations) and two values for each parameter did not yield any result even after 24.5 hours running and the experiment had to be aborted. This is something that future studies may want to consider.

3.9.5 Tuning the number of neurons for the hidden layer.

The representational capacity of the network at the level in topology is controlled by the number of neurons in the network. We tuned the number of neurons applicable to one hidden layer. Values ranging in steps of 5 from 1 to 500 were explored. For a large network optimization, more training is necessary and the batch size and number of epochs should be optimized together with the number of neurons. Functionality 3.9.4 displays the piece of code like the one used [43].

Functionality 3.9.4

```
# here we specify the grid search parameters
    neurons = [K for K in range (1,500,5)]
    parameters_grid = dict(neurons=neurons)
    grid_search = GridSearchCV(param_grid= parameters_grid, estimator=model,  n_jobs=-1)
```

3.9.6 Hyperparameter tuning optimized values

3.9.6.1 IBM Stock data: Batch Size and number of epochs.

The best RMSE was at batch size 5 producing a performance of 16.5 and 76.3 during training and test respectively. Figure 3.9-2 below suggests there could be little chance of better results after batch size of 20 but it is something that can be experimented on. As discussed in section 3.9.2, the best performing number of epochs was 100.

```
The best performance of: 0.696112 is achieved using {‘number of epoch’: 100, 'batch_size': 5}
```

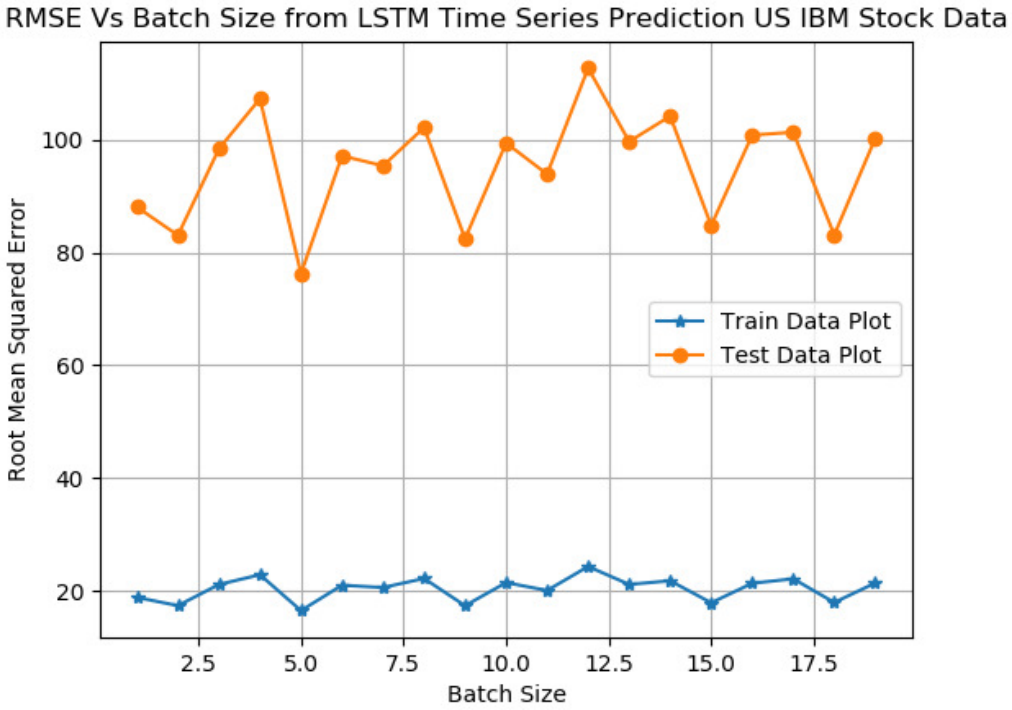


Figure 3.9-1: RMSE Vs. Batch Size from LSTM Time Series Prediction using US IBM Stock Data

Batch Size	4	**5**	6	7	8	9	10
Time (Sec)	7	**6**	6	6	5	5	5
Train RMSE	22.86	**16.49**	21.00	20.59	22.20	17.36	21.46
Test RMSE	107.30	**76.28**	97.15	95.39	102.17	82.46	99.43

Table 3.9-1: Results of RMSE Vs. Batch Size from LSTM Time Series Prediction using US IBM Stock Data

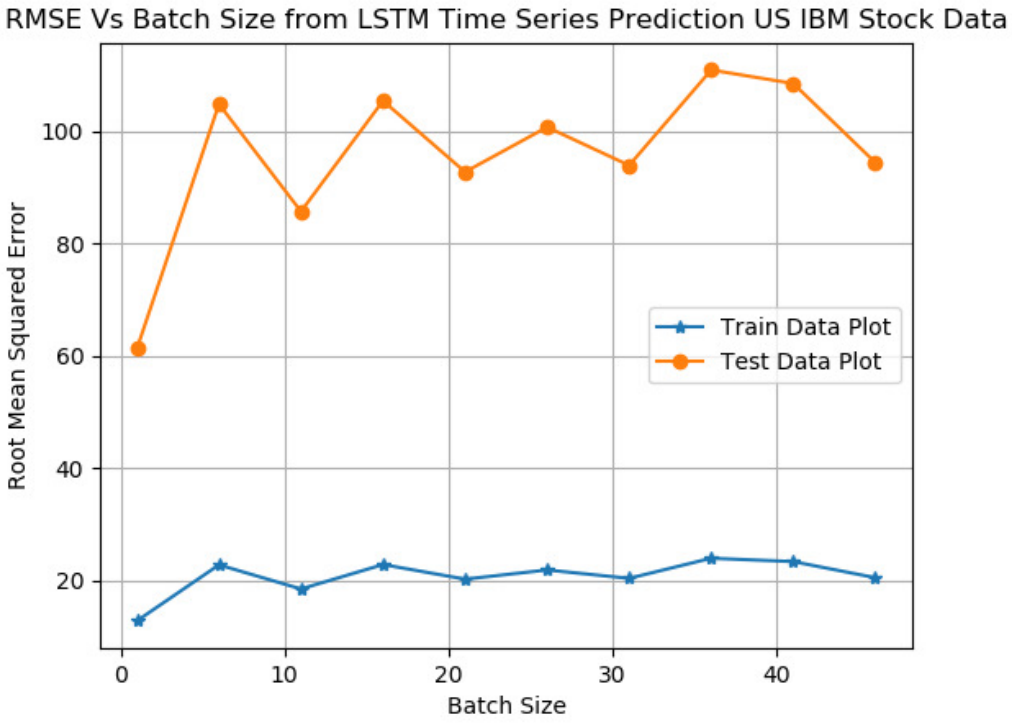


Figure 3.9-2: RMSE Vs. Batch Size from LSTM Time Series Prediction on US IBM Stock Data batch size up to 50

3.9.6.2 US CPI Data: Batch Size and number of epochs.

For this dataset, the best RMSE was observed at batch size 6 followed by size 12 producing a performance of 38.30 and 40.35 respectively during training. Figure 3.9-4 and table 3.9-2 below contains the results of the experiment.

Batch Size	2	4	**6**	8	10	**12**	14
Time	3	3	**3**	2	2	**2**	2
RMSE Train	47.78	48.15	**38.30**	43.71	49.49	**40.35**	55.30
RMSE Test	150.56	152.14	**126.85**	145.60	164.72	**134.37**	183.09

Table 3.9-2: Results of RMSE Vs. Batch Size from LSTM Time Series Prediction on US CPI Data

3.9.6.3 Energy Consumption: Batch Size and number of epochs.

This data set had some seasonality in it which required to be removed. Normalization was done on the residual information. The best RMSE was observed at batch size 4 with a performance slightly below 500 btu's for training data. Figure 3.9-4 below displays the results of the experiment.

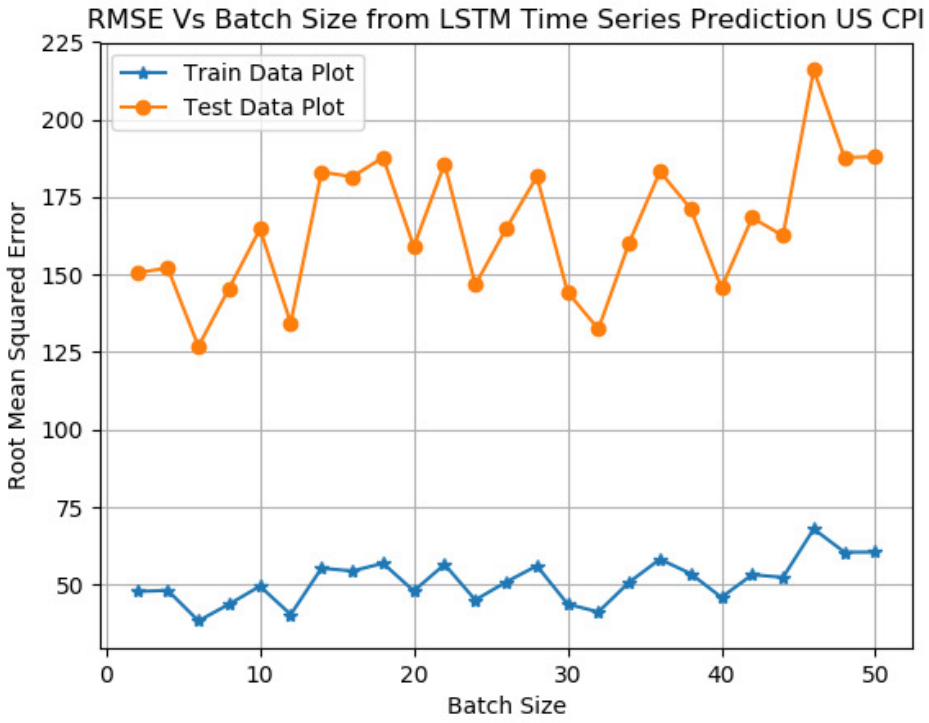


Figure 3.9-3: RMSE Vs. Batch Size from LSTM Time Series Prediction on US CPI Data

3.9.6.4 US CPI Dataset: Learning Rate.

For this dataset, the best RMSE was observed at learning rates of 0.5. and 0.4 for training and test data respectively producing a performance of 569.59 and 834.37 during training and test respectively. Figure 3.9-5 and table 3.9-3 below contains the results of the experiment.

Learning Rate	0.05	0.08	0.1	0.2	0.3	0.4	**0.5**	0.8
RMSE Train	888.86	824.87	824.47	634.93	896.08	809.87	**569.59**	919.74
RMSE Test	1210.55	1186.01	1071.85	699.21	588.67	514.45	**834.37**	492.10
Time (sec)	3	3	3	3	3	3	**4**	3

Table 3.9-3: Results of RMSE Vs. Learning Rate from LSTM Time Series Prediction on US CPI Data

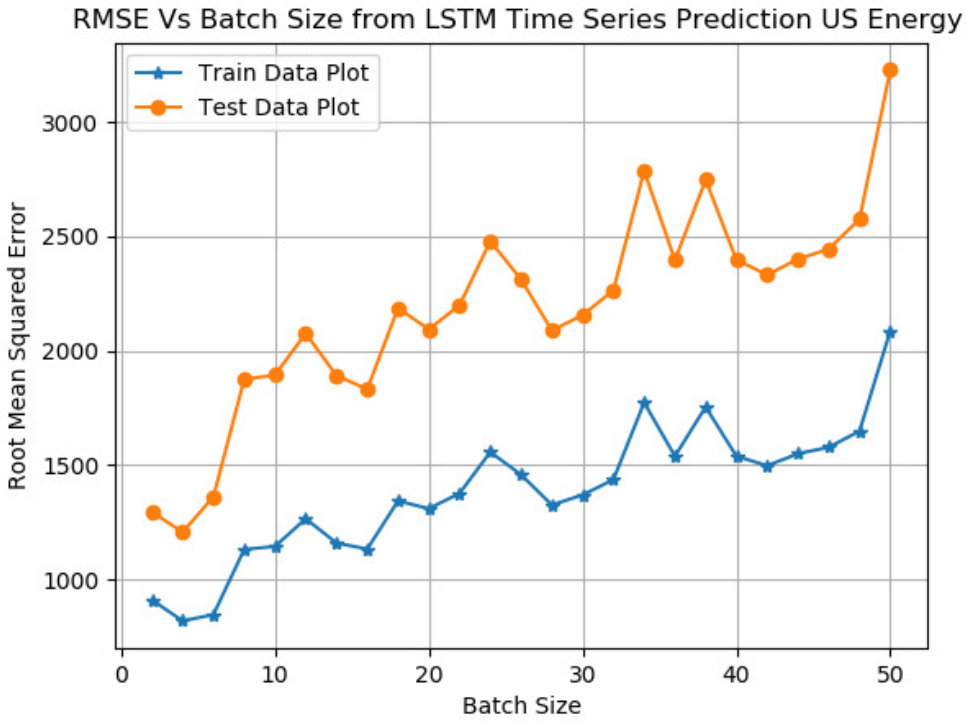


Figure 3.9-4: RMSE Vs. Batch Size from LSTM Time Series Prediction on US Energy Consumption

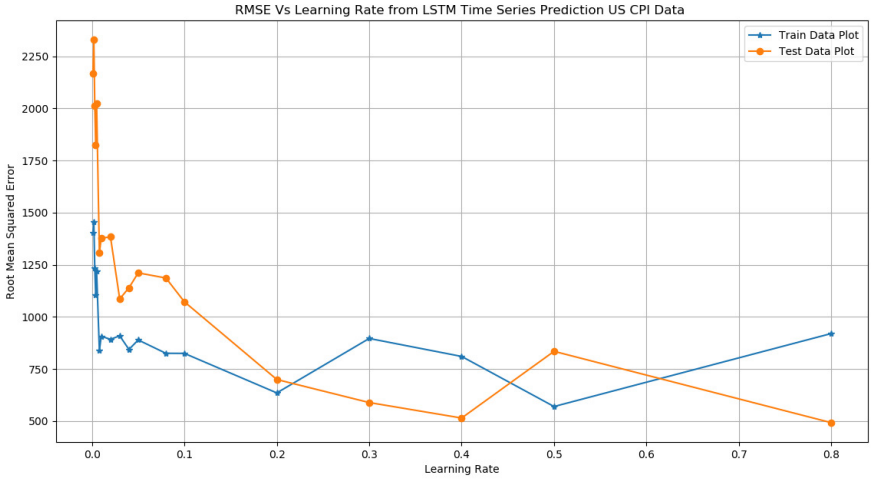


Figure 3.9-5: RMSE Vs. Learning Rate from LSTM Time Series Prediction on US CPI dataset

3.9.6.5 IBM Stock data: Learning Rate.

For this dataset, the best RMSE was observed at learning rates of 0.3. for training producing a performance of 0.61 during training. Figure 3.9-5 and table 3.9-3 below contains the results of the experiment.

Learning Rate	0.05	0.08	0.1	0.2	0.3	0.4	0.5	0.8
RMSE Train	12.02	2.58	1.98	0.73	0.61	0.67	0.71	3.78
RMSE Test	60.5	18.6	22.5	11.3	8.1	11.8	13.6	12.0
Time (sec)	6	7	7	7	7	7	7	7

Table 3.9-4: Results of RMSE Vs. Learning Rate from LSTM Time Series Prediction on US IBM *Dataset*

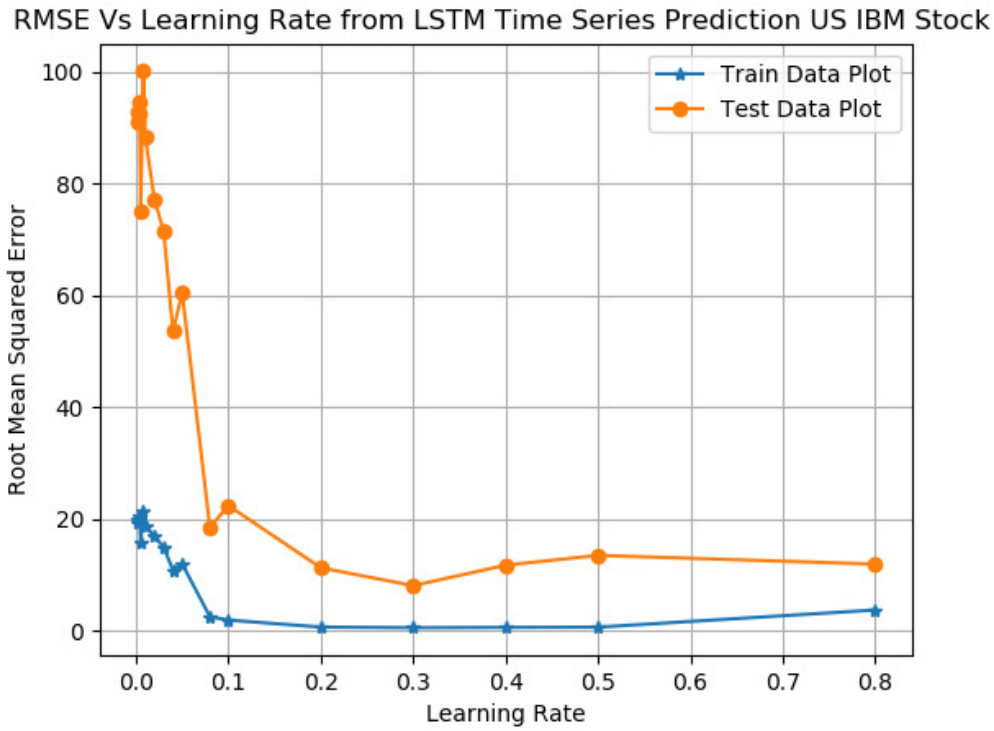


Figure 3.9-6: Results of RMSE Vs. Learning Rate from LSTM Time Series Prediction on US IBM Dataset

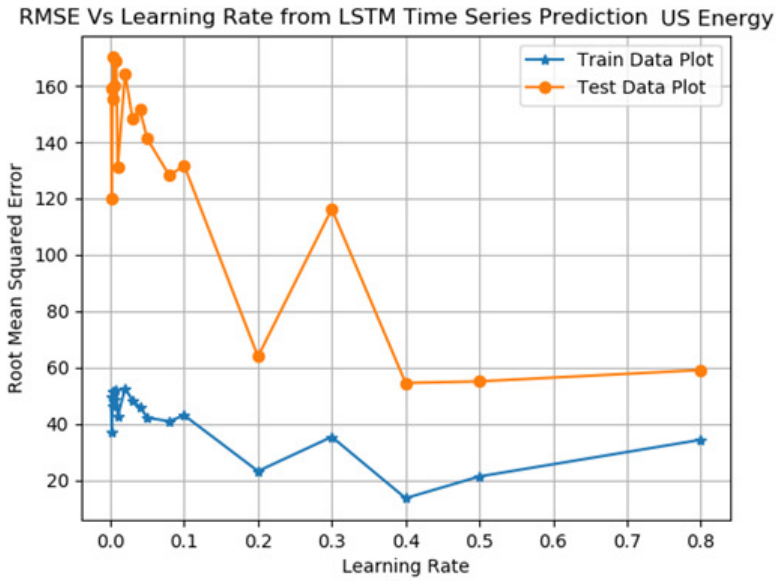


Figure 3.9-7: Results of RMSE Vs. Learning Rate from LSTM Time Series Prediction on US Energy Consumption data

Learning Rate	0.05	0.08	0.1	0.2	0.3	0.4	0.5	0.8
RMSE Test	141.20	128.17	131.72	63.87	116.28	54.50	55.06	59.08
RMSE Train	42.13	40.91	43.02	23.17	35.35	13.56	21.29	34.31
Time (sec)	4	5	4	4	5	5	5	5

Table 3.9-5: Results of RMSE Vs. Learning Rate from LSTM Time Series Prediction on US Energy Consumption data

3.9.6.6 Energy Consumption: Learning rate.

For this dataset, the best RMSE was observed at learning rate of 0.4. for training producing a performance of 13.56 during training. Figure 3.9-7 and table 3.9-5 above contains the results of the experiment.

3.9.6.7 IBM Stock data: neurons for the hidden layer.

For this dataset, 500 neurons produced the best RMSE for training with a performance of 0.61 during training. Figure 3.9-5 and table 3.9-6 below contains the results of the experiment.

Neurons	500	550	600	650	700	750	800	850	900	950	1000
Time (sec)	78	92	110	127	147	163	182	211	225	260	294
RMSE Test	1.30	1.39	1.34	1.36	1.33	1.32	1.40	1.32	1.36	1.30	1.31
RMSE Train	0.61	0.97	0.67	0.64	0.60	0.65	0.78	0.60	0.81	0.60	0.60

Table 3.9-6: Results of RMSE Vs. Neurons from LSTM Time Series Prediction on US IBM Stock data

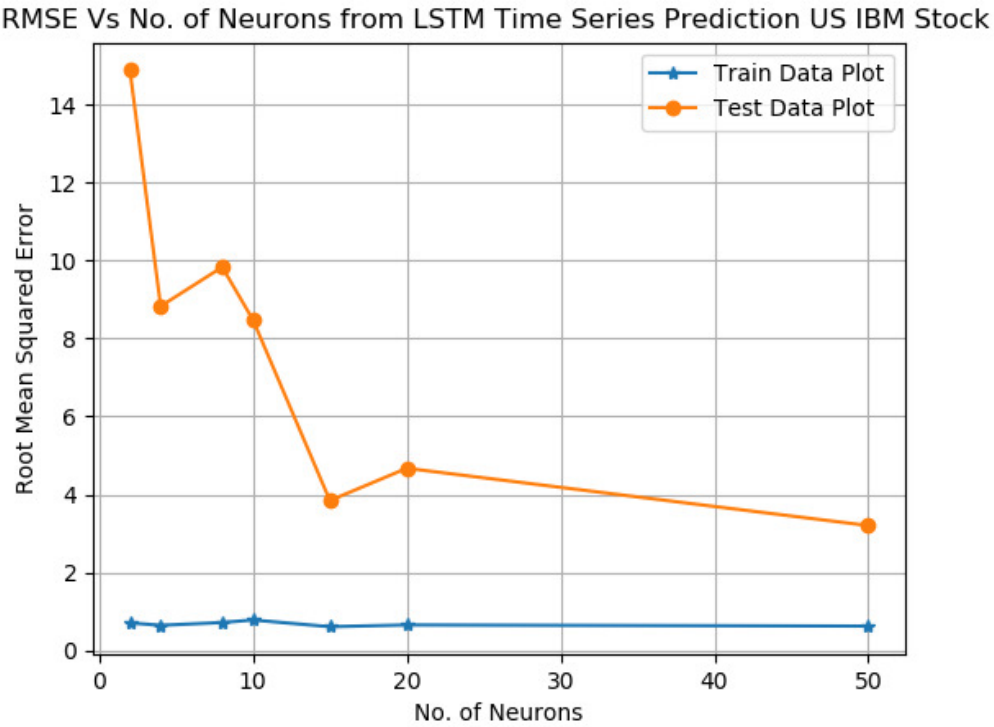


Figure 3.9-8: Results of RMSE Vs. Neurons from LSTM Time Series Prediction on US IBM Stock data

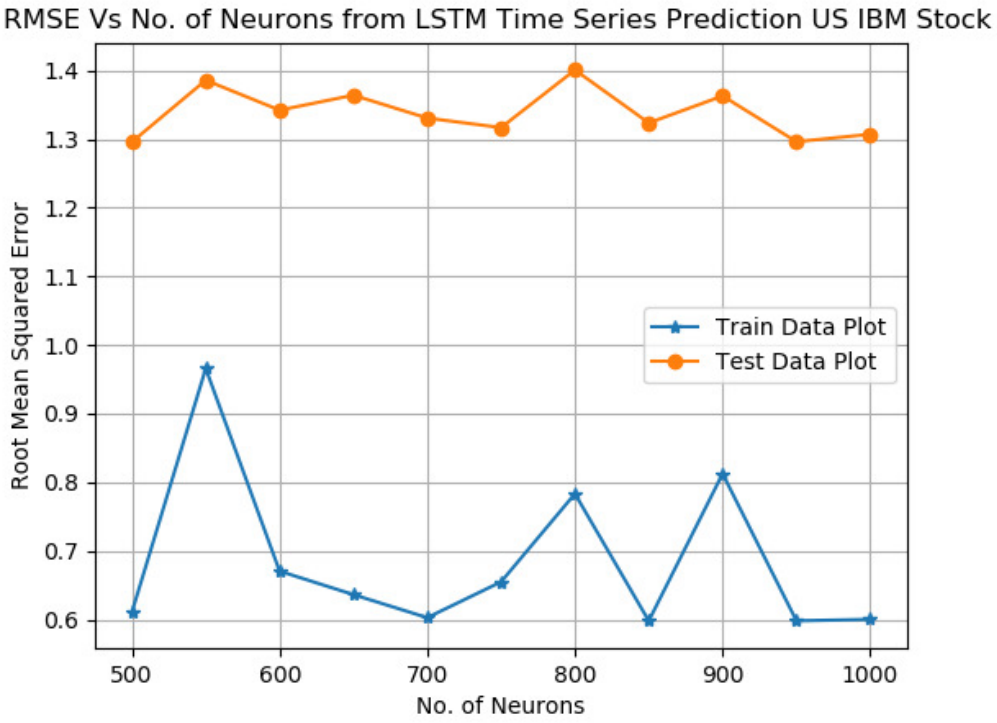


Figure 3.9-9: Results of RMSE Vs. Neurons from LSTM Time Series Prediction on US IBM Stock data-2

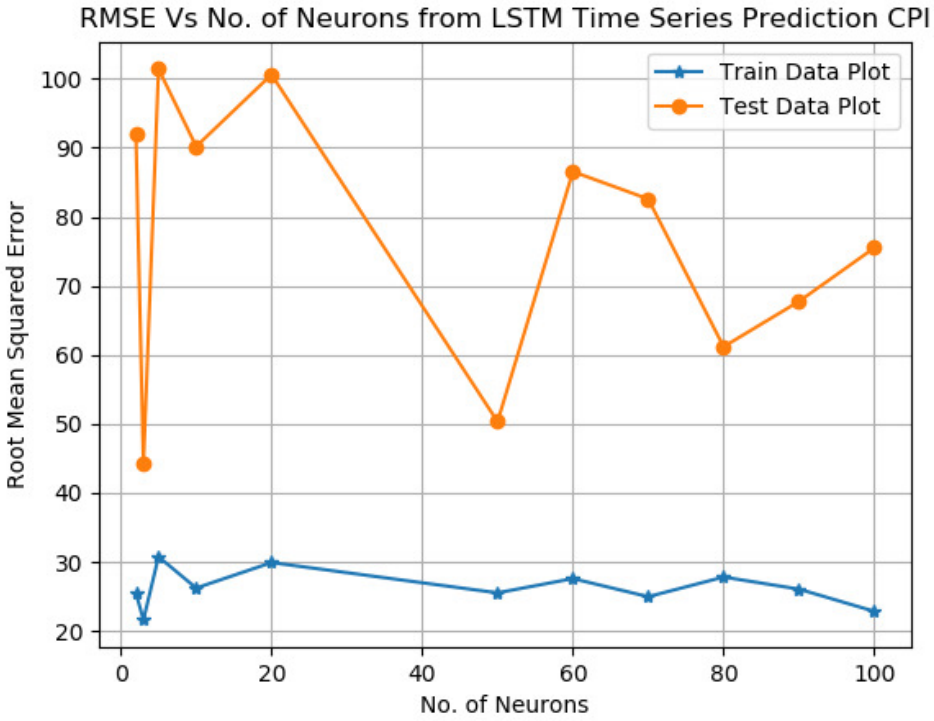


Figure 3.9-10: Results of RMSE Vs. Neurons from LSTM Time Series Prediction on US CPI data

3.9.6.8 US CPI data: neurons for the hidden layer.

For this dataset, 50 neurons produced the best RMSE for training with a performance of 48.89 during training. Figure 3.9-10 above contains the results of the experiment.

3.9.6.9 Energy Consumption: neurons for the hidden layer.

For this dataset, 50 neurons produced the best RMSE for training. Figure 3.9-11 displays the results of the experiment.

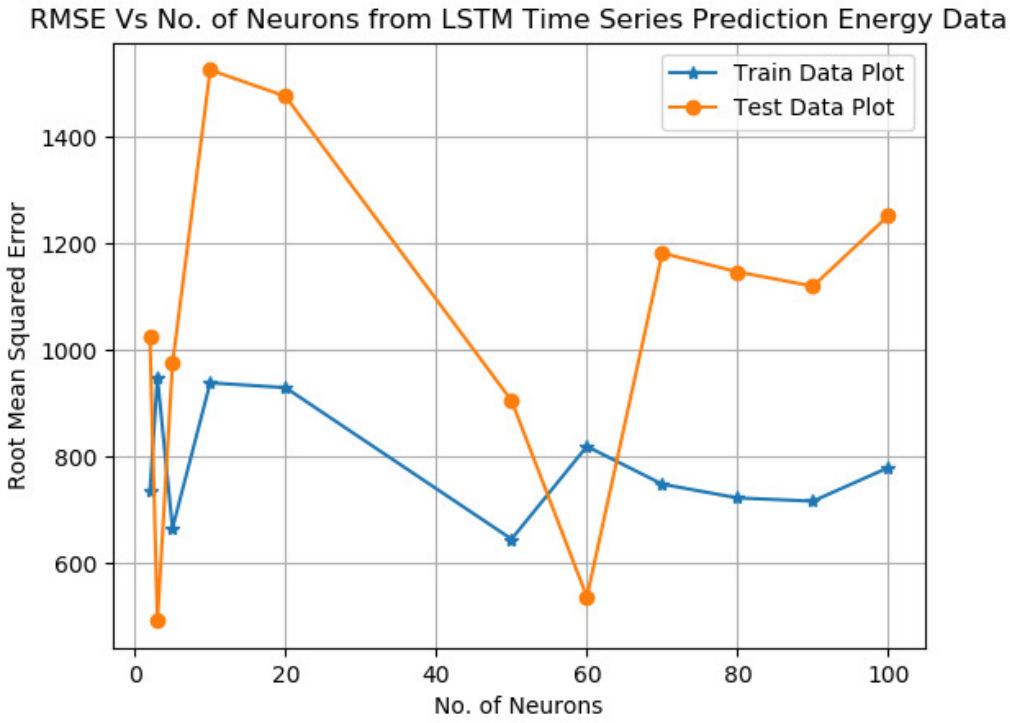


Figure 3.9-11: Results of RMSE Vs. Neurons from LSTM Time Series Prediction on US Energy data

4 Results and Analysis

4.1 Introduction

First, we built an LSTM network with the main basic layers, one hidden layer, one input layer and one output layer with default configurations of hyperparameters. To set us up, we start by choosing the optimum number of neurons using trial and error. Tanh and a linear activation function is used for the input and output layer. Linear activation is used for the input layer and the output layer while tanh is used for the hidden layer. The three data sets used in this experiment were split in the ration 20:8 which is roughly 72% for training and 28% for test. We did several experiments on learning rate, batch size, hidden units, optimizer, number of epochs. The intention was to use LSTM as a baseline to tune the network and obtain the optimal hyperparameters. RMSE was used as an error function. The resultant optimal parameters were then applied on the two networks, LSTM and GRU. The results for the two training techniques were more often similar. We made use of all the cores in the machines (section 3.9.1 describes the computer specifications). At first the configuration for hyperparameter optimization were done on a windows 10 machine configured with python 3.52. The Lenovo started freezing and eventually crashed most likely because the values stored in RAM got corrupted unpredictably and the host did not manage to recover from that. CPU usage most of the time during processing was at 96-100% with about 14GB of the 16 GB RAM being utilised. The crashing happened at the commencement of the LSTM and GRU comparative experiments. Fortunately, we had made a back of data files, source code files and hyperparameter optimization and configuration files. It was therefore not much of a challenge to set up and configure an available Macintosh machine for the same purpose. Some tests on hyperparameter configuration were done to confirm that the new environment did not introduce significant bias to the results.

We optimized the hyperparameters separately for practical purposes. Analysis of interactions of hyperparameters had discovered minimal effect on performance even for the highest influencers of network performance (learning rate and network size). This allowed us to treat hyperparameters independently. The hyperparameters were tuned

using a small network to save a lot of experimentation time [27]. Learning rate, hidden size, epoch size and batch size were independently configured.

4.2 Results

Default configuration values for all techniques and data sets:

Network activation function used was the default hyperbolic tangent (tanh) whereas for recurrent steps' activation used hard sigmoid. Dropout rate was fixed at 0.2. and momentum at 0.8. with decay rate being (learning rate / epochs). Fig 4.2.1 is a plot of GRU and LSTM on IBM Stock Data running 10 iterations using optimal parameters above.

4.2.1 IBM Stock price forecasting the optimal parameters

Learning rate: 0.3	Number of neurons: 500
Optimizer: Adam	Batch size: 5
Number of epochs: 100	Number of iterations: 10 and 20

Table 4.2-1: IBM results from optimal configuration on IBM stock data

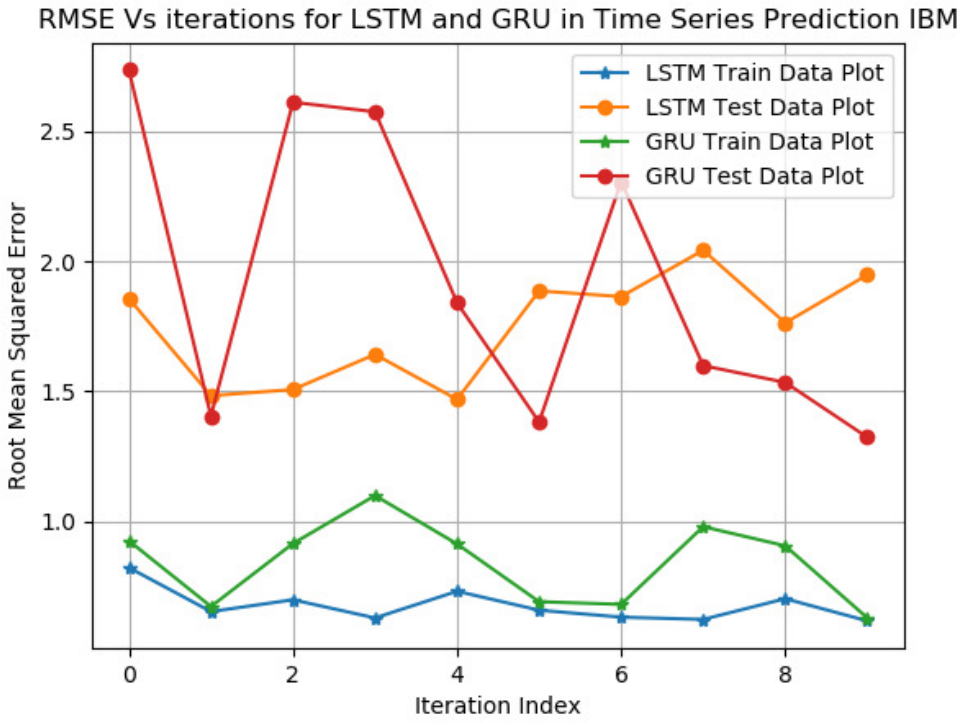


Figure 4.2-1: GRU and LSTM Results on IBM Stock Data on 10 iterations

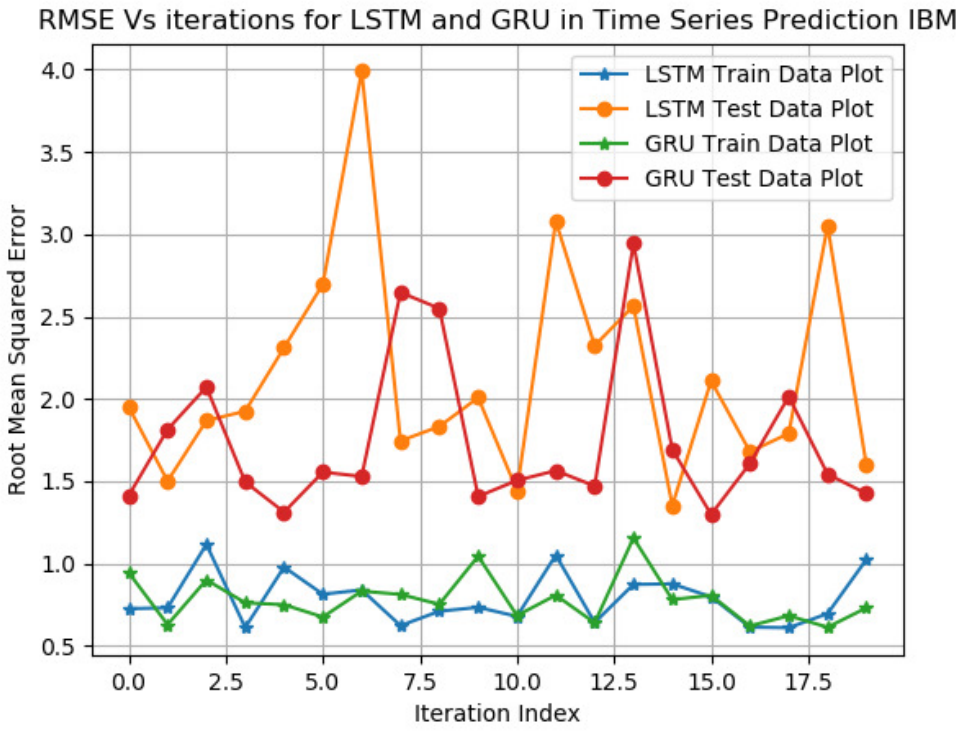


Figure 4.2-2: GRU and LSTM results on IBM Stock Data on 20 iterations

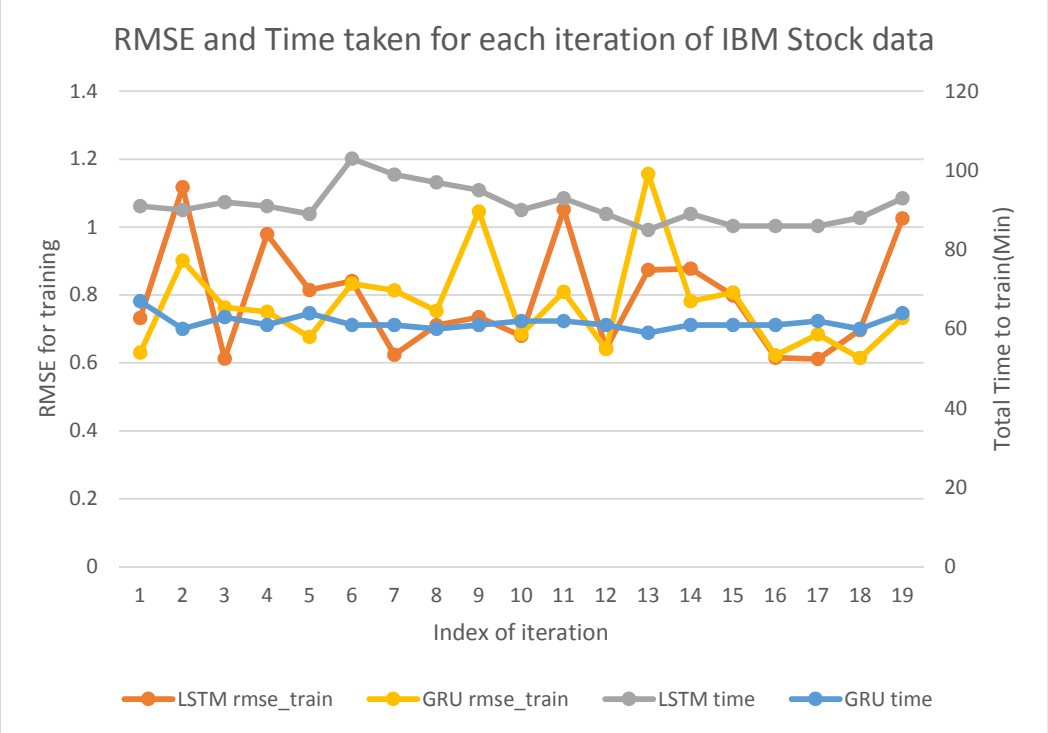


Figure 4.2-3: GRU and LSTM time and performance on IBM Stock Data (20 iterations)

LSTM	iter_id	1	2	3	4	5	6	7	8	9	10
LSTM	RMSE Train	0.73	0.73	1.12	0.61	0.98	0.81	0.84	0.62	0.71	0.74
LSTM	RMSE Test	1.95	1.50	1.87	1.93	2.31	2.70	3.99	1.75	1.83	2.01
LSTM	Time (min)	87.0	91.0	90.0	92.0	91.0	89.0	103.0	99.0	97.0	95.0
GRU	RMSE Train	0.95	0.63	0.90	0.76	0.75	0.68	0.83	0.81	0.75	1.05
GRU	RMSE Test	1.41	1.81	2.07	1.50	1.31	1.56	1.53	2.65	2.55	1.41
GRU	Time (min)	61.0	67.0	60.0	63.0	61.0	64.0	61.0	61.0	60.0	61.0

Table 4.2-2: Results from optimal configuration on IBM stock data

LSTM	iter_id	11	12	13	14	15	16	17	18	19	20
LSTM	RMSE Train	0.68	1.05	0.64	0.87	0.88	0.80	0.61	0.61	0.70	1.03
LSTM	RMSE Test	1.44	3.08	2.32	2.56	1.35	2.11	1.68	1.79	3.05	1.60
LSTM	Time (min)	90.0	93.0	89.0	85.0	89.0	86.0	86.0	86.0	88.0	93.0
GRU	RMSE Train	0.68	0.81	0.64	1.16	0.78	0.81	0.62	0.68	0.61	0.73
GRU	RMSE Test	1.51	1.57	1.47	2.94	1.69	1.30	1.61	2.01	1.54	1.43
GRU	Time (min)	62.0	62.0	61.0	59.0	61.0	61.0	61.0	62.0	60.0	64.0

Table 4.2-3: Results from optimal configuration on IBM stock data

	Mean	Std. Dev
LSTM RMSE Train	0.78844	0.15652
LSTM Time (min)	90.95	4.69574
GRU RMSE Train	0.78234	0.14270
GRU Time (min)	61.6	1.78885

Table 4.2-4a: Mean and Std. Dev IBM stock data

	P-Value (T Test)
RMSE	0.43863344
Time	0.00000000000000008441

Table 4.2.4b: T-test values

From table 4.2.4a the P Value for student t test on RMSE is **0.43863344** wwhich is more than 0.05 hence **we accept** the null hypothesis that **there is no** significant difference between LSTM and GRU for IBM stock data prediction regarding performance RMSE. GRU and LSTM had comparable performance on IBM stock data.

However, the P value for time is **0.00000000000000008441** allowing us to reject the null hypothesis that there is no difference between the means and conclude **a significant difference does exist**. Table 4.2.4a and Table 4.2.4b GRU takes less time to train as compared to LSTM.

4.2.2 Energy Consumption data forecasting using the optimal parameters

Learning rate: 0.4	Number of neurons: 50
Optimizer: Adam	Batch size: 4
Number of epochs: 100	Number of iterations: 20

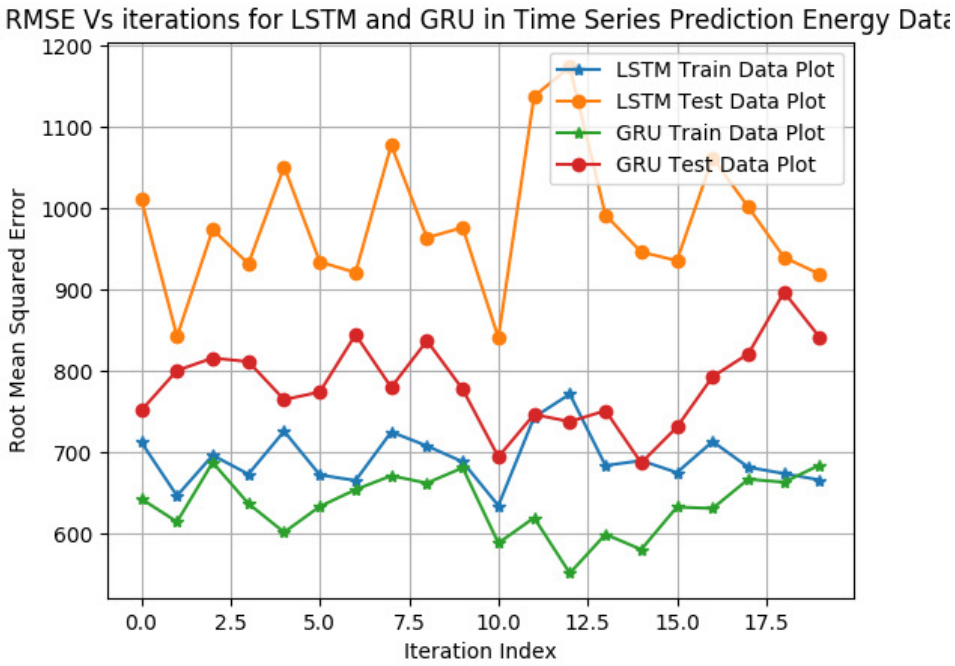


Figure 4.2-4: GRU and LSTM performance on Energy Consumption Data on 20 iterations.

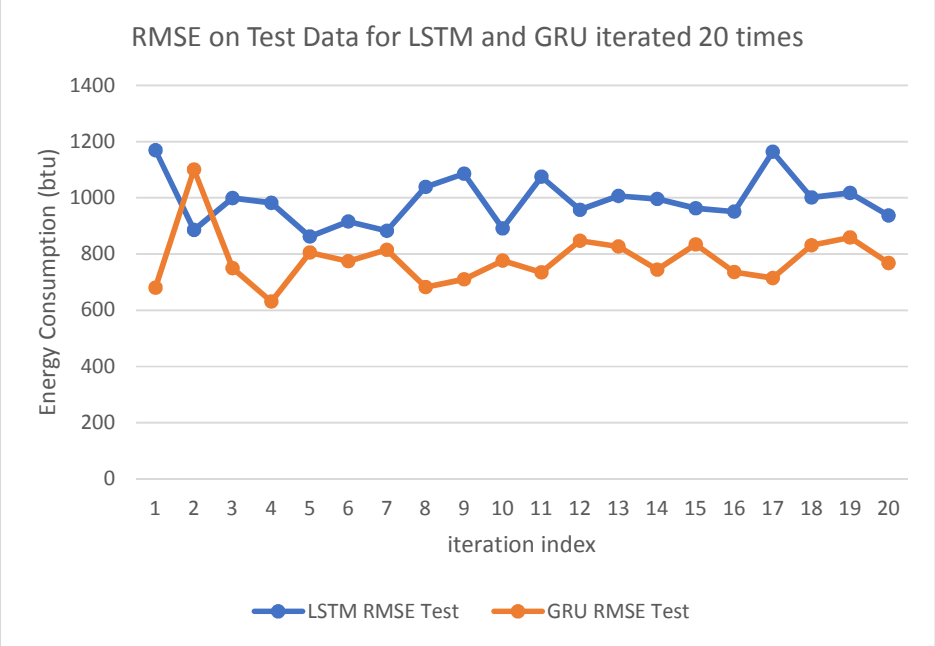


Figure 4.2-5: GRU and LSTM performance on test data on Energy Consumption Data on 20 iterations.

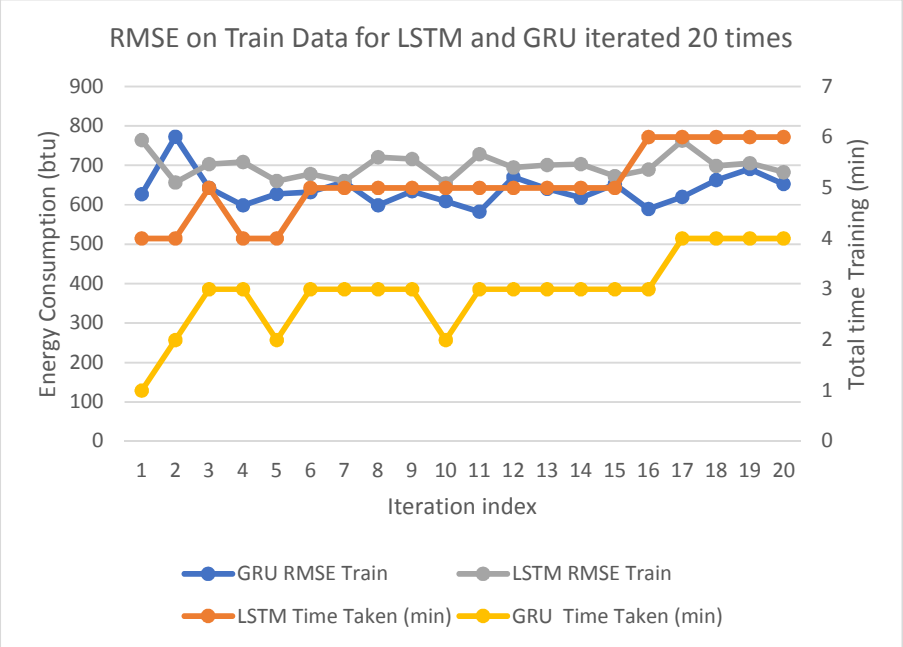


Figure 4.2-6: GRU and LSTM time and performance on training data on Energy Consumption Data on 20 iterations.

	Mean	Std. Dev
LSTM Train RMSE	697.9159353	31.12977
GRU Train RMSE	638.8481595	42.10173
LSTM Time (Min)	5.05	0.686333
GRU Time (Min)	2.95	0.759155
LSTM Test RMSE	989.2920463	86.66707
GRU Test RMSE	781.2608959	97.39702

	Paired T Test
	P Value
Training RMSE	0.000184281
Total Time Taken	0.00010312

Table 4.2-5a: Mean and Std. Dev Energy data Table 4.2.5b: T-test values

From table 4.2.5a the P Value for student t test on RMSE is **0.000184281** which is less than 0.05 hence **we reject** the null hypothesis that **there is no** significant difference between LSTM and GRU for Energy Consumption Data forecasting regarding performance RMSE. GRU and LSTM had comparable performance on IBM stock data. The test is at 95% significance value. 4.2.5a and Table 4.2.5b indicates that GRU takes less time to train as compared to LSTM and GRU's performance is significantly higher as indicated by lower RMSE on average. With this dataset GRU's overall performance is superior.

4.2.3 Consumer Price Index data forecasting using the optimal parameters

Learning rate: 0.4	Number of neurons: 50
Optimizer: Adam	Batch size: 12
Number of epochs: 100	Number of iterations: 20

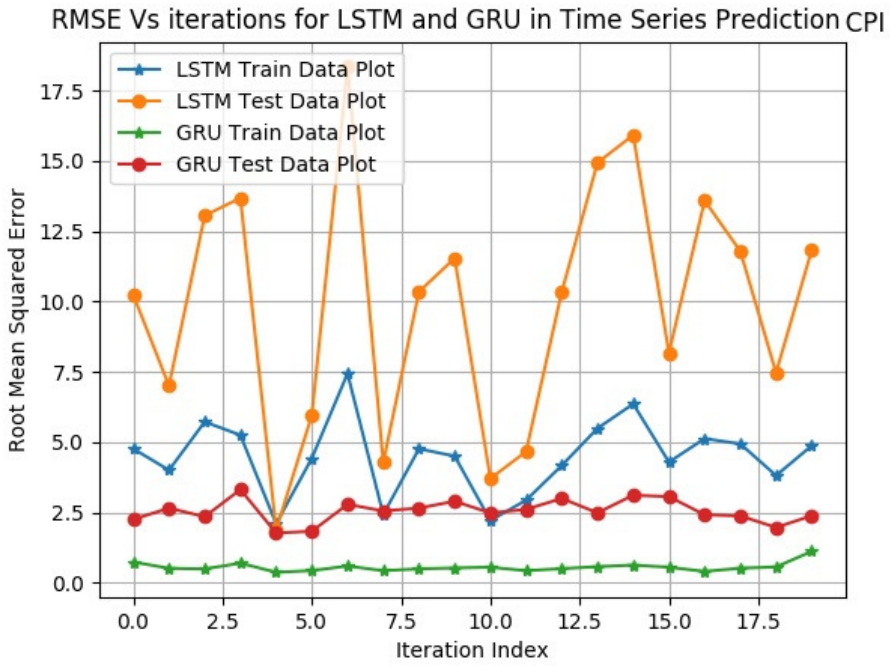


Figure 4.2-7: GRU and LSTM performance on CPI data on 20 iterations

	Mean	Std. Dev
LSTM Train	4.485262	1.348956
GRU Train	0.563188	0.161137
LSTM Time	9.35	0.933302
GRU Time	5.45	0.825578
LSTM Test	9.939039	4.445375
GRU Test	2.553771	0.413248

Table 4.2-6a: Mean and Std. Dev CPI data

	P-Value (T Test)
RMSE	0.000000000018443
Time	0.00000000000000000001

Table 4.2.6b: T-test values

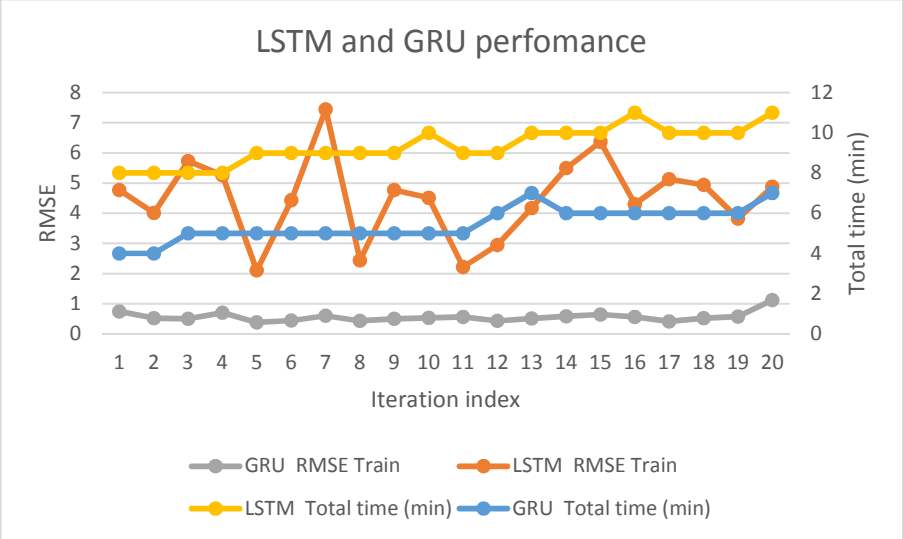


Figure 4.2-8: GRU and LSTM time and performance on CPI training data on 20 iterations

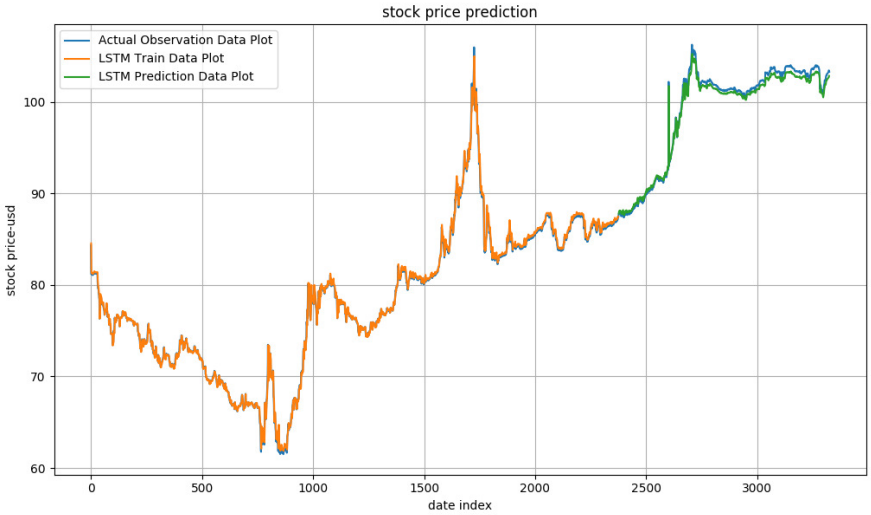


Figure 4.2-9: GRU optimized network prediction on some stock belonging to American Airlines

T-test P value for time is 0.000000000018443 and for performance is 0.00000000000000000001 at 95% significance value. We therefore **reject** the **null hypothesis** that there is no difference between the means and conclude **a significant difference does exist**. GRU takes less time to train as compared to LSTM. GRU's performance is significantly higher as indicated by lower RMSE on average. The conclusion is that that **vanilla** LSTM performs comparably to GRU on stock price prediction. However,

two datasets **CPI and Energy** consumption had GRU **performing** reasonably **better** than LSTM.

4.3 Critical evaluation

k-fold Cross Validation. There were noticeable variances in the results that were being achieved especially for LSTM majorly because of lack of cross validation. Configuration of the cross validation on a high-performance environment may be tested to check the stability of the results.

Parallelizing the training. Even though I was using all the six cores of my computer machine, neural networks have been known to be very slow to train. An exhaustive search for neural networks involves trying a lot of different parameters. Considering AWS instances may seem appropriate in the future.

Sample sets of datasets. While neural networks are slow to train, trying a smaller sample of the training dataset helps to give some general directions. Optimal configurations may not be achieved on the sample and may require the whole data set. This was the case for IBM stock prediction where samples of data set were used.

Transferring optimal results between problems. Optimal configurations on each new problem may differ. It is unlikely to have optimal results discovered in one problem getting transferred to another problem. An attempt was made to configure optimal settings for each new dataset. However, training resources were limiting and so the hyperparameter solution space was limited. At some point we defaulted the search to sequential independent searches rather than random search or Grid search. This was informed by hardware capacity.

Reproducibility is a problem. For most of the experiments the seed for the random number generator in Numpy is set in order to attempt to achieve reproducibility. However, this is not normally the case. Keras has presented more reproducibility for grid and random searches that the basic configuration we attempted to achieve here. I believe this is a subject that can be explored in the future.

4.4 Conclusion and future work

We optimized the hyperparameters separately for practical purposes. The achieved optimal configurations indicated that the vanilla LSTM performs comparably to GRU

on stock price prediction. However, two datasets on CPI and Energy consumption had GRU performing better than LSTM. Neural networks are a hurdle to new comers to the field since in most cases intuition of experts and their experiences inform most practical choices. Our attention has been to provide some evidence for some of those intuitions. New insights on architecture choice for specific datasets and hyperparameter tuning for LSTM and GRU have been presented. The study has also provided domains where LSTM can be equally useful as GRU and others where GRU is deemed superior. We evaluated different hyperparameters for the commonly known LSTM-architecture, otherwise called vanilla LSTM, for Time Series Analysis. We recommend that different tasks should consider different configurations of the hyperparameters using the approach that was adopted. In future performance between GRU and the recent MGU(MGRU) can be investigated. More complex configurations of LSTM, GRU and MGU can also be explored and or investigated to gain new insides and hopefully improve on performance. Experiments on wider sets of data may be carried out with ideally appropriate hardware architectures. We were also limited to single steps ahead in terms of prediction. This scope can be expanded to have prediction spanning several steps which may useful in long-term planning.

5 References

[1]. Hochreiter, S. (1998). The vanishing gradient problem during learning recurrent neural nets and problem solutions. *International Journal of Uncertainty, Fuzziness and Knowledge-Based Systems*, *6*(02), 107-116.

[2]. Tsay, R. S. (2005). *Analysis of financial time series* (Vol. 543). John Wiley & Sons.

[3]. Gheyas, I. A., & Smith, L. S. (2009, July). A neural network approach to time series forecasting. In *Proceedings of the World Congress on Engineering* (Vol. 2, pp. 1-3).

[4]. LeCun, Y., Bengio, Y., & Hinton, G. (2015). Deep learning. *nature*, *521*(7553), 436.

[5]. Owda, H. M., Omoniwa, B., Shahid, A. R., & Ziauddin, S. (2014). Using Artificial Neural Network Techniques for Prediction of Electric Energy Consumption. *arXiv preprint arXiv:1412.2186*.

[6]. Schmidhuber, J., & Hochreiter, S. (1997). Long short-term memory. *Neural Comput*, *9*(8), 1735-1780.

[7]. Malkiel, B. G., & Malkiel, J. A. (1973). Male-female pay differentials in professional employment. *The American Economic Review*, *63*(4), 693-705.

[8]. Zhou, G. B., Wu, J., Zhang, C. L., & Zhou, Z. H. (2016). Minimal gated unit for recurrent neural networks. *International Journal of Automation and Computing*, *13*(3), 226-234.

[9]. Längkvist, M., Karlsson, L., & Loutfi, A. (2014). A review of unsupervised feature learning and deep learning for time-series modeling. *Pattern Recognition Letters*, *42*, 11-24.

[10]. Bengio, Y., Simard, P., & Frasconi, P. (1994). Learning long-term dependencies with gradient descent is difficult. *IEEE transactions on neural networks*, *5*(2), 157-166.

[11]. Martens, J., & Sutskever, I. (2011). Learning recurrent neural networks with hessian-free optimization. In *Proceedings of the 28th International Conference on Machine Learning (ICML-11)*(pp. 1033-1040).

[12]. Gers, F. A., Schmidhuber, J., & Cummins, F. (1999). Learning to forget: Continual prediction with LSTM.

[13]. Graves, A., Mohamed, A. R., & Hinton, G. (2013, May). Speech recognition with deep recurrent neural networks. In *Acoustics, speech and signal processing (icassp), 2013 ieee international conference on* (pp. 6645-6649). IEEE.

[14]. Luong, M. T., Sutskever, I., Le, Q. V., Vinyals, O., & Zaremba, W. (2014). Addressing the rare word problem in neural machine translation. *arXiv preprint arXiv:1410.8206*.

[15]. Mou, L., Ghamisi, P., & Zhu, X. X. (2017). Deep recurrent neural networks for hyperspectral image classification. *IEEE Trans. Geosci. Remote Sens*, *55*(7), 3639-3655.

[16]. Shengnan Y. (2018). Financial Time Series Analysis of Stock Data, 2-3.

[17]. Ghiassi, M., Saidane, H., & Zimbra, D. K. (2005). A dynamic artificial neural network model for forecasting time series events. International Journal of Forecasting, 21(2), 341-362.

[18]. Yang, Q., & Wu, X. (2006). 10 challenging problems in data mining research. *International Journal of Information Technology & Decision Making*, *5*(04), 597-604.

[19]. Github: https://github.com/RaymondRono/LSTM-and-GRU-Prediction.

[20]. Hyndman, R. J., & Koehler, A. B. (2006). Another look at measures of forecast accuracy. International journal of forecasting, 22(4), 679-688.

[21]. Dietterich, T. G. (1998). Approximate statistical tests for comparing supervised classification learning algorithms. *Neural computation*, *10*(7), 1895-1923.

[22]. "Introduction to Artificial Neural Networks - Part 1." [Online]. Available: http://www.theprojectspot.com/tutorial-post/introduction-to-artificial-neural-networks-part-1/7.

[23]. LeCun, Y., Bengio, Y., & Hinton, G. (2015). Deep learning. nature, 521(7553), 436. Rumelhart, D. E., Hinton, G. E., & Williams, R. J. (1986). Learning representations by back-propagating errors. *nature*, *323*(6088), 533.

[24]. Ruder, S. (2016). An overview of gradient descent optimization algorithms. *arXiv preprint arXiv:1609.04747*.

[25]. Goodfellow, I., Bengio, Y., Courville, A., & Bengio, Y. (2016). *Deep learning* (Vol.1). Cambridge: MIT press.

[26]. Bergstra, J., & Bengio, Y. (2012). Random search for hyper-parameter optimiz-tion. *Journal of Machine Learning Research*, *13*(Feb), 281-305.

[27]. Greff, K., Srivastava, R. K., Koutník, J., Steunebrink, B. R., & Schmidhuber, J. (2017). LSTM: A search space odyssey. *IEEE transactions on neural networks and learning systems*, *28*(10), 2222-2232.

[28]. https://medium.com/mlreview/understanding-lstm-and-its-diagrams-37e2f46f1714. Accessed 08/21/2018

[29]. http://colah.github.io/posts/2015-08-Understanding-LSTMs/.Accessed 07/27/2018

[30]. https://medium.com/deep-math-machine-learning-ai/chapter-10-1-deepnlp-lstm-long-short-term-memory-networks-with-math-21477f8e4235. Accessed 08/11/2018

[31]. https://en.wikipedia.org/wiki/Long_short-term_memory. Accessed 08/15/2018

[32]. Wu, Z., & King, S. (2016). Investigating gated recurrent neural networks for speech synthesis. *arXiv preprint arXiv:1601.02539*.

[33]. A. Graves, M. Liwicki, S. Fernandez, R. Bertolami, H. Bunke, and J. Schmidhuber. A Novel Connectionist System for Unconstrained Handwriting Recognition. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 31(5):855–868, May 2009.

[34]. A. Graves, A.-R. Mohamed, and G. Hinton. Speech recognition with deep recurrent neural networks. In *2013 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 6645–6649, May 2013.

[35]. Hasim Sak, Andrew W. Senior, and Françoise Beaufays. Long short-term memory recurrent neural network architectures for large scale acoustic modeling. pages 338–342, 2014.

[36]. Graves, A. (2013). Generating sequences with recurrent neural networks. *arXiv preprint arXiv:1308.0850*.

[37]. Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I., & Salakhutdinov, R. (2014). Dropout: a simple way to prevent neural networks from overfitting. *The Journal of Machine Learning Research*, *15*(1), 1929-1958.

[38]. Han, J., & Moraga, C. (1995, June). The influence of the sigmoid function parameters on the speed of backpropagation learning. In *International Workshop on Artificial Neural Networks* (pp. 195-201). Springer, Berlin, Heidelberg.

[39]. Greff, K., Srivastava, R. K., Koutník, J., Steunebrink, B. R., & Schmidhuber, J. (2017). LSTM: A search space odyssey. *IEEE transactions on neural networks and learning systems*, *28*(10), 2222-2232.

[40]. https://keras.io/optimizers/ Accessed 07/27/2018

[41]. Kingma, D. P., & Ba, J. (2014). Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*.

[42]. Adhikari, R., & Agrawal, R. K. (2013). An introductory study on time series modeling and forecasting. *arXiv preprint arXiv:1302.6613*.

[43]. **https://machinelearningmastery.com/** Accessed 07/27/2018

[44]. Adhikari, R., & Agrawal, R. K. (2013). An introductory study on time series modeling and forecasting. *arXiv preprint arXiv:1302.6613*.

Appendix 1

PIECES OF PYTHON SOURCE CODE

Ploting and Visualizing

```
#import libraries
from math import sin
from matplotlib import pyplot
import matplotlib.pyplot as plt
import pandas as pd
import numpy as np
from math import radians
from sklearn.preprocessing import MinMaxScaler
#set file name parameter and names
#filename = 'data/ibm_us_with_average_R_only.csv'
####filename = 'data/Energy_Consumption_by_Sector_R.csv'
#filename = 'data/CPIAUCSL_seasonally_adjusted_R.csv'
names = ['date', 'mean_value']
#function to create data set
def create_dataset(dataarray, look_back=1): # converting arrays into datasets
  datasetX, datasetY = [], []
  for j in range(len(dataarray)-look_back-1):
       a = dataarray [j:(j+look_back), 0]
       datasetX.append(a)
       datasetY.append(dataarray[j + look_back, 0])
  return np.array(datasetX), np.array(datasetY)
# function to get and extract train and test sets
def get_data():
  #data_fil
  filename = 'ibm_us_with_average_R_only.csv'
  names = ['date', 'mean_value']
  #split here
  valid_ratio = (20/28)
  length = 5   #(20/4)=5 samples
  dataframe = pd.read_csv(filename, names=names, engine='python', usecols=[1])
  dataset = dataframe.values
  ds = np.array(dataset)
  print((ds.shape()))
  dataset = dataset.astype('float32')
  # here the data is being normalized
  MinMaxScaler = MinMaxScaler(feature_range=(0, 1))
  dataarray = MinMaxScaler.fit_transform(dataset)

  train_size = int(len(dataarray) * valid_ratio)
  test_size = len(dataarray) - train_size
  traindata, testdata = dataarray[0:train_size,:], dataarray[train_size:len(dataarray),:]
 #steps to lookback
  lookback = 1
  dataset= dataarray
  traindataX, traindataY = create_dataset(traindata, lookback)
  testdataX, testdataY = create_dataset(testdata, lookback)

  #traindataX = np.reshape(traindataX, (trainX.shape[0], 1, trainX.shape[1]))
```

```
  testdataX = np.reshape(testdataX, (testdataX.shape[0], 1, testdataX.shape[1]))
  return(traindataX, traindataY, testdataX, testdataY)
```

Seasonality adjustment

```
# creating / de-trending a differenced series
def deseason(dataset_train, interval=1):
        deseason_data = list()
        for i in range(interval, len(dataset_train)):
                value = dataset_train [i] - dataset_train [i - interval]
                deseason_data.append(value)
        return deseason_data

df = pd.read_csv(filename, names=names, engine='python')

X = df['date']
Y = df['mean_value']
data=Y
#ds = df.values
#print (Y)
Z = [i for i in range(len(Y))]

#plot data
pyplot.title("IBM stock over time")
pyplot.xlabel("Time period")
pyplot.ylabel("Value of stock - USD")
pyplot.plot(Y)
pyplot.show()
```

RMSE Vs No. of Epochs from LSTM Time Series Prediction US IBM Stock

```
Data#file_name_l ='_lstm_0_500'
init = 0
endi = 15
stepss = 5 #(init,endi,stepss)
file_name_l ='_lstm_cpi_' + str(init) + '_' + str(endi) + '_' + str(stepss) + ''
from keras.models import Sequential
import pandas as pd
from keras.layers.recurrent import LSTM, GRU
import numpy as np
import matplotlib.pyplot as plt
from keras.layers.core import Dense, Activation, Dropout, Flatten
import time
from sklearn.preprocessing import MinMaxScaler
from keras.callbacks import EarlyStopping
from sklearn.metrics import mean_squared_error
from keras.layers import BatchNormalization
import math
from keras.models import load_model
from datetime import datetime
from sklearn.preprocessing import RobustScaler
from keras import regularizers
import keras
import math
from sklearn.model_selection import GridSearchCV
```

```
from keras.wrappers.scikit_learn import KerasClassifier
from keras import backend as K
import seaborn as sns

trainRMSE = []
testRMSE = []
epochss = []

def post_data(data_id, rmse_train, rmse_test, tim):
  import json
  data = { }
  with open('output_data/rmse.json') as json_file:
    data = json.load(json_file)
    if data_id in (data['data_id']):
      return True
    else:
      data['time'].append(tim)
      data['rmse_test'].append(rmse_test)
      data['rmse_train'].append(rmse_train)
      data['data_id'].append(data_id)
      with open('output_data/rmse.json', 'w') as outfile:
        json.dump(data, outfile)
      return False

def create_dataset(dataarray, look_back=1): # converting arrays into datasets
  datasetX, datasetY = [], []
  for j in range(len(dataarray)-look_back-1):
       a = dataarray [i:(i+look_back), 0]
       datasetX.append(a)
       datasetY.append(dataarray[i + look_back, 0])
  return np.array(datasetX), np.array(datasetY)
# function to get and extract train and test sets

def get_data():
  filename = 'data/ibm_us_with_average_R_only.csv'
  names = ['date', 'mean_value']
  valid_ratio = (20/28)
  length = 5   #(20/4)=5 samples

  dataframe = pd.read_csv(filename, names=names, usecols=[1], engine='python')
  datasetarray = dataframe.values
  datasetarray = datasetarray.astype('float32')
  # normalize the datasetarray
  scaler = MinMaxScaler(feature_range=(0, 1))
  dataset = scaler.fit_transform(datasetarray)

  train_size = int(len(dataset) * valid_ratio)
  test_size = len(dataset) - train_size
  traindata, testdata = dataset[0:train_size,:], dataset[train_size:len(dataset),:]

  look_back = 1
  traindataX, traindataY = create_dataset(traindata, look_back)
  testdataX, testdataY = create_dataset(testdata, look_back)

  traindataX = np.reshape(traindataX, (traindataX.shape[0], 1, traindataX.shape[1]))
```

```
    testdataX = np.reshape(testdataX, (testdataX.shape[0], 1, testdataX.shape[1]))
    return(traindataX, traindataY, testdataX, testdataY)

def lstm_new_call(NUM, EPOCS):
    filename = 'CPIAUCSL_seasonally_adjusted_R.csv'
    names = ['date', 'mean_value']
    valid_ratio = (20/28)
    length = 5   #(20/4)=5 samples

    datafr = pd.read_csv(filename, names=names, usecols=[1], engine='python')
    dataarray = datafr.values
    dataarray = dataarray.astype('float32')
    # to normalize the dataarray
    MinMaxScaler = MinMaxScaler(feature_range=(0, 1))
    dataset = MinMaxScaler.fit_transform(dataarray)
    #plt.plot(dataarray)
    #plt.show()

    # split into train and test sets
    train_size = int(len(dataset) * valid_ratio)
    test_size = len(dataset) - train_size
    train, test = dataset[0:train_size,:], dataset[train_size:len(dataset),:]

    NB_EPOCH = 250
    BATCH_SIZE = 128
    VERBOSE = 1
    NB_CLASSES = 10 # number of outputs = number of digits
    OPTIMIZER = SGD() # optimizer, explained later in this chapter
    N_HIDDEN = 128
    VALIDATION_SPLIT=0.2 # how much TRAIN is reserved for VALIDATION
    DROPOUT = 0.3
    X_train /= 255
    X_test /= 255
    # class vectors are converted to binary matrix classes
    Y_train = np_utils.to_categorical(y_train, NB_CLASSES)
    Y_test = np_utils.to_categorical(y_test, NB_CLASSES)
    # M_HIDDEN hidden layers 10 outputs
    model = Sequential()
    model.add(Dense(N_HIDDEN, input_shape=(RESHAPED,)))
    model.add(Activation('relu'))
    model.add(Dense(N_HIDDEN))
    model.add(Dropout(DROPOUT))
    model.add(Activation('softmax'))
    model.add(Dense(NB_CLASSES))
    model.summary()
    model.compile(loss='categorical_crossentropy',
    optimizer=OPTIMIZER,
    metrics=['accuracy'])
    history = model.fit(X_train, Y_train, epochs=NB_EPOCH,
    batch_size=BATCH_SIZE, validation_split=VALIDATION_SPLIT,
    verbose=VERBOSE)
    score = model.evaluate(X_test, Y_test, verbose=VERBOSE)

def lstm_call(NUM, EPOCS):
    filename = 'data/ibm_us_with_average_R_only.csv'
```

```
names = ['date', 'mean_value']
valid_ratio = (20/28)
length = 5   #(20/4)=5 samples

dataframe = pd.read_csv(filename, names=names, usecols=[1], engine='python')
dataarray = dataframe.values
dataarray = dataarray.astype('float32')
# the dataarray  is normalized
MinMaxScaler = MinMaxScaler(feature_range=(0, 1))
dataarray = MinMaxScaler.fit_transform(dataarray)
#plt.plot(dataarray)
#plt.show()

# split into train and test sets
train_size = int(len(dataset) * valid_ratio)
test_size = len(dataset) - train_size
traindata, testdata = dataset[0:train_size,:], dataset[train_size:len(dataset),:]
#print(len(traindata), len(testdata))

# reshape into X=t and Y=t+1
lookback = 1
traindataX, traindataY = create_dataset(traindata, lookback)
testdataX, testdataY = create_dataset(testdata, lookback)
##print("train")

# reshape input to be [samples, time steps, features]
traindataX = np.reshape(traindataX, (traindataX.shape[0], 1, traindataX.shape[1]))
testdataX = np.reshape(testdataX, (testdataX.shape[0], 1, testdataX.shape[1]))


# create and fit the LSTM network
#scaler = MinMaxScaler(feature_range=(0, 1))
#look_back = 1
model = Sequential()
model.add(LSTM(4, input_shape=(1, look_back)))
model.add(Dense(1))
model.compile(loss='mean_squared_error', optimizer='SGD')
#print('summary')
#print(model.summary)
t1 = datetime.now()
model.fit(traindataX, traindataY, epochs=EPOCS, batch_size=1, verbose=0)
t2 = datetime.now()

delta = t2 - t1
tim = delta.seconds
model_file = 'output_data/lstm_model_' + str(NUM) + file_name_l + '.h5'
#####model.save('lstm_model.h5')
model.save(model_file)


# using to make predictions
trainPredict = model.predict(traindataX)
testPredict = model.predict(testdataX)
score = model.evaluate(testdataX, testdataY, verbose='VERBOSE')
#print(score)
```

```
    #print(score[0], score[1])
    # this is used to invert the predictions that have been made
    trainPredict = scaler.inverse_transform(trainPredict)
    traindataY = scaler.inverse_transform([traindataY])
    testPredict = scaler.inverse_transform(testPredict)
    testdataY = scaler.inverse_transform([testdataY])
    #np.savetxt('Train3.txt',traindataY)
    #np.savetxt('Test3.txt',testdataY)
    # accuracy using  root mean squared error
    trainScore = math.sqrt(mean_squared_error(traindataY[0], trainPredict[:,0]))
    testScore = math.sqrt(mean_squared_error(testdataY[0], testPredict[:,0]))
    #print('Test Score: %.2f RMSE' % (testScore))
    trainRMSE.append(trainScore)
    testRMSE.append(testScore)
    epochss.append(EPOCS)


    # train predictions are shifted for plotting
    PredictTrainPlotData = np.empty_like(dataset)
    PredictTrainPlotData [:, :] = np.nan
    PredictTrainPlotData [lookback:len(trainPredict)+lookback, :] = trainPredict
    # shift test predictions for plotting
    PredictTestPlotData = np.empty_like(dataset)
    PredictTestPlotData [:, :] = np.nan
    PredictTestPlotData [len(trainPredict)+(lookback*2)+1:len(dataset)-1, :] = testPredict
    # the predictions and the baseline are plotted
    post_data(NUM, trainScore, testScore, tim)
    main_file = 'output_data/Main_' + str(NUM) + file_name_l + '.txt'
    train_file = 'output_data/Train_' + str(NUM) + file_name_l + '.txt'
    test_file = 'output_data/Test_' + str(NUM) + file_name_l + '.txt'
    np.savetxt(main_file,scaler.inverse_transform(dataset))
    np.savetxt(train_file,trainPredictPlot)
    np.savetxt(test_file,testPredictPlot)
    import matplotlib.patches as mpatches
    #plt.legend(handles=[red_patch],loc=1)
    #plt.legend(handles=[blue_patch],loc=2)
    #plt.legend(handles=[green_patch],loc=3)
    #plt.plot(scaler.inverse_transform(dataset),label='Observed Instance', marker='o')
def main_exec(NUM, EPOCS):
    trainX, trainY, testX, testY = get_data()
    time_now1 = datetime.now()
    for i in range(init,endi,stepss):#(0,2,2):
        if (i%50==0):
            print('Percent Complete: %.2f ' % int(i*100/15))
        #lstm_call(NUM, EPOCS)
        lstm_call(NUM, i)
    time_now2 = datetime.now()

    diff = time_now2 - time_now1

    plt.plot(epochss, trainRMSE, label='Train Data Plot', marker='*')
    plt.plot(epochss, testRMSE, label='Test Data Plot', marker='o')
    plt.legend()
    plt.title('RMSE Vs No. of Epochs from LSTM Time Series Prediction US IBM Stock Data')
    plt.xlabel('Number of Epochs')
```

```
    plt.ylabel('Root Mean Squared Error')
    plt.grid(True)
    #fig = plt.figure()
    #ax = fig.add_subplot(111)
    #ax = plt.Subplot()
    #for i,j in enumerate(trainRMSE):
      # ax.annotate(j,(trainRMSE[i],epochss[i]+0.5))

    plt.show()
    print('time:' + str(diff))
    rmse_file = 'output_data/rmse_file_' + str(NUM) + file_name_l + '.txt'
    with open(rmse_file, 'ab') as rmse_file:
      np.savetxt(rmse_file,epochss,newline='\r\n')
      np.savetxt(rmse_file,trainRMSE,newline='\r\n')
      np.savetxt(rmse_file,testRMSE,newline='\r\n')

main_exec(1,1)
```