A Comparison of Recurrent Neural Networks and Variants in Stock Market Prediction

Naftali Ndeapo Indongo (naftali@aims.ac.za)
African Institute for Mathematical Sciences (AIMS)

Supervised by: Professor Ronnie Becker
AIMS South Africa

23 May 2019

Submitted in partial fulfillment of a masters degree at AIMS South Africa

Abstract

Stock market prediction has been a hot topic for the last three decades, attracting many researchers from different fields. Different machine learning models have been developed for predicting stock prices movements. In this study, we implement three such models: Recurrent Neural Networks (RNN), Long Short-Term Memory (LSTM) and Gated Recurrent Units (GRU) to predict daily closing prices of AEX index listed on Euronext Amsterdam. These models were trained using a stock dataset composed of daily closing prices of 8 indices trading on different stock markets, including AEX index. Three accuracy tests: MAE, RMSE and MAPE were used to measure the performance of each model. The results have shown that GRU performed better than RNN and LSTM in terms of predictive accuracy.

Keywords: Stock market prediction, Machine Learning, RNN, LSTM, GRU, Rolling Period, Backpropagation, Indices.

Declaration

I, the undersigned, hereby declare that the work contained in this research project is my original work, and that any work done by others or by myself previously has been acknowledged and referenced accordingly.

Naftali Ndeapo Indongo, 23 May 2019

Contents

List of Figures

List of Tables

1. Introduction

In this chapter, we give an overview of the project. This chapter is divided into four sections. Section 1.1 gives the motivation behind the project, Section 1.2 introduce the research problem to be addressed, Section 1.3 gives the main objective of the project and finally, Section 1.4 presents some related work.

1.1 Motivation

Predicting the stock market has been a topic of interest for financial analysts, portfolio managers, investments bankers and researchers. Stock market prediction is divided into two categories: *Trend prediction model* and *Time series forecasting* (Shobana and Umamakeswari, 2016). In Trend prediction model, the prediction is done by establishing the relationship between different technical variables and stock prices movement while in time series forecasting, the prediction is done by analyzing trends in historical stock data. This paper focusses on time series forecasting. Huge amounts of funds are traded through stock markets all over the world, attracting many investors to buy shares in companies whose stock values are expected to go up (Raghav et al., 2018). Forecasting helps investors to make profits while minimizing their downside risks by predicting stock prices of companies they are invested in.

Forecasting of stock prices is a very difficult task in time series analysis, due the influence of social and economic factors on stock market movement. Thus, causing frequent fluctuations in stock prices (Sneh et al., 2018). Under the *Efficient Market Hypothesis* it is nearly impossible to "beat the market". However, exploiting patterns in financial time series data helps to make an educated estimate about the future price and yield a great profit. The *Efficient Market Hypothesis* (EMH) is an investment theory derived by Malkiel and Fama (1970) which states that the value of the stock reflects the amount of information available.

Stock time series data are highly unpredictable, stochastic, with chaotic behaviours and full of noise. Thus, forecasting such data is subjects to errors. Over the years, various techniques have been designed for predicting financial markets by extracting meaningful information in the data (Abu-Mostafa and Atiya, 1996). These techniques are divided into two types: Fundamental and Technical Analysis. Fundamental analysis deals with the in-depth analysis of the stock market by making use of parameters such as interest rates, various ratios of prices and other economic parameters such as earning reports and industrial production (Kim and Han, 2000). On the other hand technical analysis make use of historical data such as past prices and volumes traded (see Sections 2.1.1 and 2.1.2 for details.) to predict future prices.

In the above-mentioned techniques, predictions are made by making use of linear and non-linear statistical models (see Section 2.1.2). However, these models fail to adapt to changes in stock trends, thus making it difficult to capture patterns in the data. As an alternative to traditional statistical models, complex deep learning techniques such as Artificial Neural Networks (ANN) and Recurrent Neural Networks (RNN) have been used for predicting stock time series data. Neural networks models can easily approximate continuous functions to some degree of accuracy, making it easy to adapt to the non-stationary and dynamic nature of stock data (Michalak and Lipinski, 2005). These models are data-driven and can be able to learn patterns in stock time series. Nowadays, neural networks algorithms are at the heart of intelligent systems used by portfolio managers, investments bankers and stockbrokers to forecast stock movements.

This study focuses on a special type of artificial neural networks called Recurrent Neural Networks (RNN) and its two variants: Long Short-Term Memory (LSTM) and Gated Recurrent Units. The three

models are applied in stock time series forecasting to see which one performs best. The codes used to produce the results in this research project can be found on my github repository [1].

1.2 Problem Statement

In this project, three deep neural network models: RNN, LSTM and GRU are applied to forecast stock time series data. These models were trained on a stock dataset composed of closing prices of 8 indices of stocks listed on different stock markets. The selected indices are AEX index, DAXINDX, CAC40, FTSE100, HNGKNGI, JAPDOWA, NASCOMP, and ATHEX Composite. These indices were used as features to predict the next day stock price of AEX index. Furthermore, three accuracy measures: MAE, RMSE, and MAPE were used to measure the performance of each model.

1.3 Objectives of the study

The main objectives of this project are:

- To compare the performance of three neural network models: RNN, LSTM and GRU in forecasting stock time series data, and
- Analyse the performance of the three models to find out which model performs best.

1.4 Related Work

When it comes to stock market prediction a lot of research has been done. Here are some similar work.

Sneh et al. (2018) developed a deep learning approach where Convolutional Neural Networks (CNN), Long Short Term Memory (LSTM) and a combination of layers of CNN and LSTM (Conv1D-LSTM) were applied to predict the next day stock prices of two companies listed on the National Stock Exchange of India (NSE). These models were trained on historical stock data for a period of 5 years.

In (Kara et al., 2011), two machine learning models based on classification techniques: Support Vector Machines (SVM) and Artificial Neural Networks (ANN) were developed to forecast the direction of stock prices movement of the daily Istanbul Stock Exchange (ISE) National 100 Index. In these models, historical stock data together with a set of technical indicators were used as inputs. Furthermore, the performance of the models was improved by doing comprehensive parameter settings experiments.

Dash and Dash (2016) developed a decision support system using a computational efficient functional link artificial neural network (CEFLANN). In this system, a set of rules based on three classes representing buy, hold and sell were used to generate trading decisions more effectively. This was done by producing a set of variables within the range (0,1) which are then used to track the stock trend and produce the trading decision. Furthermore, other machine learning techniques such as SVM, Naive Bayesian model, K nearest neighbour model (KNN) and Decision Tree (DT) were used as baseline models to evaluate the performance of the proposed model.

[1] https://github.com/naftalindeapo/Research-Project

2. Background

In this chapter, we discuss some background information relevant to the studies in this project. Section 2.1 introduces the concepts related to stock markets and Section 2.2 gives a brief introduction to Machine Learning (ML). Finally, Sections 2.3 to 2.7 describes what Artificial Neural Networks (ANN) are and how they are trained to performed a certain task.

2.1 Stock Market

A *Stock market* [1] is a collection of markets and exchanges where regular trading activities such as buying, selling and issuing of shares of publicly-held companies take place. Stock markets are very sensitive and often influenced by both economic and non-economic factors such as news, political events, investors' choices, firm policies, investors' expectations and psychological behaviours of investors (Tan, 2001). Due to volatility of such factors, it is almost impossible to predict stock prices [2] to the core. There are several terminologies associated with stock markets and below are some of the terms that we will be frequently used in this project:

- **Stock Exchange**: *A stock exchange* is the main institution, organization or association which host a financial market where securities such as bonds, shares, options, features and other commodities are traded. At the stock exchange, prices of securities are governed by the very basic forces of supply and demand.
- **Liquid Stock**: *A liquid stock* is a stock consisting of high trading volumes, whose shares are always available for trading. The more liquid the stock market is the easier it is to find buyers and sellers of different securities.
- **Indices**: A stock market *index* is a collection of stocks whose values are based on the underlying share prices. It is often computed as a weighted average of prices of selected stocks. Indices are mainly used as samples of the stock market.

2.1.1 Stock Market Data. There are different types of data available for stocks. For instance, data up to fine-grained information about a trade for each trading day in a single month (Hellström, 1998). Stock market dataset varies but the most commonly used ones are composed of seven variables: *Time*, *Open*, *Close*, *High*, *Low*, *Adjusted Close* and *Volume*. In this case, *Time* refers to the trading date. *Open* and *Close* refer to the price of the traded share at the beginning and end of the trading day respectively. *High* and *Low* refer to the highest and lowest value of the stock price achieved during a particular trading day. *Adjusted Close* refers to the closing price of the stock on a given day, which is altered to cater for any corporate actions occurring between the closing time of the current trading day and the opening time of the next trading day. Finally, *Volume* refers to the total number of shares traded during a particular trading period. According to Alagidede (2013), there are approximately 252 trading days in a year excluding weekends and public holidays. Stock prices time series data are regarded to be non-stationary, hence very difficult to use in their raw form (Hellström, 1998), and such data need to be preprocessed before analysis (see Section 4.2).

2.1.2 Stock Market Prediction. Stock time series forecasting is one of the most sophisticated problems in finance, attracting many researchers from fields such as economics, mathematics, statistics and computer science. Over the past few decades, a lot of ground breaking research has been published,

[1] Definition adopted from https://www.investopedia.com/terms/s/stockmarket.asp, last accessed on 11 April 2019

[2] A Stock price is the price at which a particular stock can be sold or bought. Stock prices are guided by the basic rule of supply and demand. A stock can also be called a *share* or an *equity*.

which led to the development of various machine learning algorithms for analysing stock prices patterns and predicting stock prices and indices change (Lv et al., 2019). Figure 2.1 demonstrate how machine learning algorithms can be applied to stock data.

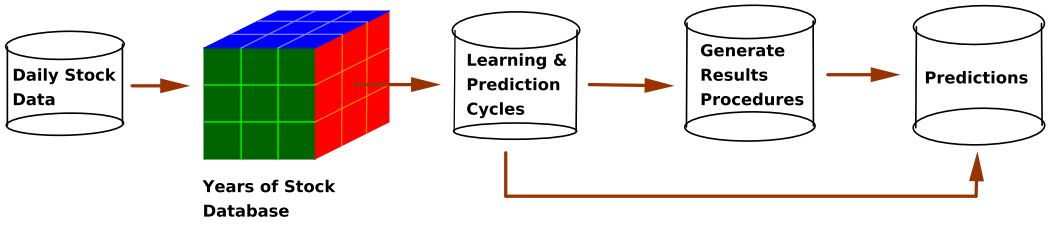


Figure 2.1: Machine learning algorithms to learn from stock data.

There are two methods for predicting stock prices, namely:

(1.) **Fundamental analysis**: *Fundamental analysis* [3] is the method of measuring a security's intrinsic value based on related economic, financial and other qualitative and quantitative factors. Fundamental analysts are concerned more on studying factors such as economy and industry condition that can influence the value of the stock then the actual stock itself. This method is best suited to analyze the long term behavior of the underlying stock.

(2.) **Technical analysis**: *Technical analysis* [4] is the form of time series analysis that is concerned about determining the future price of the underlying stock by analyzing trends in past prices. In this method, predictions are done by using traditional statistical models such as Autoregressive Integrated Moving Average (ARIMA) (Box et al., 2015), Smooth Transition Autoregressive (STAR) (Teräsvirta, 1994) and Autoregressive Conditional Heteroscedastic (ARCH) (Engle, 1982).

Machine learning combines both fundamental and technical analysis methods to see if the developed algorithms can actually learn patterns in stock time series.

2.2 Overview of Machine Learning

In this section, we will introduce Machine Learning (ML) as a field and describe the four types of machine learning algorithms and some of their applications.

We are living in the era of modern technology where a large amount of processed and unprocessed data is one of the resources we have in abundant supply. In the early 20^{th} century Machine Learning (ML) was regarded as a subfield of Artificial Intelligence (AI) which involves developing self-learning and efficient algorithms that are able to transform data into meaningful knowledge and make informed decisions (Raschka, 2015). *Artificial intelligence*, also known as *machine intelligence* is the branch of computer science that involves creating machines that perform tasks which requires a sense of intelligence (Barr and Feigenbaum, 2014). Machine learning is an exciting field attracting many researchers from various fields such as statistics, computer science, robotics, mathematics, medicine, finance and physics; to study algorithms and statistical models to equip machines with much more efficient algorithms for processing information in the data. The word "learning" in machine learning emphasizes on gradually improving

[3] Adopted from https://www.investopedia.com/terms/f/fundamentalanalysis.asp, last accessed on 13 April 2019
[4] Adopted from https://www.investopedia.com/terms/t/technicalanalysis.asp, last accessed on 13 April 2019

the performance of a certain model by constantly updating its parameters. There are four types of machine learning algorithms: Supervised, unsupervised, semi-supervised and reinforcement learning.

2.2.1 Supervised learning. Supervised learning algorithms are based on a mathematical model for mapping a set of input data X consisting of training examples X_i to a desired set of outputs Y with labels Y_i. The main objective is to learn $Y = f(X)$. Supervised learning algorithms are further divided into *regression* where the output variable is continuous and *classification* where the model outputs discrete valuables. Supervised learning algorithms have major applications in voice and facial recognition software, natural language processing and prediction of stock prices movements.

2.2.2 Unsupervised learning. In unsupervised learning, the data consists only of training examples X_i, without any labels. The algorithm explore the structure of the data to extract meaningful information and learn from it by making use of a technique called *clustering* (Raschka, 2015).

2.2.3 Semi-supervised learning. Semi-supervised learning deals with both labelled and unlabelled data. Unsupervised learning algorithms are used to learn the structure of the data, while supervised algorithms are used to predict the labels for unlabelled data. The entire dataset is then used to retrain the supervised model and make predictions on unseen data.

2.2.4 Reinforcement learning. In reinforcement learning, an algorithm learn a model from a series of actions by maximizing its reward when interacting with the environment without any human element (Raschka, 2015). Learning is done through a trial and error method or by planning. Reinforcement learning algorithms have major applications in autonomous vehicles, gaming and robotics.

In this project, we will apply three supervised learning algorithms for regression: Standard RNN, LSTM and GRU to predict stock prices movement.

2.3 Artificial Neural Networks

In this section we will explorer the discovery of Artificial Neural Networks (ANN) and the mathematical concepts used to improve their functionalities in machine learning.

Historically, ANNs emerged out of research in Artificial Intelligence (AI), where the main scientific foundation of artificial neural networks is inspired by biological neurons. The first simplified conceptual model of a brain cell called the *McCulloch-Pitts (MCP)* was developed by a neurologist, Warren McCulloch and a mathematician, Walter Pitts (McCulloch and Pitts, 1943) while trying to understand how the human brain functions. Neurons are defined as a network of nerve cells in the brain whose main function is to process and transmit chemical and electrical signals (Carnevale and Hines, 2006). Each neuron is made up of a cell body, dendrites and an axon. It works as a simple logic gate with binary outputs where pieces of information or signals arriving at the dendrites are then being processed in the cell body. An output signal is generated and passed on to by the axon when the accumulated signals exceeds a certain threshold. Figure 2.2 shows a biological neuron.

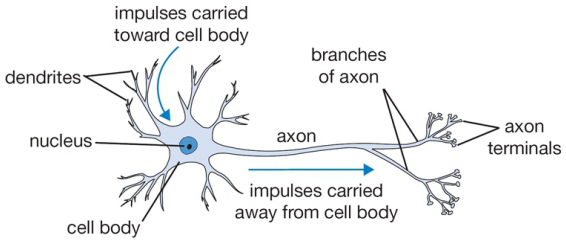


Figure 2.2: Biological neuron with dendrites, cell body and axon adopted from (wiki.tum).

Inspired by the McCulloch-Pitts (MCP) neuron, Frank Rosenblatt discovered the first version of an artificial neuron called the perceptron learning rule (Rosenblatt, 1957). Using the MCP neuron as a foundation, the perceptron learning rule was designed to imitate the functionality of a single neuron in the brain. This learning rule is an algorithm for learning weight coefficients that are multiplied with the input signals in order to decide on whether a certain neuron fires or not. Just like biological neural networks, the ANN is composed of connected artificial neurons called *nodes* or *units*, where each node has a certain number of input and output channels. Each edge is associated with a weight w_i and transfers the signal x_i from one node to another. At each node, all the input x_i are multiplied with their respective weights w_i and summed together. The sum, $\sum w_i x_i$ plus some threshold θ is then passed on an activation function ψ. If the sum exceeds the threshold θ, then the neuron outputs a value a which is either continuous or discrete depending on the nature of the activation function. Thus,

$$y = \sum_{i=1}^{n} w_i x_i + \theta \tag{2.3.1}$$

and

$$a = \psi(y) \tag{2.3.2}$$

where y is the net input, θ is the threshold and $\psi(\cdot)$ is the activation function. Activation functions used in training of ANN will be studied in more details in Section 2.4. Figure 2.3 gives an illustration of a single artificial neuron.

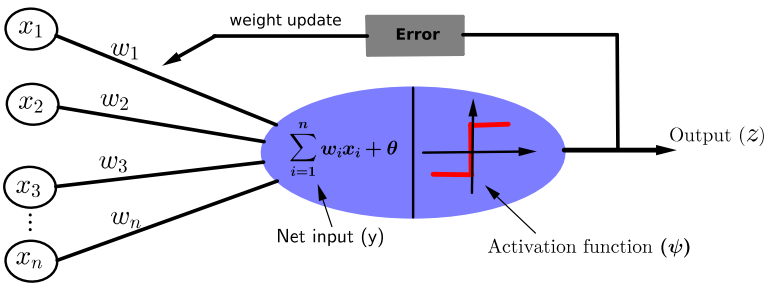


Figure 2.3: Artificial neuron with activation function ψ.

2.4 Activation functions

Activation functions play a major role in the performance of neural network models. They are used to ensure that the input space is correctly mapped to the desired output space together with a series of optimization techniques such as backpropagation and gradient descent (to be discussed in Section 2.7). The commonly used activation functions are: sigmoid, tanh, rectified linear unit (ReLU) and Softmax.

2.4.1 Sigmoid function. This function is a particular case of a logistic function with an "S" shaped curve defined by the formula:

$$\sigma(x) = \frac{1}{1 + e^{-x}} \tag{2.4.1}$$

The sigmoid function inputs real values and output values between 0 and 1. This function can be quite useful in network models for predicting probabilistic outputs. This function is non-linear and continuously

differentiable making it easy to minimize the computational capacity during the training process (Karlik and Olgac, 2011). Besides its advantage, this function also has some disadvantages.

- The sigmoid function causes the problem of vanishing gradients (see Section 3.3.1) when the neuron's activation is close to 0 or 1. If the gradient is close to zero almost no signal flow through the neuron during backpropagation, thus affecting the learning process.
- The output of the sigmoid activation function is always positive, restricting the gradients of the weights w to be all positive or all negative during backpropagation.

2.4.2 Tanh function. The tanh function is a scaled version of the sigmoid function given by the formula:

$$\tanh(x) \;=\; \frac{2}{1+e^{-2x}} - 1 \;=\; 2\sigma(2x) - 1 \tag{2.4.2}$$

The $\tanh$ activation function takes real values and output values in the range $[-1, 1]$. Unlike the sigmoid function, its outputs are zero-centered. It also have a disadvantage of vanishing gradients.

2.4.3 Rectified Linear Unit (ReLU) function. The rectified linear unit (ReLU) activation function was introduced the in paper by Nair and Hinton (2010) and it is one of the most commonly used activation functions for deep learning applications yielding the best results. This function is given by the formula:

$$f(x) \;=\; \max(0, x) \tag{2.4.3}$$

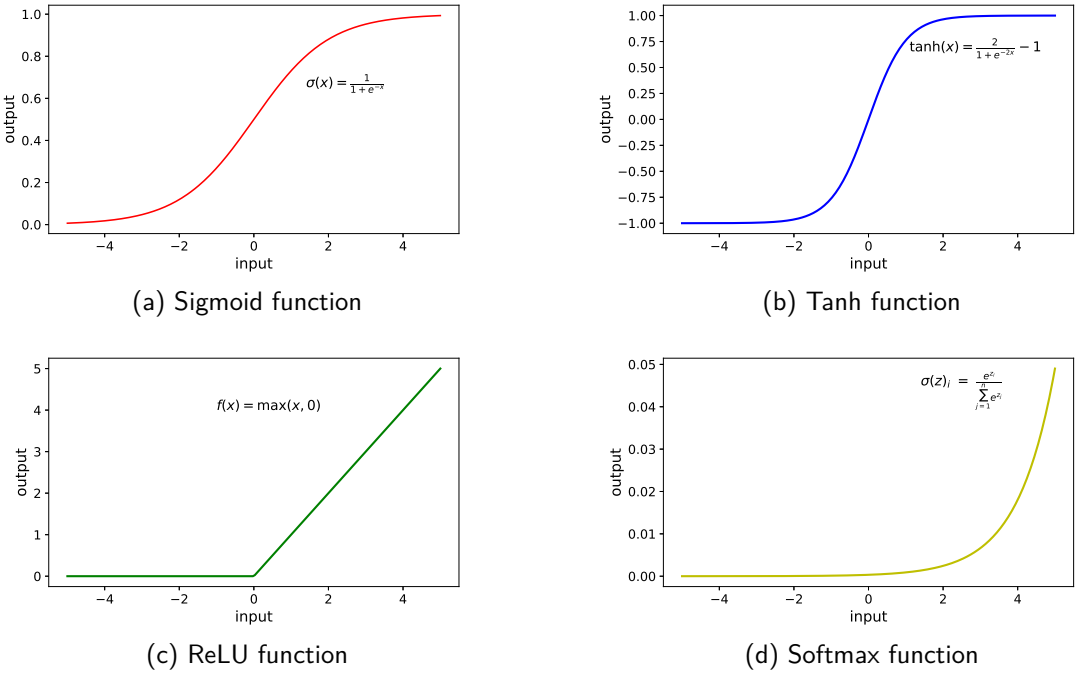
where x is the input to the neuron. The ReLU function has a threshold at zero, thus outputting 0 when the input $x \leq 0$ and x otherwise. So, $\max(0, x) \in [0, +\infty)$. It is computationally cheaper to use the ReLU function than the sigmoid function but unfortunately, the ReLU function also has a vanishing gradient problem. Studies have shown that the linear, non-saturating piece of the ReLU function speeds up the convergence of the stochastic gradient descent (Krizhevsky et al., 2012).

2.4.4 Softmax function. The softmax activation function is a generalized logistic regression function that inputs an n-dimensional vector z of real numbers and output another n-dimensional vector $\sigma(z)$ of real numbers in the interval $(0, 1)$ whose sum add up to 1. The softmax function $\sigma : \mathbb{R}^n \longrightarrow \mathbb{R}^n$ is given by the formula:

$$(\sigma(z))_i \;=\; \frac{e^{z_i}}{\sum_{j=1}^{n} e^{z_j}} \text{ for } i = 1, 2 \ldots, n \text{ and } z = (z_1, z_2 \ldots, z_n) \in \mathbb{R}^n \tag{2.4.4}$$

z_i and z_j are the i^{th} and j^{th} components of the n-dimensional vector z respectively and $\sigma(z)$ is the vector whose components corresponds to the probability of each class (Nwankpa et al., 2018). In the case of multilayer neural networks we use σ^l, z_i^l and z_j^l where the superscript l denotes the activation at the l^{th} layer. Although, using the softmax activation function is proven to be computationally expensive its main advantage is that it gives outputs which are invariant.

Figure 2.4 shows the four commonly used activation functions.



(a) Sigmoid function

(b) Tanh function

(c) ReLU function

(d) Softmax function

Figure 2.4: The four commonly used activation functions.

2.5 Multilayer Neural Network Architecture

Several machine learning models such as Deep Neural Network (DNN) have been successfully used in performing sophisticated tasks such as pattern recognition and image classification. The most popular model in DNN application is the Multilayer Perceptron(MLP). MLP is a class of feed forward ANN which is made up of at least three layers of nodes, an input layer, a hidden layer and an output layer. The complexity of the MLP model is closely dependent on the complexity of the task to be performed. Finding the optimal number of neurons (nodes) and hidden layers for setting up an MLP is still an ongoing research problem (Ramchoun et al., 2016). Figure 2.5 shows a multilayer neural network architecture with 6 layers.

Considering the ANN architecture shown in Figure 2.5, we would like to give a detailed explanation of how the architecture works by giving mathematical expressions to some of the theoretical concepts. We are adopting some of the techniques used in the paper by Higham and Higham (2018). Our MLP consists of 6 layers, each with a specified number of neurons referred to as nodes. At the input layer, each neuron receives an input vector and produces one real value which will be passed on to neurons at the next layer as inputs. The four layers in the middle are called hidden layers, each neuron in every hidden layer receives the same inputs from the previous layer and produces one output which will be passed on to the next layer. Neurons in the 6^{th} layer provide the overall output of the network.

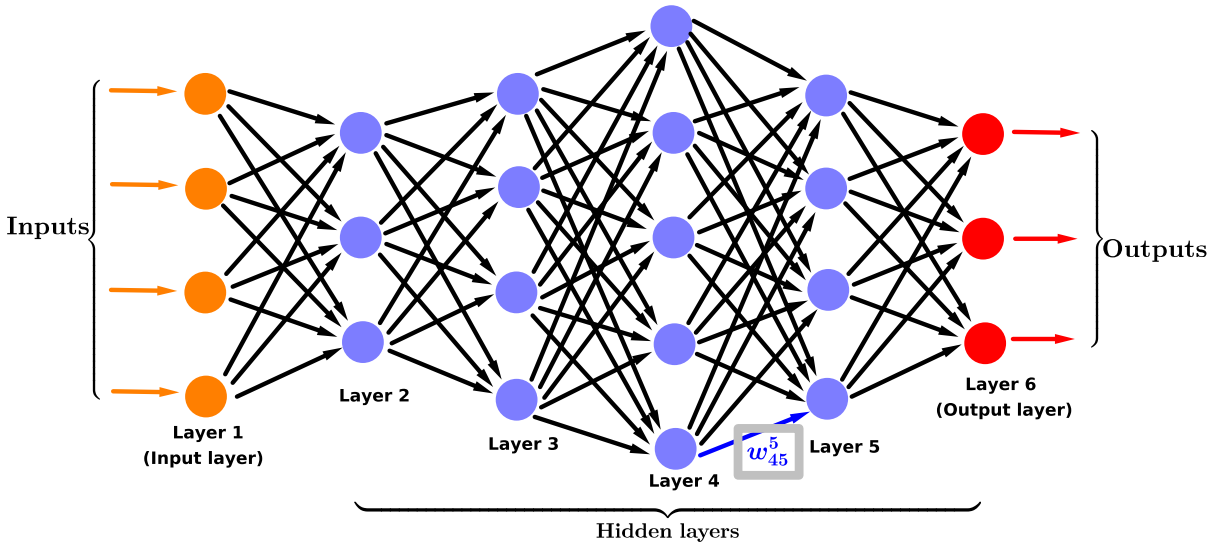


Figure 2.5: A multilayer neural network with six layers.

To extend our understanding to a general network setup, let us consider a network with L layers where the 1^{st} and L^{th} layers are the input and output layers respectively. Let us now consider a general layer l, for $l = 1, 2, \ldots, L$ which contains n_l neurons. So, n_1 and n_L are the dimensions of the input and output data. Intuitively, we can define the overall architecture as a map from $\mathbb{R}^{n_1}$ to $\mathbb{R}^{n_L}$. Let $W^l \in \mathbb{R}^{n_l \times n_{l-1}}$ be the matrix of weights at the l^{th} layer. In particular, W^l is the matrix whose entries are denoted by $w_{i,j}^l$. So, each i^{th} neuron at the l^{th} later applies a weight w_{ij}^l to the output from the $(l-1)^{\text{th}}$ layer. Similarly, $b^l \in \mathbb{R}^{n_l}$ is the vector of biases such that the i^{th} neuron at the l^{th} layer uses the bias b_i^l.

Using Figure 2.5 as an example, we have $L = 6$ layers. So, $n_1 = 4$, $n_2 = 3$, $n_3 = 4$, $n_4 = 5$, $n_5 = 4$ and $n_6 = 3$. In particular, for the edge corresponding to the weight w_{45}^5, the output from the 5^{th} neuron at the 4^{th} layer is weighted by the factor w_{45}^5 when it is fed into the 4^{th} neuron at the 5^{th} layer.

Given an input vector $x \in \mathbb{R}^{n_1}$, to summarise the action of the network, let a_i^l denote the output or activation from the i^{th} neuron at the l^{th} layer, then

$$a^1 \;=\; x \in \mathbb{R}^{n_1} \tag{2.5.1}$$

and

$$a^l \;=\; \psi(W^l a^{l-1} + b^l) \in \mathbb{R}^{n_l}, \quad \text{for } l = 1, 2, \ldots, L \tag{2.5.2}$$

where $\psi(\cdot)$ is the activation function as described in Section 2.4. Equations (2.5.1) and (2.5.2) provides an algorithm for feeding the input forward through the network which leads to the output $a^L \in \mathbb{R}^{n_L}$.

2.6 The Loss Function

The loss function also known as the *cost function*, is a function of weights w and biases b. One of the loss functions used in supervised learning particularly in regression algorithms is the quadratic loss function (MSE) defined as follow: Suppose our training dataset has N pieces of data in $\mathbb{R}^{n_1}$, $\{x^{(i)}\}_{i=1}^N$

which correspond to given target outputs, $\{y(x^{(i)})\}_{i=1}^{N}$ in $\mathbb{R}^{n_L}$ then the loss function is given by

$$C(w,b) \;=\; \frac{1}{2N}\sum_{i=1}^{N}\left(y(x^{(i)}) - a^{L}(x^{(i)})\right)^2 \tag{2.6.1}$$

This function measures how far is the network output a^L from actual target y. During training, we adjust parameters of the network so that the observed output a^L is as closer to the actual target y, thus minimizing the loss function C using various forms of gradient descent (Higham and Higham, 2018).

Neural networks learn by changing their previous outputs through updating their weights from the previous layer depending on the error committed by each neuron. This is achieved using *Backpropagation* algorithm (see Section 2.7.2).

2.7 The Learning Process

The process of training a network whose architecture is described in Figure 2.5 involves choosing the set of weights w and biases b that minimize the loss function given by Equation (2.6.1). This can be achieved by using one of the optimization techniques used in machine learning called *gradient descent.*

2.7.1 Gradient Descent (GD). Let the weights and biases be stored in a vector $p \in \mathbb{R}^n$, where n is the total number of weights and biases. Then the loss function given by Equation (2.6.1) can be written as $C(p)$ such that $C : \mathbb{R}^n \to \mathbb{R}$. Gradient descent works iteratively by computing a sequence of vectors in $\mathbb{R}^n$ in such a way that the respective sequence converges to a vector that minimizes our loss function. Let the current vector be p; we want to choose a small perturbation, Δp such that the next vector, $p + \Delta p$ represents an improvement. Since Δp is very small, by Taylor series expansion we get

$$C(p+\Delta p) \;\approx\; C(p) + \sum_{k=1}^{n}\frac{\partial C(p)}{\partial p_k}\Delta p_k \tag{2.7.1}$$

Where $\frac{\partial C(p)}{\partial p_k}$ denotes the partial derivative of the loss function with respect to the k^{th} parameter. Furthermore, let $\Delta C(p) \in \mathbb{R}^n$ be the column vector of partial derivatives. Thus $\Delta C(p)$ is the gradient vector of C.

$$\left(\Delta C(p)\right)_k \;=\; \frac{\partial C}{\partial p_k} \tag{2.7.2}$$

By substituting Equation (2.7.2) into Equation (2.7.1) we get

$$C(p+\Delta p) \;\approx\; C(p) + \Delta C(p)^T \Delta p \tag{2.7.3}$$

In order to minimize the loss function, we choose Δp such that the term $\Delta C(p)^T \Delta p < 0$. This can be achieved by choosing Δp in the direction of $-\Delta C(p)$. Since Equation (2.7.3) is only relevant for small values of Δp then we choose $\Delta p = \eta \Delta C(p)$ for $\eta > 0$ small. Therefore the update for p is given by

$$p \to p + \eta \Delta C(p) \tag{2.7.4}$$

Where η is called the learning rate. Equation (2.7.4) defines the steepest descent method. Thus, during the training process, we initialize the vector p and iterate with Equation (2.7.4) until the desired number of iterations has been achieved.

Finding the gradient vector at every iteration of the gradient descent given by Equation (2.7.4) can really be computationally expensive when we are having a large number of parameters and training data

points. Alternatively, this can be done efficiently by replacing the mean of individual gradients with the gradient at a single randomly chosen training point. This method is known as *stochastic gradient descent (SGD)*.

2.7.2 Backpropagation Algorithm. Backpropagation (BP) shorthand for "backward propagation of errors" is one of the optimization methods used when training neural networks. BP is used to compute the gradient of the loss function during the learning process. This method was first introduced in the 1970s and it became well known after a paper by Rumelhart et al. (1988). Instead of using the vector p, we will switch back to the entries of the weight matrices and bias vectors. Our aim is to compute partial derivatives, $\partial C/\partial w_{ij}^l$ and $\partial C/\partial b_i^l$ of the cost function with respect to each weight w_{ij}^l and bias b_i^l by using some of the notations from (Higham and Higham, 2018).

We consider a fixed training point. The loss function given by Equation (2.6.1) can be regarded as a function of weights w and biases b. By dropping the dependence of our function on $x^{(i)}$, we will write

$$C = \frac{1}{2}(y - a^L)^2 \tag{2.7.5}$$

From Equation (2.5.2) we have that a^L is the output from the neural network. So, the loss function depends on the weights and biases only through a^L. In order to derive meaningful expressions for computing partial derivatives, we introduce two more sets of variables. Firstly, let

$$z^l = W^l a^{l-1} + b^l \in \mathbb{R}^{n_l}, \quad \text{for } l = 1, 2, \ldots, L. \tag{2.7.6}$$

then Equation (2.5.2) can be written as

$$a^l = \psi(z^l), \quad \text{for } l = 2, 3, \ldots, L. \tag{2.7.7}$$

Equation (2.7.7) is referred to as the fundamental relation that propagates information through the network. Secondly, let $\delta^l \in \mathbb{R}^{n_l}$ to be defined by

$$\delta_i^l = \frac{\partial C}{\partial z_i^l}, \quad \text{for } i = 1, 2, \ldots, n_l \text{ and } l = 2, 3 \ldots, L \tag{2.7.8}$$

where z_i^l is the weighted input of the i^{th} neuron at the l^{th} layer and δ_i^l is called the *error* of the i^{th} neuron at the l^{th} layer. δ_i^l measures how sensitive the loss function is to the weighted input for the i^{th} neuron at the l^{th} layer. Using backpropagation, we will be able to compute a^l for each layer.

Another important notation we will be using when computing the partial derivatives $\frac{\partial C}{\partial b_i}$ and $\frac{\partial C}{\partial w_{ij}}$ is the component-wise multiplication of vectors. Given any two vectors $x, y \in \mathbb{R}^n$, then $x \odot y$ is defined by:

$$(x \odot y)_i = x_i y_i \quad \text{for } i = 1, 2, \ldots, n \tag{2.7.9}$$

This is called the component-wise multiplication. We will use the variables above to derive expressions for the partial derivatives $\frac{\partial C}{\partial b_i}$ and $\frac{\partial C}{\partial w_{ij}}$ as follow. Using Equation (2.7.7) with $l = L$, we get

$$\frac{\partial a_i^L}{\partial z_i^L} = \psi'(z_i^L), \quad \text{for } i = 1, \ldots, n_L \tag{2.7.10}$$

Using Equation (2.6.1) we have that

$$\frac{\partial C}{\partial a_i^L} = \frac{\partial}{\partial a_i^L}\left(\frac{1}{2}\sum_{i=1}^{n_L}(y_k - a_k^L)^2\right) = a_i^L - y_i \tag{2.7.11}$$

Using Equation (2.7.8) and applying Equation (2.7.11) via chain rule, we have

$$\delta_i^L = \frac{\partial C}{\partial z_i^L} = \frac{\partial C}{\partial a_i^L}\frac{\partial a_i^L}{\partial z_i^L} = (a_i^L - y_i)\psi'(z_i^L), \quad \text{for } i = 1, \ldots, n_L \tag{2.7.12}$$

where δ_i^L is the i^{th} component of δ^L, where δ^L is defined by

$$\delta^L = \psi'(z^L) \odot (a^L - y) \tag{2.7.13}$$

To calculate δ_i^l for each hidden layer l, we will use chain rule for multivariate functions to convert from z_i^l to $\{z_j^{l+1}\}_{j=1}^{n_{l+1}}$. We will apply chain rule to Equation (2.7.8) as follow:

$$\delta_i^l = \frac{\partial C}{\partial z_i^l} = \sum_{j=1}^{n_{l+1}} \frac{\partial C}{\partial z_j^{l+1}}\frac{\partial z_j^{l+1}}{\partial z_i^l} = \sum_{j=1}^{n_{l+1}} \delta_j^{l+1}\frac{\partial z_j^{l+1}}{\partial z_i^l} \tag{2.7.14}$$

We know from Equation (2.7.6) that z_j^{l+1} and z_i^l are connect by

$$z_j^{l+1} = \sum_{k=1}^{n_l} w_{jk}^{l+1}\psi(z_k^l) + b_k^{l+1} \tag{2.7.15}$$

Thus,

$$\frac{\partial z_j^{l+1}}{\partial z_i^l} = w_{ji}^{l+1}\psi'(z_i^l) \tag{2.7.16}$$

Therefore,

$$\delta_i^l = \sum_{j=1}^{n_{l+1}} \delta_j^{l+1} w_{ji}^{l+1}\psi'(z_i^l) \tag{2.7.17}$$

Equation (2.7.17) can be rearranged as

$$\delta_i^l = \psi'(z_i^l)((W^{l+1})^T\delta^{l+1})_i \tag{2.7.18}$$

where δ_i^l is the i^{th} component of δ^l, where δ^l is given by

$$\delta^l = \psi'(z_i^l) \circ (W^{l+1})^T\delta^{l+1} \tag{2.7.19}$$

where W is the weight matrix.

To derive an expression for $\frac{\partial C}{\partial b_i^l}$ note that z_i^l is connected to b_i^l by

$$z_i^l = \left(W^l\psi(z^{l-1})\right)_i + b_i^l \tag{2.7.20}$$

Since z^{l-1} does not depend on b_i^l, then

$$\frac{\partial z_i^l}{\partial b_i^l} = 1. \tag{2.7.21}$$

By applying chain rule and applying Equation (2.7.8) we have that

$$\frac{\partial C}{\partial b_i^l} = \frac{\partial C}{\partial z_i^l}\frac{\partial z_i^l}{\partial b_i^l} = \frac{\partial C}{\partial z_i^l} = \delta_i^l, \quad \text{for } i = 1, \ldots, n_l \text{ and } l = 2, \ldots, L. \tag{2.7.22}$$

Therefore, Equation (2.7.22) gives the partial derivative of our loss function with respect to the bias b_i^l.

Finally, the partial derivative of the loss function C with respect to the weight w_{ij} is computed as follow. We will start with the component version of Equation (2.7.6)

$$z_i^l = \sum_{j=1}^{n_{l-1}} w_{ij}^l a_j^{l-1} + b_i^l, \tag{2.7.23}$$

and we have that

$$\frac{\partial z_i^l}{\partial w_{ij}} = a_j^{l-1}, \quad \text{which does not depend on } j \tag{2.7.24}$$

and

$$\frac{\partial z_k^l}{\partial w_{ij}^l} = 0, \quad \text{if } k \neq i \tag{2.7.25}$$

Equations (2.7.24) and (2.7.25) follow directly, because the i^{th} neuron at the l^{th} layer uses the weight w_{ij} only from the i^{th} row of the weight matrix W^l and apply such weights linearly. By applying chain rule and using Equations (2.7.24) and (2.7.8).

$$\frac{\partial C}{\partial w_{ij}} = \sum_{k=1}^{n_l} \frac{\partial C}{\partial z_k^l}\frac{\partial z_k^l}{\partial w_{ij}} = \frac{\partial C}{\partial z_i^l}\frac{\partial z_i^l}{\partial w_{ij}} = \delta_i^l a_i^{l-1} \tag{2.7.26}$$

Therefore, Equation (2.7.26) gives the partial derivative of our loss function C with respect to the weight w_{ij}^l.

The new updates for the weights and biases at each layer are given by the following equations:

$$w_{ij}^{l_{new}} = w_{ij}^{l_{old}} - \eta\frac{\partial C}{\partial w_{ij}^l} = w_{ij}^{l_{old}} - \eta\sum_i a_i^{l-1}\delta_i^l \quad \text{and} \quad b_i^{l_{new}} = b_i^{l_{old}} - \eta\frac{\partial C}{\partial b_i^l} = \eta\sum_i \delta_i^l \tag{2.7.27}$$

where η is the learning rate and it must be carefully chosen to ensure that our optimization problem converges to a global minimum. If η is too small the algorithm might take a long time to converge. On the other hand, large values of η increase the error at each step causing the algorithm to diverge. Therefore, during the training process, we choose initial weight matrix W and bias vector b, and iterate with Equations (2.7.27) until some stopping criterion has been achieved.

2.7.3 Dropout. In neural networks, dropout refers to randomly dropping some neurons in a network during the training process. It is one of the regularization techniques which is used to prevent over-fitting by reducing interdependent learning amongst neurons (Srivastava et al., 2014).

3. Recurrent Neural Networks

In this chapter, we give a brief introduction to RNN and discuss architecture of a simple recurrent neural network, how it is trained using backpropagation through time algorithm and stochastic gradient descent, and discuss the vanishing and exploding gradient problem. Furthermore, we will discuss the two RNN variants, namely: Long Short-Term Memory (LSTM) and Gated Recurrent Units (GRU).

3.1 Introduction

Recurrent neural networks are a special type of artificial neural networks that are designed for modeling of sequential data. Thanks to applications of RNNs algorithms in performing sophisticated tasks such as language translation, image captioning and text generation; we are able to enjoy amazing features such as auto-correct, spell checking, image tagging and facial recognition on our mobile devices. RNNs were discovered in the early 1980s and first published by Rumelhart et al. (1988), but their full potential only came to be realized in the early 1990s. RNNs are defined as special type of artificial neural networks which contain hidden layers composed of recurrently connected neurons. The hidden layers in RNNs preserve the flow of information in adjacent time steps, enabling the network to have the memory of past information (Raschka and Mirjalili, 2017). These hidden layers are commonly known as *memory states*. Unlike traditional ANNs, a hidden layer in a single hidden layer RNN get its input signals from the input layer and the hidden layer from the previous time step.

3.2 RNN Architecture

There are different types of RNN design pattern that can be used for modelling of sequential data. These differs in terms of the type of recurrent architecture they use (Goodfellow et al., 2016). Each RNN architecture is composed of at least an input layer, at least one hidden layer and an output layer. Each hidden layer consists of a self loop known as the *recurrent edge*. The number of hidden layers to use in an RNN are dependent on the complexity of the task at hand, so there is no limit to the number of layers to use. Figure 3.1 shows an unfolded single hidden layer RNN architecture.

Let us consider the standard RNN consisting of an input layer, a single hidden layer and an output layer as shown in Figure 3.1. Supposed the input layer has T input units, so it inputs a sequence of time dependent vectors $(x_0, x_1, \ldots, x_{t-1}, x_t, x_{t+1}, \ldots, x_T)$. Let $I, H, O \in \mathbb{N}$ be the sizes of the input, hidden and output vectors at time-step t respectively. Thus, $x_t \in \mathbb{R}^I$, $h_t \in \mathbb{R}^H$, $y_t \in \mathbb{R}^O$ and $\hat{y}_t \in \mathbb{R}^O$ denotes the input, output, hidden state and corresponding target vectors at time step t respectively. Furthermore, let L be the overall loss and L_t be the loss at each time step t which measures how far each input y_t is from the corresponding training target $\hat{y}_t$. Each directed edge in the RNN architecture is associated with either one of the three weight matrices W^{xh}, W^{hh} and W^{hy}. These weight matrices are independent of time, they are shared across all time steps at each layer. Thus

- The hidden layer is connected to the output layer at each time by the weight matrix $W^{xh} \in \mathbb{R}^{H \times I}$.
- $W^{hh} \in \mathbb{R}^{H \times H}$ is the weight matrix associated with recurrent edge and it connects two successive hidden states.
- The hidden layer is connected to the output layer by the weight matrix $W^{hy} \in \mathbb{R}^{O \times H}$.

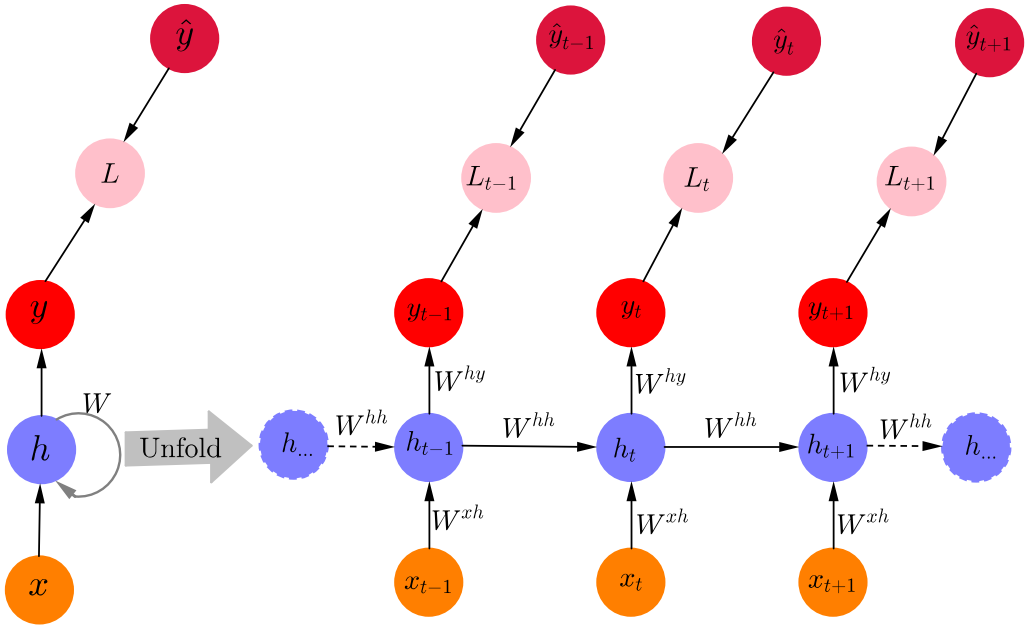


Figure 3.1: An unfolded single hidden layer RNN architecture indicating the loss L of the network, which measures how far is the output y from the corresponding training target $\hat{y}$.

Similar to a standard feedforward NN, activations in an RNN can be computed by applying activation functions to a series of matrices and vectors operations. Thus, the net input of the hidden layer at time step t can be computed using the following formula:

$$z_t = W^{xh}x_t + W^{hh}h_{t-1} + b_h \tag{3.2.1}$$

Therefore, the activation of the hidden units at time step t is given by:

$$h_t = \psi_h(z_t) = \psi_h(W^{xh}x_t + W^{hh}h_{t-1} + b_h) \tag{3.2.2}$$

where b_h is the bias vector of the hidden units and ψ_h is the activation function of the hidden state. After activating the hidden state at each time step t we will activate the output units as follows:

$$y_t = \psi_y(W^{hy}h_t + b_y) \tag{3.2.3}$$

Where b_y is the bias vector of the output units and ψ_y is the activation function of the output layer.

Training RNN is similar to the training of standard ANN, the only difference is that in RNN, training is done across the time axis using *Backpropagation Through Time (BPTT)* which we will discuss in Section 3.3. The main objective is to map our input x to the corresponding output y by choosing suitable parameters for weight matrices W^{xh}, W^{hh} and W^{hy} and biases vectors b_h and b_y such that the overall loss L tends to zero. Since the weights and biases at each layer are shared across all time steps then the gradient at each time step t is dependent on the gradient at the previous time step, $t-1$.

3.3 Back Propagation Through Time

Back propagation through time (BPTT) algorithm is similar to the standard back propagation algorithm discussed in Section 2.7.2, but with some slight adjustments. The algorithm was first derived by Werbos

et al. (1990). BPTT algorithm make use of chain rule across all time steps to compute the gradients of the loss function L with respect to the weight matrices W^{xh}, W^{hh} and W^{hy} and bias vectors b_h and b_y. Since the output units at each time step t in the RNN model given by Equations (3.2.2) and (3.2.3) are defined by nested functions, then the we use the Jacobian matrix to compute the gradients.

The overall loss across all time steps is the sum of all loss functions at time steps $t = 1, 2, \ldots, T$, defined by:

$$L = \frac{1}{T}\sum_{t=1}^{T} L_t \tag{3.3.1}$$

where L_t is the loss function at each time step t, given by:

$$L = \frac{1}{2}(\hat{y}_t - y_t)^2 \tag{3.3.2}$$

where $\hat{y}_t$ is the actual target, y_t is the predicted output and T is the length of the time series. Therefore, the gradient vector for each loss function L_t with respect to the output vector y_t is given by:

$$\frac{\partial L_t}{\partial y_t} = y_t - \hat{y}_t \tag{3.3.3}$$

Let $W \in \{W^{xh}, W^{hh}, W^{hy}, b_h, b_y\}$ then the gradient of the loss function L with respect to W is defined as the sum of the gradients across all time steps. Thus,

$$\frac{\partial L}{\partial W} = \sum_{t=1}^{T} \frac{\partial L_t}{\partial W} \tag{3.3.4}$$

where $\frac{\partial L_t}{\partial W}$ is defined by the chain rule

$$\frac{\partial L_t}{\partial W} = \sum_{k=1}^{t} \frac{\partial L_t}{\partial y_t}\frac{\partial y_t}{\partial h_t}\frac{\partial h_t}{\partial h_k}\frac{\partial h_k}{\partial W} \tag{3.3.5}$$

where $\frac{\partial h_t}{\partial h_k}$ is the partial derivative of h_t with respect to all the hidden units at the previous time steps $k = 1, 2, \ldots, t-1$ defined by:

$$\frac{\partial h_t}{\partial h_k} = \prod_{i=k+1}^{t} \frac{\partial h_i}{\partial h_{i-1}} \tag{3.3.6}$$

where $\frac{\partial h_i}{\partial h_{i-1}}$ is the Jacobian matrix of the hidden units, defined by

$$\frac{\partial h_i}{\partial h_{i-1}} = \begin{bmatrix} \frac{\partial h_i}{\partial h_{i-1,1}} & \cdots & \frac{\partial h_i}{\partial h_{i-1,m}} \end{bmatrix} = \begin{bmatrix} \frac{\partial h_{i,1}}{\partial h_{i-1,1}} & \cdots & \frac{\partial h_{i,1}}{\partial h_{i-1,m}} \\ \vdots & \ddots & \vdots \\ \frac{\partial h_{i,m}}{\partial h_{i-1,1}} & \cdots & \frac{\partial h_{i,m}}{\partial h_{i-1,m}} \end{bmatrix} \tag{3.3.7}$$

Therefore, by substituting Equation (3.3.6) into Equation (3.3.5) then into Equation (3.3.4) we will get,

$$\frac{\partial L}{\partial W} = \sum_{t=1}^{T}\sum_{k=1}^{t} \frac{\partial L_t}{\partial y_t}\frac{\partial y_t}{\partial h_t}\left(\prod_{i=k+1}^{t}\frac{\partial h_i}{\partial h_{i-1}}\right)\frac{\partial h_k}{\partial W} \tag{3.3.8}$$

Each term of the products $\frac{\partial L_t}{\partial y_t}\frac{\partial y_t}{\partial h_t}\frac{\partial h_t}{\partial h_k}\frac{\partial h_k}{\partial W}$ measures how $W \in \{W^{xh}, W^{hh}, W^{hy}, b_h, h_y\}$ at each time step k affect the the loss function at time step $t > k$. The components for which $t >> k$ is called the *long term dependency*, while the components for which $t << k$ is called the *short dependency*.

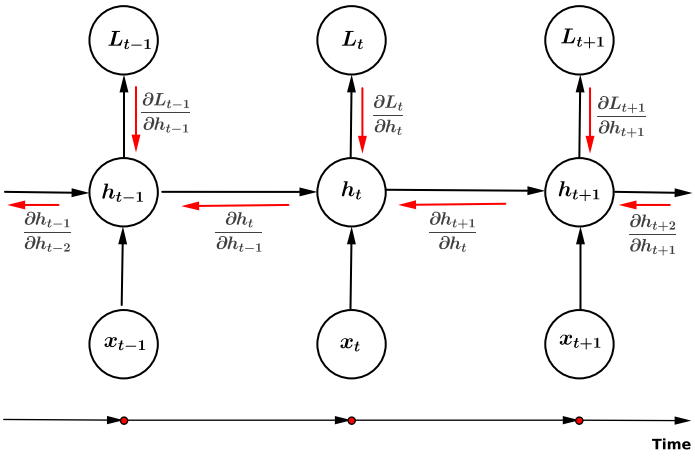


Figure 3.2: RNN unrolling in time by replicating the model for each time step t.

Figure 3.2 shows an illustration of how BPTT is applied to an unrolled recurrent neural network model which is represented as a multi-layer architecture with an unbounded number of layers. As the network receives new inputs over time, the contribution of the inputs at time step $t-1$ vanishes to time step $t+1$. On the other hand, the loss function at time step t with respect to the hidden state h_t in BPTT is greater than the loss function at previous time steps.

3.3.1 Vanishing and Exploding Gradient Problem. The vanishing and exploding gradient problem is the common challenge experienced during the training of standard RNNs, as it was first studied by Bengio et al. (1994). According to Bengio et al. (1994), the *exploding gradient* problem is defined as the exponential increase in the magnitude of the gradients during the training of RNNs. This is caused by explosions of the long term components. On the other hand, the *vanishing gradient* problem is referred to as the reduction in the magnitude of the gradients which limits the number of steps needed for the gradients to propagate the error and properly update the weights and biases. The solution to the vanishing and exploding gradient problem was proposed by Bengio et al. (1994); which involves placing a predefined threshold on the gradient, to prevent it from exploding without changing its direction. This can be achieved using a technique called *gradient clipping* (Mikolov, 2012). Gradient clipping involves resetting the gradients to small numbers every time they reach the specified threshold.

3.3.2 Stochastic Gradient Descent. Stochastic Gradient Descent (SGD) algorithms plays a major role in the training of RNNs. The algorithm has unique properties such as scalability and robustness, and it is proven to be performing really well in many fields with smooth, convex loss functions (Sutskever, 2013). SGD efficiently minimizes the gradients and update the weights at each time step. The weight parameters are updated using the following equation:

$$W_{t+1} = W_t - \frac{\eta}{T} \sum_{t=1}^{T} \frac{\partial L_t(W, x_t, y_t)}{\partial W} \tag{3.3.9}$$

where T is the length of the time series, η is the learning rate, $W \in \{W^{xh}, W^{hh}, W^{hy}, b_h, b_y\}$ and (x_t, y_t) is the pair from the training set. We first initialise the weight and biases with some small random numbers and propagate back in time using BPTT algorithm while constantly updating the weight matrices and bias vectors until some stopping criterion has been attained.

3.4 Long Short-Term Memory Neural Network

Conventional RNN models have been successful in forecasting time series with short-range sequences. However, the models are proven to be difficult to train on time series problems with long-range temporal dependencies (Bengio et al., 1994). As an alternative to shortcomings of conventional RNN, another RNN architecture with the appropriate gradient-based learning algorithm called *Long Short-Term Memory (LSTM)* was developed by Hochreiter and Schmidhuber (1997) and it has been successful in modelling long-range time series sequences. This model was designed to solve the vanishing and exploding gradients problem experienced in standard RNN. Unlike typical RNN, LSTM has the ability to capture autoregressive structure of times series sequences of any length. LSTM contains special units called *memory cells*. Each memory cell consist of a recurrent edge with an associated value called a *cell state*. Figure 3.3 shows the unfolding structure of an LSTM cell.

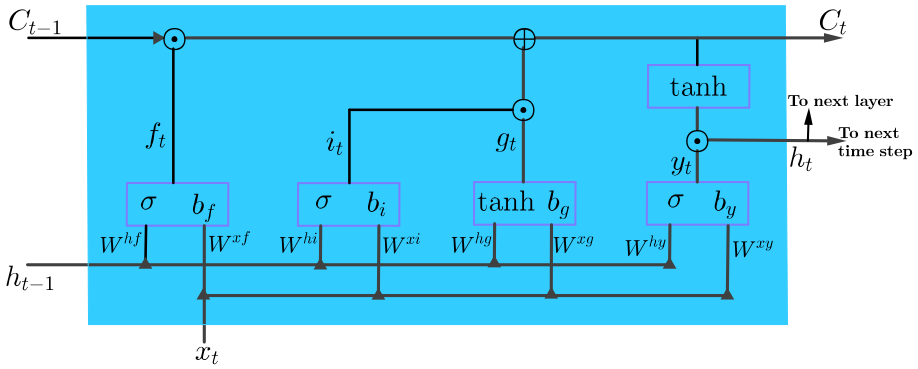


Figure 3.3: Unfolding structure of an LSTM memory cell.

From Figure 3.3, the cell state, C_{t-1} from the previous time step will be updated to get the cell state, C_t at the current time step without directly multiplying it with any weight factor. The two main operation used to control the flow of information in an LSTM cell are: element-wise multiplication (defined by Equation (2.7.10)), $\odot$ and element-wise addition, $\oplus$. Furthermore, x_t and h_{t-1} represents the input vector at time step t and the vector of hidden units at time step $t-1$ respectively. An LSTM cell consists of 3 different types of gate; the forget, input and output gates described as follow:

- **Forget gate**: This gate allows the LSTM memory cell to reset its state by deciding on which information will be kept and which will be forgotten. It outputs values close to 1 for part of the information that need to be kept, and zero for values that need to be neglected. This gate was added to the LSTM cell a few years later to improve the performance of the model (Gers et al., 1999). The forget gate can be computed as follow:

$$f_t = \sigma(W^{xf}x_t + W^{hf}h_{t-1} + b_f) \tag{3.4.1}$$

  where σ is the activation function and b_f is the bias vector.

- **Input gate**: The input gate i_t together with the input node g_t are responsible for controlling the input activation and updating the cell state. The input gate learn conditions for which information should be stored. The input gate can be computed as follow:

$$i_t = \sigma(W^{xi}x_t + W^{hi}h_{t-1} + b_i) \tag{3.4.2}$$

and the input node can be computed using the formula,

$$i_g = \tanh(W^{xg}x_t + W^{hg}h_{t-1} + b_g) \tag{3.4.3}$$

where b_i and b_g are the bias vectors.

After computing the input gate and node, we can then update the cell state at time t as follows:

$$C_t = (C_{t-1} \odot f_t) \oplus (i_t \odot g_t) \tag{3.4.4}$$

- **Output gate**: The output gate y_t updates the values of the hidden units and control the flow of the cell outputs to the rest of the network. The output gate can be computed as follows:

$$y_t = \sigma(W^{xy}x_t + W^{hy}h_{t-1} + b_y) \tag{3.4.5}$$

where b_y is the bias vector of output units. Given the value of the output y_t and cell state at time t, C_t we can then compute the hidden units at time t as follows:

$$h_t = y_t \odot tanh(C_t) \tag{3.4.6}$$

The output gate separates the final memory and the hidden state. It also make decisions on what part of the memory is needed to be exposed to the hidden state.

In a multi-layer LSTM model, its hidden layers are just multiple copies of the cell shown in Figure 3.3 stacked together. These layers work successively to control the flow of information from the input layer to the hidden layer and finally to output layer. Any cell state in the LSTM is multiplied only by the output of the forget gate, which is a value between 0 and 1. So, the forget gate in the LSTM cell deals with both the weights and activation function of the cell state (Raghav et al., 2018). Thus, information from the cell state C_{t-1} at each layer can pass through the memory without being increased or decreased exponentially. This helps the LSTM to solve the exploding and vanishing gradient problem experienced in the conventional RNN model.

3.5 Gated Recurrent Unit

The Gated Recurrent Unit (GRU) is a much more recent RNN variant, proposed by (Cho et al., 2014). Similar to the LSTM, the GRU was developed to efficiently reset or update its memory state depending on input information to the unit (Chung et al., 2015). Each GRU consist of a *reset gate* r_t and an *update gate* z_t which are similar to the forget and output gates of the LSTM memory cell. Figure 3.4 shows the unfolding structure of the GRU.

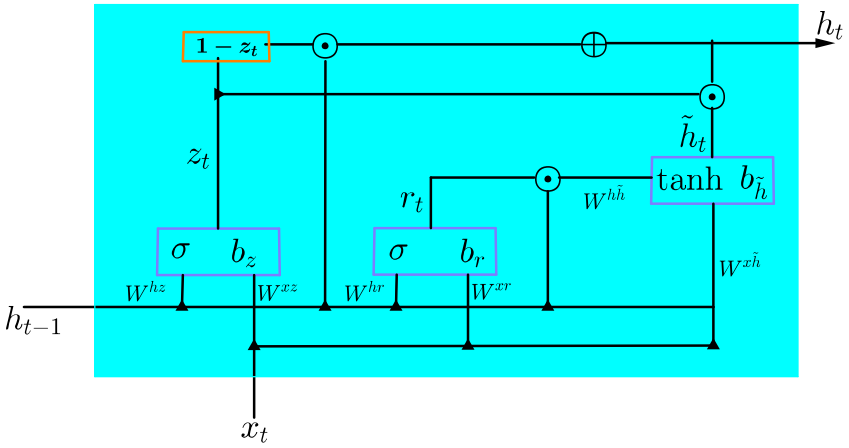


Figure 3.4: Unfolding structure of a Gated Recurrent Unit.

The GRU is similar to an LSTM cell, however it fully exposes its memory state at each time step t and offset the memory state at time t with the memory state at time step $t-1$ by strictly taking integrals of its input, though using the update gate z_t to control its adaptive time constant. The tasks of the reset and update gates are discussed as follow:

- **Update gate**: The update gate z_t controls how much of the information from the previous memory state need to be forgotten and how much information need to be added to the new memory state. The update gate can be computed as follow:

$$z_t = \sigma(W^{xz}x_t + W^{hz}h_{t-1} + b_z) \tag{3.5.1}$$

  where W^{xz} and W^{hz} are the weight matrices of the input and output units, and b_z is the bias vector.

- **Reset gate**: The reset gate r_t models the states of the previous hidden state h_{t-1}, by considering the input units at time t and the hidden state at the previous time step h_{t-1}. The reset gate permits the GRU to only keep the part of previous memory state that is considered to be necessary. It can be computed as follows:

$$r_t = \sigma(W^{xr}x_t + W^{hr}h_{t-1} + b_r) \tag{3.5.2}$$

  where b_r is the bias vector.

Given the reset gate, the new memory state can be computed as follows:

$$\tilde{h}_t = \tanh(W^{x\tilde{h}}x_t + W^{h\tilde{h}}(h_{t-1} \odot r_t) + b_{\tilde{h}}) \tag{3.5.3}$$

Finally, the hidden state of the GRU at time step t can be computed using the following formula:

$$h_t = ((1 - z_t) \odot h_{t-1}) \oplus (z_t \odot h_t) \tag{3.5.4}$$

The mechanism of constantly updating the GRU increases its performance in capturing long term dependencies (Chung et al., 2015). The update gate will be closed to carry the current memory across all time steps every time a previous feature which is considered to be important for future use is detected. The GRU efficiently use the model to capacity by resetting and delete unnecessary information.

4. Data and Methodology

This chapter aims to discuss the stock data used to train the proposed models, as well all the procedures used to obtain the results.

4.1 Data Selection

This section discuss the data samples and input variables chosen for training the three RNN models. The historical stock dataset used in this research project was provided by Prof. Hans George Zimmerman. This dataset is composed of closing prices of 8 different stock indices of industries listed on different stock markets. Each of these indices is considered to be the average of the most influential stocks listed on the underlying stock market. Table 4.1 gives details of the 8 stocks indices used.

Ticker Symbols	Name	Stock Market	Country
AMSTEOE	AEX Index	Euronext amsterdam	Netherlands
DAXINDX	DAX Index	Frankfurt Stock Exchange	Germany
FRCAC40	CAC40 Index	Euronext Paris	France
FTSE100	Financial Times Stock Exchange Index	London Stock Exchange	England
HNGKNGI	Hang Seng Index	Hong Kong Stock Exchange	China
JAPDOWA	Nikkei 225 stock average	Tokyo Stock Exchange	Japan
NASCOMP	NASDAQ Composite	NASDAQ Stock Exchange	USA
GRAGENL	ATHEX Composite	Athens Stock Exchange	Greece

Table 4.1: Selected stock indices.

The stock indices in Table 4.1 are described as follow:

AEX Index: Is a stock market index composed of 25 most frequently traded stocks on the Euronext Amsterdam. This index is made up of shares of 25 largest companies from different sectors of economy in Netherlands.

DAX Index: Is the stock market-weighted index composed of shares of 30 large Germany companies listed on the Frankfurt Stock Exchange. This index measures the average performance of 30 well-established and financially sound Germany companies with billions of euros in market capitalization.

CAC40 Index: is the French benchmark stock index representing the weighted average price of shares of 40 most influential companies listed on the Euronext Paris.

FSTE100: is the stock market index composed of shares of the largest 100 companies listed on the London Stock Exchange with the highest market capitalization.

Hang Seng Index: is the adjusted stock market-weighted index composed of shares of the largest companies trading on the Hong Kong Stock Exchange. The stocks of the Hang Seng index are divided into four sub-indices: Hang Seng Finance, Utilities, Properties, and Commerce and Industries respectively.

Nikkei 225 stock average: is the stock market index representing the weighted-price index of up to 225 largest companies trading on the Tokyo Stock Exchange (Chia et al., 2015).

NASDAQ Composite: Is the stock market index composed of common stocks and similar securities traded on the NASDAQ Stock Exchange. It is one of the three most influential stocks trading in the US stock market together with Dow Jones Industrial Average(DJIA) and Standard and Poor 500 (S&P500).

ATHEX Composite: Is an index composed of 20 largest companies listed on Athens Stock Exchange.

These indices will be used as features to predict the stock price for AEX index as will be described in Section 4.4.

4.2 Data Preprocessing

Financial times series data in its raw form contains a large amount of noise and non-stationary characteristics which will affect the performance of the model (Kara et al., 2011). Feeding the data to the model in its raw state leads to over or under fitting. Therefore, stock market data must be preprocessed before fed into neural networks models. As shown by Asadi et al. (2012), preprocessing plays a major role in the performance of neural network models. The stock data used were preprocessed as follow:

4.2.1 Data Discretization. The selected dataset is reduced to a dataset with only columns which are of importance to our study. Usually, the stock dataset with features described in Section 2.1.1 is reduced to a dataset with only four columns: *Open*,*High*, *Low* and *Close* (OHLC) (Chang et al., 2004), which are then used as features to predict the opening price for the next day. However, in this study all the columns are relevant for predicting the first column.

4.2.2 Data Cleaning. The dataset is analysed for missing values. Unhandled missing values often affect the performance of the model. The paper by Saar-Tsechansky and Provost (2007) presented three techniques for handling missing values: Discard Instances, Acquire missing values and Imputation. The dataset used in this study is already cleared of all missing values.

4.2.3 Data Scaling and Transformation. According to the study by Patro and Sahu (2015), it is computationally efficient to use normalized stock prices than actual stock prices. The most commonly used normalization techniques in time series forecasting are: Mini-Max, Z-score and Decimal Scaling normalizations. In this study, the Min-Max normalization technique was used to transform stock prices to values between 0 and 1 by using the python built-in function MinMaxScaler. The Min-Max scaling technique adopted from (Mustaffa and Yusof, 2011) is defined as follows: Let $\mathcal{D}$ be a dataset whose value at time step t is x_t then

$$\hat{x}_t = \frac{x_t - \min(x_t)}{\max(x_t) - \min(x_t)} \tag{4.2.1}$$

where $\hat{x}$ is the scaled value at time step t, $\min(x_t)$ and $\max(x_t)$ are the minimum and maximum values at time step t respectively. After scaling the stock prices, the scaled values are then transformed into the desired shape and stored into a numpy array of 3 dimensions (T, P, F) where T is the number of training samples, P is the number of time steps (Rolling period) and F is the number of features.

4.3 Data Splitting

After the chosen dataset is cleaned, scaled and transformed into the desired shape, it is then split into training and test sets. During the training process, the training set will further be split into training and validation sets. Training, validation and test sets are defined as follow:

- **Training set:** Is the portion of the input dataset consisting of training examples used for fitting the parameters of the model. The model learn from training examples to predict the output values.

- **Validation set:** Is the portion of the input dataset consisting of examples used for hyperparameters tuning. This set is used to find the optimal parameters of the model.
- **Test set:** The test dataset is different from the training dataset. However, it follows the same probability distribution as the training dataset [1]. Prior to fitting the model to the training set, we then use the test set to predict output values which will be compared to actual stock prices to evaluate the performance of neural network models.

Figure 4.1 demonstrates the training, validation and test split on one of the indices of the selected stock dataset (AEX Index). Although our training, validation and test samples are made up of 8 columns, we used the first column to give a feel of how our training, validation and test targets look like.

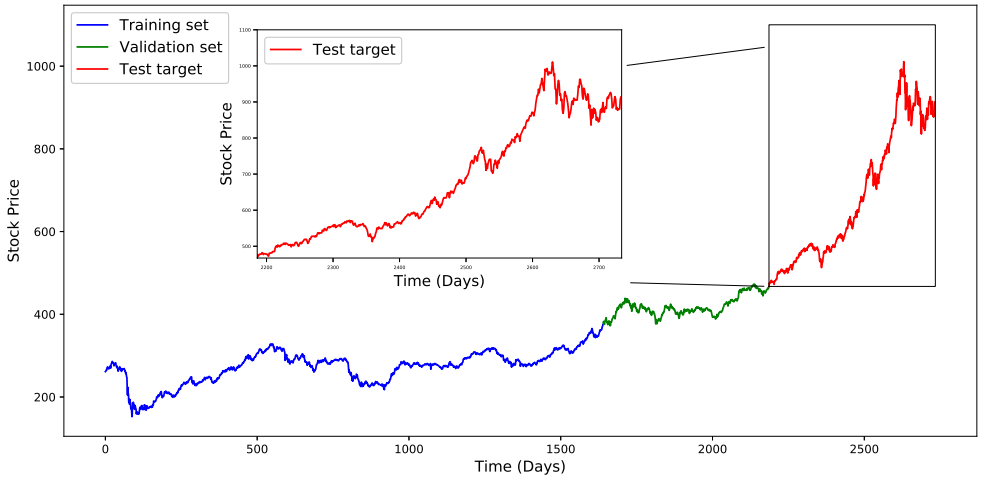


Figure 4.1: Training, Validation and Test sets split.

The dataset of stock prices used in the study was split into 60% training, 20% validation and 20% testing. From Figure 4.1 The red part of the graph is our test target and will be compared to the obtained results.

4.4 Problem Formulation

In this study, 8 stock indices listed on different stock markets were used as input features for the three models: Standard RNN, LSTM and GRU, to predict the stock price of AEX Index, trading on Euronext Amsterdam. Each training sample $x^{(i)} = \{x_{t-(P-1),i}, x_{t-(P-2),i}, \ldots, x_{t,i}\}$ is considered as a set of historical prices over the rolling period of P days. The rolling periods of 5 and 10 days were used to obtain the results. Furthermore, the output value is the stock price of AEX index for the next day defined by $y_{t+1} \in \mathbb{R}$ (predicting 1 time-step ahead). Each input sample is composed of stock prices of the 8 indices at time steps $t-(P-1)$, $t-(P-2)$, $\ldots$, t and the output value is the stock price of AEX index at time step $t+1$. Figure 4.2 demonstrates the structure of the input data.

[1] Adopted from https://en.wikipedia.org/wiki/Training,_validation,_and_test_sets, last accessed on 1 May 2019

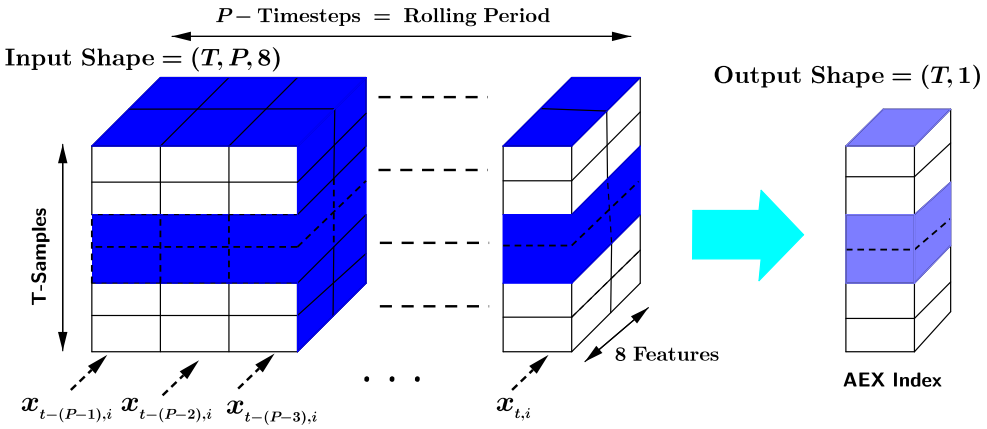


Figure 4.2: Shape of input and output data (T-Samples, P- shape, 8-Features).

4.5 Construction of RNN Models

In this section, we will discuss how different architecture for three models: RNN, LSTM and GRU were implemented in python as well as different parameters of the models used to obtain the results.

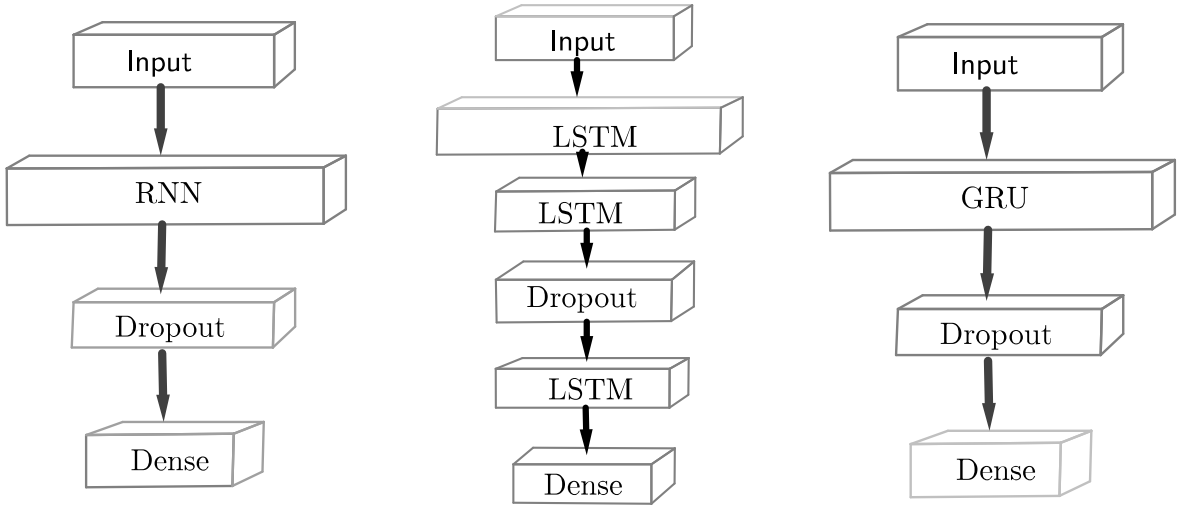
4.5.1 Parameters and architectures of the models. The three RNN models were constructed using python deep learning library called *Keras* [2]. This library provides the efficient way to set neural network models in python. Table 4.2 show the sets of parameters used to train the models.

Model	Parameters	Rolling Period, $P = 5$	Rolling Period, $P = 10$
RNN	Learning rate (η)	0.00133	0.0012
	Layers and input units	RNN layer with 16 units	RNN layer with 16 input units
	Dropout layers	1 Dropout layer, 0.001	No droput layer
	Batch size	300	300
	Number of Epochs	250	350
	Activation	Linear	ReLU
LSTM	Learning rate (η)	0.00183	0.01
	Layers and input units	3 LSTM each with 16 input units	2 LSTM each with 32 input units
	Dropout layers	1 Dropout layer, 0.001	No Dropout layer
	Batch size	300	300
	Number of Epochs	350	250
	Activation	Linear	Linear
GRU	Learning rate (η)	0.00143	0.0114
	Layers and input units	GRU layer with 16 units	GRU layer with 128 unit
	Dropout layers	1 Dropout layer, 0.1811	1 Dropout Layer, 0.1801
	Batch size	300	300
	Number of Epochs	300	250
	Activation	Linear	RELU

Table 4.2: Parameters used for predicting stock price of AEX index with the three RNN models.

[2] See keras documentation at https://keras.io/

Figure 4.3 shows the architectures of RNN, LSTM and GRU models used to obtain the results.



(a) RNN Architecture (b) LSTM Architecture (c) GRU Architecture

Figure 4.3: Layers of the three models used to obtain the outputs.

4.5.2 Stock market prediction algorithm. Algorithm 1 shows the procedures used in stock market prediction using the three models: RNN, LSTM and GRU.

Algorithm 1 RNN, LSTM and GRU stock market prediction algorithm

Input: Historical closing prices of 8 stock indices
Output: Predicted prices for AEX stock index

1. Start
2. Preprocessing of stock data
3. Preprocessed stock data is reshaped into a numpy array of 3 dimensions (T, P, F) where
   - T is the number of training samples.
   - P is the rolling period of 5 and 10 days.
   - F is the number of features for each training sample ($F = 8$).
4. Training, validation and test split.
5. Build each of RNN, LSTM or GRU model in keras with architecture shown in Figure 4.3.
6. Train and validate the model using the training and validation sets (Cross-validation).
7. Use output of the last layer of the model as prediction for the next time step.
8. Repeat steps 6 and 7 until optimal parameters are obtained.
9. Make predictions on the test set.
10. Evaluate the accuracy of each model by using the accuracy tests described in Section 4.7.
11. End

In each of the models, optimizer Adam was used to update the network weights during training due to its high performance and fast convergence compared to other optimizers such as SGD, RMSprop and Adagrad as studied by Reimers and Gurevych (2017).

4.6 Predictions

The three models were fit on the training set composed of scaled values for stock prices of 8 indices described in Table 4.1. The validation set was used to find the optimal parameters for each model. After the validation stage, prediction was performed on the test set.

4.7 Model Performance and Evaluation

Mean Squared Error (MSE) was used to measure the loss during training and validation. Furthermore, the three accuracy measures: Root Mean Squared Error (RMSE), Mean Absolute Error (MAE) and Mean Absolute Percentage Error (MAPE) were used to measure the performance of each model. These accuracy measures are commonly used in time series forecasting (Zhang et al., 2014). These accuracy measures are defined as follow:

Mean Squared Error (MSE) and Root Mean Squared Error (RMSE): MSE used during training and validation is the loss function given by Equation (3.3.2) which measures how far the predicted price is from the actual stock prices at each time step t. RMSE measures the difference between the prices predicted by the model and the actual prices. RMSE is computed as the square root of the overall loss function given by Equation (3.3.1) as defined below.

$$RMSE = \sqrt{\frac{1}{T}\sum_{t=1}^{T}(\hat{y}_t - y_t)^2} \tag{4.7.1}$$

where $\hat{y}_t$ and y_t are the actual target and predicted prices respectively.

Mean Absolute Error (MAE): We use MAE to measure the absolute value of the difference between the predicted and actual stock price. It is calculated as the average of the absolute value of the differences between the predicted and actual prices over the time interval T as shown by the formula below:

$$MAE = \frac{1}{T}\sum_{t=1}^{T}|\hat{y}_t - y_t| \tag{4.7.2}$$

Mean Absolute Percentage Error (MAPE): MAPE is used as the predictive accuracy measure for evaluating and comparing the predictive capability of different neural networks models. Thus

$$MAPE = \frac{1}{T}\sum_{t=1}^{T}\left|\frac{\hat{y}_t - y_t}{y_t}\right| \tag{4.7.3}$$

The accuracy measures were computed using build-in functions from Sklearn library [3]. The accuracy of each of the three RNN models were compared and the lowest value of RMSE, MAE and MAPE indicates the best performing model.

[3] See sklearn documentation at https://scikit-learn.org/stable/

5. Results and Discussion

This chapter is divide into three sections. Section 5.1 presents figures to demonstrate the loss (MSE) committed during training and validation as well as the results obtained from stock market prediction using the three models: RNN, LSTM and GRU. Section 5.2 presents the results obtained from the accuracy measures discussed in Section 4.7. The performance of the three models will be compared to find out which model performs best on the dataset discussed in Section 4.1. Finally, Section 5.3 discusses the results obtained.

The same network inputs for the three models were set, including scaled time series values of the 8 indices: AEX, DAX, CAC40, FTSE100, Hang Seng (HNGKNGI), Nikkei 225 (JAPDOWA) stock average, NASDAQ Composite (NASCOMP) and ATHEX Composite (GRAGENL) and a rolling period (P) of a fixed number of days. Three models were trained using the parameters shown in Table 4.2 to predict one time step ahead.

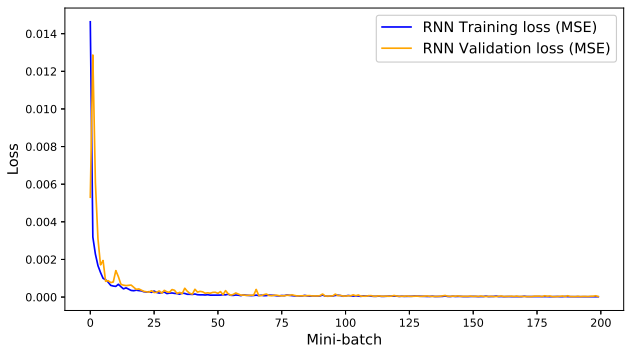
5.1 Visualization of Results

This section present figures to illustrate the actual stock prices of AEX (AMSTEOE) index and the prices predicted by the three models: RNN, LSTM, and GRU. Furthermore, we will present the loss (MSE) committed during training and validation.

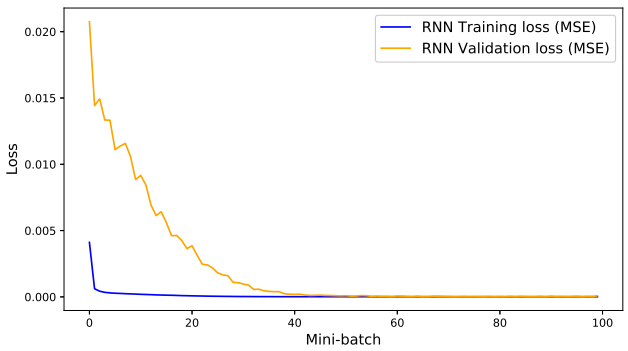
Figures 5.1a, 5.1c and 5.1e shows the training and validation loss (MSE) committed by the three models using a rolling period $P = 5$ days, while Figures 5.1b, 5.1d and 5.1f shows the training and validation loss (MSE) committed by the three models using a rolling period, $P = 10$ days. The comparison of actual prices of AEX index and the prices predicted by the three RNN models are shown in Figure 5.2, where Figures 5.2a, 5.2c and 5.2e show the results predicted using a rolling period $P = 5$ days and Figures 5.2b, 5.2d and 5.2e show the results predicted using a rolling period $P = 10$ days. In addition, Figures 5.2g and 5.2h compare the average of the prices predicted by the three models and the actual stock prices of AEX index.

The actual prices and prices predicted by the three models were plotting on the same figure to see which model performs better. Figures 5.3 and 5.4 show the comparison of the actual stock prices of AEX index and the price predicted by the three models with a rolling period of $P = 5$ and 10 days respectively. The presented results will be discussed in Section 5.3.

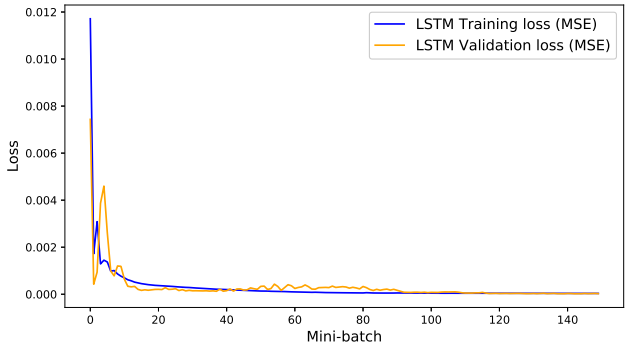
1. **Loss (MSE) committed by the three RNN variants during training and validation.**



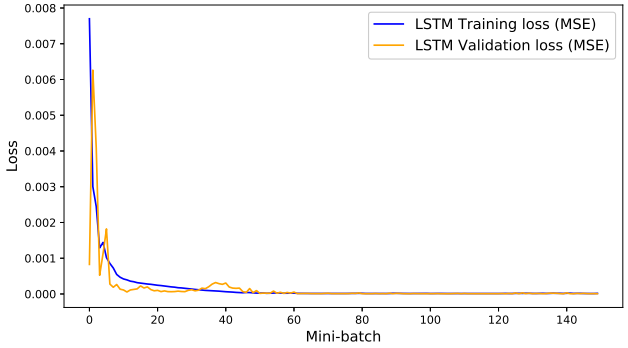
(a) Mini-batch training and validation loss (MSE) committed by RNN model with rolling period, $P = 5$ days.



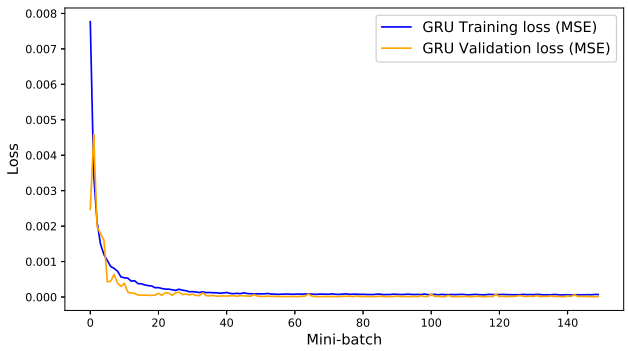
(b) Mini-batch training and validation loss (MSE) committed by RNN model with rolling period, $P = 10$ days.



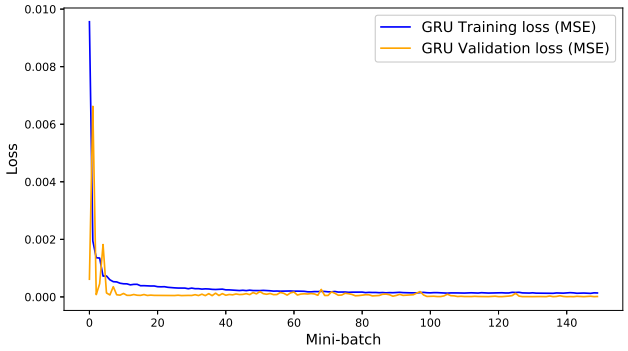
(c) Mini-batch training and validation loss (MSE) committed by LSTM model with rolling period, $P = 5$ days.



(d) Mini-batch training and validation loss (MSE) committed by LSTM model with rolling period, $P = 10$ days.



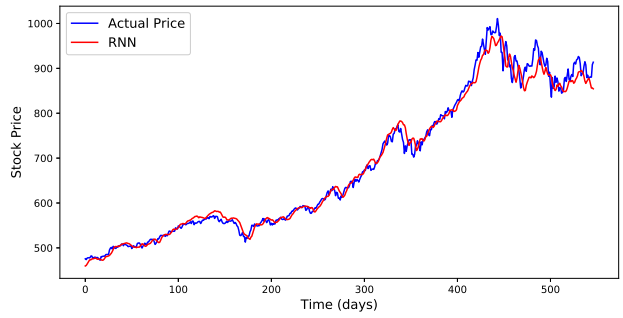
(e) Mini-batch training and validation loss (MSE) committed by GRU model with rolling period, $P = 5$ days.



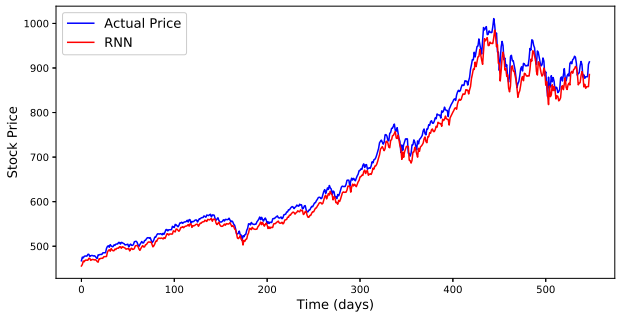
(f) Mini-batch training and validation loss (MSE) committed by GRU model with rolling period, $P = 10$ days.

Figure 5.1: Training and validation loss (MSE) made during forecasting AEX index price using RNN, LSTM and GRU models.

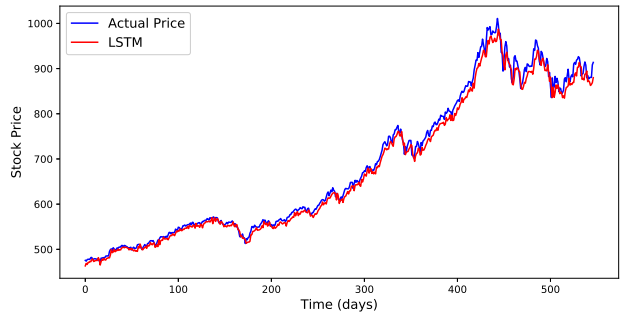
2. **Results for AEX index Prices Predicted by the three RNN variants**



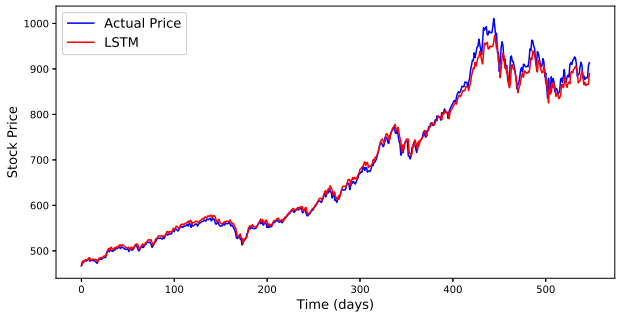
(a) Actual and predicted prices of AEX index using Standard RNN model with $\eta = 0.00183$, $H_s = 16$ and $P = 5$ days



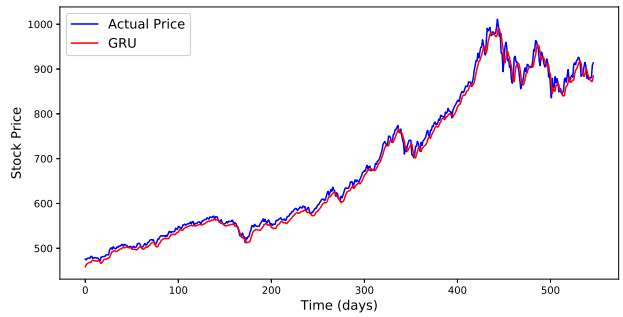
(b) Actual and predicted prices of AEX index using Standard RNN model with $\eta = 0.0012$, $H_s = 16$ and $P = 10$ days



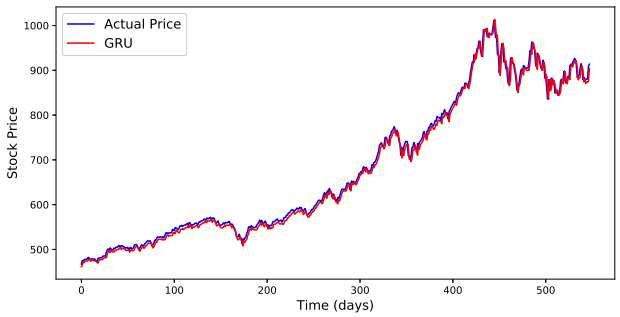
(c) Actual and predicted prices of AEX index using LSTM model with $\eta = 0.0011$, $H_s = 16$ and $P = 5$ days



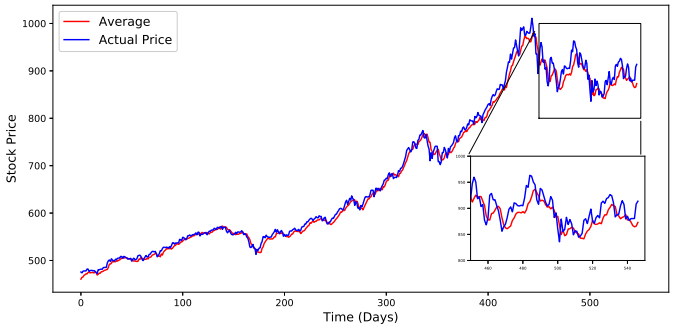
(d) Actual and predicted prices of AEX index using LSTM model with $\eta = 0.011$, $H_s = 32$ and $P = 10$ days



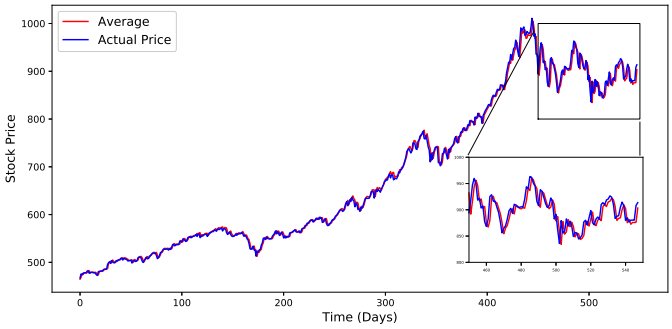
(e) Actual and predicted prices of AEX index using GRU model with $\eta = 0.0143$, $H_s = 16$ and $P = 5$ days



(f) Actual and predicted prices of AEX index using GRU model with $\eta = 0.0114$, $H_s = 128$ and $P = 10$ days



(g) Actual and predicted prices of AEX index using average of the three model with $P = 5$ days



(h) Actual and predicted prices of AEX index using average of the three model with $P = 10$ days

Figure 5.2: Results obtained from forecasting stock prices of AEX index using three RNN models with different parameters for learning rate (η), rolling period (P) and number of hidden states (H_s).

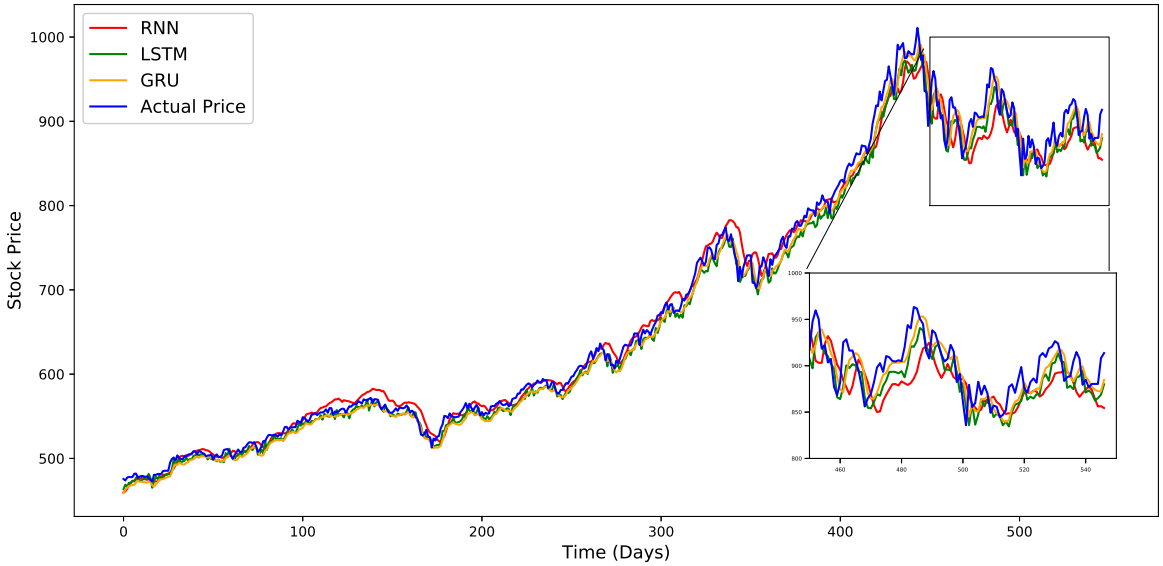


Figure 5.3: A Comparison of actual prices of AEX index and the prices Predicted using RNN, LSTM and GRU with a rolling period $P = 5$ days.

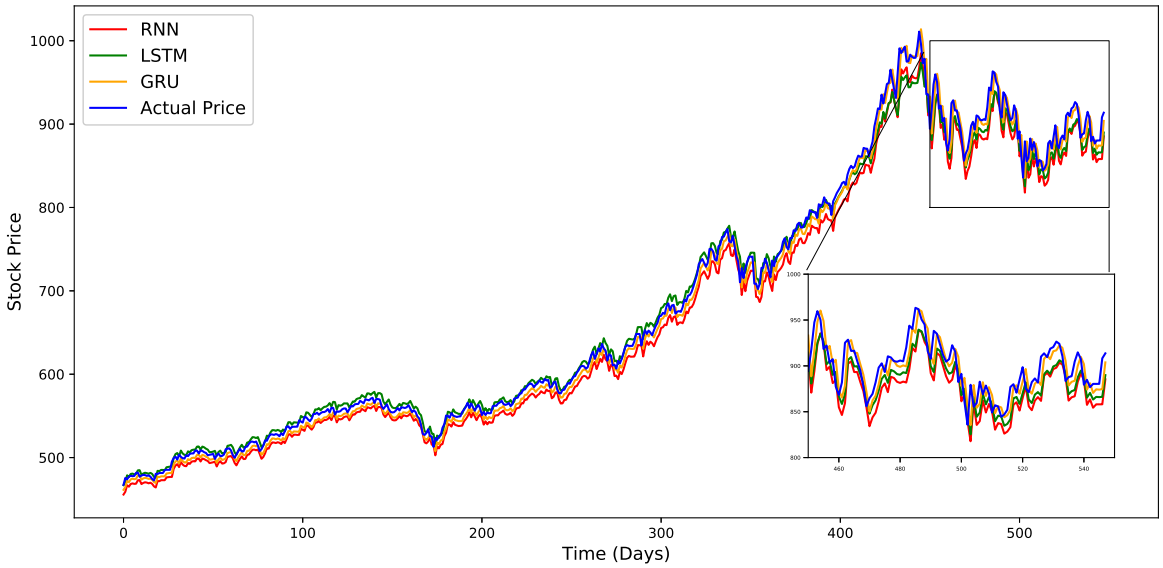


Figure 5.4: A Comparison of actual prices of AEX index and the prices Predicted using RNN, LSTM and GRU with a rolling period $P = 10$ days.

5.2 Predictive Accuracy Test Results

After prediction was done on the test set, both the predicted and test targets values were transformed back to real stock prices using the inverse linear transform function. The accuracy measures: MAE, RMSE and MAPE described by Equations (4.7.1), (4.7.2) and (4.7.3) are computed to evaluate the performance of each model. These were computed using metrics from Sklearn library [3]. MAE, RMSE

and MAPE values for each of the three RNN models with a rolling period $P = 5$ and 10 days are shown in Tables 5.1a and 5.1b.

Model	MAE	RMSE	MAPE
RNN	12.56	18.23	1.70
LSTM	11.40	14.59	1.60
GRU	10.79	13.35	1.58
Average	10.08	13.79	1.34

(a) Models performance with rolling period, $P = 5$ days.

Model	MAE	RMSE	MAPE
RNN	15.99	18.75	2.33
LSTM	9.27	13.30	1.25
GRU	8.38	10.82	1.22
Average	9.13	12.54	1.17

(b) Models performance with rolling period, $P = 10$ days.

Table 5.1: Models performance: MAE, RMSE and MAPE predictive accuracy measures for AEX index.

5.3 Discussion

From the subfigures of Figure 5.1, we observe that initially, the validation loss fluctuates. However, it drops below the training loss after a couple of epochs. As the number epochs increase, training and validation loss (MSE) become constant and equal when each of the three models is fitted on the selected dataset. The prices for the AEX index predicted by the three models: RNN, LSTM and GRU were really close to the actual test targets as shown in Figure 5.2. Although all the models were able to follow the direction and trend of AEX stock index, it is evident that the GRU model is able to capture the dynamics of the time series better than the other two models as shown in Figures 5.3 and 5.4. As the rolling period increases the predictive performance of each model improves. In addition, taking the average of the three models gives better results than the models themselves as shown in Figures 5.2g and 5.2h.

We can deduce that the GRU model performs much better than the RNN and LSTM models in predicting AEX stock index prices. For instance, the MAE, RMSE, MAPE values are 10.79, 13.35 and 1.58 respectively for the rolling period $P = 5$ days, and 8.38, 10.82 and 1.22 for the rolling period $P = 10$ days as shown in Tables 5.1a and 5.1b. Based on the results in these tables, the trained network for all models gives substantial results, as their MAPE values are really low and tolerable. Thus, the lesser the MAPE ratio the better. GRU performs better followed by LSTM then RNN ($1.58 < 1.60 < 1.7$ and $1.22 < 1.25 < 2.33$ for the rolling periods, $P = 5$ and 10 days respectively).

6. Conclusion and Future Work

In this chapter, we will conclude on our research project. This chapter is divided into two sections. Section 6.1 conclude on the findings of our study and Section 6.2 presents some of the future work do be done as an extension of the project.

6.1 Conclusion

In this project, we implemented three deep neural network models: RNN, LSTM and GRU to forecast the closing price of one of the most influential stock indices (AEX index) listed on Euronext Amsterdam one time-step ahead. The models were trained on a dataset composed of closing prices of 8 different indices listed on different stock markets as shown in Section 4.1. The procedures used to obtain the results are as follow: (1.) Data preprocessing for a selected dataset (see Section 4.2), processed data were then used as input to our models; (2.) splitting of the preprocessed data into training, validation and test sets (see Section 4.3); (3.) each model was trained and relevant hyper-parameter tuning was applied using the validation set; (4.) daily stock market prediction is made on the test set using the optimal parameters and a rolling period of 5 or 10 days. After the prediction is done, predicted values were transformed back to original prices and (5.) accuracy tests were performed on each model by using three metrics MAE, RMSE and MAPE.

The performance of the proposed models in terms of accuracy is compared using the error metrics described above. From the results, we observed that GRU has the lowest MAE, RMSE and MAPE values. The reduction in values of MAE, RMSE and MAPE for GRU model compared to RNN and LSTM is an indication that the GRU model can fit the data more accurately. Furthermore, looking at the average of the three models, the values of MAE, RMSE and MAPE obtained are lower than those of the three models. However, the average of the three models is not a good predictive measure since it does not represent all available information. Although the proposed models give satisfactory results, they are still not sufficient for accurate prediction of stock market movement.

6.2 Future Work

The inputs to the model are minimal, so technical indicators such as Moving Average (MA), Exponential Moving Average (EMA) and Converging Divergence (MACD) could also be added as inputs to the models. A more advanced system can be developed to combine these models with classification models such as SVM and ANN. In addition, a trading strategy combining both supervised and unsupervised learning can be implemented for better hyper-parameter selection.

As an extension of the work done in this project, further investigation can be done to improve the performance of the models. In the future, we would like to combine these models with classification models such as SVM and ANN in order to improve the directional accuracy and make precise predictions. Since the stock market is a reflection of human emotions, we can implement a predictive system by exploring the use of text mining, clustering and semi-supervised learning models to predict stock price movements more than one time step ahead.

Acknowledgements

Firstly, I would like to express my heartfelt appreciation to my supervisor Prof. Ronnie Becker for his motivation, patience and expertise. His constant guidance and support helped me throughout the period of this research project. I could not have imagined having a better supervisor for my research project.

A very special gratitude goes out to the African Institute for Mathematical Sciences (AIMS), Stellenbosch University and the Mastercard Foundation for providing the funding for my master studies. A special thanks goes to Uncle Jordan Masikuna for his assistance during the coding phase of my project and Bernard Lebeko for his insightful comments. I would also like to express my gratitude to the entire AIMS family (my extended family away from home) for making my studies both rewarding and memorable.

Lastly, I would like to express my profound gratitude to my parents Malakia Indongo and Josefina Amupadhi, my brothers and sister and my girlfriend for their constant support and encouragement throughout my master studies.

References

Abu-Mostafa, Y. S. and Atiya, A. F. Introduction to financial forecasting. *Applied Intelligence*, 6(3): 205–213, 1996.

Alagidede, P. Month of the year and pre-holiday effects in African stock markets. *South African Journal of Economic and Management Sciences*, 16(1):64–74, 2013.

Asadi, S., Hadavandi, E., Mehmanpazir, F., and Nakhostin, M. M. Hybridization of evolutionary levenberg–marquardt neural networks and data pre-processing for stock market prediction. *Knowledge-Based Systems*, 35:245–258, 2012.

Barr, A. and Feigenbaum, E. A. *The handbook of artificial intelligence*, volume 2. Butterworth-Heinemann, 2014.

Bengio, Y., Simard, P., Frasconi, P., et al. Learning long-term dependencies with gradient descent is difficult. *IEEE transactions on neural networks*, 5(2):157–166, 1994.

Box, G. E., Jenkins, G. M., Reinsel, G. C., and Ljung, G. M. *Time series analysis: forecasting and control*. John Wiley & Sons, 2015.

Carnevale, N. T. and Hines, M. L. *The NEURON book*. Cambridge University Press, 2006.

Chang, P.-C., Wang, Y.-W., and Yang, W.-N. An investigation of the hybrid forecasting models for stock price variation in Taiwan. *Journal of the Chinese Institute of Industrial Engineers*, 21(4):358–368, 2004.

Chia, R., Liew, V., and Wafa, S. The Day-of-the-week effect in the Hang Seng Index. *The Empirical Economics Letters*, 14:107–111, 02 2015.

Cho, K., Van Merriënboer, B., Gulcehre, C., Bahdanau, D., Bougares, F., Schwenk, H., and Bengio, Y. Learning phrase representations using RNN encoder-decoder for statistical machine translation. *arXiv preprint arXiv:1406.1078*, 2014.

Chung, J., Gulcehre, C., Cho, K., and Bengio, Y. Gated Feedback Recurrent Neural Networks. In *International Conference on Machine Learning*, pages 2067–2075, 2015.

Dash, R. and Dash, P. K. A hybrid stock trading framework integrating technical analysis with machine learning techniques. *The Journal of Finance and Data Science*, 2(1):42–57, 2016.

Engle, R. F. Autoregressive conditional heteroscedasticity with estimates of the variance of united kingdom inflation. *Econometrica: Journal of the Econometric Society*, pages 987–1007, 1982.

Gers, F. A., Schmidhuber, J., and Cummins, F. Learning to Forget: Continual prediction with LSTM. *IDSIA*, 1999.

Goodfellow, I., Bengio, Y., and Courville, A. *Deep learning*. MIT Press, 2016.

Hellström, T. *A Random Walk Through the Stock Market*. PhD thesis, Umea Univiversity, 1998.

Higham, C. F. and Higham, D. J. Deep learning: An introduction for applied mathematicians. *arXiv preprint arXiv:1801.05894*, 2018.

Hochreiter, S. and Schmidhuber, J. Long Short-Term Memory. *Neural computation*, 9(8):1735–1780, 1997.

Kara, Y., Boyacioglu, M. A., and Baykan, Ö. K. Predicting direction of stock price index movement using Artificial Neural Networks and Support Vector Machines: The sample of the Istanbul Stock Exchange. *Expert systems with Applications*, 38(5):5311–5319, 2011.

Karlik, B. and Olgac, A. V. Performance analysis of various activation functions in generalized MLP architectures of neural networks. *International Journal of Artificial Intelligence and Expert Systems*, 1(4):111–122, 2011.

Kim, K.-j. and Han, I. Genetic algorithms approach to feature discretization in Artificial Neural Networks for the prediction of stock price index. *Expert systems with Applications*, 19(2):125–132, 2000.

Krizhevsky, A., Sutskever, I., and Hinton, G. E. Imagenet classification with Deep Convolutional Neural Networks. In *Advances in neural information processing systems*, pages 1097–1105, 2012.

Lv, D., Yuan, S., Li, M., and Xiang, Y. An Empirical Study of Machine Learning Algorithms for Stock Daily Trading Strategy. *Mathematical Problems in Engineering*, 2019, 2019.

Malkiel, B. G. and Fama, E. F. Efficient capital markets: A review of theory and empirical work. *The journal of Finance*, 25(2):383–417, 1970.

McCulloch, W. S. and Pitts, W. A logical calculus of the ideas immanent in nervous activity. *The bulletin of mathematical biophysics*, 5(4):115–133, 1943.

Michalak, K. and Lipinski, P. Prediction of high increases in stock prices using neural networks. *Neural Network World*, 15(4):359, 2005.

Mikolov, T. Statistical language models based on neural networks. *Presentation at Google, Mountain View, 2nd April*, 80, 2012.

Mustaffa, Z. and Yusof, Y. A comparison of normalization techniques in predicting Dengue outbreak. In *International Conference on Business and Economics Research*, volume 1, 2011.

Nair, V. and Hinton, G. E. Rectified linear units improve restricted boltzmann machines. In *Proceedings of the 27th international conference on machine learning (ICML-10)*, pages 807–814, 2010.

Nwankpa, C., Ijomah, W., Gachagan, A., and Marshall, S. Activation Functions: Comparison of trends in Practice and Research for Deep Learning. *arXiv preprint arXiv:1811.03378*, 2018.

Patro, S. and Sahu, K. K. Normalization: A preprocessing stage. *arXiv preprint arXiv:1503.06462*, 2015.

Raghav, N., Uttamraj, K. R., Vishal, R., and Lokeswari, Y. V. Stock Price Prediction using Long Short Term Memory. *International Research Journal of Engineering and Technology*, 2018.

Ramchoun, H., Idrissi, M. A. J., Ghanou, Y., and Ettaouil, M. Multilayer Perceptron: Architecture Optimization and Training. *IJIMAI*, 4(1):26–30, 2016.

Raschka, S. *Python machine learning*. Packt Publishing Ltd, 2015.

Raschka, S. and Mirjalili, V. *Python machine learning*. Packt Publishing Ltd, 2017.

Reimers, N. and Gurevych, I. Optimal hyperparameters for deep LSTM-networks for sequence labeling tasks. *arXiv preprint arXiv:1707.06799*, 2017.

Rosenblatt, F. *The perceptron, a perceiving and recognizing automaton Project Para*. Cornell Aeronautical Laboratory, 1957.

Rumelhart, D. E., Hinton, G. E., Williams, R. J., et al. Learning representations by back-propagating errors. *Cognitive modeling*, 5(3):1, 1988.

Saar-Tsechansky, M. and Provost, F. Handling missing values when applying classification models. *Journal of machine learning research*, 8(Jul):1623–1657, 2007.

Shobana, T. and Umamakeswari, A. A Review on Prediction of Stock Market using Various Methods in the Field of Data Mining. *Indian Journal of Science and Technology*, 9, 12 2016. doi: 10.17485/ijst/2016/v9i48/107985.

Sneh, J., Gupta, R., and Moghe, A. Stock Price Prediction on Daily Stock Data using Deep Neural Networks. *Internation Jounal of Neural Networks and Advanced Applications*, 5(ISSN:2313-0563): 45–54, 2018.

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

Sutskever, I. *Training Recurrent Neural Networks*. Phd, University of Toronto, 2013.

Tan, C. N. *Artificial Neural Networks: applications in financial distress prediction & foreign exchange trading*. Wilberto, 2001.

Teräsvirta, T. Specification, estimation, and evaluation of smooth transition autoregressive models. *Journal of the american Statistical association*, 89(425):208–218, 1994.

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

wiki.tum. Artificial Neural Networks. Tum wiki, https://wiki.tum.de, Accessed 14 March 2019.

Zhang, X., Zhang, T., Young, A. A., and Li, X. Applications and comparisons of four time series models in epidemiological surveillance data. *PLoS One*, 9(2):e88075, 2014.