

# **The Effect of News Sentiment and Google Search Indices on the Performance of a Deep Learning Approach to Modelling Bitcoin Prices**

by

PARBHU Varun

Project Supervisor: Professor (Dr) BHURUTH MUDDUN OSK

Project submitted to the Department of Mathematics,

Faculty of Science, University of Mauritius,

as a partial fulfillment of the requirement

for the degree of

**M.Sc. Mathematical and Scientific Computing (Part time)**

# Contents

List of Figures . . . . .	iv
List of Tables . . . . .	v
Acknowledgement . . . . .	vi
Abstract . . . . .	vii
Terms and Definitions . . . . .	viii
<b>1 Introduction</b>	<b>1</b>
<b>2 Deep Learning Networks</b>	<b>4</b>
2.1 The Perceptron . . . . .	5
2.2 Feedforward Network . . . . .	21
2.3 Activation Function . . . . .	28
2.4 Backpropagation . . . . .	30
2.5 Sequential Neural Networks . . . . .	36
<b>3 Optimization</b>	<b>41</b>
3.1 Optimization algorithms . . . . .	43
3.2 Backpropagation using Adam . . . . .	50
<b>4 Sentiment Analysis</b>	<b>52</b>
4.1 Rule-Based Methods . . . . .	53
4.2 Word Embedding . . . . .	54
4.3 Transformer Model (Attention Mechanism) . . . . .	55
<b>5 Analysis of Bitcoin Price using Deep Learning Models</b>	<b>58</b>
5.1 Data Collection . . . . .	58

5.2	Feature Engineering . . . . .	62
5.3	Modelling Bitcoin Price using LSTM NN . . . . .	66
5.4	Results . . . . .	67
<b>6</b>	<b>Conclusion and Future Works</b>	<b>72</b>

# List of Figures

2.1	Rosenblatt perceptron . . . . .	6
2.2	Set <b>A</b> and <b>B</b> with a random separating line . . . . .	7
2.3	Set <b>A</b> and <b>B</b> with a random separating line and its weight vector . . .	8
2.4	Set <b>A</b> and <b>B</b> after adjusting the separating line and its weight vector . . .	9
2.5	AND Function with separating line . . . . .	15
2.6	OR Function with separating line . . . . .	16
2.7	XOR function . . . . .	18
2.8	XOR approximate solution . . . . .	21
2.9	Representation of Truth Table for function $f_1^*$ . . . . .	22
2.10	Representation of Truth Table for function $f_2^*$ . . . . .	23
2.11	FFN hidden layer . . . . .	25
2.12	Representation of the hidden layer . . . . .	26
2.13	Feedforward network for XOR problem . . . . .	27
2.14	Neural Network with $N$ number of layers . . . . .	30
2.15	Consecutive layers, $l$ and $l + 1$ , in a neural network . . . . .	31
2.16	An RNN with a hidden state . . . . .	37
2.17	Computing the hidden state in an LSTM model . . . . .	38
2.18	Architecture of a bidirectional RNN . . . . .	40
3.1	Minimum and Maximum points . . . . .	42
3.2	Network Error over different weight . . . . .	44
4.1	Part-of-speech tagging of sentence at 4.1 using spaCy . . . . .	53
4.2	Transformer Layer (Vaswani, Shazeer, Parmar, Uszkoreit, Jones, Gomez, Kaiser & Polosukhin 2017) . . . . .	56

5.1	Example of raw tweet scraped using the tweepy package through Python	61
5.2	Distribution of tweet count over each day at the end of every hour over 24 hours . . . . .	61
5.3	Bitcoin closing price over every hour for everyday . . . . .	62
5.4	Sentiment score distribution using roBERTa model on all tweets . . .	64
5.5	Sentiment score distribution of Twitter influencers (grouped using KMeans clustering) using roBERTa model on all tweets . . . . .	65
5.6	Feature engineering pipeline . . . . .	66
5.7	Train-test split . . . . .	67
5.8	LSTM architecture with 50 hidden states, input sequence of 25 and 6 input features . . . . .	67
5.9	Model training loss (MSE) and actual training loss in USD (RMSE) over 250 epochs of training . . . . .	68
5.10	Model evolution during training for epochs 1, 5, 25 and 250 . . . . .	69
5.11	Trained model and rolling window forecast for Bitcoin price for next 24 hours (using whole dataset) . . . . .	70
5.12	Trained model and rolling window forecast for Bitcoin price for next 24 hours (using influencers' dataset) . . . . .	70

# List of Tables

2.1	Truth Table for $f_1^*$	22
2.2	Truth Table for $f_2^*$	23
2.3	Truth Table for $f^*$	24
2.4	Truth Table from hidden layer	25
5.1	Summary of error results from model using the complete dataset and the influencers' dataset using the same training and predicting scenario ( $\alpha = 0.00001$ ).	71
5.2	Summary of error results from model using different feature combinations under the same training and predicting scenario ( $\alpha = 0.00001$ ).	71

# Acknowledgement

First and foremost, I would like to show my deepest gratitude to my supervisor Professor (Dr) M. Bhuruth for his guidance and recommendations throughout the project. Also, I would like to thank my family and friends for their constant support during the coursework.

Lastly, I would like to thank my colleagues for their constant support.

# Abstract

Cryptocurrencies have been gaining much attention in the past few years due to their steep increase in value. They are speculative, which fuels the necessity of understanding their behaviour over time. This dissertation studies the principle of deep learning and different sequential neural network architecture and an approach to modelling Bitcoin price over a 24hr period using an LSTM NN is proposed. The Tweepy API is used to get news about Bitcoin from Twitter. We continuously scraped tweets and the total number of tweets hourly, containing 'bitcoin', 'BTC' or '#btc' for 42 days. The roBERTa model is then used to get the sentiment score (positive, neutral, negative) from the cleaned tweets. We then aggregate the scores hourly based on the time the tweets were posted online. Finally, we retrieve the Google search index of the word 'bitcoin' hourly for the same time period using the pytrends API. The hourly tweet counts, Google search index, averaged sentiment of tweets, and the value of Bitcoin in the last hour were used as the input features of the LSTM NN model while the Bitcoin value in the next hour was the target. During experimentation, the combination of hourly tweet count, the hourly average of tweet sentiment, and the value of Bitcoin in the last hour resulted in the lowest training and testing RMSE, \$296.33 and \$189.66, respectively.

# Terms and Definitions

$\alpha$	Learning Rate
$\eta$	Reduced Learning Rate
<b>I</b>	Identity Matrix
$\odot$	Hadamard Product
NN	Neural Network
LSTM	Long Short-Term Memory
MSE	Mean Squared Error
MAE	Mean Absolute Error
NLP	Natural Language Processing

# Chapter 1

## Introduction

A cryptocurrency is a decentralised digital asset distributed over an extensive network of computers. Bitcoin was the first cryptocurrency to be operational in January 2009 after the appearance of the mysterious paper titled "Bitcoin: A Peer-to-Peer Electronic Cash System", published in 2008 under the alias "Satoshi Nakamoto" (a person or group of people) (Nakamoto 2009). The paper described a peer-to-peer payment system using electronic currency (cryptocurrencies) that could be sent directly from one party to another without using a third party (often a financial institution) to validate the transaction. The main idea behind the paper was to emulate an online shared ledger on the peer-to-peer network to validate all transactions via a blockchain, eliminating the risk of forging the register. Hence, the rise of blockchain technology provides security, privacy and a distributed ledger applicable in IoT, distributed storage systems and many more. In exchange for maintaining the blockchain (run and validate), which is energy dependent on running electronic machines, "miners" are rewarded with cryptocurrency.

The value of such a type of currency is highly volatile and speculative. It is different from any other asset on the financial market and thereby creates new possibilities for stakeholders with regard to risk management, portfolio analysis and consumer sentiment analysis (Dyhrberg 2016). Bitcoin had a market cap of over \$ 300B in December 2022. From basically nothing in January 2009 to reaching the highest price (known to date) of \$67,566.83 on November 8, 2021, Bitcoin has been highly volatile compared to other traditional forms of FIAT payment. During that

time, there have been substantial price changes over short periods. In 2017 the value of a single Bitcoin increased from \$863 on January 9, 2017, to a high of \$17,550 on December 11, 2017 (Abraham, Higdon, Nelson & Ibarra 2018). Due to its volatility and never seen behaviour like traditional currencies, Bitcoin (and cryptocurrencies in general) are extremely difficult to predict. Nevertheless, it was found that the best model for Bitcoin price volatility is the AR-CGARCH model, highlighting the significance of including both a short-run and a long-run component of the conditional variance (Katsiampa 2017). Halvor et al. find that the heterogeneous autoregressive model is suitable for Bitcoin volatility whereby trading volume further improves this volatility model (Aalborg, Molnár & de Vries 2019).

In this modern age, social media platforms allow people to share sentiments on a large scale. Hence, prompting research to study the impact of sentiment on different complex problems. Asur and Huberman attempted to solve the revenue prediction of box-office for movies problem using tweet volume and sentiment (Asur & Huberman 2010). The same kind of data and polling results were also used by Bermingham & Smeaton to train a linear regression model to predict election results (Bermingham & Smeaton 2011). Zhang et al. leveraged sentiments on Twitter to help predict the movement of stock market indices such as Dow Jones, S&P500 and NASDAQ (Zhang, Fuehres & Gloor 2011). It was shown that significant negative emotions and opinions caused Dow to go down the next day, and lower negative emotions caused Dow to go up the next day. Consequently, studies are being conducted to correlate the speculative nature of Bitcoin prices and their sentiments on social media. Banerjee et al. studied the impact of cryptocurrency returns and Covid 19-news. A nonlinear technique of transfer entropy was used to investigate the relationship between the top 30 cryptocurrencies by market capitalisation and COVID-19 news sentiment. Results show that COVID-19 news sentiment influences cryptocurrency returns (Banerjee, Akhtaruzzaman, Dionisio, Almeida & Sensoy 2022). Zhang et al. explored the cryptocurrency market's reaction to issuers' Twitter sentiments. It was found that cryptocurrency prices react positively to Twitter sentiments (Zhang & Zhang 2022). In contrast, the trading volume reacts positively to the absolute value of Twitter sentiments in a timely manner (within a

period of 24 h).

This project introduces the essential concepts of deep learning, optimisation, and sentiment analysis. We then follow up with an application in the prediction of Bitcoin's price. Chapter 2 introduces the perceptron, its development into feedforward neural networks and a derivation of the backpropagation algorithm. In Chapter 3, we looked at the different optimisation algorithms and the application of the Adam algorithm in backpropagation. Then, in Chapter 3, we reviewed the traditional and modern sentiment analysis techniques. An application for the prediction of Bitcoin price using data from different sources, sentiment analysis and LSTM NN is studied in Chapter 5. Finally, the project concludes by summarising all our results, findings, and potential improvements.

# Chapter 2

## Deep Learning Networks

Deep learning networks use many hidden layers to learn structure in large datasets, and the critical aspect is that the learning process is not human-engineered. Starting from a simple mathematical model of a neuron (McCulloch & Pitts 1943) and the development of the perceptron (Rosenblatt 1958), deep learning networks were complicated to train. The introduction of the backpropagation algorithm by David Rumelhart overcome the training problem of deep neural networks. Thus, allowing networks to tune their internal parameters iteratively over training datasets. The procedure repeatedly adjusts the weights of the connections in the network to minimise a measure of the difference between the actual output vector of the net, and the desired output vector (Rumelhart, Hinton & Williams 1986). Using a general-purpose learning procedures, deep learning networks can find non-linear relationships within high-dimensional data. Kunihiko Fukushima, Yann LeCun, Geoffrey Hinton and Yoshua Bengio contributed to creating more prosperous and more complex multilayered neural networks to tackle different tasks in processing images, video, speech, audio and text. With features surpassing conventional machine-learning techniques, access to high-end graphic processors and computing power, and the rise of Big Data, deep learning networks are being applied to many domains of science, business and government (LeCun, Bengio & Hinton 2015). Hence, tasks such as image recognition, speech recognition, natural language processing, sentiment analysis, fraud detection, etc., can be tackled with promising results.

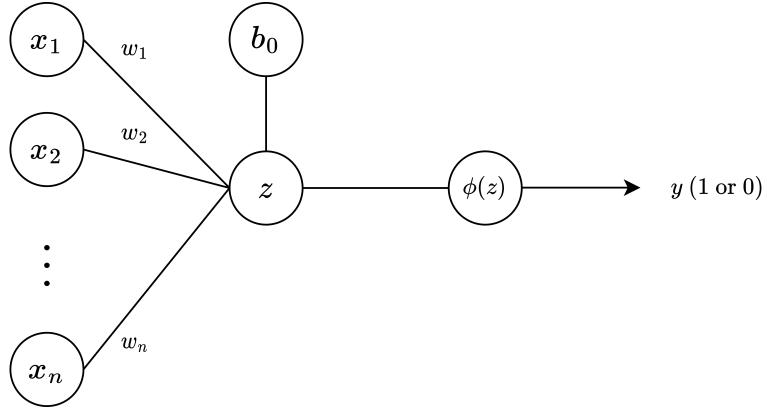
We introduce the perceptron to classify linearly separable datasets and a proof of

the perceptron convergence theorem. The solution and limitations of the perceptron for logic functions are explored before introducing feedforward networks. We also explore some standard activation functions used in feedforward networks. We then introduce and derive the backpropagation algorithm of feedforward networks with multiple hidden layers and introduce different activation functions. Finally, we define the RNN, LSTM and BRNN neural network architecture.

## 2.1 The Perceptron

In 1943, McCulloch and Walter Pitts demonstrated that simple binary threshold units wired up as logical gates could be used to build a digital computer (McCulloch & Pitts 1943). The McCulloch-Pitts (MCP) neuron is a simple mathematical model of a biological neuron. It was the earliest mathematical model of a neural network and had only three types of weights; excitatory (1), inhibitory (-1) and inactive (0). The model had an activation function which had a value of 1 if the weighted sum was greater or equal to a given threshold, else 0. Using the MCP neuron, one of the first digital computers that contained stored programs was built. However, the MCP neuron was very restrictive.

The perceptron algorithm proposed by Rosenblatt (Rosenblatt 1958) is motivated by and overcomes some limitations of the MCP Neuron. For example, the input was not restricted to boolean values but expanded to real numbers. Rosenblatt proved that if the data used to train the perceptron are linearly separable classes, then the perceptron algorithm converges and separates the two classes by a hyperplane.



**Figure 2.1:** Rosenblatt perceptron

Let  $\mathbf{x}$  be a vector of inputs where each  $x_i \in \mathbb{Z}$  and  $\mathbf{w}$  be a vector of weights corresponding to the input signals where each  $w_i \in \mathbb{Z}$ .

$$\mathbf{x} = [x_0, x_1, \dots, x_n]^T \quad \text{and} \quad \mathbf{w} = [w_0, w_1, \dots, w_n]^T$$

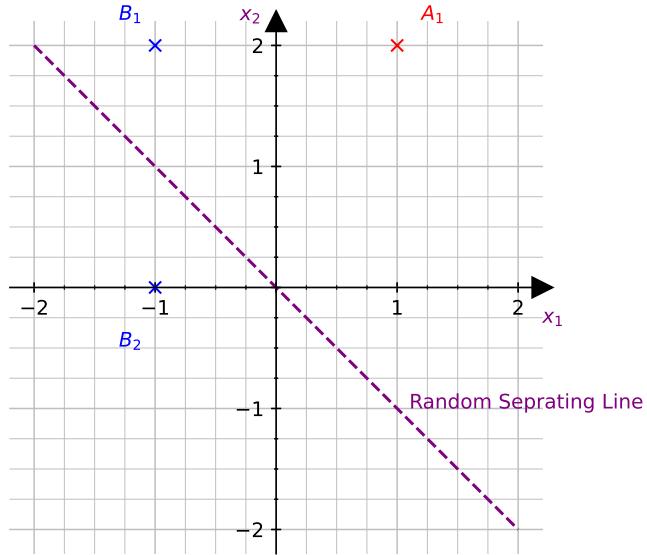
Then, the mathematical definition of the perceptron is given by

$$\phi(z) = \begin{cases} 1 & \text{if } \mathbf{w}^T \mathbf{x} + b \geq 0 \\ 0 & \text{otherwise} \end{cases} \quad (2.1)$$

where  $\phi(z)$  is known as the hard delimiter.

Consider sets in  $\mathbb{R}^2$  given by  $\mathbf{A} = \{a_1\}$  and  $\mathbf{B} = \{b_1, b_2\}$  where  $a_1 = (1, 2)^T$ ,  $b_1 = (-1, 2)^T$  and  $b_2 = (0, -1)^T$ . The two sets are linearly separable in  $\mathbb{R}^2$  as shown in Figure 2.2. There are an infinite number of separating hyperplanes. One separating hyperplane through the origin is the line  $x_1 + x_2 = 0$ . The normal vector to this line is given by  $w = (1, 1)$  and this line points in the direction of  $A$ .

$$\mathbf{A} = \left\{ \begin{bmatrix} 1 \\ 2 \end{bmatrix} \right\} \quad \mathbf{B} = \left\{ \begin{bmatrix} -1 \\ 2 \end{bmatrix}, \begin{bmatrix} 0 \\ -1 \end{bmatrix} \right\} \quad (2.2)$$



**Figure 2.2:** Set **A** and **B** with a random separating line

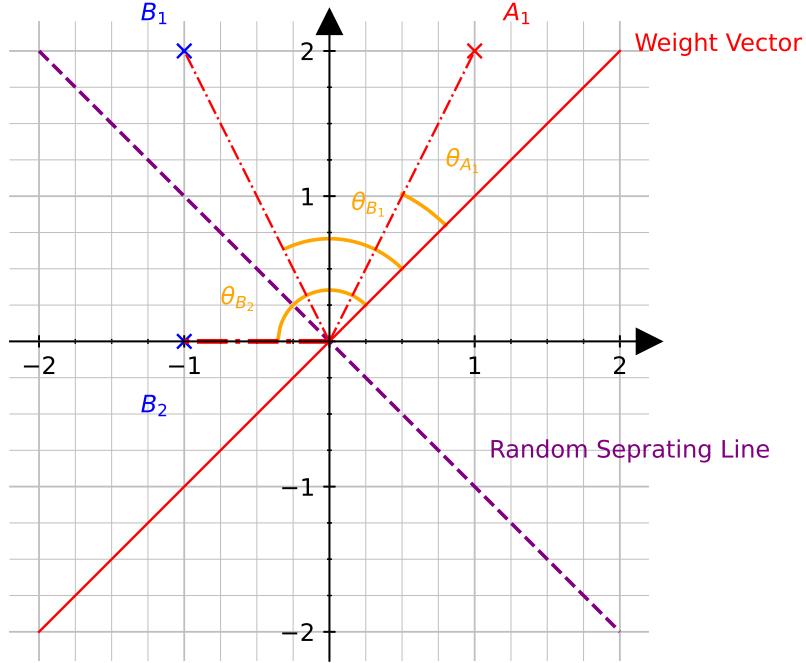
Comparing to the equation (2.1); we can deduce that

$$w_1 = 1 \quad w_2 = 1 \quad (2.3)$$

Visually, we can already see that the separating line does not split the points properly. The computation of the binary classification using the random line is given by

$$\begin{aligned} \phi(A_1) &= \phi \left( \begin{bmatrix} 1 \\ 1 \end{bmatrix} \cdot \begin{bmatrix} 1 \\ 2 \end{bmatrix} \right) = \phi(3) = 1 \\ \phi(B_1) &= f \left( \begin{bmatrix} 1 \\ 1 \end{bmatrix} \cdot \begin{bmatrix} -1 \\ 2 \end{bmatrix} \right) = \phi(1) = 1 \\ \phi(B_2) &= f \left( \begin{bmatrix} 1 \\ 1 \end{bmatrix} \cdot \begin{bmatrix} -1 \\ 0 \end{bmatrix} \right) = \phi(-1) = 0 \end{aligned}$$

The computation confirms the visual representation and groups **A**<sub>1</sub> and **B**<sub>1</sub> together and **B**<sub>2</sub> separated which is incorrect. We can either alter the separating line (by moving the weight vector) with respect to **A**<sub>1</sub> and/or **B**<sub>1</sub>.



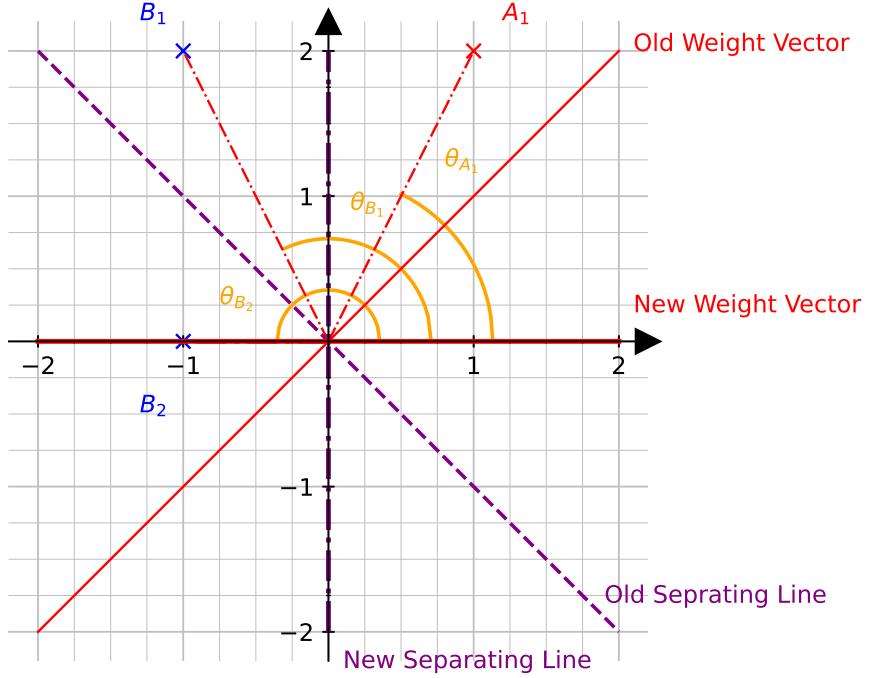
**Figure 2.3:** Set **A** and **B** with a random separating line and its weight vector

Since **B**<sub>1</sub> is incorrectly classified, using the property of subtraction of vectors; we move the weight vector (**w**) away from **B**<sub>1</sub> and check the classification again.

$$\mathbf{w}^{(1)} = \mathbf{w} - \mathbf{B}_1 \quad (2.4)$$

$$= \begin{bmatrix} 1 \\ 1 \end{bmatrix} - \begin{bmatrix} -1 \\ 2 \end{bmatrix} \quad (2.5)$$

$$= \begin{bmatrix} 2 \\ -1 \end{bmatrix} \quad (2.6)$$



**Figure 2.4:** Set **A** and **B** after adjusting the separating line and its weight vector

Based on the new separating line, we can observe visually that the points are correctly classified. The computation of the binary classification using the new separating line  $\mathbf{w}^{(1)}$  is given by

$$\begin{aligned}\phi(A_1) &= \phi \left( \begin{bmatrix} 2 \\ -1 \end{bmatrix} \cdot \begin{bmatrix} 1 \\ 2 \end{bmatrix} \right) = \phi(0) = 1 \\ \phi(B_1) &= \phi \left( \begin{bmatrix} 2 \\ -1 \end{bmatrix} \cdot \begin{bmatrix} -1 \\ 2 \end{bmatrix} \right) = \phi(-4) = 0 \\ \phi(B_2) &= \phi \left( \begin{bmatrix} 2 \\ -1 \end{bmatrix} \cdot \begin{bmatrix} -1 \\ 0 \end{bmatrix} \right) = \phi(-2) = 0\end{aligned}$$

Thus, the perceptron learned how to classify the two sets of points. The function  $f(\mathbf{x})$  could be re-used to classify new points added to the sets.

## Perceptron Algorithm

The perceptron algorithm adjusts the weights and as a result, the hard delimiter as well in order to linearly separate a set of binary labelled input.

---

### Algorithm 1 Perceptron Algorithm

---

**Require:**

- 1: Let  $t = 0$  and  $\mathbf{w} = [0 \ 0 \ \dots \ 0]^T$
  - 2: Consider the training set  $D$  s.t  $D = C_1 \cup C_2$
  - 3:  $C_1 \leftarrow$  input with label 1
  - 4:  $C_2 \leftarrow$  input with label 0
  - 5: **Start**
  - 6: **while** !convergence **do**
  - 7:     Select a random input  $\mathbf{x}$
  - 8:     **if**  $\mathbf{x} \in A$  and  $\mathbf{w} \cdot \mathbf{x} < 0$  **then**
  - 9:          $\mathbf{w} = \mathbf{w} + \mathbf{x}$
  - 10:      **end if**
  - 11:     **if**  $\mathbf{x} \in B$  and  $\mathbf{w} \cdot \mathbf{x} \geq 0$  **then**
  - 12:          $\mathbf{w} = \mathbf{w} - \mathbf{x}$
  - 13:      **end if**
  - 14: **end while**
- 

If the input data are binary and linearly separable; then algorithm (1) converges. The proof of convergence of the algorithm is known as the **perceptron convergence theorem**.

### Theorem

Consider algorithm (1) and let  $D$  be a set of training vectors which are linearly separable. Let  $\mathbf{w}^*$  be the weight vectors which defines the separating line with  $\|\mathbf{w}^*\| = 1$ . Then the perceptron convergence theorem states that the number of mistakes  $m$  made by the perceptron algorithm satisfies

$$m \leq \frac{1}{\gamma^2} \quad \text{where} \quad \gamma = \min_{\mathbf{x} \in D} \frac{|\mathbf{w}^{*T} \mathbf{x}|}{\|\mathbf{x}\|_2} \quad (2.7)$$

### Note

1. Since,  $\|\mathbf{w}^*\| = 1 \implies \cos(\theta) = \frac{\mathbf{w}^{*T} \mathbf{x}}{\|\mathbf{x}\|_2}$
2. If  $\|\mathbf{x}\|_2 = 1$ , that is, we scale all the training examples to have unit norm (which has no effect their orientation), then  $\gamma = \min_{\mathbf{x} \in D} |\mathbf{w}^{*T} \mathbf{x}|$  is the minimum distance from any example  $\mathbf{x} \in D$  to the separating line.

## Proof

We first prove the inequality given by

$$(\mathbf{w}^{(t+1)})^T \mathbf{w}^* \geq (\mathbf{w}^{(t)})^T \mathbf{w}^* + \gamma \quad (2.8)$$

### State 1

In state 1, let us consider  $\mathbf{x}$  being positive ( $\mathbf{x} \in C_1$ ) and incorrectly classified then

$$(\mathbf{w}^{(t+1)})^T \mathbf{w}^* = (\mathbf{w}^{(t)} + \mathbf{x})^T \mathbf{w}^* = (\mathbf{w}^{(t)})^T \mathbf{w}^* + \mathbf{x}^T \mathbf{w}^*$$

Assuming that all  $\|\mathbf{x}\|_2 = 1$  then  $\mathbf{x}^T \mathbf{w}^* \geq \gamma$  since  $\gamma$  is the minimum.

$$(\mathbf{w}^{(t+1)})^T \mathbf{w}^* \geq (\mathbf{w}^{(t)})^T \mathbf{w}^* + \gamma \quad (2.9)$$

Thus, we have proved (2.8) under **State 1**.

### State 2

In state 2, let us consider  $\mathbf{x}$  being negative ( $\mathbf{x} \in C_2$ ) and incorrectly classified then

$$(\mathbf{w}^{(t+1)})^T \mathbf{w}^* = (\mathbf{w}^{(t)} - \mathbf{x})^T \mathbf{w}^* = (\mathbf{w}^{(t)})^T \mathbf{w}^* - \mathbf{x}^T \mathbf{w}^*$$

Since  $\mathbf{x} \in C_2 \implies |\mathbf{w}^{*T} \mathbf{x}| = -\mathbf{x}^T \mathbf{w}^* \geq 0$  and  $|\mathbf{x}^T \mathbf{w}^*| \geq \gamma$

$$\begin{aligned} (\mathbf{w}^{(t+1)})^T \mathbf{w}^* &= (\mathbf{w}^{(t)})^T \mathbf{w}^* + |\mathbf{w}^{*T} \mathbf{x}| \\ (\mathbf{w}^{(t+1)})^T \mathbf{w}^* &\geq (\mathbf{w}^{(t)})^T \mathbf{w}^* + \gamma \end{aligned} \quad (2.10)$$

Thus, we have proved (2.8) under **State 2**.

From (2.9) and (2.10), we have shown that  $\forall \mathbf{x} \in D$  that inequality (2.8) holds.

Assume that the inequality (2.8) holds for an arbitrary integer value  $m$  and  $m - 1$

$$(\mathbf{w}^{(m)})^T \mathbf{w}^* \geq (\mathbf{w}^{(m-1)})^T \mathbf{w}^* + \gamma \quad (2.11)$$

$$(\mathbf{w}^{(m-1)})^T \mathbf{w}^* \geq (\mathbf{w}^{(m-2)})^T \mathbf{w}^* + \gamma \quad (2.12)$$

Merging inequality (2.11) and (2.12) gives us

$$(\mathbf{w}^{(m)})^T \mathbf{w}^* \geq (\mathbf{w}^{(m-2)})^T \mathbf{w}^* + 2\gamma \quad (2.13)$$

Hence, by induction, after  $M$  mistakes, inequality (2.13) becomes

$$(\mathbf{w}^{(m)})^T \mathbf{w}^* \geq (\mathbf{w}^{(0)})^T \mathbf{w}^* + m\gamma \quad (2.14)$$

Since  $\mathbf{w}^{(0)} = \mathbf{0}$  (the zero vector), we have

$$(\mathbf{w}^{(m)})^T \mathbf{w}^* \geq m\gamma \quad (2.15)$$

Next, we show that

$$\|\mathbf{w}^{(t+1)}\|_2^2 \leq \|\mathbf{w}^{(t)}\|_2^2 + 1 \quad (2.16)$$

### State 1

Consider the same state 1 as before, thus we have

$$(\mathbf{w}^{(t)})^T \mathbf{x} \leq 0 \quad \text{and} \quad \mathbf{w}^{(t+1)} = \mathbf{w}^{(t)} + \mathbf{x} \quad (2.17)$$

then

$$\begin{aligned} \|\mathbf{w}^{(t+1)}\|_2^2 &= (\mathbf{w}^{(t)} + \mathbf{x})^T (\mathbf{w}^{(t)} + \mathbf{x}) \\ &= (\mathbf{w}^{(t)} + \mathbf{x})^T \mathbf{w}^{(t)} + (\mathbf{w}^{(t)} + \mathbf{x})^T \mathbf{x} \\ &= (\mathbf{w}^{(t)})^T \mathbf{w}^{(t)} + \mathbf{x}^T \mathbf{w}^{(t)} + (\mathbf{w}^{(t)})^T \mathbf{x} + \mathbf{x}^T \mathbf{x} \\ &= (\mathbf{w}^{(t)})^T \mathbf{w}^{(t)} + 2(\mathbf{w}^{(t)})^T \mathbf{x} + \mathbf{x}^T \mathbf{x} \end{aligned} \quad (2.18)$$

Since (2.17) and  $\mathbf{x}^T \mathbf{x} = 1$ , equation (2.18) becomes

$$\|\mathbf{w}^{(t+1)}\|_2^2 \leq \|\mathbf{w}^{(t)}\|_2^2 + 1 \quad (2.19)$$

## State 2

Consider the same state 2 as before, thus we have

$$(\mathbf{w}^{(t)})^T \mathbf{x} > 0 \quad \text{and} \quad \mathbf{w}^{(t+1)} = \mathbf{w}^{(t)} - \mathbf{x} \quad (2.20)$$

then

$$\begin{aligned} \|\mathbf{w}^{(t+1)}\|_2^2 &= (\mathbf{w}^{(t)} - \mathbf{x})^T (\mathbf{w}^{(t)} - \mathbf{x}) \\ &= (\mathbf{w}^{(t)} - \mathbf{x})^T \mathbf{w}^{(t)} - (\mathbf{w}^{(t)} - \mathbf{x})^T \mathbf{x} \\ &= (\mathbf{w}^{(t)})^T \mathbf{w}^{(t)} - \mathbf{x}^T \mathbf{w}^{(t)} - (\mathbf{w}^{(t)})^T \mathbf{x} + \mathbf{x}^T \mathbf{x} \\ &= (\mathbf{w}^{(t)})^T \mathbf{w}^{(t)} - 2(\mathbf{w}^{(t)})^T \mathbf{x} + \mathbf{x}^T \mathbf{x} \end{aligned} \quad (2.21)$$

Since (2.20) and  $\mathbf{x}^T \mathbf{x} = 1$ , equation (2.21) becomes

$$\|\mathbf{w}^{(t+1)}\|_2^2 \leq \|\mathbf{w}^{(t)}\|_2^2 + 1 \quad (2.22)$$

Thus, we have proved (2.16) under **State 2**.

By the same induction defined in (2.14), we have

$$\|\mathbf{w}^{(m)}\|_2^2 \leq m \quad (2.23)$$

Finally, we merge result (2.15) and (2.23) using the Cauchy-Schwarz (CS) inequality

Recall that the CS inequality is given by

$$|\mathbf{w}^{*T} \mathbf{x}| \leq \|\mathbf{w}^*\|_2 \|\mathbf{x}\|_2 \quad (2.24)$$

Hence, using (2.24) we get

$$|(\mathbf{w}^m)^T \mathbf{w}^*| \leq \|(\mathbf{w}^m)^T\|_2 \|\mathbf{w}^*\|_2 \quad (2.25)$$

From the result (2.15), (2.25) becomes

$$m\gamma \leq |(\mathbf{w}^m)^T \mathbf{w}^*| \leq \|(\mathbf{w}^m)^T\|_2 \|\mathbf{w}^*\|_2 \quad (2.26)$$

Using the result (2.23) and the fact that  $\|\mathbf{w}^*\|_2 = 1$ , we get

$$m\gamma \leq \|(\mathbf{w}^m)^T\|_2 \quad (2.27)$$

$$m^2\gamma^2 \leq \|(\mathbf{w}^m)^T\|_2^2 \quad (2.27)$$

$$m^2\gamma^2 \leq m \quad (2.28)$$

$$m \leq \frac{1}{\gamma^2} \quad (2.29)$$

This proves the Perceptron Convergence Theorem that the number of mistakes is at most  $\frac{1}{\gamma^2}$ , where  $\gamma$  is the margin.

Even though the perceptron converges, alone it is not always a good model to solve binary classification problem. It has certain limitation by nature of its definition. Consider the logic functions AND, OR and XOR across these 4 points:

$$\mathbf{p}_1 = \begin{bmatrix} 0 \\ 0 \end{bmatrix}, \mathbf{p}_2 = \begin{bmatrix} 1 \\ 0 \end{bmatrix}, \mathbf{p}_3 = \begin{bmatrix} 1 \\ 1 \end{bmatrix}, \mathbf{p}_4 = \begin{bmatrix} 0 \\ 1 \end{bmatrix} \quad (2.30)$$

### AND Function

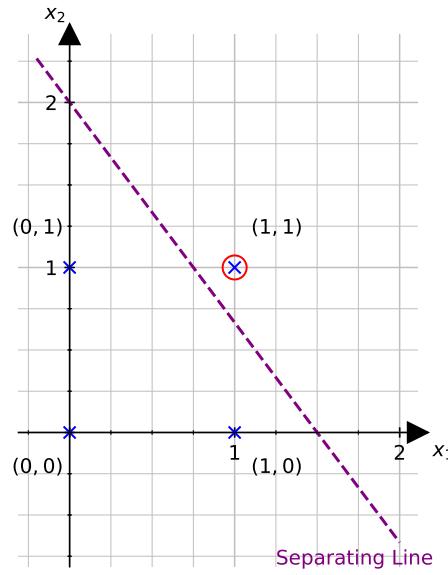
The AND function returns a value of 1 if both of the inputs is 1. Let  $f$  be the AND function such that

$$f(\mathbf{p}_1) = 0 \quad f(\mathbf{p}_2) = 0 \quad f(\mathbf{p}_3) = 1 \quad f(\mathbf{p}_4) = 0$$

Consider the AND function with the separating line below

The equation of the separating line in figure 2.5 is given by

$$x_1 + x_2 - 1.5 = 0 \quad (2.31)$$



**Figure 2.5:** AND Function with separating line

We can transform the separating line in the form of  $\mathbf{w}^T \mathbf{x} = 0$  to fit the perceptron.

Let

$$\mathbf{w} = \begin{bmatrix} -1.5 \\ 1 \\ 1 \end{bmatrix} \quad \text{and} \quad \mathbf{x} = \begin{bmatrix} 1 \\ x_1 \\ x_2 \end{bmatrix} \quad (2.32)$$

Applying function (2.1) on each point to check whether they are correctly classified.

$$\phi(p_1) = \phi \left( \begin{bmatrix} -1.5 \\ 1 \\ 1 \end{bmatrix} \cdot \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix} \right) = \phi(-1.5) = 0$$

$$\phi(p_2) = \phi \left( \begin{bmatrix} -1.5 \\ 1 \\ 1 \end{bmatrix} \cdot \begin{bmatrix} 1 \\ 1 \\ 0 \end{bmatrix} \right) = \phi(-0.5) = 0$$

$$\phi(p_3) = \phi \left( \begin{bmatrix} -1.5 \\ 1 \\ 1 \end{bmatrix} \cdot \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix} \right) = \phi(0.5) = 1$$

$$\phi(p_4) = \phi \left( \begin{bmatrix} -1.5 \\ 1 \\ 1 \end{bmatrix} \cdot \begin{bmatrix} 1 \\ 0 \\ 1 \end{bmatrix} \right) = \phi(-0.5) = 0$$

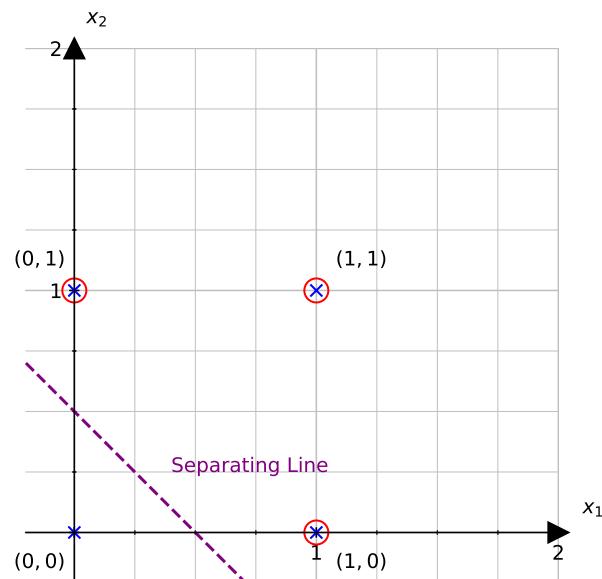
Thus, the perceptron is able to solve the AND function.

### OR Function

The OR function returns a value of 1 if either of the inputs is 1. Let  $f$  be the OR function such that

$$f(\mathbf{p}_1) = 0 \quad f(\mathbf{p}_2) = 1 \quad f(\mathbf{p}_3) = 1 \quad f(\mathbf{p}_4) = 1$$

Consider the OR function with the separating line below



**Figure 2.6:** OR Function with separating line

The equation of the separating line in figure 2.6 is given by

$$x_1 + x_2 - 0.5 = 0 \quad (2.33)$$

We can transform the separating line in the form of  $\mathbf{w}^T \mathbf{x} = 0$  to fit the perceptron.

Let

$$\mathbf{w} = \begin{bmatrix} -0.5 \\ 1 \\ 1 \end{bmatrix} \quad \text{and} \quad \mathbf{x} = \begin{bmatrix} 1 \\ x_1 \\ x_2 \end{bmatrix} \quad (2.34)$$

Applying function (2.1) on each point to check whether they are correctly classified.

$$\phi(p_1) = \phi \left( \begin{bmatrix} -0.5 \\ 1 \\ 1 \end{bmatrix} \cdot \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix} \right) = \phi(-0.5) = 0$$

$$\phi(p_2) = \phi \left( \begin{bmatrix} -0.5 \\ 1 \\ 1 \end{bmatrix} \cdot \begin{bmatrix} 1 \\ 1 \\ 0 \end{bmatrix} \right) = \phi(0.5) = 1$$

$$\phi(p_3) = \phi \left( \begin{bmatrix} -0.5 \\ 1 \\ 1 \end{bmatrix} \cdot \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix} \right) = \phi(1.5) = 1$$

$$\phi(p_4) = \phi \left( \begin{bmatrix} -0.5 \\ 1 \\ 1 \end{bmatrix} \cdot \begin{bmatrix} 1 \\ 0 \\ 1 \end{bmatrix} \right) = \phi(0.5) = 1$$

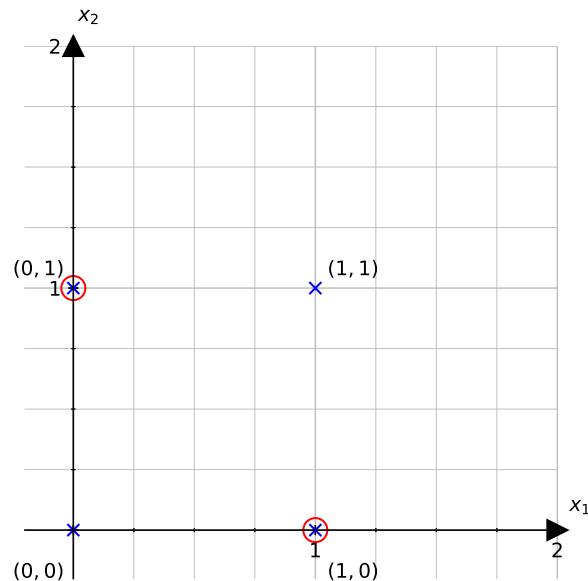
Thus, the perceptron is able to solve the OR function.

## XOR Function

The XOR function returns a value of 1 if either of the inputs is 1 but not both. Let  $f$  be the XOR function such that

$$f(\mathbf{p}_1) = 0, \quad f(\mathbf{p}_2) = 1, \quad f(\mathbf{p}_3) = 0, \quad f(\mathbf{p}_4) = 1 \quad (2.35)$$

Consider a plot of the XOR function below



**Figure 2.7:** XOR function

From figure 2.7; we can observe that no separating line alone could group the two different classes together. We attempt to find a combination of weight for the perceptron which reduces the error (margin) as much as possible. Let  $f^*$  be the exact function and  $f(\mathbf{x}, \mathbf{w}) = \mathbf{w}^T \mathbf{x} + b$ , where  $\mathbf{w} = [w_1 \ w_2]^T$ , be the approximate function. Consider the error function (the difference between the exact and approximate value)

$$\mathbb{E}(\mathbf{w}, b) = \frac{1}{4} \sum_{x \in D} (f^*(\mathbf{x}) - \mathbf{w}^T \mathbf{x} - b)^2 \quad (2.36)$$

we minimise the error function  $\mathbb{E}$  by setting the partial derivatives to 0. The partial derivative with respect to the bias is given by

$$\begin{aligned}\frac{\partial \mathbb{E}}{\partial b} &= -\frac{1}{2} \sum_{x \in D} (f^*(\mathbf{x}) - \mathbf{w}^T \mathbf{x} - b) \\ \frac{\partial \mathbb{E}}{\partial b} = 0 \rightarrow \sum_{x \in D} f^*(\mathbf{x}) &= \sum_{x \in D} \mathbf{w}^T \mathbf{x} - \sum_{x \in D} b\end{aligned}\quad (2.37)$$

The partial derivative with respect to the weight vector is given by

$$\begin{aligned}\frac{\partial \mathbb{E}}{\partial \mathbf{w}} &= \frac{1}{2} \sum_{x \in D} (f^*(\mathbf{x}) - \mathbf{w}^T \mathbf{x} - b) \frac{\partial (-\mathbf{w}^T \mathbf{x})}{\partial \mathbf{w}} \\ &= -\frac{1}{2} \sum_{x \in D} (f^*(\mathbf{x}) - \mathbf{w}^T \mathbf{x} - b) \mathbf{x} \\ \frac{\partial \mathbb{E}}{\partial \mathbf{w}} = 0 \rightarrow \sum_{x \in D} \mathbf{x} f^*(\mathbf{x}) &= \sum_{x \in D} (\mathbf{w}^T \mathbf{x}) \mathbf{x} + b \sum_{x \in D} \mathbf{x}\end{aligned}\quad (2.38)$$

Substituting for values in equation (2.37)

$$\begin{aligned}\sum_{x \in D} f^*(\mathbf{x}) &= f^*(\mathbf{p}_1) + f^*(\mathbf{p}_2) + f^*(\mathbf{p}_3) + f^*(\mathbf{p}_4) \\ &= 0 + 1 + 1 + 0 \\ &= 2\end{aligned}\quad (2.39)$$

$$\begin{aligned}\sum_{x \in D} \mathbf{w}^T \mathbf{x} &= \mathbf{w}^T \begin{bmatrix} 0 \\ 0 \end{bmatrix} + \mathbf{w}^T \begin{bmatrix} 1 \\ 0 \end{bmatrix} + \mathbf{w}^T \begin{bmatrix} 1 \\ 1 \end{bmatrix} + \mathbf{w}^T \begin{bmatrix} 0 \\ 1 \end{bmatrix} \\ &= 2w_1 + 2w_2\end{aligned}\quad (2.40)$$

$$\begin{aligned}\sum_{x \in D} b &= b \sum_{x \in D} 1 \\ &= 4b\end{aligned}\quad (2.41)$$

Thus, we get

$$\begin{aligned}2 &= 2w_1 + 2w_2 + 4b \\ w_1 + w_2 + 2b &= 1\end{aligned}\quad (2.42)$$

Substituting for values in equation (2.38)

$$\begin{aligned} \sum_{x \in D} \mathbf{x} f^*(\mathbf{x}) &= \begin{bmatrix} 0 \\ 0 \end{bmatrix} (0) + \begin{bmatrix} 1 \\ 0 \end{bmatrix} (1) + \begin{bmatrix} 1 \\ 1 \end{bmatrix} (0) + \begin{bmatrix} 0 \\ 1 \end{bmatrix} (1) \\ &= \begin{bmatrix} 1 \\ 1 \end{bmatrix} \end{aligned} \quad (2.43)$$

$$\begin{aligned} \sum_{x \in D} (\mathbf{w}^T \mathbf{x}) \mathbf{x} &= \begin{bmatrix} w_1 & w_2 \end{bmatrix} \begin{bmatrix} 0 \\ 0 \end{bmatrix} \begin{bmatrix} 0 \\ 0 \end{bmatrix} + \begin{bmatrix} w_1 & w_2 \end{bmatrix} \begin{bmatrix} 1 \\ 0 \end{bmatrix} \begin{bmatrix} 1 \\ 0 \end{bmatrix} \\ &\quad + \begin{bmatrix} w_1 & w_2 \end{bmatrix} \begin{bmatrix} 1 \\ 1 \end{bmatrix} \begin{bmatrix} 1 \\ 1 \end{bmatrix} + \begin{bmatrix} w_1 & w_2 \end{bmatrix} \begin{bmatrix} 0 \\ 1 \end{bmatrix} \begin{bmatrix} 0 \\ 1 \end{bmatrix} \\ &= \begin{bmatrix} 2w_1 + w_2 \\ w_1 + 2w_2 \end{bmatrix} \end{aligned} \quad (2.44)$$

$$b \sum_{x \in D} \mathbf{x} = b \begin{bmatrix} 0 \\ 0 \end{bmatrix} + b \begin{bmatrix} 1 \\ 0 \end{bmatrix} + b \begin{bmatrix} 1 \\ 1 \end{bmatrix} + b \begin{bmatrix} 0 \\ 1 \end{bmatrix} \quad (2.45)$$

$$= b \begin{bmatrix} 2 \\ 2 \end{bmatrix} \quad (2.46)$$

Thus, we get

$$\begin{bmatrix} 1 \\ 1 \end{bmatrix} = \begin{bmatrix} 2w_1 + w_2 \\ w_1 + 2w_2 \end{bmatrix} + b \begin{bmatrix} 2 \\ 2 \end{bmatrix} \quad (2.47)$$

From equations (2.42) and (2.47), we get the set of equations that forms the approximate function with the minimum error

$$w_1 + w_2 + 2b = 1$$

$$2w_1 + w_2 + 2b = 1$$

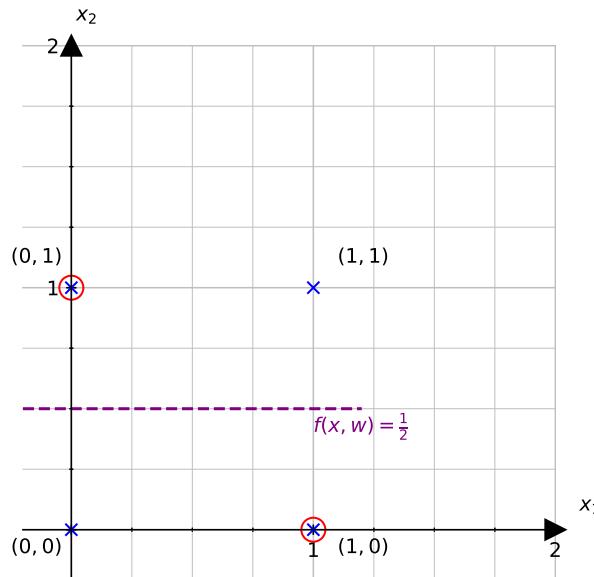
$$w_1 + 2w_2 + 2b = 1$$

Solving the system of linear equations we get

$$w_1 = 0 \quad w_2 = 0 \quad b = \frac{1}{2} \quad (2.48)$$

Hence, the approximate function  $f(\mathbf{x}, \mathbf{w})$  is given by

$$f(\mathbf{x}, \mathbf{w}) = \frac{1}{2} \quad (2.49)$$



**Figure 2.8:** XOR approximate solution

Clearly from figure 2.8, the perceptron model cannot solve the XOR logic function. The XOR problem is actually not linearly separable. Thus, we introduce **feedforward networks** to overcome this problem.

## 2.2 Feedforward Network

We have seen that the single perceptron cannot solve the XOR problem. In order to solve the problem, we use a set of perceptrons that learns two different functions within a connected network.

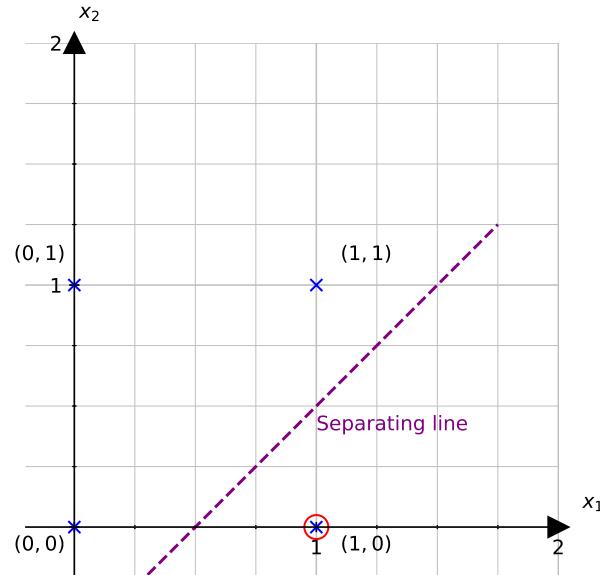
First, let's consider two NOT boolean function  $f_1^* = x_1 \text{ AND } (\text{NOT } x_2)$  and  $f_2^* = (\text{NOT } x_1) \text{ AND } x_2$ .

Consider  $f_1^*$  which returns 1 if  $x_1$  and (not  $x_2$ ) are equal to 1.

$x_1$	$x_2$	$\text{NOT } (x_2)$	$x_1 \text{ AND } (\text{NOT } x_2)$
0	0	1	0
1	0	1	1
1	1	0	0
0	1	0	0

**Table 2.1:** Truth Table for  $f_1^*$

The function  $f_1^*$  can be separated using the perceptron.



**Figure 2.9:** Representation of Truth Table for function  $f_1^*$

The equation of the separating line in figure 2.9 is given by

$$x_1 - x_2 - \frac{1}{2} = 0 \quad (2.50)$$

We can transform the separating line in the form of  $\mathbf{w}^T \mathbf{x} = 0$  to fit the perceptron.

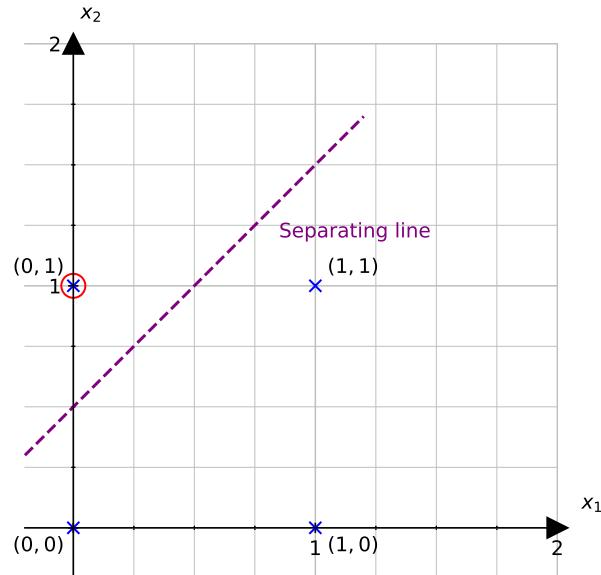
$$\mathbf{w} = \begin{bmatrix} -0.5 \\ 1 \\ -1 \end{bmatrix} \quad \text{and} \quad \mathbf{x} = \begin{bmatrix} 1 \\ x_1 \\ x_2 \end{bmatrix} \quad (2.51)$$

Similarly, consider  $f_2^*$  which returns 1 if (not  $x_1$ ) and  $x_2$  are equal to 1.

$x_1$	$x_2$	NOT ( $x_1$ )	(NOT $x_1$ ) AND $x_2$
0	0	1	0
1	0	0	0
1	1	0	0
0	1	1	1

**Table 2.2:** Truth Table for  $f_2^*$

The function  $f_2^*$  can be separated using the perceptron.



**Figure 2.10:** Representation of Truth Table for function  $f_2^*$

The equation of the separating line in figure 2.10 is given by

$$-x_1 + x_2 - \frac{1}{2} = 0 \quad (2.52)$$

We can transform the separating line in the form of  $\mathbf{w}^T \mathbf{x} = 0$  to fit the perceptron.

$$\mathbf{w} = \begin{bmatrix} -0.5 \\ -1 \\ 1 \end{bmatrix} \quad \text{and} \quad \mathbf{x} = \begin{bmatrix} 1 \\ x_1 \\ x_2 \end{bmatrix} \quad (2.53)$$

Consequently, the XOR function defined in (2.35) can be expressed as  $f^*$  such that

$$f^*(x_1, x_2) = f_1^*(x_1, x_2) \text{ OR } f_2^*(x_1, x_2)$$

$x_1$	$x_2$	$f_1^*(x_1, x_2)$	$f_2^*(x_1, x_2)$	$f_2^*(x_1, x_2)$
0	0	0	0	0
1	0	1	0	1
1	1	0	0	0
0	1	0	1	1

**Table 2.3:** Truth Table for  $f^*$

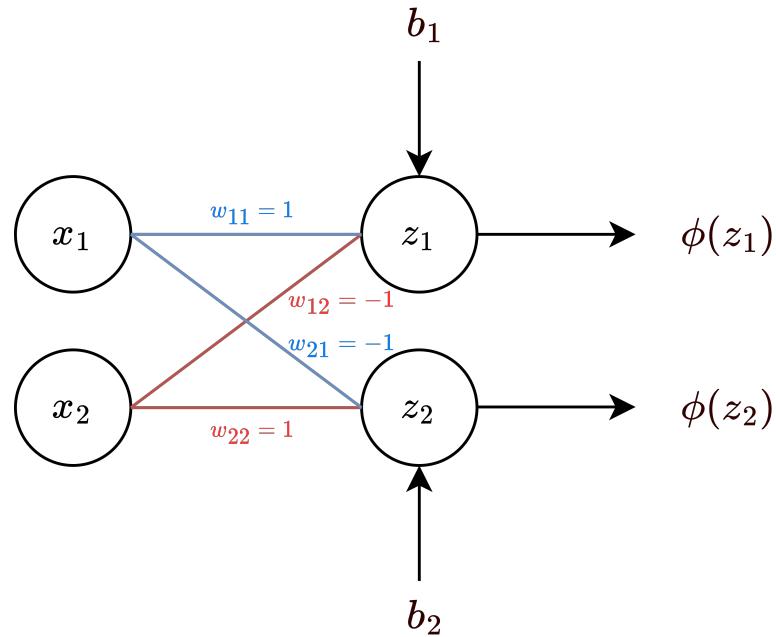
Consider a feedforward network with a hidden layer with two units  $h_1$  and  $h_2$  such that  $h_1$  learns  $f_1^*(x_1, x_2)$  and  $h_2$  learns  $f_2^*(x_1, x_2)$ .

$$\begin{aligned} h_1 &= f_1^*(x_1, x_2) \\ &= \phi(z_1) \end{aligned}$$

where  $z_1 = w_{11}x_1 + w_{12}x_2 + b_1$  and  $w_{11} = 1$ ,  $w_{12} = -1$  and  $b_1 = -0.5$  from separating line in (2.51).

$$\begin{aligned} h_2 &= f_2^*(x_1, x_2) \\ &= \phi(z_2) \end{aligned}$$

where  $z_2 = w_{21}x_1 + w_{22}x_2 + b_2$  and  $w_{21} = -1$ ,  $w_{22} = 1$  and  $b_2 = -0.5$  from separating line in (2.53).



**Figure 2.11:** FFN hidden layer

Thus we obtain the system below for the hidden layer.

$$\begin{bmatrix} z_1 \\ z_2 \end{bmatrix} = \begin{bmatrix} b_1 & w_{11} & w_{12} \\ b_1 & w_{21} & w_{22} \end{bmatrix} \begin{bmatrix} 1 \\ x_1 \\ x_2 \end{bmatrix} \quad (2.54)$$

We then feed forward the information from the hidden layer to another layer.

$$\begin{bmatrix} h_1 \\ h_2 \end{bmatrix} = \phi \left( \begin{bmatrix} z_1 \\ z_2 \end{bmatrix} \right) \quad (2.55)$$

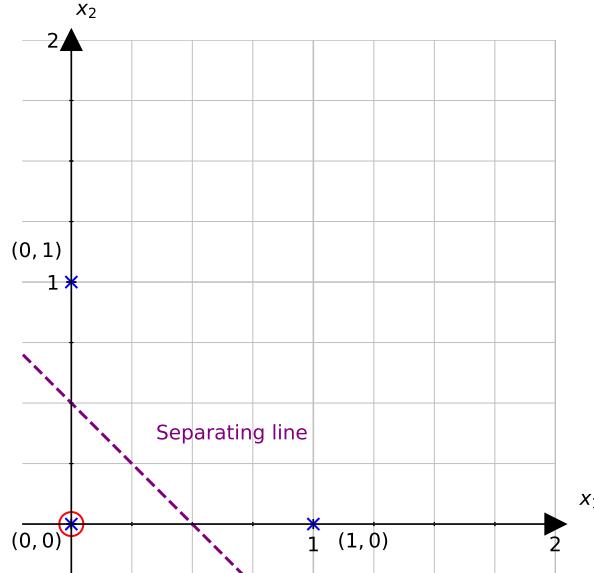
At the hidden layer, we are left with the following truth table.

$x_1$	$x_2$	XOR
0	0	0
1	0	1
0	0	0
0	1	1

redundant

**Table 2.4:** Truth Table from hidden layer

The XOR problem can not be separated from using another perceptron after the hidden layer.



**Figure 2.12:** Representation of the hidden layer

The equation of the separation line is in figure (2.12) is given binary

$$h_1 + h_2 - \frac{1}{2} = 0 \quad (2.56)$$

We can now transform the separating line in the form of  $\mathbf{w}^T \mathbf{x} = 0$  to fit the perceptron.

$$\mathbf{w} = \begin{bmatrix} -0.5 \\ 1 \\ 1 \end{bmatrix} \quad \mathbf{h} = \begin{bmatrix} 1 \\ h_1 \\ h_2 \end{bmatrix} \quad (2.57)$$

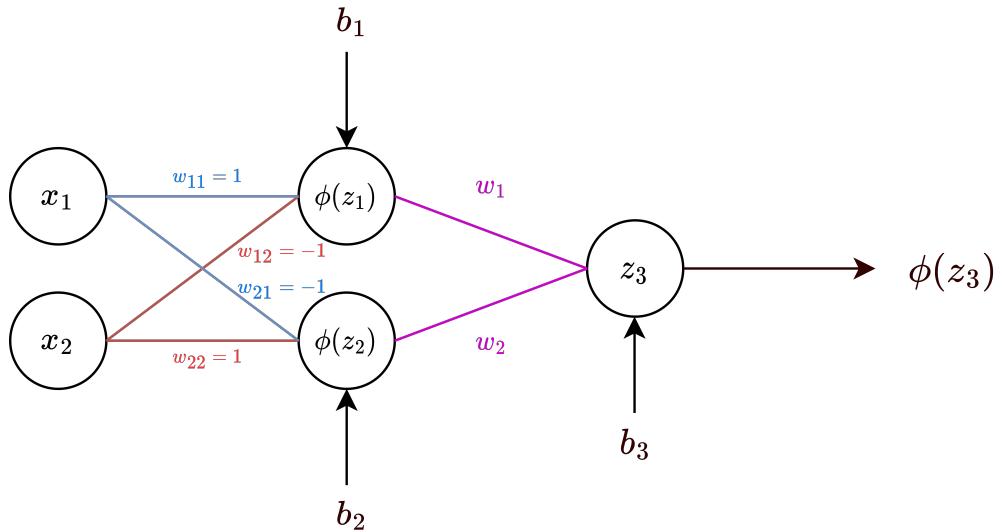
Finally, the XOR function can be represented using the set of transform defined below.

$$f(x_1, x_2) = \phi' \left( \mathbf{w}_2^T \left( \phi \left( \mathbf{w}_1^T \mathbf{x} \right) \right) \right) \quad (2.58)$$

where

$$\mathbf{x} = \begin{bmatrix} 1 \\ x_1 \\ x_2 \end{bmatrix} \quad \mathbf{w}_1 = \begin{bmatrix} -0.5 & -0.5 \\ 1 & -1 \\ -1 & 1 \end{bmatrix} \quad \mathbf{w}_2 = \begin{bmatrix} -0.5 \\ 1 \\ 1 \end{bmatrix} \quad (2.59)$$

$\phi^*$  takes an augmented input from the calculation of  $\phi(\mathbf{w}_1^T \mathbf{x})$  to accommodate for the bias.



**Figure 2.13:** Feedforward network for XOR problem

Checking for  $\mathbf{x} = [1 \ 0 \ 0]^T$

$$\begin{aligned} \mathbf{w}_1^T \mathbf{x} &= \begin{bmatrix} -0.5 & 1 & -1 \\ -0.5 & -1 & 1 \end{bmatrix} \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix} = \begin{bmatrix} -0.5 \\ -0.5 \end{bmatrix} \\ \phi(\mathbf{w}_1^T \mathbf{x}) &= \phi \left( \begin{bmatrix} -0.5 \\ -0.5 \end{bmatrix} \right) = \begin{bmatrix} 0 \\ 0 \end{bmatrix} \\ \phi^* \left( \mathbf{w}_2^T \left( \phi \left( \mathbf{w}_1^T \mathbf{x} \right) \right) \right) &= \phi^* \left( \begin{bmatrix} -0.5 & 1 & 1 \end{bmatrix} \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix} \right) = \phi^*(-0.5) \end{aligned}$$

Thus, for  $\mathbf{p}_1 = [0, 0]^T$

$$f(x_1, x_2) = \phi^*(-0.5) = 0$$

**Checking for  $\mathbf{x} = [1 \ 0 \ 1]^T$**

$$\begin{aligned}\mathbf{w}_1^T \mathbf{x} &= \begin{bmatrix} -0.5 & 1 & -1 \\ -0.5 & -1 & 1 \end{bmatrix} \begin{bmatrix} 1 \\ 0 \\ 1 \end{bmatrix} = \begin{bmatrix} 1 \\ 0 \\ 1 \end{bmatrix} = \begin{bmatrix} -1.5 \\ 0.5 \end{bmatrix} \\ \phi(\mathbf{w}_1^T \mathbf{x}) &= \phi\left(\begin{bmatrix} -1.5 \\ 0.5 \end{bmatrix}\right) = \begin{bmatrix} 0 \\ 1 \end{bmatrix} \\ \phi^*\left(\mathbf{w}_2^T \left(\phi\left(\mathbf{w}_1^T \mathbf{x}\right)\right)\right) &= \phi^*\left(\begin{bmatrix} -0.5 & 1 & 1 \end{bmatrix} \begin{bmatrix} 1 \\ 0 \\ 1 \end{bmatrix}\right) = \phi^*(0.5)\end{aligned}$$

Thus, for  $\mathbf{p}_4 = [0, 0]^T$

$$f(x_1, x_2) = \phi^*(0.5) = 1$$

## 2.3 Activation Function

The hard-delimiters used in our previous feedforward network are linear. However, different activation functions can be used with a perceptron to introduce non-linearity in our network. The non-linearity enables a wider range of problems to be tackled. The function to be used is subjective to the problem being solved and the form of the desired result we want. Some common activation functions are defined below.

Linear

$$\phi(z) = z \tag{2.60}$$

Unit Step (Heaviside Function)

$$\phi(z) = \begin{cases} 0 & z < 0 \\ 0.5 & z = 0 \\ 1 & z > 0 \end{cases} \quad (2.61)$$

Signum

$$\phi(z) = \begin{cases} -1 & z < 0 \\ 0 & z = 0 \\ 1 & z > 0 \end{cases} \quad (2.62)$$

Sigmoid

$$\phi(z) = \frac{1}{1 + e^{-z}} \quad (2.63)$$

Hyperbolic Tangent(tanh)

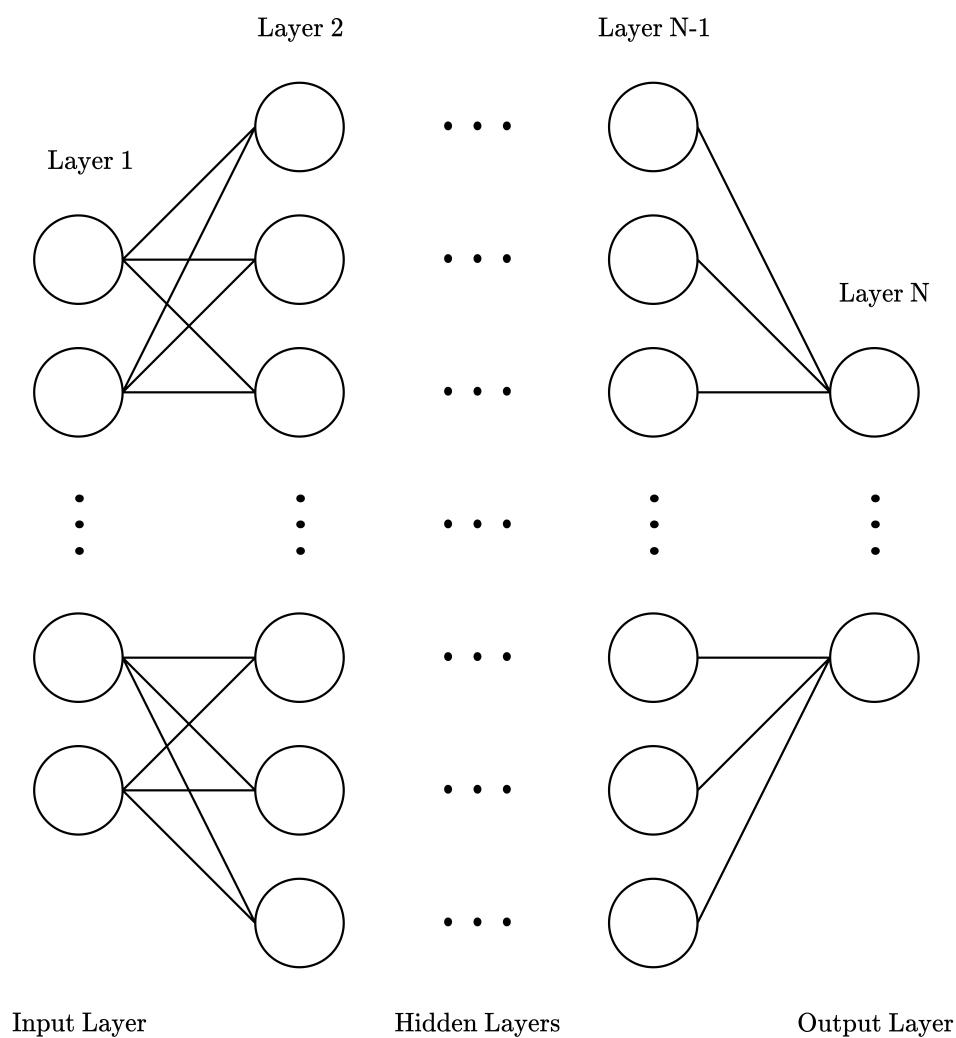
$$\phi(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}} \quad (2.64)$$

ReLU

$$\phi(z) = \begin{cases} 0 & z < 0 \\ z & z > 0 \end{cases} \quad (2.65)$$

## 2.4 Backpropagation

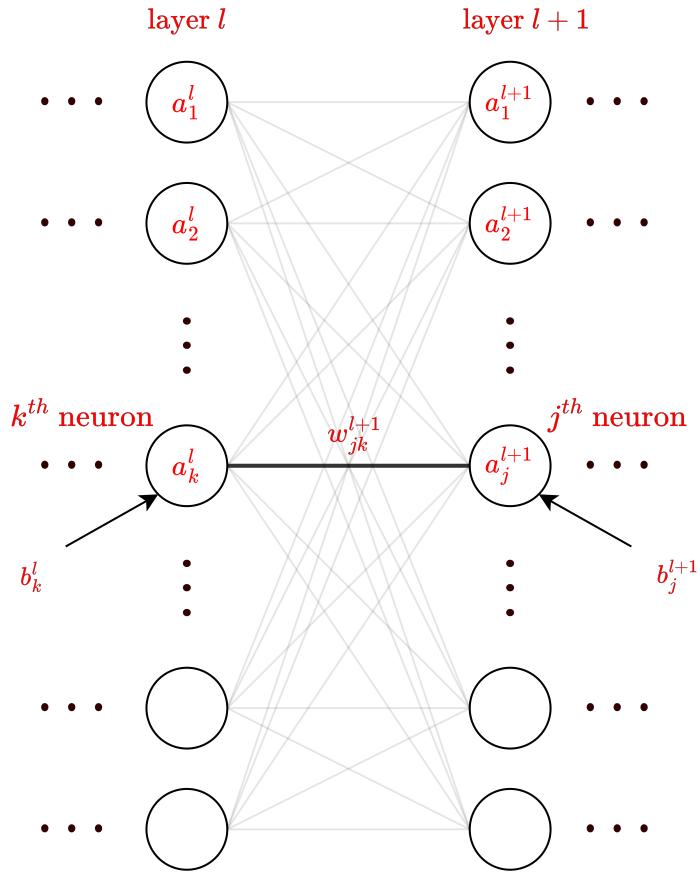
We have seen how information flows in a feedforward neural network. Backpropagation algorithm is the process of spreading the error at the output layer throughout the other layers in the network. The goal is to understand how does the error changes with respect to the weights and biases. Consider the neural network below with  $N$  number of layers (including the input and output layer).



**Figure 2.14:** Neural Network with  $N$  number of layers

The annotation between two consecutive layers,  $l$  and  $l + 1$ , in the network is given by the following set of indexed values.

- $b_j^{l+1}$  → Bias input at the  $j^{th}$  neuron in the  $l + 1^{th}$  layer
- $w_{jk}^{l+1}$  → Weight between the  $k^{th}$  and  $j^{th}$  neuron in the  $l^{th}$  and  $(l + 1)^{th}$  layer respectively
- $z_j^{l+1}$  → Input of the activation function at the  $j^{th}$  neuron in the  $(l + 1)^{th}$  layer
- $a_j^l$  → Output from the activation function at the  $j^{th}$  neuron in the  $l^{th}$  layer



**Figure 2.15:** Consecutive layers,  $l$  and  $l + 1$ , in a neural network

Let the activation function be denoted as  $\sigma$  such that

$$a_j^{(l+1)} = \sigma(z_j^{(l+1)}) \quad (2.66)$$

$$z_j^{(l+1)} = \sum_k w_{jk}^{(l+1)} a_k^{(l)} + b_j^{(l+1)} \quad (2.67)$$

We start at the end of the network by calculating the value of the cost/error function.

Let  $C$  be the cost function which takes as input the output layer

$$\begin{aligned} C &= C(a_j^L) \\ &= \frac{1}{2} \sum_j (y_j - a_j^L)^2 \end{aligned} \tag{2.68}$$

where  $y_j$  is the desired exact value that we want the network to predict. Also, let the change in the cost function with respect to the input of the activation function be given by  $\delta_j^L$  where

$$\delta_j^L = \frac{\partial C}{\partial z_j^L}$$

Using result (2.66) and the definition (2.68)

$$\begin{aligned} \delta_j^L &= \frac{\partial C(a_j^L)}{\partial z_j^L} \\ &= \frac{\partial C(\sigma(z_j^L))}{\partial z_j^L} \end{aligned}$$

Applying the chain rule

$$\begin{aligned} \delta_j^L &= \frac{\partial C(\sigma(z_j^L))}{\partial z_j^L} \\ &= \frac{\partial C}{\partial a_j^L} \frac{\partial a_j^L}{\partial z_j^L} \end{aligned}$$

From equation (2.66)

$$\begin{aligned} a_j^{(L)} &= \sigma(z_j^{(L)}) \\ \frac{\partial a_j^{(L)}}{\partial z_j^{(L)}} &= \sigma'(z_j^{(L)}) \end{aligned}$$

Thus, the change in the cost function with respect to the input of the activation

function in the last layer is given by

$$\delta_j^{(L)} = \frac{\partial C}{\partial a_j^{(L)}} \sigma'(z_j^{(L)}) \quad (2.69)$$

Next, we find how to back-propagate the changes from an upper layer  $l + 1$  to the previous layer  $l$ . Consider the change in the cost function based at the  $l^{th}$  layer

$$\delta_j^{(l)} = \frac{\partial C}{\partial z_j^{(l)}} \quad (2.70)$$

From figure (2.15); an element in the  $l^{th}$  layer influences all the neurons in the  $(l + 1)^{th}$  layer. Thus, when applying the chain rule we sum over all the neurons which gives

$$\delta_j^l = \sum_k \frac{\partial C}{\partial z_k^{(l+1)}} \frac{\partial z_k^{(l+1)}}{\partial z_j^{(l)}} \quad (2.71)$$

From equations (2.66) and (2.67)

$$\begin{aligned} \frac{\partial z_k^{(l+1)}}{\partial z_j^{(l)}} &= \frac{\partial}{\partial z_j^{(l)}} \left( \sum_s w_{ks}^{(l+1)} \sigma(z_s^{(l)}) + b_k^{(l+1)} \right) \\ &= \sum_s w_{ks}^{(l+1)} \frac{\partial \sigma(z_s^{(l)})}{\partial z_j^{(l)}} \\ &= \sum_s w_{ks}^{(l+1)} \sigma'(z_s^{(l)}) \frac{\partial z_s^{(l)}}{\partial z_j^{(l)}} \end{aligned}$$

The partial derivative only exist and is 1 when  $s = j$ ; else the value is 0.

$$\frac{\partial z_k^{(l+1)}}{\partial z_j^{(l)}} = w_{kj}^{(l+1)} \sigma'(z_j^{(l)}) \quad (2.72)$$

Substituting (2.72) and (2.70) into (2.71)

$$\delta_j^l = \sum_k \delta_k^{(l+1)} w_{kj}^{(l+1)} \sigma'(z_j^{(l)}) \quad (2.73)$$

Equation (2.73) are the changes in  $l^{th}$  layer from the changes that happened in

the  $(l + 1)^{th}$  layer. Since we do not explicitly adjust the output and input of the activation functions, we derive the changes of the cost function with respect to the bias and weight. Consider the effect of the bias at each layer given by

$$\frac{\partial C}{\partial b_j^{(l)}} = \frac{\partial C}{\partial z_j^{(l)}} \frac{\partial z_j^{(l)}}{\partial b_j^{(l)}}$$

Using equation (2.67)

$$\begin{aligned} \frac{\partial z_j^{(l)}}{\partial b_j^{(l)}} &= \frac{\partial}{\partial b_j^{(l)}} \left( \sum_k w_{jk}^{(l+1)} a_k^{(l)} + b_j^{(l+1)} \right) \\ &= 1 \end{aligned}$$

Thus, the change of in the cost function with respect to the bias at any layer is given by

$$\frac{\partial C}{\partial b_j^{(l)}} = \delta_i^{(j)} \quad (2.74)$$

Finally, consider the rate of change of the cost function with respect to any weight in the network given by

$$\frac{\partial C}{\partial w_{jk}^{(l)}} = \frac{\partial C}{\partial z_m^{(l)}} \frac{\partial z_m^{(l)}}{\partial w_{jk}^{(l)}} \quad (2.75)$$

Using equation (2.67)

$$\begin{aligned} \frac{\partial z_m^{(l)}}{\partial w_{jk}^{(l)}} &= \frac{\partial}{\partial w_{jk}^{(l)}} \left( \sum_s w_{ms}^{(l)} a_s^{(l-1)} + b_m^{(l)} \right) \\ &= \sum_s a_s^{(l-1)} \frac{\partial w_{ms}^{(l)}}{\partial w_{jk}^{(l)}} \end{aligned}$$

The partial derivative only exist and is 1 when  $s = k$  and  $m = j$ ; else the value is 0.

$$\frac{\partial z_j^{(l)}}{\partial w_{jk}^{(l)}} = a_k^{(l-1)} \quad (2.76)$$

Substituting (2.72) and (2.76) into (2.75)

$$\frac{\partial C}{\partial w_{jk}^{(l)}} = \delta_j^{(l)} a_k^{(l-1)} \quad (2.77)$$

The four main back-propagation equations are given by

$$\delta_j^{(L)} = \frac{\partial C}{\partial a_j^{(L)}} \sigma'(z_j^{(L)}) \quad (2.78)$$

$$\delta_j^l = \sum_k \delta_k^{(l+1)} w_{kj}^{(l+1)} \sigma'(z_j^{(l)}) \quad (2.79)$$

$$\frac{\partial C}{\partial b_j^{(l)}} = \delta_i^{(j)} \quad (2.80)$$

$$\frac{\partial C}{\partial w_{jk}^{(l)}} = \delta_j^{(l)} a_k^{(l-1)} \quad (2.81)$$

We vectorize the above set of equations by using the Hadamard product. Given two matrices of the  $A$  and  $B$  of the same dimension,  $m \times n$ , the Hadamard product is given as

$$(A \odot B)_{mn} = (A)_{mn} (B)_{mn}$$

The vectorized form of the back-propagation equations are given by

$$\delta^{(L)} = \nabla_{a^{(L)}} C \odot \sigma'(z^{(L)}) \quad (2.82)$$

$$\delta^{(l)} = (w^{(l+1)})^T \sigma^{(l+1)} \odot \sigma'(z^{(l)}) \quad (2.83)$$

$$\nabla_{b^{(l)}} C = \delta^{(l)} \quad (2.84)$$

$$\nabla_{w^{(l)}} C = \delta^{(l)} a^{(l-1)} \quad (2.85)$$

---

**Algorithm 2** Back-Propagation Algorithm

---

```
1: Start
2: Step 1 (Input) Compute the first activation layer  $a^{(1)}$  using input values  $x$ 
3:    $a^{(1)} = \sigma(w^{(1)}x + b^{(1)})$ 
4: Step 2 (Feed-Forward)
5:   Feed forward the input to the next layers
6: for  $l = 2, 3, \dots, L$  do
7:    $z^{(l)} = w^{(l)}a^{(l-1)} + b^{(l)}$ 
8:    $a^{(l)} = \sigma(z^{(l)})$ 
9: end for
10: Step 3 (Cost/Error Calculation)
11:   Compute the cost function,  $C$ , at the output layer  $L$ 
12: Step 4 (Backpropagate)
13:   Compute the rate of change of the cost function w.r.t to  $z^{(L)}$ 
14:    $\delta^{(L)} = \nabla_{a^{(L)}}C \odot \sigma'(z^{(L)})$ 
15: for  $l = L - 1, L - 2, \dots, 2$  do
16:    $\delta^{(l)} = (w^{(l+1)})^T \sigma^{(l+1)} \odot \sigma'(z^{(l)})$ 
17:    $\nabla_{b^{(l)}}C = \delta^{(l)}$ 
18:    $\nabla_{w^{(l)}}C = \delta^{(l)}a^{(l-1)}$ 
19: end for
```

---

## 2.5 Sequential Neural Networks

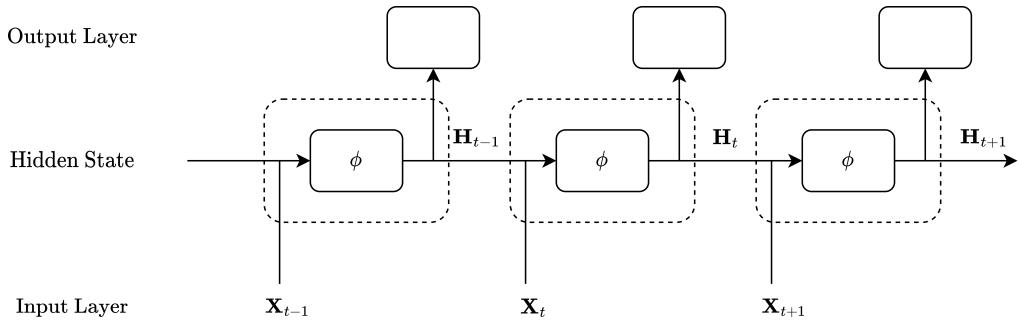
### Recurrent Neural Network(RNN)

Developed through the work of (Rumelhart et al. 1986), a recurrent neural network is a type of neural network capable of modelling sequential data for predictions. Like to the feedforward network, the RNN consists of an input layer and an output layer with a hidden state instead of a hidden layer. The hidden state act as a feedback loop. That is, the output at a particular step in the sequence is fed to the next step's input to compute the next step's output. Hence, recurrent neural networks are neural networks with hidden states. Consider the sequential input data  $\mathbf{X}_t \in \mathbb{R}^{n \times d}$ , with batch size  $n$  and  $d$  inputs, at a particular time step  $t$ ,  $\mathbf{H}_t \in \mathbb{R}^{n \times h}$  be the hidden layer,  $\mathbf{W}_{xh} \in \mathbb{R}^{d \times h}$  be the weight applied on the input data,  $\mathbf{W}_{hh} \in \mathbb{R}^{h \times h}$  be the weight applied to the output hidden state,  $\mathbf{b}_h \in \mathbb{R}^{1 \times h}$  and  $\phi$  be the activation function; thus the calculation at the hidden layer is given by:

$$\mathbf{H}_t = \phi(\mathbf{X}_t \mathbf{W}_{xh} + \mathbf{H}_{t-1} \mathbf{W}_{hh} + \mathbf{b}_h) \quad (2.86)$$

Consider the weights  $\mathbf{W}_{hq} \in \mathbb{R}^{d \times h}$  and the bias  $\mathbf{b}_q \in \mathbb{R}^{1 \times q}$ , thus the computation at the output layer is given by:

$$\mathbf{O}_t = \mathbf{H}_t \mathbf{W}_{hq} + \mathbf{b}_q \quad (2.87)$$



**Figure 2.16:** An RNN with a hidden state

As we unfold a recurrent neural network for more time steps, the number of parameters does not increase. The input of the activation function gets recursively very large or very small. Consequently, we are limited by the problem of vanishing or exploding gradients when optimizing recurrent neural networks (Bengio, Simard & Frasconi 1994) (Kolen & Kremer 2001). A preventive measure would be to truncate the recursive gradient term during backpropagation. RNNs also suffer from short-term memory; important information in very long sequence from the beginning are left out.

### Long Short Term Memory(LSTM)

Introduced in 1997 by (Hochreiter & Schmidhuber 1997), the long short-term memory model was introduced to solve the vanishing and exploding problem in the RNN. While the architecture is similar to the RNN, the LSTM replaces the hidden state with a memory cell. The memory cell comprises three main “gates” and an input node: the forget gate, the input gate, the output gate and the input node, respectively. The forget gate controls whether or not we keep the input from the previous hidden state. The input gate determines how much of the input should contribute to the internal cell memory and be passed to the input node. The memory cell is

calculated at the input node using the input value and the value from the input gate. Finally, the output gate determines how much of the memory cell of the current time step impacts the next time step in the next memory cell. Consider the input  $\mathbf{X}_t \in \mathbb{R}^{n \times d}$ , the hidden state of the previous time step be  $\mathbf{H}_{t-1} \in \mathbb{R}^{n \times h}$ , the input gate be given by  $\mathbf{I}_t \in \mathbb{R}^{n \times h}$ , the forget gate given by  $\mathbf{F}_t \in \mathbb{R}^{n \times h}$ , the input node given by  $\tilde{\mathbf{C}}_t \in \mathbb{R}^{n \times h}$ , the output gate be  $\mathbf{O}_t \in \mathbb{R}^{n \times h}$  and the memory cell state be given by  $\mathbf{C}_t \in \mathbb{R}^{n \times h}$  then the computations of the LSTM are given by:

$$\mathbf{I}_t = \sigma(\mathbf{X}_t \mathbf{W}_{xi} + \mathbf{H}_{t-1} \mathbf{W}_{hi} + \mathbf{b}_i) \quad (2.88)$$

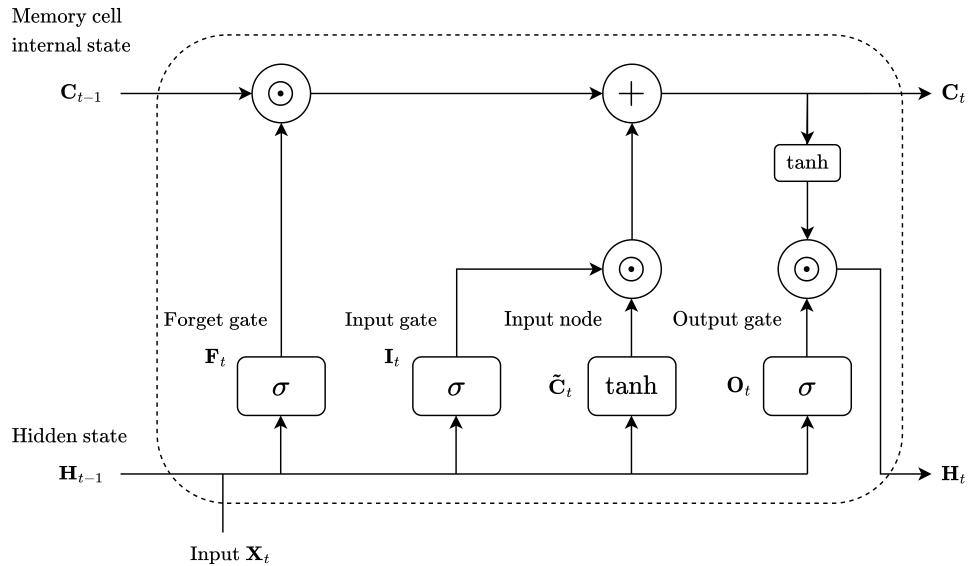
$$\mathbf{F}_t = \sigma(\mathbf{X}_t \mathbf{W}_{xf} + \mathbf{H}_{t-1} \mathbf{W}_{hf} + \mathbf{b}_f) \quad (2.89)$$

$$\mathbf{O}_t = \sigma(\mathbf{X}_t \mathbf{W}_{xo} + \mathbf{H}_{t-1} \mathbf{W}_{ho} + \mathbf{b}_o) \quad (2.90)$$

$$\tilde{\mathbf{C}}_t = \tanh(\mathbf{X}_t \mathbf{W}_{xc} + \mathbf{H}_{t-1} \mathbf{W}_{hc} + \mathbf{b}_c) \quad (2.91)$$

$$\mathbf{C}_t = \mathbf{F}_t \odot \mathbf{C}_{t-1} + \mathbf{I}_t \odot \tilde{\mathbf{C}}_t \quad (2.92)$$

where  $\mathbf{W}_{xi}, \mathbf{W}_{xf}, \mathbf{W}_{xo}, \mathbf{W}_{xc} \in \mathbb{R}^{d \times h}$  and  $\mathbf{W}_{hi}, \mathbf{W}_{hf}, \mathbf{W}_{ho}, \mathbf{W}_{hc} \in \mathbb{R}^{h \times h}$  are weight parameters and  $\mathbf{b}_i, \mathbf{b}_f, \mathbf{b}_o, \mathbf{b}_c \in \mathbb{R}^{1 \times h}$  are the bias parameters.



**Figure 2.17:** Computing the hidden state in an LSTM model

LSTM addresses the vanishing and exploding gradient problem of the RNN by controlling the amount of information being carried to the memory and providing

a continuous gradient flow to facilitate backpropagation. Many variants of LSTM have been developed over the years due to their dominance in sequence learning. Some LTSM variants are: The Peephole Variant, the Coupled Gate Variation, Gated Recurrent Units (GRU). However, they are pretty costly to compute due to their long-range dependencies on the sequence.

### Bidirectional RNN(BRNN)

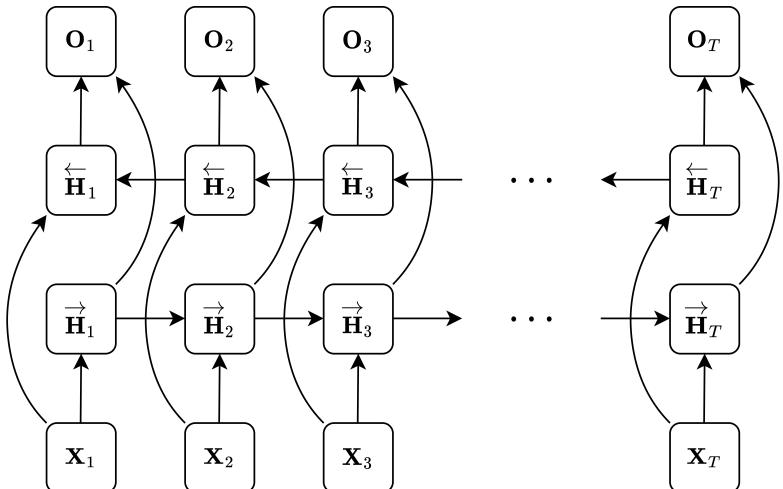
In order to overcome the uni-direction learning of RNN and LSTM,(Schuster & Paliwal 1997) introduced the bidirectional recurrent neural network. The implementation consists of two unidirectional layers chained together in opposite directions but taking the same inputs. The output is obtained by concatenating the two directional outputs from the hidden layers. The obtained layers is then computed to get the prediction. Consider the input  $X_t \in \mathbf{R}^n$  for time step  $t$ , the forward and backward hidden unidirectional layers for time step  $t$  be given by  $\vec{H}_t \in \mathbb{R}^{n \times h}$  and  $\overleftarrow{H}_t \in \mathbb{R}^{n \times h}$  respectively and the output layer be given by  $\mathbf{O}_t$ , then the computations of the BRNN is given by:

$$\vec{H}_t = \phi(\mathbf{X}_t \mathbf{W}_{xh}^{(f)} + \vec{\mathbf{H}}_{t-1} \mathbf{W}_{hh}^{(f)} + \mathbf{b}_h^{(f)}) \quad (2.93)$$

$$\overleftarrow{H}_t = \phi(\mathbf{X}_t \mathbf{W}_{xh}^{(b)} + \overleftarrow{\mathbf{H}}_{t+1} \mathbf{W}_{hh}^{(b)} + \mathbf{b}_h^{(b)}) \quad (2.94)$$

$$\mathbf{O}_t = \mathbf{H}_t \mathbf{W}_{tq} + \mathbf{b}_q$$

where  $\mathbf{W}_{xh}^{(f)}$  and  $\mathbf{W}_{xh}^{(b)} \in \mathbb{R}^{d \times h}$ ,  $\mathbf{W}_{hh}^{(f)}$  and  $\mathbf{W}_{hh}^{(b)} \in \mathbb{R}^{h \times h}$ ,  $\mathbf{W}_{hq} \in \mathbb{R}^{2h \times q}$  are the weight parameters and  $\mathbf{b}_h^{(f)}$  and  $\mathbf{b}_h^{(b)} \in \mathbb{R}^{1 \times h}$  and  $\mathbf{b}_q \in \mathbb{R}^{1 \times q}$  are the bias parameters.



**Figure 2.18:** Architecture of a bidirectional RNN

Bidirectional recurrent neural networks are very efficient in sequential modelling in cases where context is a priority. Natural language processing, speech recognition, handwriting recognition, and part-of-speech tagging; leverages BRNN as context input is required.

# Chapter 3

## Optimization

The optimization problem is a computational problem in which the objective is to find the best of all possible solutions. Deep neural networks is a form of an optimization problem to find the best possible set of weights in order to reduce the error in a network.

A generic form of an optimization problem is given by

$$\begin{aligned} & \underset{x}{\text{minimize/maximize}} && f(x) \\ & \text{subject to} && g_i(x) \leq 0, \quad i = 1, \dots, m \\ & && h_j(x) = 0, \quad j = 1, \dots, p \end{aligned} \tag{3.1}$$

where

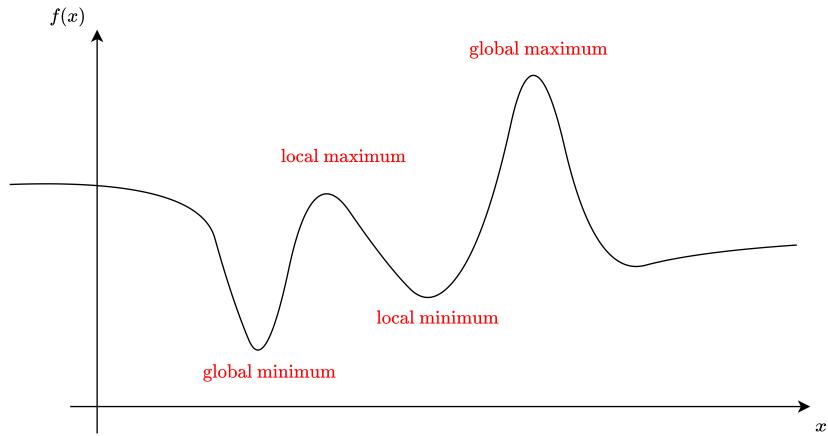
$f : \mathbb{R}^n \rightarrow \mathbb{R}$  is the objective/loss function to be minimised,  
 $g_i(x) \leq 0$  are called inequality constraints,  
 $h_j(x) = 0$  are called equality constraints, and  
 $m \geq 0$  and  $p \geq 0$ . If  $m = p = 0$ , the problem is an unconstrained optimization problem.

### Minimum/Maximum Point

Let  $x^*$  be points of the function  $f(x)$  defined in (3.1) and  $u \in \mathbb{N}(x^*, \delta)$  where  $\mathbb{N}$  is a  $\delta$ -neighbourhood of  $x^*$ . Then,  $x^*$  is said to be a

- (i) local minimum if  $f(x^*) < f(u) \forall u \in \mathbb{N}$

- (ii) local maximum if  $f(x^*) > f(u) \forall u \in \mathbb{N}$
- (iii) global minimum if  $f(x^*) < f(u) \forall u \in \mathbb{R}$
- (iv) global maximum if  $f(x^*) > f(u) \forall u \in \mathbb{R}$



**Figure 3.1:** Minimum and Maximum points

### Determining Minimum/Maximum Point

For a function  $f(\mathbf{x})$  where  $\mathbf{x} = (x_1, x_2, \dots, x_n) \in \mathbb{R}^n$ , the condition for the presence of a stationary point (minimum or maximum point) is given by

$$\mathbf{G} = \nabla f \left( \frac{\partial f}{\partial x_1}, \frac{\partial f}{\partial x_2}, \dots, \frac{\partial f}{\partial x_n} \right) = \mathbf{0} \quad (3.2)$$

The second derivative of  $f(x)$  is given by the Hessian matrix,  $\mathbf{H}$

$$\mathbf{H} = \begin{bmatrix} \frac{\partial^2 f}{\partial x_1^2} & \frac{\partial^2 f}{\partial x_1 \partial x_2} & \cdots & \frac{\partial^2 f}{\partial x_1 \partial x_n} \\ \frac{\partial^2 f}{\partial x_2 \partial x_1} & \frac{\partial^2 f}{\partial x_2^2} & \cdots & \frac{\partial^2 f}{\partial x_2 \partial x_n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial^2 f}{\partial x_n \partial x_1} & \frac{\partial^2 f}{\partial x_n \partial x_2} & \cdots & \frac{\partial^2 f}{\partial x_n^2} \end{bmatrix} \quad (3.3)$$

The nature of the stationary points of  $f(x)$  can be determined by studying the positive definiteness of (3.3) using its eigenvalues. If all the eigenvalues of  $\mathbf{H}$  are

positive, then  $\mathbf{H}$  is symmetric positive definite, indicating the presence of a **local minimum**.

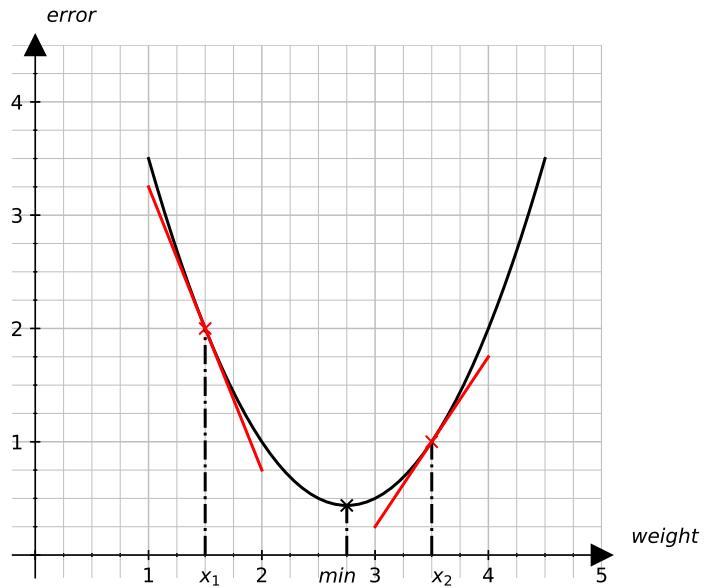
### 3.1 Optimization algorithms

The solutions to the optimization problem are vital in modern machine learning and artificial intelligence algorithms, which includes weight optimization in deep learning. There are a number of popular optimization algorithm currently developed to solve the problem. Hence, choosing the right algorithm can be challenging as well. Let the network cost function,  $c$ , be given by the squared difference between the predicted value and the true value:

$$c = \frac{1}{2n} \sum_i^n (\hat{y}_i - y_i)^2 \quad (3.4)$$

where  $n$  is the total number of input.

The difference is squared to avoid the sum of errors of multiple input vector to be zero which can mislead the network to have perfect predictive power. The goal of the network is to adjust the weight to reduce the network error as much as possible. The idea of reducing a function to a value is synonymous to (3.1) an optimization problem. The network error in (3.4) is a quadratic equation in this case.



**Figure 3.2:** Network Error over different weight

Figure (3.2) is a graphical representation of the network error function. Our goal is to reach the minimum of the function by adjusting the weight value.

## Gradient Descent

In order to reach the bottom of the function; we need to adjust the weight such that the derivative of the error function is 0. The approach of updating the weight based on the gradient of the error function is known as **gradient descent**. The derivative with respect to the weights of the network error (3.4) for a single input is given by

$$\frac{\partial e}{\partial w_{ij}} = (\hat{y}_i - y_i) \frac{\partial \hat{y}_i}{\partial w_{ij}} \quad (3.5)$$

From figure (3.2); we can observe that if the gradient is negative then we have underestimated the predicted value and need to increase the weight to reach the optimal value. On the other hand, if the gradient is positive then we have overestimated the predicted value and need to decrease the weight to reach the optimal value. The equation (3.5) provides us with the opposite direction and amount to adjust the weight. Hence, the update rule of the weights for gradient descent method is given

by

$$w_{ij}^{t+1} = w_{ij}^t - \frac{de}{dw_{ij}} \quad (3.6)$$

In the gradient descent method, the network learns from the gradient of the error function and adjust the weights accordingly to reduce the error. However, the gradient alone can be quite large, causing oscillations as we go down the error function. This problem by introducing a **learning rate**  $\alpha$  prior to adjusting the weight.

### Learning Rate

The learning rate is typically denoted by the Greek letter alpha  $\alpha$ . It helps the network to control the rate at which the weights are changing. Having a system with a high learning rate may lead to an oscillating network when trying to find the optimal weight and having a slow learning rate increases the number of iterations required when optimizing the network. The adjusted update rule of the weight is given by

$$w_{ij}^{t+1} = w_{ij}^t - \alpha \frac{\partial e}{\partial w_{ij}} \quad (3.7)$$

where  $\alpha \in (0, 1)$

### Stochastic Gradient Descent

The Stochastic Gradient Descent dates back to the Robbins-Monro algorithm from the 1950s and is still an important optimization algorithm (Robbins & Monro 1951). Instead of adjusting the weight to the average loss function, we can approximate the gradient by only one random directional derivative of the error. Thus, the number of computation in the gradient descent method can be significantly reduced by a factor of  $n$  (where  $n$  is the number of direction). The stochastic gradient method

(SGM) is given by

$$w^{t+1} = w^t - \alpha_t (\nabla e)_i \quad (3.8)$$

where  $\alpha_t$  is the learning rate at the  $t^{th}$  iteration which may vary with iterations.

As we move closer to the bottom of the minimum of the function; the solution starts to oscillate since we are moving the weight in random directions. Thus, the learning rate should be reduced gradually as we iterate. Some commonly used reduction of learning rate is given by

$$\eta(t) = \eta_i \quad \text{if } t_i \leq t \leq t_{i+1} \quad \text{piecewise constant} \quad (3.9)$$

$$\eta(t) = \eta_0 e^{-\lambda t} \quad \text{exponential decay} \quad (3.10)$$

$$\eta(t) = \eta_0 (\beta t + 1)^{-\alpha} \quad \text{polynomial decay} \quad (3.11)$$

where  $\lambda$ ,  $\beta$ , and  $\alpha$  are known as hyperparameter.

Training dataset can be very large in certain cases which leads to a greater cost of compute at each iteration for the gradient descent, so stochastic gradient descent is preferred in these cases.

## Mini-Batch Stochastic Gradient Descent

In order to make use of the features of both gradient descent and stochastic gradient descent; we introduce the mini-batch stochastic gradient descent. In this approach, instead of iterating in the direction of the full gradient or only in one direction of the full gradient, we split the dataset into small batches and compute the gradient of each batch. The formula of the stochastic gradient descent is given by

$$\mathbf{w}^{t+1} = \mathbf{w}^t - \eta_t \mathbf{g}_t \quad (3.12)$$

$$\mathbf{g}_t = \nabla_{\mathcal{B}_t} e(\mathbf{x}, \mathbf{w})$$

where  $\mathcal{B}_t$  is a mini-batch of elements drawn uniformly at random from the training set (the expectation of the gradient remains unchanged).

Since we are using a mini-batch, the updates are closer to the full gradient but at a reduced cost.

## Newton's Method

For minimizing  $f(x)$ ,  $x \in \mathbb{R}$ , we need to solve  $g(x) = f'(x) = 0$ . Newton's iteration is given by

$$x_{n+1} = x_n - \frac{g(x_n)}{g'(x_n)} \quad (3.13)$$

$$= x_n - \frac{f'(x_n)}{f''(x_n)} \quad (3.14)$$

For multivariate functions we need to minimise  $f(\mathbf{x})$  over  $\mathbf{x} \in \mathbb{R}^n$ , that it

$$\min_{\mathbf{x} \in \mathbb{R}^n} f(\mathbf{x}), \quad \mathbf{x} = (x_1, x_2, \dots, x_n)^T \in \mathbb{R}^n \quad (3.15)$$

The Newton's iteration for multivariate function is given by

$$\mathbf{x}_{n+1} = \mathbf{x}_n - H(\mathbf{x}_n)^{-1} \nabla f(\mathbf{x}_n) \quad (3.16)$$

where  $H(\mathbf{x}_n)$  is the Hessian matrix of  $f(\mathbf{x})$ .

We can observe from (3.3) that calculating the inverse of the Hessian matrix can be computationally very expensive for higher dimensions. Replacing  $H(\mathbf{x}_n)^{-1}$  by  $\alpha \mathcal{I}$  where  $\mathcal{I}$  is the identity matrix; we get the **method of steepest descent** given by

$$\mathbf{x}_{n+1} = \mathbf{x}_n - \alpha \mathcal{I} \nabla f(\mathbf{x}_n) \quad (3.17)$$

where  $\alpha \in (0, 1)$

## Momentum

### Exponentially Weighted Moving Average

The exponentially weighed moving average (EWMA), also known as exponential moving average (EMA), of a series of data points  $S_t$  is given by

$$\mu_t = \beta\mu_{t-1} + (1 - \beta)S_t \quad (3.18)$$

$$\mu_0 = c$$

where  $c \in \mathbb{R}$  and  $\beta \in (0, 1)$  is known as the smoothing constant.  $\beta$  represents the weightage that is going to be assigned to the past values. The average number of previous reading is approximately given by  $n = (1 - \beta)^{-1}$ . The higher the value of  $\beta$  the greater the number of points we average over.

### SGD with Momentum

The momentum method was mentioned in Rumelhart, Hinton and Williams' paper on back-propagation learning in 1986 (Rumelhart et al. 1986). From the SGD, it was observed that the weight update can be very noisy. In order to smoothen the search direction in the SGD, an exponentially moving average is implemented on the gradient. The SGD with momentum can be written in the form

$$\mathbf{w}^{t+1} = \mathbf{w}^t - \eta\boldsymbol{\mu}^t \quad (3.19)$$

where the exponential moving average of the gradient is given by

$$\boldsymbol{\mu}^t = \beta\boldsymbol{\mu}^{t-1} + (1 - \beta)\nabla(e)^t$$

$$\boldsymbol{\mu}_0 = \mathbf{c}$$

## RMSProp

Proposed by Geoffrey Hinton in lecture 6 of the online course "Neural Network for Machine Learning" (Hinton & Tieleman 2012), the Root Mean Square Propagation

(RMSProp) algorithm alleviates undesirable fluctuations when adjusting the weight during optimization. RMSProp is an adaptive learning rate. The idea is to reduce the step size at very large gradient to avoid fluctuations and increase the step size at smaller gradient to move steadily in the correct direction of the optimal solution; hence an adaptive learning rate at each iteration. The equations for the RMSProp is given by

$$\begin{aligned} S_i^t &= \gamma S_i^{t-1} + (1 - \gamma)(\nabla e^t)_i^2 \\ w_i^{t+1} &= w_i^t - \frac{\eta}{\sqrt{S_i^t} + \epsilon} (\nabla e^t)_i \end{aligned} \quad (3.20)$$

where  $\epsilon \approx 10^{-6}$  to ensure that we avoid dividing by zero during iterations.

## Adam

The Adam algorithm (Adaptive Moment Estimation algorithm) is a robust update rule for the weight optimization (Kingma & Ba 2014). The algorithm combines the benefits of momentum (3.19) and RMSProp (3.20). Adam is the most popular generalized algorithm performs very well in many cases. It is considered as a state-of-the-art algorithm for deep neural network optimization. The set of equations for the Adam algorithm is given by

$$\begin{aligned} \mu_i^{(t)} &= \beta_1 \mu_i^{(t-1)} + (1 - \beta_1) \nabla(e^{(t)})_i && \text{Momentum} \\ S_i^{(t)} &= \beta_2 S_i^{(t-1)} + (1 - \beta_2) (\nabla e^{(t)})_i^2 && \text{RMSProp} \end{aligned}$$

To prevent  $S^t$  and  $\mu^t$  from becoming zero during the initial steps, a bias correction is introduced, such that

$$\hat{\mu}^{(t)} = \frac{\mu^t}{1 - \beta_1^t} \quad \hat{S}^{(t)} = \frac{S^t}{1 - \beta_2^t}$$

Finally, the update rule is given by

$$w_i^{(t+1)} = w_i^{(t)} - \frac{\eta}{\sqrt{S_i^t} + \epsilon} \hat{\mu}_i \quad (3.21)$$

The parameters  $\beta_1$ ,  $\beta_2$  and  $\epsilon$  are known as the hyperparameters of the Adam algorithm. A common set of values that works well in literature are  $\beta_1 = 0.9$ ,  $\beta_2 = 0.9999$  and  $\epsilon = 10^{-8}$  (Kingma & Ba 2014).

## 3.2 Backpropagation using Adam

The Adam algorithm for optimization can be introduced in the weight update when training deep neural network. The problem of finding the right set of weights and biases for a deep neural network can be reduced to an optimization problem same as we defined in (3.4). After finding the gradients through the backpropagation algorithm, the set of updates proposed in the Adam algorithm are used to update the weights and biases in the neural network.

---

**Algorithm 3** Backpropagation using Adam optimization with  $n$  total number of inputs for  $N$  epochs

---

```

1: Start
2: for  $t = 1, 2, \dots, N$  do
3:   for  $s = 1, 2, \dots, n$  do
4:     Compute the activation layer  $a^{(1)}$  using  $n$  input values in a batch
5:      $a^{(1)} = \sigma(w^{(1)}x + b^{(1)})$ 
6:     Feed forward the  $n$  input to the next layers
7:     for  $l = 2, 3, \dots, L$  do
8:        $z^{(l)} = w^{(l)}a^{(l-1)} + b^{(l)}$ 
9:        $a^{(l)} = \sigma(z^{(l)})$ 
10:      end for
11:    end for
12:    Compute the average cost function,  $C$ , at the output layer  $L$ 
13:     $c^{(t)} = \frac{1}{2n} \sum_s^n (\hat{y}_s - y_s)^2$ 
14:    Compute the rate of change of the cost function w.r.t to  $z^{(L)}$ 
15:     $\delta^{(L)} = \nabla_{a^{(L)}} C \odot \sigma'(z^{(L)})$ 
16:    for  $l = L - 1, L - 2, \dots, 2$  do
17:      Computing the gradients
18:       $\delta^{(l)} = (w^{(l+1)})^T \sigma^{(l+1)} \odot \sigma'(z^{(l)})$ 
19:       $\nabla_{b^{(l)}} C = \delta^{(l)}$ 
20:       $\nabla_{w^{(l)}} C = \delta^{(l)} a^{(l-1)}$ 
21:
22:      Bias update
23:       $\mu_i^{(t)} = \beta_1 \mu_i^{(t-1)} + (1 - \beta_1) \delta_i^{(l)}$ 
24:       $S_i^{(t)} = \beta_2 S_i^{(t-1)} + (1 - \beta_2) (\delta_i^{(l)})^2$ 
25:       $\hat{\mu}^{(t)} = \frac{\mu^t}{1 - \beta_1^t}$ 
26:       $\hat{S}^{(t)} = \frac{S^t}{1 - \beta_2^t}$ 
27:       $b_i^{(t+1)} = b_i^{(t)} - \frac{\eta}{\sqrt{S_i^t} + \epsilon} \mu_i$ 
28:
29:      Weight update
30:       $\mu_i^{(t)} = \beta_1 \mu_i^{(t-1)} + (1 - \beta_1) \delta^{(l)} a_i^{(l-1)}$ 
31:       $S_i^{(t)} = \beta_2 S_i^{(t-1)} + (1 - \beta_2) (\delta^{(l)} a_i^{(l-1)})^2$ 
32:       $\hat{\mu}^{(t)} = \frac{\mu^t}{1 - \beta_1^t}$ 
33:       $\hat{S}^{(t)} = \frac{S^t}{1 - \beta_2^t}$ 
34:       $w_{ij}^{(t+1)} = w_{ij}^{(t)} - \frac{\eta}{\sqrt{S_i^t} + \epsilon} \mu_i$ 
35:    end for
36:  end for

```

---

# Chapter 4

## Sentiment Analysis

People's opinions, feelings and sentiments towards entities such as products, services, other people, events, news, issues, topics, etc., can be in substantial volume, complex and challenging to be understood and processed by machines and computers. Thus, sentiment analysis, also known as opinion mining, started to popularise along with the rise of social media when a large amount of digital text data were suddenly available for mining. Natural language processing (NLP) helps computers process and understand human-based language to perform a repetitive task. Sentiment analysis is a niche of NLP, and it aims at quantifying the positivity, negativity and/or neutrality of implied or expressed in a given text.

Social media have provided large platforms for people to freely share their opinions and express their views on any subject across various geographical and spatial boundaries. They have also allowed people to connect, influence and be influenced by such opinions and views. Management science researchers studied these interactions in the 1940s and 1950s among people in organizations. Since 2002, with social media, those studies have been performed at grand scales with much data. Thus, advanced sentiment analysis research has been performed in political science, economics, finance and management science as they are heavily dependent on public opinions.

## 4.1 Rule-Based Methods

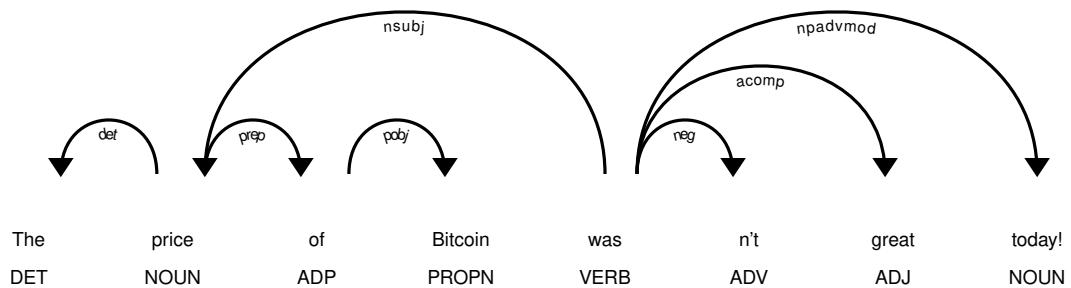
Conventional NLP techniques rely on rules to process textual data to extract opinion, polarity, topic, and other information within the data. Tokenization, part-of-speech tagging, lemmatization and removal of stop words are some rules and techniques used to process data prior to analysis. Tokenization is the separation/breaking down of text data into words, and the space between words in a document is commonly used as the delimiter of separation. Consider the example below:

“The price of Bitcoin wasn’t great today!” (4.1)

Tokenizing the result above results in the list:

[‘The’, ‘price’, ‘of’, ‘Bitcoin’, ‘was’, ‘n’t’, ‘great’, ‘today’, ‘!’] (4.2)

In sentiment analysis, the tokenized sentence would be compared to a predefined list of polarized (positive and negative) words. A naive way to get the polarity of the sentence would be to count the number of positive and negative words in the predefined polarized lists. The limitations are obvious; we are not considering the context in which the words are used nor the preceding words. Part-of-speech tagging refers to the classification of the token in a document based on a predefined assignment. Tokens are classified as adjectives, adverbs, nouns, verbs, etc. (the complete list can be obtained through the Universal POS tags). POS tagging is used to describe the syntactic structuring of a sentence.



**Figure 4.1:** Part-of-speech tagging of sentence at 4.1 using spaCy

Lemmatization refers to the process of reducing words to their base form, also known

as lemma. For example, the words below can be reduced as follows:

great → good

happiness → happy

stemming → stem

It allows mapping the meaning of multiple words at one time by reducing the former to its base. The assumption is that both forms retain the same meaning syntactically. Moreover, the removal of stop words is often performed to eliminate neutral words or those that bring no additional value within a sentence. Examples of such words would be “the”, “a”, “is”, etc. Since rule-based models are performed individually over each word, the simple removal of stop words reduced the computation time of such models. However, such models are minimal, computationally demanding and often inaccurate. Thus, with the rise of computation power available and machine learning, statistical and embedding-based models were developed to tackle those limitations.

## 4.2 Word Embedding

Processing raw data from ruled based method is not ideal. A numerical representation of text data would be more appropriate for mathematical models to perform NLP. Word embedding is the representation of text data as vectors in a vector space. Words with similar meanings would be considered very close to each other in such a space. Moreover, from linear algebra, we would expect that normalised vectors of words with similar meanings to be close. Word embedding techniques are categorised among two types: frequency-based embedding and prediction-based embedding.

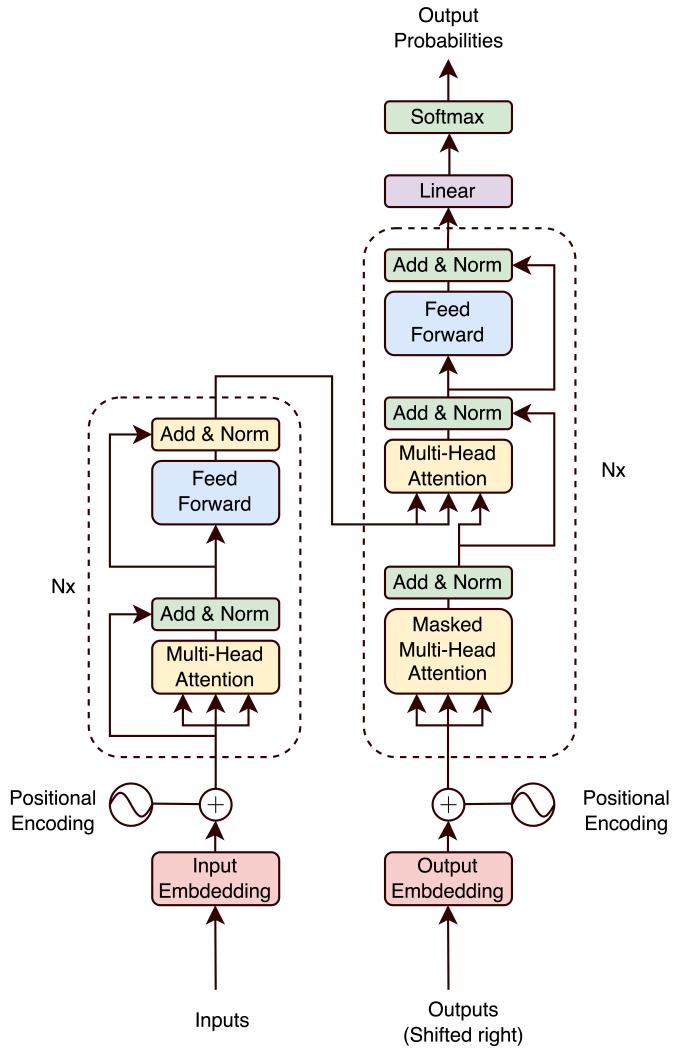
Frequency-based embedding refers to the vectorization of words based on the frequency of occurrence of words in a document. The three frequency-based embedding types are Count Vector, TF-IDF (Term Frequency-Inverse Term Frequency) Vector and Co-Occurrence Matrix with a fixed context window. Such embedding results in

a very sparse vector of high dimensionality which are computationally challenging to manipulate.

Prediction-based embedding predicts a target word based on the context (neighbouring words). Developed by (Mikolov, Chen, Corrado & Dean 2013), the word2vec was introduced. Word2vec is a word embedding method derived from two techniques: CBOW (Continuous Bag of Words) and the Skip-Gram Model. The CBOW attempts to predict the probability of a word in the centre from a given context, while the Skip-Gram model is the opposite of the CBOW model; it tries to predict the context of words given a centre word.

### 4.3 Transformer Model (Attention Mechanism)

Attention in neural networks attempts to emulate the human brain’s cognitive attention, focusing on the only relevant part of the data while neglecting the rest. Developed by a team at Google Brain in 2017 (Vaswani et al. 2017), transformers are the current state-of-the-art attention techniques (instead of recurrence) for processing contextual data; text, audio, video, speech, etc. Like the RNN, LSTM and GRU, transformers process sequential data as well, with the exception that transforms process all input at once, thus allowing parallelization. Transformers consist of two main parts: the encoder and the decoder. The encoder processes the input document identifies the critical text, and creates an embedding for the text based on the latter’s importance to other words in the document. On the other hand, the decoder tries to get the text back from the embedding created by the encoder.



**Figure 4.2:** Transformer Layer (Vaswani et al. 2017)

The left part of the architecture is the encoder, while the right side is the decoder. This structure allows the transformer model for parallelization computations instead of sequential (RNN, LSTM and GRU). Transformer models can now solve natural language processing problems with higher accuracy. We introduce two commonly used transformation models: BERT and RoBERTa.

Bidirectional Encoder Representations from Transformers (BERT) was developed by Google (Devlin, Chang, Lee & Toutanova 2018). BERT uses only the encoder part of the transformer, similar to the original transformer in (Vaswani et al. 2017).

BERT can create word representations that are dynamically influenced by the

surrounding words, whereas word2vec has a fixed representation for each word independent of the context in which it occurs. When it was published, it led BERT to achieve multiple state-of-the-art NLP tasks, including sentiment analysis (Chiorrini, Diamantini, Mircoli & Potena 2021). Several pre-trained BERT models from the unlabeled text are currently available (due to a large number of data and parameters required for training) and can be re-adapted using transfer learning.

RoBERTa, the Robustly Optimized BERT Pretaining Approach, was also developed by Google (Liu, Ott, Goyal, Du, Joshi, Chen, Levy, Lewis, Zettlemoyer & Stoyanov 2019). The approach found that BERT was currently under-trained and could potentially do much better. RoBERTa uses an extensively large amount of hyperparameters, tuning and optimization. Additionally, much more data was also used during training. BERT used 16 Gb of data from CC-news while RoBERTa was trained on 160gb of data from CC-news (92 Gb), OpenWebText (38 Gb), and addition data from Stories (31 Gb) (Liu et al. 2019).

# Chapter 5

## Analysis of Bitcoin Price using Deep Learning Models

Building on the work by Abraham on Cryptocurrency Price Prediction Using Tweet Volumes and Sentiment Analysis (Abraham et al. 2018), we gathered data from Twitter, Google Trends and Yahoo Finance about Bitcoin. That is, Tweets and Tweet Volume from Twitter, Search Value Index (SVI) from Google Trend, and Open, High, Low and Close value of Bitcoin hourly. We first explore the raw data gathered for any observable trend. Then, we perform feature engineering by cleaning the data, generating sentiment scores of Tweets using the state-of-art model roBERTa, identifying potential influencers using K-Means clustering, performing hourly aggregations and finally scaling the data. We then modelled the resulting dataset in two different ways. We used an LSTM NN model to predict the Bitcoin closing prices for the next 24hr.

### 5.1 Data Collection

In order to tackle the problem of prediction of Bitcoin prices, we gathered a large amount of data from different sources, which we believe might contribute information to predict the price. We first consider the sentiment analysis of Tweets from Twitter as our first input. We then get Bitcoin’s Search Value Index (SVI) from Google Trends. Finally, we retrieved Bitcoin market prices from Yahoo Finance. All data

are scrapped between 01 June 2022 and ending 13 July 2022.

## Twitter

In order to get Tweets related to Bitcoin from Twitter, we leverage Tweepy - an open-source Python Library for accessing the Twitter API. The API has different access levels: Essential, Elevated and Academic Research. We requested Academic Research access through the Twitter Developers Portal to get the appropriate API key and API token. Once access was provided, we used the query features from the documentation to scrape Tweets with either the word “bitcoin” or hashtag bitcoin (#bitcoin). For each query, we collected the following information: username, user description, location, friends count, followers count, total tweets count, retweet count, hashtags in the tweets and the tweet.

```
1     query = 'bitcoin OR #bitcoin lang:en -is:retweet'
```

**Listing 5.1:** Tweepy query for bitcoin, #bitcoin and excluding retweets

The request cap for academic access is 100 per 15 minutes. We set up our scrapping strategy for 10 consecutive days at a rate of 65 Tweets a minute to match the request cap. This strategy allowed us to scrap 814,818 tweets with the query mentioned above. Additionally, we used the same access to get the tweet volume pertaining to the query in listing 5.1 and predefined start time and end time.

```
1     query_params = {'query': query, 'granularity': 'hour',  
2                     'start_time': start_time, 'end_time': end_time}
```

**Listing 5.2:** Tweepy query to get tweet count withing each hour between predefined start time and end time

## Google Trends

Google is a massive search engine dealing with a vast amount of data daily, including the volume of searches of people based on keywords. Google trend data is a reflection of the volume of searches that people do. However, the data is extensive to be consumed directly due to the number of global users. The trend data is

anonymised (personally identifiable information removed), categorised (topic associated with each search query) and aggregated (grouped). The trend data provided by Google are indexed to 100, where 100 is the maximum search interest for the time and location selected. The combination of the pre-processing of raw volume searches provides a measure of interest in a particular topic across different regions or globally. We used the pytrends package - an unofficial open-source Python Library for accessing the Google trend data. We pass the least ambiguous keyword, “bitcoin”, into the library and retrieve the hourly search index for the said keyword.

```

1 kw_list = ['bitcoin']
2 search_df = pytrends.get_historical_interest(kw_list, year_start
=2022, month_start=6, day_start=1, hour_start=0, year_end=2022,
month_end=7, day_end=16, hour_end=0, cat=0, geo='', gprop='')
```

**Listing 5.3:** pytrend query to get google trend index within each hour between predefined start time and end time

## Yahoo Finance

In order to get Bitcoin prices, we used the yfinance package - an unofficial open-source Python Library for accessing market data on cryptocurrencies, regular currencies, stocks and bonds, fundamental and options data, and market analysis and news. We used the yfinance package to scrape the open, high, low and close prices and the volume of bitcoin traded hourly.

```

1 BTC_Ticker = yf.Ticker("BTC-USD")
2 BTC_Data_long = BTC_Ticker.history(start="2022-06-01", end="
2022-07-16", interval= '1h')
```

**Listing 5.4:** yfinance query to get bitcoin price and traded volume within each hour between predefined start time and end time

We briefly explore the raw data scrapped from the different sources to get a better understanding of the latter. The tweets from Twitter had several issues. A large number contained URLs, symbols, excessive punctuations and were in different languages (even if the query was restricted to English tweets only).

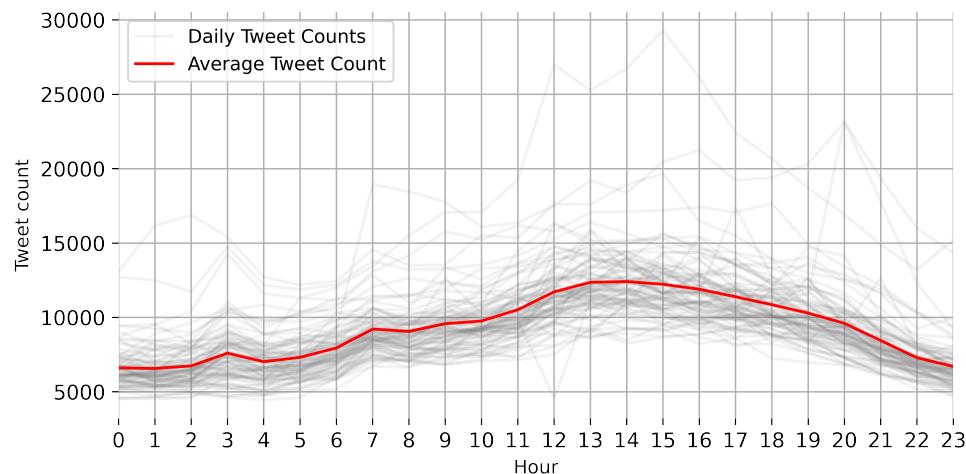
```

 The Psychonaut Ape Division 
"TOO WEIRD TO LIVE AND TOO RARE TO DIE"
https://t.co/GpWKXTr68J
 A bold collection of 7,777 Psychonaut Apes by Internationally recognised artist WoahJonny
NFT NFTS Ethereum ETH ETH crypto solana BNB avax Tezos Tron Bitcoin BTC https://t.co/0n1tkWwa4n

```

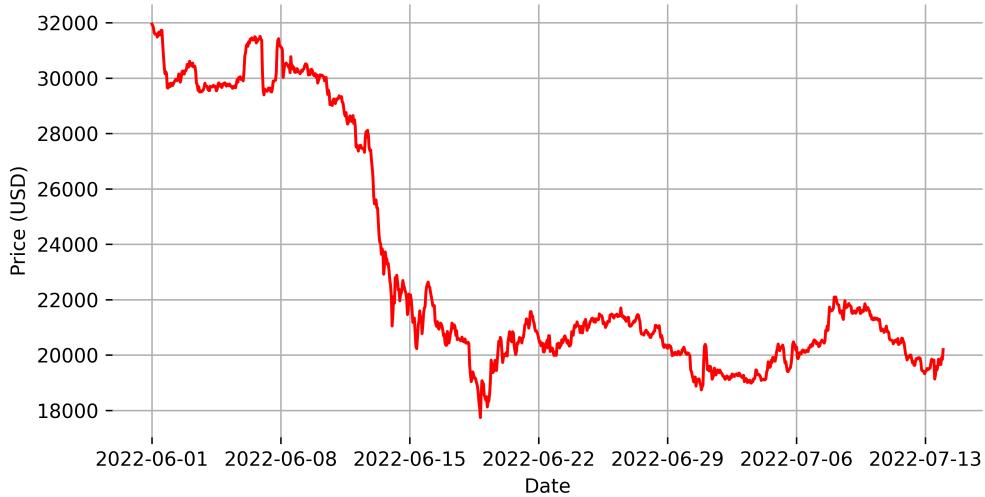
**Figure 5.1:** Example of raw tweet scraped using the tweepy package through Python

The number of tweets at the end of each hour was continuous and had no major issues from the API response.



**Figure 5.2:** Distribution of tweet count over each day at the end of every hour over 24 hours

The data for the open, high, low, and close prices scrapped from Yahoo Finance was clean and had no issues. However, we did see that not all closing hours have the associated volume. Thus, volume was not included in the final dataset.



**Figure 5.3:** Bitcoin closing price over every hour for everyday

## 5.2 Feature Engineering

Feature engineering is the manipulation of our dataset to provide extra features to input in our model to improve accuracy. Manipulation includes addition, deletion, combination and transformation of the dataset. Practical feature engineering is dependent on the business problem and our objective. Common types of feature engineering include scaling and transformation, fitting missing values, feature coding, feature construction and feature extraction.

### Sentiment Analysis on Tweet Data

In order to be able to quantify the tweets as features in our model, we transform the tweets into numerical values by performing sentiment analysis on them. However, as seen in the tweet example in listing (5.1), the raw tweet would be difficult for the existing natural language processing model to transform them. Using regular expression syntax in Python, we cleaned the tweets first.

```

1  def pre_process_tweet (text):
2      #Remove all links starting with http...
3      text = re.sub('https?:\/\/.*[\r\n]*', '', text)
4      #Remove RT

```

```

5     text = re.sub('^\RT[\s]+', '', text)
6     #Remove @[User]
7     text = re.sub('@[^ ]+', '', text)
8     #Remove Punctuation first
9     text = text.translate(str.maketrans('', '', string.punctuation))
10    #Convert text to lower case
11    text = text.lower()
12    #Remove new line
13    text = re.sub('\n', ' ', text)
14    #Replace emojis with description
15    text = demoji.replace_with_desc(text, sep=' ')
16    #Reducing whitespaces to one everywhere
17    text = re.sub('\s+', ' ', text)
18    return text

```

**Listing 5.5:** Python function to clean Tweets using RegEx (regular expression)

Applying the python function defined in listing 5.5 to the raw tweet in 5.1 leads to the following string:

'mushroom the psychonaut ape division mushroom too weird to live and too rare to die framed picture a bold collection of psychonaut apes by internationally recognised artist woahjonny nft nfts ethereum eth eth crypto solana bnb avax tezos tron bitcoin btc' (5.1)

We apply the pre-process tweet function across the 814,818 tweets that were scraped. Thus, making the tweets easier to process in sentiment analysis models. Once the tweets are clean, we use the RoBERTa model defined in Section 4.3 to perform sentiment analysis. We used the version *cardiffnlp/twitter-roberta-base-sentiment* which is made available Hugging Face. 'Twitter-roBERTa-base' for Sentiment Analysis is a pretrained on approximately 58 million tweets and fine-tuned for sentiment analysis (Barbieri, Camacho-Collados, Neves & Anke 2020). The output of the model are is the percentage of a tweet being negative, neural and positive.

```

1     encoded_text = tokenizer(preprocessed_tweet, return_tensors='pt'
2     )

```

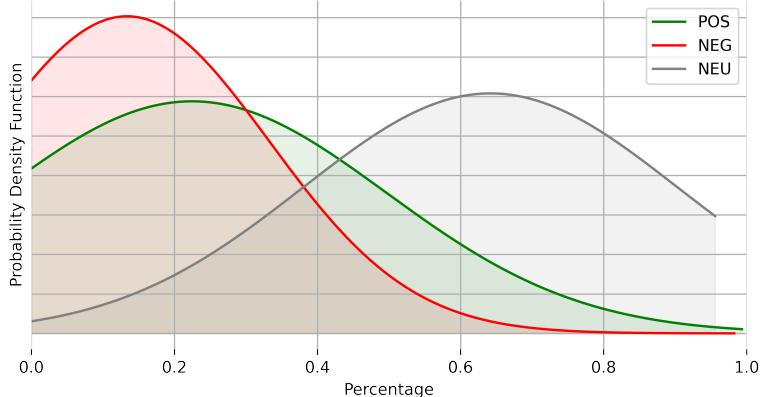
```

2     output = model(**encoded_text)
3     scores = output[0][0].detach().numpy()
4     scores = softmax(scores)
5     scores_dict = {'roberta_neg':scores[0], 'roberta_neu':scores
6     [1], 'roberta_pos':scores[2]}
7     print(scores_dict)
8
>> {'roberta_neg':0.298603, 'roberta_neu':0.5974009, 'roberta_pos':
: 0.10399604}

```

**Listing 5.6:** Applying the twitter-roberta-base-sentiment model to the preprocessed tweet in (5.1)

We apply the same function through the whole preprocessed dataset to get the scraped tweets' scores. Tweets that were too long or had unknown characters that the model could not process were dropped (less than 0.5% of the tweet dataset).

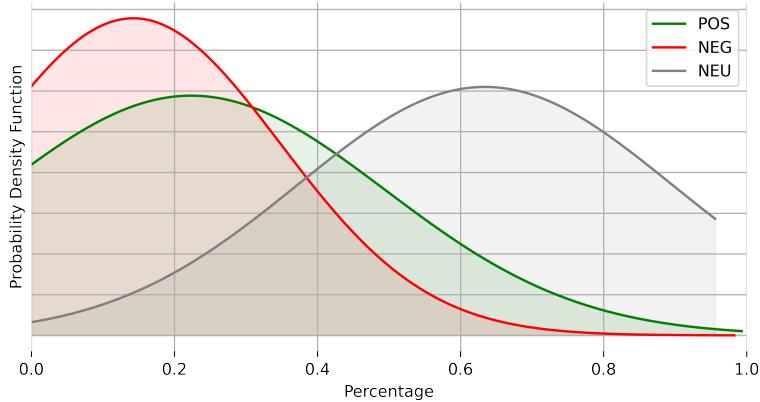


**Figure 5.4:** Sentiment score distribution using roBERTa model on all tweets

## Data Encoding

Data encoding involves choosing a set of symbolic values to represent different categories. The symbolic values can take input a single or multiple columns and produce a category on top of the dataset. For example, indicating whether the collected data on a holiday can be coded as 1 and 0 otherwise. In our case, we attempt to separate the influencers from ordinary people by studying the total number of followers of the user that tweeted about Bitcoin. Using KMeans clustering, we group users based on

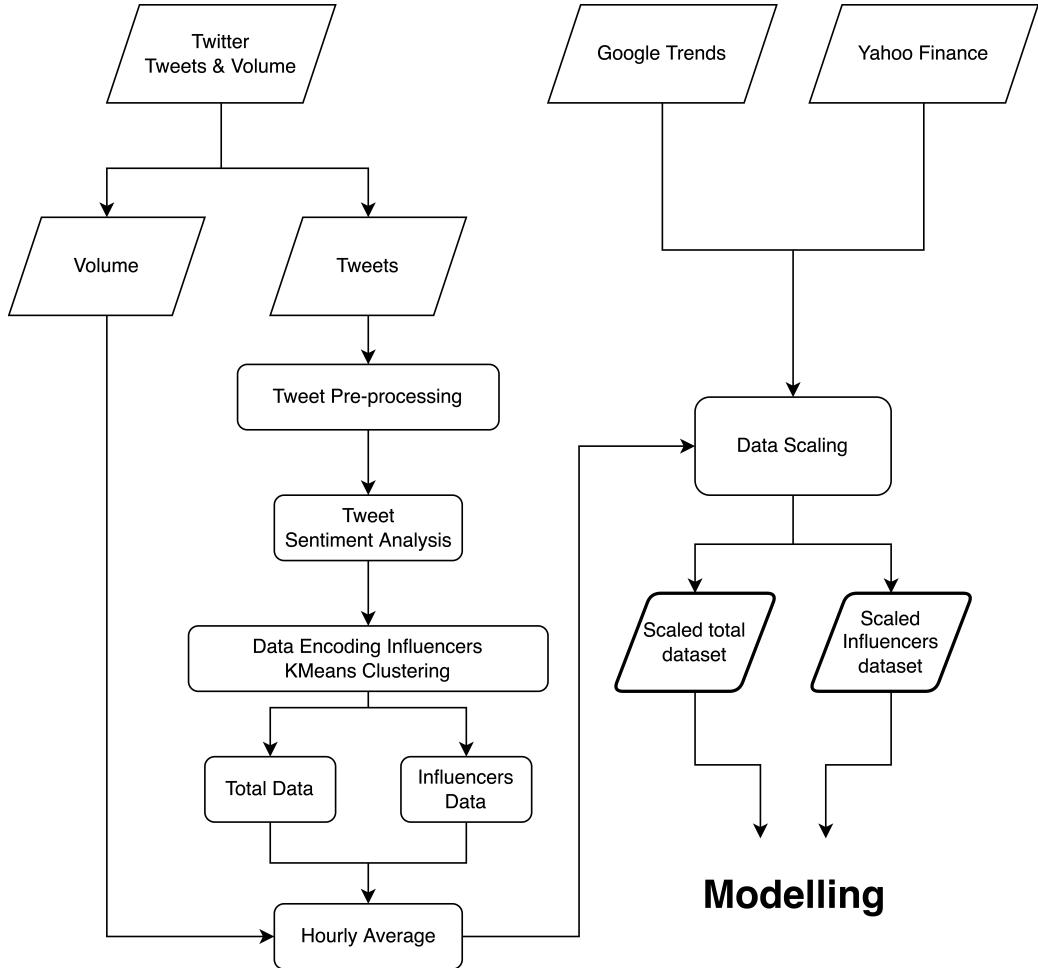
their number of followers. We then excluded users within the group with the least number of followers to get only "influencer" users.



**Figure 5.5:** Sentiment score distribution of Twitter influencers (grouped using KMeans clustering) using roBERTa model on all tweets

## Scaling

Scaling refers to the adjustment of large dataset to ease the learning process and also prevent large values to dominate when building a predictive model. Processing the point data of all the scrapped tweets can be overly demanding in terms of computational power. This may result to an overfitting and not being able to model the general trend of a single day. Additionally, we do not have point data for price and google trends search index. To overcome this problem, we aggregated the sentiment scores (POS, NEU, NEG) over each hour across the time period. The resulting dataset was  $1032 \times 3$  (24 hourly intervals for 43 days). Moreover, our dataset consists of various data sources with overly different means. In order to prevent any feature to dominate the dataset, we scaled our features using the min/max feature scaling so that our values fall between -1 and 1 for all features.

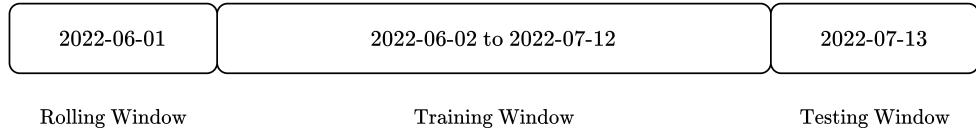


**Figure 5.6:** Feature engineering pipeline

### 5.3 Modelling Bitcoin Price using LSTM NN

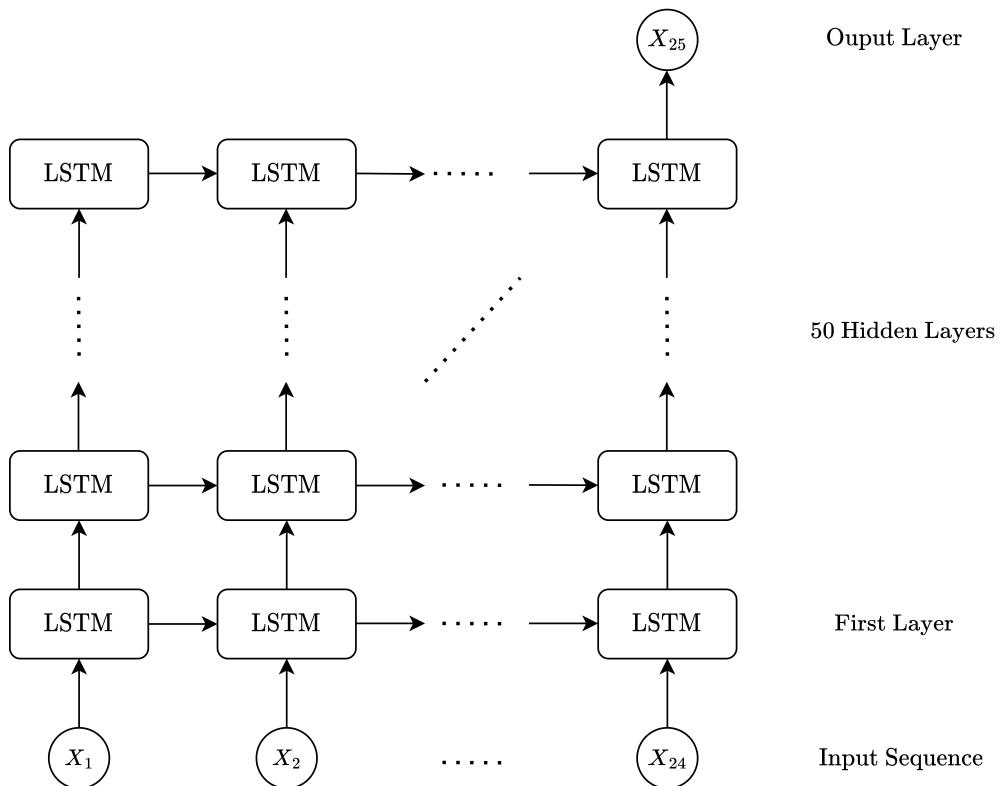
In order to predict the price of Bitcoin for the next 24 hours based on the information from the past day, we need to use sequential models. We used a Long-Short-Term-Memory neural network model for prediction because of its ability to retain long- and short-term information. Using the feedback loop in the architecture, we can model dependencies within the data, and the hidden cell allows for a generalisation of the data for the window (24 hours in our case). We used the scaled preprocessed from the previous section with a rolling forecast of 1 day (24 hours) to predict the next day (24 hours). The training dataset contained 41 days (42, if the rolling window is included). The latter comprises 6 features from each hour: tweet volume, Google trend search index, average tweet sentiment (positive, negative, neutral), and price

of Bitcoin, and the label (target) was the price of Bitcoin in the next hour.



**Figure 5.7:** Train-test split

After experimentation, we found that using an LSTM NN with 50 hidden states, 250 epochs and a learning rate  $\alpha = 0.00001$  converges. The loss was calculated using the mean squared error within the network. We used the Adam optimizer to train the network faster.

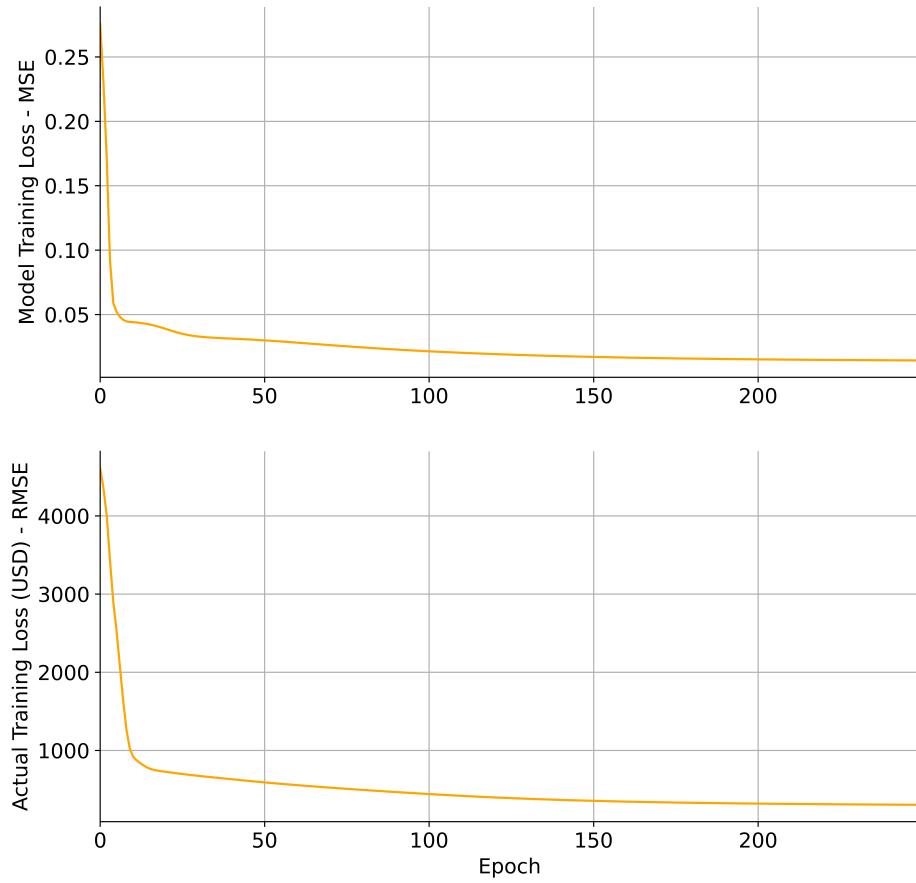


**Figure 5.8:** LSTM architecture with 50 hidden states, input sequence of 25 and 6 input features

## 5.4 Results

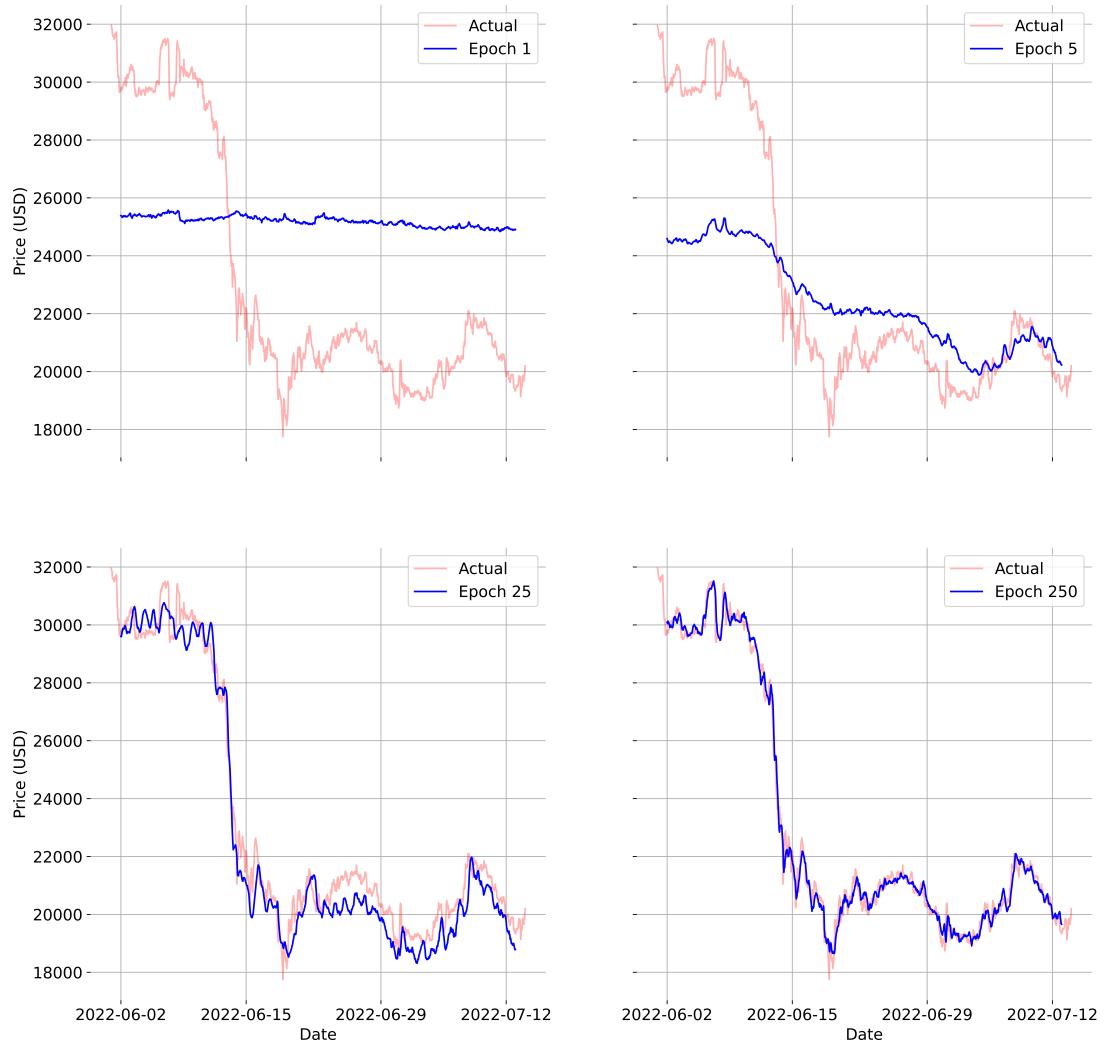
The training is set to run for 250 epochs which took around 1 hour to complete. We observed that the MSE in the model training was down to 0.0146 (ADD REF

FIGURE). However, it is not interpretable since the data was scaled when processed in the model. The training at each epoch is extracted and the inverse transform is performed. We then computed the RMSE (for interpretability) for the training and observed that it was \$303.96 after 250 epochs. The testing error for the next 24 hours was \$325.96.



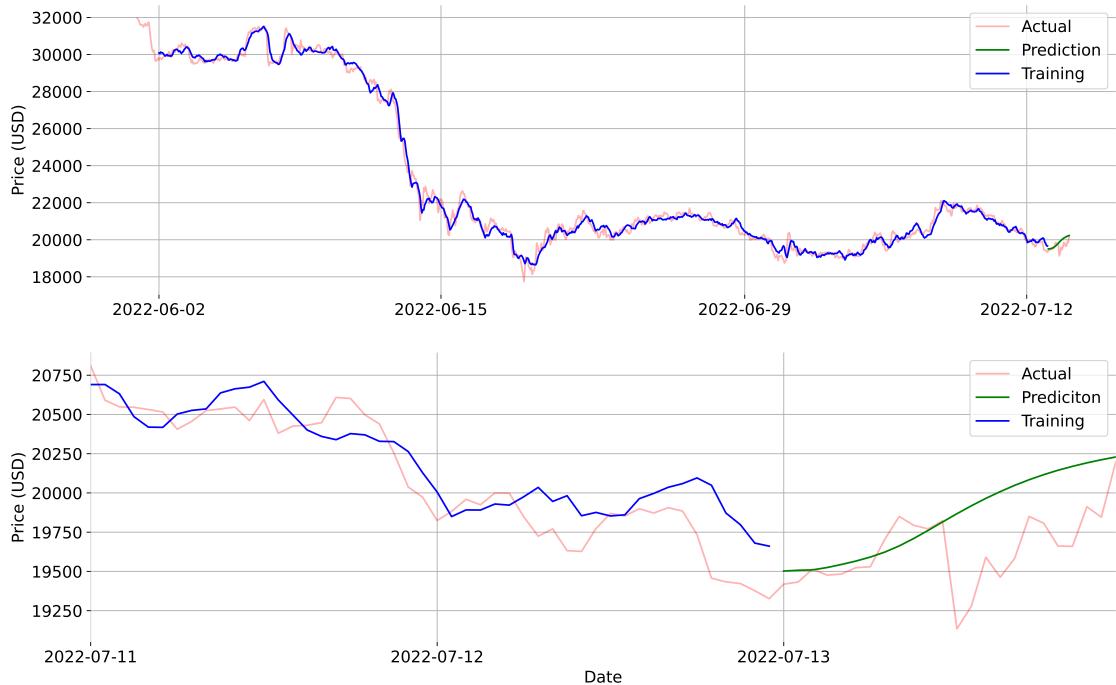
**Figure 5.9:** Model training loss (MSE) and actual training loss in USD (RMSE) over 250 epochs of training

We also noticed that after only 25 epochs, the general profile of the Bitcoin price is replicated by the model.



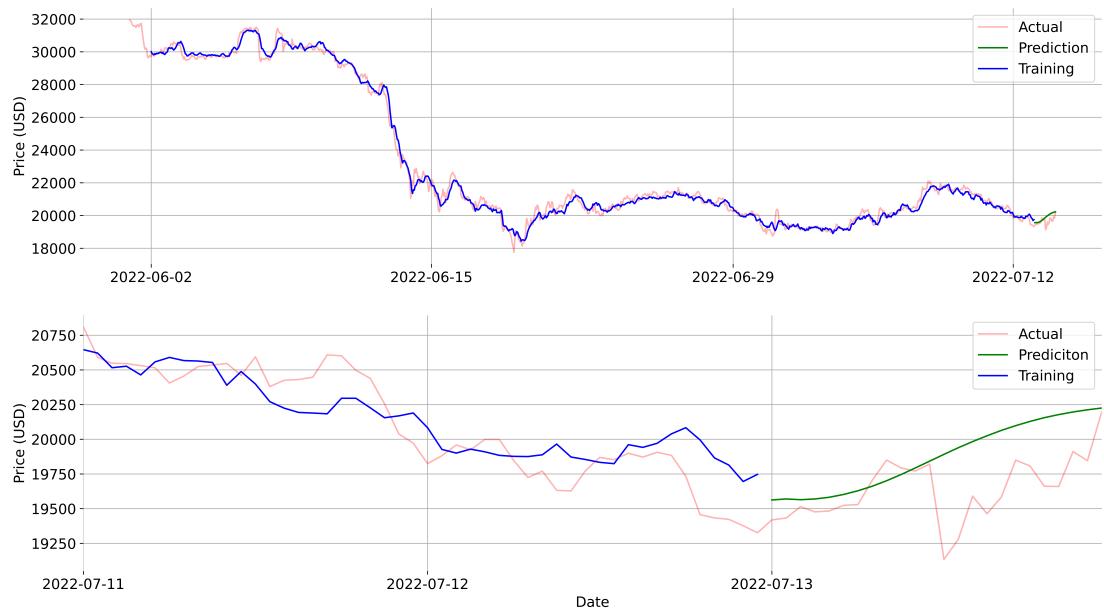
**Figure 5.10:** Model evolution during training for epochs 1, 5, 25 and 250

After training for 250 epochs, we used the model to calculate a rolling window forecast on the price for the next 24 hours. We observe that the general profile of the price movement is modelled but sudden dips and rise in price were not replicated during testing.



**Figure 5.11:** Trained model and rolling window forecast for Bitcoin price for next 24 hours (using whole dataset)

We ran a different experiment with the limited dataset containing only influencers but did not observe any major improvement from the previous experiment. On the contrary, we noticed a slight increase in RMSE in training and testing.



**Figure 5.12:** Trained model and rolling window forecast for Bitcoin price for next 24 hours (using influencers' dataset)

	Exp.1 Complete Dataset	Exp.2 Influencers' Dataset
Model Training MSE	0.0146	0.0209
Actual Training RMSE (\$)	303.96	317.45
Actual Testing RMSE (\$)	325.96	335.99

**Table 5.1:** Summary of error results from model using the complete dataset and the influencers' dataset using the same training and predicting scenario ( $\alpha = 0.00001$ ).

Since the influencer's dataset does not add value to the model, we dropped the latter. We repeat the experiment using every combination of feature to identify any potential delimiter. Let the features of the model be indexed as such:

$$\begin{aligned} \text{Tweet Count} &\rightarrow (1) \\ \text{Google Trend Index} &\rightarrow (2) \\ \text{Sentiment (POS, NEU, NEG)} &\rightarrow (3) \end{aligned}$$

	Exp.1 (1)(2)(3)	Exp.2 (1)(2)	Exp.3 (1)(3)	Exp.4 (2)(3)	Exp.5 (1)	Exp.6 (2)	Exp.7 (3)
Epochs	250	100	250	250	100	100	100
Training Time (s)	1829	681	2038	1690	653	625	800
Model Training MSE	0.0146	0.0034	0.0162	0.0104	0.0077	0.0053	0.0077
Actual Training RMSE (\$)	303.96	380.27	296.33	322.89	395.73	358.27	395.75
Actual Testing RMSE (\$)	325.96	306.14	189.66	263.79	398.24	302.95	398.24

**Table 5.2:** Summary of error results from model using different feature combinations under the same training and predicting scenario ( $\alpha = 0.00001$ ).

# Chapter 6

## Conclusion and Future Works

This study investigated how we can leverage deep learning models and sentiment analysis on tweets to predict the value of Bitcoin in the next hour.

We reviewed the basics and derived the backpropagation algorithm of feed-forward neural networks. An application of the backpropagation algorithm for feed-forward neural networks was derived. Moreover, we investigated the literature and defined different architectures of sequential neural networks used in problems involving sequential data. The literature on optimization algorithms was also reviewed. We looked at the application of the Adam optimizer in the backpropagation algorithm, the current state-of-the-art algorithm for the optimization problem. We also briefly examined sentiment analysis literature using traditional methods and the latest deep learning models involving transformers for natural language processing, roBERTa (Robustly Optimized BERT Pre-training Approach).

Finally, we defined and coded a whole pipeline from scrapping data to predicting the value of Bitcoin in the next hour using the LSTM NN model. Scrapping, cleaning and feature engineering raw data is tedious. The RMSE for the training and testing on the whole dataset were \$303.96 and \$325.96, respectively. We saw a slight decrease in performance when using only the influencers' data which might be due to the naive way we used to identify influencers: by only considering their number of followers. Furthermore, deep learning models are not easily interpretable due to their BlackBox-like architecture. We experimented with different feature combination and found that the Tweet Count and Sentiment (POS, NEU, NEG) had

the lowest RMSE during training and testing under similar training and predicting scenario.

As a continuity to this work, it would be interesting to compare our model to traditional statistical models. Additionally, we could investigate the features to be added in the LSTM NN model and/or identify the dominant features to reduce our dataset (potentially decreasing training time).

# Bibliography

- Aalborg, H. A., Molnár, P. & de Vries, J. E. (2019), ‘What can explain the price, volatility and trading volume of bitcoin?’, *Finance Research Letters* **29**, 255–265.
- URL:** <https://www.sciencedirect.com/science/article/pii/S1544612318302058>
- Abraham, J., Higdon, D. W., Nelson, J. & Ibarra, J. (2018), Cryptocurrency price prediction using tweet volumes and sentiment analysis.
- Asur, S. & Huberman, B. (2010), ‘Predicting the future with social media’, *Proceedings - 2010 IEEE/WIC/ACM International Conference on Web Intelligence, WI 2010* **1**.
- Banerjee, A. K., Akhtaruzzaman, M., Dionisio, A., Almeida, D. & Sensoy, A. (2022), ‘Nonlinear nexus between cryptocurrency returns and covid-19 news sentiment’, *Journal of Behavioral and Experimental Finance* **36**, 100747.
- URL:** <https://www.sciencedirect.com/science/article/pii/S2214635022000703>
- Bar-Haim, R., Dinur, E., Feldman, R., Fresko, M. & Goldstein, G. (2011), Identifying and following expert investors in stock microblogs, in ‘Proceedings of the 2011 Conference on Empirical Methods in Natural Language Processing’, Association for Computational Linguistics, Edinburgh, Scotland, UK., pp. 1310–1319.
- URL:** <https://aclanthology.org/D11-1121>
- Barbieri, F., Camacho-Collados, J., Neves, L. & Anke, L. E. (2020), ‘Tweeteval: Unified benchmark and comparative evaluation for tweet classification’, *CoRR*

**abs/2010.12421.**

**URL:** <https://arxiv.org/abs/2010.12421>

Bengio, Y., Simard, P. & Frasconi, P. (1994), ‘Learning long-term dependencies with gradient descent is difficult’, *IEEE Transactions on Neural Networks* **5**(2), 157–166.

Bermingham, A. & Smeaton, A. (2011), On using Twitter to monitor political sentiment and predict election results, in ‘Proceedings of the Workshop on Sentiment Analysis where AI meets Psychology (SAAIP 2011)’, Asian Federation of Natural Language Processing, Chiang Mai, Thailand, pp. 2–10.

**URL:** <https://aclanthology.org/W11-3702>

Chiorrini, A., Diamantini, C., Mircoli, A. & Potena, D. (2021), Emotion and sentiment analysis of tweets using bert.

Devlin, J., Chang, M., Lee, K. & Toutanova, K. (2018), ‘BERT: pre-training of deep bidirectional transformers for language understanding’, *CoRR* **abs/1810.04805**.

**URL:** <http://arxiv.org/abs/1810.04805>

Dyhrberg, A. H. (2016), ‘Hedging capabilities of bitcoin. is it the virtual gold?’, *Finance Research Letters* **16**, 139–144.

**URL:** <https://www.sciencedirect.com/science/article/pii/S1544612315001208>

Hinton, G. & Tieleman, T. (2012), ‘Lecture 6.5-rmsprop: Divide the gradient by a running average of its recent magnitude.’, COURSERA: Neural Networks for Machine Learning, 4, 26-30.

Hochreiter, S. & Schmidhuber, J. (1997), ‘Long Short-Term Memory’, *Neural Computation* **9**(8), 1735–1780.

**URL:** <https://doi.org/10.1162/neco.1997.9.8.1735>

Katsiampa, P. (2017), ‘Volatility estimation for bitcoin: A comparison of garch models’, *Economics Letters* **158**, 3–6.

**URL:** <https://www.sciencedirect.com/science/article/pii/S0165176517302501>

- Kingma, D. P. & Ba, J. (2014), ‘Adam: A method for stochastic optimization’.  
**URL:** <https://arxiv.org/abs/1412.6980>
- Kolen, J. F. & Kremer, S. C. (2001), *Gradient Flow in Recurrent Nets: The Difficulty of Learning LongTerm Dependencies*, pp. 237–243.
- LeCun, Y., Bengio, Y. & Hinton, G. (2015), ‘Deep learning’, *Nature* **521**(7553), 436–444.
- Liu, Y., Ott, M., Goyal, N., Du, J., Joshi, M., Chen, D., Levy, O., Lewis, M., Zettlemoyer, L. & Stoyanov, V. (2019), ‘Roberta: A robustly optimized BERT pretraining approach’, *CoRR* **abs/1907.11692**.  
**URL:** <http://arxiv.org/abs/1907.11692>
- McCulloch, W. S. & Pitts, W. (1943), ‘A logical calculus of the ideas immanent in nervous activity’, *The bulletin of mathematical biophysics* **5**(4), 115–133.
- Mikolov, T., Chen, K., Corrado, G. & Dean, J. (2013), ‘Efficient estimation of word representations in vector space’.  
**URL:** <https://arxiv.org/abs/1301.3781>
- Nakamoto, S. (2009), ‘Bitcoin: A peer-to-peer electronic cash system’, *Cryptography Mailing list at https://metzdowd.com* .
- Robbins, H. & Monro, S. (1951), ‘A Stochastic Approximation Method’, *The Annals of Mathematical Statistics* **22**(3), 400 – 407.  
**URL:** <https://doi.org/10.1214/aoms/1177729586>
- Rosenblatt, F. (1958), ‘The perceptron: a probabilistic model for information storage and organization in the brain.’, *Psychol Rev* **65**(6), 386–408.
- Rumelhart, D. E., Hinton, G. E. & Williams, R. J. (1986), ‘Learning representations by back-propagating errors’, *Nature* **323**(6088), 533–536.
- Schuster, M. & Paliwal, K. (1997), ‘Bidirectional recurrent neural networks’, *IEEE Transactions on Signal Processing* **45**(11), 2673–2681.

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, L. & Polosukhin, I. (2017), ‘Attention is all you need’.

**URL:** <https://arxiv.org/abs/1706.03762>

Zhang, J. & Zhang, C. (2022), ‘Do cryptocurrency markets react to issuer sentiments? evidence from twitter’, *Research in International Business and Finance* **61**, 101656.

**URL:** <https://www.sciencedirect.com/science/article/pii/S0275531922000447>

Zhang, X., Fuehres, H. & Gloor, P. A. (2011), ‘Predicting stock market indicators through twitter “i hope it is not as bad as i fear”’, *Procedia - Social and Behavioral Sciences* **26**, 55–62. The 2nd Collaborative Innovation Networks Conference - COINs2010.

**URL:** <https://www.sciencedirect.com/science/article/pii/S1877042811023895>