

Predicting Future Stock Price with Sentiment Analysis: Recurrent vs. Attention Based Learning for Regression Tasks



UNIVERSITY OF THE
WITWATERSRAND,
JOHANNESBURG

Bernard McDonald

Supervisor: Dr. Justine Nasejje

School of Statistics and Actuarial Science
University of Witwatersrand

This dissertation is submitted for the degree of
Master of Science

August 2023

Declaration

I declare that this dissertation is my own, unaided work. It is being submitted for the Degree of Master of Science at the University of the Witwatersrand, Johannesburg. It has not been submitted before for any degree or examination at any other University.



(Signature of candidate)

_____ 21st _____ day of _____ August _____ 20 _____ 23 _____ at _____ Johannesburg _____

Bernard McDonald

August 2023

Abstract

Stock price prediction is a lucrative challenge as successful prediction could yield significant profits for investors – attracting research utilising novel data sources and modelling techniques. This research aimed to accurately predict the future closing price of the top five stocks of the NASDAQ100 index by leveraging Twitter data and recent advancements in machine learning. Three representations of large-scale Twitter data were derived: company, stock market, and general public sentiment. Company sentiment and stock market sentiment were Granger-causal ($p < 0.10$) for the closing price of four and two of the five companies considered, respectively. Five stock price prediction models were built: ARIMA, RNN, LSTM, GRU, and a novel Transformer model. A hyperparameter grid search selected feature subsets containing sentiment data as optimal in sixteen of the twenty (80%) model-dataset combinations fitted. Assessed using the RMSE, all the machine learning models outperformed the ARIMA model. The attention-based Transformer model outperformed the recurrent models in both predictive performance and model computational training efficiency. The model produced test RMSEs of 1.22, 2.07, 35.54, 16.61, and 4.95 when predicting the closing price of Apple, Microsoft, Amazon, Alphabet, and Facebook respectively.

Acknowledgements

I would like to thank my parents, Marchand and Dorothea McDonald, for their love and support throughout my academic endeavours.

I would like to thank my supervisor, Dr. Justine Nasejje – this project would not have been possible without your invaluable input. Furthermore, I would like to thank Dr. Chantelle Clohessy and Dr. Warren Brettenny – your passion sparked my interest for mathematical statistics and for that I am forever grateful.

This research was funded by the Wits Post Graduate Merit Award.

Table of Contents

List of Figures	viii
List of Tables	xi
List of Symbols	xiii
1 Introduction	1
1.1 Introduction	1
1.2 Problem Statement	2
1.3 Aims and Objectives	3
2 Literature Review	5
2.1 Theoretical Background	5
2.1.1 Auto-Regressive Integrated Moving Average Model	6
2.1.2 Model Structure	7
2.1.2.1 Artificial Neural Network	7
2.1.2.2 Recurrent Neural Network	12
2.1.2.3 Long Short-Term Memory	14
2.1.2.4 Gated Recurrent Unit	17
2.1.2.5 Transformer	20
2.1.3 Model Training	34
2.1.3.1 Artificial Neural Network	34
2.1.3.2 Recurrent Neural Network	39
2.1.3.3 Transformer	43
2.2 Methodological Background	46
2.2.1 Sentiment Analysis	46
2.2.2 Stock Price Prediction using Machine Learning	49
2.2.3 Attention-Based Regression Models	53

3	Data	56
3.1	Technical Data	57
3.2	Sentiment Data	60
3.2.1	Twitter Data	61
3.2.2	Sentiment Analysis	65
3.2.3	Levels of Sentiment	67
4	Methodology	76
4.1	ARIMA Model	76
4.2	Machine Learning Models	77
4.2.1	Pre-Processing	77
4.2.2	Training Examples	80
4.2.2.1	Recurrent Models	80
4.2.2.2	Transformer	82
4.2.3	Modelling	86
4.2.3.1	Callbacks	86
4.2.3.2	Hyperparameters	87
4.2.3.3	Hyperparameter Tuning	89
4.2.4	Post-Processing	90
4.2.5	Evaluation	91
5	Results	94
5.1	Model Performance	94
5.2	Sentiment Analysis	101
5.3	Model Computational Training Efficiency	108
6	Discussion and Conclusion	115
6.1	Discussion	115
6.2	Conclusion	119
7	Future Recommendations	121
	References	124
A	Data	129
A.1	Technical Data	129
A.2	Sentiment Data	132
A.2.1	Twitter Data	132

A.2.2	Levels of Sentiment	139
B	Results	141
B.1	Model Performance	141

List of Figures

2.1	A biological neuron	8
2.2	An artificial neuron	8
2.3	Layers of an Artificial Neural Network	9
2.4	An example highlighting how neurons interact between two layers	10
2.5	A feed-forward Artificial Neural Network	12
2.6	A Recurrent Neural Network	12
2.7	An unfolded Recurrent Neural Network	13
2.8	An LSTM cell	14
2.9	A gate used to regulate learning in an LSTM cell	15
2.14	A GRU cell	18
2.18	A Transformer	21
2.19	Positionally encoded input embeddings	23
2.20	An encoder	24
2.21	The self-attention mechanism	25
2.22	The query, key and value projections in the self-attention mechanism . .	26
2.23	A score matrix in the self-attention mechanism	26
2.24	A residual block	30
2.25	A decoder	31
2.26	A look-ahead mask applied to a score matrix	33
2.27	The function and variable dependencies in a feed-forward Artificial Neural Network	36
2.28	Stochastic Gradient Descent in a 3D plane without mini batching	38
2.29	Stochastic Gradient Descent in a 3D plane with mini batching	38
2.30	The function and variable dependencies in a Recurrent Neural Network .	39
2.31	Truncated Backpropagation Through Time in an unrolled Recurrent Neural Network	43

3.1	A plot of each technical feature in the Facebook dataset	57
3.2	The closing price difference for the Facebook dataset	59
3.3	The sentiment analysis pipeline used to process the Twitter data	61
3.4	An example of a tweet in JavaScript Object Notation	63
3.5	The most popular cashtags identified in the Twitter data	64
3.6	The additional stock terms identified for the considered companies	65
3.7	An example of scoring the sentiment of a tweet	66
3.8	The daily sentiment scores of the Apple dataset	68
3.9	The daily sentiment scores for the stock market	68
3.10	The daily sentiment scores for the general public	69
4.1	The training, validation, and testing partitions for the closing price of the Amazon dataset	78
4.2	A training example illustrated on the Transformer architecture	84
5.1	The closing price difference predictions for the Apple dataset	98
5.2	The closing price predictions for the Apple dataset	98
5.5	The closing price difference predictions for the Amazon dataset	100
5.6	The closing price predictions for the Amazon dataset	100
5.9	The Transformer model closing price difference predictions for the Apple dataset	104
5.10	The Transformer model closing price predictions for the Apple dataset	104
5.13	The GRU model closing price difference predictions for the Amazon dataset	106
5.14	The GRU model closing price predictions for the Amazon dataset	107
5.17	A box and whisker plot of the number of training parameters per model grid search	110
5.18	A box and whisker plot of the number of training epochs per model grid search	112
5.19	A box and whisker plot of the training time per model grid search	113
A.1	The technical features for each considered company	130
A.2	The tweets extracted from the Twitter archive	133
A.3	The tweets scraped using web scraping for January 2021	134
A.4	The tweets scraped using web scraping for April 2021	134
A.5	The tweets filtered and scraped using web scraping	136
A.6	The tweets scored using sentiment analysis	138
A.7	The sentiment scores for each considered company	140

B.1	The closing price predictions for the Apple dataset	144
B.2	The closing price predictions for the Microsoft dataset	145
B.3	The closing price predictions for the Amazon dataset	146
B.4	The closing price predictions for the Alphabet dataset	147
B.5	The closing price predictions for the Facebook dataset	148

List of Tables

2.1	The sigmoid and tanh activation functions	10
2.2	The ReLU activation function	28
2.3	The derivatives of the sigmoid and tanh activation functions	42
2.4	The derivative of the ReLU activation function	44
3.1	The top 5 stocks (by market capitalization as at January 2021) of the NASDAQ100 index	56
3.2	Descriptions of each technical feature	57
3.3	The Augmented Dickey-Fuller hypothesis test results	59
3.4	Descriptive statistics of the closing price difference for each dataset . . .	60
3.5	The top 5 most popular cashtags in the Twitter data	64
3.6	Components of the computer used in this study	66
3.7	Descriptions of each sentiment feature	70
3.8	Descriptive statistics of the general public and stock market sentiment features	71
3.9	Descriptive statistics of the company scores sentiment features	72
3.10	Descriptive statistics of the company counts sentiment feature	73
3.11	The Granger causality hypothesis test results (for $p < 0.10$)	74
4.1	Descriptions of each feature subset	78
4.2	The features included in each feature subset	79
4.3	The input of a recurrent model training example	81
4.4	The output of a recurrent model training example	81
4.5	The encoder input of a Transformer training example	82
4.6	The decoder input of a Transformer training example	83
4.7	The decoder output of a Transformer training example	83
4.8	The first autoregressive step of a Transformer training example	85
4.9	The second autoregressive step of a Transformer training example	86

4.10	The hyperparameter grid for the recurrent models	89
4.11	The hyperparameter grid for Transformer model	90
5.1	The optimal hyperparameters for the Recurrent Neural Network model per dataset	94
5.2	The optimal hyperparameters for the Transformer model per dataset . .	95
5.3	Descriptive statistics of the test RMSEs produced in each hyperparameter grid search	96
5.4	The test RMSE for each model-dataset combination	97
5.5	The feature subsets selected after performing a grid search	102
5.6	The number of times each feature subset was selected after performing a grid search	102
5.7	The best test RMSEs for each feature subset of the Transformer-Apple model-dataset combination	103
5.8	The best test RMSEs for each feature subset of the GRU-Amazon model- dataset combination	106
5.9	A summary of the time taken to perform each model grid search	109
5.10	Descriptive statistics of the number of training parameters per model grid search	110
5.11	Descriptive statistics of the number of training epochs per model grid search	112
5.12	Descriptive statistics of the training time per model grid search	113
B.1	The optimal hyperparameters for the Recurrent Neural Network model per dataset	141
B.2	The optimal hyperparameters for the LSTM model per dataset	142
B.3	The optimal hyperparameters for the GRU model per dataset	142
B.4	The optimal hyperparameters for the Transformer model per dataset . .	143

List of Symbols

Mathematical Operators

$\odot$ Hadamard product (elementwise product of two vectors)

Neural Network Symbols

η Learning rate

a Activation

b Bias parameter

C Cost function

f Activation function

w Weight parameter

Acronyms / Abbreviations

ACF Autocorrelation Function

ADF Augmented Dickey-Fuller

AIC Akaike Information Criterion

ANN Artificial Neural Network

API Application Programming Interface

ARIMA Auto-Regressive Integrated Moving Average

BPTT Backpropagation Through Time

CNN Convolutional Neural Network

CPU Central Processing Unit

GPU	Graphics Processing Unit
GRU	Gated Recurrent Unit
HTML	HyperText Markup Language
JSON	JavaScript Object Notation
KFCV	K-Fold Cross-Validation
KSFCV	K-Fold Sequential Cross-Validation
LSTM	Long Short-Term Memory
MAE	Mean Absolute Error
MAPE	Mean Absolute Percentage Error
NLP	Natural Language Processing
PACF	Partial Autocorrelation Function
RAM	Random Access Memory
ReLU	Rectified Linear Unit
RMSE	Root Mean Square Error
RNN	Recurrent Neural Network
SGD	Stochastic Gradient Descent
SOFNN	Self-Organizing Fuzzy Neural Network
SVM	Support Vector Machine
TBPTT	Truncated Backpropagation Through Time

Chapter 1

Introduction

1.1 Introduction

Stock market prediction is the act of predicting the future value of a company's stock traded on an exchange. This can be a lucrative endeavour as successful prediction could yield significant profits for investors. An investor typically attempts to buy a stock before the value rises or sell before the value declines. The stock market is termed bullish when stock prices rise and bearish when stock prices decline. An early knowledge of a bullish or bearish market contributes to wise investing strategies. The prediction of stock markets has been the focus of research for many decades and has posed a great challenge due to its complex, chaotic, and dynamic environment ([Nelson et al., 2017](#)).

The Random Walk Hypothesis proposed by [Malkiel \(1973\)](#) in his book *A Random Walk down Wall Street* claims that predicting the future value of a stock is impossible. The hypothesis suggests that a stock price exhibits signs of a random walk and that successive deviations are random and unpredictable. In the same book [Malkiel \(1973\)](#) proposes the Efficient Market Hypothesis. This hypothesis suggests that the current stock price reflects all available information and that changes in price are only a function of time and new information. Therefore, the future value of a stock is inherently unpredictable. Others disagree and use multiple methods and technologies to gain information on future stock value. Warren Buffet, one of the most successful investors of all time, rebutted these hypotheses and uses fundamental analysis to gain insight on a future stock price ([Buffet, 1984](#)).

In classical economics, there are two major schools of thought when assessing companies for future success. Investors use both technical and fundamental analysis to research companies. These two methods provide different insights and each contributes to predicting future stock price ([Majaski, 2022](#)).

Technical analysis examines the trends of past stock prices to predict their future value. The goal of technical analysis is to use technical indicators to generate short-term trading signals. Technical analysis is considered a short-term investment strategy and is based on the assumption that a stock price moves in trends, mainly because of market psychology ([Majaski, 2022](#)).

Fundamental analysis examines the company that underlies the stock itself. The company's past performance (sales, earnings, profits, etc.) and credibility of their accounts are investigated. The goal of fundamental analysis is to compare the true value of a stock to its current traded value and determine if the stock is undervalued or overvalued. Fundamental analysis is considered a long-term investment strategy and is based on the assumption that if a company operates well, it should be rewarded with additional capital and an increase in stock price ([Majaski, 2022](#)).

Sentiment analysis – the use of natural language processing to study the state of subjective information – is a new method of evaluating company performance. The goal of sentiment analysis is to determine how the public feels regarding a specific topic by analysing the text of news and social media commentary. Public sentiment can influence a company's stock price under the assumption that “public sentiment” and “market sentiment” are correlated ([Bollen et al., 2011](#)).

1.2 Problem Statement

This research aims to utilise both sentiment analysis and advancements in machine learning to predict future stock price.

This study attempts to build on the foundational work done by [Bollen et al. \(2011\)](#) that famously showed that the correlation between “public sentiment” and “market sentiment” exists (in their paper *Twitter mood predicts the stock market*). This research attempts to use Twitter data as an alternative data source to inform changes in stock price. Sentiment analysis will be used to translate the information from the Twitter data into representations that (hopefully) retain the required information to inform changes in stock price.

Many of the recent advancements in machine learning can be attributed to the Transformer – a machine learning architecture that uses attention mechanisms structured for sequence-to-sequence modelling, first developed by [Vaswani et al. \(2017\)](#). This research aims to utilise the Transformer architecture for time series regression tasks – specifically to predict the future stock price of the top 5 stocks of the NASDAQ100 (an index consisting of the 100 largest non-financial companies listed on the NASDAQ stock market). Given the complexity and subsequent number of training parameters, the model’s computational training efficiency will be assessed. Recurrent Neural Networks currently serve as a benchmark for machine learning regression tasks. Therefore, three variation of Recurrent Neural Networks will be used to compare both predictive performance and model computational training efficiency. More specifically, the Recurrent Neural Network and the Long Short-Term Memory and Gated Recurrent Unit variations are used as benchmarks for comparison. A thorough understanding of the architectures of these machine learning models is essential to innovate in future work. Therefore, an important focus of this study is the understanding, application, and evaluation of these machine learning models that will be compared to a more traditional statistical Auto Regressive Integrated Moving Average model.

1.3 Aims and Objectives

The objectives of this study are:

1. Accurately predict the future stock price of the top 5 stocks of the NASDAQ100 (as of January 2021).
2. Use sentiment analysis to translate information from alternative Twitter data into representations that retain the required external information of current events to inform changes in stock price.
3. Assess the predictive performance of different representations of the Twitter data.
4. Build a time series regression model utilising the Transformer architecture.
5. Assess the predictive performance and model computational training efficiency against three variations of Recurrent Neural Networks, and an Auto Regressive Integrated Moving Average model.

The rest of this document is structured as follows: Chapter 2 covers past literature from both a theoretical and methodological perspective. Chapter 3 discusses the technical and sentiment data used in this study. Chapter 4 outlines the methodology used to build and

evaluate the machine learning models, and Chapter 5 presents these results. Chapter 6 discusses and concludes the study performed. Finally, Chapter 7 suggests avenues for future research.

Chapter 2

Literature Review

This literature review covers both a theoretical and methodological background of previous research. The theory ensures sufficient technical understanding before discussing the methodology, and covers the structure and training of the machine learning models used in this study. The structure of these machine learning models is investigated to understand the fundamental differences of each model. Thereafter, the training algorithms of these machine learning models are investigated to understand the vanishing gradient problem – a problem that prevents machine learning models from sufficiently learning information from previous time steps – and the solutions proposed to mitigate this. The methodological background investigates past research that utilised sentiment analysis and machine learning for stock price prediction. Additionally, the innovations of attention-based time series regression models are discussed to inspire the Transformer model built in this study.

2.1 Theoretical Background

The theoretical background first briefly covers the Auto-Regressive Integrated Moving Average model ([Box and Jenkins, 1970](#)). This traditional forecasting model will be used as a baseline for comparison to the more advanced machine learning models. Next, the structure of Artificial Neural Networks ([McCulloch and Pitts, 1943](#)) is covered to lay the foundation for the discussions that follow. Thereafter, the structure of Recurrent Neural Networks ([Rumelhart et al., 1986](#)) are discussed. These neural networks are adapted for time series data by using recurrent edges to reference previous time steps when predicting a future time step. Recurrent Neural Networks suffer from the vanishing gradient problem

where contributions from distant time steps become zero. The Long Short-Term Memory network (Hochreiter and Schmidhuber, 1997) is a variant of the Recurrent Neural Network that attempts to address this problem. The Gated Recurrent Unit network (Cho et al., 2014) is a variant of the Long Short-Term Memory network that mimics its behaviour with fewer parameters. Both these variants attempt to address the vanishing gradient problem, but the use of recurrent edges continues to expose these models to the negation of important information from previous time steps. The Transformer (Vaswani et al., 2017) addresses this by using attention mechanisms rather than recurrence in its network architecture. The attention mechanism allows the model to reference any previous time step when predicting a future time step. A thorough discussion of the structure of these machine learning models – progressing in complexity – will assist in the comprehension of the fundamental differences of each model. Finally, the Backpropagation Algorithm (used to train Artificial Neural Networks) and Backpropagation Through Time Algorithm (used to train Recurrent Neural Networks) are discussed to clearly illustrate the vanishing gradient problem, and the solutions that the Transformer proposes to mitigate it.

2.1.1 Auto-Regressive Integrated Moving Average Model

The Auto-Regressive Integrated Moving Average (ARIMA) model was first introduced by Box and Jenkins (1970). The ARIMA model is used to forecast future values of a time series. This model assumes that the time series can be decomposed into trend, seasonal, and error components. Trend is the observation of an increase or decrease while seasonality is the observation of cycles of patterns recurring in a time series over time. When the statistical properties of a time series change over time – such as trend and seasonality – it is termed *non-stationary*. Alternatively, a time series with constant statistical properties over time is termed *stationary*. The ARIMA models the trend, seasonal, and error components. A brief discussion of each component follows.

An initial differencing step (the Integrated component) can be applied one or more times (up to order d) to eliminate trend and seasonality. The Integrated component (differencing) is simply the subtraction of adjacent time steps

$$y'_t = y_t - y_{t-1}$$

until the time series is deemed stationary.

The resulting time series is modelled using a combination of its lagged values (the Auto-Regressive component) and lagged forecasting errors (the Moving Average component).

The Auto-Regressive component assumes time step y_t is regressed on prior values (up to order p) and is modelled using

$$y_t = \beta_0 + \beta_1 y_{t-1} + \beta_2 y_{t-2} + \dots + \beta_p y_{t-p} + \epsilon_t,$$

where β_0 is the intercept, $\beta_1, \dots, \beta_p$ are the coefficients, and ϵ_t is the error.

The Moving-Average component assumes the regression error ϵ_t is regressed on prior errors (up to order q) and is defined as

$$\epsilon_t = \theta_0 + \theta_1 \epsilon_{t-1} + \theta_2 \epsilon_{t-2} + \dots + \theta_q \epsilon_{t-q},$$

where θ_0 is the intercept and $\theta_1, \dots, \theta_q$ are the coefficients.

Therefore, after applying differencing of order d to ensure a stationary time series, the model is defined as

$$y_t = \alpha + \beta_1 y_{t-1} + \beta_2 y_{t-2} + \dots + \beta_p y_{t-p} + \theta_1 \epsilon_{t-1} + \theta_2 \epsilon_{t-2} + \dots + \theta_q \epsilon_{t-q}.$$

ARIMA models are typically specified by the tuple (p, d, q) , where p , d and q specify the order of the Auto-Regressive, Integrated, and Moving Average components respectively.

The ARIMA model is simple in design compared to the complexity of modern day machine learning models. This benefits both implementation and interpretation. However, the ARIMA model has several shortcomings – especially for the complex task of stock price prediction. Firstly, the ARIMA model can only capture linear dependencies of previous time steps. Secondly, the ARIMA model is an univariate model, and only considers the information of a single variable for future time step predictions. Finally, the ARIMA model relies on strong assumptions – such as stationarity.

Modern machine learning models attempt to solve for more complex use cases – such as stock price prediction – albeit at the expense of additional complexity in architecture. Next, the structure of these machine learning models is thoroughly investigated – progressing in complexity – to understand the fundamental differences of each model.

2.1.2 Model Structure

2.1.2.1 Artificial Neural Network

A neural network is a network of neurons connected by synapses where a single neuron can be connected to multiple other neurons. Artificial Neural Networks (ANNs) are

computational models based on mathematics and algorithms inspired by the biological structure of the neural networks within the brain (McCulloch and Pitts, 1943). Figure 2.1 illustrates a biological neuron and Figure 2.2 illustrates an artificial neuron.

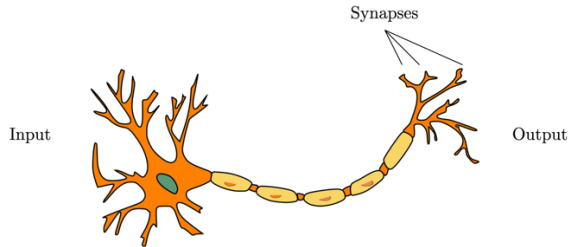


Fig. 2.1 A biological neuron

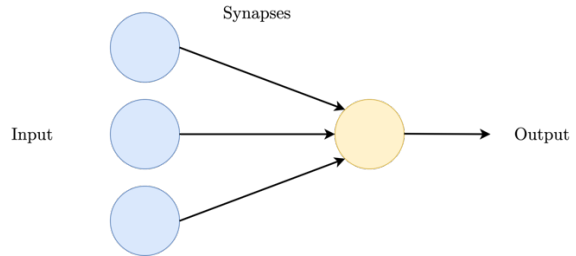


Fig. 2.2 An artificial neuron

Artificial neurons and synapses are modelled by nodes and edges. Each circle in Figure 2.2 is termed a node and represents a neuron. Each arrow in Figure 2.2 is termed an edge and represents a synapse. Multiple nodes can be connected, where the output of one node is the input of another. This is illustrated in Figure 2.2, where the three blue nodes are connected to the yellow node. The edges (arrows) are directed and indicate the flow of information. The output of the blue nodes is the input of the yellow node. These connected nodes and edges form a network of artificial neurons. ANNs will simply be referred to as neural networks from here onwards.

Neural networks consist of layers of nodes as depicted in Figure 2.3. This expands on Figure 2.2, where the information now flows from the three nodes in the left-most layer $l - 1$, to the three nodes in the right-most layer $l + 1$, through the three nodes in the middle layer l . The neural network in Figure 2.3 only consists of three layers for illustration purposes but can be expanded to contain more layers with varying numbers of nodes per layer. Connected layers need not have the same number of neurons.

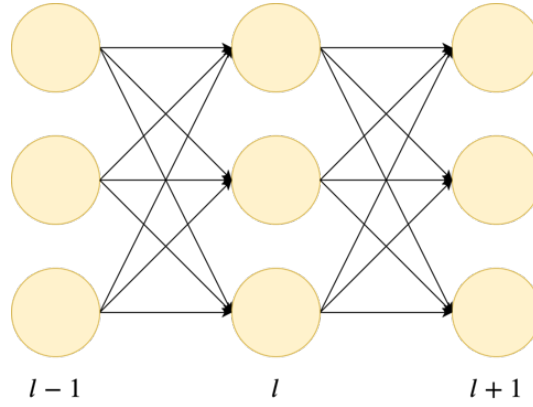


Fig. 2.3 Layers of an Artificial Neural Network

Each node is connected to all the nodes in the previous and next layer by edges with the exception of the input and output layers as these layers do not have a previous and next layer, respectively. This is illustrated in Figure 2.3. Each edge represents a weight. A weight expresses the importance of an output from one node to the respective input of the next node. The weight from the k^{th} node in the $(l-1)^{th}$ layer to the j^{th} node in the l^{th} layer is denoted w_{jk}^l . Note that the weights are indexed “to node – from node” and that w_{jk}^l and w_{kj}^l do not represent the same weight.

Each node has a bias. A bias shifts the output of a node, analogous to a constant in a linear function. The bias of the j^{th} node in the l^{th} layer is denoted b_j^l .

Each node has a scalar activation value, simply termed an *activation*. The activation of the j^{th} node in the l^{th} layer is denoted a_j^l . The activation for node a_j^l is calculated as

$$a_j^l = f(z_j^l), \quad (2.1)$$

where

$$z_j^l = \sum_k w_{jk}^l a_k^{l-1} + b_j^l, \quad (2.2)$$

and f is an activation function that regulates the output of nodes in the neural network. See Table 2.1 for illustrations and definitions of the sigmoid σ and tanh ϕ activation functions. These are common activation functions used in neural networks.

Table 2.1 The sigmoid and tanh activation functions

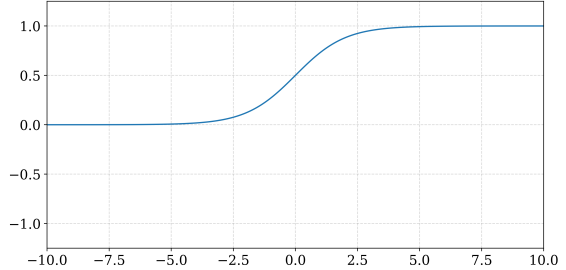
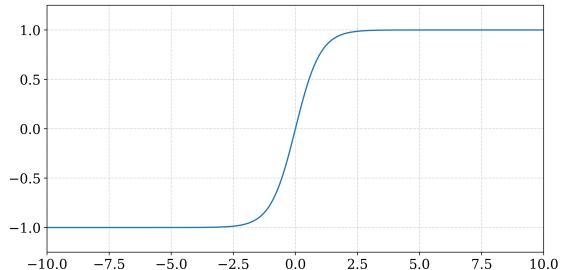
Sigmoid	
$\sigma(x) = \frac{1}{1+e^x}$	
Tanh	
$\phi(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$	

Figure 2.4 provides an example of how the activation of a neuron is calculated using Equation 2.1, 2.2 and the sigmoid activation function σ .

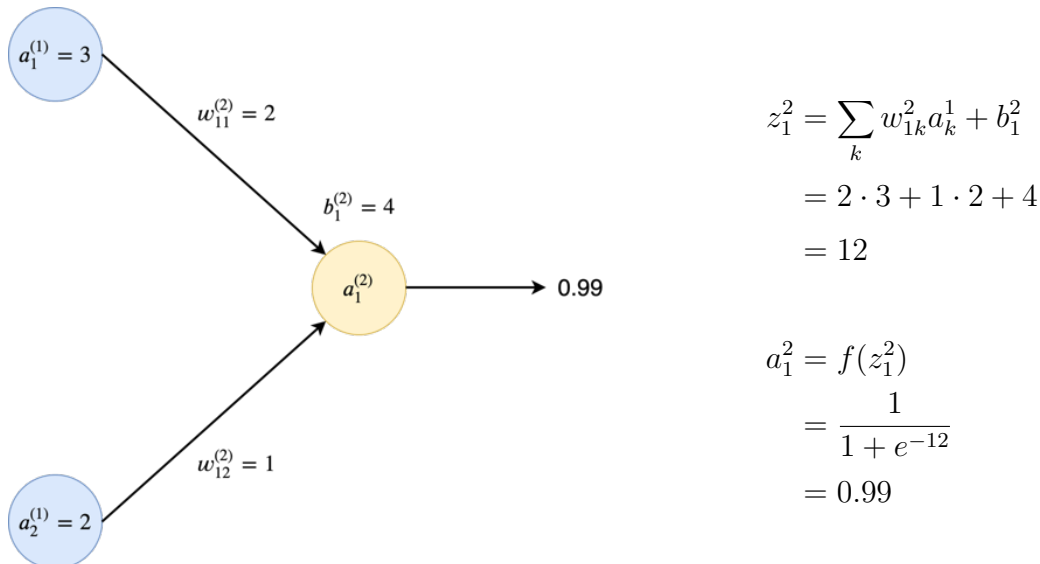


Fig. 2.4 An example highlighting how neurons interact between two layers

Using vectors and matrices, Equation 2.1 and 2.2 can be expressed for an entire layer. Define a vector a^l for each layer l that contains the activations a_j^l for $j = 1, 2, \dots, n_l$ where n_l is the number of neurons in layer l . Similarly, define a vector z^l for each layer l that contains z_j^l for $j = 1, 2, \dots, n_l$. Define a weight matrix W^l for each layer l that contains the weights from the k^{th} node in the $(l-1)^{th}$ layer to the j^{th} node in the l^{th} layer such that the weight w_{jk}^l is in the j^{th} row and k^{th} column. Therefore, the weights are indexed “to node – from node”. Using this notation Equation 2.1 can be written as

$$a^l = f(z^l),$$

and Equation 2.2 as

$$z^l = W^l a^{l-1} + b^l.$$

The topology of a neural network refers to how the network is structured and how the nodes are connected. A common neural network topology is the feed-forward neural network that has a simple structure consisting of consecutive layers of nodes, as depicted in Figure 2.5. A feed-forward neural network is a supervised model that attempts to map an input x to an output y by producing a predicted output $\hat{y}$. There are multiple calculations involved in producing a predicted output $\hat{y}$. An input vector x sets the values of the nodes in the first layer. This first layer is termed the *input layer* and contains the *input nodes* (depicted by blue circles in Figure 2.5). The activations for each successive layer are calculated, hence the name *feed-forward*. The final layer represents the predicted output vector $\hat{y}$ and is termed the *output layer* and contains the *output nodes* (depicted by red circles in Figure 2.5). All layers between the input and output layers are referred to as the *hidden layers* (depicted by yellow circles in Figure 2.5). The *depth* of a neural network is determined by the number of hidden layers. The term *deep learning* is often used when referring to the use of a neural network with many hidden layers.

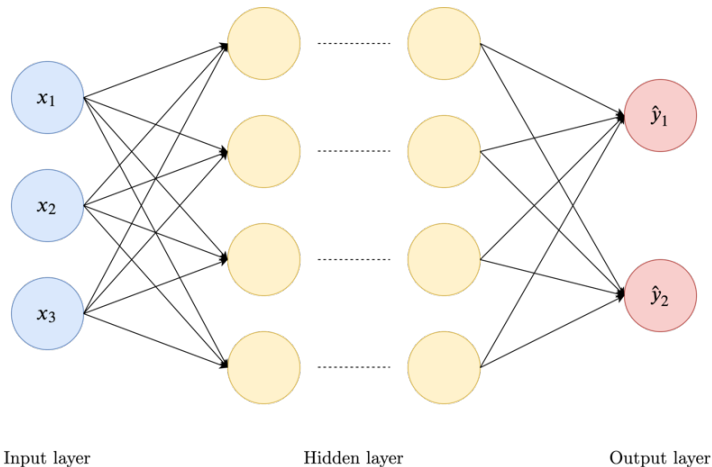


Fig. 2.5 A feed-forward Artificial Neural Network

2.1.2.2 Recurrent Neural Network

Recurrent Neural Networks (RNNs) were first introduced by [Rumelhart et al. \(1986\)](#). RNNs present the idea of time to the feed-forward neural network by using recurrent edges to connect the hidden layer to adjacent time steps, as depicted in Figure 2.6. Here, the input, hidden and output layers have each been condensed into a single node for the simplification of illustration.

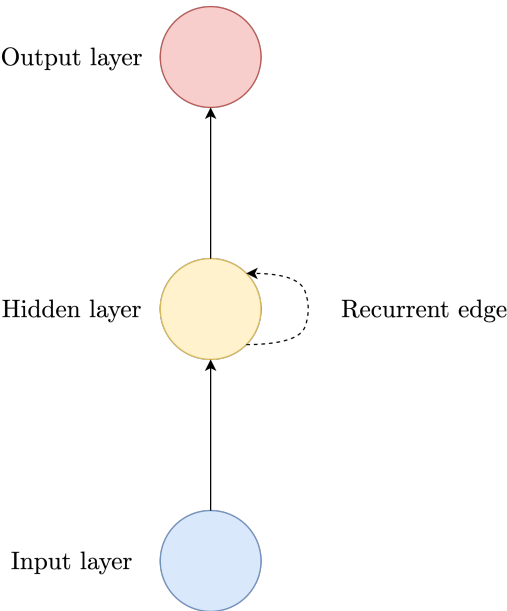


Fig. 2.6 A Recurrent Neural Network

RNNs also introduce a hidden state h_t which acts as the memory of the neural network. The hidden state h_t is a vector that is transferred to each successive time step and contains information from all previous time steps. The recurrent edge in the hidden layer allows information to persist from one input to another. Figure 2.7 illustrates this by unfolding the RNN over multiple time steps.

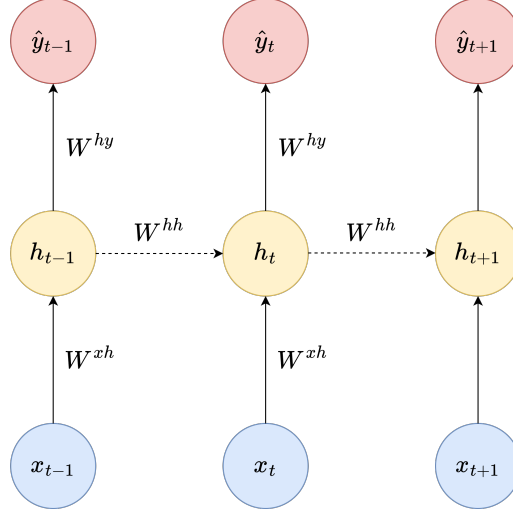


Fig. 2.7 An unfolded Recurrent Neural Network

The current hidden state h_t considers the current input x_t and the hidden state from the previous time step h_{t-1} . The current hidden state h_t is calculated using

$$h_t = f(W^{xh}x_t + W^{hh}h_{t-1} + b_h), \quad (2.3)$$

where W^{xh} is the weight matrix between the input layer and the hidden layer, W^{hh} is the weight matrix between the hidden layer of the previous time step and the current time step, b_h is a bias vector, and f is an activation function.

The current output $\hat{y}_t$ is calculated using

$$\hat{y}_t = f(W^{hy}h_t + b_y),$$

where W^{hy} is the weight matrix between the hidden layer and the output layer, b_y is a bias vector, and f is an activation function. As such, information from previous layers can influence the output of the current layer.

The parameters W^{xh} , W^{hh} , W^{hy} , b_h and b_y are shared across time steps and need to be trained.

2.1.2.3 Long Short-Term Memory

Long Short-Term Memory networks (LSTMs), introduced by [Hochreiter and Schmidhuber \(1997\)](#), are a special kind of RNN. RNNs suffer from the *vanishing gradient problem* where contributions from distant time steps become zero. An LSTM is a variant of the RNN that attempts to address this problem by better learning long term dependencies.

In LSTMs each node in the hidden layer is replaced with an LSTM cell. The LSTM cell is illustrated in Figure 2.8. The figures used to illustrate the LSTM cell are inspired by works of [Olah \(2015\)](#). In Figure 2.8, each line carries an entire vector. Yellow rectangles represent a neural network layer, that is, two layers of nodes (two vectors) connected by edges (weight matrix). The red circles represent the pointwise operation of two vectors, such as pointwise vector multiplication (Hadamard product) or vector addition. Lines merging denote concatenation and lines forking denote the copying of vectors.

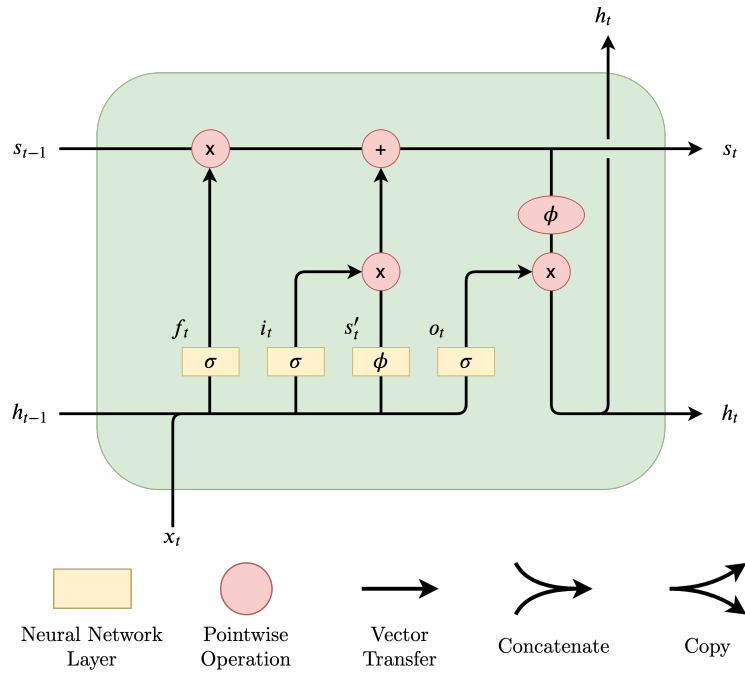


Fig. 2.8 An LSTM cell

The key to the LSTM is the cell state s_t , depicted as the horizontal line running through the top of Figure 2.8. This vector contains the information from the previous LSTM cell. The information in this vector is regulated using gates. Each LSTM cell has three gates that carefully determine what information is forgotten, added, and outputted. Gates consist of a sigmoid neural network layer and a pointwise vector multiplication operation

(Hadamard product), as depicted in Figure 2.9. The sigmoid layer outputs a vector of values between 0 and 1 which determines how much information should pass through the layer – where 0 means “do not let anything through” and 1 means “let everything through”.

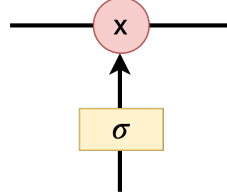


Fig. 2.9 A gate used to regulate learning in an LSTM cell

First, the LSTM cell determines what information should be forgotten from the cell state s_t . This is achieved using a forget gate f_t (illustrated in Figure 2.10 and defined in Equation 2.4). This is not part of the original paper published by Hochreiter and Schmidhuber (1997). The forget gate f_t was introduced by Gers et al. (2000). The forget gate considers the current input x_t and the previous hidden state h_{t-1} , and uses corresponding neural network layers W^{xf} and W^{hf} , and bias vector b_f to regulate what information should be forgotten from the cell state s_t .

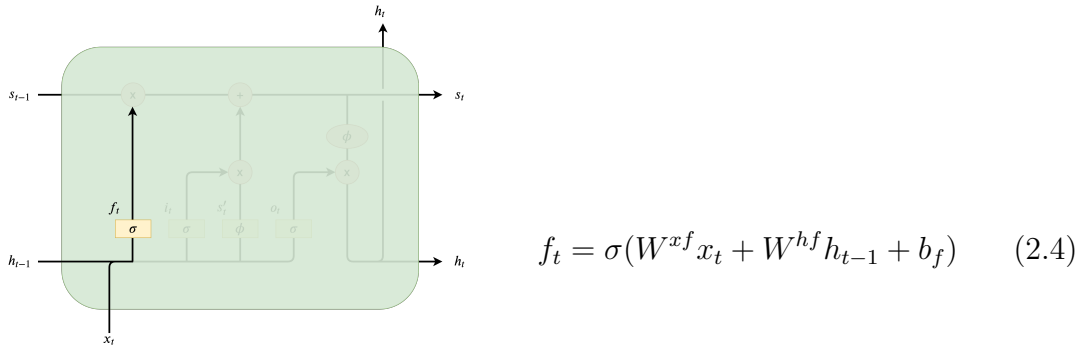


Fig. 2.10 An LSTM cell – highlighting the forget gate

Next, the LSTM cell determines if and what new information should be added to the cell state s_t . This is achieved using an input gate i_t and candidate cell state s'_t . Both are illustrated in Figure 2.11 and defined in Equation 2.5 and 2.6 respectively. Firstly, the input gate i_t considers the current input x_t and the previous hidden state h_{t-1} , and uses corresponding neural network layers W^{xi} and W^{hi} , and bias vector b_i to regulate if

new information should be added to the cell state s_t . Next, the candidate cell state s'_t considers the current input x_t and the previous hidden state h_{t-1} , and uses corresponding neural network layers $W^{xs'}$ and $W^{hs'}$, and bias vector $b_{s'}$ to regulate *what* new information should be added to the cell state s_t . The candidate cell state s'_t in the original paper published by [Hochreiter and Schmidhuber \(1997\)](#) uses a sigmoid function σ , but modern architectures use a tanh function ϕ ([Lipton et al., 2015](#)).

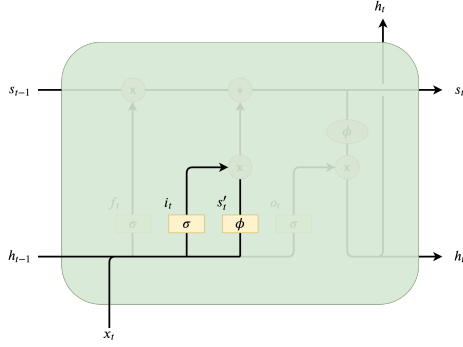


Fig. 2.11 An LSTM cell – highlighting the input gate and candidate cell state

$$i_t = \sigma(W^{xi}x_t + W^{hi}h_{t-1} + b_i) \quad (2.5)$$

$$s'_t = \phi(W^{xs'}x_t + W^{hs'}h_{t-1} + b_{s'}) \quad (2.6)$$

Next, the LSTM cell updates the cell state s_t by forgetting and adding the information determined in the previous two steps. This is illustrated in Figure 2.12 and defined in Equation 2.7.

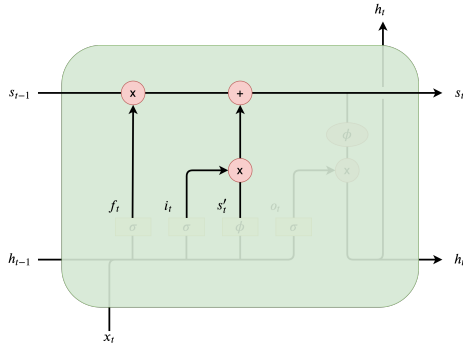
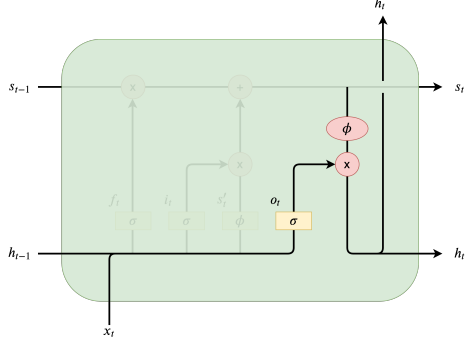


Fig. 2.12 An LSTM cell – highlighting the update of the cell state

$$s_t = f_t \odot s_{t-1} + i_t \odot s'_t \quad (2.7)$$

Finally, the LSTM cell determines the information to output via the hidden state h_t . This is achieved using an output gate o_t (illustrated in Figure 2.13 and defined in Equation

2.8 and 2.9). Firstly, the output gate o_t considers the current input x_t and the previous hidden state h_{t-1} , and uses corresponding neural network layers W^{xo} and W^{ho} , and bias vector b_o to regulate *if* new information should be added to the hidden state h_t . Next, Equation 2.9 regulates *what* combination of information from the cell state s_t and the output gate o_t to output via the hidden state h_t .



$$o_t = \sigma(W^{xo}x_t + W^{ho}h_{t-1} + b_o) \quad (2.8)$$

$$h_t = \phi(s_t) \odot o_t \quad (2.9)$$

Fig. 2.13 An LSTM cell – highlighting the output of the cell

The parameters $W^{xf}, W^{hf}, b_f, W^{xi}, W^{hi}, b_i, W^{xs'}, W^{hs'}, b_g, W^{xo}, W^{ho}$ and b_o are shared across time steps and need to be trained.

2.1.2.4 Gated Recurrent Unit

Gated Recurrent Unit networks (GRUs) are a variant of LSTMs first introduced by [Choe et al. \(2014\)](#). GRUs attempt to imitate LSTMs with fewer parameters, making them faster to train. The GRU cell is illustrated in Figure 2.14. The figures used to illustrate the GRU cell are inspired by works of [Olah \(2015\)](#).

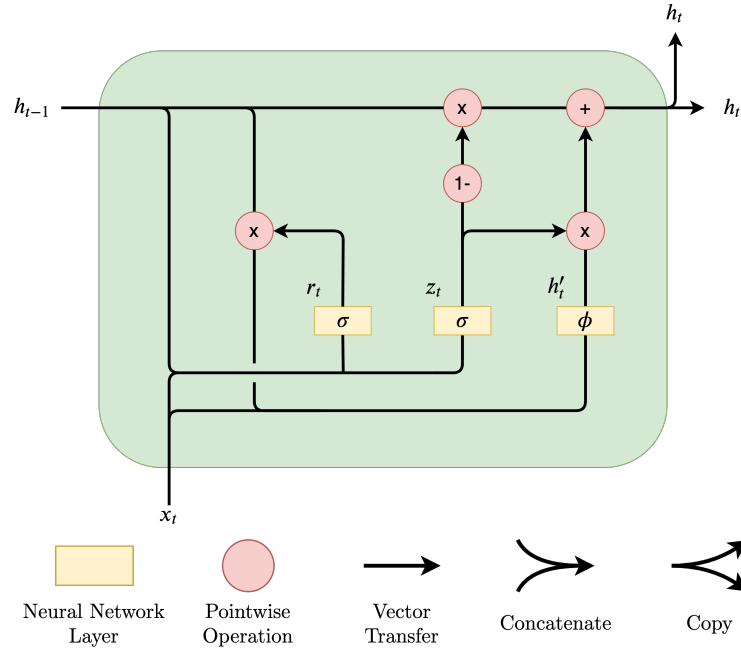


Fig. 2.14 A GRU cell

First, the GRU cell determines if new information should be added to the hidden state h_t . This is achieved using an update gate z_t (illustrated in Figure 2.15 and defined in Equation 2.10). The update gate z_t considers the current input x_t and the previous hidden state h_{t-1} , and uses two neural network layers to regulate if new information should be added to the hidden state h_t .

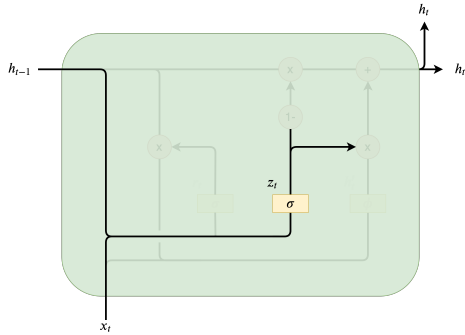
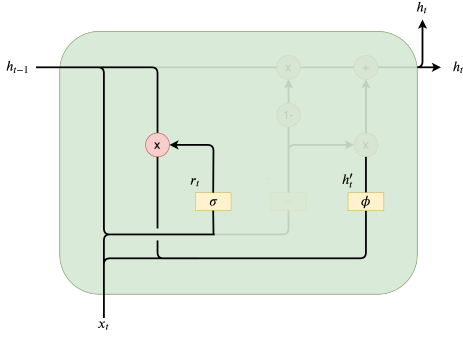


Fig. 2.15 A GRU cell – highlighting the update gate

$$z_t = \sigma(W^{xz}x_t + W^{hz}h_{t-1} + b_z) \quad (2.10)$$

Next, the GRU cell determines if and what new information should be added to the hidden state h_t . This is achieved using a reset gate r_t and candidate hidden state h'_t . Both are illustrated in Figure 2.16 and defined in Equation 2.11 and 2.12 respectively.

First, the reset gate r_t considers the current input x_t and the previous hidden state h_{t-1} , and uses two neural network layers to regulate *if* new information should be added to the hidden state h_t . Next, the candidate hidden state h'_t considers the current input x_t , the reset gate r_t , and the previous hidden state h_{t-1} , and uses two neural network layers to regulate *what* new information should be added to the hidden state h_t . The current input x_t and the previous hidden state h_{t-1} are connected to the candidate hidden state h'_t , each via a neural network layer. The reset gate r_t is combined by multiplication (Hadamard product) with the output of the neural network layer $W^{hh'} h_{t-1}$ to remove (reset) information from the previous hidden state h_{t-1} . The reset gate r_t retains the relevant, and removes the irrelevant, contributions from the previous hidden state h_{t-1} . If no information from previous time steps are relevant, the reset gate r_t removes all contributions from the previous hidden state h_{t-1} , and the candidate hidden state h'_t only contains information from the current input x_t .

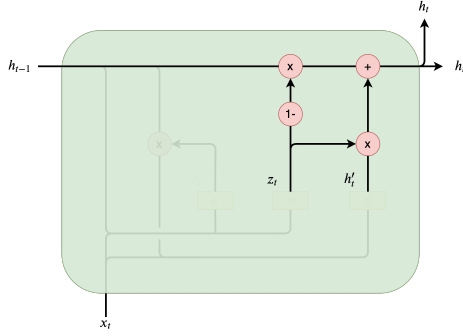


$$r_t = \sigma(W^{xr} x_t + W^{hr} h_{t-1} + b_r) \quad (2.11)$$

$$h'_t = \phi(W^{xh'} x_t + r_t \odot W^{hh'} h_{t-1} + b'_h) \quad (2.12)$$

Fig. 2.16 A GRU cell – highlighting the reset gate and candidate hidden state

Finally, the GRU cell updates the hidden state h_t . This is illustrated in Figure 2.17 and defined in Equation 2.13. The update gate z_t is combined with the hidden state h_t by multiplication (Hadamard product), and the candidate hidden state h'_t is combined with the term $1 - z_t$ by multiplication (Hadamard product). The resulting vectors are added elementwise. Here, the update gate z_t retains the relevant, and removes the irrelevant, contributions of the previous hidden state h_{t-1} and candidate hidden state h'_t .



$$h_t = z_t \odot h_{t-1} + (1 - z_t) \odot h'_t \quad (2.13)$$

Fig. 2.17 A GRU cell – highlighting the new hidden state

The parameters W^{xz} , W^{hz} , b_z , W^{xr} , W^{hr} , b_r , $W^{xh'}$, $W^{hh'}$, and b'_h are shared across time steps and need to be trained.

2.1.2.5 Transformer

The Transformer was first introduced by [Vaswani et al. \(2017\)](#). The Transformer uses attention mechanisms rather than recurrence in its network architecture. As the model predicts the next time step it can reference – or *attend* – to previous time steps that are relevant to the predicted time step. The model learns to identify which time steps to attend to during training. As previously discussed, RNNs also reference previous time steps through the hidden state vector h_t , but due to the vanishing gradient problem RNNs have a very limited reference window and therefore cannot learn long term dependencies. Additionally, the sequence of hidden states generated by RNNs precludes them from parallelisation (the separation of training tasks amongst different computing updates) which significantly increases training time ([Vaswani et al., 2017](#)). An LSTM extends the reference window but is still limited due to the fundamental constraint of sequential computation. Transformers can have a much larger reference window, and in theory – given infinite computational resources – have an infinite reference window. That is, *any* of the previous time steps can be referenced in the prediction of the next time step. The attention mechanisms model dependencies between time steps regardless of distance and exclude recurrence altogether. Transformers have therefore become a very compelling part of sequence modelling.

The architecture of the Transformer as proposed by [Vaswani et al. \(2017\)](#) is structured for sequence-to-sequence modelling – specifically the Workshop on Statistical Machine Translation (WMT) 2014 English-to-German Natural Language Processing (NLP) translation

task. Architectures based only on attention mechanisms have shown tremendous success in NLP translation tasks while being computationally superior (Song et al., 2018). A modification of the original Transformer architecture (Vaswani et al., 2017) is required to accommodate time series data for regression tasks. The adapted architecture as described in this study follows the work of Wu et al. (2020) and is illustrated in Figure 2.18. Note that both the time series windows used as input for the encoder and decoder in Figure 2.18 only consist of the time steps x_1 to x_4 and x_5 to x_8 respectively. This is to reduce the complexity of the examples used in the discussion. In practice, however, these windows are often much larger.

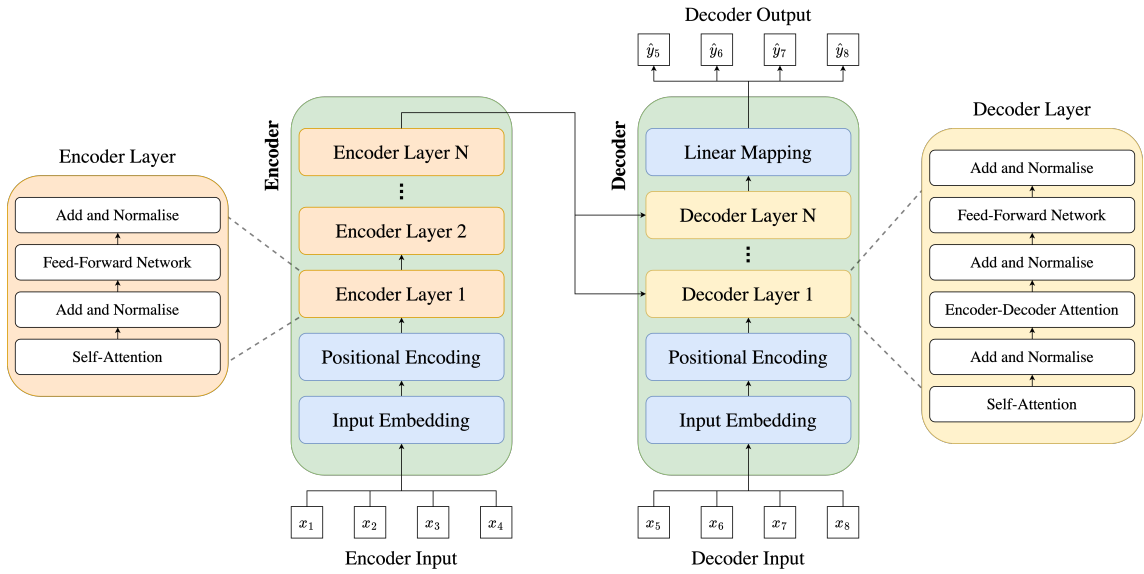


Fig. 2.18 A Transformer

There are three major components to the Transformer model: the encoder, decoder, and the self-attention mechanism (which occurs in both the encoder and decoder). The following discussion will thoroughly cover all three major components. First the encoder and self-attention mechanism is discussed, and thereafter the decoder.

Before the time series can be consumed by the encoder or decoder, each time step needs to be mapped to a vector of a selected size d_{model} . That is, the sequence of time steps is mapped to a sequence of vectors where each vector is of size d_{model} . This is performed by an *input embedding* layer which serves as a lookup table to represent each time step as a learned vector representation. Note that the sequence of vectors contains no positional information, that is, the vectors contain no information regarding their position in the ordered sequence of time steps. Since the Transformer precludes

recurrence, the model would not perceive the positional information of a vector. This is addressed using *positional encoding*, where information is added regarding a vector's position in the sequence of vectors. What follows is a discussion of input embedding and positional encoding. Thereafter, the encoder and decoder are discussed.

Input Embedding

To accommodate the Transformer architecture, the time series needs to be mapped into a high-dimensional vector space. The time series goes through an input embedding layer that performs this mapping. This layer serves as a lookup table to represent each time step as a learned vector representation, that is, each time step is mapped to a vector of a selected size d_{model} . This is illustrated in Figure 2.19.

In the case of a translation task – where each time step is a word – the time series would need to be mapped to a high-dimensional vector space to facilitate the sequence modelling (Song et al., 2018). Consider a regression task where each time step has R measurements. The input embedding layer is tasked to obtain d_{model} -dimensional embeddings for each time step such that $d_{\text{model}} > R$ and serves as a lookup table to represent each time step as a learned vector representation. This is achieved via a neural network layer (Wu et al., 2020). Note that this input embedding is performed without concern for the temporal (time related) information, which is addressed using positional encoding.

Positional Encoding

Positional encoding is used to inject positional information into the input embeddings. Since the Transformer architecture precludes recurrence, the model has no information regarding the temporal order of the original time series. Positional encodings are added to the input embeddings to include information about the relative or absolute positions of the time steps (Song et al., 2018).

There are many different positional encoding techniques. Vaswani et al. (2017) proposes a positional encoding technique using the cosine function

$$\text{PE}_{(\text{pos}, 2i+1)} = \cos\left(\frac{\text{pos}}{10000^{2i/d_{\text{model}}}}\right), \quad (2.14)$$

and the sin function

$$\text{PE}_{(\text{pos}, 2i)} = \sin\left(\frac{\text{pos}}{10000^{2i/d_{\text{model}}}}\right). \quad (2.15)$$

For every odd indexed time step in the time series, a vector is created using the cosine function in Equation 2.14 and for every even indexed time step in the time series, a vector is created using the sine function in Equation 2.15, where pos is the position, i is the dimension of the original time series, and d_{model} is the size of each vector. Each of these vectors is added to their corresponding input embedding and this gives the model information on the position of each vector in relation to the original input sequence. Both the input embedding and positional encoding steps are illustrated in Figure 2.19.

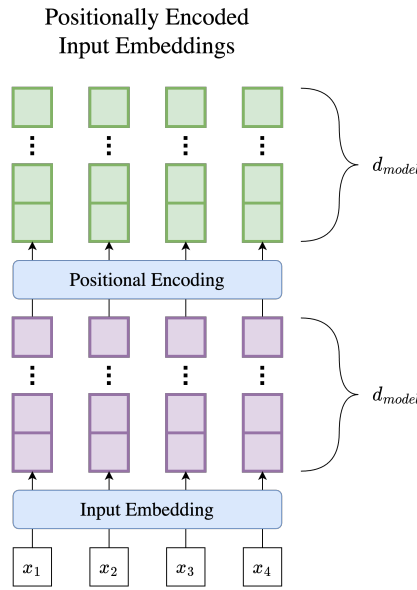


Fig. 2.19 Positionally encoded input embeddings

The time series has now been embedded into a high-dimensional positionally encoded vector space to facilitate the Transformer architecture, that is, the sequence of time steps has been mapped to a sequence of d_{model} -dimensional vectors and positional information has been injected into each vector such that the chronological order information of the original time series is retained. Next, the encoder is discussed.

Encoder

The encoder maps the time series into a continuous representation that contains the learned attention information for this time series. The encoder is illustrated in Figure 2.20.

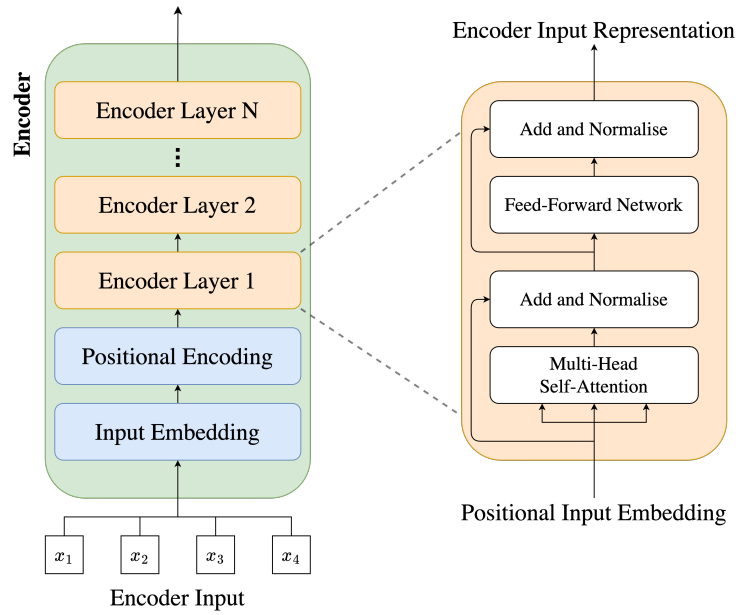


Fig. 2.20 An encoder

The encoder is composed of a stack of identical encoder layers as illustrated on the left of Figure 2.20. Each encoder layer consists of two sub-layers. This is illustrated on the right of Figure 2.20. The first sub-layer is a multi-head self-attention mechanism. The second sub-layer is a position-wise fully connected feed-forward network. A residual connection is employed around each sub-layer, followed by a normalisation layer. Finally, the encoder produces a vector of size d_{model} that contains the learned information of the time series, which is then consumed by the decoder. What follows is a discussion of each sub-layer.

The multi-head self-attention mechanism (first sub-layer) is an essential component of the Transformer architecture and allows the model to preclude recurrence whilst referencing previous time steps. A multi-head self-attention mechanism consists of concatenating multiple individual self-attention mechanisms. A single self-attention mechanism, followed by the concatenation procedure to obtain a multi-head self-attention mechanism, is discussed.

Self-Attention Mechanism

The self-attention mechanism as proposed by Vaswani et al. (2017) is illustrated in Figure 2.21. What follows is a discussion of each component. The figures used to illustrate each component are inspired by works of Phi (2020).

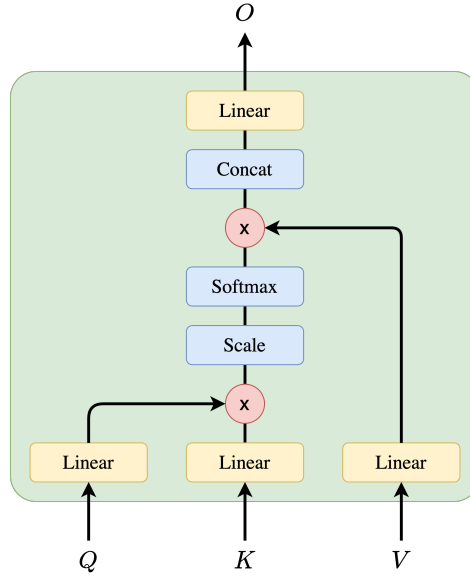


Fig. 2.21 The self-attention mechanism

Self-attention uses a query, key, and value analogue that originates from retrieval systems (Phi, 2020). An intuitive example of a retrieval system is searching for a video on YouTube. The search engine maps the query (the query entered in the search bar) to a set of keys (video titles, descriptions, etc.) and values (the respective videos). These key-value pairs are then used to return an output (the associated videos in their database). The attention function can be described as mapping a query q and a set of key-value pairs k, v to an output o , where the query q , keys k , values v , and outputs o are all vectors (Vaswani et al., 2017).

First, the positionally encoded input embeddings are fed into three distinct linear layers with weight matrices W^Q , W^K and W^V to create the query q , key k and value v vectors of sizes d_k , d_k , and d_v respectively. This is illustrated in Figure 2.22. Note that the query and key vectors are both of size d_k .

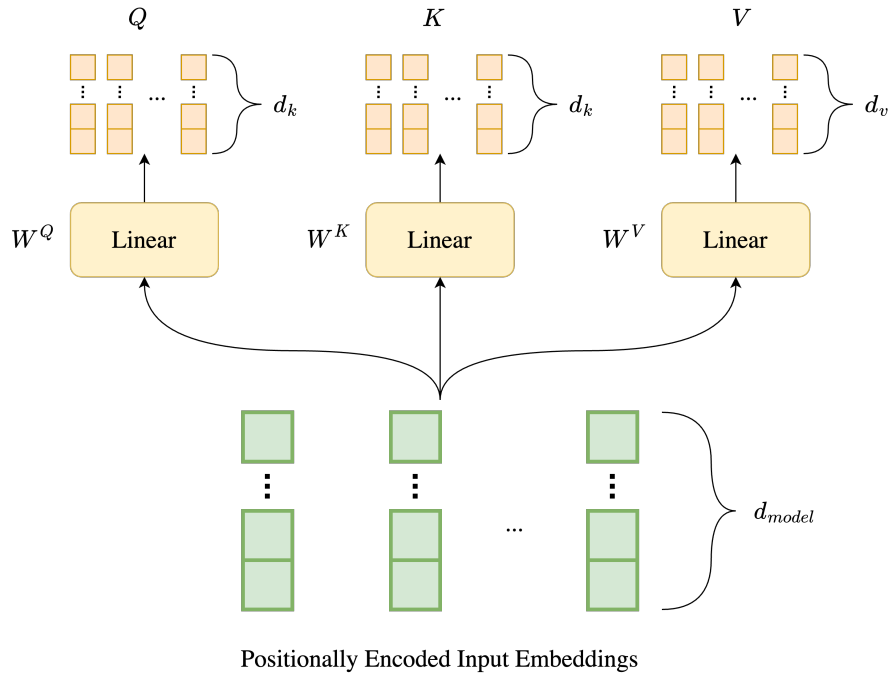


Fig. 2.22 The query, key and value projections in the self-attention mechanism

The query q and key k vectors are multiplied (dot product) to produce a matrix that contains the attention weightings between time steps. This is termed the *score matrix*. The score matrix determines how much emphasis – or *attention* – is placed on the temporal dependency between any given pair of time steps. The higher the score, the more attention is placed on the relationship between those time steps. A score matrix is illustrated in Figure 2.23.

	x_1	x_2	x_3	x_4
x_1	98	34	32	7
x_2	34	94	40	12
x_3	32	40	88	38
x_4	7	12	38	96

Fig. 2.23 A score matrix in the self-attention mechanism

This is how the self-attention mechanism references previous time steps without the use of recurrence. Regarding the retrieval system analogue, the queries have been mapped to keys.

In practise, the attention function of a set of queries are computed simultaneously, packed in a matrix Q . Similarly, the keys and values are packed together in matrices K and V respectively (Vaswani et al., 2017). Using this matrix notation, the attention function can be expressed as

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V, \quad (2.16)$$

where

$$\text{softmax}(z) = \frac{e_i^z}{\sum_{j=1}^K e_j^z}$$

for each row vector z .

The softmax function ensures that each value in the score matrix is in the range $[0, 1]$ and that larger scores are heightened, and lower scores are depressed. This gives the model more confidence of which relationships to attend to (Phi, 2020).

Finally, the attention weights are multiplied with the values V to produce an output. The larger attention weights will retain the values of the time steps that the model has learned are important, while the smaller attention weights will drown out the irrelevant time steps. Regarding the retrieval system analogue, the queries and key-value pairs have been mapped to the output.

Multi-Head Attention

Instead of computing a single score matrix – where each value in the score matrix represents the temporal dependencies – multiple score matrices defined by different parameters can be computed (Vaswani et al., 2017). Each of these score matrices interprets the temporal dependencies between time steps differently and allows the model to jointly attend to information from different representation subspaces at different positions. This is termed *multi-head attention* (Vaswani et al., 2017).

In a multi-headed attention computation, the query, key, and value vectors are separated into N different vectors before applying self-attention. That is, the query, key, and value vectors are of size $d_k = d_v = d_{\text{model}}/N$. Each of these N vectors go through the same self-attention process individually, allowing the self-attention function to be performed in

parallel, yielding N different outputs. Each self-attention process is called a *head*. Due to the reduced dimension of each head and the parallelisation of computation, the total computational cost is similar to that of a single-head attention with full dimensionality (Vaswani et al., 2017). Each head provides an output that gets concatenated before going through a final linear layer. That is,

$$\text{MultiHead}(Q, K, V) = \text{Concat}(\text{head}_1, \dots, \text{head}_N)W^O,$$

where

$$\text{head}_i = \text{Attention}(QW_i^Q, QW_i^K, QW_i^V),$$

and the weight matrices W_i^Q and W_i^K are of dimension $d_{\text{model}} \times d_k$; and the weight matrices W_i^V and W^O are of dimensions $d_{\text{model}} \times d_v$, and $Nd_v \times d_{\text{model}}$ respectively.

Each head provides the model with additional complementary information about the temporal dependencies between time steps by learning different attention weights and allows the model to jointly attend to information from different representations (Vaswani et al., 2017). Next, the position-wise feed-forward network (second sub-layer) is discussed.

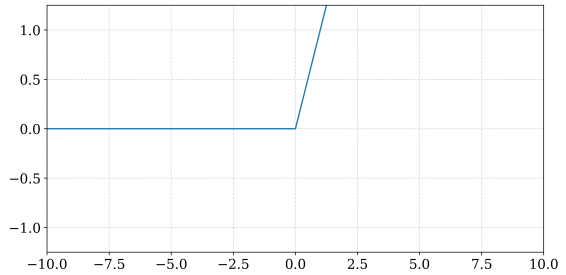
Position-wise Feed-Forward Network

The position-wise fully connected feed-forward network consists of two linear layers with a Rectified Linear Unit (ReLU) activation function in between. Additionally, a residual connection is employed around each sub-layer, followed by a normalisation layer. Each of these components are briefly discussed.

ReLU activation function

The ReLU activation function is defined and illustrated in Table 2.2.

Table 2.2 The ReLU activation function

ReLU	
$\text{ReLU}(x) = \max(0, x)$ $= \begin{cases} x & \text{if } x \geq 0 \\ 0 & \text{if } x < 0 \end{cases}$	

The ReLU activation function reduces computational time complexity as it is simple to compute (compared to the sigmoid and tanh activation functions). If the input is less than 0, the output is equal to 0; if the input is greater than 0, the output is equal to the input. Additionally, this function introduces sparsity (many zero valued elements) to the activation vectors in the neural network (Bai, 2022). This reduces the computational space complexity as integers require less compute space than non-integers. The position-wise fully connected feed-forward network with a ReLU activation function can be expressed as

$$\text{FFN}(x) = \max(0, xW_1 + b_1)W_2 + b_2,$$

where W_1 and W_2 are the weight matrices, and b_1 and b_2 are the bias vectors of the first and second linear layer respectively.

Residual Connections

A residual connection is employed around each sub-layer, followed by a normalisation layer. Residual connections were first introduced by He et al. (2015a). In this work it was shown that it is easier to optimise a residual mapping rather than the original unreferenced mapping.

In a standard feed-forward neural network, the data flows through each layer sequentially. That is, the output of one layer is the input for the next layer where each layer acts as a learned mapping from input to output. He et al. (2015a) proposed a residual block (as illustrated in Figure 2.24) that maps the residuals rather than the original data.

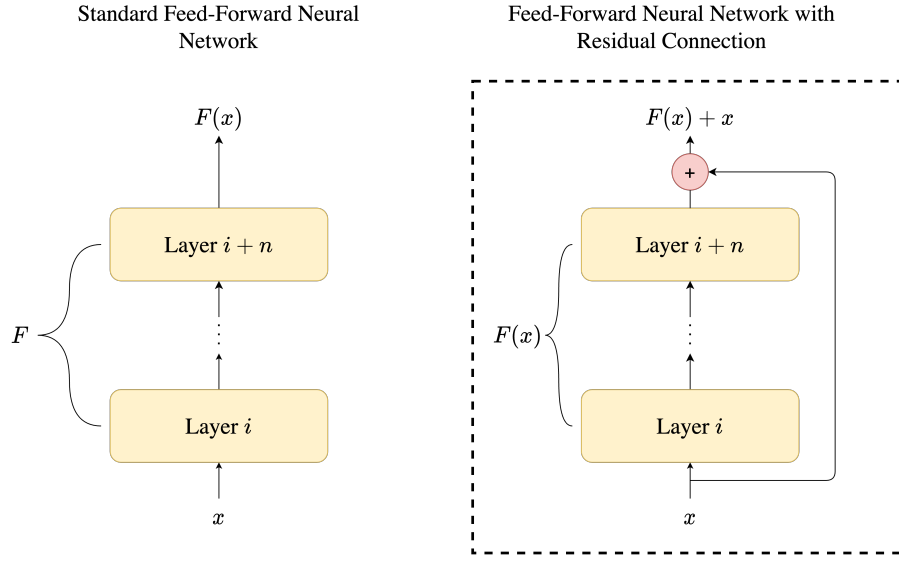


Fig. 2.24 A residual block

Consider a sequence of layers i to $i + n$ (as illustrated in Figure 2.24) where x denotes the input for layer i and let F be the function that represents all layers that follow. In the context of the Transformer, F could represent the multi-head attention mechanism (first sub-layer) or the position-wise fully connected feed-forward network (second sub-layer). In a standard feed-forward neural network, the input x would flow through each layer sequentially and output $F(x)$ as illustrated on the left of Figure 2.24. A residual block rather performs the element wise addition

$$F(x) + x,$$

as illustrated on the right of Figure 2.24. If $F(x)$ and x do not have the same dimensions, a linear transformation

$$F(x) + Wx,$$

where W is a weight matrix, can be used to ensure compatible dimensions.

Layer Normalisation

Ba et al. (2016) first proposed layer normalisation to stabilise the activations in a neural network. The mean μ and variance σ for a given layer l are calculated using

$$\mu^l = \frac{1}{H} \sum_{i=1}^H a_i^l$$

and

$$\sigma^l = \sqrt{\frac{1}{H} \sum_{i=1}^H (a_i^l - \mu^l)^2},$$

where H is the number of hidden nodes in layer l .

Each activation in layer l is then normalised with

$$\frac{a_i^l - \mu^l}{\sigma^l}$$

before being passed on to the next layer. This calculation is performed for every example (training and testing) passing through the normalisation layer. Layer normalisation has been shown to substantially reduce training time (Ba et al., 2016).

The encoder has now mapped the time series into a continuous representation with attention information. This assists the decoder to attend to the appropriate time steps. Next, the decoder is discussed.

Decoder

The decoder considers the output of the encoder and additional input from the original time series to generate predictions. The decoder is illustrated in Figure 2.25.

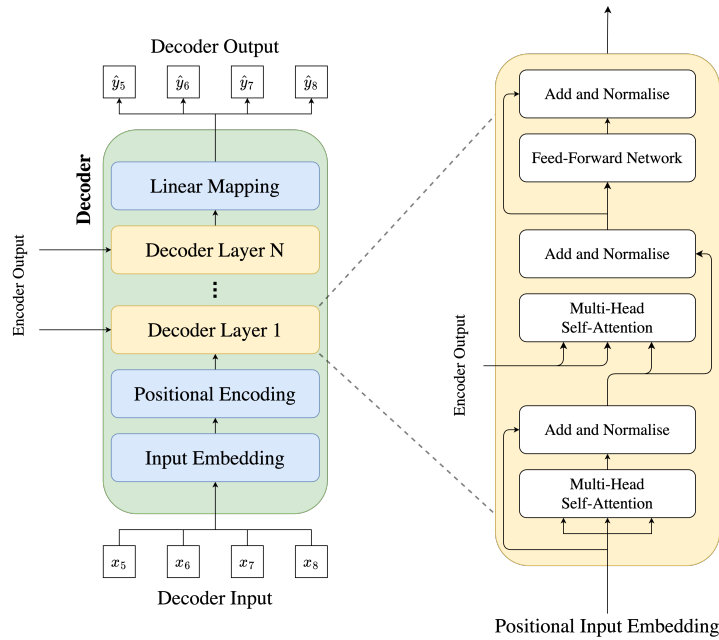


Fig. 2.25 A decoder

The decoder is autoregressive. That is, it predicts one time step at a time and each prediction considers all previous predictions (and the output of the encoder). For illustration purposes, consider Figure 2.25. The decoder first predicts $\hat{y}_5$ based on the encoder output and x_5 of the original time series (at the bottom of the decoder in Figure 2.25). Next, the decoder predicts $\hat{y}_6$ based on the encoder output and x_5, x_6 and the previous prediction $\hat{y}_5$. Next, the decoder predicts $\hat{y}_7$ based on the encoder output and x_5, x_6, x_7 and the previous predictions $\hat{y}_5, \hat{y}_6$. This is repeated until the final prediction $\hat{y}_8$ is made. In the context of a regression use case, the architecture of Figure 2.25 is specified such that multiple time steps into the future are predicted (each time step informed by the previous predicted time steps). However, this can be altered such that a single future time step prediction is produced (only considering the output of the encoder).

The decoder is very similar in structure to that of the encoder. The input embedding and positional encoding is identical. The only difference occurs in the decoder layer. The decoder is composed of a stack of identical decoder layers as illustrated on the left of Figure 2.25. Each decoder layer consists of three sub-layers. This is illustrated on the right of Figure 2.25.

The first sub-layer is a multi-head self-attention mechanism that operates slightly differently to that of the encoder. The autoregressive nature of the decoder cannot consider future time steps when predicting an earlier time step. For example, x_6 cannot be considered when predicting $\hat{y}_5$. In the encoder, the score matrix (Figure 2.23) determines how much attention is placed on the temporal dependency between any given pair of time steps. In the decoder however, the score matrix cannot contain temporal information of a future time step. This is addressed using a *look-ahead mask*. A look-ahead mask is a matrix of identical dimension to that of the score matrix where entries below the diagonal (lower triangle) are filled with zeros and entries above the diagonal (upper triangle) are filled with negative infinities. This is added to the scaled score matrix as illustrated in Figure 2.26. The zeros leave the respective scores unchanged, however the negative infinities change the respective scores to negative infinity. When the softmax function is applied to the masked scores, the negative infinity scores become zero – restricting information to future time steps. The masked score matrix allows for no time step to access any information with regards to future time steps.

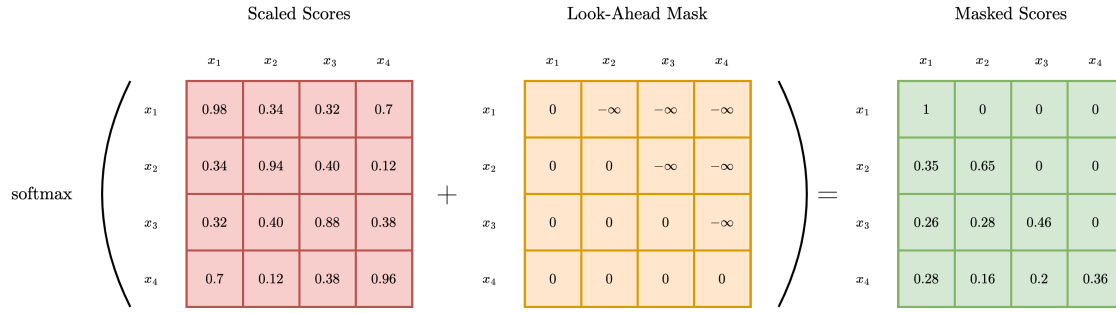


Fig. 2.26 A look-ahead mask applied to a score matrix

This is the only difference between the first sub-layer in the encoder and decoder. This multi-head self-attention mechanism outputs a vector with information on how the model should attend to the original time series used for input in the decoder.

The second sub-layer is a multi-head self-attention mechanism where the encoder output are the queries and keys, and the first sub-layer's output are the values (see Figure 2.21). This matches the encoder output to the decoder input and allows the decoder to determine which time steps to attend to.

The third sub-layer is a position-wise fully connected feed-forward network. This sub-layer operates the same as the position-wise fully connected feed-forward network in the encoder.

A residual connection is employed around each sub-layer, followed by a normalisation layer.

From this discussion, it can be concluded that in the context of a regression problem where only a single future observation is to be predicted, the time series window used as input for the encoder should be large and the time series window used as input for the decoder should be small. The encoder contains the reference window to previous time steps and which time steps to attend to. The larger this reference window, the more information is used to predict the next time step. The time series window used as input for the decoder should be small to limit the number of predictions made. The Transformer is autoregressive, and each prediction considers all previous predictions (and the output of the encoder). In the context of a sequence-to-sequence translation task, it makes sense to consider all previously predicted words when predicting the next word. However, in a regression task where only a single future observation is forecast, predictions need only be based on previously *known* values. Considering too many previous estimates could decrease predictive performance.

The RNN, LSTM, and GRU models use recurrence in their network architecture and are therefore termed the *recurrent models* for the remainder of this study. Next, the training of ANNs and RNNs are discussed. ANNs are trained using the Backpropagation Algorithm. The RNN, LSTM, and GRU recurrent models are trained using Backpropagation Through Time. This exposes these models to the vanishing gradient problem where important information from previous time steps are negated. The Transformer uses attention mechanisms to exclude recurrence from its network architecture, and can therefore be trained using the standard Backpropagation Algorithm. Additionally, the Transformer further uses the ReLU activation function, residual connections, and layer normalisation to further mitigate the vanishing gradient problem and stabilise training.

2.1.3 Model Training

2.1.3.1 Artificial Neural Network

To train a neural network is to optimise the weight and bias parameters such that every predicted output is as close as possible to its respective expected output. Optimising a neural network is a NP-complete problem (Blum and Rivest, 1988). NP-completeness is a classification of the complexity of a problem (Garey and Johnson, 1990). Simply put, there does not exist an algorithm to solve the problem of finding an optimal set of weight and bias parameters for a neural network in polynomial time. A numerical method is employed to optimise these parameters. The training procedure can be summarised in the three abstract steps provided in Algorithm 1.

Algorithm 1: The training procedure of an Artificial Neural Network

1. An input vector x is propagated through the neural network.
 2. The predicted output vector $\hat{y}$ is compared to the expected output vector y .
 3. The weight and bias parameters are adjusted such that the predicted output vector $\hat{y}$ is closer to the expected output vector y .
-

Step 1 of Algorithm 1 has been discussed in conjunction with Figure 2.5, where an input vector x sets the values of the nodes in the first layer and the activations for each successive layer are calculated to produce the predicted output vector $\hat{y}$.

Step 2 of Algorithm 1 uses a cost function C to measure the error between the expected output y and the predicted output $\hat{y}$. An example of a simple cost function is the

quadratic cost function. This is defined as

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

where j represents a node in the L^{th} (output) layer (Equation 2.1). This can also be expressed for the entire output layer as

$$C = \frac{1}{2} ||y - a^L||^2.$$

Step 3 of Algorithm 1 is prefaced by first determining how each weight and bias parameter should be adjusted to decrease the cost. The Backpropagation Algorithm introduced by Rumelhart et al. (1987) is a common algorithm used for this purpose. The goal of the Backpropagation Algorithm is to calculate the gradient of the cost function with respect to each individual weight and bias parameter in the neural network. Every gradient indicates how sensitive the cost function is to a change in that parameter. These gradients are then used in conjunction with a numerical method to optimally adjust each parameter such that the cost decreases.

The Backpropagation Algorithm

The Backpropagation Algorithm consists of four fundamental equations. These equations are used in sequence to determine the gradient of the cost function with respect to any weight or bias parameter in the neural network. The chain rule is often used in the derivation of these equations and explicitly illustrating the function and variable dependencies would be useful in better understanding the equations that follow. These dependencies are illustrated from the cost function (located at the right of Figure 2.27) backwards to each of the adjacent layers. Note that the weight and bias parameters, weighted input, and activation for each layer is illustrated.

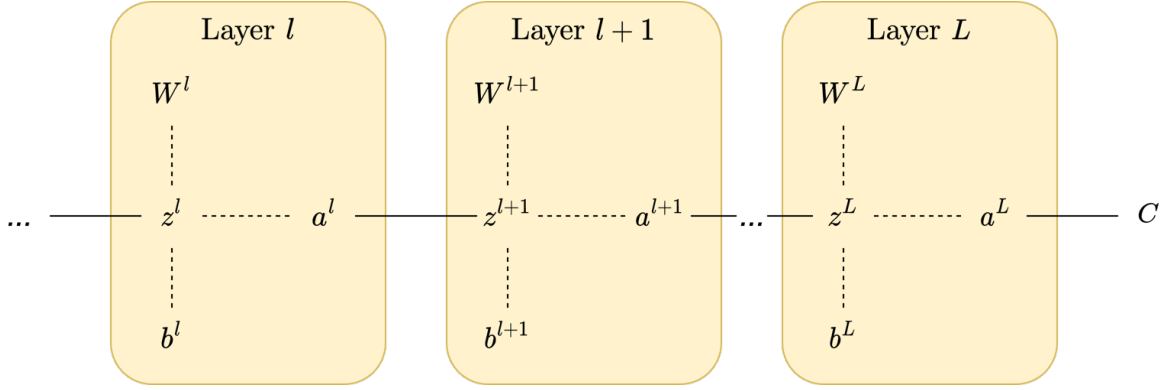


Fig. 2.27 The function and variable dependencies in a feed-forward Artificial Neural Network

First, Equation BP1 is used to calculate the error in the output layer δ^L . Using matrix algebra, this equation can be expressed for the entire output layer δ^L as

$$\delta^L = \nabla_a C \odot f'(z^L), \quad (\text{BP1})$$

where $\nabla_a C$ is the gradient vector consisting of the partial derivatives of the cost function C with respect to each activation in the output layer L .

Next, Equation BP2 is used to write the error for any layer in terms of the error of the next layer. This enables the calculation of the error in any layer, given that the error of the next layer is known. The error for every layer can then be calculated in sequence $\delta^{L-1}, \delta^{L-2}, \dots, \delta^2$. Using matrix algebra this equation can be expressed for the entire l^{th} layer as

$$\delta^l = ((W^{l+1})^T \cdot \delta^{l+1}) \odot f'(z^l). \quad (\text{BP2})$$

Assume the error for the next layer δ^{l+1} is known. Intuitively, by multiplying the vector $f'(z^l)$ with the transpose of the weight matrix $(W^{l+1})^T$ and error δ^{l+1} of the next layer, the error is moved backwards through the network. The error for any layer can be calculated using Equation BP1 and BP2. Starting at the output layer L the error is propagated backwards for each successive layer, hence the name *backpropagation*.

Next, Equation BP3 is used to write the gradient of the cost function, with respect to any weight, in terms of the error of its node δ_j^l and is defined as

$$\frac{\partial C}{\partial w_{jk}^l} = \delta_j^l \cdot a_k^{l-1}. \quad (\text{BP3})$$

Finally, Equation BP4 is used to write the gradient of the cost function, with respect to any bias, in terms of the error of its node δ_j^l and is defined as

$$\frac{\partial C}{\partial b_j^l} = \delta_j^l. \quad (\text{BP4})$$

From these equations, the Backpropagation Algorithm is provided in Algorithm 2.

Algorithm 2: The Backpropagation Algorithm

1. Set the values of the activations in the input layer a^1 with the input vector x
 2. Calculate the activations a^l for each layer $l = 2, 3, \dots, L$
 3. Calculate the error in the output layer δ^L (BP1)
 4. Calculate the error δ^l for each layer $l = L - 1, L - 2, \dots, 2$ (BP2)
 5. Calculate the gradient of the cost function with respect to every weight $\nabla_w C$ (BP3)
 6. Calculate the gradient of the cost function with respect to every bias $\nabla_b C$ (BP4)
-

The gradient vector ∇C consists of the $\nabla_w C$ and $\nabla_b C$ vectors. This gradient vector can be used in conjunction with a numerical method to minimise the cost function. Stochastic Gradient Decent is a numerical method for optimising an objective function and can be used to minimise the cost function.

Stochastic Gradient Decent

Stochastic Approximation was introduced by Robbins and Monro (1951) and building on this work, Stochastic Gradient Descent (SGD) was introduced a year later by Kiefer and Wolfowitz (1952). The gradient vector of the cost function ∇C indicates how sensitive the cost function C is to every weight and bias parameter in the neural network. Changing every parameter by some value proportional to its corresponding value in the gradient vector ∇C can minimise the cost function C locally.

The gradient vector ∇C is calculated using the Backpropagation Algorithm (Algorithm 2) and the weight and bias parameters are adjusted using

$$w \leftarrow w - \eta \nabla_w C \quad (2.17)$$

and

$$b \leftarrow b - \eta \nabla_b C, \quad (2.18)$$

where η is termed the *learning rate*.

A single pass through all the training examples is referred to as an *epoch*. Training neural networks often require multiple epochs to converge the cost function to a local minimum. This can be computationally expensive. Mini batching is an optimisation technique used in SGD to reduce the number of gradient vectors ∇C calculated and as a result reduce training time.

Mini Batching

Neural networks often have thousands, millions, and even billions of weight and bias parameters and calculating the gradient vector ∇C after each training example is computationally expensive. Propagating mini batches of training examples alleviates this problem while still converging the cost function C to a local minimum. A mini batch of m training examples are propagated through the neural network and the average of the cost function C over these m examples is calculated. The gradient vector ∇C is calculated using the average cost of the mini batch and the weight and bias parameters are adjusted using Equation 2.17 and 2.18.

Figure 2.28 and 2.29 contrast SGD without and with mini batches. In Figure 2.28 and 2.29 the red plane represents a function and the yellow dotted line represents the path traversed to a local minimum (by adjusting the parameters using SGD). Without mini batches, the weight and bias parameters are adjusted after each training example. The cost function converges to a local minimum slowly, with precision in the journey (illustrated in Figure 2.28). With mini batches, the weight and bias parameters are adjusted after each mini batch. The cost function converges to the same local minimum faster, albeit with less precision in the journey (illustrated in Figure 2.29).

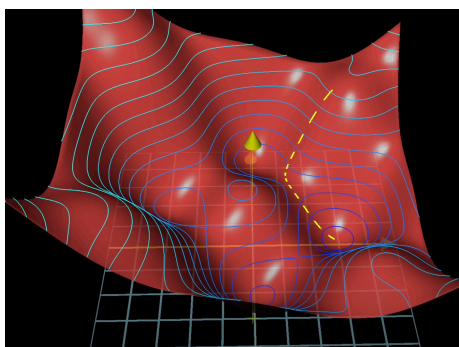


Fig. 2.28 Stochastic Gradient Descent in a 3D plane without mini batching

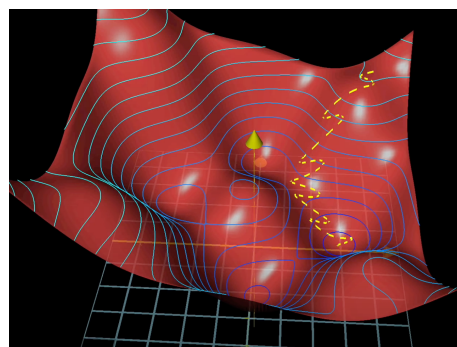


Fig. 2.29 Stochastic Gradient Descent in a 3D plane with mini batching

2.1.3.2 Recurrent Neural Network

RNNs are trained using Backpropagation Through Time (BPTT) introduced by [Werbos \(1990\)](#). Again, the goal is to calculate the gradient of the cost function with respect to each weight and bias parameter. A cost function C_t measures the cost at a specific time step t and the total cost is calculated over the entire sequence. The gradients of the cost function with respect to the parameters W^{xh} , W^{hh} , W^{hy} , b_h and b_y are then calculated and the parameters updated with their accumulated gradients.

The key difference between backpropagation for feed-forward ANNs and RNNs is that the RNN hidden state h_t depends on itself in the previous time step h_{t-1} . This needs to be accounted for when differentiating and, again, explicitly illustrating the function and variable dependencies would be useful in better understanding the equations that follow. These dependencies are illustrated from the cost function (located at the right of Figure 2.30) backwards to each of the adjacent time steps. Note that the hidden state, and weight and bias parameters for each time step are illustrated.

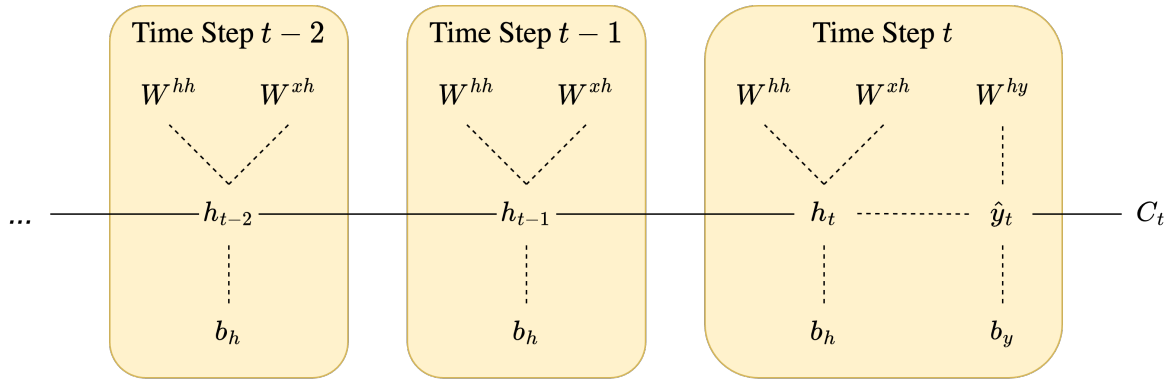


Fig. 2.30 The function and variable dependencies in a Recurrent Neural Network

The gradients of the cost function with respect to the weight matrix between the hidden and output layer W^{hy} are easy to calculate, since neither depend on the hidden state h_t . Following the dependencies in Figure 2.30, the chain rule is employed

$$\frac{\partial C_t}{\partial W^{hy}} = \frac{\partial C_t}{\partial \hat{y}_t} \cdot \frac{\partial \hat{y}_t}{\partial W^{hy}}.$$

The same holds for the gradient of the cost function with respect to the bias vector of the output layer b_y . Following the dependencies in Figure 2.30, the chain rule is employed

$$\frac{\partial C_t}{\partial b_y} = \frac{\partial C_t}{\partial \hat{y}_t} \cdot \frac{\partial \hat{y}_t}{\partial b_y}.$$

Now, the gradient of the cost function with respect to the weight matrix between the hidden layers W^{hh} depends on the hidden state h_t and this needs to be accounted for when differentiating. Following the dependencies in Figure 2.30, the chain rule is employed

$$\frac{\partial C_t}{\partial W^{hh}} = \frac{\partial C_t}{\partial \hat{y}_t} \cdot \frac{\partial \hat{y}_t}{\partial h_t} \cdot \frac{\partial h_t}{\partial W^{hh}}.$$

But h_t depends on W^{hh} indirectly through $h_{t-1}, h_{t-2}, \dots, h_1$

$$\frac{\partial h_t}{\partial W^{hh}} = \frac{\partial h_t}{\partial W^{hh}} + \frac{\partial h_t}{\partial h_{t-1}} \cdot \frac{\partial h_{t-1}}{\partial W^{hh}} + \frac{\partial h_t}{\partial h_{t-2}} \cdot \frac{\partial h_{t-2}}{\partial W^{hh}} + \dots + \frac{\partial h_t}{\partial h_1} \cdot \frac{\partial h_1}{\partial W^{hh}},$$

and thus

$$\frac{\partial C_t}{\partial W^{hh}} = \frac{\partial C_t}{\partial \hat{y}_t} \cdot \frac{\partial \hat{y}_t}{\partial h_t} \cdot \sum_{k=1}^t \left(\frac{\partial h_t}{\partial h_k} \cdot \frac{\partial h_k}{\partial W^{hh}} \right).$$

But h_t depends on h_k directly through $h_{t-1}, h_{t-2}, \dots, h_{k-1}$

$$\frac{\partial h_t}{\partial h_k} = \frac{\partial h_t}{\partial h_{t-1}} \cdot \frac{\partial h_{t-1}}{\partial h_{t-2}} \cdot \dots \cdot \frac{\partial h_{k+2}}{\partial h_{k+1}} \cdot \frac{\partial h_{k+1}}{\partial h_k},$$

and thus

$$\frac{\partial C_t}{\partial W^{hh}} = \frac{\partial C_t}{\partial \hat{y}_t} \cdot \frac{\partial \hat{y}_t}{\partial h_t} \cdot \sum_{k=1}^t \left(\prod_{j=k+1}^t \left[\frac{\partial h_j}{\partial h_{j-1}} \right] \cdot \frac{\partial h_k}{\partial W^{hh}} \right). \quad (2.19)$$

The same holds for the gradient of the cost function with respect to the weight matrix between the input and hidden layer W^{xh} . Following the dependencies in Figure 2.30, the chain rule is employed

$$\frac{\partial C_t}{\partial W^{xh}} = \frac{\partial C_t}{\partial \hat{y}_t} \cdot \frac{\partial \hat{y}_t}{\partial h_t} \cdot \sum_{k=1}^t \left(\prod_{j=k+1}^t \left[\frac{\partial h_j}{\partial h_{j-1}} \right] \cdot \frac{\partial h_k}{\partial W^{xh}} \right). \quad (2.20)$$

Again, the same holds for the gradient of the cost function with respect to the bias of the hidden layer b_h . Following the dependencies in 2.30, the chain rule is employed

$$\frac{\partial C_t}{\partial b_h} = \frac{\partial C_t}{\partial \hat{y}_t} \cdot \frac{\partial \hat{y}_t}{\partial h_t} \cdot \sum_{k=1}^t \left(\prod_{j=k+1}^t \left[\frac{\partial h_j}{\partial h_{j-1}} \right] \cdot \frac{\partial h_k}{\partial b_h} \right). \quad (2.21)$$

The accumulated gradients of the cost function with respect to the weight and bias parameters can now be calculated and subsequently adjusted using Equation 2.17 and 2.18 such that the cost function decreases.

The Vanishing Gradient Problem

The vanishing gradient problem was first formally identified by Hochreiter (1991) and further discussed for RNNs by Hochreiter et al. (2001). The proof for the vanishing gradient problem (Pascanu et al., 2013) is beyond the scope of this study. What follows is an intuitive, informal explanation as presented by Britz (2015).

The following term found in Equation 2.19, 2.20, and 2.21 is responsible for the vanishing gradient phenomenon:

$$\prod_{j=k+1}^t \frac{\partial h_j}{\partial h_{j-1}}. \quad (2.22)$$

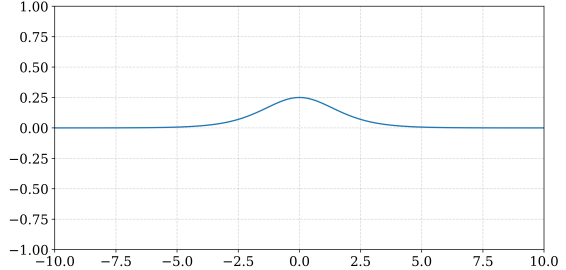
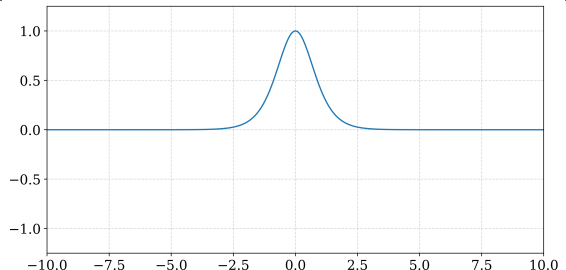
Equation 2.22 consists of many multiplications over a sequence of terms. Because RNNs need to backpropagate gradients over a long sequence (with small valued terms in the multiplication), gradient values will shrink layer after layer, and eventually vanish.

Recall from Equation 2.3

$$h_t = f(W^{xh}x_t + W^{hh}h_{t-1} + b_h),$$

where f is an activation function. The sigmoid and tanh functions (Table 2.1) are commonly used activation functions and their derivatives are bounded by $[0, 0.25]$ and $[0, 1]$ respectively, as illustrated in Table 2.3. As the $t - k$ number of gradient multiplications becomes large, the product shrinks exponentially fast as a function of time t , and eventually completely vanishes after a few time steps.

Table 2.3 The derivatives of the sigmoid and tanh activation functions

Sigmoid	
$\sigma'(x) = \frac{1}{1+e^{-x}} \cdot \left(1 - \frac{1}{1+e^{-x}}\right)$	
Tanh	
$\phi'(x) = 1 - \frac{(e^x - e^{-x})^2}{(e^x + e^{-x})^2}$	

Gradient contributions from distant time steps become zero. Those time steps do not contribute to learning and the RNN does not end up learning long range dependencies.

The Exploding Gradient Problem

Alternatively, with values larger than 1 in the multiplication sequence the gradient values increase layer over layer, and eventually explodes. This is easily detectable, since the gradient values become too large and training stops due to memory constraints of the computer used for training. The exploding gradient problem is easily and effectively solved by constraining gradients to a pre-defined threshold ([Pascanu et al., 2013](#)).

Truncated Backpropagation Through Time

BPTT unrolls the RNN and propagates the error backward over the entire input sequence, one time step at a time. This slows RNN training with long input sequences due to the large number of gradients calculated and the vanishing gradient problem. Truncated Backpropagation Through Time (TBPTT) is a modification of BPTT that limits the number of time steps used on the backward pass and, in effect, estimates the gradient used to update the parameters rather than calculate it fully. Figure 2.31 illustrates TBPTT in an unrolled RNN using 3 time steps to estimate the gradients.

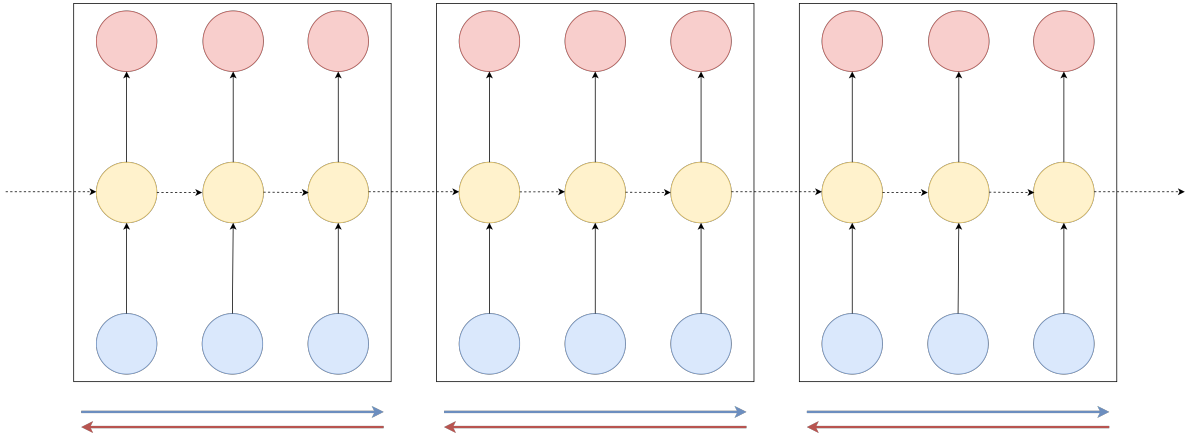


Fig. 2.31 Truncated Backpropagation Through Time in an unrolled Recurrent Neural Network

2.1.3.3 Transformer

The Transformer uses attention mechanisms rather than recurrence in its network architecture. The attention mechanisms model dependencies between time steps regardless of distance and excludes recurrence all together. No weight or bias parameters in the Transformer reference previous time steps. Therefore, the standard Backpropagation Algorithm (Algorithm 2) is sufficient to train the Transformer and BPTT is *not* required. This enables the Transformer to learn long term dependencies – attending to any time steps within the time series window used as input for the encoder. Additionally, the standard Backpropagation Algorithm enables the parallelisation of training compute updates – significantly increasing model computational training efficiency.

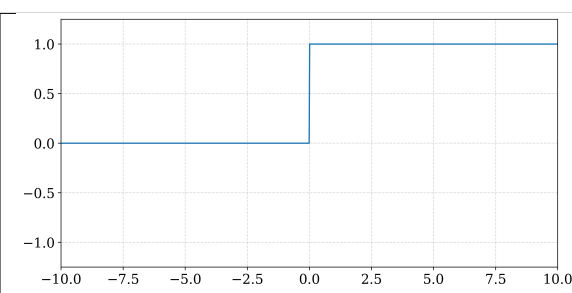
The Transformer uses many techniques to stabilise training and mitigate the vanishing gradient problem. Note that a simple mitigation is already performed in the calculation of the score matrix. In Equation 2.16, for large values of d_k , the dot product in the calculation of the score matrix grows large in magnitude pushing the softmax function into regions where it has extremely small gradients. Therefore, the score matrix is scaled by $\frac{1}{\sqrt{d_k}}$ to stabilise the gradients (Vaswani et al., 2017) and help mitigate the vanishing gradient problem (Phi, 2020). Additional mitigation techniques include the use of the ReLU activation function, residual connections, and layer normalisation.

ReLU Activation Function

The ReLU activation function (defined and illustrated in Table 2.2) is considered one of the big milestones in deep learning and permits routine training of very deep neural

networks (Goodfellow et al., 2016). The ReLU activation function is subsequently often used in modern neural network architectures. The ReLU activation function addresses the vanishing gradient problem caused by the “S” shape of the sigmoid and tanh activation functions (see Table 2.1). For both these functions, when the input slightly deviates from zero, the gradients become very small – almost zero (Bai, 2022) (see Table 2.3). The ReLU activation function addresses this by avoiding non-integer gradients all together. The derivative of the ReLU activation function is illustrated and defined in Table 2.4.

Table 2.4 The derivative of the ReLU activation function

ReLU Derivative	
$\text{ReLU}'(x) = \begin{cases} 1 & \text{if } x > 0 \\ 0 & \text{if } x < 0 \end{cases}$	

This derivative is simple to compute and introduces sparsity – reducing computational time and space complexity. If the input is less than 0, the output is equal to 0; if the input is greater than 0, the output is equal to 1. In training, the gradient vector ∇C calculated using the Backpropagation Algorithm (Algorithm 2) now only consists of 0’s and 1’s. Subsequently, adjustments to the weight and bias parameters (Equation 2.17 and 2.18) now only depend on the size of the learning rate η (with no gradient size or sign considered). If the gradient is 1, the corresponding weight or bias parameter is updated with the learning rate η ; if a gradient is 0, no adjustment is made.

Note the importance of the initial values of the weight and bias parameters. For example, by initialising the weight and bias parameters from a random distribution centered on 0, approximately half of the nodes in the network will output a zero value before training occurs. “He” initialisation is a common method used to initialise weight and bias parameters when using the ReLU activation function (He et al., 2015b).

One disadvantage of the ReLU activation function is that it prevents learning from training examples where activations are zero. If adjustments to the weight and bias parameters (Equation 2.17 and 2.18) are made such that the output (and subsequent gradient) of a node is always zero – regardless of the input – the node is considered *dead*. The weight and bias parameters of this neuron will never be updated again. This is

analogous to brain damage in a biological context. Dead neurons are expected to arise in neural networks using the ReLU activation function. However, too many dead neurons can be problematic. Consider the case where every neuron in a given hidden layer is dead. The gradients of all the weight and bias parameters in the previous layer (and all layers that precede it) will be zero – preventing any learning to occur behind the layer of dead neurons. This can be addressed using different variations of the ReLU activation function such as Leaky ReLU (LReLU), Exponential Linear Unit (ELU), or the Parametric ReLU (PReLU) activation functions. The performance of the Transformer model could vary using different variations of the ReLU activation function. This can be explored in future research, is beyond the scope of this study.

Residual Connections

He et al. (2015a) first showed that it is easier to optimise a residual mapping rather than the original unreferenced mapping. Deep neural networks (consider the depth of stacked encoder layers of the Transformer) are desired for better predictive performance. However, the deeper the neural network, the harder it is to train and converge to an optimal solution. Balduzzi et al. (2018) has shown that in a standard feed-forward neural network the gradients decay exponentially as the depth of the neural network increases – to the point where the gradients start to resemble white noise (random and with no correlation). This has been termed the *shattered gradients problem* where the structure of the gradients are progressively obliterated as neural networks deepen. In contrast, neural networks with residual connections are far more resistant to “shattering” (Balduzzi et al., 2018). Jastrzębski et al. (2018) has also shown that this approach iteratively refines the representations of each layer. Earlier layers concentrate on representation learning while later layers perform iterative refinement. Residual layers provide paths from earlier layers to later layers in deep neural networks, allowing gradients to better flow through the neural network and ultimately allowing for easier and faster training.

In summary, the RNN, LSTM, and GRU recurrent models reference previous time steps through the hidden state vector h_t . This recurrent structure requires these models to be trained using BPTT – exposing them to the vanishing gradient problem. Subsequently, these models have a limited reference window to previous time steps and cannot learn long term dependencies. The Transformer uses attention mechanisms to exclude recurrence from its network architecture, and can therefore be trained using the standard Backpropagation Algorithm. Additionally, the Transformer uses the ReLU activation function, residual connections, and layer normalisation to further mitigate the vanishing gradient problem and stabilise training.

2.2 Methodological Background

2.2.1 Sentiment Analysis

Public sentiment can influence a company's stock price under the assumption that "public sentiment" and "market sentiment" are correlated (Bollen et al., 2011). Only considering technical data limits a model to inform predictions based solely on patterns and trends within the historic stock price with no external information of current events. There are indisputable examples where tweets affect stock price. The business magnate Elon Musk tweeted "Tesla Stock Price is too high imo [in my opinion]" on 1 May 2020 (Musk, 2020). This had an immediate effect on the Tesla stock price – dropping by 10% and wiping \$14 billion off the company's market value in a single day (Siddiqui, 2020). This showcases the power of a single tweet. 6000 tweets are published to Twitter per second equating to 200 billion tweets published every year. Twitter serves as a valuable source of external information – especially the sentiment towards certain people, places, or companies.

Bollen et al. (2011) famously showed that the correlation between "public sentiment" and "market sentiment" exists. The study investigated and confirmed that measurements of collective mood states derived from large-scale Twitter feeds are correlated to the value of the Dow Jones Industrial Average (DJIA) over time. Public sentiment was measured using two tools – OpinionFinder (OP) and Google-Profile of Mood States (GPOMS). Both tools were used to analyse the text of 9.8M tweets posted by approximately 2.7M users from February to December in 2008 to generate a daily time series of public mood. OP is the simpler of the two tools, adhering to a unidimensional model of mood by only determining public mood as either "positive" or "negative". GPOMS expresses public mood with more variety and adheres to a six-dimensional model of mood – calm, alert, sure, vital, kind, and happy. The study used a Granger causality statistical hypothesis test to determine if any of the OP and GPOMS mood time series are useful in forecasting DJIA values over different lag periods. The test revealed that only the "calm" mood GPOMS dimension exhibits a significant correlation with DJIA values over time. Thereafter, the study used a Self-Organizing Fuzzy Neural Network (SOFNN) model to attempt and predict future DJIA values. The SOFNN model predicted the next day's DJIA value using the previous three days of DJIA values and various permutations of the OP and GPOMS mood time series as inputs. Although only the "calm" mood dimension was significant, all mood time series were considered for inputs, as a combination of mood time series could contain predictive information. This is true in concept, however the SOFNN model that only considered the "calm" mood dimension (and the previous three

days of DJIA values) performed best with an 86.7% accuracy and 1.83% Mean Absolute Percentage Error (MAPE). The study used a binomial test to confirm that the direction accuracy is not a result of chance or the selection of a favourable testing period.

Goel and Mittal (2012) performed a study that built on the work of Bollen et al. (2011). The study found that the OP sentiment analysis tool is inadequate for this use case and used a similar tool to that of GPOMS that adheres to a four-dimensional model of mood – calm, happy, alert, and kind. The dataset in the study consisted of more than 476M tweets from more than 17M users from June to December in 2009. A Granger causality analysis revealed that both the calm and happy mood dimensions are most helpful in predicting DJIA values. Thereafter, the study attempted to predict future DJIA values using four different models – Linear Regression, Logistic Regression, Support Vector Machines (SVM) and a SOFNN. Each model used the previous three days DJIA values and various permutations of the mood time series as inputs. The classification models used the up/down DJIA information as inputs, whereas the regression models used the DJIA values. The goal of the study was to predict future DJIA movement and therefore all models were assessed on their directional prediction accuracy, rather than predicting future DJIA values. The study used a validation technique called *K*-Fold Sequential Cross-Validation (*KSFCV*) where models are trained and tested on the data in chronological order. As in *KFCV*, the original dataset is partitioned into *K* folds (subsets) of equal size and is assessed over *K* iterations. However, in each iteration the model is only trained on the partitions that occur *before* the testing partition. This accounts for the temporal nature of the time series data and avoids data leakage – where the model is trained on future data. The study selected $K = 5$. The study found that using the calm and happy mood dimensions (and the previous three days of DJIA values) as input performed best on average across all models. The SOFNN model performed best with 74.56% accuracy and 11.03% MAPE. The study attributes the decrease in accuracy, when compared to the 86.7% accuracy obtained by Bollen et al. (2011), to the use of the *KSFCV*. The study states that this is clearly a better indicator of predictive performance, as it validates predictive performance over the entire period.

Both papers published by Bollen et al. (2011) and Goel and Mittal (2012) were first to prove the utility in considering public sentiment for stock price prediction. This breakthrough has resulted in some level of sentiment analysis in most research concerned with stock price prediction.

Huang et al. (2019) examined using trust filters to enhance the quality of the Twitter user sentiment considered. This paper introduced four trust filters: expertise, experience,

authority, and reputation. These filters are based on the Page Rank and Hyperlink-Induced Topic Search (HITS) algorithms. Each filter returns a trust score (a real number between 0 and 1) and measures the trustworthiness of a Twitter user with respect to that trust filter. These trust scores are multiplied with the respective user's sentiment as a weighting factor. [Huang et al. \(2019\)](#) discovered that all trust filter-based trading strategies outperformed the simple accumulation strategy – a trading strategy where all sentiment is considered equal – with the authority and experience trust filters obtaining the best performance. These trading strategies were solely based on sentiment alone using simple rule-based models – leaving room for further research where trust filters can be incorporated with more advanced modelling techniques.

VADER (Valence Aware Dictionary for sEntiment Reasoning) developed by [Hutto and Gilbert \(2014\)](#) has long been a popular sentiment analysis tool. VADER is a lexicon and rule-based sentiment analysis tool that has been trained on sentiments expressed in social media – making it ideal for use on a Twitter dataset. The VADER lexicon is compiled of lexical features and their corresponding sentiment scores rated on a scale from -4 to +4 representing “extremely negative” and “extremely positive” sentiment respectively. These sentiment scores have been empirically validated by multiple independent human judges. In the compilation of the lexicon, each lexical feature was assigned 10 scores from 10 different independent human judges. Lexical features where the mean of the scores is non-zero (indicating sentiment) and the standard deviation of the scores are less than 2.5 (indicating agreement between judges) were kept. The lexicon consists of over 7500 lexical features with validated sentiment scores. A compound score is computed for a given sentence by summing the sentiment scores of each word in the lexicon, adjusted according to the rules of the model, and then normalised between -1 and 1. The rules of the model account for the different scenarios in which the sentiment of the lexical feature might be altered, and score the sentiment accordingly. For example, these rules account for negation, degree modifiers, multi-word phrases, etc. The compound score provides a single unidimensional measure of sentiment for a given sentence on a scale from -1 (most extreme negative) to +1 (most extreme positive).

VADER has been a popular sentiment analysis tool due to its simplicity, however a simple rule-based model is seldom the best when considering recent developments throughout NLP machine learning models. Since the introduction of VADER, many more advanced pre-trained sentiment analysis models have been developed, such as SpaCy ([Honribal and Montani, 2017](#)) and TextBlob ([Loria, 2018](#)).

Akbik et al. (2019) developed FLAIR, a NLP framework designed to facilitate state-of-the-art sequence labelling, text classification and language models. In NLP machine learning models, words are represented by real-valued vectors to facilitate the mathematical architecture of a machine learning model. Simply put, models understand numbers, not words. Word embedding is the representation of words as real-valued vectors that encode the meaning of the word. Words that are closer in the vector space are expected to have similar meaning. There are many different word embedding techniques. The largest contribution of the FLAIR framework is that it provides a simple, unified interface for all word embedding techniques, allowing researchers to build a single model architecture that can make use of any word embedding technique without any additional engineering effort. Additionally, Akbik et al. (2018) has introduced contextualised embeddings – a word embedding technique that produces different embeddings for the same word depending on its contextual use case. For example, the word “Washington” is embedded differently depending on whether the context indicates that it is used as a location or a last name. Importantly, FLAIR has provided a “model zoo” – a selection of pre-trained models making state-of-the-art NLP models accessible for research, one of which has been made available for public use: a pre-trained sentiment classification model trained on the Internet Movie Database (IMDB) dataset (Maas et al., 2011). The dataset consists of 50 000 highly polar movie reviews for binary sentiment classification. The model outputs a single unidimensional measure of sentiment for a given sentence on a scale from -1 (most extreme negative) to +1 (most extreme positive). Note that this model has not been trained on social media sentiments. However, due to the advancement in NLP models, it is assumed that this model is preferred when compared to a simple rule-based model such as VADER.

2.2.2 Stock Price Prediction using Machine Learning

Roondiwala et al. (2017) used an LSTM model to predict the future stock returns of the NIFTY50. The NIFTY50 is a benchmark Indian stock market index representing a weighted average of the 50 largest Indian companies listed on the National Stock Exchange. Their dataset consisted of 5 years of data from 2011 to 2016 where the open, high, low, and close variables were captured daily. The data was transformed using normalisation, often termed *min-max scaling*, to ensure that each value in the dataset was in the range $[0, 1]$. This allows the network to converge to an optimal state faster and therefore increases training performance. Of the 1312 observations in the dataset, 1180 were used for training and 132 were used for the validation of the model. Multiple models were trained where four different subsets of input features were considered. Note

that variables are often termed *features* in the context of machine learning models. These two terms (variables and features) will be used interchangeably for the remainder of this study. Each network was structured as follows:

1. An input layer with a node for each input feature.
2. An LSTM layer with a hidden vector size of 128.
3. An LSTM layer with a hidden vector size of 64.
4. A layer of 16 nodes using the ReLU activation function.
5. An output layer with a single node using the linear activation function, representing the closing price.

The model used the Root Mean Square Error (RMSE) function as a cost function. [Roondiwala et al. \(2017\)](#) encourages the use of the RMSE over the Mean Absolute Error (MAE) as the RMSE severely punishes large errors. The network was optimised using the RMSprop algorithm, famous for being unpublished and introduced by Geoff Hinton in his online course *Neural Networks for Machine Learning* ([Hinton, 2015](#)). The study concluded that the model which included all four input features and trained over 500 epochs produced the best results with a train RMSE of 0.00983 and a test RMSE of 0.00859.

[Selvin et al. \(2017\)](#) similarly attempted to predict the stock price for companies listed on the National Stock Exchange. The study focused on the IT and Pharma sectors, capturing historic stock data of Infosys, TCS, and Cipla. The dataset consisted of minute wise stock prices from July 2014 to June 2015. The variables captured include day stamp, time stamp, stock price, and volume of stock sold each minute. The data was normalised to map each value to the range $[0, 1]$. This normalised data was used to fit three different models: a Recurrent Neural Network (RNN), LSTM, and Convolutional Neural Network (CNN). Each model was trained over 1000 epochs. In initial testing, layer sizes were varied to determine each optimal hidden layer size. The model that produced the smallest RMSE was selected as the final model for prediction. The MAPE was used to evaluate model performance. The CNN outperformed the other models on both the Infosys and Cipla stocks with an MAPE of 2.36% and 3.63% respectively. However, the CNN performed the worst on the TCS stock with an MAPE of 8.86%. The study fitted an Auto-Regressive Integrated Moving Average (ARIMA) model to the same dataset, which obtained an MAPE of 31.91%, 21.16%, and 36.53% for the Infosys, TCS, and Cipla stocks respectively. The study attributes the CNN's success to its sliding window

approach, where only the current relevant information is considered and motivates that the changes in stock price do not necessarily occur in regular patterns.

Nelson et al. (2017) used an LSTM model to predict the movement of future stock prices. The study used data of 5 different stocks that form part of the IBovespa index from the Brazilian Mercantile and Futures (BM&F) Bovespa stock exchange. This dataset consisted of the open, high, low, close and volume variables captured in 15-minute intervals from 2008 to 2015. This data was transformed using a log-return transformation to stabilise the mean and variance along the time series. Next, exponential smoothing was applied to the time series data to reduce random variation. The open-source Technical Analysis (Ta-Lib) library was used to generate 175 technical indicators from the transformed dataset to represent a diverse set of characteristics of a given stock. The LSTM model was structured as follows:

1. An input layer of 180 input nodes, one for each variable and technical indicator.
2. An LSTM layer with an unspecified hidden vector size.
3. An output layer of a single node using the sigmoid activation function. This ensures that the predicted output is in the range $[0, 1]$ and a threshold value can be used to classify the output.

The study used a threshold value of 0.5 to classify the output as either a 1 (increase in stock price) or a 0 (decrease or no change in stock price). Multi-Layer Perceptron, Random Forest, and Pseudo-Random models were used as a baseline for comparison of performance. The LSTM model outperformed all the baseline models with an average prediction accuracy of 55.9%. To strengthen this conclusion, Nelson et al. (2017) performed a Kruskal-Wallis test to confirm that there is a statistically significant difference between the results of the LSTM and the baseline models. The study commended the LSTM models' ability to learn from such a large input feature dimension. Furthermore, the variation in predictions of the LSTM model was smaller than that of the baseline models which reduces risk in trading operations.

Nguyen and Yoon (2019) proposed a novel approach to stock price movement prediction by using transfer learning. The study attempted to predict the movement of the top 5 stocks in both the KOSPI200 and S&P500 indices. The KOSPI200 (Korean Composite Stock Price Index 200) is a Korean benchmark index that tracks the 200 largest publicly-traded common stocks in Korea. The S&P500 (Standard and Poor's 500) is an American benchmark index that tracks the 500 largest companies listed on stock exchanges in the United States of America. The study used an approach where two models are trained – a

base model and a predictive model. The base model is trained on the historic stock data of the top 50 similar stocks (ranked in terms of market capitalisation). This large dataset was captured from 2012 to 2018. The trained parameters of the base model are then used as the initial parameters of the prediction model before training. This is termed *transfer learning*. The prediction model is then trained on a specific stock to fine tune the parameters to the specific stock considered. Six different subsets of input features were considered where each subset varied in size. In order to perform transfer learning, both the base and predictive models are required to have the same structure. These models were structured as follows:

1. An input layer with a node for each input feature.
2. An LSTM layer with a hidden vector size of 16.
3. An LSTM layer with a hidden vector size of 16.
4. An output layer with a single node using the sigmoid activation function to classify the future stock price movement, similar to the study performed by [Nelson et al. \(2017\)](#).

Both models were optimised using the RMsprop algorithm. The base model was optimised over 5000 epochs with a learning rate of 0.001. The predictive model was optimised over 500 epochs with a learning rate of 0.0005. The smaller learning rate was chosen to avoid distorting the pre-trained parameters too quickly. SVM, Random Forest, and K-Nearest Neighbours models were used as benchmarks. The best performing LSTM model obtained an average accuracy of 60.64% across the 10 stocks considered. The LSTM model improved on the average accuracy of the benchmarked models by 3.34%.

[Maqsood et al. \(2020\)](#) investigated whether local and global event sentiment influenced stock markets. The study analysed 11.42 million tweets to determine the sentiment of events occurring in the US, Hong Kong, Turkey, and Pakistan from 2012 to 2016. In addition to the sentiment data, the dataset included the open, high, low, adjusted close, and volume variables captured on a daily interval from January 2000 to October 2018. The study used three models to forecast the stock prices of the top listed companies in each country and measured the impact of local and global event sentiment by including and excluding the sentiment information and measuring the difference in predictive performance. The three different forecasting models used were SVM, Linear Regression and Deep Learning (an ANN with many hidden layers). These models were evaluated with the RMSE and MAE performance metrics. [Maqsood et al. \(2020\)](#) concluded that local and global events have an immediate impact on stock markets and including the

sentiment of these events increase the performance of the stock forecasting models – especially when using Deep Learning techniques.

2.2.3 Attention-Based Regression Models

At the time of writing, little research has been conducted using attention-based regression models for stock price prediction. Therefore, two other studies are discussed that developed attention-based regression models for medical time series data.

[Song et al. \(2018\)](#) proposed a Simply Attend and Diagnose (SAnD) model for clinical time series analysis that *only* uses attention mechanisms. The study proposed using self-attention through multi-head attention mechanisms to model the temporal dependencies in the multivariate clinical time series data. The study attempted to model hospital mortality, decompensation (organ failure), length of stay, and phenotyping using the MIMIC-III benchmark dataset. Hospital mortality and decompensation are both binary classification problems and are respectively defined as the outcome of whether a patient dies during hospital admission or dies due to the deterioration of organ functionality within the next 24 hours. Length of stay was reposed into a classification problem using ten buckets, namely: A bucket for less than a day, seven buckets for each day of the first week, a bucket for stays more than a week but less than two weeks, and a bucket for stays greater than two weeks. Phenotyping is a multi-label classification problem defined as the outcome of diagnosing a patient with one or more of the 25 conditions considered in the dataset. Typically, a patient is diagnosed with multiple conditions.

The study used positional encoding, as well a dense interpolation embedding technique, to add positional information to the input embedding. The model was structured as follows:

1. The input sequence is embedded into a high-dimensional vector space (input embedding).
2. Positional information is added to the embedded input sequence (positional encoding).
3. The positionally encoded input embeddings are fed into a multi-head attention mechanism stacked N times.
4. The outputs of the multi-head attention mechanisms are concatenated via a feed-forward network.
5. A dense interpolation layer is used to encode additional temporal information.

6. An output layer with the appropriate activation function is used to produce the output of the model. A sigmoid activation function was used for binary classification (hospital mortality and decompensation). A softmax activation function was used for multi-label classification as the classes were not mutually exclusive and an outcome for each class could be considered simultaneously (length of stay and phenotyping).

The model was optimised using the Adam optimiser with parameters $\beta_1 = 0.9$, $\beta_2 = 0.98$ and $\epsilon = 10^{-8}$. An additional model, SaND-Multi, was developed to use joint inferencing from multiple tasks to improve performance. In this model, a cost function that jointly evaluates the performance of all tasks was used. This is expressed as

$$l_{mt} = \lambda_i l_{ihm} + \lambda_d l_{dc} + \lambda_l l_{los} + \lambda_p l_{ph}$$

where l_{ihm} , l_{dc} , l_{los} , and l_{ph} correspond to the losses for each of the four tasks (hospital mortality, decompensation, length of stay, and phenotyping), and λ_i , λ_d , λ_l , and λ_p are pre-selected weights. The input embedding and attention modules are shared across each task, while the final representations and the prediction layers are unique to each task. The model was assessed using the Micro AUC, Macro AUC, and Weighted AUC performance metrics. These metrics measure the performance of a classification model as the area under an ROC curve. The ROC curve measures the true positive rate against the false positive rate at different classification thresholds. These metrics are therefore *not* dependent on the selection of a classification threshold. The final results were compared to the current state-of-the-art results (where an LSTM model is used) and the SaND-Multi model obtained superior performance in every prediction task.

[Wu et al. \(2020\)](#) developed a Transformer model leveraging self-attention mechanisms to learn complex patterns and dynamics from time series data. The study attempted to predict the volumes of United States citizens that will show symptoms of Influenza-like-Illnesses (ILI). The weekly ILI reports used in the study were obtained from the Centres for Disease Control and Prevention (CDC). Forecasting ILI activity enables public health agencies to plan for potential pandemics and allocate resources accordingly.

The attention-based regression model in the study mimicked the original Transformer architecture developed by [Vaswani et al. \(2017\)](#). Both the encoder and decoder were composed of:

1. An input layer with input embedding and positional encoding.

2. A stack of 4 identical encoder/decoder layers – each consisting of a self-attention sub-layer, a fully-connected feed-forward sub-layer, where each sub-layer was followed by a normalisation layer.

Min-max scaling was applied to the dataset. A fixed-length sliding time window was traversed over the scaled dataset to produce training examples with features and labels. The features consisted of observations from the previous 10 weeks, and the labels consisted of ILI ratios for the next 4 weeks. That is, the encoder input $x_1, x_2, \dots, x_{10}$ and the decoder input $x_{10}, \dots, x_{13}$ was used to predict the decoder output $x_{11}, \dots, x_{14}$.

The model used an Adam optimiser to fit the weight and bias parameters and applied a dropout rate (a fraction of randomly selected neurons to ignore in a training step) of 0.2 in each sub-layer to prevent overfitting. The Pearson correlation coefficient and RMSE was used to assess model performance. The study used 3 models as benchmarks for comparison: ARIMA, LSTM and an attention-based Seq2Seq model.

All the machine learning models outperformed the ARIMA model in both the Pearson correlation coefficient and RMSE performance metrics. The study found that the Transformer model produced slightly higher Pearson correlation coefficients than the LSTM and Seq2Seq model. Additionally, the Transformer model outperformed both the LSTM and Seq2Seq models with a relative reduction in RMSE of 27% and 8.4% respectively. The study concluded that the Transformer can better capture the complex dynamic patterns in the weekly ILI reports than any of the other models tested.

Chapter 3

Data

This study will aim to produce a model that can accurately predict the future price of the top 5 stocks (by market capitalization as at January 2021) of the NASDAQ100 index (listed in Table 3.1). This index includes 100 of the largest (based on market capitalization) domestic and international non-financial companies listed on the NASDAQ stock market. These top 5 companies are all technology companies on the forefront of innovation and each defining their modern-day industry. These companies have been selected for this study as their technical and sentiment data is easily accessible and freely available.

Table 3.1 The top 5 stocks (by market capitalization as at January 2021) of the NASDAQ100 index

Rank	Company	Symbol
1	Apple Inc.	AAPL
2	Microsoft Corporation	MSFT
3	Amazon.com Inc.	AMZN
4	Alphabet Inc.	GOOG
5	Facebook Inc.	FB

The time series data collected for this study is captured on a daily interval and consists of two parts: technical and sentiment data.

3.1 Technical Data

The technical data has been retrieved from Yahoo! Finance ([Yahoo!, 2021](#)) and consists of 5 features. Table 3.2 provides a description of each feature.

Table 3.2 Descriptions of each technical feature

Feature	Description
Open	Price of the stock at the beginning of the business day when the NASDAQ stock market opens at 14:30 (GMT).
High	Highest price of the stock for the given day.
Low	Lowest price of the stock for the given day.
Close	Price of the stock at the end of the business day when the NASDAQ stock market closes at 21:00 (GMT).
Volume	The number of stocks traded (bought and sold) for the given day.

Each of these features have been retrieved for each of the considered companies in Table 3.1. Figure 3.1 illustrates each technical feature captured for the Facebook dataset. Similar illustrations for the other considered companies are provided in Figure A.1 in Appendix A.

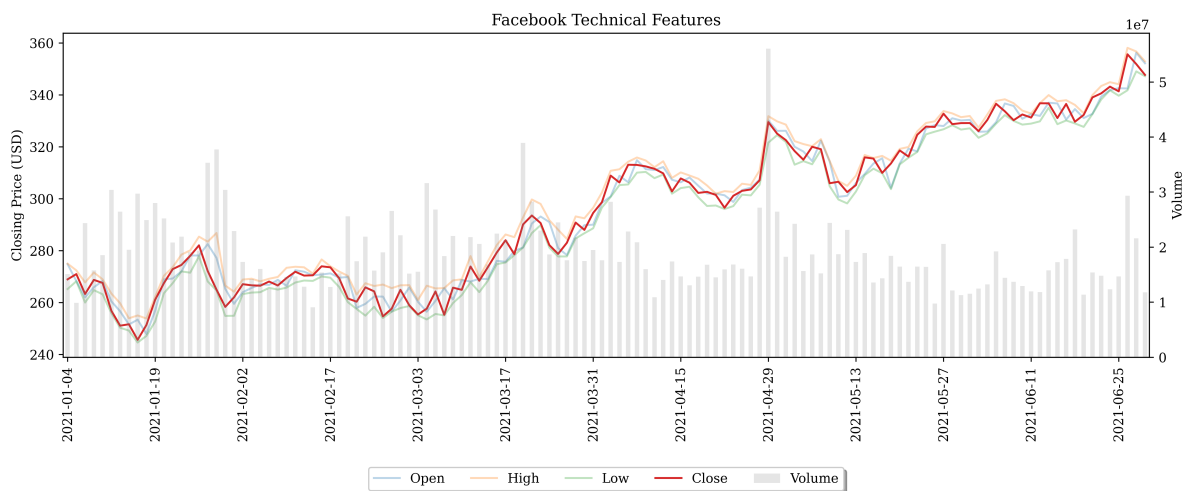


Fig. 3.1 A plot of each technical feature in the Facebook dataset

The closing price is the feature to be predicted and is therefore termed the *target feature*. Note the increasing trend over time in Figure 3.1. Consider the case where a model is

trained on the first 60% of the time series, validated on the next 20%, and tested on the final 20%. Here, the closing price values in the training data would be smaller than that of the testing data. The model would not be exposed to similar training and testing data, and would subsequently decrease the predictive performance of the model on unseen data. Here, the model would be required to predict a data point it has never seen in the training data. When the statistical properties of a time series changes over time – such as a change in mean, variance and autocorrelation – it is termed *non-stationary*. This is observed in Figure 3.1. Alternatively, a time series with constant statistical properties over time is termed *stationary*. A non-stationary time series can be transformed into a stationary time series by *differencing* the time series. A differencing (of order 1) is simply the subtraction of adjacent time steps – resulting in a time series of *changes* in value rather than absolute value. The Augmented Dickey-Fuller statistical hypothesis test is used to determine the stationarity of a time series.

Augmented Dickey-Fuller Test

The Augmented Dickey-Fuller (ADF) test was first introduced by [Dickey and Fuller \(1979\)](#). The ADF test tests the null hypothesis H_0 that a time series has a unit root and is subsequently non-stationary. The alternative hypothesis H_A states that the time series does not have a unit root and is subsequently stationary. The ADF test is simply used and not discussed as the derivation of this statistical test surpasses the scope of this study. These hypotheses can be expressed as:

Null Hypothesis (H_0): The time series has a unit root and is non-stationary.

Alternative Hypothesis (H_A): The time series has no unit root and is stationary.

The ADF test was applied to the closing price of each dataset with a lag of 5. That is, for every time step y_t , the previous 5 time steps $y_{t-1}, \dots, y_{t-5}$ are considered to determine if the time series is stationary or non-stationary.

The resulting p-values of the closing price and closing price difference are provided in Table 3.3.

Table 3.3 The Augmented Dickey-Fuller hypothesis test results

Dataset	Feature	p-value	Outcome
Apple	Closing Price	0.645	Non-stationary
	Close Price Difference	0.003	Stationary
Microsoft	Closing Price	0.528	Non-stationary
	Close Price Difference	0.022	Stationary
Amazon	Closing Price	0.561	Non-stationary
	Close Price Difference	0.009	Stationary
Alphabet	Closing Price	0.615	Non-stationary
	Close Price Difference	0.000	Stationary
Facebook	Closing Price	0.814	Non-stationary
	Close Price Difference	0.000	Stationary

From Table 3.3, we can conclude (at a 0.10 level of significance) that the closing price for each dataset (at lag 5) is non-stationary as we fail to reject the null hypothesis H_0 ($p > 0.10$). After applying a single differencing, we can conclude (at a 0.10 level of significance) that the closing price difference for each dataset (at lag 5) is stationary as we reject the null hypothesis H_0 ($p < 0.10$). The closing price difference is illustrated for the Facebook dataset in Figure 3.2.

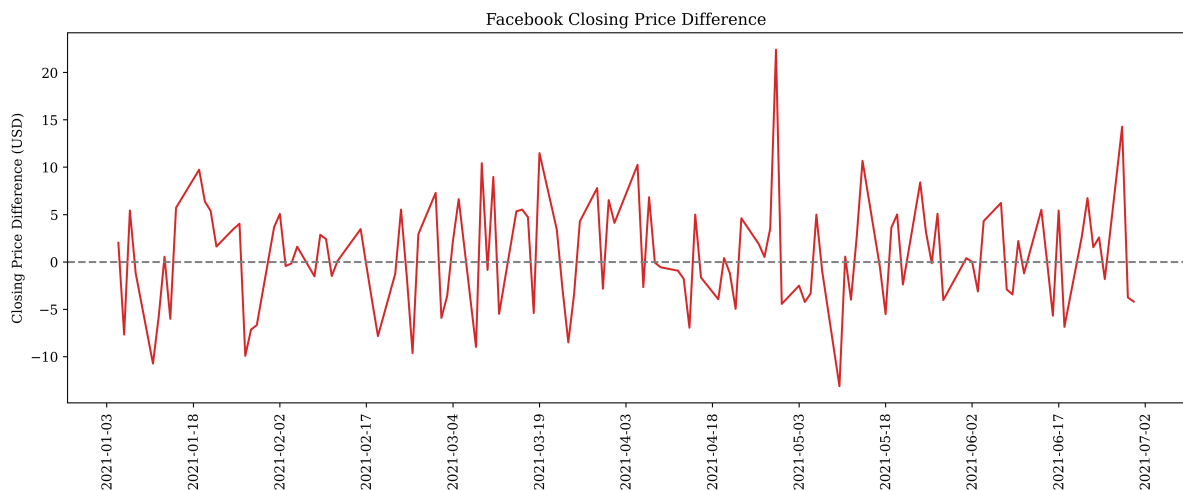


Fig. 3.2 The closing price difference for the Facebook dataset

Note how the closing price difference values in Figure 3.2 have more consistent statistical properties throughout the time series compared to the absolute closing price values in Figure 3.1. This ensures that the model is exposed to similar data in training and testing. Descriptive statistics of the closing price difference for each dataset are provided in Table 3.4.

Table 3.4 Descriptive statistics of the closing price difference for each dataset

<i>Closing Price Difference</i>	Dataset				
Descriptive Statistic	Apple	Microsoft	Amazon	Alphabet	Facebook
Count	123	123	123	123	123
Mean	0.06	0.43	2.06	6.33	0.64
Standard Deviation	2.2	3.32	47.59	34.17	5.6
Minimum	-5.08	-7.41	-107.74	-86.45	-13.11
Q_1	-1.15	-1.43	-30.32	-10.08	-3.37
Q_2	0.01	0.21	6.62	5.25	0.11
Q_3	1.63	2.96	34.6	20.42	4.67
Maximum	6.48	7.9	142.62	142.56	22.41

In conclusion, the target feature for this study has been selected as the closing price difference. That is, the models built will attempt to predict how the closing price will *change* from the current day to the next - rather than predicting the absolute closing price of the following day.

3.2 Sentiment Data

The sentiment data was prepared using the sentiment analysis pipeline illustrated in Figure 3.3. This pipeline consists of the following three steps:

1. Twitter Data – obtaining and pre-processing the Twitter data.
2. Sentiment Analysis – predicting the sentiment of the tweets in the Twitter data.
3. Levels of Sentiment – targeted sentiment using filters on the sentiment predicted tweets.

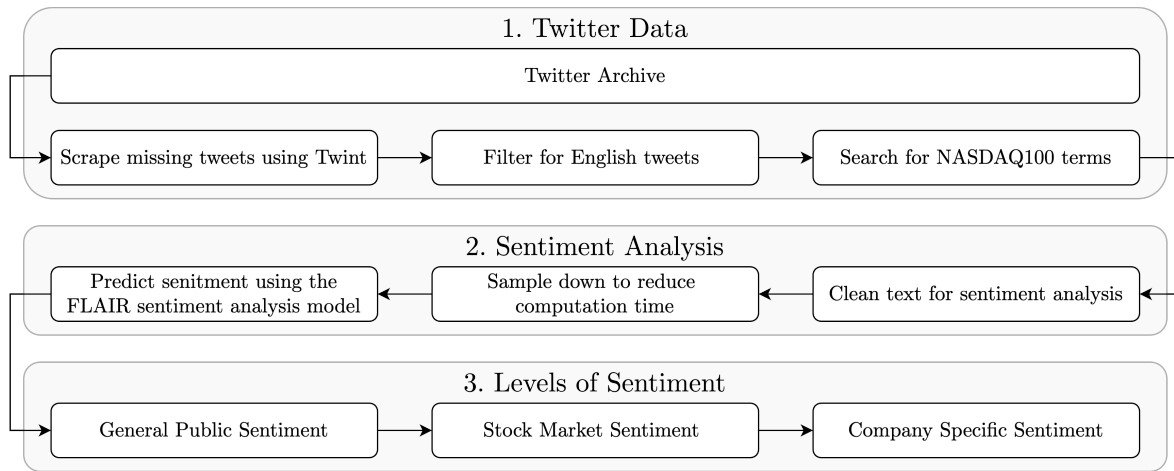


Fig. 3.3 The sentiment analysis pipeline used to process the Twitter data

What follows is a discussion of each step.

3.2.1 Twitter Data

The data used to predict the sentiment of public opinion has been retrieved from Twitter. The raw, unprocessed data is a collection of JavaScript Object Notation (JSON) grabbed from the main Twitter feed. The data is hosted on Internet Archive (<https://archive.org/details/twitterstream>). From this archive, 830 209 231 tweets (311.3GB) posted from January 2021 to June 2021 (6 months) has been extracted for this study. The number of tweets and the respective storage size (in GB) for each day in this period is illustrated in Figure A.2 in Appendix A.

The archive is incomplete in that for some days no tweets have been archived. No tweets were captured from 7 January 2021 to 24 January 2021 (18 days) and 11 April 2021 to 26 April 2021 (16 days). This is evident in Figure A.2 in Appendix A. Since the sentiment of each day will be a time series used as an input feature for the machine learning models, these missing values need to be addressed. Various interpolation techniques exist to replace these missing values, however considering the number of consecutive time steps (days) for which no tweets have been archived, adequate imputation could be difficult. Additionally, considering the importance of this feature in the context of this study, another solution is required.

This has been remedied by using a data mining technique called *web scraping*. Web scraping (simply termed *scraping* moving forward) is the process of using software bots to automate the repetitive task of extracting content and data from a website. This is

achieved by copying the underlying HyperText Markup Language (HTML) code. HTML is the code used to structure a webpage and its content. Tweets have been scraped from the main Twitter feed using the Twint package (<https://github.com/twintproject/twint>). The Twint package is a Twitter scraping tool (written in the Python programming language) that allows for scraping tweets from Twitter profiles *without* using Twitter's Application Programming Interface (API). The Twitter API only allows for the retrieval of the last 3200 tweets posted. Twint scrapes tweets from the main Twitter feed and therefore allows for the anonymous collection of all tweets with no limitations.

The scraping operation was configured to collect English tweets for the two periods (7 January 2021 to 24 January 2021 and 11 April 2021 to 26 January 2021) for which the archive is incomplete. The operation was configured to visit the main Twitter feed every 2 minutes, scrape as many tweets as possible, and save the position of the last scraped tweet. Saving the position of the last scraped tweet allows the operation to continue where it left off on the next visit.

The scrape operation collected approximately 7500 tweets per hour. Obtaining a large sample is very time-consuming. It was therefore decided to run the scraping operation until approximately 100 000 scraped tweets were collected for each day in the considered period. In comparison, this is a small daily sample size compared to that of the archive. However, considering the number of *consecutive* time steps (days) for which no tweets have been archived, this should outperform available interpolation techniques. In total, the scrape operation ran for 426 hours (17 days) to collect 3 268 705 scraped tweets (1.55GB). The number of tweets collected for each day in this period is illustrated in Figure A.3 and A.4 in Appendix A.

Each raw, unprocessed tweet is stored in the JSON format. An example of a tweet in this format is illustrated in Figure 3.4.

```
{
  "created_at": "Wed Jan 06 07:29:00 +0000 2021",
  "favorited": false,
  "id": 1346720407521550337,
  "lang": "en",
  "retweeted": false,
  "text": "
    Just watched the Apple 🍏 keynote(https://www.apple.com/apple-events/).
    He (@TimCook) is hitting it out the park.
    So glad I invested in $AAPL yesterday. To the moon! 🌕 ",
  "user": {
    "created_at": "Sat Dec 02 05:57:24 +0000 2017",
    "followers_count": 270,
    "id": 936836680136429568,
    "lang": "en",
    "name": "Bernard McDonald",
    "screen_name": "I'm just a Tr8derBoy"
  }
}
```

Fig. 3.4 An example of a tweet in JavaScript Object Notation

Every attribute of the tweet is stored as a key-value pair. To simplify the sentiment analysis, the tweets have been filtered to only include English tweets. That is, only tweets where the “*lang*” key has the “*en*” value is included.

Note that this data contains retweets. That is, if a user retweeted a tweet of another user, the text of that tweet will be considered twice. This ensures that the sentiment of tweets deemed important by multiple users are considered more than once and subsequently contributes a larger weight to the average sentiment for a given day.

After filtering for only English tweets and collecting scraped tweets for the two periods for which the archive was incomplete, the dataset consisted of 299 988 331 tweets (28.54GB). This is illustrated in Figure A.5 in Appendix A.

Users can use a “cashtag” when referring to a stock symbol or cryptocurrency. The functionality of a “cashtag” is similar to that of a “hashtag”. By using “tags”, tweets can be categorized and grouped together with similar “tags” – making the content more discoverable on platform searches. A cashtag is a company ticker symbol or cryptocurrency preceded by the U.S. dollar sign e.g., \$AAPL (Apple Inc.) or \$BTC (Bitcoin).

In the context of this study, cashtags can be used to identify tweets referring to one of the considered companies in Table 3.1. Note that this would require the user to *specifically*

use the ticker symbol. The most popular cashtags and frequencies of the considered companies in Table 3.1 is illustrated in Figure 3.5.

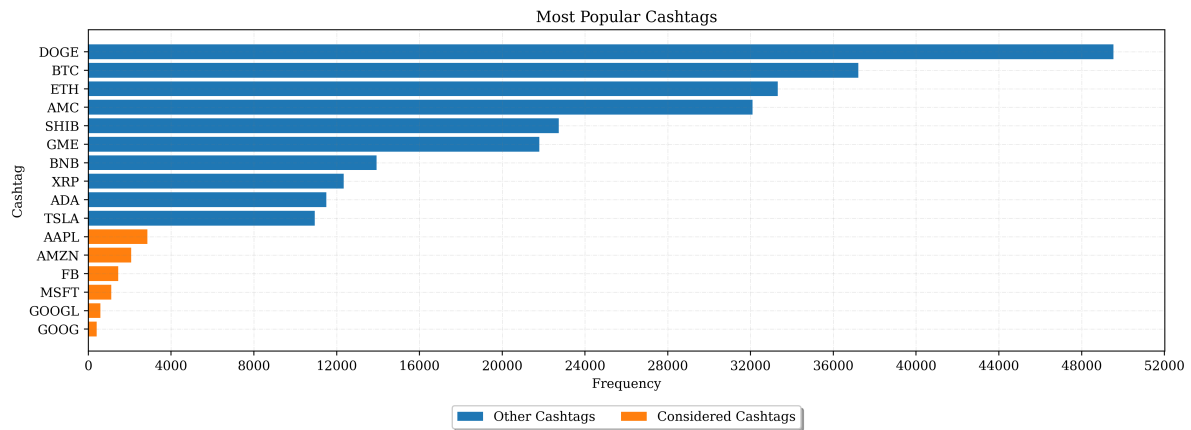


Fig. 3.5 The most popular cashtags identified in the Twitter data

Note that none of the considered companies in Table 3.1 are in the top 5 cashtags. Descriptions of these top 5 cashtags are provided in Table 3.5.

Table 3.5 The top 5 most popular cashtags in the Twitter data

Cashtag	Description	Frequency
DOGE	Dogecoin Cryptocurrency	49 542
BTC	Bitcoin Cryptocurrency	37 214
ETH	Ethereum Cryptocurrency	33 321
AMC	AMC Entertainment (offers movie theaters worldwide)	32 096
SHIB	Shiba Inu Cryptocurrency	22 733

From Table 3.5 it is evident that cryptocurrencies are the most popular financial instruments discussed on Twitter in this dataset. To identify more stock related tweets – especially those of the considered companies in Table 3.1 – an additional search was performed to identify tweets that contain terms related to all companies listed on the NASDAQ100. That is, the tweets have been searched to identify the names or ticker symbols of the companies listed on the NASDAQ100. The number of additional tweets identified for each considered company in Table 3.1 is illustrated in Figure 3.6.

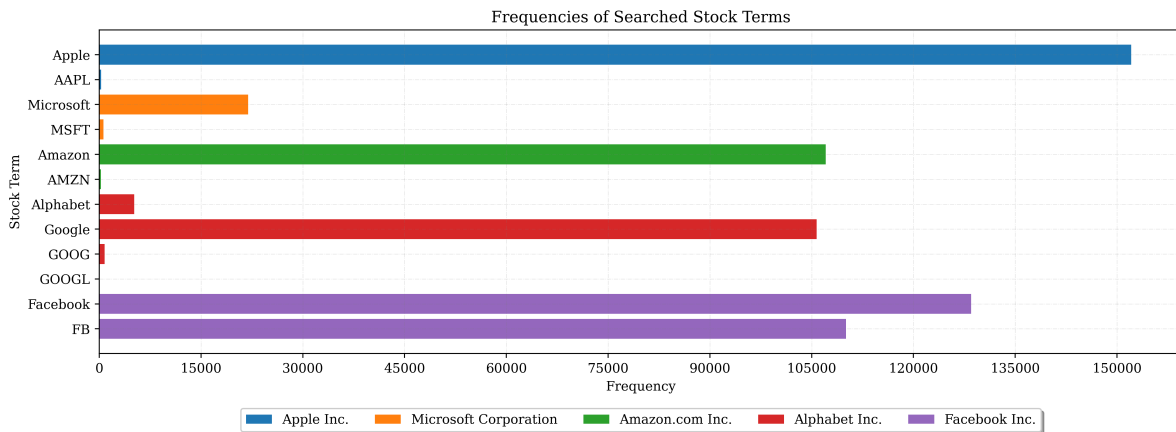


Fig. 3.6 The additional stock terms identified for the considered companies

3.2.2 Sentiment Analysis

After building the Twitter data, the state-of-the-art FLAIR sentiment analysis model (Akbik et al., 2019) has been used to assign a sentiment score to the text of each tweet. The model outputs a single unidimensional measure of sentiment for a given sentence on a scale from -1 (most extreme negative) to +1 (most extreme positive). The average of the sentiment scores for each sentence in a tweet is the final sentiment score assigned to that tweet.

Before the sentiment analysis model is used, it is beneficial to “clean” the text of each tweet. In the context of the sentiment analysis, “cleaning” includes removing text that does not contribute towards sentiment. That is, removing:

1. Twitter handles (e.g., @User)
2. Markup (e.g., italics, bold)
3. Hyperlinks (e.g., www.twitter.com)
4. Unnecessary punctuation characters (e.g., %, *, /, etc.)
5. Unnecessary white space (e.g., multiple spaces, newlines, tabs, etc.)

Next, each tweet is separated into sentences and each sentence is scored using the FLAIR sentiment analysis model. The average of these sentence sentiment scores is used to determine the final sentiment of the tweet. An example of scoring the sentiment of a tweet is illustrated in Figure 3.7.

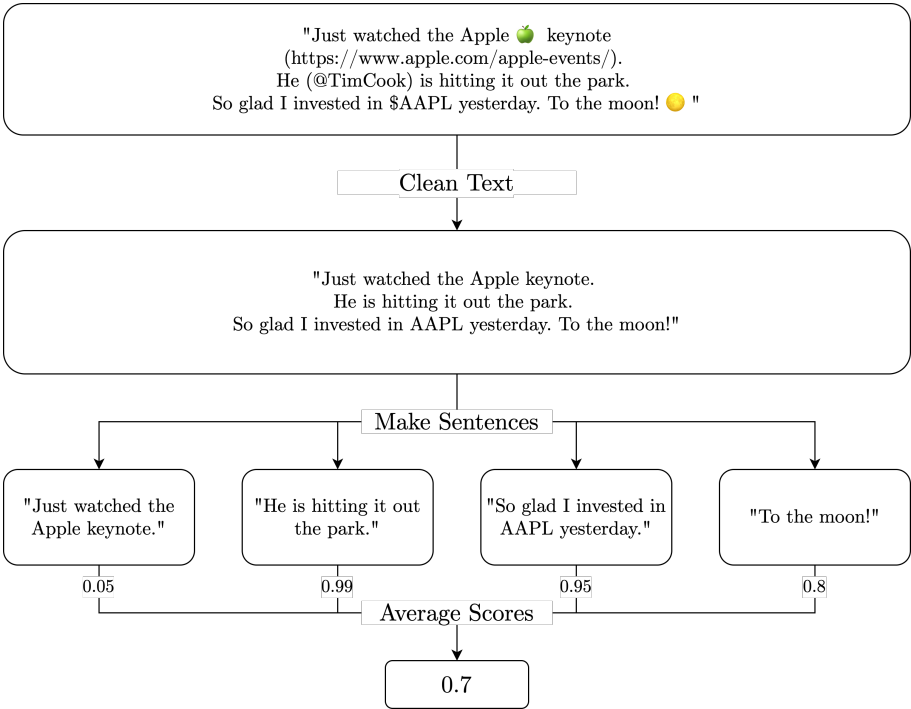


Fig. 3.7 An example of scoring the sentiment of a tweet

Predicting the sentiment of tweets at this scale is very computationally demanding. The components of the computer used in this study – especially those that could affect the compute time of the FLAIR sentiment analysis model – is provided in Table 3.6.

Table 3.6 Components of the computer used in this study

Component Name	Component Installed
Central Processing Unit (CPU)	Intel® Core™ i5-8400 CPU @ 2.8GHz, 6 Core(s), 6 Logical Processor(s)
Graphics Processing Unit (GPU)	NVIDIA GeForce GTX 1060 3GB
Random Access Memory (RAM)	2 x 8GB (16GB)
Storage	250GB Solid State Drive (SSD)

The FLAIR sentiment analysis model builds directly on PyTorch. PyTorch is an open-source machine learning framework (Paszke et al., 2019). PyTorch has been optimised to use CUDA. CUDA is a parallel computing platform and application programming interface that allows software to use certain types of graphics processing units for general purpose processing. This allows the model computations to run in parallel directly on

the Graphics Processing Unit (GPU) – rather than the Central Processing Unit (CPU) – and significantly increases the speed at which the model can produce predictions.

This GPU (specified in Table 3.6) has the computational capacity to predict the sentiment of 1 000 000 tweets (using the FLAIR sentiment analysis model) in 4 hours. Predicting the sentiment of all 299 988 331 tweets would require 1200 hours (50 days) of computation time. Given the time already spent on the scraping operation (17 days) and the time restrictions for this study, the number of tweets for sentiment prediction was filtered down by order of importance (in the context of this study) to allow for a more reasonable computation time. For each day, the following tweets have been included for the sentiment prediction:

1. All tweets containing a cashtag, NASDAQ100 ticker symbol, or NASDAQ100 company name.
2. From the remaining tweets, a random sample of 500 000 tweets.

This has reduced the number of tweets considered to 78 649 968 tweets and subsequently the required computation time to 314 hours (13 days). This is illustrated in Figure A.6 in Appendix A.

3.2.3 Levels of Sentiment

From the sentiment scored tweets data, three levels of sentiment have been extracted: company, stock market, and general public sentiment.

Company Sentiment

Tweets containing the name or ticker symbol of the company considered is used to determine the company level of sentiment. For a given day, the sentiment scores of these tweets are averaged to determine the final sentiment score for that day. The daily sentiment scores for Apple are illustrated in Figure 3.8. The daily sentiment scores for each of the considered companies is illustrated in Figure A.7 in Appendix A.

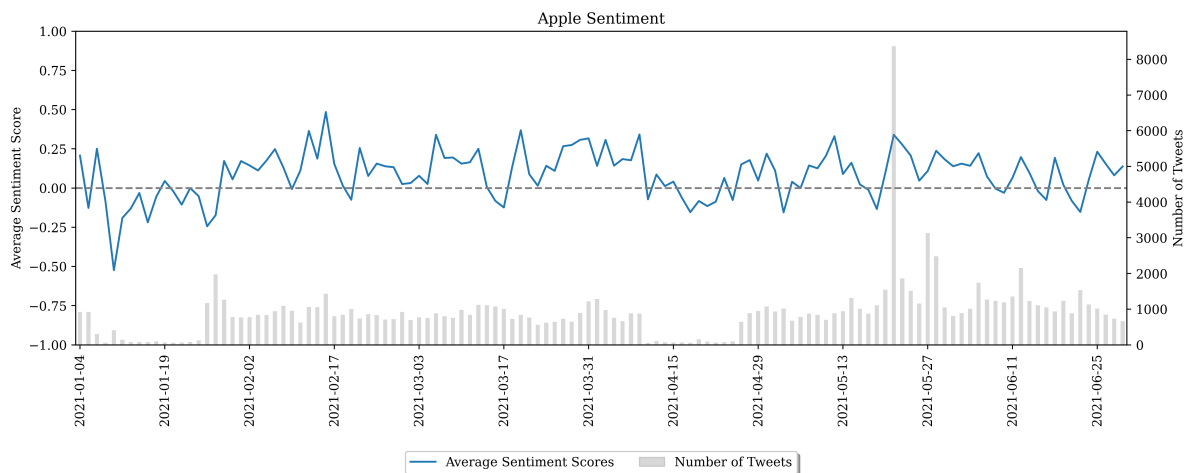


Fig. 3.8 The daily sentiment scores of the Apple dataset

Note that in Figure 3.8 there is significant increase in the number of tweets on the 21st of May 2021 with respect to Apple. This is the date that Apple started selling their new iMac, iPad Pro, and Apple TV 4K products ([Apple Newsroom, 2021](#)).

Stock Market Sentiment

Tweets containing the name or ticker symbol of any NASDAQ100 listed company is used to determine the stock market level of sentiment. Again, for a given day, the sentiment scores of these tweets are averaged to determine the final sentiment score for that day. The daily sentiment scores for the stock market are illustrated in Figure 3.9.

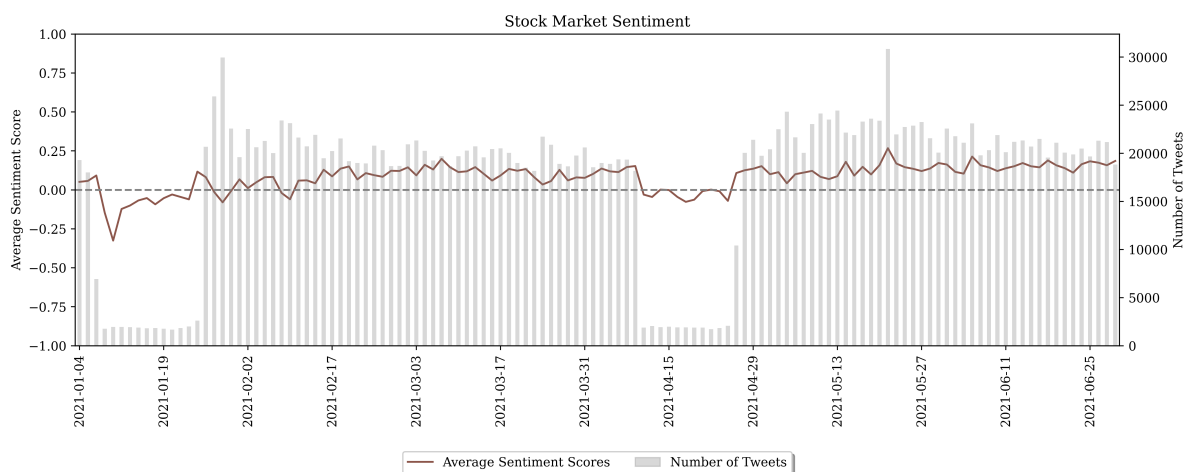


Fig. 3.9 The daily sentiment scores for the stock market

General Public Sentiment

Finally, the remaining tweets are used to determine the general public level of sentiment. That is, the tweets that do not contain any NASDAQ100 company names or ticker symbols. Again, for a given day, the sentiment scores of these tweets are averaged to determine the final sentiment score for that day. The daily sentiment scores for the general public are illustrated in Figure 3.10.

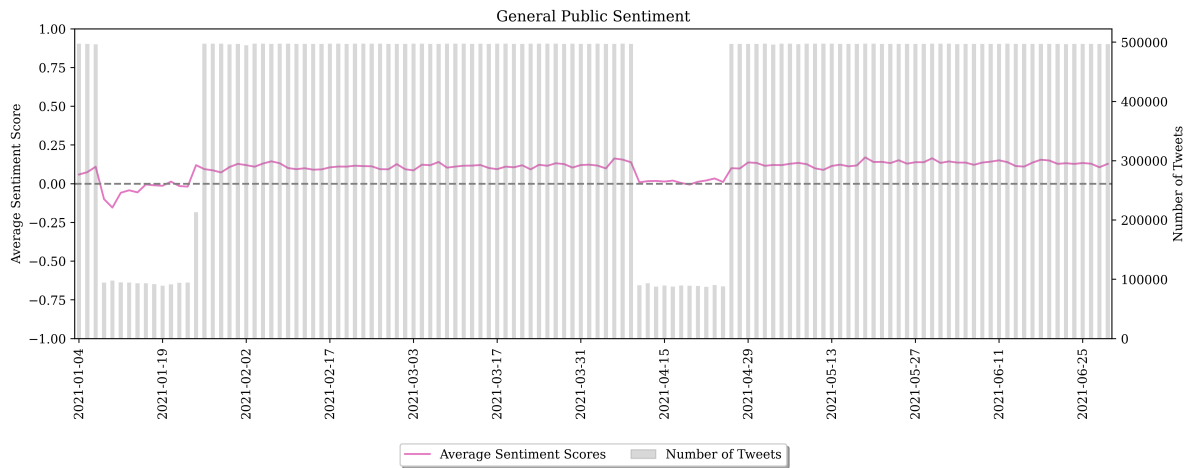


Fig. 3.10 The daily sentiment scores for the general public

In conclusion, the sentiment data consists of 7 features. Table 3.7 provides a description for each feature.

Table 3.7 Descriptions of each sentiment feature

Feature	Description
Apple Sentiment	The daily average sentiment of the tweets containing the cashtag \$APPL, the company name Apple, or the stock symbol AAPL.
Microsoft Sentiment	The daily average sentiment of the tweets containing the cashtag \$MSFT, the company name Microsoft, or the stock symbol MSFT.
Amazon Sentiment	The daily average sentiment of the tweets containing the cashtag \$AMZN, the company name Amazon, or the stock symbol AMZN.
Alphabet Sentiment	The daily average sentiment of the tweets containing the cashtags \$GOOG or \$GOOGL, the company names Google or Alphabet, or the stock symbols GOOG or GOOGL.
Facebook Sentiment	The daily average sentiment of the tweets containing the cashtag \$FB, the company name Facebook, or the stock symbol FB.
Stock Market Sentiment	The daily average sentiment of the tweets containing any company name or stock symbol listed on the NASDAQ100.
General Public Sentiment	The daily average sentiment of the tweets that do not contain any company names or stock symbols listed on the NASDAQ100.

Descriptive statistics for the general public sentiment (General Public Scores) and stock market sentiment (Stock Market Scores) features are provided in Table 3.8. Note that the General Public “Counts” and Stock Market “Counts” features represent the number of daily tweets considered to derive the General Public Scores and Stock Market Scores respectively. For brevity, a feature followed by “Counts” represents the number of daily tweets considered to derive the respective sentiment feature.

Table 3.8 Descriptive statistics of the general public and stock market sentiment features

<i>General Public and Stock Market</i>	Feature			
Descriptive Statistic	General Public Scores	General Public Counts	Stock Market Scores	Stock Market Counts
Count	124	124	124	124
Mean	0.10	422 929.38	0.08	17 268.15
Standard Deviation	0.06	156 689.72	0.09	7755.46
Minimum	-0.15	87 302.00	-0.33	1677.00
Q_1	0.09	496 954.75	0.05	18 668.00
Q_2	0.11	497 212.50	0.10	20 078.00
Q_3	0.13	497 381.50	0.14	21 548.25
Maximum	0.17	497 678.00	0.27	30 861.00

The general public sentiment considers an average of 422 929 tweets per day, with 87 302 tweets being the least number of tweets considered for a given day. This small minimum (compared to the mean) is attributed to the scrape operation performed for each day where no tweets were archived. For these days, approximately 100 000 scraped tweets were collected. An average score of 0.10 for the General Public Scores feature suggests that the general public appears to be slightly positive. However, a minimum and maximum score of -0.15 and 0.17 (and a standard deviation of 0.06) for the General Public Scores feature suggests that the general public sentiment does not deviate much on a daily basis from the neutral score (0).

The stock market sentiment considers an average of 17 268 tweets per day. An average score of 0.08 for the Stock Market Scores feature suggests that the stock market has a very neutral sentiment. However, a minimum and maximum score of -0.33 and 0.27 (and a standard deviation of 0.09) for the Stock Market Scores feature suggests that the stock market sentiment does deviate daily. The days for which the stock market sentiment does deviate from the neutral score (0) could inform a change in stock price for the following day.

Descriptive statistics for the company sentiment features are provided in Table 3.9.

Table 3.9 Descriptive statistics of the company scores sentiment features

<i>Company Scores</i>	Feature				
Descriptive Statistic	Apple	Microsoft	Amazon	Alphabet	Facebook
Count	124	124	124	124	124
Mean	0.08	-0.03	0.13	-0.26	-0.34
Standard Deviation	0.15	0.19	0.16	0.15	0.18
Minimum	-0.52	-0.67	-0.31	-0.50	-0.78
Q_1	-0.01	-0.13	0.06	-0.32	-0.43
Q_2	0.10	-0.01	0.14	-0.28	-0.35
Q_3	0.18	0.07	0.22	-0.23	-0.24
Maximum	0.48	0.66	0.74	0.97	0.97

The sentiment towards Apple and Amazon appear to be positive with mean daily sentiment scores of 0.08 and 0.13 respectively. The sentiment towards Microsoft, Alphabet, and Facebook appears to be negative with mean daily sentiment scores of -0.03, -0.26, and -0.34 respectively. The large standard deviations and ranges between the minimum and maximum for each feature suggests that the sentiment towards these considered companies vary from one day to the next. These variations in sentiment could inform future changes in stock price.

Descriptive statistics for the number of daily tweets considered to derive a company's sentiment are provided in Table 3.10. Again, note that a feature followed by "Counts" represents the number of daily tweets considered to derive the respective sentiment feature.

Table 3.10 Descriptive statistics of the company counts sentiment feature

<i>Company Counts</i>	Feature				
Descriptive Statistic	Apple	Microsoft	Amazon	Alphabet	Facebook
Count	124	124	124	124	124
Mean	902.77	144.98	669.27	410.64	483.65
Standard Deviation	843.31	87.38	425.93	190.15	312.40
Minimum	54.00	6.00	55.00	2.00	2.00
Q_1	664.50	108.75	524.00	356.75	344.00
Q_2	864.50	143.50	655.50	433.50	444.50
Q_3	1064.75	187.00	815.00	525.50	645.75
Maximum	8370.00	462.00	2442.00	893.00	1973.00

The sentiment for all the companies considered are derived from significantly fewer tweets (on average) than those of the general public sentiment and stock market sentiment. Apple is the most popular company discussed with an average of 903 tweets considered per day to derive company sentiment – far higher than any of the other companies considered. Microsoft is the least discussed company with an average of 145 tweets considered per day to derive company sentiment.

Note that for a given day the company sentiment for Alphabet and Facebook was derived from only 2 tweets. These do however appear to be outliers as the Microsoft Count and Facebook Count features have first quartiles of 356.75 and 344 respectively – suggesting a thin left tail for both their distributions. For the days where a company’s sentiment is derived from very few tweets, one could argue that the sample is too small to accurately measure sentiment. Additionally, this highlights the difficulty of obtaining tweets discussing a specific company.

Granger Causality Test

The Granger causality test ([Granger, 1969](#)) is a statistical hypothesis test used to determine whether one time series is useful in forecasting another. If this is the case, the first time series is termed to *Granger-cause* the second time series. Granger causality should not be confused with true causality and is rather a term used to describe temporally related time series. The Granger causality test tests the null hypothesis H_0 that time series t_a does not Granger-cause time series t_b . The alternative hypothesis H_A states

that the time series t_a does Granger-cause time series t_b . Again, the Granger causality test is simply used and not discussed as the derivation of this statistical test surpasses the scope of this study. These hypotheses can be expressed as:

Null Hypothesis (H_0): Time series t_a does not Granger-cause time series t_b

Alternative Hypothesis (H_A): Time series t_a does Granger-cause time series t_b

As previously concluded, the closing price difference is non-stationary and is therefore the target feature. Therefore, the company sentiment time series is tested for Granger causality with respect to the corresponding company closing price difference. Additionally, the stock market and general public sentiment is tested for Granger causality with respect to the closing price difference for each of the considered companies. Each Granger causality test was performed 5 times starting from lag 1 accumulating to lag 5. That is, for every time step y_t the previous n time steps of time series t_a are considered to test the ability to forecast time series t_b . This is repeated 5 times for $n = 1, 2, \dots, 5$. A summary of the hypothesis tests that produced $p < 0.10$ is provided in Table 3.11.

Table 3.11 The Granger causality hypothesis test results (for $p < 0.10$)

Dataset	Sentiment Feature	Lag	p-value
Apple	Company Scores	3	0.026
Apple	Company Scores	4	0.054
Microsoft	Company Scores	1	0.032
Microsoft	Company Scores	3	0.083
Microsoft	Company Scores	4	0.037
Microsoft	Company Scores	5	0.060
Alphabet	Company Scores	2	0.077
Alphabet	Company Scores	4	0.084
Facebook	Company Scores	2	0.060
Facebook	Company Scores	3	0.083
Amazon	Stock Market Scores	2	0.060
Facebook	Stock Market Scores	1	0.081
Facebook	Stock Market Scores	2	0.057
Facebook	Stock Market Scores	3	0.055

From Table 3.11, it can be concluded (at a 0.10 level of significance) that all company sentiment features – except for the Amazon sentiment – is Granger-causal for their respective company closing price difference. Additionally, the stock market sentiment is Granger-causal for the Amazon and Facebook closing price differences.

Chapter 4

Methodology

4.1 ARIMA Model

ARIMA models are typically specified by the tuple (p, d, q) , where p , d and q specify the order of the Auto-Regressive, Integrated, and Moving Average components respectively.

As previously concluded, using the ADF statistical hypothesis test, the order of differencing required to make the closing price stationary is $d = 1$.

For optimal predictive performance, the ARIMA model will be re-fitted at every time step. That is, the model will be re-fitted every day when new information is available (i.e., the closing price of the current day). Once refitted, the model predicts the closing price of the next day.

Traditionally, the order of the Auto-Regressive p and Moving Average q components are determined via the use of the Autocorrelation Function (ACF) and Partial Autocorrelation Function (PACF) respectively. The ACF investigates the correlation between the time series and its prior values. This is used to determine the number of correlated lagged values – and subsequently the number of Auto-Regressive terms to include in the model. The PACF investigates the correlation between the time series and its prior error values. This is used to determine the number of correlated lagged values – and subsequently the number of Moving Average terms to include in the model. However, since the model will be re-fitted at every time step, it would not be feasible to manually determine the optimal values for p and q . Therefore, an automated solution is used. For every model re-fitted, each combination of p and q (over a predefined range) is searched such that the

Akaike Information Criterion (AIC) is minimised. The AIC is defined as

$$\text{AIC} = -2\log(L) + 2(p + q + \alpha),$$

where L is the likelihood of the data, p is the order of the Auto-Regressive component, q is the order of the Moving Average component, and α is the intercept of the ARIMA model.

The range of values to consider for p and q need to be predefined. Given the small size of the datasets (124 observations each, as limited by the data and computational availability for the sentiment analysis) these values should be small enough to minimise the number of observations “lost” to training at initial time steps, but also large enough to give the model enough information for initial predictions. Therefore, the range of values considered for p and q have been selected as $[1, 5]$. Given larger datasets, future work could explore and test for larger values of p and q .

4.2 Machine Learning Models

4.2.1 Pre-Processing

Before the data can be used for modelling, it needs to be transformed into an appropriate format. This is termed *pre-processing* and for this study includes:

1. Partitioning
2. Feature Subsets
3. Normalisation

Partitioning

Each dataset is separated into training, validation, and testing partitions. The training partition is the subset of the examples used to train the model (fit the weight and bias parameters). The validation partition is the subset of the examples used to measure the cost of the model after each epoch. This is used during training as an unbiased evaluation of the current model performance and to identify overfitting of the training partition. The testing partition is the subset of the examples used to measure the cost of the model after training has been completed. This is used to determine if the model can generalise and fit new and unseen data outside of the training partition.

This partitioning is consistent for every dataset since each dataset is of equal size (124 observations). The training partition consists of the first 60% of the data (74 observations). The validation partition consists of the following 20% (25 observations). The testing partition consists of the remaining 20% (25 observations). The training, validation, and testing partitioning is illustrated for the closing price of the Amazon dataset in Figure 4.1.

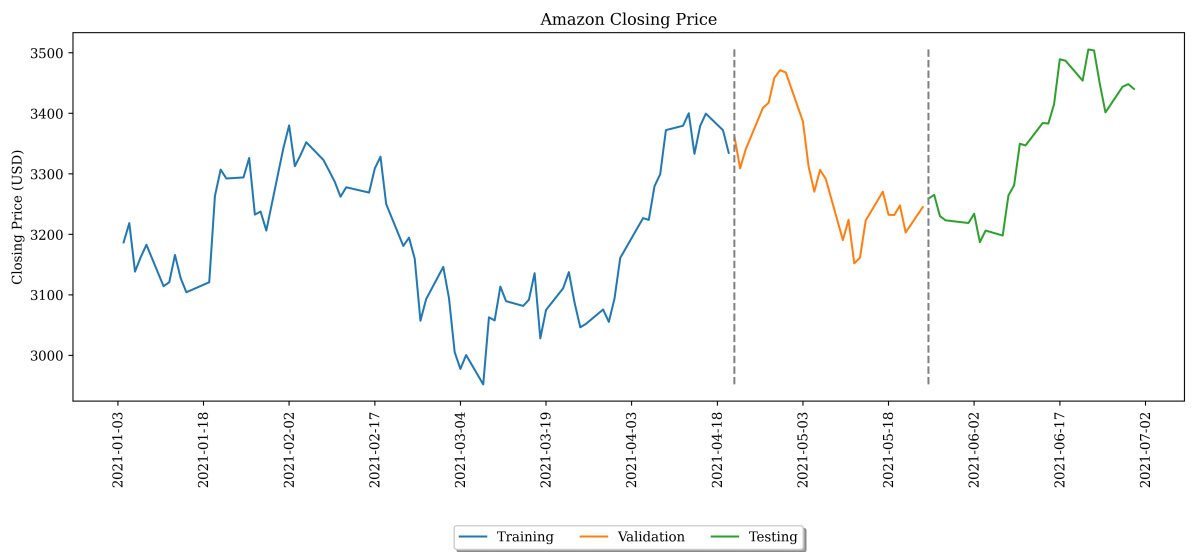


Fig. 4.1 The training, validation, and testing partitions for the closing price of the Amazon dataset

Feature Subsets

A component of this study is to test which subset of sentiment data (if any) contributes towards the predictive power of the models considered. That is, each model will be fitted with different subsets of the features to measure the difference in performance across the different subsets. Each feature subset tested for is defined in Table 4.1 and 4.2.

Table 4.1 Descriptions of each feature subset

Feature Subset	Description
Only Technical	Only technical data and no sentiment data
General Public Sentiment	Only technical data and general public sentiment
Stock Market Sentiment	Only technical data and stock market sentiment
Company Sentiment	Only technical data and company sentiment
All Data	All technical and sentiment data

Table 4.2 The features included in each feature subset

Feature	Feature Subset				
	Only Technical	General Public Sentiment	Stock Market Sentiment	Company Sentiment	All Data
Open	✓	✓	✓	✓	✓
High	✓	✓	✓	✓	✓
Low	✓	✓	✓	✓	✓
Close	✓	✓	✓	✓	✓
Volume	✓	✓	✓	✓	✓
General Public Scores		✓			✓
General Public Counts		✓			✓
Stock Market Scores			✓		✓
Stock Market Counts			✓		✓
Company Scores				✓	✓
Company Counts				✓	✓

Note that the number of tweets considered for a given day (General Public Counts, Stock Market Counts, Company Counts) for each respective sentiment feature (General Public Scores, Stock Market Scores, Company Scores) have been included as features. This could provide the model some additional information – possibly serving as a weighting to determine how much emphasis should be placed on a sentiment feature for a given day. Note that this hypothesis is not being tested. The model is provided with the available data to identify feature interactions during training.

Normalisation

Features with large values will have large contributions to the output error. Therefore, the optimiser will favour large valued features and neglect the information from small valued features ([Sola and Sevilla, 1997](#)).

The goal of normalisation is to change all the input features to a common scale, without distorting the differences in the ranges of the values. For example, note the difference in range between the volume feature and all other features in Figure 3.1. The minimum to

which the network converges can be shifted by a linear transformation. The distance to the desired local minimum is drastically changed by the scale of the data. Normalising the data and changing the range of each feature to $[0, 1]$ significantly improves the performance of the optimiser to converge to the local minimum (Sola and Sevilla, 1997).

The data is normalised using

$$X_{norm} = \frac{X - X_{min}}{X_{max} - X_{min}}$$

for each respective feature. If the value of $X = X_{min}$ the normalised value $X_{norm} = 0$. If the value of $X = X_{max}$ the normalised value $X_{norm} = 1$. If the value of $X \in (X_{min}, X_{max})$ the normalised value $X_{norm} \in (0, 1)$. This is also referred to as *min-max scaling* and ensures that every feature has the range $[0, 1]$.

Only the training partition is used to calculate the X_{min} and X_{max} constants. This ensures that scaling remains consistent with the scaling used during training and avoids data leakage but allows for new data to fall outside the range $[0, 1]$. It is difficult to determine the absolute minimum and maximum (considering all observable values) of each feature, however, the X_{min} and X_{max} constants can deviate from the absolute minimum and maximum by up to 30% with no significant impact on performance (Sola and Sevilla, 1997).

4.2.2 Training Examples

The data needs to be transformed into training examples to train the model. A training example consists of an input-output pair. The input is propagated through the network and the predicted output is measured against the expected output using a cost function. The training examples for the recurrent and Transformer models are structured differently. The recurrent models simply consist of a input-output pair, where the Transformer model has an encoder input and decoder input-output pair.

4.2.2.1 Recurrent Models

Before creating a training example, the size of the historic reference window needs to be selected. That is, the number of previous days to consider when predicting the change in closing price for the following day. Given the small size of the datasets (124 observations each, as limited by the data and computational availability for the sentiment analysis) the historic reference window should be small enough to allow for enough training examples to be created from the data, but also large enough to give the model enough information

for a given prediction. Therefore, the size of the historic reference window has been selected as 5 days. Given larger datasets, future work could explore and test for the optimal historic reference window size.

The input of a training example is a matrix where the number of columns vary in size depending on the number of features considered. For example, if the Technical Data feature subset (Table 4.1 and 4.2) is considered, the input of a training example is a 5×6 matrix and consists of each technical feature for the past 5 days. Note that the lagged target feature Close Diff is included as an input feature. The output of a training example is a single value and consists of the change in closing price of the following day. See Table 4.3 and 4.4 for the first input-output pair training example of the Apple dataset.

Note that the values in Table 4.3 and 4.4 would have been normalised (scaled to the range $[0, 1]$) before the training examples are created, however, these values have been left in their original state (before pre-processing) to provide additional clarity in the illustration.

Table 4.3 The input of a recurrent model training example

	Open	High	Low	Close	Volume	Close Diff
x_1	127.75	130.57	127.29	129.85	97 664 900	1.59
x_2	126.59	129.89	125.26	125.48	155 088 000	-4.37
x_3	127.22	130.47	126.73	129.76	109 578 200	4.28
x_4	131.26	131.46	129.08	130.88	105 158 200	1.12
x_5	128.05	129.02	127.36	127.84	100 384 500	-3.04

Table 4.4 The output of a recurrent model training example

	Close Diff
x_6	-0.18

The time series is traversed from start to finish and “cut” into a series of 5-day training examples. This results in 69 training examples, 19 validation examples, and 20 testing examples.

4.2.2.2 Transformer

The encoder has a reference window to previous time steps and learns which previous time steps to attend to. Similar to the historic reference window in the recurrent models, the input window for the encoder should be large enough to give the model enough information for a given prediction. Given the constraints of the small size of the datasets, the size of the encoder input window has been selected as 5 days – the same size as the historic reference window for the recurrent models. The encoder input is a matrix of 5 rows and columns that vary depending on the number of features selected.

Note that selecting the same size for both the recurrent models historic reference window and the Transformer model encoder input window prevents the Transformer model from utilising one of its largest strengths – the ability to learn long term dependencies. RNNs have a very limited reference window due to the vanishing gradient problem and therefore cannot learn long term dependencies. LSTMs and GRUs extends the reference window but are still limited due to the fundamental constraint of sequential computation. Transformers can have a much larger (technically infinite) reference window and this is *not* being explored since small sized historic reference windows are used. This could be further explored in a simulation study. See Table 4.5 for the first encoder input of the Apple dataset.

Table 4.5 The encoder input of a Transformer training example

	Open	High	Low	Close	Volume	Close Diff _{<i>t</i>-1}	Close Diff
x_1	127.75	130.57	127.29	129.85	97 664 900	1.59	-4.37
x_2	126.59	129.89	125.26	125.48	155 088 000	-4.37	4.28
x_3	127.22	130.47	126.73	129.76	109 578 200	4.28	1.12
x_4	131.26	131.46	129.08	130.88	105 158 200	1.12	-3.04
x_5	128.05	129.02	127.36	127.84	100 384 500	-3.04	-0.18

Note the inclusion of the Close Diff_{*t*-1} feature. This is the closing price difference of the previous day. This is required in the decoder input and is added to the dataset during pre-processing before training examples are created. Therefore, the Close Diff_{*t*-1} feature is included in the encoder input as well. Also note that the first observation in the dataset is removed when creating this feature as the previous closing price difference cannot be calculated for this observation.

The Transformer model is autoregressive in that each prediction considers all previous predictions (and the output of the encoder). When predicting a large sequence it makes sense to inform the next prediction using the information from all previous predictions. For example, consider a translation task. It makes sense to consider all previously predicted words when predicting the next word. However, in this regression task only a single future observation is to be predicted (not a sequence). By limiting the size of the decoder input, predictions are (mostly) based on previously *known* values (not previous predictions). Therefore, the size of the decoder input window has been selected as 2 days. That is, the decoder input is a matrix of 2 rows and columns that vary depending on the number of features selected. The size of the decoder output window corresponds to that of the decoder input window. See Table 4.6 and 4.7 for the first decoder input-output pair of the Apple dataset.

Table 4.6 The decoder input of a Transformer training example

	Open	High	Low	Close	Volume	Close Diff _{<i>t-1</i>}	Close Diff
x_6	127.36	128.54	125.74	127.66	91 951 100	-0.18	-0.18
x_7	127.62	130.29	127.35	129.73	88 636 800	2.07	2.07

Table 4.7 The decoder output of a Transformer training example

	Close Diff
x_7	2.07
x_8	-1.96

In the decoder input, the Close Diff feature is used as a placeholder to assume the values of previous predictions and subsequently inform the next prediction. Note that initially the target feature Close Diff assumes the same values as the previous day's target Close Diff_{*t-1*}. This is temporary, and the values of this feature are updated after each prediction.

To better illustrate the autoregressive nature of the Transformer model, the process of producing a prediction is discussed step-by-step in detail. This is discussed in tandem with Figure 4.2 – illustrating the exact Transformer architecture used in this study.

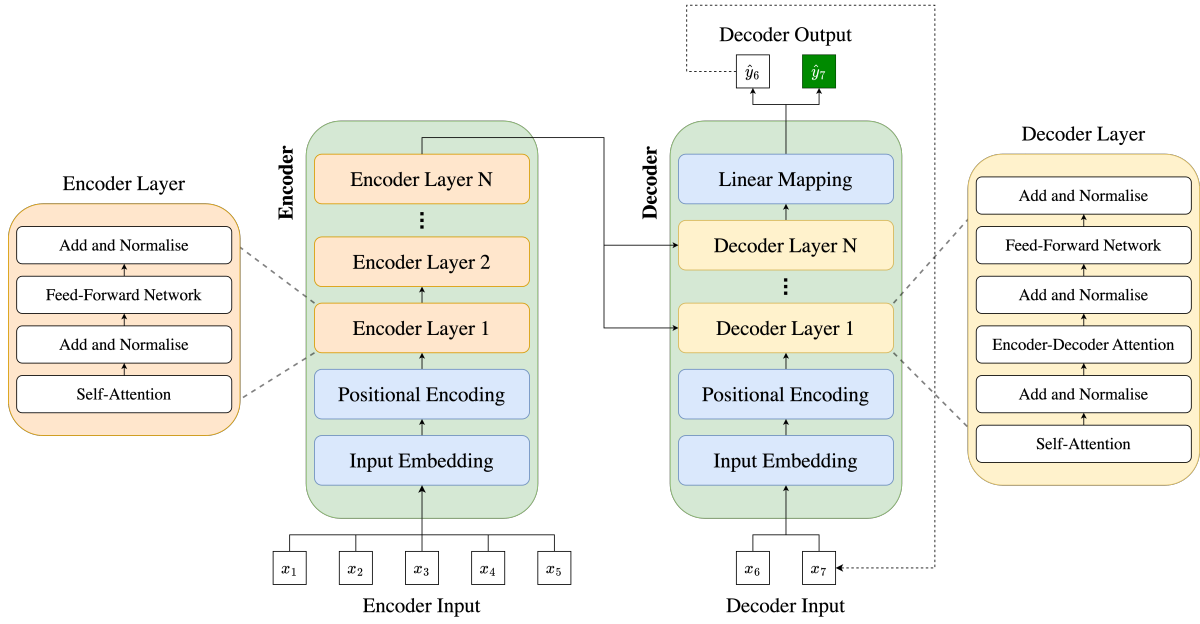


Fig. 4.2 A training example illustrated on the Transformer architecture

Step 1

The first time step of the decoder input x_6 is used as a “start token” to produce a first prediction $\hat{y}_6$. Here the term *token* is borrowed from NLP – as the Transformer was originally built for a NLP use case (Vaswani et al., 2017). A token is an instance of a sequence of characters grouped together as a semantic unit for processing. Simply put, a token is a component of a sentence, such as a word, space, punctuation, etc. A *start token* identifies the start of a sentence and is used to inform the model that a new sentence is starting to be predicted. Since our data does not consist of tokens (and specifically does not contain a start token), the next time step x_6 is used as a start token to produce the first prediction $\hat{y}_6$. This is illustrated in Table 4.8 and Figure 4.2 (the first time step of the decoder input x_6 and output $\hat{y}_6$).

Table 4.8 The first autoregressive step of a Transformer training example

Decoder Input							
	Open	High	Low	Close	Volume	Close Diff _{<i>t</i>-1}	Close Diff
<i>x</i> ₆	127.36	128.54	125.74	127.66	91 951 100	-0.18	-0.18

Decoder Output

Close Diff
$\hat{y}_6$

Step 2

The first prediction $\hat{y}_6$ is added to the decoder input and a prediction for the second time step $\hat{y}_7$ is produced. The next prediction $\hat{y}_7$ is produced using

1. The encoder output
2. The first time step of the decoder input x_6
3. The second time step of the decoder input x_7 (where the target feature Close Diff has been replaced with the prediction $\hat{y}_6$).

This is illustrated in Table 4.9 and Figure 4.2 (the dashed line from the first time step of the decoder output $\hat{y}_6$ to the second time step of the decoder input x_7). Here, the autoregressive nature of the Transformer model is clearly illustrated as the prediction $\hat{y}_6$ is used to predict $\hat{y}_7$.

Table 4.9 The second autoregressive step of a Transformer training example

Decoder Input							
	Open	High	Low	Close	Volume	Close Diff _{<i>t-1</i>}	Close Diff
x_6	127.36	128.54	125.74	127.66	91 951 100	-0.18	-0.18
x_7	127.62	130.29	127.35	129.73	88 636 800	2.07	$\hat{y}_6$

Decoder Output

Close Diff
$\hat{y}_7$

Selecting the size of the decoder input window as 2 days limits the autoregressive nature of the Transformer model to use only a single prediction to produce the final prediction. This could be both an advantage or disadvantage. The model could use the first prediction as additional information to better inform the current “thinking” of the model and produce a more accurate final prediction. Alternatively, the first prediction could be incorrect, forcing the model to base the final prediction on incorrect information, making it harder to produce an accurate final prediction. Note that this hypothesis is not being tested. Future research could explore the effects of varying the number of predictions made to inform the final prediction.

4.2.3 Modelling

The RNN, LSTM, and GRU models are built using the TensorFlow machine learning framework (Abadi et al., 2016) based on the Keras library (Chollet et al., 2015). The Transformer model is built using the PyTorch machine learning framework (Paszke et al., 2019) based on the Torch library (Collobert et al., 2011). All the code for the pre-processing, modelling, post-processing, and evaluation is written in the Python programming language.

4.2.3.1 Callbacks

The Keras API documentation defines a callback as “a set of functions to be applied at given stages of the training procedure”. To optimise the training procedure for this study, two callbacks have been implemented.

Learning Rate Scheduler

The first callback implemented is termed a *Learning Rate Scheduler* and reduces the learning rate once no improvement has been detected for a specified number of epochs. This specified number of epochs is termed the *patience* of the callback. The patience of this callback has been set to 10. The starting learning rate is set at 0.01. Once the callback is triggered (no reduction in validation cost has been detected in the previous 10 epochs) the learning rate is reduced by a factor of 0.5. Reducing the learning rate reduces the size of the adjustments made to the weight and bias parameters – potentially allowing the model to converge into a local minimum slowly and precisely. This callback can be triggered until the specified minimum learning rate of 0.00001 is reached.

Early Stopping

The second callback implemented is termed *Early Stopping* and stops training once no improvement (or improvement below a specified threshold) has been detected for a specified number of epochs. The patience for this callback has been set to 50 epochs. That is, training is stopped once no reduction in validation cost has been detected in the previous 50 epochs. During training this callback stores the weight and bias parameters for each of the previous 50 epochs. Once this callback is triggered, the weight and bias parameters that produced the smallest validation cost are reinstated. If this callback is never triggered, training will stop once the maximum number of specified epochs have been reached. The maximum number of epochs has been set to 1000.

4.2.3.2 Hyperparameters

A parameter used to control training is termed a *hyperparameter* and is set before the training procedure occurs.

The output layer uses the linear activation function defined as

$$f(x) = x.$$

This activation function has been selected as the change in closing price $y \in R$.

Weights are initialised using the Glorot normal initialiser, also called the Xavier normal initialiser ([Glorot and Bengio, 2010](#)). Samples are drawn from a truncated normal distribution with

$$\mu = 0 \quad \text{and} \quad \sigma = \sqrt{\frac{2}{\text{fan in} + \text{fan out}}},$$

where fan in and fan out are the number of input and output units in the weight tensor. A tensor (in the context of machine learning models) is data organised in a multidimensional array. This is the default weight initialiser as recommended by the TensorFlow API documentation.

Biases are initialised with zeros as recommended by the TensorFlow API documentation.

The cost function used to train the model is the Mean Square Error (MSE) function defined as

$$C = \frac{1}{N} \sum (y - \hat{y})^2.$$

The weight and bias parameters are optimised using SGD. As previously discussed, SGD uses the gradient of the cost function ∇C and updates the weight and bias parameters using

$$\theta = \theta - \eta \nabla_{\theta} C,$$

where θ is a weight or bias parameter.

Momentum ([Qian, 1999](#)) can be used to accelerate SGD and dampen oscillations by adding a fraction γ of the previous update vector to the current update vector. Therefore, the weight and bias parameters are updated using

$$\begin{aligned} \nu_t &= \gamma \nu_{t-1} + \eta \nabla_{\theta} C \\ \theta &= \theta - \nu_t, \end{aligned}$$

where ν_t is the update vector at time t . The momentum term ν is set to 0.9 as recommended by [Ruder \(2016\)](#).

Nesterov accelerated gradient ([Nesterov, 1983](#)) gives the momentum term prescience. Calculating $\theta - \gamma \nu_{t-1}$ approximates the next value of the parameter. Including this term when calculating the update vector calculates the gradient with respect to the approximate future value of the parameter, rather than the current value of the parameter. Finally, the weight and bias parameters are updated using

$$\begin{aligned} \nu_t &= \gamma \nu_{t-1} + \eta \nabla_{\theta} C(\theta - \gamma \nu_{t-1}) \\ \theta &= \theta - \nu_t. \end{aligned}$$

4.2.3.3 Hyperparameter Tuning

Hyperparameter tuning is the process of selecting the optimal hyperparameters. Simply selecting arbitrary hyperparameters could result in sub-optimal model performance. In contrast, finding the optimal hyperparameters over a given set of options could be beneficial in improving model performance. To ensure that all available options are exhausted, all permutations of the hyperparameter options are considered. This is considered a *grid* of hyperparameters, and a *grid search* is performed to determine the optimal hyperparameters. The grid size has an exponential relationship with the number of hyperparameter options considered. Therefore, the number of hyperparameter options considered need to be limited with respect to available computation power and time.

The RNN, LSTM, and GRU recurrent models share a similar architecture, and subsequently, hyperparameters. Therefore, the same grid search is used for the recurrent models. All hyperparameters considered for the recurrent models are listed in Table 4.10.

Table 4.10 The hyperparameter grid for the recurrent models

Hyperparameter	Options	Description
Random Seed	1, 2, 3, 4, 5	The random seed used to sample the initial weight parameters from the Glorot normal distribution.
Batch Size	2, 4, 8	The number of training examples used to calculate the adjustments made to the weight and bias parameters in a single training loop.
Number of Layers	1, 2, 3	The number of layers in the neural network.
Number of Units	16, 32, 64	The size of the hidden vector h_t .
Dropout	0, 0.5	The fraction of randomly selected neurons to ignore during a single training loop – used to reduce overfitting.

The hyperparameter options listed in Table 4.10 result in a grid size of 270 hyperparameter permutations. Note that each of these hyperparameter permutations are considered for each feature subset option listed in Table 4.2. This results in 6750 fits per model and a total of 20 250 fits for the combined RNN, LSTM and GRU recurrent models.

Given the different architecture of the Transformer model, the hyperparameter options differ from those of the recurrent models. All hyperparameters considered for the Transformer model are listed in Table 4.11.

Table 4.11 The hyperparameter grid for Transformer model

Hyperparameter	Options	Description
Random Seed	1, 2, 3, 4, 5	The random seed used to sample the initial weight parameters from the Glorot normal distribution.
Batch Size	2, 4, 8	The number of training examples used to calculate the adjustments made to the weight and bias parameters in a single training loop.
Number of Embeddings	8, 32	The number of embedding vectors to map the encoder and decoder input to.
Number of Channels	8, 32	The size of the embedding vectors d_{model} to map the encoder and decoder input to.
Number of Encoder and Decoder Layers	1, 2, 3	The number of encoder and decoder layers.
Number of Attention Heads	1, 2, 4	The number of attention heads in the multi-head attention mechanism.
Dropout	0, 0.5	The fraction of randomly selected neurons to ignore during a single training loop – used to reduce overfitting the training data.

The hyperparameter options listed in Table 4.11 result in a grid size of 1080 hyperparameter permutations. Again, each of these hyperparameter permutations are considered for each feature subset option listed in Table 4.2. This results in a total of 27 000 fits for the Transformer model.

4.2.4 Post-Processing

The model output needs to be transformed back into an appropriate format for interpretation. This is termed *post-processing* and, in this case, includes

1. Rescaling Predicted Output

2. Prediction Transformation

The predicted output is rescaled to the original range of the closing price difference feature using

$$Y = Y_{norm} \cdot (X_{max} - X_{min}) + X_{min} ,$$

where Y is the rescaled predicted output, Y_{norm} is the normalised predicted output, and X_{min} and X_{max} are the constants used for the normalisation of the closing price difference feature during pre-processing.

Once rescaled, the prediction is transformed from the *change* in closing price to the *absolute* closing price of the next day. This is achieved by adding the closing price of the current day to the prediction.

4.2.5 Evaluation

The evaluation of this study considers three different components:

1. Model Performance
2. Sentiment Analysis
3. Model Computational Training Efficiency

Model Performance

The first component of this study will compare the performance of the RNN, LSTM, and GRU recurrent models against the attention-based Transformer model for a future stock price prediction regression task.

The optimal hyperparameters are first determined using a grid search as previously defined. Once these optimal hyperparameters have been determined, each of these models are assessed using the Root Mean Squared Error (RMSE) on the testing partition. The RMSE is defined as

$$\text{RMSE} = \sqrt{\frac{1}{N} \sum (y - \hat{y})^2} .$$

Note that the MSE is used as the cost function to *fit* the model, but the RMSE of the testing partition to *evaluate* the final model performance. The RMSE is measured in the same units as the target feature and is therefore more intuitive to interpret.

Each model is compared to a benchmark. The benchmark used is predicting no change in closing price and is defined as

$$\hat{y}_{t+1} = y_t.$$

That is, predicting the closing price of the next day will remain unchanged and the same as the current closing price.

The models of the overall best performing dataset and the overall worst performing dataset will be discussed.

Sentiment Analysis

The second component of this study will compare the predictive performance (if any) of the different levels of sentiment. Predictive performance of the 5 different feature subsets defined in Table 4.1 and 4.2 will be assessed to determine which (if any) levels of sentiment are most predictive. The feature subsets tested for are:

1. Only Technical
2. Company Sentiment
3. Stock Market Sentiment
4. General Public Sentiment
5. All Data

These feature subsets will be assessed across all the models and datasets considered to test thoroughly for the predictive performance of the sentiment data. After performing the hyperparameter tuning, the number of times each feature subset has been selected as optimal will be used to inform which level of sentiment is most predictive.

The feature subsets of the best performing model-dataset combination and the worst performing model-dataset combination will be discussed.

Model Computational Training Efficiency

The third and final component of this study will compare the training efficiency of each model by comparison of training time on the same computational resources. Three factors, namely

1. Number of parameters to be trained (training parameters)
2. Number of epochs to train for (training epochs)
3. Training time

are investigated to determine which architecture (recurrent or attention-based) is more computationally efficient to train.

Chapter 5

Results

5.1 Model Performance

A grid search has been performed to determine the optimal hyperparameters. The optimal hyperparameters for the RNN model for each dataset are provided in Table 5.1. The optimal hyperparameters for the LSTM and GRU models for each dataset are provided in Table B.2 and B.3 in Appendix B.

Table 5.1 The optimal hyperparameters for the Recurrent Neural Network model per dataset

<i>RNN</i>	Dataset				
Hyperparameter	Apple	Microsoft	Amazon	Alphabet	Facebook
Feature Subset	Only Technical	General Public Sentiment	Only Technical	General Public Sentiment	Stock Market Sentiment
Batch Size	4	4	2	2	4
Number of Layers	3	2	2	1	1
Number of Units	32	32	32	32	64
Dropout	0.5	0	0	0	0.5
Test RMSE	1.47	2.35	37.25	16.66	5.03

The optimal hyperparameters for the Transformer model for each dataset are provided in Table 5.2.

Table 5.2 The optimal hyperparameters for the Transformer model per dataset

<i>Transformer</i>	Dataset				
Hyperparameter	Apple	Microsoft	Amazon	Alphabet	Facebook
Feature Subset	General Public Sentiment	Only Technical	Only Technical	Stock Market Sentiment	Stock Market Sentiment
Batch Size	2	2	2	2	2
Number of Embeddings	32	32	32	8	32
Channels	32	8	8	8	32
Number of Encoder Layers	3	3	2	1	3
Number of Decoder Layers	3	3	2	1	3
Number of Attention Heads	2	1	4	1	4
Dropout	0.5	0.5	0.5	0.5	0.5
Test RMSE	1.22	2.07	35.54	16.61	4.95

Descriptive statistics of the grid search test RMSE for each model are provided in Table 5.3. Test RMSEs larger than 1000 have been excluded as outliers.

Table 5.3 Descriptive statistics of the test RMSEs produced in each hyperparameter grid search

<i>Test RMSE</i>	Model			
Descriptive Statistic	RNN	LSTM	GRU	Transformer
Count	6750	6750	6660	27 000
Mean	14.65	13.90	19.28	17.58
Standard Deviation	22.23	15.41	49.70	19.99
Minimum	1.26	1.31	1.24	0.94
Q_1	2.51	2.51	2.58	2.33
Q_2	5.13	5.09	5.13	5.79
Q_3	18.82	18.21	19.01	34.85
Maximum	998.22	55.43	957.49	141.67

The large standard deviations and ranges between the minimum and maximum test RMSE for each model suggests that the effort of performing the grid search was worthwhile and that arbitrarily selecting hyperparameters for each model would result in sub-optimal performance.

The first component of this study compared the performance of the RNN, LSTM, and GRU recurrent models against the attention-based Transformer model for a future stock price prediction regression task. This was evaluated by comparing the performance of the RMSE on the testing partition. The test RMSE for each model-dataset combination is provided in Table 5.4. The best and worst performing model-dataset combinations are highlighted in green and red respectively. The worst performing machine learning model-dataset combinations are highlighted in orange.

Table 5.4 The test RMSE for each model-dataset combination

<i>Test RMSE</i>	Dataset				
Model	Apple	Microsoft	Amazon	Alphabet	Facebook
Benchmark	1.36	2.39	38.43	17.07	5.04
ARIMA	1.73	3.20	41.34	33.81	6.56
RNN	1.47	2.35	37.25	16.66	5.03
LSTM	1.42	2.35	39.57	16.88	5.01
GRU	1.47	2.45	40.35	16.96	5.02
Transformer	1.22	2.07	35.54	16.61	4.95

As indicated in Table 5.4, all the machine learning models outperformed the ARIMA model, and the attention-based Transformer model outperformed the recurrent models (in every dataset considered). This suggests that, of all the models considered, machine learning models – especially the Transformer model – are better able to capture the complex patterns within the stock data and subsequently better predict future stock price.

Next, the best and worst performing datasets are inspected to better understand the predictive performance of each model. Similar plots for all other datasets are provided in Appendix B.

Overall, all the models performed best on the Apple dataset with test RMSEs ranging between 1.22 and 1.73. The closing price difference and closing price predictions for the Apple dataset on the training, validation, and testing partitions for each model are illustrated in Figure 5.1 and 5.2. A closer look at the predictions on the testing partition is illustrated in Figure 5.3 and 5.4.

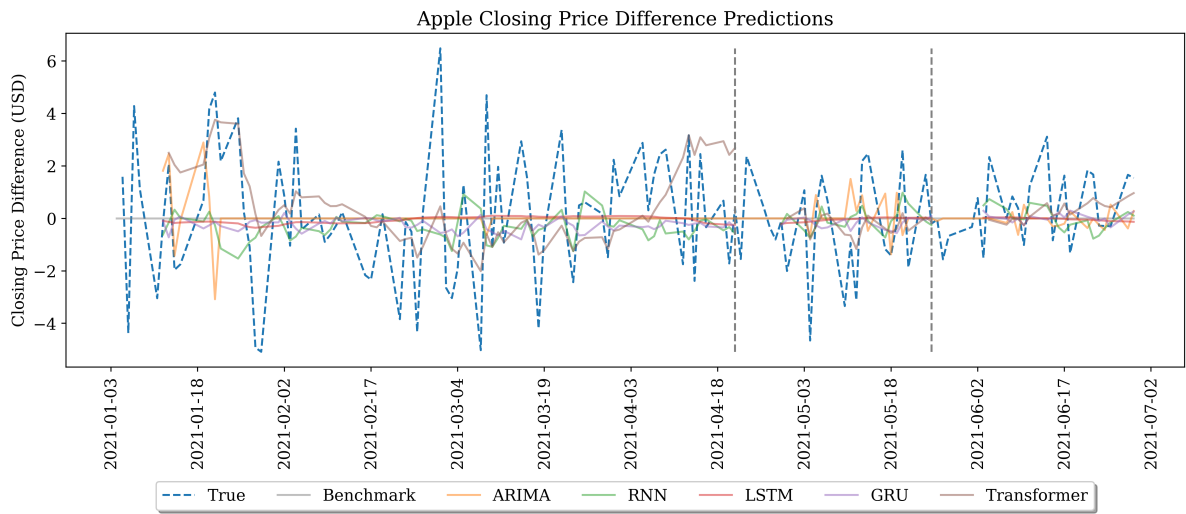


Fig. 5.1 The closing price difference predictions for the Apple dataset

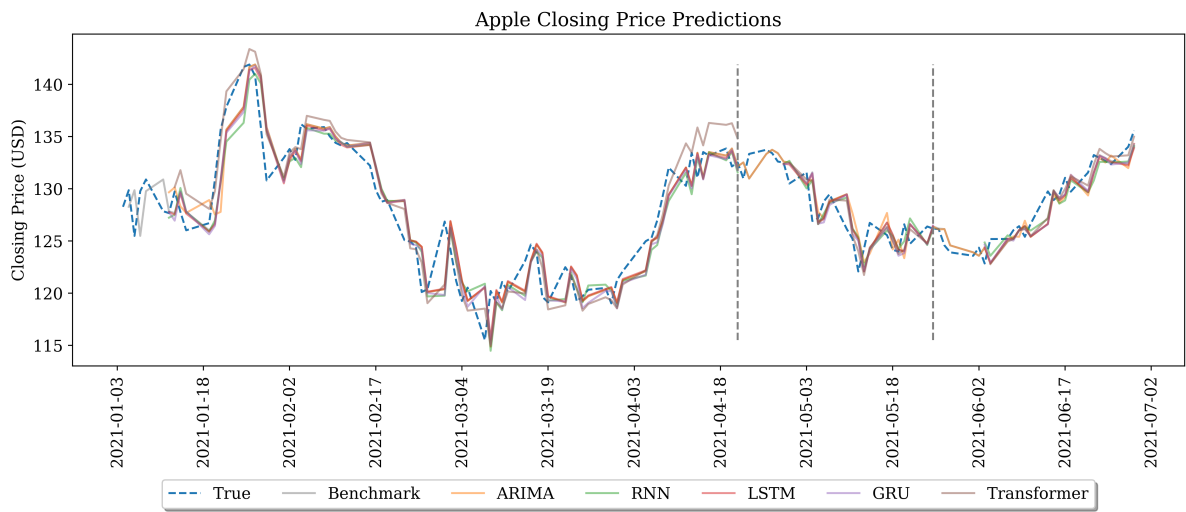


Fig. 5.2 The closing price predictions for the Apple dataset

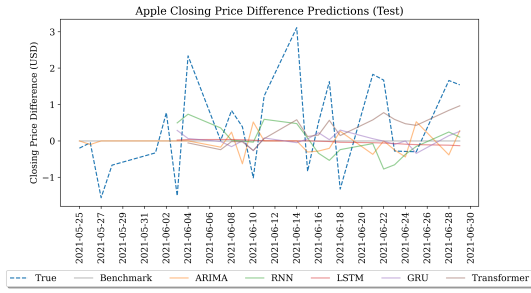


Fig. 5.3 The closing price difference predictions of the testing partition for the Apple dataset

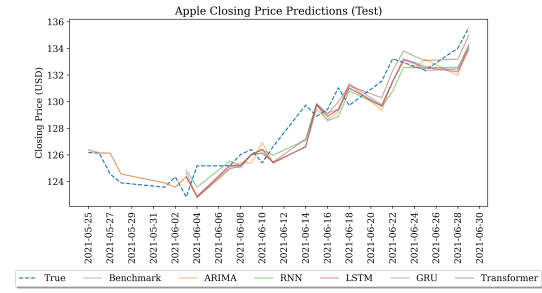


Fig. 5.4 The closing price predictions of the testing partition for the Apple dataset

Figure 5.1 and 5.3 suggests that the LSTM model is conservative. The LSTM model consistently predicts almost no change in stock price with very little deviations from zero. This could suggest that the LSTM model considers the change in stock price random, and therefore predicting that no change will occur is the optimal solution to minimise the error. This is not a bad strategy considering the benchmark (predicting no change in closing price) performs second best with a test RMSE of 1.36 (provided in Table 5.4).

In Figure 5.2 the predictions of all the models appear to have a similar shape to that of the true stock price – but lagged by a single day. This is caused by the model predicting the *change* in closing price. The model uses the closing price of the current day as a starting point for a given prediction. By not predicting a significant change in price, the model simply predicts that the closing price of the next day will remain unchanged and the same as the closing price of the current day – subsequently causing the lagged visual effect in Figure 5.2.

Figure 5.2 and 5.4 illustrates that the Transformer model attempts to predict the change in closing price with large deviations from zero. The LSTM almost predicts no change in closing price with very small deviations from zero. The RNN and GRU models attempt to predict the change in closing price – albeit not very confidently – with moderate sized deviations from zero. The Transformer model not only predicts significant changes in stock price with large deviations from zero but succeeds (more than any other model) with the smallest test RMSE of 1.22 (provided in Table 5.4).

Overall, the models performed worst on the Amazon dataset with test RMSEs ranging between 35.54 and 41.34. The closing price difference and closing price predictions for the Amazon dataset on the training, validation, and testing partitions for each model are

illustrated in Figure 5.5 and 5.6. A closer look at the predictions on the testing partition is illustrated in Figure 5.7 and 5.8.

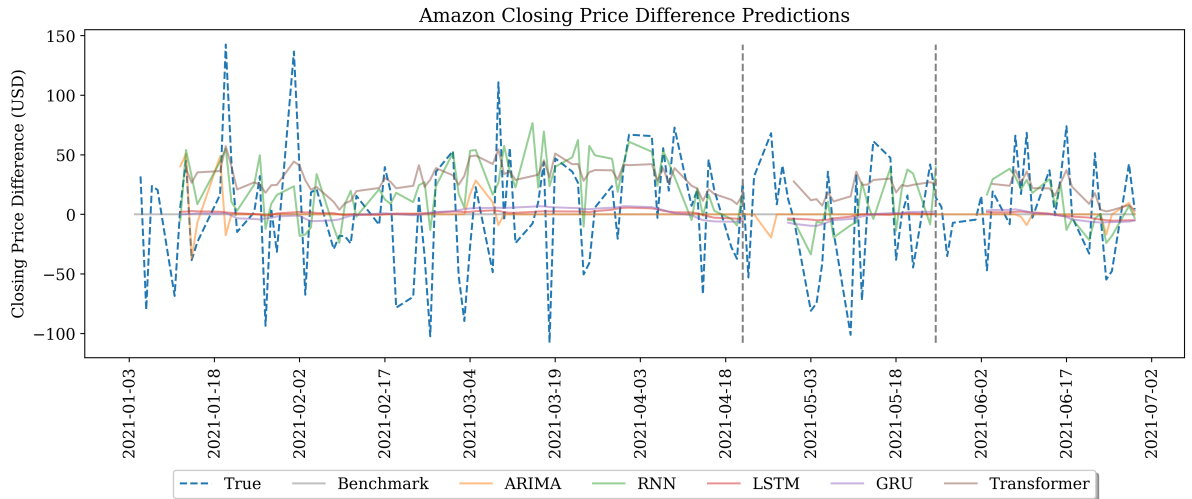


Fig. 5.5 The closing price difference predictions for the Amazon dataset

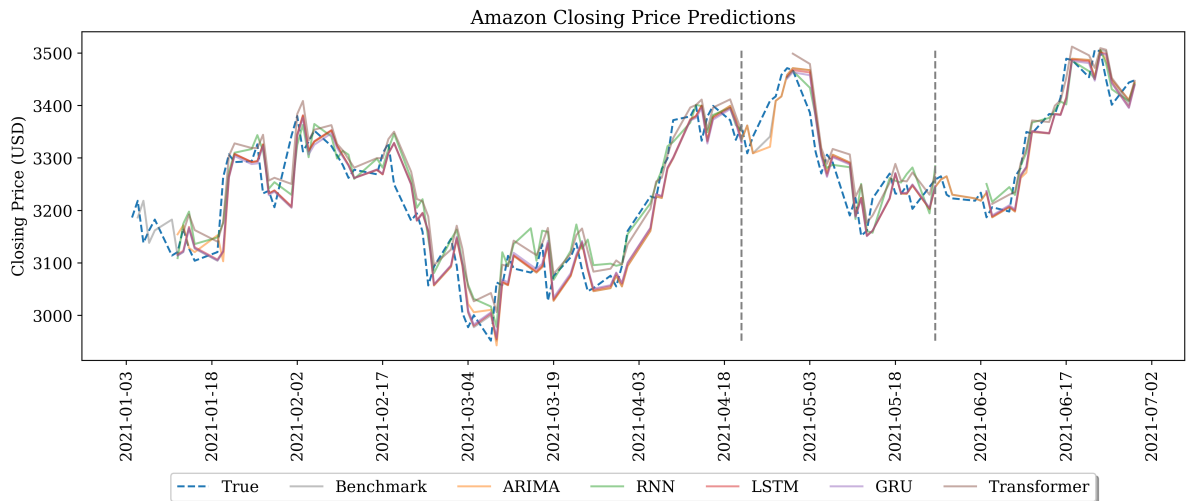


Fig. 5.6 The closing price predictions for the Amazon dataset

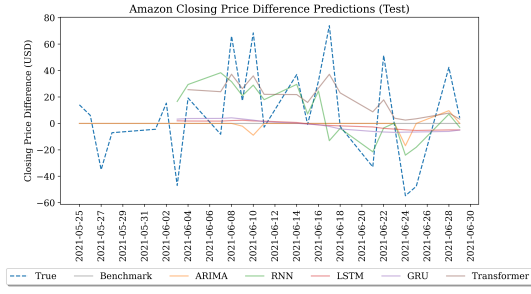


Fig. 5.7 The closing price difference predictions of the testing partition for the Amazon dataset

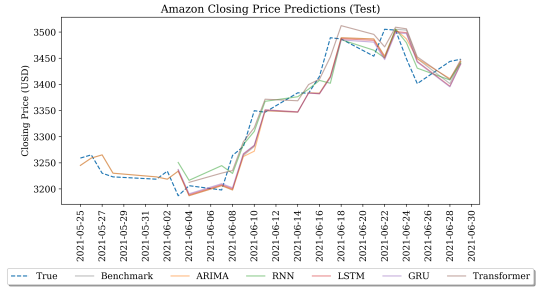


Fig. 5.8 The closing price predictions of the testing partition for the Amazon dataset

Figure 5.5 and 5.7 illustrates that both the RNN and Transformer models attempt to predict the change in closing price with large deviations from zero. Both the LSTM and GRU models are conservative with small deviations from zero. For the Apple dataset, being conservative was beneficial, however this is not the case for the Amazon dataset. Both the RNN and Transformer models outperform the LSTM and GRU models with smaller test RMSEs (provided in Table 5.4). Again, the Transformer model not only predicts significant changes in stock price with large deviation from zero but succeeds (more than any other model) with the smallest test RMSE of 35.54 (provided in Table 5.4).

A similar lagged visual effect (as discussed for Figure 5.2) is present in Figure 5.6.

The difference in overall performance between the Apple and Amazon datasets could be explained by the difference in volatility. As provided in Table 3.4, the standard deviation of the closing price difference for the Apple and Amazon datasets are 2.2 and 47.59 respectively. It could be concluded that the additional volatility of the Amazon dataset makes it harder for the models to accurately predict the closing price difference of the next day.

5.2 Sentiment Analysis

The second component of this study compares the predictive performance (if any) of different levels of sentiment analysis. Each feature subset that performed best (determined with the grid search) for all model-dataset combinations is presented in Table 5.5 (this corresponds to the entries in Table 5.4).

Table 5.5 The feature subsets selected after performing a grid search

<i>Selected Feature Subset</i>	Dataset				
Model	Apple	Microsoft	Amazon	Alphabet	Facebook
RNN	Only Technical	General Public Sentiment	Only Technical	General Public Sentiment	Stock Market Sentiment
LSTM	General Public Sentiment	All Data	Company Sentiment	Company Sentiment	All Data
GRU	Company Sentiment	All Data	Stock Market Sentiment	All Data	All Data
Transformer	General Public Sentiment	Only Technical	Only Technical	Stock Market Sentiment	Stock Market Sentiment

A summary of the number of times each feature subset has been selected in Table 5.5 is presented in Table 5.6.

Table 5.6 The number of times each feature subset was selected after performing a grid search

Feature Subset	Count
Only Technical	4
General Public Sentiment	4
Stock Market Sentiment	4
Company Sentiment	3
All Data	5
Total	20

Table 5.6 suggests that the sentiment data is indeed predictive. The Only Technical feature subset (that excludes all sentiment data) has only been selected in 4 of the top 20 models (20% of the time). The General Public Sentiment, Stock Market Sentiment, Company Sentiment and All Data feature subsets (all containing sentiment data) have been selected in the remaining 16 of the top 20 models (80% of the time). Note that All Data has been selected as the top feature subset most often. This suggests that providing the models with all available technical and sentiment data provides the best predictive performance. Also note that Company Sentiment has been selected as the top feature subsets least often. This could be attributed to the difficulty of identifying tweets related to a specific company. The sentiment data in this feature subset consists of tweets containing the company cashtag, name, or stock symbol. The text “Amazon”, “Apple”, or “Alphabet” are common English words and could have included unrelated tweets in the sentiment calculation and subsequently created noise in the data.

Next, the best and worst model-dataset combination predictions for each feature subset is inspected to better understand the predictive contributions of each feature subset.

The Transformer-Apple model-dataset combination performed best with a test RMSE of 1.22 (provided in Table 5.4). The best test RMSEs for each feature subset of the Transformer-Apple model-dataset combination (as determined with the grid search) is provided in Table 5.7. The best and worst performing feature subsets are highlighted in green and red respectively.

Table 5.7 The best test RMSEs for each feature subset of the Transformer-Apple model-dataset combination

Feature Subset	Test RMSE
Only Technical	1.26
Company Sentiment	1.28
Stock Market Sentiment	1.26
General Public Sentiment	1.22
All Data	1.30

The Transformer model closing price difference and closing price predictions for the Apple dataset on the training, validation, and testing partitions for each feature subset is illustrated in Figure 5.9 and 5.10. A closer look at the predictions on the testing partition is illustrated in Figure 5.11 and 5.12.

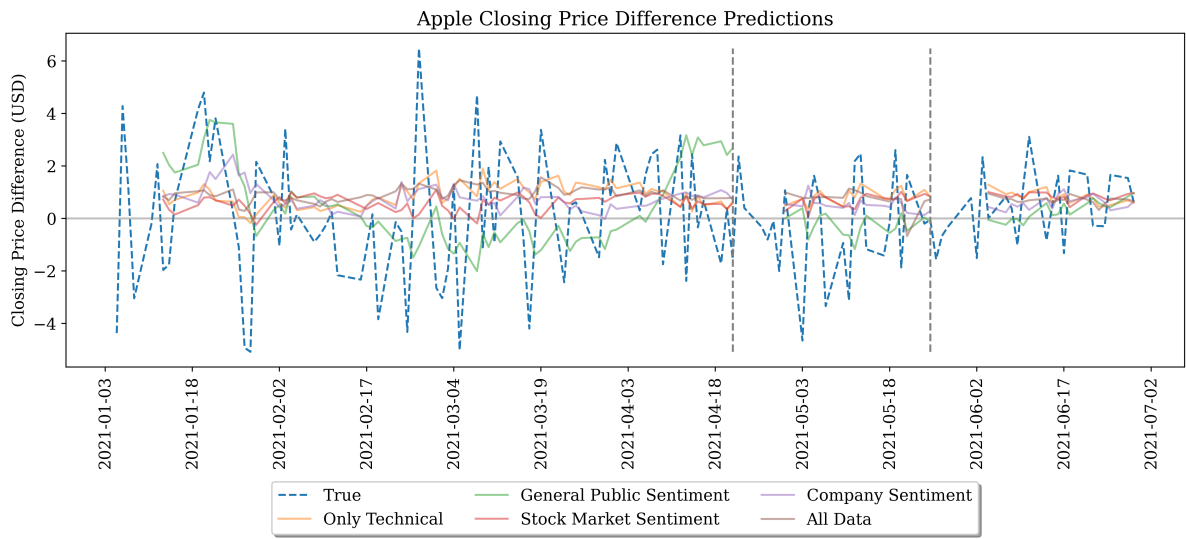


Fig. 5.9 The Transformer model closing price difference predictions for the Apple dataset

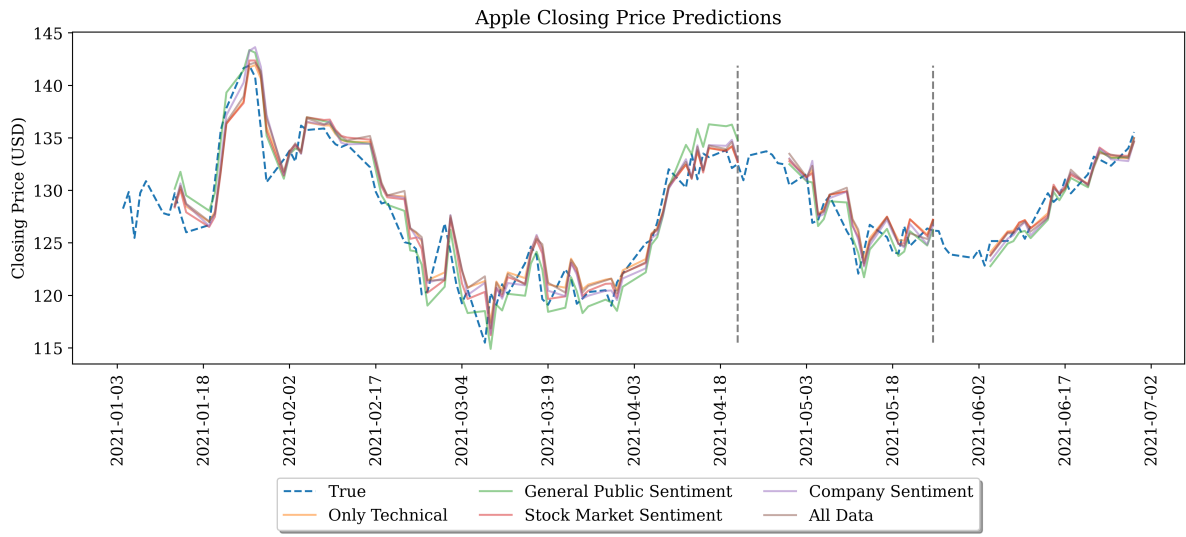


Fig. 5.10 The Transformer model closing price predictions for the Apple dataset

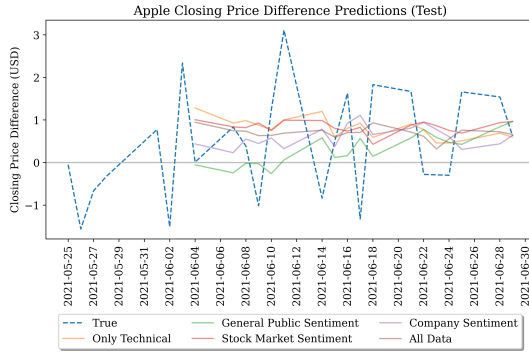


Fig. 5.11 The Transformer model closing price difference predictions of the testing partition for the Apple dataset

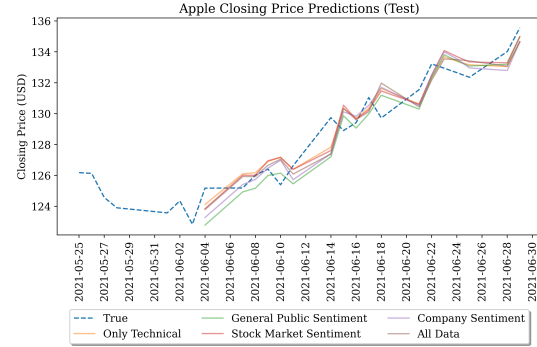


Fig. 5.12 The Transformer model closing price predictions of the testing partition for the Apple dataset

The test RMSEs in Table 5.7 barely deviate with a minimum and maximum of 1.22 and 1.30 respectively. These results almost appear counter-intuitive. The General Public Sentiment feature subset performed best with a test RMSE of 1.22. The sentiment data in this feature subset consists of tweets that do not contain any company names or stock symbols listed on the NASDAQ100 – and therefore does not contain any stock related sentiment data. The All Data feature subset performed worst with a test RMSE of 1.30. One would expect that providing the model with all available sentiment data would provide the model with more information and subsequently increase predictive performance. However, it seems that the combination of sentiment data (the addition of the sentiment data in the Company Sentiment, Stock Market Sentiment, or both feature subsets) have created noise – subsequently reducing the predictive power of the sentiment data. The illustrations provided in Figure 5.9 and 5.11 suggest that the General Public Sentiment feature subset produces predictions with significant and different (compared to those produced by other feature subsets) deviations from zero. Given these observations, it is difficult to conclude how the sentiment data has contributed towards the predictive power in this model-dataset combination.

The GRU-Amazon model-dataset combination performed worst with a test RMSE of 40.35 (provided in Table 5.4). The best test RMSEs for each feature subset of the GRU-Amazon model-dataset combination (as determined with the grid search) is provided in Table 5.8. The best and worst performing feature subsets are highlighted in green and red respectively.

Table 5.8 The best test RMSEs for each feature subset of the GRU-Amazon model-dataset combination

Feature Subset	Test RMSE
Only Technical	41.44
Company Sentiment	41.35
Stock Market Sentiment	40.35
General Public Sentiment	40.92
All Data	40.79

The GRU closing price difference and closing price predictions for the Amazon dataset on the training, validation, and testing partitions for each feature subset is illustrated in Figure 5.13 and 5.14. A closer look at the predictions on the testing partition is illustrated in Figure 5.15 and 5.16.

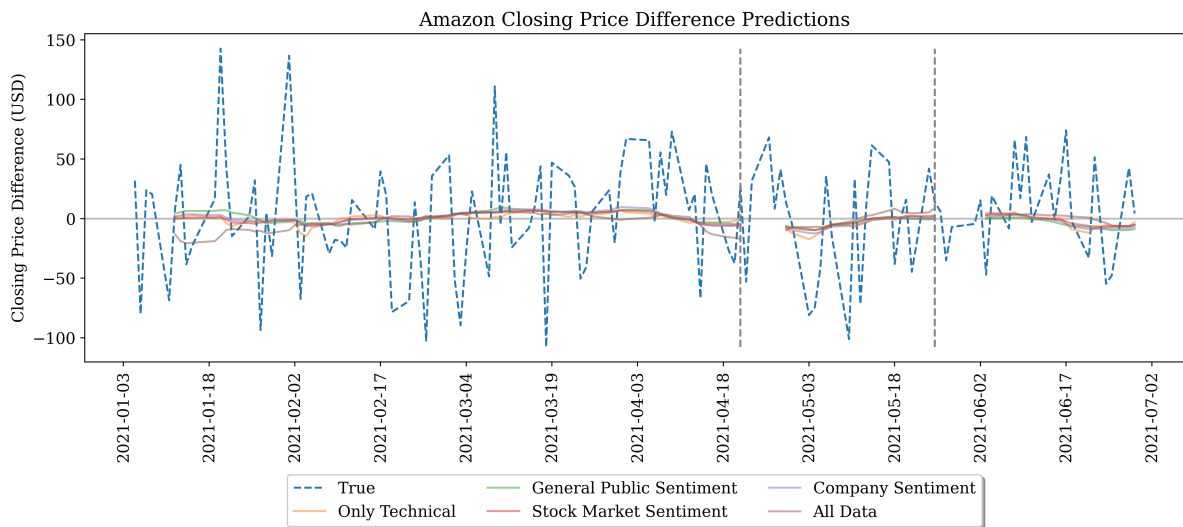


Fig. 5.13 The GRU model closing price difference predictions for the Amazon dataset

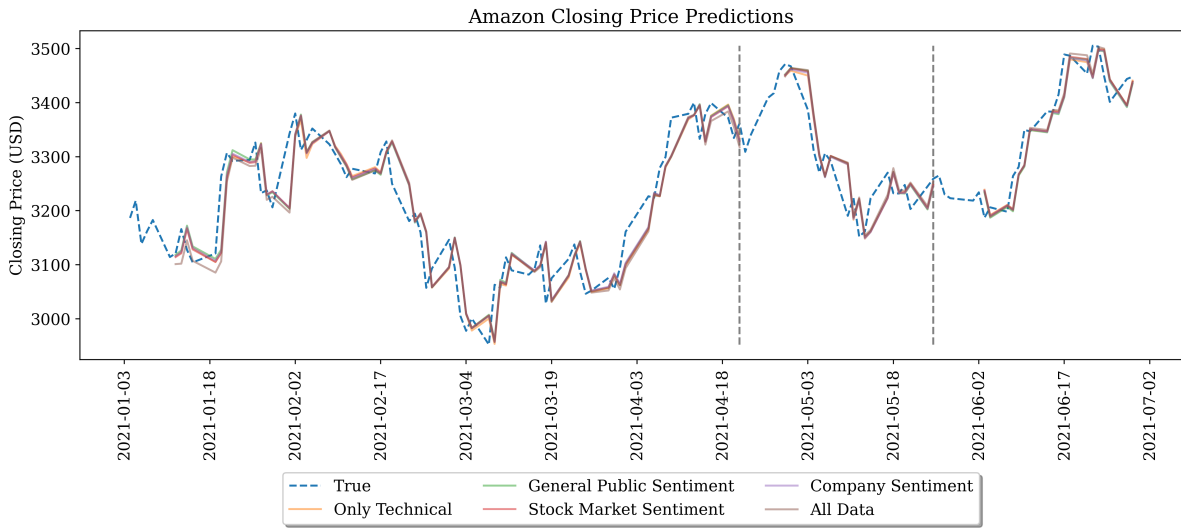


Fig. 5.14 The GRU model closing price predictions for the Amazon dataset

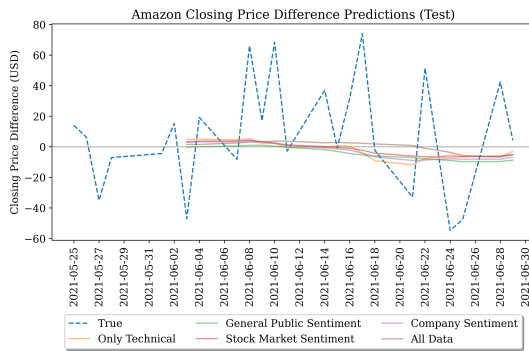


Fig. 5.15 The GRU model closing price difference predictions of the testing partition for the Amazon dataset

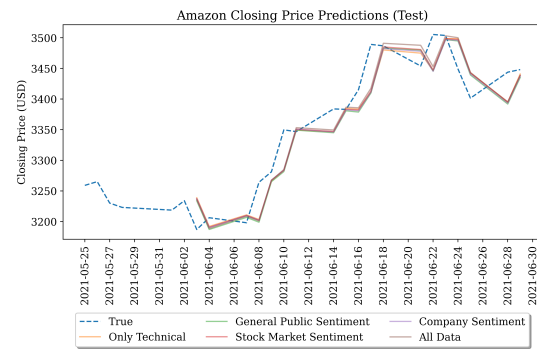


Fig. 5.16 The GRU model closing price predictions of the testing partition for the Amazon dataset

Again, the test RMSEs in Table 5.8 barely deviate with a minimum and maximum of 40.35 and 41.44 respectively. These results seem more intuitive compared to those provided in Table 5.7. The Technical Data feature subset performed worst with a test RMSE of 41.44. The Technical Data feature subset does not contain any sentiment data and therefore one would expect the model to perform worse – given that the model has access to less information to produce predictions. The Stock Market Sentiment feature subset performed best with a test RMSE of 40.35. The sentiment data in this feature subset consists of tweets containing any company name or stock symbol listed on the NASDAQ100. The illustrations provided in Figure 5.15 and 5.16 suggest that the predictions produced for the General Public Sentiment, Stock Market Sentiment, and

Company Sentiment feature subsets are almost identical with no significant differences. This could suggest that the General Public Sentiment, Stock Market Sentiment, and Company Sentiment feature subsets contain very similar sentiment data.

In conclusion, the General Public Sentiment, Stock Market Sentiment, Company Sentiment and All Data feature subsets (all containing sentiment data) have been selected as the best feature subsets in 16 of the top 20 models (80% of the time). This does provide some evidence that the sentiment data could be predictive. However, given that the Only Technical feature subset does perform best in 4 of the top 20 models (20% of the time), it is difficult to conclude that the sentiment data in this study conclusively provides predictive performance. Additionally, given the small deviations in the test RMSEs in both Table 5.7 and 5.8, it is difficult to conclude that there are significant differences in predictive performance between different feature subsets. Additional research is required to confidently conclude that sentiment data – especially a specific level of sentiment data – is predictive for stock price prediction.

5.3 Model Computational Training Efficiency

The third and final component of this study compares the training efficiency of each model by comparison of training time on the same computational resources. Three factors, namely

1. Number of parameters to be trained (training parameters)
2. Number of epochs to train for (training epochs)
3. Training time

are investigated to determine which training type (recurrent or attention-based) is more computationally efficient.

The RNN, LSTM, and GRU recurrent models are built using the TensorFlow machine learning framework (Abadi et al., 2016) based on the Keras library (Chollet et al., 2015). The Transformer model is built using the PyTorch machine learning framework (Paszke et al., 2019) based on the Torch library (Collobert et al., 2011). For the purposes of this comparison, it is assumed that the code written in these libraries are equally optimal.

As an initial comparison, the amount of time taken (in hours) for each model grid search is provided in Table 5.9.

Table 5.9 A summary of the time taken to perform each model grid search

Model	Grid Size	Grid Search Time (Hours)	Average Time per Hyperparameter Permutation (Seconds)
RNN	6750	27.83	14.84
LSTM	6750	40.83	21.77
GRU	6750	42.69	22.76
Transformer	27 000	30.99	4.13

Note the significant reduction in average time taken per hyperparameter permutation for the Transformer model (4.13 seconds) when compared to the RNN (14.84 seconds), LSTM (21.77 seconds), and GRU (22.76 seconds) recurrent models. This suggests that training the attention-based Transformer model is much more computationally efficient. However, this initial comparison does not account for the number of training parameters or the number of training epochs completed per hyperparameter permutation in the grid search. If each hyperparameter permutation in the grid search trained for the same number of epochs, a simple comparison of the training time relative to the number of parameters trained would suffice to conclude which model has the best computational training efficiency. However, given the implementation of the Early Stopping callback, each model hyperparameter permutation in the grid search has not been trained for the same number of epochs. To account for this, the number of training parameters, training epochs, and training time (in seconds) for each model grid search is investigated.

Note that in 22 of the 47 250 models fitted (0.04%) an error occurred where 0 training epochs have been completed. These models have been excluded for the remainder of the analysis.

First, the number of training parameters for each model grid search is investigated. Descriptive statistics of the number of training parameters for each model grid search is provided in Table 5.10 and illustrated in the box and whisker graph in Figure 5.17. Note that all outliers have been removed from the box and whisker plots. Outliers are defined as observations that are below $Q_1 - 1.5 \times IQR$ or above $Q_3 + 1.5 \times IQR$ where $IQR = Q_3 - Q_1$.

Table 5.10 Descriptive statistics of the number of training parameters per model grid search

<i>Number of Training Parameters</i>	Model			
Descriptive Statistic	RNN	LSTM	GRU	Transformer
Count	6749	6732	6749	26 998
Mean	5800.31	23 070.71	17 556.19	32 983.26
Standard Deviation	6574.22	26 232.33	19 854.34	33 191.77
Minimum	385	1489	1169	2313
Q_1	1345	5281	4065	4745
Q_2	3425	13 601	10 401	30 753
Q_3	5505	21 921	16 737	61 697
Maximum	21 505	85 825	64 961	92 065

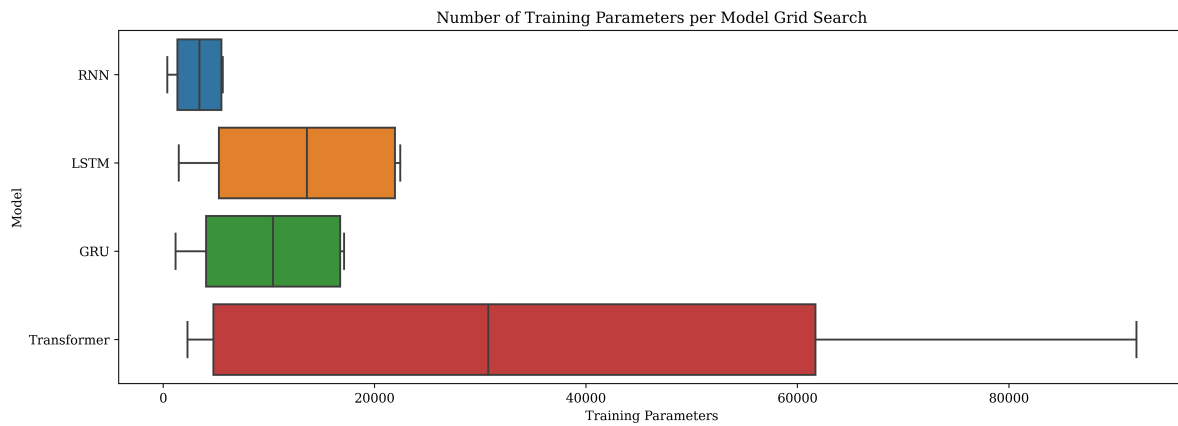


Fig. 5.17 A box and whisker plot of the number of training parameters per model grid search

Note that the number of training parameters for the recurrent model grid searches are consistent with the literature previously discussed. The RNN model has the simplest architecture of the three recurrent models, and therefore the average number of training parameters for the RNN model grid search is the least (5800) of the three recurrent models. The LSTM model attempts to solve the vanishing gradient problem the RNN model poses with additional complexity to learn long term dependencies. Therefore, the

LSTM model grid search has a larger average number of training parameters (23 070) than that of the RNN model grid search. Finally, the GRU model attempts to imitate the LSTM model with fewer parameters, and therefore the GRU model grid search has a smaller average number of training parameters (17 556) than that of the LSTM model grid search.

The number of training parameters for the Transformer model grid search has a significantly larger average (32 983) than any of the recurrent model grid searches. Additionally, the large standard deviation of the number of training parameters for the Transformer model grid search suggests that the number of training parameters significantly vary over the hyperparameter permutations explored in the grid search. This variation in training parameters can be attributed to the variation of the encoder and decoder layers explored. As illustrated in Figure 2.20, each encoder layer consists of two sub-layers:

1. Multi-head self-attention mechanism
2. Position-wise fully connected feed-forward network

For each an additional encoder layer, 2 additional sub-layers are added to the model. As illustrated in Figure 2.25, each decoder layer consists of three sub-layers:

1. Multi-head self-attention mechanism (with a look ahead mask)
2. Multi-head self-attention mechanism
3. Position-wise fully connected feed-forward network

For each additional decoder layer, 3 additional sub-layers are added to the model. Therefore, increasing the number of encoder and decoder layers significantly increase the number of training parameters.

The Transformer model grid search has a larger average number of training parameters (32 983) than the RNN (5800), LSTM (23 070), and GRU (17 556) recurrent models. Additionally, the distribution of the number of training parameters for the Transformer model grid search is more left skewed than any of the other recurrent model grid searches (as illustrated in Figure 5.17). Therefore, it can be concluded that the Transformer models in the Transformer model grid search have more training parameters than the other recurrent models considered.

Next, the number of training epochs for each model grid search is investigated. Descriptive statistics of the number of training epochs for each model grid search is provided in Table 5.11 and illustrated in the box and whisker graph in Figure 5.18.

Table 5.11 Descriptive statistics of the number of training epochs per model grid search

<i>Number of Training Epochs</i>	Model			
Descriptive Statistic	RNN	LSTM	GRU	Transformer
Count	6749	6732	6749	26 998
Mean	70.27	72.16	67.54	94.39
Standard Deviation	27.22	32.36	19.62	34.84
Minimum	50	50	50	50
Q_1	53	58	55	68
Q_2	61	66	63	88
Q_3	79	78	75	112
Maximum	784	957	701	429

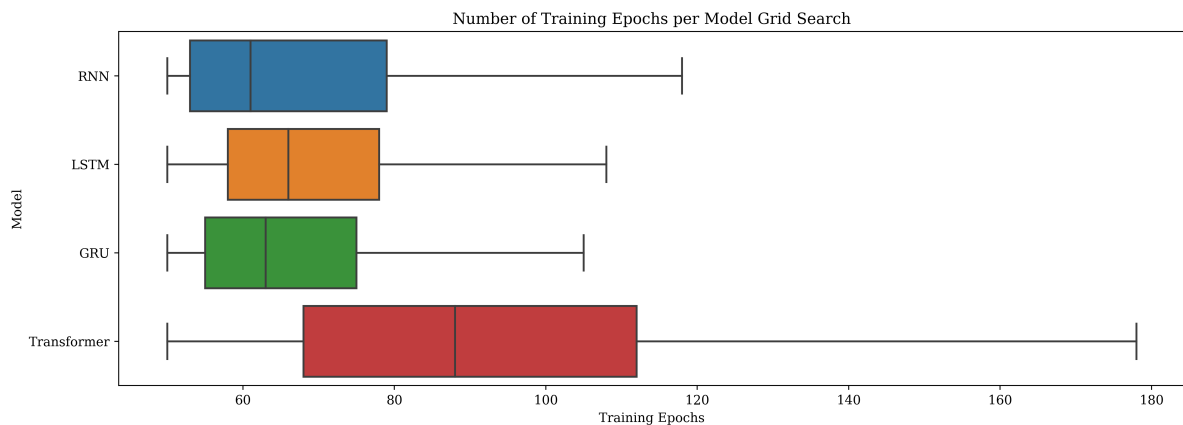


Fig. 5.18 A box and whisker plot of the number of training epochs per model grid search

The Transformer model grid search has a larger average number of training epochs (94) than the RNN (70), LSTM (72), and GRU (67) recurrent models. The distributions for the number of training epochs for both the Transformer and recurrent model grid searches are similar – right skewed with long right-hand tails. However, the distribution of the number of training epochs for the Transformer model grid search is more left skewed than any of the other recurrent model grid searches (as illustrated in Figure 5.18). Therefore, it can be concluded that the Transformer models in the Transformer model

grid search have been trained over more epochs than any of the other recurrent models considered.

As previously observed, the Transformer models in the Transformer model grid search have more training parameters. This suggests that the more training parameters a model has, the more training epochs are required to train these additional training parameters.

Finally, the training time for each model grid search is investigated. Descriptive statistics of the training time for each model grid search is provided in Table 5.12 and illustrated in the box and whisker graph in Figure 5.19.

Table 5.12 Descriptive statistics of the training time per model grid search

<i>Training Time (Seconds)</i>	Model			
Descriptive Statistic	RNN	LSTM	GRU	Transformer
Count	6749	6732	6749	26 998
Mean	14.82	21.44	22.76	4.13
Standard Deviation	11.07	44.20	51.08	1.43
Minimum	3.76	4.78	4.58	1.89
Q_1	7.98	11.31	11.17	3.11
Q_2	11.49	16.765	16.45	3.9
Q_3	18.19	24.55	26.67	4.93
Maximum	287.98	2182.52	2085.56	15.99

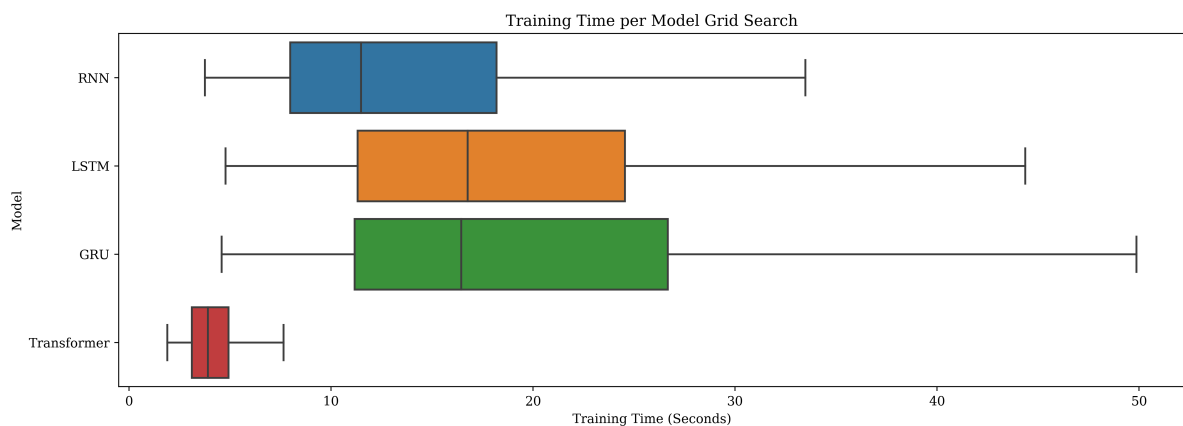


Fig. 5.19 A box and whisker plot of the training time per model grid search

From the previous two observations (the Transformer models have more parameters that are trained over more epochs) one would expect that the average training time for the Transformer model grid search would be larger than those of the recurrent models. However, this is not the case. The Transformer model grid search has a smaller average training time (4.13 seconds) than the RNN (14.82 seconds), LSTM (21.44 seconds), and GRU (22.76 seconds) recurrent models. Additionally, the distribution of the training time for the Transformer model grid search is significantly more right skewed than any of the other recurrent model grid searches (as illustrated in Figure 5.19). Therefore, it can be concluded that the Transformer models in the Transformer model grid search have trained faster than any of the other recurrent models considered.

In conclusion, three observations have been made. First, the Transformer model grid search has a larger average number of training parameters (32 983) than the RNN (5800), LSTM (23 070), and GRU (17 556) recurrent models (and a distribution that is more left skewed). Next, the Transformer model grid search has a larger average number of training epochs (94) than the RNN (70), LSTM (72), and GRU (67) recurrent models (and a distribution that is more left skewed). Finally, the Transformer model grid search has a smaller average training time (4.13 seconds) than the RNN (14.82 seconds), LSTM (21.44 seconds), and GRU (22.76 seconds) recurrent models (with a distribution that is more right skewed). That is, the models in the Transformer grid search have *more* training parameters that have been trained over *more* epochs, but still trained *faster* than any of the other recurrent models considered. Therefore, it can be concluded that the computational training efficiency of the Transformer model in this study is better than that of the recurrent models considered. This could be attributed to the Transformer model precluding recurrence, which allows the training computation to be performed in parallel and ultimately results in more computational training efficiency.

Chapter 6

Discussion and Conclusion

6.1 Discussion

This research aimed to accurately predict the future stock price of the top 5 stocks (by market capitalization) of the NASDAQ100 (as of January 2021). To realise this objective, this research aimed to use alternative data to better inform models of current events and also look towards recent advancements in machine learning as inspiration to build a time series regression model utilising the Transformer architecture.

The first component of this research aimed to use Twitter data as an alternative data source for information of current events and to translate this information into a representation that informs changes in stock price. The predictive performance of different representations – or *levels* – of the sentiment data was assessed, while accounting for the computational requirements of obtaining, storing, and processing large-scale Twitter data. 830 209 231 tweets (311.3GB) posted from January 2021 to June 2021 (6 months) was extracted from an online archive for this study. The archive was incomplete in that for some days no tweets have been archived. The data mining technique *web scraping* was employed to visit the main Twitter feed every 2 minutes and scrape as many tweets as possible. The scrape operation collected approximately 7500 tweets per hour and, in total, the scrape operation ran for 426 hours (17 days) to collect 3 268 705 scraped tweets (1.55GB). After filtering for only English tweets, the dataset used for the sentiment analysis consisted of 299 988 331 tweets (28.54GB).

Before performing the sentiment analysis, the text of each tweet was “cleaned” to remove any text that does not contribute towards sentiment. Next, each tweet was separated

into sentences and each sentence was scored using the state-of-the-art FLAIR sentiment analysis model. The model outputs a single unidimensional measure of sentiment for a given sentence on a scale from -1 (most extreme negative) to +1 (most extreme positive). The average of the sentiment scores for each sentence in a tweet was the final sentiment score assigned to that tweet. Predicting the sentiment of tweets at this scale was very computationally demanding. The FLAIR sentiment analysis model builds directly on PyTorch that has been optimised to use CUDA – allowing the model computations to run in parallel directly on the GPU rather than the CPU. This significantly increased the speed at which the model can produce predictions. The GPU available for this study had the computational capacity to predict the sentiment of 1 000 000 tweets (using the FLAIR sentiment analysis model) in 4 hours. Predicting the sentiment of all 299 988 331 tweets would have required 1200 hours (50 days) of computation time. Therefore, the number of tweets for sentiment prediction was required to be filtered down to allow for a more reasonable computation time. For each day, the following tweets were included by order of importance for the sentiment analysis of this study: First, all tweets containing a cashtag, NASDAQ100 ticker symbol, or NASDAQ100 company name were selected. From the remaining tweets, a random sample of 500 000 tweets were selected. This reduced the total tweets considered to 78 649 968 tweets and subsequently the required computation time to 314 hours (13 days).

From the sentiment scored tweets data, three levels of sentiment was extracted: company, stock market, and general public sentiment. Tweets containing the name or ticker symbol of the company considered was used to determine the company level of sentiment. Tweets containing the name or ticker symbol of any NASDAQ100 listed company was used to determine the stock market level of sentiment. The remaining tweets were used to determine the general public level of sentiment. For a given day, the sentiment scores of these tweets were averaged to determine the final sentiment score for that day. This produced 7 features – sentiment for the general public, stock market, and one for each of the 5 companies considered – to translate the information from the Twitter data into a representation consumable for a time series regression model.

Using the Augmented Dickey-Fuller statistical hypothesis test, it was concluded (at a 0.10 level of significance) that the closing price of every dataset (at lag 5) is non-stationary. This was remedied by a single differencing operation, and it was concluded (at a 0.10 level of significance) that the closing price *difference* of every dataset (at lag 5) is stationary. That is, the models predicted the *change* in closing price rather than the *absolute* closing price of the following day.

Using the Granger causality statistical hypothesis test, it was concluded (at a 0.10 level of significance) that all sentiment features – except for the Amazon sentiment feature – is Granger-causal for their respective company closing price difference. Additionally, it was concluded (at a 0.10 level of significance) that the stock market sentiment is Granger-causal for the Amazon and Facebook closing price difference.

The second component of this research aimed to build a time series regression model by utilising the Transformer architecture currently powering the recent advancements in machine learning. Given the additional complexity and subsequent increase in training parameters, the model computational training efficiency was assessed – using Recurrent Neural Networks as a benchmark to compare both predictive performance and model computational training efficiency. During model development, a thorough hyperparameter tuning grid search was performed to determine the optimal hyperparameters for each respective model. Additionally, the optimal feature subsets (containing different levels of sentiment data) were selected and investigated to compare the predictive performance (if any) of the different representations of the Twitter data.

A thorough hyperparameter tuning grid search was performed to select the optimal hyperparameters for each machine learning model built. For the Transformer model, a grid of 27 000 hyperparameter permutations was searched and fitted over 30.99 hours. For the recurrent models, a grid of 20 250 hyperparameter permutations was searched and fitted over 27.83, 40.83, and 42.68 hours for the RNN, LSTM, and GRU recurrent models respectively. Large standard deviations and ranges between the minimum and maximum RMSE on the testing partition for each model fitted in the grid search suggests that the effort of performing the grid search was worthwhile and that arbitrarily selecting hyperparameters for each model would result in sub-optimal performance.

All the machine learning models outperformed the ARIMA model, and the attention-based Transformer model outperformed the recurrent models (in every dataset considered). This suggests that, of all the models considered, machine learning models – especially the Transformer model – is better able to capture the complex patterns within the stock data and subsequently better predict future stock price. Closer inspection of the best performing Apple dataset has shown that the Transformer model attempted to predict the change in closing price with large deviations from zero (predicting zero suggests no change in future stock price). The LSTM was conservative and consistently predicted almost no change in stock price with few deviations from zero. This could suggest that the LSTM model considers the change in stock price random, and therefore predicting that no change will occur is the optimal solution to minimise the error. The RNN and

GRU models attempted to predict the change in closing price – albeit not very confidently – with “medium” sized deviations from zero. The Transformer model not only predicted significant changes in stock price with large deviation from zero but succeeded (more than any other model) with the smallest test RMSE of 1.22.

The predictive performance of the different representations of the Twitter data was assessed using 5 different feature subsets. Each feature subset that performed best (as determined with the grid search) for all model-dataset combinations have been captured. This resulted in 20 models – each of the 4 models fitted to each of the 5 datasets.

The Only Technical feature subset (all technical data and no sentiment data) was only selected as the optimal feature subset in 4 of the top 20 models (20% of the time). The General Public Sentiment, Stock Market Sentiment, Company Sentiment and All Data feature subsets (all containing sentiment data) was selected in the remaining 16 of the top 20 models (80% of the time).

The All Data feature subset (all technical data and all sentiment data) was selected as the top feature subset most often. This suggests that providing the models with all available technical and sentiment data provides the best predictive performance. The Company Sentiment feature subset (all technical data and company level sentiment data) was selected as the top feature subset least often. This could be attributed to the difficulty of identifying tweets related to a specific company. The sentiment data in this feature subset consisted of tweets containing the company cashtag, name, or stock symbol. The text “Amazon”, “Apple”, or “Alphabet” are common English words and could have included unrelated tweets in the sentiment calculation and subsequently created noise in the data.

Closer inspection of the best performing Transformer-Apple model-dataset combination has shown counter-intuitive results that question the validity of the predictive performance of the sentiment data. The test RMSEs between the different feature subsets for this model-dataset combination barely deviated with a minimum and maximum of 1.22 and 1.30 respectively. The General Public Sentiment feature subset performed best with a test RMSE of 1.22. The sentiment data in this feature subset consisted of tweets that do not contain any company names or stock symbols listed on the NASDAQ100, and therefore does not contain any stock related sentiment data. The All Data feature subset performed worst with a test RMSE of 1.30. Given these observations in the best performing model-dataset combination, it is difficult to conclude how the sentiment data has contributed towards the predictive power. Additional research is required to confidently conclude that this sentiment data – especially specific levels of sentiment data – is predictive for stock price prediction.

The training efficiency of each model was assessed by comparison of training time on the same computational resources.

The average time taken per hyperparameter permutation for the Transformer model grid search (4.13 seconds) was significantly smaller than that of the RNN (14.84 seconds), LSTM (21.77 seconds), and GRU (22.76 seconds) recurrent model grid searches. This suggests that training the Transformer model is much more computationally efficient. However, this initial comparison does not account for the number of training parameters, or the number of training epochs completed per hyperparameter permutation in the grid search. An Early Stopping callback was implemented during model development to stop training once no reduction in validation cost has been detected in the previous 50 epochs. Therefore, three factors of the model grid searches have been investigated.

Firstly, the Transformer model grid search had a larger average number of training parameters (32 983) than the RNN (5800), LSTM (23 070), and GRU (17 556) recurrent models (and a distribution that is more left skewed). Secondly, the Transformer model grid search had a larger average number of training epochs (94) than the RNN (70), LSTM (72), and GRU (67) recurrent models (and a distribution that is more left skewed). Finally, the Transformer model grid search has a smaller average training time (4.13 seconds) than the RNN (14.82 seconds), LSTM (21.44 seconds), and GRU (22.76 seconds) recurrent models (with a distribution that is more right skewed).

That is, the models in the Transformer model grid search had *more* training parameters that have been trained over *more* epochs, but still trained *faster* than any of the other models in the recurrent model grid searches. Therefore, it can be concluded that the computational training efficiency of the Transformer model is better than that of the recurrent models built in this study. This could be attributed to the Transformer model precluding recurrence, which allows the training computation to be performed in parallel and ultimately results in more computational training efficiency.

6.2 Conclusion

This research aimed to accurately predict the future closing price of the top 5 stocks of the NASDAQ100 (an index consisting of the 100 largest non-financial companies listed on the NASDAQ stock market) by leveraging alternative data and the recent advancements in machine learning. The alternative data was produced via the sentiment analysis of large-scale Twitter data. The recent advancements in machine learning inspired the development of a time series regression model utilising the Transformer architecture.

The Twitter data represented as company sentiment (filtering for tweets containing the name or ticker symbol of the company considered) and stock market sentiment (filtering for tweets containing the name or ticker symbol of any NASDAQ100 company) is Granger-causal ($p < 0.10$) for the closing price of 4 and 2 of the 5 companies considered respectively.

This study built 5 stock price prediction models: a traditional statistical model (ARIMA), three recurrent machine learning models (RNN, LSTM, and GRU), and an attention-based Transformer machine learning model. The performance of each model was assessed using the RMSE on a testing partition of each dataset considered. All the machine learning models outperformed the ARIMA model, and the attention-based Transformer model outperformed the recurrent models (in every dataset considered). This suggests that, of all the models considered, machine learning models – especially the Transformer model – is better able to capture the complex patterns within the stock data and subsequently better predict future stock price.

The computational efficiency of the machine learning models was assessed by comparison of training time on the same computational resources. During a hyperparameter tuning grid search (to select the optimal hyperparameters) the number of training parameters, training epochs, and training time for each model fitted was captured and compared. On average, the models fitted in the Transformer model grid search had *more* training parameters that have been trained over *more* epochs, but still trained *faster* than any of the other models in the recurrent model grid searches. This suggests that training the Transformer model (trained using standard Backpropagation) is more computational efficient than training the recurrent models (trained using Backpropagation Through Time).

After model development, closer inspection revealed counter-intuitive results that question the validity of the predictive performance of the sentiment data. Different feature subsets considering the three different levels of sentiment (company, stock market, and general public) were considered to select the optimal feature subset. Predictions produced using the different feature subsets were almost identical. Additionally, the feature subset considering company level sentiment was selected as optimal *least* often. These two observations suggest that additional research is required to confidently conclude that this sentiment data – especially specific levels of sentiment data – is predictive for stock price prediction.

Chapter 7

Future Recommendations

Sentiment Analysis

When predicting the sentiment of large-scale Twitter data, it is difficult to validate how accurate the FLAIR sentiment analysis model is. To remedy this, one could manually label a subset of the tweets as either positive, neutral, or negative, and measure the accuracy of the model using this labelled subset. However, determining sentiment of any given text is subjective to the reader. What one reader considers positive, another reader could consider neutral, or negative, or vice versa. To account for this, one could use multiple sentiment analysis models that use different methodologies of determining sentiment. To account for the strengths and weaknesses of each model, one could use many different models to predict the sentiment for a given tweet and use the average sentiment scores to label the final sentiment for that tweet. This is analogous to using multiple judges to determine a final verdict and could contribute towards a more accurate sentiment prediction.

Another component of sentiment analysis that can be explored is the use of more dimensions of human emotion – rather than a simple positive or negative sentiment. Determining the sentiment of tweet at a more granular level could contribute more important information and subsequently more predictive power. Note that validating the accuracy of multiple dimensions of mood could prove difficult, as more subjectivity is introduced into the system.

As discovered in this study, determining the sentiment of tweets at a large-scale is very computationally demanding. This study had to sample down the number of tweets considered for the sentiment analysis to make the computational time feasible. Assuming

that Twitter data is predictive for stock price prediction, one would logically assume that retrieving and predicting the sentiment for more tweets would contribute more predictive power. The models in this study have been limited to be trained and tested on 6 months of data. Ideally, one would have more (multiple years) of data to train and test these models. Future work could explore using distributed computing technologies or cloud computing to enable this.

The scrape operation used in this study collected approximately 7500 tweets per hour. At this rate, only 180 000 tweets would be captured in a 24 hour period. Obtaining a large sample is very time consuming and, in a live real-time environment, only 180 000 tweets would be considered for the sentiment used to predict the closing price of the next day. Future work would need to remedy the rate at which tweets are captured and their sentiment predicted to ensure that enough sentiment data is available at the time of the next closing price prediction.

It is unlikely that each tweet contributes the same amount towards the change in the stock price. Some users are more influential than others. Future work could explore methods of weighting sentiment of certain tweets to place more emphasis on the sentiment of influential users.

In this study, company sentiment was derived from tweets containing the name or ticker symbol of the company considered. To further extract relevant sentiment information, future research could explore including tweets containing stock terms related to company acquisitions. For example, Instagram and WhatsApp are two companies which have both been acquired by Facebook Inc. (recently renamed to Meta Platforms Inc.). Sentiment towards acquired companies could affect the sentiment towards their respective parent company, and subsequently inform changes in the stock price of the parent company.

Hyperparameter Tuning

In this study, a thorough hyperparameter grid search was performed (over a substantial set of hyperparameter permutations) to determine the optimal hyperparameters for each model. It is impossible to know if the optimal hyperparameters have been selected to fit each model such that the cost function has converged to a *global* minimum. Future research could explore the use of more computational resources to increase the size of the hyperparameter grid search and possibly find more optimal hyperparameters.

Modelling

In this study, both the recurrent models historic reference window and the Transformer model encoder input window have been selected as the same size at 5 time steps (days). This prevents the Transformer model from utilising one of its largest strengths – the ability to learn long term dependencies. The size of the stock datasets has been limited to 6 months due to the data and computational availability for the sentiment analysis. If future work could remedy the rate at which tweets are captured and their sentiment predicted, larger historic reference windows for both the recurrent and Transformer models could be explored.

This study was limited to a future stock price prediction use case. Future research could explore a simulation study. Here, multiple (thousands) of time series datasets can be generated. The recurrent and Transformer models can then be fitted and tested on these datasets to help motivate the conclusion on whether the time series regression model utilising the Transformer architecture developed in this study outperforms recurrent models.

References

- Abadi, M., Barham, P., Chen, J., Chen, Z., Davis, A., Dean, J., Devin, M., Ghemawat, S., Irving, G., Isard, M., and Others (2016). Tensorflow: A system for large-scale machine learning. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 265–283.
- Akbik, A., Bergmann, T., Blythe, D., Rasul, K., Schweter, S., and Vollgraf, R. (2019). FLAIR: An Easy-to-Use Framework for State-of-the-Art NLP. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics (Demonstrations)*, pages 54–59, Minneapolis, Minnesota. Association for Computational Linguistics.
- Akbik, A., Blythe, D., and Vollgraf, R. (2018). Contextual String Embeddings for Sequence Labeling. In *Proceedings of the 27th International Conference on Computational Linguistics*, pages 1638–1649, Santa Fe, New Mexico, USA. Association for Computational Linguistics.
- Apple Newsroom (2021). Imac, iPad Pro, and Apple TV 4K in stores Friday. <https://www.apple.com/za/newsroom>. Accessed: 2022-01-20.
- Ba, J. L., Kiros, J. R., and Hinton, G. E. (2016). Layer Normalization. University of Toronto and Google Inc.
- Bai, Y. (2022). Relu-function and derived function review. *SHS Web Conf.*, 144:02006.
- Balduzzi, D., Frean, M., Leary, L., Lewis, J. P., Ma, K. W.-D., and McWilliams, B. (2018). The Shattered Gradients Problem: If resnets are the answer, then what is the question?
- Blum, A. and Rivest, R. L. (1988). Training a 3-node neural network is NP-complete. *Proceedings of the 1st Annual Workshop on Computational Learning Theory, COLT 1988*, pages 9–18.
- Bollen, J., Mao, H., and Zeng, X. (2011). Twitter mood predicts the stock market. *Journal of Computational Science*, 2(1):1–8.
- Box, G. and Jenkins, G. (1970). *Time Series Analysis: Forecasting and Control*. Holden-Day series in time series analysis and digital processing. Holden-Day.
- Britz, D. (2015). Recurrent Neural Networks Tutorial, Part 3 – Backpropagation Through Time and Vanishing Gradients.

- Buffet, W. E. (1984). The Superinvestors of Graham-and-Doddsville. Columbia Business School Magazine (Hermes).
- Cho, K., Van Merriënboer, B., Gulcehre, C., Bahdanau, D., Bougares, F., Schwenk, H., and Bengio, Y. (2014). Learning phrase representations using RNN encoder-decoder for statistical machine translation. In *EMNLP 2014 - 2014 Conference on Empirical Methods in Natural Language Processing, Proceedings of the Conference*, pages 1724–1734.
- Chollet, F. et al. (2015). Keras. <https://keras.io>.
- Collobert, R., Kavukcuoglu, K., and Farabet, C. (2011). Torch7: A Matlab-like Environment for Machine Learning. In *BigLearn, NIPS Workshop*.
- Dickey, D. A. and Fuller, W. A. (1979). Distribution of the estimators for autoregressive time series with a unit root. *Journal of the American Statistical Association*, 74(366a):427–431.
- Garey, M. R. and Johnson, D. S. (1990). *Computers and Intractability; A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., USA.
- Gers, F. A., Schmidhuber, J., and Cummins, F. (2000). Learning to forget: Continual prediction with LSTM. *Neural Computation*, 12(10):2451–2471.
- Glorot, X. and Bengio, Y. (2010). Understanding the difficulty of training deep feedforward neural networks. *Journal of Machine Learning Research*, 9:249–256.
- Goel, A. and Mittal, A. (2012). Stock Prediction Using Twitter Sentiment Analysis. Stanford University Project.
- Goodfellow, I. J., Bengio, Y., and Courville, A. (2016). *Deep Learning*. MIT Press, Cambridge, MA, USA. <http://www.deeplearningbook.org>.
- Granger, C. W. J. (1969). Investigating causal relations by econometric models and cross-spectral methods. *Econometrica*, 37(3):424–438.
- He, K., Zhang, X., Ren, S., and Sun, J. (2015a). Deep Residual Learning for Image Recognition. Microsoft Research.
- He, K., Zhang, X., Ren, S., and Sun, J. (2015b). Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification. Microsoft Research.
- Hinton, G. (2015). Neural Networks for Machine Learning. Coursera Online Course <https://www.coursera.org>.
- Hochreiter, S. (1991). Untersuchungen zu dynamischen neuronalen Netzen. Technical University Munich, Institute of Computer Science.
- Hochreiter, S., Bengio, Y., Frasconi, P., and Schmidhuber, J. (2001). *Gradient Flow in Recurrent Neural Nets : The Difficulty of Learning Long-Term Dependencies*.
- Hochreiter, S. and Schmidhuber, J. (1997). Long Short-Term Memory. *Neural Computation*, 9(8):1735–1780.

- Honnibal, M. and Montani, I. (2017). spaCy 2: Natural language understanding with Bloom embeddings, convolutional neural networks and incremental parsing. *Empirical Methods in Natural Language Processing*.
- Huang, T. C., Zaeem, R. N., and Barber, K. S. (2019). It is an equal failing to trust everybody and to trust nobody: Stock price prediction using trust filters and enhanced user sentiment on twitter. *ACM Transactions on Internet Technology*, 19(4).
- Hutto, C. and Gilbert, E. (2014). VADER: A Parsimonious Rule-Based Model for Sentiment Analysis of Social Media Text. *Eighth International AAAI Conference on Weblogs and Social Media*, page 18.
- Jastrzbski, S., Arpit, D., Ballas, N., Verma, V., Che, T., and Bengio, Y. (2018). Residual Connections Encourage Iterative Inference. *International Conference on Learning Representations (ICLR)*.
- Kiefer, J. and Wolfowitz, J. (1952). Stochastic Estimation of the Maximum of a Regression Function. *The Annals of Mathematical Statistics*, 23(3):462–466.
- Lipton, Z. C., Berkowitz, J., and Elkan, C. (2015). A Critical Review of Recurrent Neural Networks for Sequence Learning. Technical report.
- Loria, S. (2018). textblob Documentation. Package Documentation. Release 0.15.
- Maas, A. L., Daly, R. E., Pham, P. T., Huang, D., Ng, A. Y., and Potts, C. (2011). Learning Word Vectors for Sentiment Analysis. In *Proceedings of the 49th Annual Meeting of the Association for Computational Linguistics: Human Language Technologies*, pages 142–150, Portland, Oregon, USA. Association for Computational Linguistics.
- Majaski, C. (2022). Understanding Fundamental vs. Technical Analysis. www.investopedia.com. Accessed: 2021-01-20.
- Malkiel, B. G. (1973). *A Random Walk down Wall Street*. W. W. Norton & Company, Inc.
- Maqsood, H., Mehmood, I., Maqsood, M., Yasir, M., Afzal, S., Aadil, F., Selim, M. M., and Muhammad, K. (2020). A local and global event sentiment based efficient stock exchange forecasting using deep learning. *International Journal of Information Management*, 50:432–451.
- McCulloch, W. S. and Pitts, W. H. (1943). A Logical Calculus of the Ideas Immanent in the Nervous Activity. *Bulletin of Mathematical Biophysics*, 5:115–133.
- Musk, E. (2020). "Tesla stock price is too high imo". Twitter Post <https://twitter.com/elonmusk>.
- Nelson, D. M., Pereira, A. C., and De Oliveira, R. A. (2017). Stock market’s price movement prediction with LSTM neural networks. *Proceedings of the International Joint Conference on Neural Networks*, pages 1419–1426.
- Nesterov, Y. E. (1983). A method for solving the convex programming problem with convergence rate $O(1/k^2)$. *Dokl. Akad. Nauk SSSR*, 269:543–547.
- Nguyen, T. T. and Yoon, S. (2019). A novel approach to short-term stock price movement prediction using transfer learning. *Applied Sciences (Switzerland)*, 9(22).

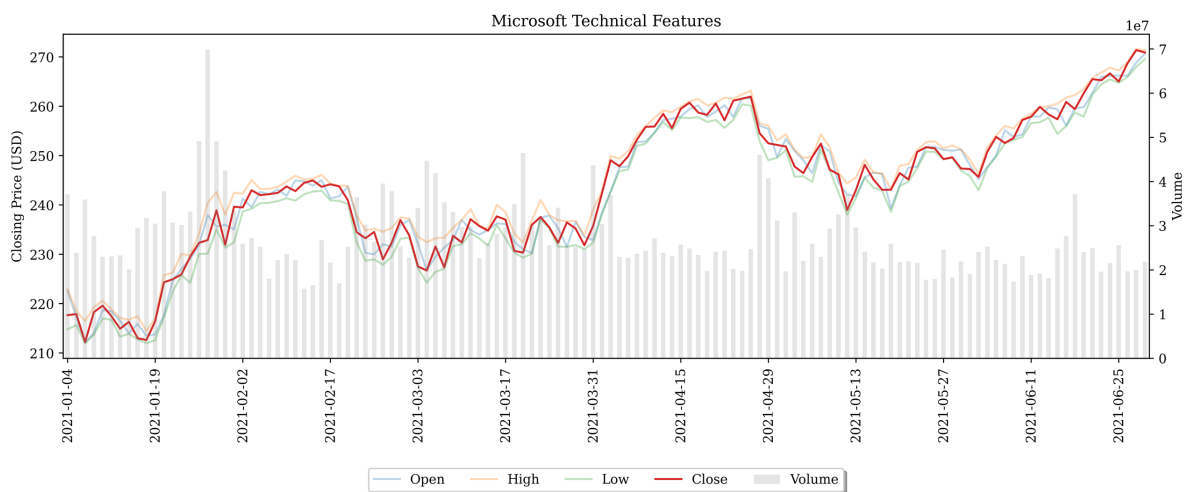
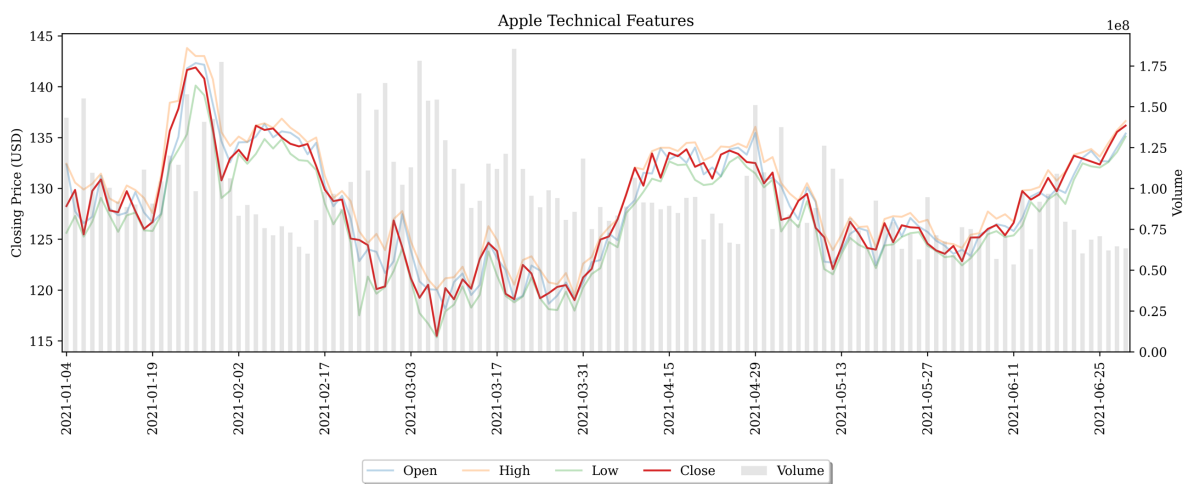
- Olah, C. (2015). Understanding LSTM Networks. Colah's Blog. Accessed: 2021-03-15.
- Pascanu, R., Mikolov, T., and Bengio, Y. (2013). On the difficulty of training recurrent neural networks. *Proceedings of the 30th International Conference on Machine Learning*, 28:1310–1318.
- Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L., Desmaison, A., Kopf, A., Yang, E., DeVito, Z., Raison, M., Tejani, A., Chilamkurthy, S., Steiner, B., Fang, L., Bai, J., and Chintala, S. (2019). PyTorch: An Imperative Style, High-Performance Deep Learning Library. In *Advances in Neural Information Processing Systems 32*, pages 8024–8035. Curran Associates, Inc.
- Phi, M. (2020). Illustrated Guide to Transformers - Step by Step Explanation. Towards Data Science Post <https://towardsdatascience.com>. Accessed: 2021-04-29.
- Qian, N. (1999). On the Momentum Term in Gradient Descent Learning Algorithms Acknowledgments. *Neural Networks*, 12(1):145–151.
- Robbins, H. and Monro, S. (1951). A Stochastic Approximation Method. *The Annals of Mathematical Statistics*, 22(3):400–407.
- Roondiwala, M., Patel, H., and Varma, S. (2017). Predicting Stock Prices Using LSTM. *International Journal of Science and Research*, 6(4):1754–1756.
- Ruder, S. (2016). An overview of gradient descent optimization algorithms. Technical report.
- Rumelhart, D. E., Hinton, G. E., and Williams, R. J. (1986). Learning Representations by Back-Propagating Errors. *Nature*, 323:533–536.
- Rumelhart, D. E., Hinton, G. E., and Williams, R. J. (1987). Learning Internal Representations by Error Propagation. In *Parallel Distributed Processing: Explorations in the Microstructure of Cognition*, pages 318 – 362.
- Selvin, S., Vinayakumar, R., Gopalakrishnan, E. A., Menon, V. K., and Soman, K. P. (2017). Stock price prediction using LSTM, RNN and CNN-sliding window model. In *2017 International Conference on Advances in Computing, Communications and Informatics (ICACCI)*, pages 1643–1647.
- Siddiqui, F. (2020). Tesla stock plummets more than 10 percent after Elon Musk tweets valuation is "too high". The Washington Post <https://www.washingtonpost.com>.
- Sola, J. and Sevilla, J. (1997). Importance of input data normalization for the application of neural networks to complex industrial problems. *IEEE Transactions on Nuclear Science*, 44(3):1464–1468.
- Song, H., Rajan, D., Thiagarajan, J. J., and Spanias, A. (2018). Attend and diagnose: Clinical time series analysis using attention models. *32nd AAAI Conference on Artificial Intelligence, AAAI 2018*, pages 4091–4098.
- Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, Ł., and Polosukhin, I. (2017). Attention is all you need. *Advances in Neural Information Processing Systems*, pages 5999–6009.

-
- Werbos, P. J. (1990). Backpropagation Through Time: What It Does and How to Do It. *Proceedings of the IEEE*, 78(10):1550–1560.
- Wu, N., Green, B., Ben, X., and O’Banion, S. (2020). Deep transformer models for time series forecasting: The influenza prevalence case. *arXiv*.
- Yahoo! (2021). Yahoo! Finance. <https://finance.yahoo.com/>. Accessed: 2021-01-20.

Appendix A

Data

A.1 Technical Data



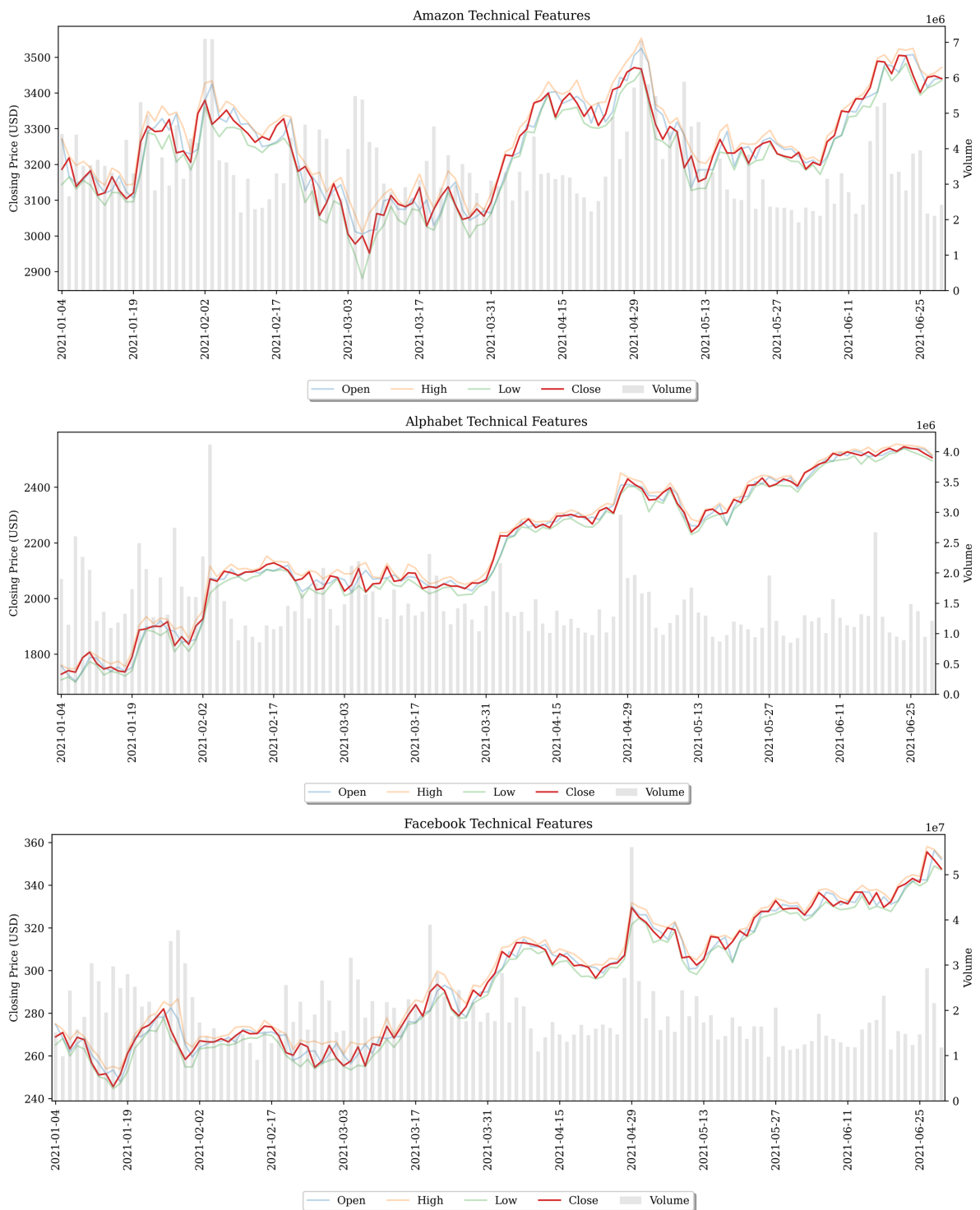
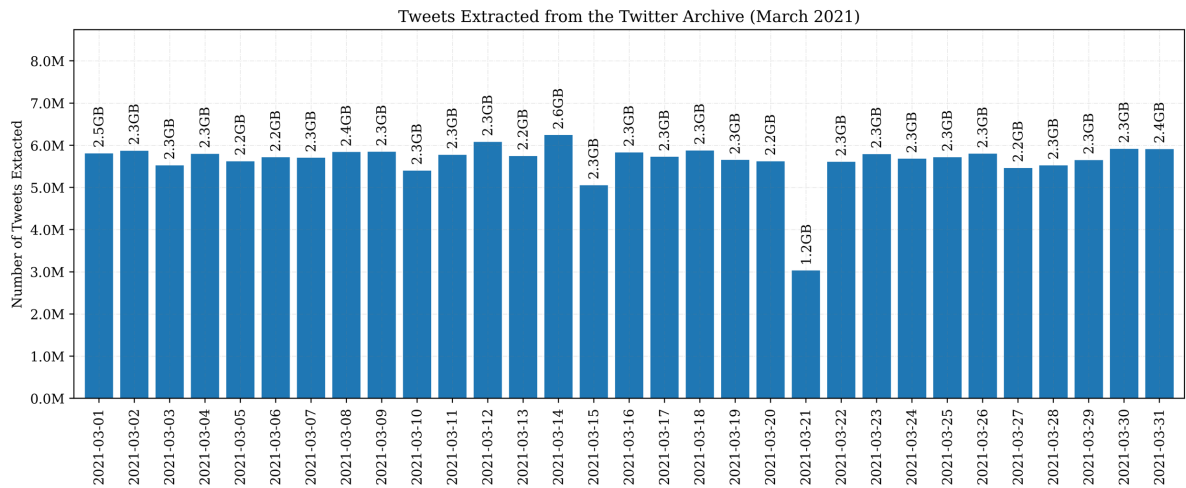
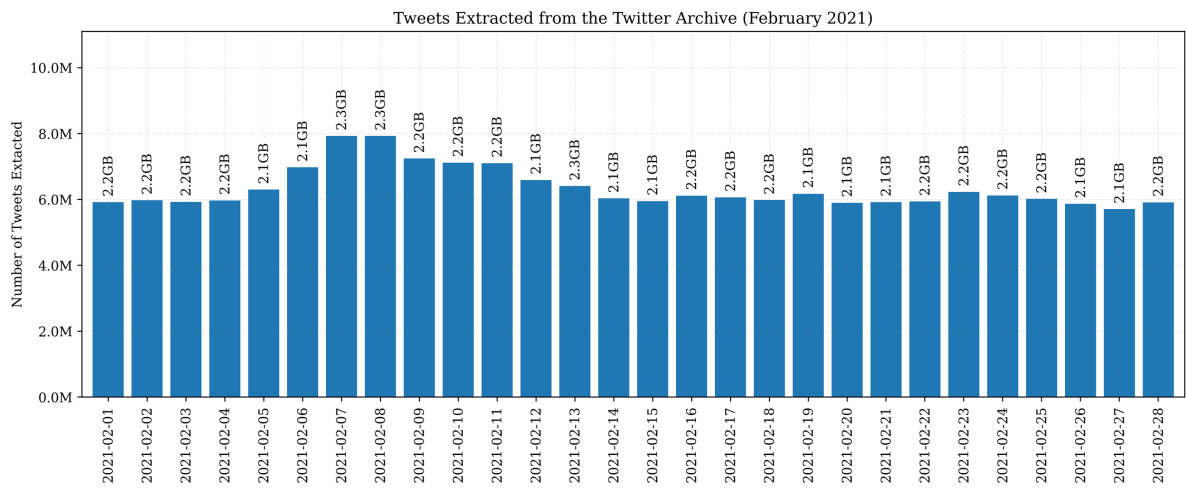
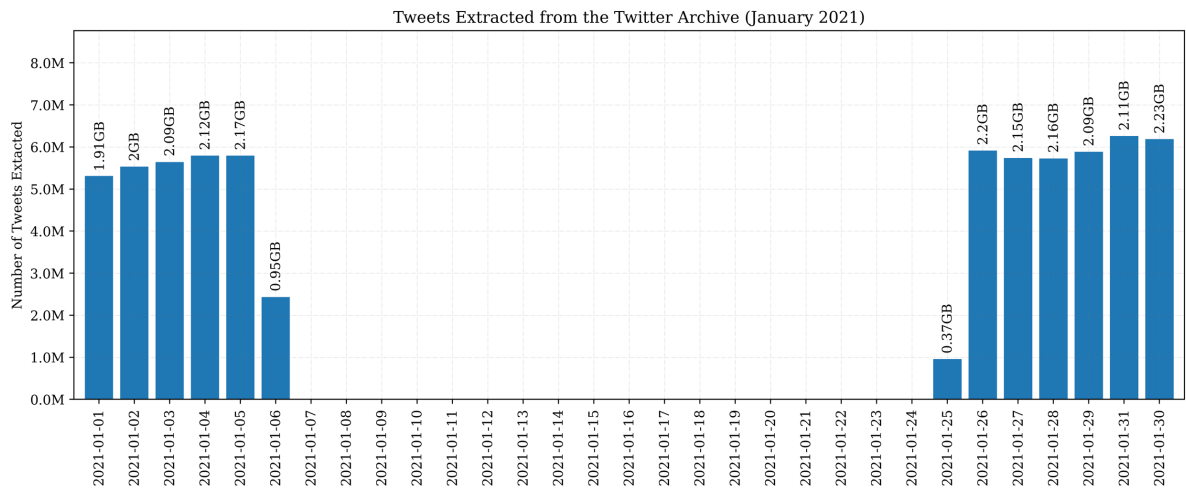


Fig. A.1 The technical features for each considered company

A.2 Sentiment Data

A.2.1 Twitter Data



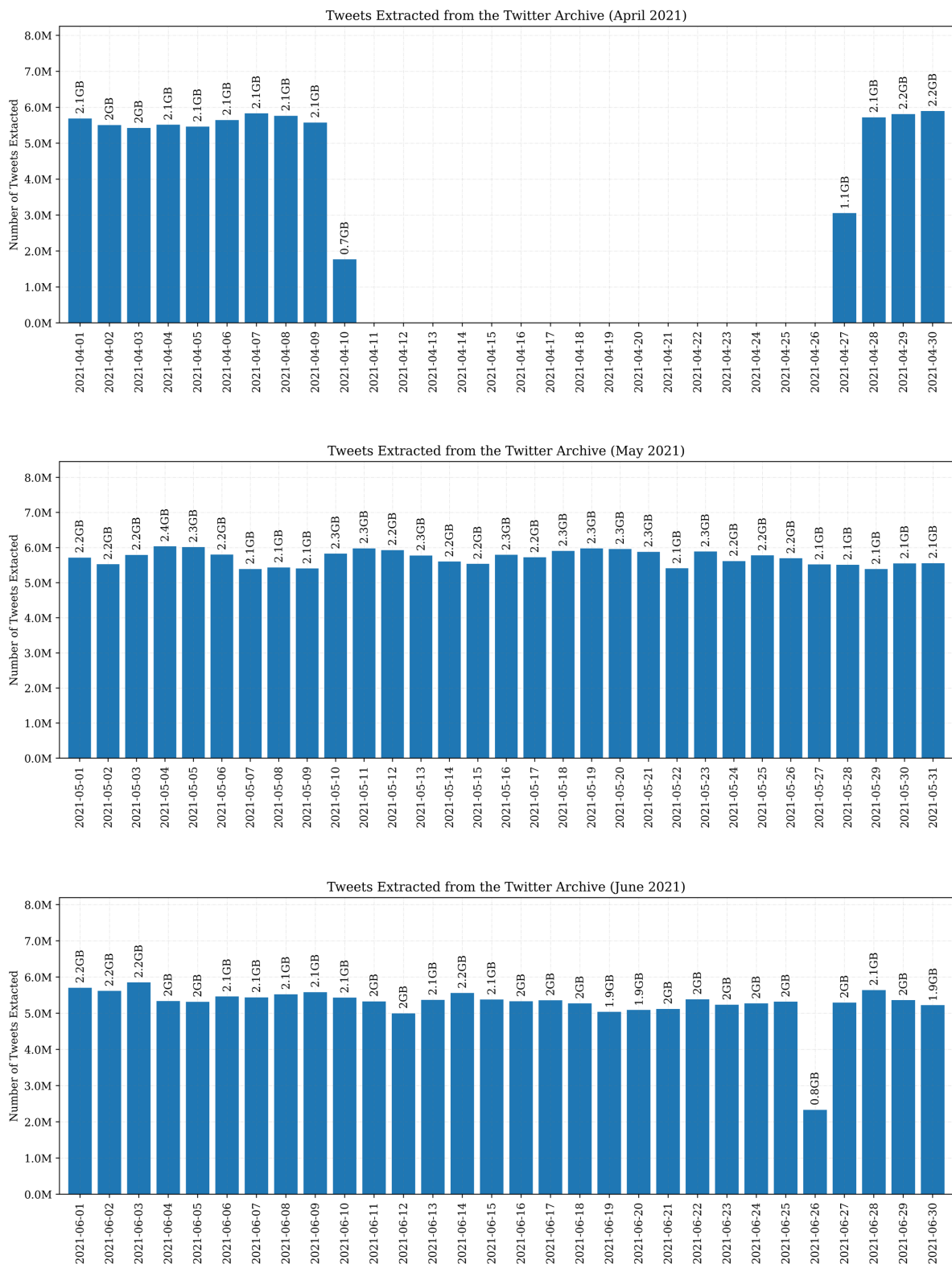


Fig. A.2 The tweets extracted from the Twitter archive

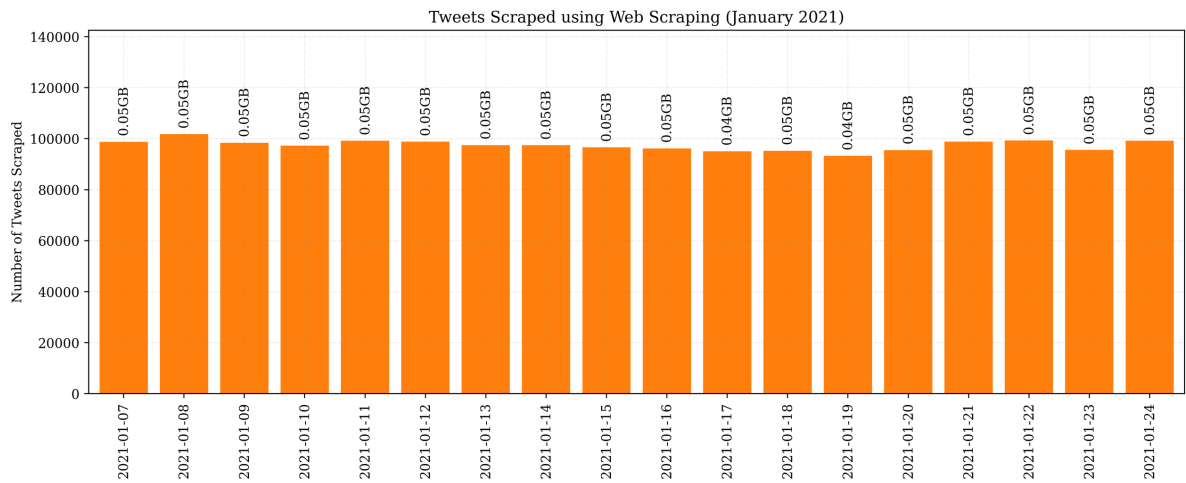


Fig. A.3 The tweets scraped using web scraping for January 2021

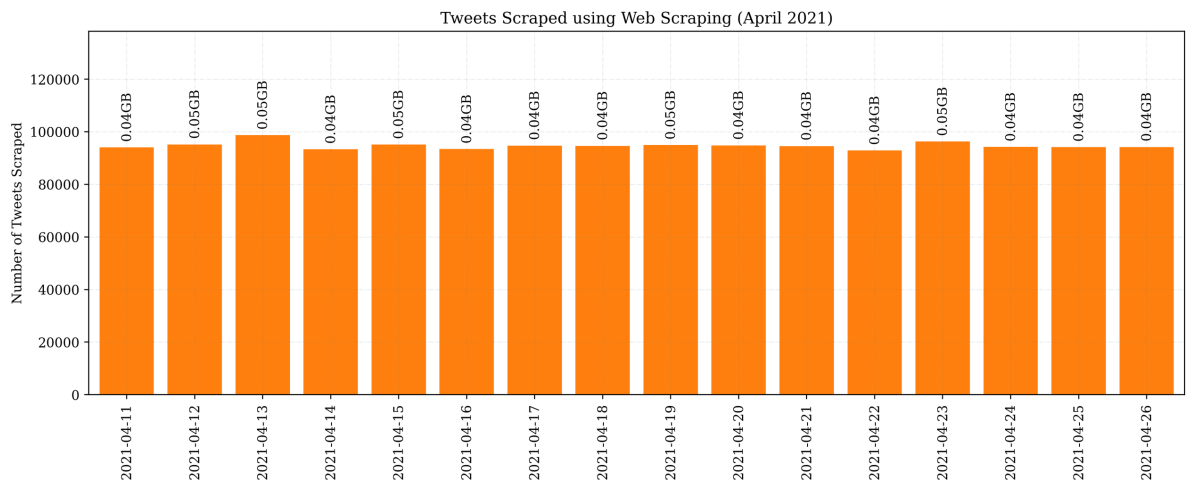
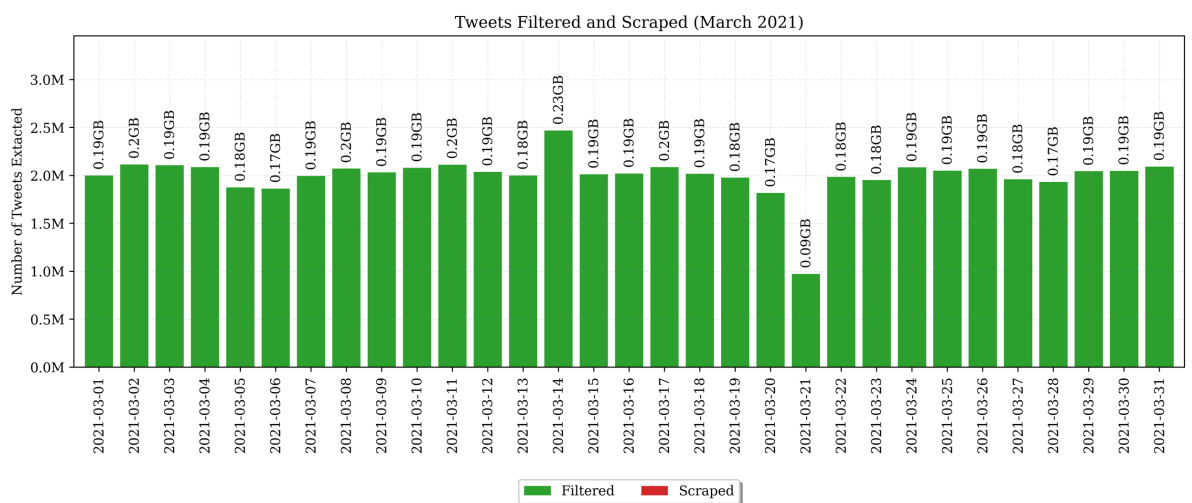
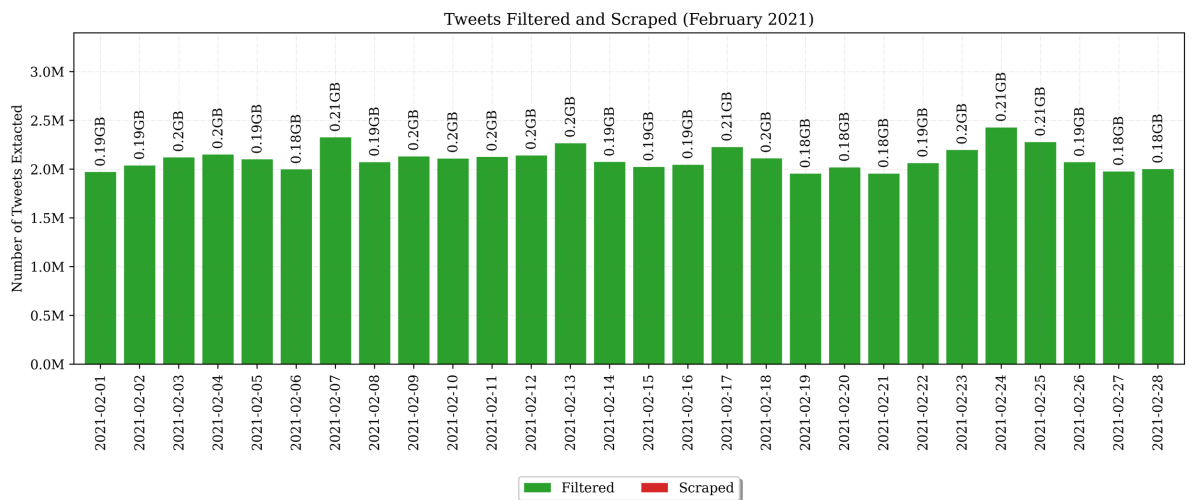
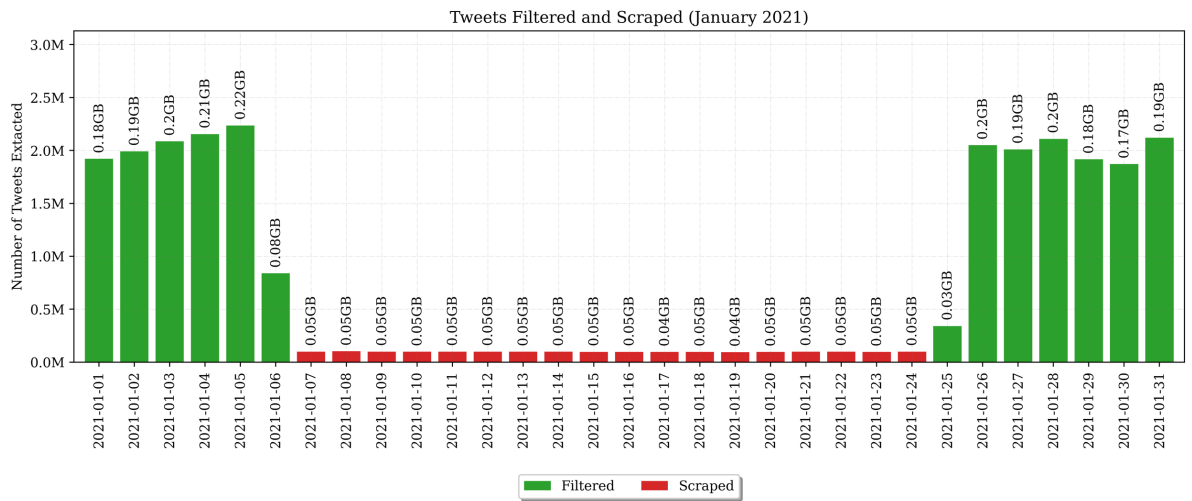


Fig. A.4 The tweets scraped using web scraping for April 2021



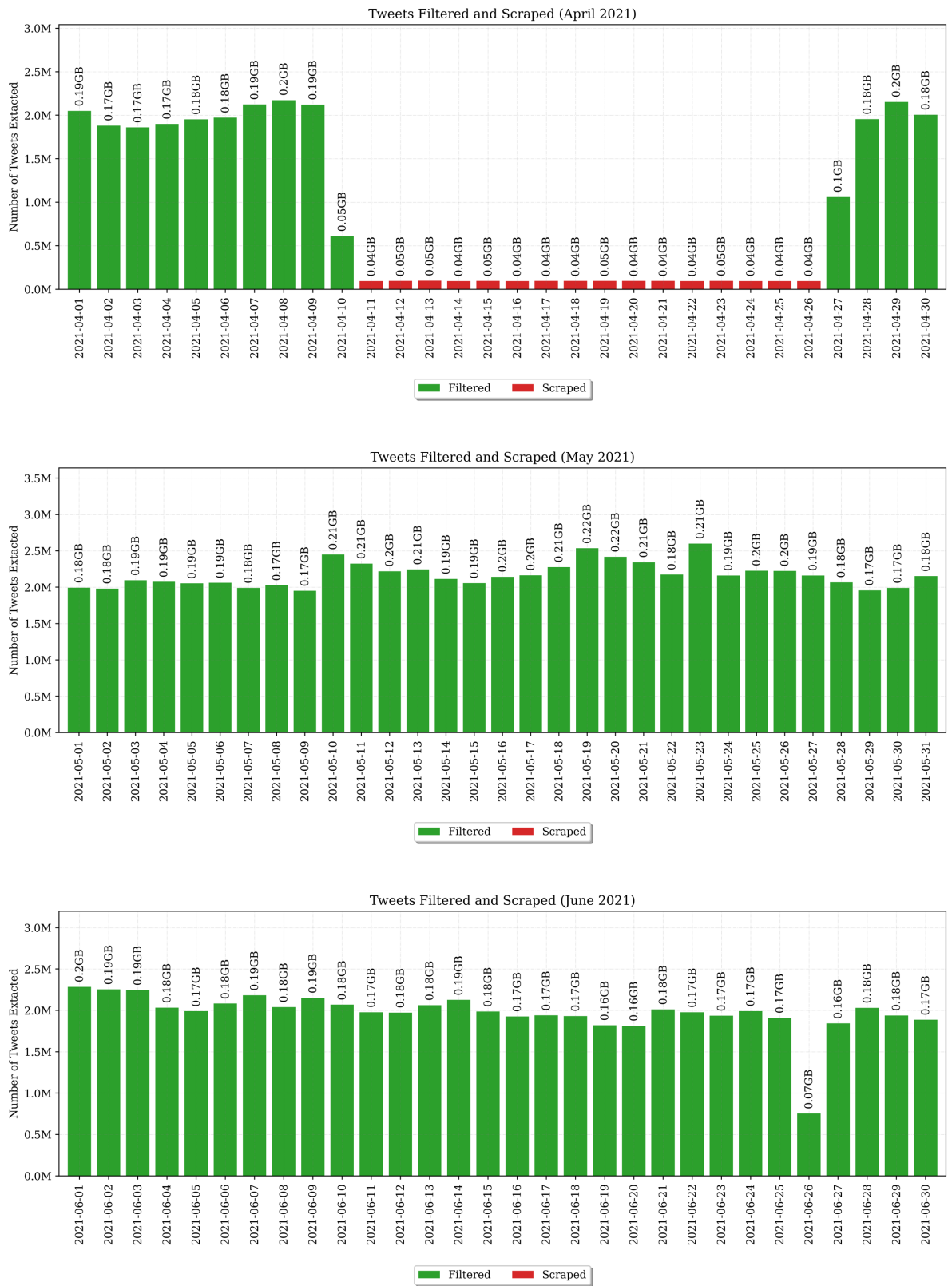
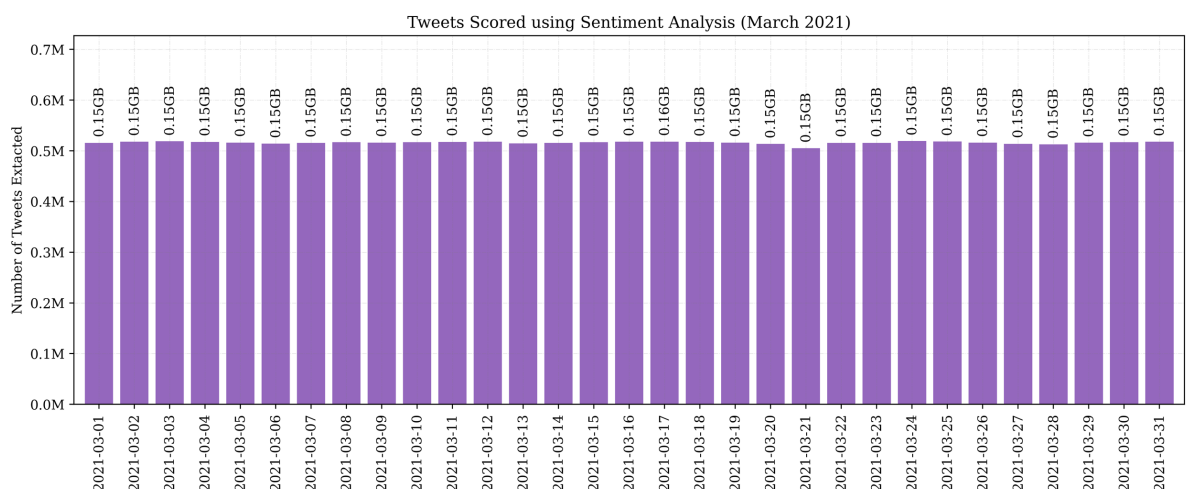
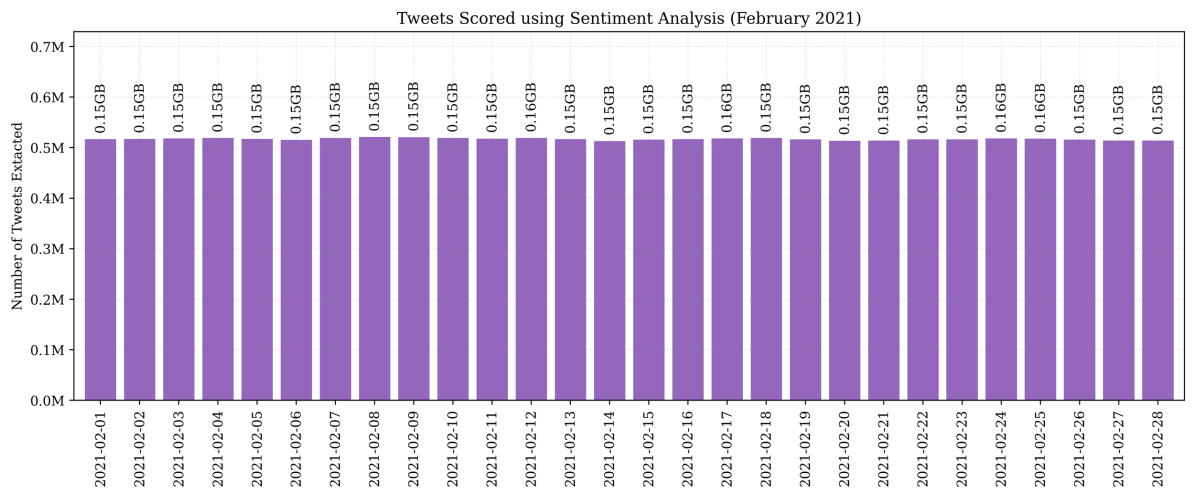
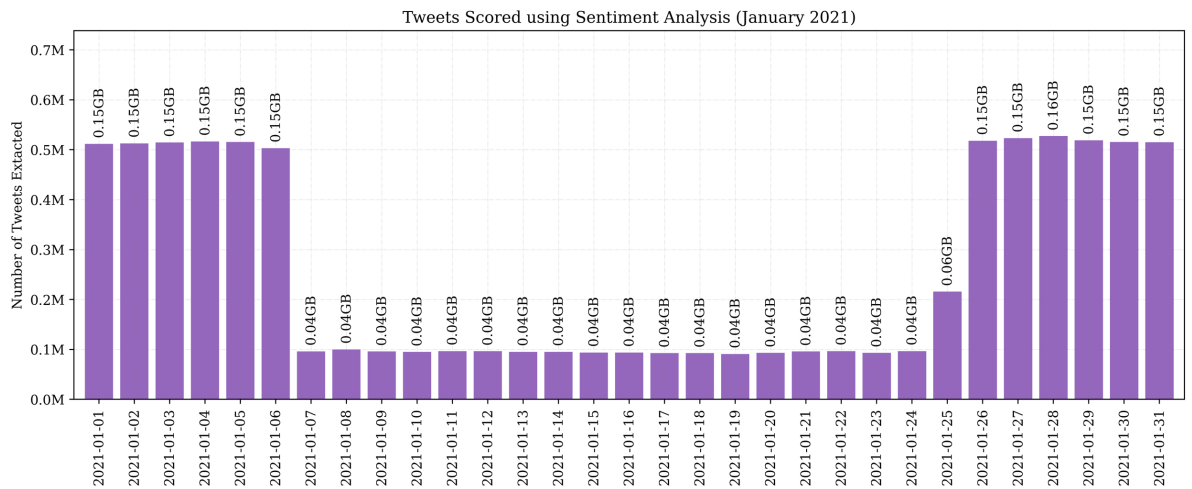


Fig. A.5 The tweets filtered and scraped using web scraping



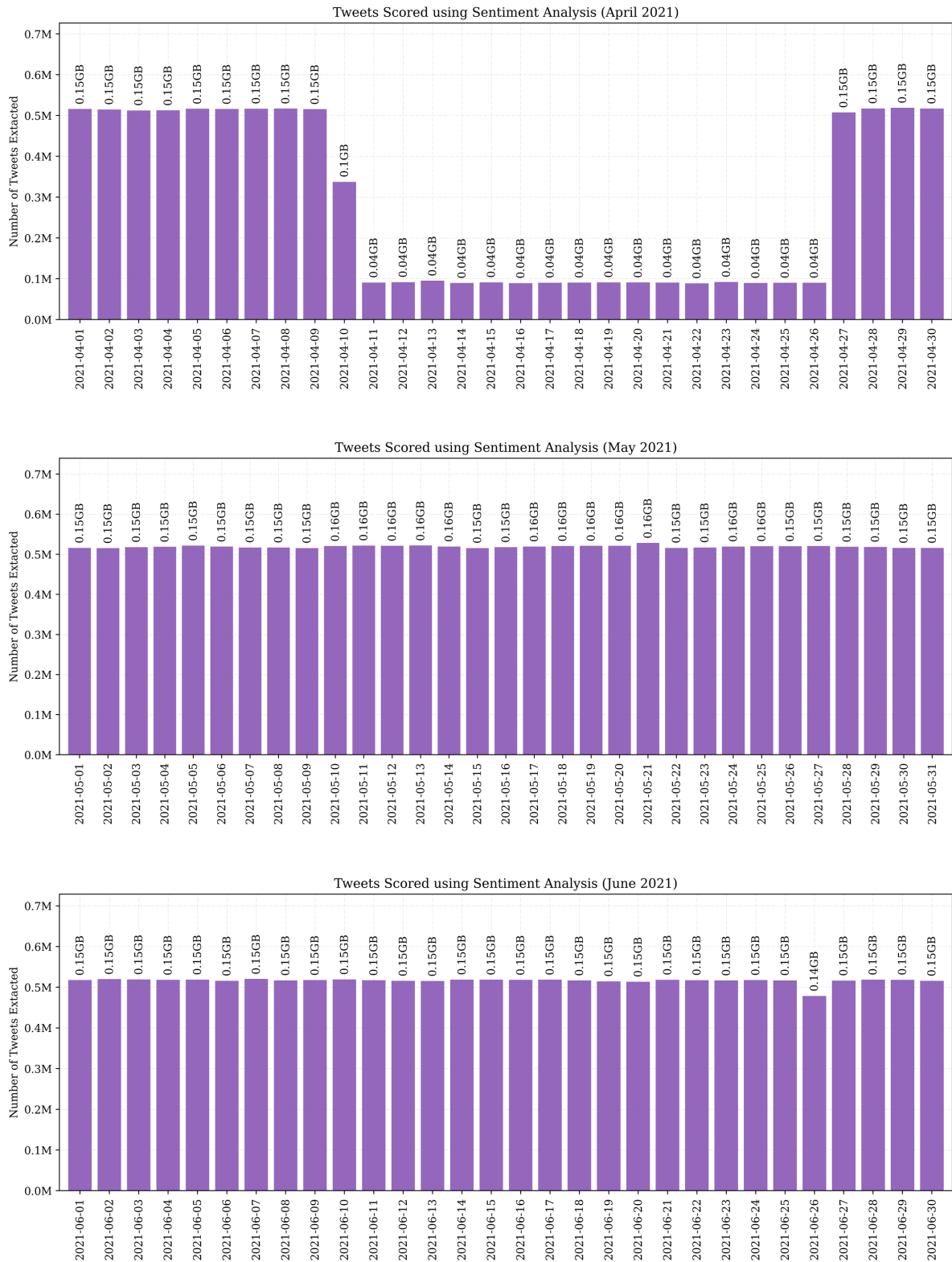


Fig. A.6 The tweets scored using sentiment analysis

A.2.2 Levels of Sentiment

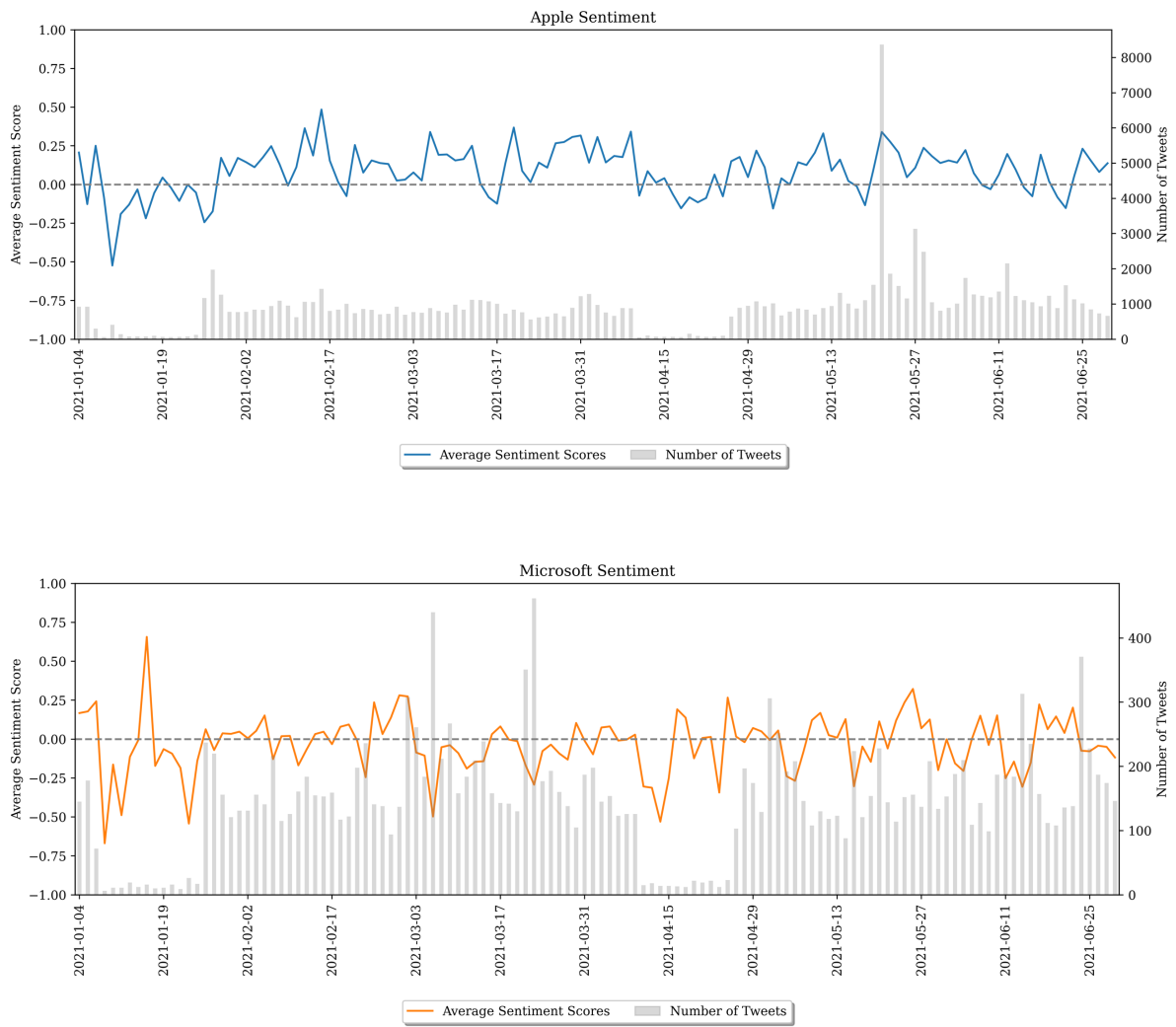




Fig. A.7 The sentiment scores for each considered company

Appendix B

Results

B.1 Model Performance

Table B.1 The optimal hyperparameters for the Recurrent Neural Network model per dataset

<i>RNN</i>	Dataset				
Hyperparameter	Apple	Microsoft	Amazon	Alphabet	Facebook
Feature Subset	Only Technical	General Public Sentiment	Only Technical	General Public Sentiment	Stock Market Sentiment
Batch Size	4	4	2	2	4
Number of Layers	3	2	2	1	1
Number of Units	32	32	32	32	64
Dropout	0.5	0	0	0	0.5
Test RMSE	1.47	2.35	37.25	16.66	5.03

Table B.2 The optimal hyperparameters for the LSTM model per dataset

<i>LSTM</i>	Dataset				
Hyperparameter	Apple	Microsoft	Amazon	Alphabet	Facebook
Feature Subset	General Public Sentiment	All Data	Company Sentiment	Company Sentiment	All Data
Batch Size	8	8	8	8	2
Number of Layers	2	3	2	3	1
Number of Units	16	16	32	16	64
Dropout	0.5	0.5	0.5	0.5	0
Test RMSE	1.42	2.35	39.57	16.88	5.01

Table B.3 The optimal hyperparameters for the GRU model per dataset

<i>GRU</i>	Dataset				
Hyperparameter	Apple	Microsoft	Amazon	Alphabet	Facebook
Feature Subset	Company Sentiment	All Data	Stock Market Sentiment	All Data	All Data
Batch Size	2	8	4	2	4
Number of Layers	3	2	2	3	1
Number of Units	32	16	16	16	16
Dropout	0.5	0.5	0.5	0.5	0
Test RMSE	1.47	2.45	40.35	16.96	5.02

Table B.4 The optimal hyperparameters for the Transformer model per dataset

<i>Transformer</i>	Dataset				
Hyperparameter	Apple	Microsoft	Amazon	Alphabet	Facebook
Feature Subset	General Public Sentiment	Only Technical	Only Technical	Stock Market Sentiment	Stock Market Sentiment
Batch Size	2	2	2	2	2
Number of Embeddings	32	32	32	8	32
Channels	32	8	8	8	32
Number of Encoder Layers	3	3	2	1	3
Number of Decoder Layers	3	3	2	1	3
Number of Attention Heads	2	1	4	1	4
Dropout	0.5	0.5	0.5	0.5	0.5
Test RMSE	1.22	2.07	35.54	16.61	4.95



Fig. B.1 The closing price predictions for the Apple dataset

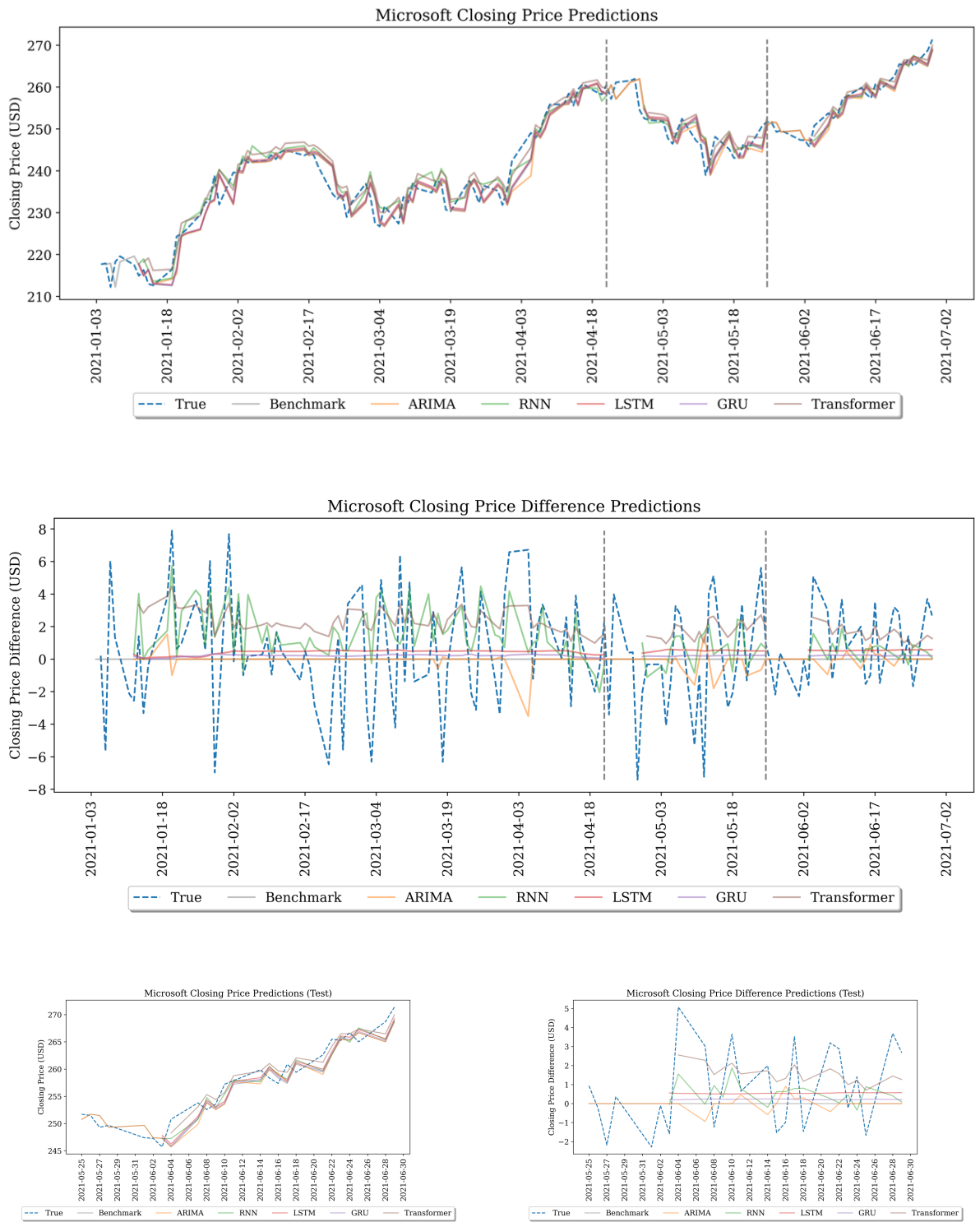


Fig. B.2 The closing price predictions for the Microsoft dataset



Fig. B.3 The closing price predictions for the Amazon dataset

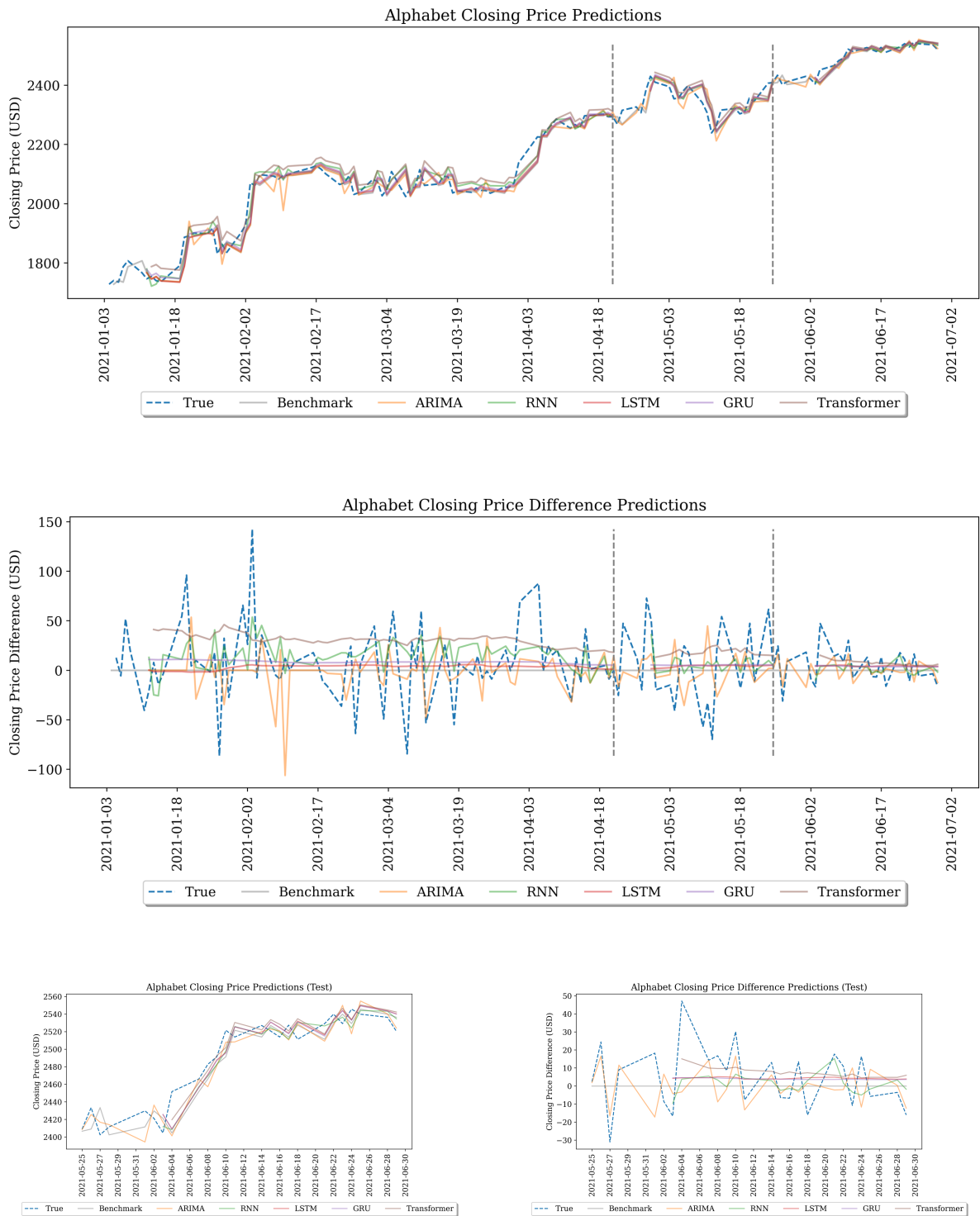


Fig. B.4 The closing price predictions for the Alphabet dataset

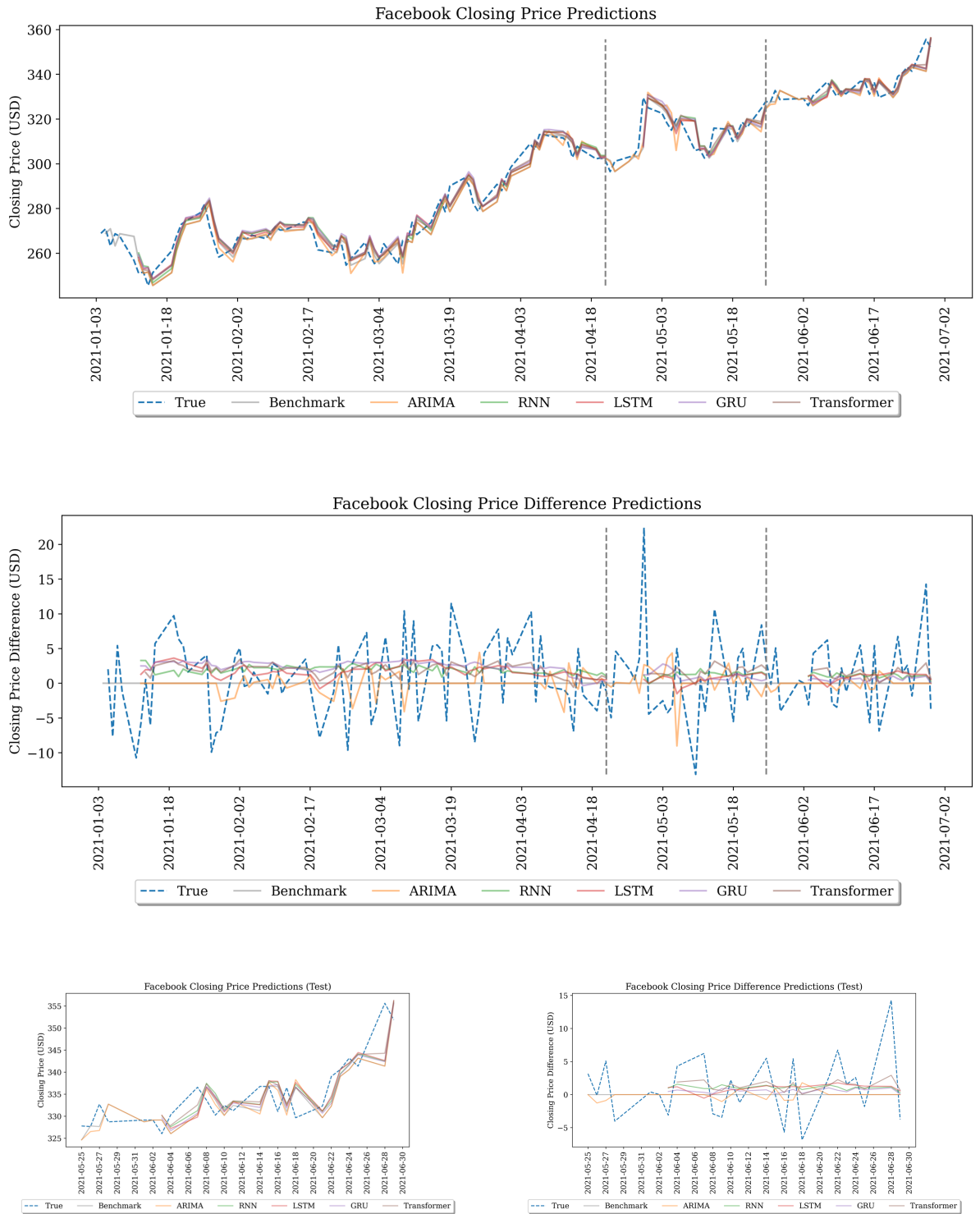


Fig. B.5 The closing price predictions for the Facebook dataset