Stock Market Predictions With Neural Networks

by

Harvey Dhillon (2102212)

MA4J5 Structures of Complex Systems

January, 2025

Contents

1 Notebook Information

The code for this project can be found in the Python notebook. The Python notebook should already have all cells executed when opened.

The Python notebook was originally written and run on Google Colab; if there are any issues running it on other environments, try running it through Google Colab.

If you are running the Python notebook on your own device, make sure the following libraries are installed:

- numpy
- matplotlib
- tensorflow
- sklearn
- datetime
- random
- yfinance

otherwise the notebook will not run successfully.

The Python notebook contains code descriptions and comments about how the model is working computationally; this summary document instead focuses on what is happening mathematically.

2 Introduction

Stock price prediction is a challenging problem because of the inherent complexity and volatility of financial markets. Many factors influence stock prices, including historical trends (how the market has performed in the past), macroeconomic indicators (such as the current GDP or employment data), and the general public sentiment towards the stock (whether investors are confident in the company/stock's ability or not). Therefore, to develop a predictive model that can accurately forecast stock prices, we need to be able to capture, for example, temporal dependencies and market sentiments.

Stock price prediction can be considered a complex system: this arises from the fact that there are several connected factors that impact (or are impacted by) stocks. Dynamic market conditions, like supply and demand and global economic events, constantly cause stock prices to change. Likewise, the mean and variance of stock price data changes all the time, complicating predictive modelling even further. On a larger scale, the financial market's high instability, caused by things like policies changing nationally or globally, can result in more unpredictable price movements.

Temporal dependencies are also highly important, as we need to model the relationship between current and historical data to capture long term trends and short term fluctuations in stock prices. Public sentiment, which we can attempt to quantify through text (in tweets relating to the stock), directly affects market behaviour, and is also affected by the market, too. These two factors will be the focus of our research in this project.

Stock price movements often include a lot of noise (random fluctuations not linked to any concrete factor), making it a struggle to discover meaningful patterns. Similarly, rare events like sudden market crashes or booms make it hard to train models: these occurrences are incredibly difficult to predict.

Deep learning is a powerful tool for tackling time series predictions and sentiment analysis, which are strategies to deal with the two factors we will be focussing on. Recurrent neural networks (RNNs) and Long short Term Memory networks (LSTMs) are widely used for time series data, as they great for learning temporal patterns and dependencies. Convolutional neural networks (CNNs) are effective for natural language processing tasks, especially sentiment analysis, since they can detect hierarchical and local patterns in textual data.

In this project, we will be studying how well these two models handle stock market forecasts on their own, then combining their results together. By using this hybrid approach, we can take advantage of the complementary strengths of each model. Theoretically, the LSTM should capture long term trends and understand historical data in stock prices, while the CNN should process meaningful signals from tweets that predict market reactions. The results from each of these models individually should be somewhat indicative of the real stock market data, but hopefully the combined output from these two strategies should give us a very accurate prediction of the market. We will use real stock data, specifically the stock price data from the Google #GOOG stock, to compare to the predicted prices to see how well all three approaches do.

3 Sentiment Analysis

Firstly, we will perform sentiment analysis on synthetically generated tweets and perform a stock price prediction using the results. We will use convolutional neural networks for this task. The synthetically generated tweets, which can be seen in the Python notebook, should be somewhat correct in their predictions, however many will be incorrect predictions, just as you would expect in reality.

3.1 Convolutional Neural Networks

Convolutional neural networks (CNNs) are neural networks which take advantage of the convolution operation. The convolution operation $*$ is

$$(\boldsymbol{W} * \boldsymbol{h})(i,j) = \sum_{m=1}^{k} \sum_{n=1}^{k} \boldsymbol{W}(m,n) \cdot \boldsymbol{h}(i+m-1, j+n-1),$$

where $\boldsymbol{W}$ is the convolutional kernel of size $k \times k$, and $\boldsymbol{h}$ is the input feature map [11, p.327-329]. The kernel essentially slides over the input, calculating a weighted sum of overlapping values at each position (i,j), producing the output feature map. Convolution allows us to learn about spatial features by capturing local patterns while preserving structural information.

Explicitly, a CNN is a function $F_\theta : \mathbb{R}^n \to \mathbb{R}^m$, parametrised by a set of weights $\theta \in \mathbb{R}^p$, where n is the dimension of the input space, m is the dimension of the output space, and p is the number of trainable parameters. The CNN F_θ is composed of layers, where some layers perform convolutions and others may perform other operations like pooling. The convolutional layer is represented as

$$\boldsymbol{h}^{(l)} = \sigma^{(l)} \left(\boldsymbol{W}^{(l)} * \boldsymbol{h}^{(l-1)} + \boldsymbol{b}^{(l)} \right),$$

where $*$ is the convolution operation and $l \in \{1, 2, \ldots, L\}$ is the current layer [13, p.97]. Here, $\boldsymbol{h}^{(l)} \in \mathbb{R}^{d_l \times h_l \times w_l}$ is the activation tensor of layer l, $\boldsymbol{h}^{(0)} = \boldsymbol{x} \in \mathbb{R}^{d_0 \times h_0 \times w_0}$ is the input tensor to the network, and $\boldsymbol{h}^{(L)}$ is the output of the network. $\boldsymbol{W}^{(l)} \in \mathbb{R}^{k_l \times k_l \times d_{l-1} \times d_l}$ is the convolutional kernel for layer l, $\boldsymbol{b}^{(l)} \in \mathbb{R}^{d_l}$ is the bias vector and $\sigma^{(l)} : \mathbb{R} \to \mathbb{R}$ is the activation function for layer l applied element-wise.

3.2 CNNs for Sentiment Analysis

CNNs are great for sentiment analysis because they can capture hierarchical and local patterns in text. This is because their convolutional layers use filters to extract features invariant to shifts in input structure, such as phrases like “great growth” or “likely drop” - these can help determine sentiment. CNNs are able to capture local dependencies by sliding this filter, which allows the network to focus on key word combinations within a fixed context window, identifying sentiment-relevant patterns. CNNs efficiently process variable-length inputs by applying a technique called max pooling after convolution. Max pooling down-samples feature maps, retaining the most important information while reducing complexity. This operation reduces the effect that variations in word order or phrasing can have on the model, improving its ability to classify sentiment accurately [13, p.96-97].

Additionally, CNNs are computationally efficient due to parallelisation. Parallelisation is a technique where computations are divided into smaller tasks that can be executed simultaneously

across multiple processing units, such as GPU cores. This significantly accelerates operations like convolutions and pooling [13, p.102].

3.3 Sentiment Input Data Preprocessing

Natural language is inherently unstructured and needs to be converted into a numerical representation for processing by the neural network. This involves two steps: tokenisation and padding. The input sequence $\boldsymbol{h}^{(0)}$ is

$$\boldsymbol{h}^{(0)} = \boldsymbol{x} \in \mathbb{R}^{d_0 \times h_0 \times w_0},$$

where d_0 is the number of input channels (1 for single text streams), h_0 is the sequence length (the number of tokens in each text), and w_0 is the dimension of each token (embedding size), which will be discussed in the layer-by-layer breakdown. The tweets are pre-processed as follows: we map words to integers. Specifically, we convert the text data into sequences of integers, where the most frequent 1000 words are retained in the vocabulary. A mathematical vocabulary $\mathcal{V} = \{w_1, w_2, \ldots, w_{|\mathcal{V}|}\}$ is constructed, where $|\mathcal{V}| \leq 1000$ is the vocabulary size. Each word w_i is mapped to an integer t_i:

$$f_{\text{token}}(w_i) = t_i, \text{ for } t_i \in \{1, 2, \ldots, |\mathcal{V}|\}.$$

For example, the tweet "Growth predicted soon" may be tokenised into $S_{\text{token}} = \{24, 54, 15\}$, where the integers correspond to indices in the vocabulary [1].

To ensure that all sequences have the same length h_0, sequences are padded. In this case, $h_0 = 20$ is the fixed sequence length. For a sequence S_{token} of length n, if $n < h_0$, padding adds $(h_0 - n)$ zeros at the end:

$$S_{\text{padded}} = \{t_1, t_2, \ldots, t_n, 0, 0, \ldots, 0\}.$$

If $n > h_0$, the sequence is truncated to the first h_0 tokens [1].

3.4 CNN Layer Breakdown

Next, let's break down the 6 layers we use in the Python notebook for this CNN.

3.4.1 Embedding Layer

The first layer of our CNN is an embedding layer, which maps the tokenised, padded sequences of integers (with length 20) to dense vector representations. Each integer corresponds to a word in the vocabulary. The layer learns a weight matrix $\boldsymbol{W}^{(1)} \in \mathbb{R}^{1000 \times 64}$, where each row represents a word's embedding. For token t_i in the input sequence $\boldsymbol{x} \in \mathbb{R}^{20}$, the embedding vector is

$$\boldsymbol{h}_i^{(1)} = \boldsymbol{W}^{(1)}[t_i],$$

the ${t_i}^{\text{th}}$ row of $\boldsymbol{W}^{(1)}$. The output for the sequence is

$$\boldsymbol{h}^{(1)} = \begin{bmatrix} \boldsymbol{h}_1^{(1)} \\ \boldsymbol{h}_2^{(1)} \\ \vdots \\ \boldsymbol{h}_{20}^{(1)} \end{bmatrix} \in \mathbb{R}^{20 \times 64}.$$

This operation transforms discrete tokens into continuous vectors, capturing semantic relationships (for example, similar words like "growth" and "increase" should have close embeddings). The

embeddings are optimised during training: this is a computationally efficient way to process the input data [5].

3.4.2 Convolutional Layer

The convolutional layer is the next layer and applies 128 convolutional filters of size 3 to the input tensor $\boldsymbol{h}^{(1)} \in \mathbb{R}^{20\times 64}$, detecting patterns which indicate sentiment (this is done by using labelled tweets and seeing what they have in common). Each filter produces a feature map, giving an output tensor $\boldsymbol{h}^{(2)} \in \mathbb{R}^{18\times 128}$. The sequence length is reduced from 20 to 18 since the kernel size is 3.

In this case, the convolution operation calculates

$$\boldsymbol{h}^{(2)}_{i,k} = \mathrm{ReLU}\left(\sum_{j=1}^{3}\sum_{m=1}^{64} \boldsymbol{W}^{(2)}_{j,m,k} \cdot \boldsymbol{h}^{(1)}_{i+j-1,m} + b^{(2)}_{k}\right),$$

where $\boldsymbol{W}^{(2)}_{j,m,k}$ and $b^{(2)}_{k}$ are the kernel weights and bias for the k^{th} filter, respectively. The activation function $\mathrm{ReLU}(x) = \max(0, x)$ introduces non-linearity by setting negative values to zero [10, p.8].

The output tensor $\boldsymbol{h}^{(2)} \in \mathbb{R}^{18\times 128}$ represents the learned feature maps, capturing phrases such as "good growth" or "likely drop" which are useful for understanding sentiment. These feature maps then get passed to subsequent layers of the CNN [2].

3.4.3 Max Pooling Layer

The max pooling layer comes next and performs down-sampling by applying the max pooling operation over non-overlapping windows of size 2, selecting the maximum value from each window. This reduces the dimension of the feature maps so they retain the most important features, while simultaneously lowering computational complexity and mitigating overfitting (when the neural network learns too much about a specific dataset and becomes less precise as a result).

For the input tensor $\boldsymbol{h}^{(2)} \in \mathbb{R}^{18\times 128}$, the pooling operation gives $\boldsymbol{h}^{(3)} \in \mathbb{R}^{9\times 128}$, halving the sequence length while preserving the depth. For the k^{th} feature map,

$$\boldsymbol{h}^{(3)}_{i,k} = \max\left(\boldsymbol{h}^{(2)}_{2i-1,k}, \boldsymbol{h}^{(2)}_{2i,k}\right),$$

where $\boldsymbol{h}^{(2)}_{2i-1,k}$ and $\boldsymbol{h}^{(2)}_{2i,k}$ are the values at positions $2i-1$ and $2i$ in the k^{th} feature map.

Max pooling reduces dimensions by halving the temporal dimension, and makes sure that more important features, like text patterns that give sentiment information, are preserved while unimportant signals are ignored. This is useful when dealing with input variations and simplifies processing to further layers [8].

3.4.4 Flatten Layer

The flatten layer reshapes the pooled feature maps $\boldsymbol{h}^{(3)} \in \mathbb{R}^{9\times 128}$ into a single one dimensional vector $\boldsymbol{h}^{(4)} \in \mathbb{R}^{1152}$. This operation prepares the data for input into dense layers, which require

vector representations. For example, if

$$\boldsymbol{h}^{(3)} = \begin{bmatrix} \boldsymbol{h}^{(3)}_{1,1} & \cdots & \boldsymbol{h}^{(3)}_{1,128} \\ \vdots & \ddots & \vdots \\ \boldsymbol{h}^{(3)}_{9,1} & \cdots & \boldsymbol{h}^{(3)}_{9,128} \end{bmatrix},$$

then the flattened vector $\boldsymbol{h}^{(4)} = \left[\boldsymbol{h}^{(3)}_{1,1}, \boldsymbol{h}^{(3)}_{1,2}, \ldots, \boldsymbol{h}^{(3)}_{1,128}, \boldsymbol{h}^{(3)}_{2,1}, \boldsymbol{h}^{(3)}_{2,2}, \ldots, \boldsymbol{h}^{(3)}_{9,128}\right]^\top$.

This operation is computationally cheap and makes it easier for the following dense layers to process the feature maps [6].

3.4.5 Dense Layers

We finish off this neural network with two dense layers [3]. The first dense layer maps the input vector $\boldsymbol{h}^{(4)} \in \mathbb{R}^{1152}$ to the output vector $\boldsymbol{h}^{(5)} \in \mathbb{R}^{64}$. The map is

$$\boldsymbol{h}^{(5)} = \mathrm{ReLU}(\boldsymbol{W}^{(5)} \cdot \boldsymbol{h}^{(4)} + \boldsymbol{b}^{(5)}),$$

where, as usual, $\boldsymbol{W}^{(5)} \in \mathbb{R}^{64\times 1152}$ is the weight matrix and $\boldsymbol{b}^{(5)} \in \mathbb{R}^{64}$ is the bias.

The final dense layer outputs a single probability value representing the likelihood of positive sentiment. The map here is

$$\hat{y} = \boldsymbol{h}^{(6)} = \sigma(\boldsymbol{W}^{(6)} \cdot \boldsymbol{h}^{(5)} + b^{(6)}),$$

where $\boldsymbol{W}^{(6)} \in \mathbb{R}^{64}$ is the weight vector, $b^{(6)} \in \mathbb{R}$ is the bias, and $\sigma : \mathbb{R} \to \mathbb{R}$ is the sigmoid activation function [10, p.8],

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

The sigmoid function squeezes $\boldsymbol{W}^{(6)} \cdot \boldsymbol{h}^{(5)} + b^{(6)}$ into the range $[0, 1]$, making it suitable for probability outputs. The sigmoid's derivative, $\sigma'(x) = \sigma(x)(1 - \sigma(x))$, guarantees smooth gradients for backpropagation when training.

3.5 CNN Training Process

The dataset of synthetic tweets is split into a labelled and an unlabelled subset. Labelled tweets have sentiment labels (0 for negative, 1 for positive) forming the target vector $\boldsymbol{y} = [y_1, y_2, \ldots, y_n]$, for n tweets. We handle different amounts of positive and negative tweets in the labelled subset by using class weights

$$w_s = \frac{n}{k \cdot n_s},$$

where $k = 2$ is the number of classes and n_s is the number of samples in class s. These weights make sure that learning is balanced as they modify loss contributions [9].

The binary cross-entropy loss function with weights,

$$L_\theta(\boldsymbol{y}, \hat{\boldsymbol{y}}) = \frac{1}{m} \sum_{i=1}^{m} w_{y_i} \left[y_i \log \hat{y}_i + (1 - y_i) \log(1 - \hat{y}_i) \right],$$

where $\boldsymbol{y}$ and $\hat{\boldsymbol{y}}$ are true and predicted sentiment labels, is used for training [9]. Our CNN model uses the Adam (Adaptive Moment Estimation) optimiser, which adjusts weights based on gradients. In the Adam optimiser, we use moving averages for two moments: the first moment ($\boldsymbol{m}_t$) and second

moment ($\boldsymbol{v}_t$), which each get updated by

$$\boldsymbol{m}_t = \beta_1 \boldsymbol{m}_{t-1} + (1-\beta_1)\nabla_\theta L_\theta(\boldsymbol{y}, \hat{\boldsymbol{y}}), \text{ and } \boldsymbol{v}_t = \beta_2 \boldsymbol{v}_{t-1} + (1-\beta_2)(\nabla_\theta L_\theta(\boldsymbol{y}, \hat{\boldsymbol{y}}))^2,$$

where $\beta_1, \beta_2 \in [0, 1)$ are decay rates, and $L_\theta(\boldsymbol{y}, \hat{\boldsymbol{y}})$ is the loss. We update the weights using

$$\theta_{t+1} = \theta_t - \alpha \frac{\boldsymbol{m}_t}{\sqrt{\boldsymbol{v}_t} + \varepsilon},$$

where α is the learning rate and $\varepsilon > 0$ is a small constant to prevent division by 0 [12].

To avoid overfitting, we implement an early stopping method to stop training if the loss value does not improve for three consecutive epochs, restoring the best weights [15]. An epoch is one complete iteration through the entire training dataset during this training process. The model is then trained for up to 10 epochs with a 20% validation split.

After training, the model predicts sentiment for unlabelled tweets. The probability of positive sentiment is thresholded at 0.5, using the output $\hat{\boldsymbol{y}}$,

$$\hat{y}_i = \begin{cases} 1 \text{ (positive)} & \text{if } \hat{y}_i > 0.5, \\ 0 \text{ (negative)} & \text{otherwise,} \end{cases}$$

and these predictions are assigned to the unlabelled tweets.

3.6 Analysis of CNN-Based Predictions

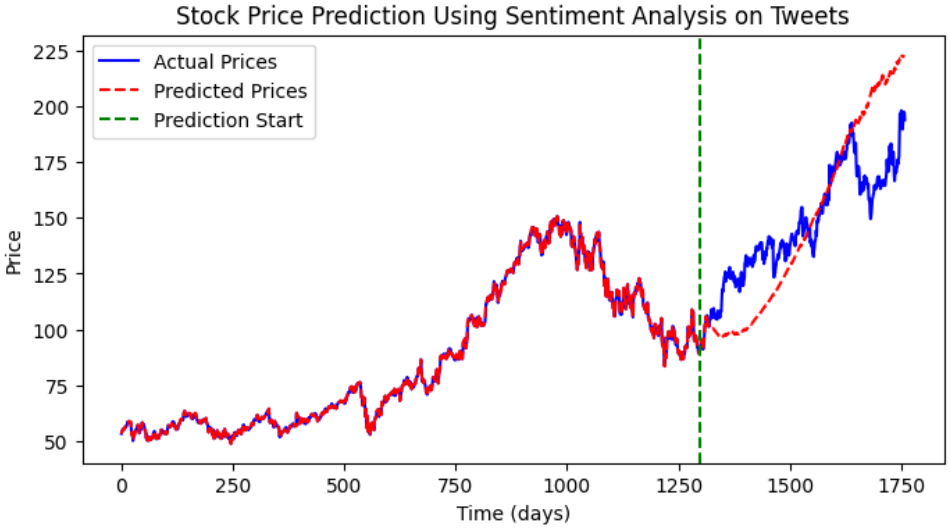

Figure 1: Stock Prices Predicted Using Sentiment Analysis

Figure 1 illustrates how effective the CNN-based sentiment analysis model is for predicting stock prices using synthetic tweets which mimic real tweets. The blue line represents actual stock prices, while the red dashed line shows predicted prices based on aggregated sentiment data. The vertical green line indicates the start of predictions: we make sure only past data is used for training, with extrapolation beyond this point. Essentially, we are simulating predictions after a certain point and comparing the real stock data to it.

After using the CNN to process and compute synthetic tweets sentiments, we work out daily

sentiment scores $S_{\text{sentiment},j}$ as the average sentiment of all tweets for day j,

$$S_{\text{sentiment},j} = \frac{1}{n_j} \sum_{i=1}^{n_j} \hat{y}_i,$$

where n_j is the number of tweets for day j. The stock price prediction for day j, $P_{\text{sentiment},j}$, is updated iteratively using:

$$P_{\text{sentiment},j} = P_{\text{sentiment},j-1} \cdot (1 + \alpha(S_{\text{sentiment},j} - 0.5)),$$

where α is a scaling factor controlling price sensitivity to sentiment.

From Figure 1, beyond the prediction start point, the extrapolated prices diverge somewhat from the real data but still capture the general upward trend, demonstrating the model's ability to generalise sentiment of tweets and evaluate future price movements. This behaviour emphasises the importance of the CNN accurately classifying tweet sentiments, as small errors in sentiment analysis could lead to significant price deviations.

The training performance, as seen in the Python notebook, confirms the CNN model has learnt well. The validation loss after all epochs is incredibly small when training the model. Because of our use of a balanced dataset, class weights, and the CNN architecture, we have created a model which is capable of portraying long term trends correctly in a general way.

While the model demonstrates the effectiveness of sentiment analysis for stock price prediction, there are several limitations. Firstly, the model assumes a linear relationship between aggregated sentiment scores and price movements, which may oversimplify the dynamics of real-world stock markets. The sharp upward trend and higher end price in the predicted prices highlights sensitivity to small sentiment shifts, amplified by the scaling factor α, which could potentially lead to unrealistic price predictions.

Additionally, the model only accounts for tweet sentiment and does not account for external factors such as market news, economic indicators, or macroeconomic conditions, which can significantly influence stock prices. These limitations suggest that while the CNN model captures general trends, it cannot predict the noisy movements in stock prices that may arise from other factors.

4 Time Series Prediction

Next, we will use time series prediction on stock price data to see how precise our stock price prediction is from this method. We will use recurrent neural networks (RNNs) - specifically Long short Term Memory networks - to do this. Theoretically, the stock data patterns (including noise) should inspire our stock predictions.

4.1 Recurrent Neural Networks

An RNN is a function $F_\theta : \mathbb{R}^n \to \mathbb{R}^m$, parameterised by a set of weights $\theta \in \mathbb{R}^p$, where n is the dimension of the input space, m is the dimension of the output space, and p is the number of trainable parameters. The function F_θ processes sequences of data by maintaining a hidden state $\boldsymbol{h}^{(t)}$ that evolves over time t based on the current input and the previous hidden state. The hidden state update is

$$\boldsymbol{h}^{(t)} = \sigma_h \left(\boldsymbol{W}_h \boldsymbol{h}^{(t-1)} + \boldsymbol{W}_x \boldsymbol{x}^{(t)} + \boldsymbol{b}_h \right), \text{ for } t \in \{1, 2, \ldots, T\},$$

where σ_h is an activation function, $\boldsymbol{W}_h \in \mathbb{R}^{d_h \times d_h}$ is the recurrent weight matrix, $\boldsymbol{W}_x \in \mathbb{R}^{d_h \times n}$ is the input weight matrix, and $\boldsymbol{b}_h \in \mathbb{R}^{d_h}$ is the bias vector.

The output $\boldsymbol{y}^{(t)} \in \mathbb{R}^m$ at each time step t is

$$\hat{\boldsymbol{y}}^{(t)} = \sigma_y \left(\boldsymbol{W}_y \boldsymbol{h}^{(t)} + \boldsymbol{b}_y \right),$$

where σ_y is an activation function, $\boldsymbol{W}_y \in \mathbb{R}^{m \times d_h}$ is the output weight matrix, and $\boldsymbol{b}_y \in \mathbb{R}^m$ is the output bias vector.

The network processes sequences by iterating through the input over time, with the final output $F_\theta(\boldsymbol{x})$ being a sequence $\{\hat{\boldsymbol{y}}^{(1)}, \hat{\boldsymbol{y}}^{(2)}, \ldots, \hat{\boldsymbol{y}}^{(T)}\}$ or possibly a single output $\hat{\boldsymbol{y}}^{(T)}$ [11, p.369-376].

4.2 Long Short Term Memory Networks

A Long short Term Memory network (LSTM) is a special type of RNN which captures long term dependencies in sequential data. An LSTM maintains a cell state $\boldsymbol{c}^{(t)} \in \mathbb{R}^{d_h}$ in addition to the hidden state $\boldsymbol{h}^{(t)} \in \mathbb{R}^{d_h}$. At each time step t, the LSTM performs some additional operations. The forget gate $\boldsymbol{f}$ determines which information to discard from the cell state:

$$\boldsymbol{f}^{(t)} = \sigma \left(\boldsymbol{W}_f \boldsymbol{x}^{(t)} + \boldsymbol{U}_f \boldsymbol{h}^{(t-1)} + \boldsymbol{b}_f \right),$$

where $\boldsymbol{W}_f \in \mathbb{R}^{d_h \times n}$, $\boldsymbol{U}_f \in \mathbb{R}^{d_h \times d_h}$, and $\boldsymbol{b}_f \in \mathbb{R}^{d_h}$ are the forget gate weights, and σ is the sigmoid activation function. The input gate decides which information to update in the cell state:

$$\boldsymbol{i}^{(t)} = \sigma \left(\boldsymbol{W}_i \boldsymbol{x}^{(t)} + \boldsymbol{U}_i \boldsymbol{h}^{(t-1)} + \boldsymbol{b}_i \right) \text{ and } \tilde{\boldsymbol{c}}^{(t)} = \tanh \left(\boldsymbol{W}_c \boldsymbol{x}^{(t)} + \boldsymbol{U}_c \boldsymbol{h}^{(t-1)} + \boldsymbol{b}_c \right),$$

where $\boldsymbol{W}_i, \boldsymbol{W}_c \in \mathbb{R}^{d_h \times n}$, $\boldsymbol{U}_i, \boldsymbol{U}_c \in \mathbb{R}^{d_h \times d_h}$ and $\boldsymbol{b}_i, \boldsymbol{b}_c \in \mathbb{R}^{d_h}$ are the weights for the input gate and candidate cell state, respectively. The cell state is updated by

$$\boldsymbol{c}^{(t)} = \boldsymbol{f}^{(t)} \odot \boldsymbol{c}^{(t-1)} + \boldsymbol{i}^{(t)} \odot \tilde{\boldsymbol{c}}^{(t)},$$

where $\odot$ is the element-wise Hadamard product. Finally, the output gate determines the hidden state and output by

$$\boldsymbol{o}^{(t)} = \sigma \left(\boldsymbol{W}_o \boldsymbol{x}^{(t)} + \boldsymbol{U}_o \boldsymbol{h}^{(t-1)} + \boldsymbol{b}_o \right) \text{ and } \boldsymbol{h}^{(t)} = \boldsymbol{o}^{(t)} \odot \tanh \left(\boldsymbol{c}^{(t)} \right),$$

where $\boldsymbol{W}_o \in \mathbb{R}^{d_h \times n}$, $\boldsymbol{U}_o \in \mathbb{R}^{d_h \times d_h}$, and $\boldsymbol{b}_o \in \mathbb{R}^{d_h}$ are the output gate weights.

Like with RNNs, the final output of the LSTM can be a sequence or a single output [14, p.3-4].

4.3 LSTMs for Time Series Prediction

LSTMs are suitable for time series prediction because they can model sequential data while retaining historical information through hidden and cell states. This enables them to predict future values even with complex temporal dependencies, which should be ideal for stock price prediction where past trends are often indicative of future movements in the data.

One of the advantages of LSTMs is they can handle variable-length sequences, processing data one time step at a time while retaining relevant information. Bidirectional LSTMs extend this even further by processing input sequences in both forward and backward directions, capturing both historical and future context simultaneously by concatenating their results:

$$\boldsymbol{h}_{\text{bi}}^{(t)} = \begin{bmatrix} \boldsymbol{h}_{\text{fwd}}^{(t)} \\ \boldsymbol{h}_{\text{bwd}}^{(t)} \end{bmatrix},$$

where $\boldsymbol{h}_{\text{fwd}}^{(t)}$ and $\boldsymbol{h}_{\text{bwd}}^{(t)}$ are the hidden states from the forward and backward passes [7].

LSTMs are excellent at identifying trends by modelling both local variations and global behaviours. A variety of layers, including dropout layers after LSTMs, counteract overfitting, to ensure the learned representation generalises well to new, unseen data, alongside dense layers which allow the model to combine temporal patterns with non-linear transformations for predictions.

4.4 Time Series Input Data Preprocessing

The input data for the LSTM model consists of historical closing prices of the stock and additional features derived from moving averages. The closing prices P_{actual} are retrieved from 1$^{\text{st}}$ January 2018 to December 2024, for T days.

To capture the trends, we calculate two moving averages over the training period:

$$\text{7-day MA}_t = \frac{1}{7}\sum_{i=t-6}^{t} P_{\text{actual},i} \text{ and 30-day MA}_t = \frac{1}{30}\sum_{i=t-29}^{t} P_{\text{actual},i},$$

where t is the current day, and rows with insufficient data are omitted. The closing prices and moving averages are normalised to $[0, 1]$. We denote the normalised prices by $\boldsymbol{x}_{\text{price}} \in [0, 1]^T$ and the normalised moving averages by $\boldsymbol{x}_{\text{ma}} \in [0, 1]^{T\times 2}$ and combine them into a feature matrix, $\boldsymbol{x} = [\boldsymbol{x}_{\text{price}}, \boldsymbol{x}_{\text{ma}}]$, where $\boldsymbol{x} \in [0, 1]^{T\times 3}$.

For training, the data is divided into overlapping sequences of 100 days, with the target being the closing price on the 101$^{\text{st}}$ day. The input sequences are

$$\boldsymbol{X} = \{\boldsymbol{x}_1, \boldsymbol{x}_2, \ldots, \boldsymbol{x}_{T-100}\},$$

where each $\boldsymbol{x}_i$ contains 100 days of normalised features. The targets for training are the closing prices for the following days,

$$\boldsymbol{y} = \{P_{\text{actual},101}, P_{\text{actual},102}, \ldots, P_{\text{actual},T}\}.$$

This setup allows the model to predict the next day's price based on the previous 100 days of data. The stock price dataset is split into training (first 75%) and test (remaining 25%) subsets. With this, the model leverages both past prices and trends from moving averages for accurate future predictions.

4.5 LSTM Layer Breakdown

Next, let's break down the layers used in the Python notebook for this LSTM neural network.

4.5.1 Bidirectional LSTM Layers

The first and third layers of this particular neural network are bidirectional LSTM layers. As mentioned briefly earlier, this type of layer processes sequences of time series data, learning both forward and backward dependencies, capturing both past and future information about stock prices.

For the first layer, the LSTM cell processes input sequences of dimensions $1000 \times 100 \times 3$: we have 1000 samples, the sequence length is 100, and there are 3 feature dimensions (normalised price and both moving averages). At each time step t, the LSTM maintains and updates hidden ($\boldsymbol{h}_t$) and cell ($\boldsymbol{c}_t$) states. The bidirectional LSTM applies two separate LSTM cells to the input: one processes the sequence forward ($t = 1 \to 100$), and the other backward ($t = 100 \to 1$). The output at each time step combines the forward and backward hidden states, resulting in an output tensor of dimensions $1000 \times 100 \times 256$, where 128 is the hidden state size of each LSTM cell, and is doubled to account for the concatenation between the forward and backward cell.

The third layer instead focuses on producing a single hidden state vector $\boldsymbol{h}^{(3)} \in \mathbb{R}^{128}$, and this summary captures the most significant temporal dependencies learned from the sequence, condensing them into a fixed-length representation suitable for the dense layers later on [7].

4.5.2 Dropout Layers

We apply the dropout layer after each Bidirectional LSTM layer: this is a regularisation technique to prevent overfitting. We use a dropout rate of 0.3, so there is a 30% chance of neurons in the preceding layer being randomly set to zero during each training iteration. This stops the model from relying heavily on specific neurons and encourages learning generalised features instead.

For the output $\boldsymbol{h}^{(t-1)} \in \mathbb{R}^d$, a binary mask $\boldsymbol{r} \in \{0,1\}^d$ is sampled from a Bernoulli distribution,

$$r_i \sim \text{Bernoulli}(1 - 0.3),$$

where r_i indicates whether the i^{th} neuron is active. The dropout layer output is

$$\tilde{\boldsymbol{h}}^{(t)} = \frac{\boldsymbol{r} \odot \boldsymbol{h}^{(t-1)}}{1 - 0.3},$$

with $\odot$ as element-wise multiplication and $1/0.7$ maintaining the expected sum of activations. During inference (the stage where predictions are made), dropout is disabled, and all neurons remain active. This ensures the full network capacity is used for predictions. When dropout is used, the learned features are more evenly distributed across all neurons, avoiding overfitting. The dropout layers in this LSTM model are layers 2 and 4, following both LSTM layers [4].

4.5.3 Dense Layers

Like the CNN model, this LSTM model features two dense layers to convert the extracted temporal features into a stock price prediction that is useful for us to analyse.

The first dense layer, configured with 64 units and a ReLU activation function, processes the summarised temporal features $\boldsymbol{h}^{(4)} \in \mathbb{R}^{128}$ from the third LSTM layer after dropout regularisation

in the fourth layer. This layer reduces the dimension from 128 to 64 while applying transformations to capture the complex dependencies. The ReLU activation introduces sparsity, focusing the network on the most important patterns, which prepares the data for the final layer.

The final layer is a second dense layer, which has a 1 dimension output layer with no activation function. It transforms the vector from the 5^{th} layer into the predicted stock price. The final output is the normalised stock price, later rescaled to the original price range (recall that earlier we scaled everything to fit in a range of $[0, 1]$) [3].

4.6 LSTM Training Process

The training process is similar to that of the CNN, again using the Adam optimiser. Like the CNN model, there is an Early Stopping callback to prevent further training if the loss does not improve: this is another method for preventing overfitting and wasting resources [15].

Differently this time, I have opted to use a custom loss function in this model. The loss function

$$L_\theta(\boldsymbol{y}, \hat{\boldsymbol{y}}) = \underbrace{\frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2}_{\text{Value Loss (MSE)}} + \underbrace{\frac{\lambda_1}{n-1}\sum_{i=1}^{n-1}(\Delta y_i - \Delta \hat{y}_i)^2}_{\text{Trend Loss}} + \underbrace{\lambda_2 \sum_{i=1}^{n}|\hat{y}_i|}_{\text{Regularisation}} ,$$

where:

$$\Delta y_i = y_{i+1} - y_i, \quad \Delta \hat{y}_i = \hat{y}_{i+1} - \hat{y}_i.$$

is meant to improve prediction accuracy by balancing point-wise errors and trend consistency; this is critical for time series forecasting. It combines three different components: mean squared error (MSE), trend loss and regularisation [11, p.233-236]. In the Python notebook, I've used $\lambda_1 = 0.1$ to penalise deviations in trends, and $\lambda_2 = 0.01$ to regularise predictions to prevent large values.

4.7 Analysis of LSTM-Based Predictions

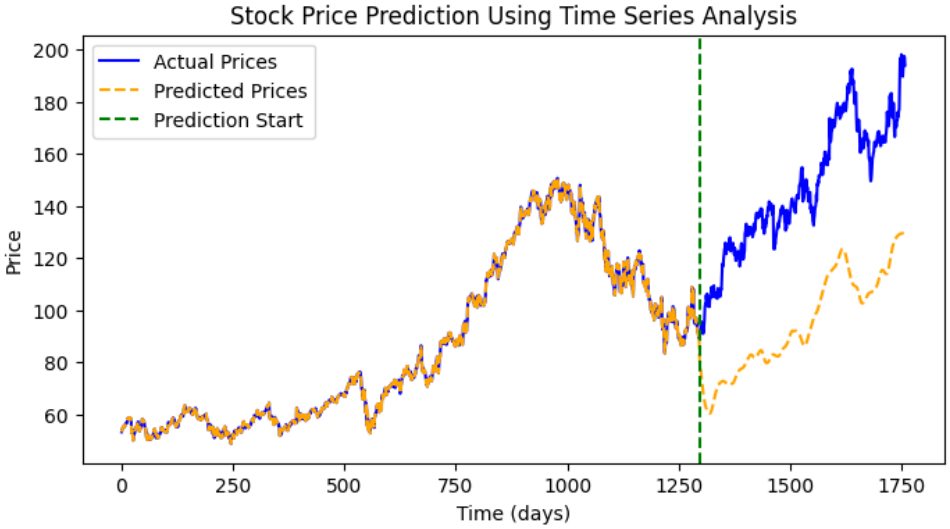

Figure 2: Stock Prices Predicted Using Time Series Prediction

Figure 2 illustrates the performance of the LSTM-based model in predicting stock prices over time.

After the split index, the predicted prices initially follow the actual trend but diverge over time

due to the challenges of long term predictions in volatile stock markets. External factors absent from the training data, such as market news, contribute to this divergence. However, the graph showcases the general trends, reflectingthe model's ability to learn long term dependencies.

The minimal gap between training and validation loss, as seen in the Python notebook, shows strong generalisation and makes clear that overfitting has been avoided; the custom loss function's trend penalties has helped achieve this. This ensures the model captures both point-wise accuracy and overall price directionality. However, the patterns are captured effectively by the model, showing a strong aptitude for recreating the noise and behaviour we see in the real model.

It should be noted that the model only uses 5 epochs instead of 10 like the sentiment analysis CNN model: the training time was significantly longer than that of the CNN training period, and although I personally am not as interested in the computational efficiency of the models compared to their results, this is a penalty for the time series prediction model.

Overall, the LSTM model effectively learns temporal dependencies and trends, performing well on training data and capturing general patterns in predictions, despite the inherent challenges of long term stock price forecasting. It fails in its precision to match the actual prices shown on the graph, likely for similar reasons that the CNN model fail: we are not considering many factors and only using one approach. The next section should remedy this, though.

5 Combining Both Forecasting Models

Now, we attempt to recover the advantages of both models we have explored by using them together for a final prediction.

5.1 Why Combine Models?

The individual limitations of the sentiment analysis and time series models highlight the need for a combined framework to improve predictive accuracy. While the CNN model is effective for abrupt sentiment shifts, this approach overlooks historical trends and long term price dynamics, often overemphasising short term fluctuations during periods of weak sentiment signals or inconsistent market behaviour.

Conversely, the time series model captures temporal dependencies but lacks awareness of external factors, such as market sentiment or news events, limiting its performance during volatile periods driven by these factors.

Combining these models addresses their respective weaknesses partially. The sentiment model captures overall market behaviour, while the time series model provides stability by learning patterns and dependencies. Here, predictions are generated by averaging outputs from the sentiment ($P_{\text{sentiment}}$) and time series ($P_{\text{time series}}$) models in a weighted manner,

$$P_{\text{combined},i} = W_{\text{sentiment}} \cdot P_{\text{sentiment},i} + W_{\text{time series}} \cdot P_{\text{time series},i},$$

where $P_{\text{sentiment}}$ and $P_{\text{time series}}$ are the predictions for each model and $W_{\text{sentiment}}$ and $W_{\text{time series}}$ are the weights for the average. This basic combination balances short and long term perspectives, reduces biases, and smooths prediction errors, hopefully giving a more accurate representation of future stock prices.

5.2 Analysis of Combined Models Predictions

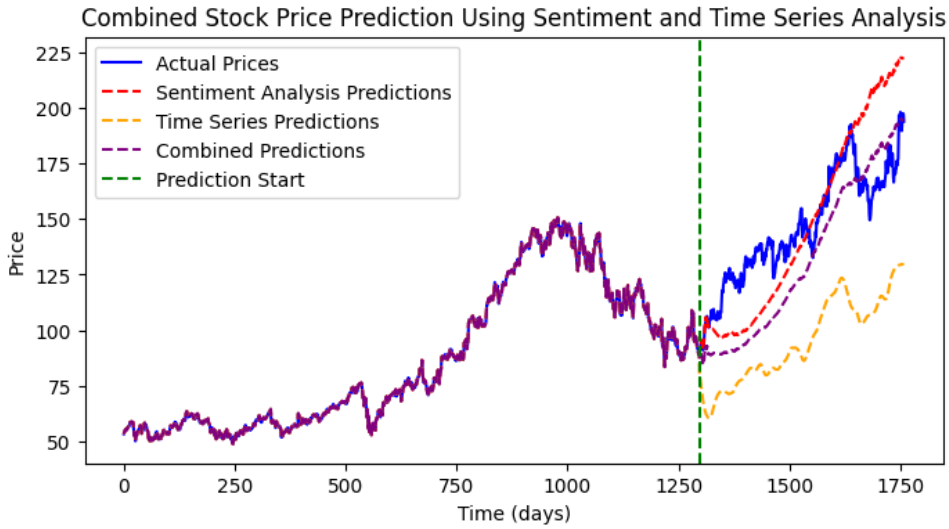

Figure 3: Stock Prices Predicted From All 3 Strategies

The combined model integrates the strengths of sentiment analysis and time series models to produce balanced stock price predictions. Figure 3 shows the effectiveness of this approach compared to the two other strategies on their own. The combined model leverages sentiment analysis for market responsiveness and time series prediction for trend stability, giving us a more accurate and robust forecast.

Unfortunately, the prediction is not completely accurate: the patterns captured in the time series model are dulled by averaging with the results of the sentiment analysis model, so perhaps we could improve the prediction by taking into account more factors of this complex system, such as macroeconomic data.

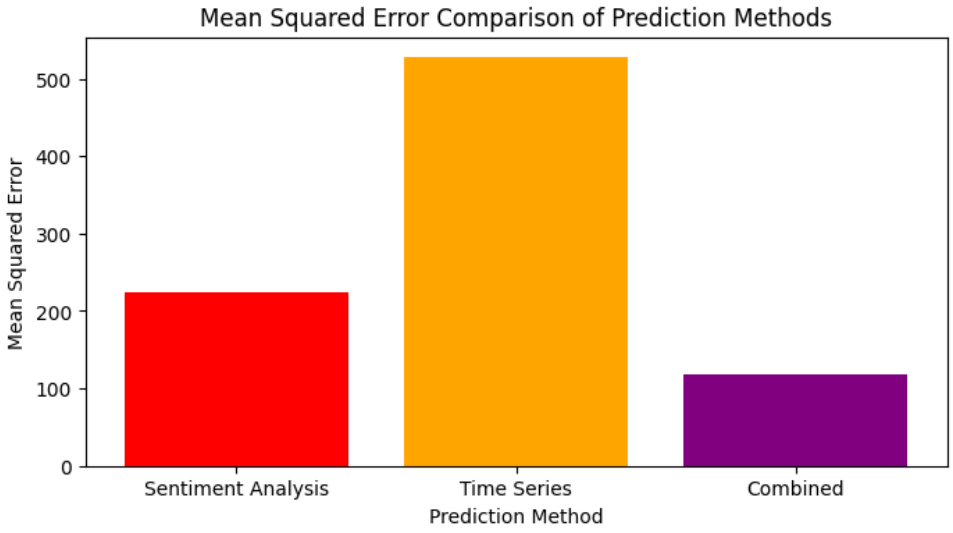

Figure 4: Mean Squared Error of All 3 Strategies

The bar chart in Figure 4 offers another measure of effectiveness, showing that the combined model

is reducing the mean squared error (MSE). The MSE for the combined model is

$$\text{MSE}_{\text{combined}} = \frac{1}{n}\sum_{i=1}^{n}\left(P_{\text{actual},i} - P_{\text{combined},i}\right)^2,$$

where $P_{\text{actual},i}$ is the actual stock price and $P_{\text{combined},i}$ is the combined prediction. Averaging the square error reduces error variance, improving predictive performance. The sentiment analysis model ($\text{MSE}_{\text{sentiment}}$) achieves lower MSE than the time series model ($\text{MSE}_{\text{time series}}$), indicating better short term predictions. However, the combined model achieves the lowest MSE ($\text{MSE}_{\text{combined}}$), indicating superiority over both other approaches.

6 Conclusion

Throughout this project, we have presented a comprehensive approach to stock price prediction by considering sentiment analysis and time series modelling individually using deep learning techniques, then bringing both results together by combining both approaches. This hybrid strategy integrates the strengths of CNNs and LSTMs to somewhat address the inherent complexity of financial markets and stock prices. Since it leverages market sentiment data from synthetic tweets and historical price trends, the combined model offers a balanced and robust prediction strategy.

Individually considering the strategies, the CNN sentiment analysis model effectively captured short term market dynamics by analysing text data to classify sentiments as positive or negative. This allowed the model to take advantage of tweets about the market to give forecasts on stock prices. Meanwhile, the LSTM time series model focused on capturing long term dependencies and historical trends in stock prices, giving a strong prediction of patterns in the stock data.

Furthermore, the inclusion of custom loss functions, moving average features, and learning rate schedules enhanced performance and stability. The integration of these two models by averaging mitigated some of the shortcomings that each individual approach had. The combined model balanced short term and long term perspectives, leading to higher accuracy and smaller errors. We can see from the reduced mean squared error of the combined predictions compared to those generated by the individual models that there was a positive impact from the hybrid strategy. The combined approach aligns closer with actual stock prices, making it a promising tool for stock price forecasting.

Despite its strengths, the combined model still has limitations: it still relies on synthetic sentiment data, which could be fixed by using paid Twitter APIs for real sentiment data. Additionally, potential challenges in adapting to sudden market shifts caused by unforeseen events will still hinder the model. Future work could involve implementing other factors in this complex system to give a holistic view of the stock market (as we discussed earlier). Additionally, advanced techniques such as attention mechanisms or ensemble learning could be applied to further improve accuracy.

Overall, this project highlights the potential of using deep learning techniques for solving complex financial problems. By combining sentiment analysis and time series modelling, we have demonstrated a framework capable of capturing the nature of stock price movements. This approach can serve as a foundation for future advancements in stock market forecasting.

References

[1] Preparing text data for transformers. medium.com/@lokaregns/preparing-text-data-for-transformers-tokenization-mapping-and-padding-9fbfbce28028, 2023.

[2] tf.keras.layers.conv1d. tensorflow.org/api_docs/python/tf/keras/layers/Conv1D, 2024.

[3] tf.keras.layers.dense. tensorflow.org/api_docs/python/tf/keras/layers/Dense, 2024.

[4] tf.keras.layers.dropout. tensorflow.org/api_docs/python/tf/keras/layers/Dropout, 2024.

[5] tf.keras.layers.embedding. tensorflow.org/api_docs/python/tf/keras/layers/Embedding, 2024.

[6] tf.keras.layers.flatten. tensorflow.org/api_docs/python/tf/keras/layers/Flatten, 2024.

[7] tf.keras.layers.lstm. tensorflow.org/api_docs/python/tf/keras/layers/LSTM, 2024.

[8] tf.keras.layers.maxpool1d. tensorflow.org/api_docs/python/tf/keras/layers/MaxPool1D, 2024.

[9] How to improve class imbalance using class weights in machine learning? analyticsvidhya.com/blog/2020/10/improve-class-imbalance-class-weights/, 2024.

[10] Julius Berner, Philipp Grohs, Gitta Kutyniok, and Philipp Petersen. *The Modern Mathematics of Deep Learning*. Cambridge Press, 2021.

[11] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016.

[12] Diederik Kingma. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[13] Ragav Venkatesan and Baoxin Li. *Convolutional Neural Networks in Visual Computing: A Concise Guide*. CRC Press, 2018.

[14] Christian Bakke Vennerød, Adrian Kjærran, and Erling Stray Bugge. Long short-term memory rnn. 2021.

[15] Xue Ying. An overview of overfitting and its solutions. *Journal of Physics: Conference Series*, 2019.