Stock Prediction With Deep Learning

Parthraj Veluri, Chien-Yun Hsu, Siu Lun Alan Choi, Zenan Tang
AASD 4010 DL I - Group 1

February 8, 2025

Team Contributions

The following table outlines the contributions of each team member to the project:

Team Member	Contribution
Parthraj Veluri	LSTM Model training
Chien-Yun Hsu	CNN Model training
Siu Lun Alan Choi	RNN Model training
Zenan Tang	BERT Model training

Contents

1 Introduction

Inflation represents the gradual increase in prices of goods and services over time, which erodes the purchasing power of money. A dollar today will buy less in the future as prices rise—what once cost $1.00 might require $1.03 or more to purchase just a year later. This continuous decline in purchasing power poses a significant challenge for individuals looking to preserve and grow their wealth over the long term.

Historical data shows that keeping money in traditional savings accounts rarely keeps pace with inflation. While bank savings accounts might offer interest rates of 0.5–2% annually, inflation in developed economies typically averages 2–3% per year, with some periods seeing much higher rates. This means that money sitting in savings accounts actually loses real value over time, despite appearing to maintain its nominal value.

This reality creates a compelling need for individuals to invest in assets that have historically provided returns exceeding the inflation rate. Stocks and bonds have proven to be effective tools for this purpose. Over the long term, the stock market has delivered average annual returns of approximately 7–10% after adjusting for inflation, while bonds have typically provided returns of 3–5%. These investment vehicles allow investors to not only preserve their purchasing power but potentially grow their wealth in real terms.

Another related problem in a society structured on capitalism is the imbalance of power associated with returns on capital depending on initial investment size. For example, a return of 100% is significant but meaningless if the investment is low (i.e., $1), whereas even a return of 1% can generate significant capital with a sufficiently large investment (i.e., $1M).

The need to outpace inflation while generating yields that are financially meaningful to individuals brought about a desire to make money fast by predicting stock prices. In doing so, we leverage various models (LSTMs, CNNs, RNNs) combined with news sentiment analysis (BERT) to attempt to predict daily directional movements in stock prices by using previous news and historical closing prices. In this pilot project, to reduce computational complexity, we create models for stock prediction based on only one stock: Nvidia (ticker: NVDA). NVDA was chosen for its high market capitalization and an abundance of news relevant to the corporation.

2 Data Preparation

2.1 Importing and Fetching Data

The stock market data for NVIDIA (NVDA) is acquired from Yahoo Finance for the period from 2000 to 2024. The dataset includes key financial metrics such as Open, High, Low, Close prices, and Volume.

2.2 Computing Technical Indicators

To enhance model performance, additional features are computed:

- **10-day Moving Average (MA10):** The average of the closing prices over the past 10 days.
- **50-day Moving Average (MA50):** The average of the closing prices over the past 50 days.
- **Relative Strength Index (RSI):** A momentum indicator that evaluates the speed and change of price movements.

2.3 Scaling Data

To ensure consistency across features, Min-Max scaling is applied. This process normalizes each feature to a range between 0 and 1, according to the transformation:

$$\mathbf{x}' = \frac{\mathbf{x} - \min(\mathbf{x})}{\max(\mathbf{x}) - \min(\mathbf{x})}.$$

2.4 Reshaping Data into Tensor Format

To facilitate the use of time-series models, the data is reshaped into a 3D tensor.

2.5 Train-Test Split

The dataset is divided into three subsets to support robust model evaluation:

- **Training Set (80%):** Used to train the models.
- **Test Set (20%):** Reserved for the final evaluation of the model's performance.

With these steps, the dataset is now fully prepared for training and evaluating four different predictive models.

3 RNN Encoder-Decoder Architeture in stock price prediction

Using an RNN Encoder-Decoder for stock price prediction offers a robust approach to capture temporal dependencies in time-series data. The model is designed with the following key aspects:

- A custom RNN cell that employs Leaky ReLU activation.
- Processing historical stock data to forecast future prices.
- Training by minimizing the loss between predicted and actual prices.

The RNN Encoder-Decoder model consists of two main parts:

- **Encoder:** Converts the input sequence (historical stock prices) into a set of hidden states.
- **Decoder:** Uses the hidden states to generate future stock price sequences.

In the following, we will introduce why do we use **Seq2Seq architectures** rather than traditional **RNN**.

3.1 Time-Shift Problem in Traditional RNNs for Stock Prediction

Recurrent Neural Networks (RNNs) have been widely used for time-series forecasting, including stock price prediction. However, a **time-shift problem** arises when using traditional RNNs in such tasks. This issue occurs because RNNs find a shortcut to minimize to loss function rather than capturing real underlying trend.

In stock prediction, we model the true price at a given time step as consisting of a slowly varying trend component and a random noise term. A naive RNN model often learns to copy the last observed value rather than capturing the actual trend. While this strategy minimizes short-term prediction errors, it fails to generalize well.

This section introduces a formal mathematical analysis comparing two types of estimators: 1. A **naive estimator** that simply copies the last observed value. 2. An **explicit trend estimator** that attempts to model the underlying trend directly.

By analyzing the Mean Squared Error (MSE) of these approaches, we demonstrate why naive models often outperform explicit trend models unless the trend estimation is highly accurate. This leads to the motivation for **Seq2Seq architectures**, which are designed to capture long-term dependencies in time-series data more effectively.

3.2 General Formulation and Comparison of Estimators

Assume that the true stock price at time t is given by

$$y_t = f(t) + \epsilon_t,$$

where:

- $f(t)$ is the underlying trend, which is slowly varying (i.e., $f(t) \approx f(t-1)$),
- ϵ_t is a zero-mean noise term with variance σ^2.

We now consider two different estimators for y_t:

3.2.1 Naive Estimator

The naive estimator simply copies the last observed value:

$$\hat{y}_t^{(\text{naive})} = y_{t-1} = f(t-1) + \epsilon_{t-1}.$$

Thus, the difference from the previous observation is

$$\hat{y}_t^{(\text{naive})} - y_{t-1} = y_{t-1} - y_{t-1} = 0.$$

The prediction error for the naive estimator is:

$$\begin{aligned} e_t^{(\text{naive})} &= y_t - \hat{y}_t^{(\text{naive})} \\ &= \big[f(t) + \epsilon_t\big] - \big[f(t-1) + \epsilon_{t-1}\big] \\ &= \underbrace{\big[f(t) - f(t-1)\big]}_{\approx 0} + \big[\epsilon_t - \epsilon_{t-1}\big]. \end{aligned}$$

Since $f(t) \approx f(t-1)$, we approximate

$$e_t^{(\text{naive})} \approx \epsilon_t - \epsilon_{t-1}.$$

Assuming ϵ_t and ϵ_{t-1} are independent, the expected Mean Squared Error (MSE) is:

$$\mathbb{E}\left[(e_t^{(\text{naive})})^2\right] = \mathbb{E}\left[(\epsilon_t - \epsilon_{t-1})^2\right] = 2\sigma^2.$$

3.2.2 Explicit Trend Estimator

Now, consider a model that attempts to predict the trend explicitly. Let its prediction be:

$$\hat{y}_t^{(\text{exp})} = f_\theta(t),$$

where $f_\theta(t)$ is the model's estimation of the true trend $f(t)$. Define the estimation error as

$$\Delta_t = f(t) - f_\theta(t).$$

Then the prediction error becomes:

$$\begin{aligned} e_t^{(\text{exp})} &= y_t - \hat{y}_t^{(\text{exp})} \\ &= \big[f(t) + \epsilon_t\big] - f_\theta(t) \\ &= \big[f(t) - f_\theta(t)\big] + \epsilon_t \\ &= \Delta_t + \epsilon_t. \end{aligned}$$

Also, the difference from the previous observation is:

$$\begin{aligned} \hat{y}_t^{(\text{exp})} - y_{t-1} &= f_\theta(t) - \big[f(t-1) + \epsilon_{t-1}\big] \\ &\approx \big[f(t) - \Delta_t\big] - f(t-1) - \epsilon_{t-1} \\ &\approx \big[f(t) - f(t-1)\big] - \Delta_t - \epsilon_{t-1} \\ &\approx -\Delta_t - \epsilon_{t-1}, \end{aligned}$$

since $f(t) \approx f(t-1)$.

The expected MSE for the explicit estimator is:

$$\begin{aligned}\mathbb{E}\left[(e_t^{(\text{exp})})^2\right] &= \mathbb{E}\left[(\Delta_t + \epsilon_t)^2\right] \\ &= \Delta_t^2 + 2\Delta_t\,\mathbb{E}[\epsilon_t] + \mathbb{E}\left[\epsilon_t^2\right] \\ &= \Delta_t^2 + \sigma^2,\end{aligned}$$

because $\mathbb{E}[\epsilon_t] = 0$ and $\mathbb{E}[\epsilon_t^2] = \sigma^2$.

3.2.3 Comparison of the Two Estimators

- **Naive Estimator:** Expected MSE is $2\sigma^2$.
- **Explicit Trend Estimator:** Expected MSE is $\Delta_t^2 + \sigma^2$.

The explicit estimator will perform better (i.e., yield a lower MSE) than the naive estimator only if:

$$\Delta_t^2 + \sigma^2 < 2\sigma^2 \quad \Longleftrightarrow \quad \Delta_t^2 < \sigma^2.$$

If the estimation error Δ_t^2 is not small—specifically, if $\Delta_t^2 \geq \sigma^2$—then the explicit estimator will incur a higher loss than the naive estimator.

Conclusion: Because $f(t)$ is slowly varying, simply copying y_{t-1} yields an error primarily due to the noise difference $\epsilon_t - \epsilon_{t-1}$ (with expected loss $2\sigma^2$). In contrast, any attempt to predict the trend explicitly introduces an additional error term Δ_t. Unless this estimation error is very small ($\Delta_t^2 < \sigma^2$), the explicit estimator will have a higher expected MSE, leading the model to favor short-term memorization rather than capturing the true long-term trend.

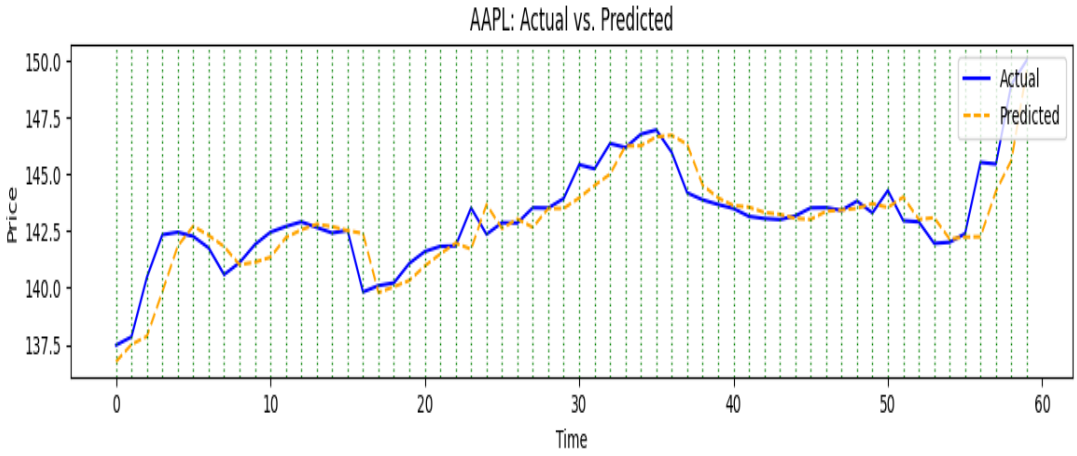


Figure 1: Illustration of the time-shift problem where the model predicts values close to the previous time step instead of capturing real trends.

Seq2Seq Architecture Due to the time-shift problem in traditional RNNs, where models tend to rely excessively on short-term memorization rather than capturing the true underlying trend, we introduce the Sequence-to-Sequence (Seq2Seq) Encoder-Decoder architecture. This model effectively addresses the issue by utilizing an encoder to capture long-term dependencies and a decoder to generate accurate future predictions. The following section details the structure and functioning of this architecture.

3.3 RNN Encoder-Decoder Architecture

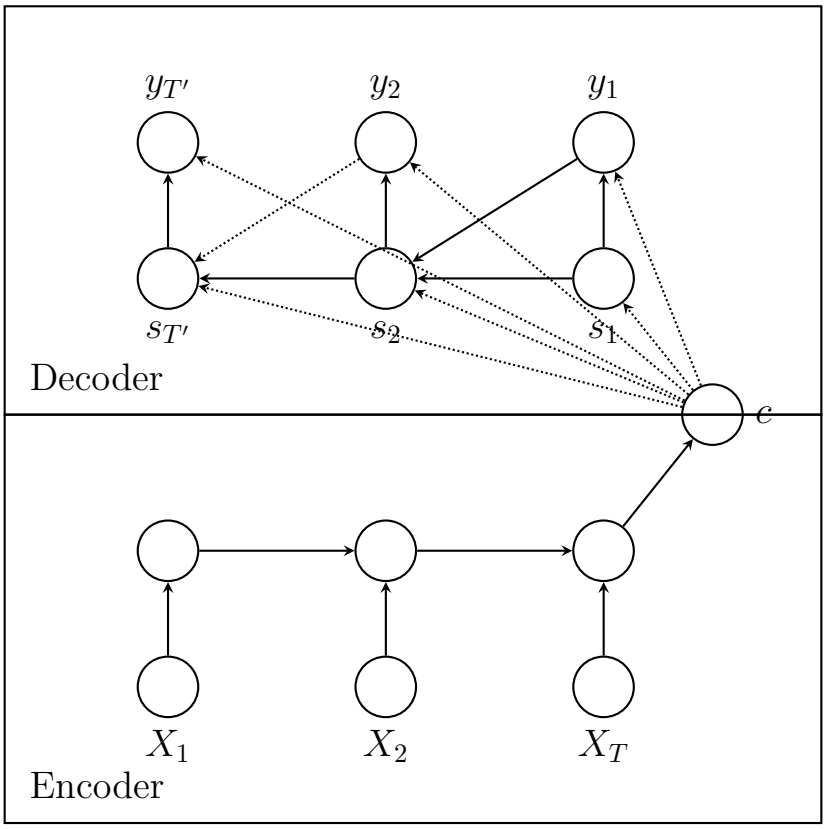


Figure 2: RNN Encoder-Decoder Architecture

Encoder-Decoder Architecture The RNN Encoder-Decoder architecture consists of an encoder that processes the input sequence $\{X_1, X_2, \ldots, X_T\}$ and compresses it into a context vector c, which captures the sequence's information. The decoder then uses c to generate the output sequence $\{y_1, y_2, \ldots, y_{T'}\}$, predicting each step based on the previous state.

Custom RNN Cell To improve the model's ability to capture long-term dependencies and optimize gradient flow, we incorporate a custom RNN cell. This cell enhances traditional RNNs by introducing Leaky ReLU activation and gated mechanisms, ensuring more stable training and better sequence learning.

3.4 Custom RNN Cell

The custom RNN cell is designed to enhance gradient flow during training by incorporating Reset and Update gates along with a Leaky ReLU activation function. In this architecture, the reset gate r_t selectively forgets portions of the previous hidden state h_{t-1}, while the update gate z_t balances the retention of past information with the integration of new input. The candidate hidden state $\tilde{h}_t$ is computed by modulating h_{t-1} with r_t and combining it with the current input x_t through a weighted transformation and a Leaky ReLU non-linearity. Finally, the new hidden state h_t is obtained by interpolating between the old hidden state and the candidate state according to the update gate z_t.

3.4.1 Equations

$$r_t = \sigma(W_r x_t + U_r h_{t-1}), \quad \text{(Reset Gate)} \tag{1}$$

$$z_t = \sigma(W_z x_t + U_z h_{t-1}), \quad \text{(Update Gate)} \tag{2}$$

$$\tilde{h}_t = \text{LeakyReLU}\Big(W x_t + U(r_t \odot h_{t-1})\Big), \quad \text{(Candidate Hidden State)} \tag{3}$$

$$h_t = z_t\, h_{t-1} + (1 - z_t)\, \tilde{h}_t, \quad \text{(Final Hidden State)} \tag{4}$$

3.5 Encoder Design

The encoder processes input stock price sequences and encodes temporal dependencies into a series of hidden states. The final hidden state is used as the context vector for the decoder.

$$\mathbf{H}, \mathbf{h}_T = \text{Encoder}(\mathbf{X})$$

- $\mathbf{X} \in \mathbb{R}^{T \times d}$: Input sequence with T time steps and d features per step.
- $\mathbf{H} \in \mathbb{R}^{T \times u}$: Hidden state sequence with u hidden units.
- $\mathbf{h}_T \in \mathbb{R}^u$: Final hidden state.

3.6 Decoder Design

The decoder generates future ***Close*** stock prices one step at a time using the context vector c from the encoder. It supports teacher forcing during training to stabilize the learning process.

3.6.1 Initialization

The initial hidden state is obtained by mapping the context vector c through a function ϕ:

$$s_0 = \phi(c)$$

where $\phi(\cdot)$ can be, for example, a linear transformation followed by a nonlinearity.

3.7 Teacher Forcing

We use teacher forcing to accelerate the convergence of the training process by feeding the actual target output from the training dataset as the next input to the decoder, rather than using the model's own predicted output. This helps the model learn faster by providing correct guidance at each step, reducing error propagation during training.

3.7.1 Decoder Recurrence

Let:

- c be the context vector from the encoder,
- h_t be the decoder's hidden state at time t,

- y_t be the true stock price at time t (used during training),
- $\hat{y}_t$ be the predicted stock price at time t,
- $f(\cdot)$ be the recurrent update function , and
- $g(\cdot)$ be the output function mapping the hidden state to a stock price prediction.

During Training (with Teacher Forcing): At each time step, the decoder uses the true previous stock price y_{t-1} as input:

$$h_t = f\Big(h_{t-1},\, y_{t-1},\, c\Big) \quad \text{for } t \geq 1,$$

$$y_t = g\big(s_t\big).$$

During Testing (without Teacher Forcing): At each time step, the decoder uses its own previous prediction $\hat{y}_{t-1}$ as input:

$$h_t = f\Big(h_{t-1},\, \hat{y}_{t-1},\, c\Big) \quad \text{for } t \geq 1,$$

$$\hat{y}_t = g\big(s_t\big).$$

The decoder thus produces an output sequence $\hat{Y} \in \mathbb{R}^{T'}$ given $c \in \mathbb{R}^u$:

$$\hat{Y} = \text{Decoder}(c)$$

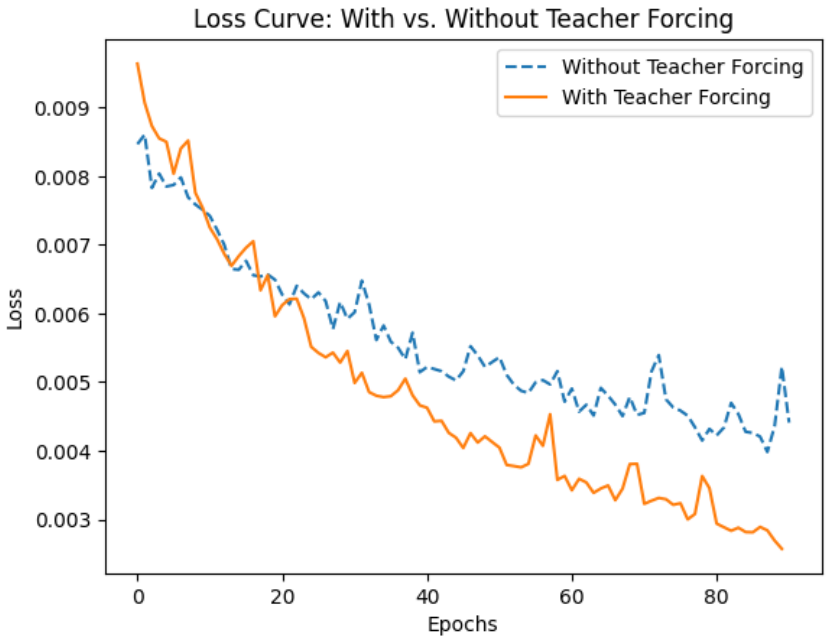


Figure 3: Loss Curve: With vs. Without Teacher Forcing

Loss with/without Teacher Forcing Figure 3 shows that training with teacher forcing (orange line) leads to faster convergence. Without teacher forcing, the loss decreases more slowly and fluctuates more, indicating less stable learning.

3.8 Experimental Setup

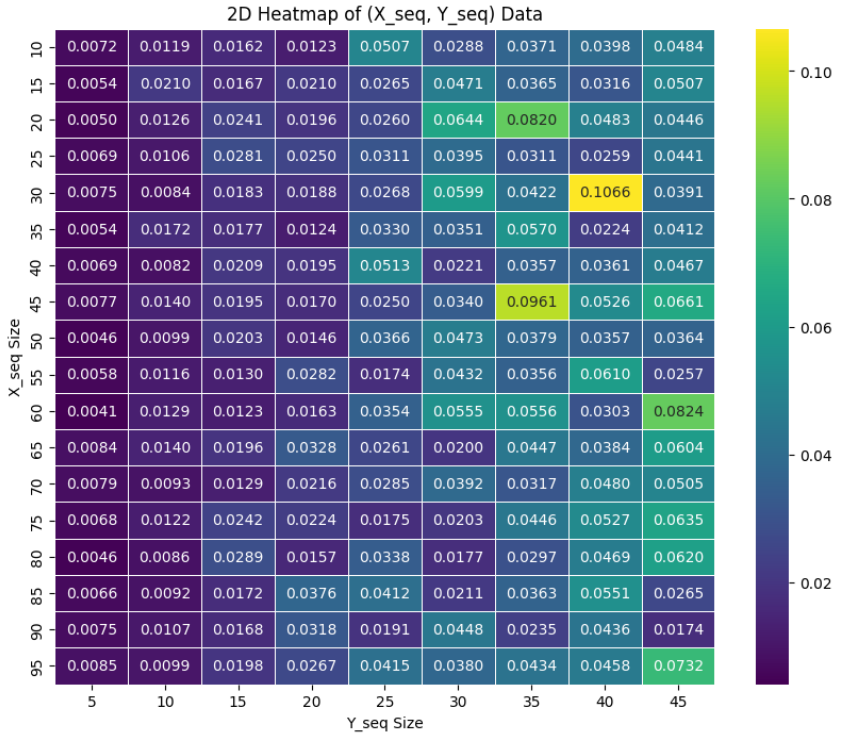


Figure 4: Heatmap of Prediction Loss Based on the Window Size of Past and Future Stock Prices

Optimal Window of Historical Data This heatmap visualizes the relationship between past (X_{seq}) and future (Y_{seq}) stock prices in terms of model loss. The X_{seq} axis represents the number of past stock prices used as input, while the Y_{seq} axis represents the number of future stock prices the model predicts. The color intensity indicates the loss, where lower values (darker colors) signify better predictive performance. The trend suggests that using a moderate number of past and future prices results in lower loss, while very short or very long sequences tend to increase error. This implies that there is an optimal window of historical data that maximizes predictive accuracy, beyond which additional data may introduce noise or overfitting.

3.9 Hyperparameter

- **Data Split:** 80% training and 20% testing.
- **Batch Size:** 32.
- **Loss Function:** Mean Squared Error (MSE).
- **Optimizer:** Adam.
- **Model Structure:**
  - Encoder: Encoder(batch_size, units) = Encoder(32, 64).
  - Decoder: Decoder(units, 1) = Decoder(64, 1).
- **Input/Output Sizes:**
  - Input sequence: $\mathbf{X}_{\text{seq}} \in \mathbb{R}^{32\times 20\times d}$.
  - Target sequence: $\mathbf{y}_{\text{seq}} \in \mathbb{R}^{32\times 10\times 1}$.

3.10 Training Progress Over Epochs

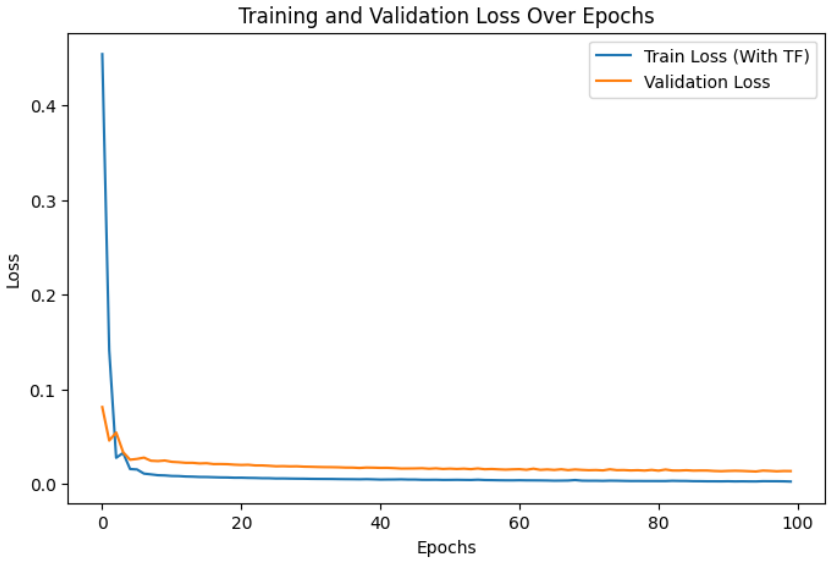
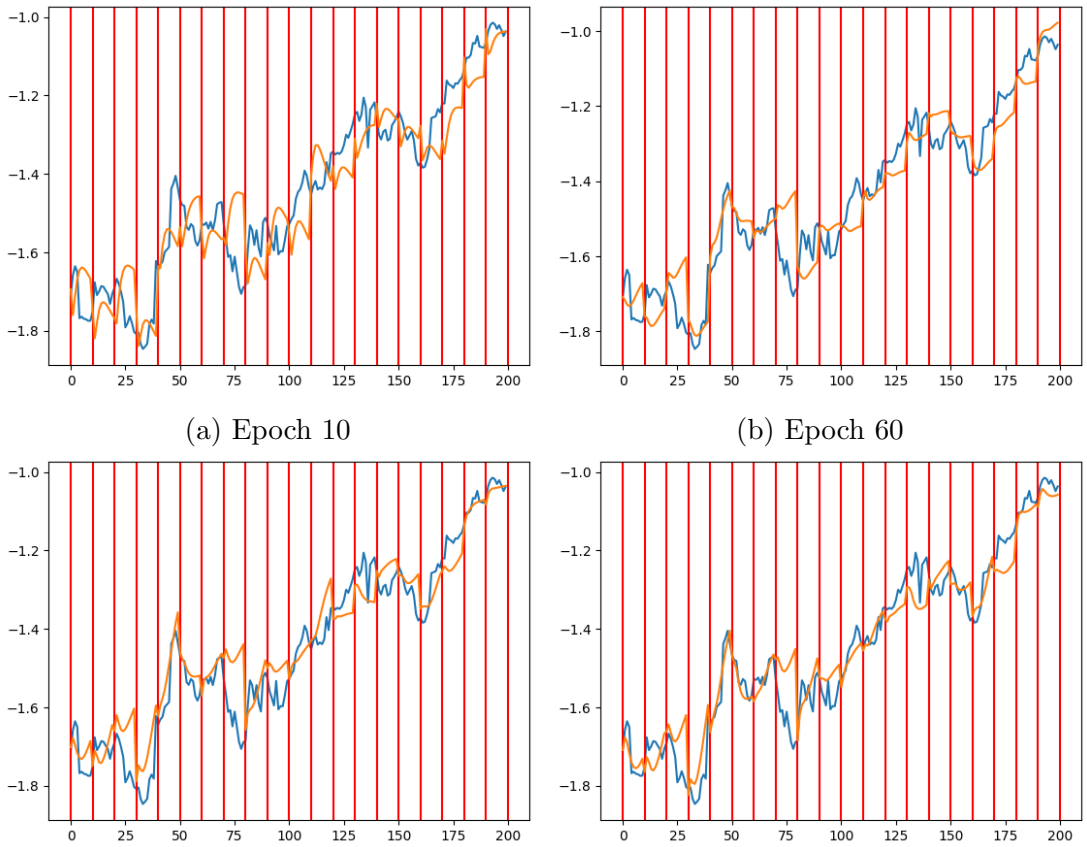


Figure 5: Training and Validation Loss Over Epochs



(a) Epoch 10

(b) Epoch 60

(c) Epoch 150

(d) Epoch 300

Figure 6: Training progress visualized at different epochs.

4 Price Prediction Using LSTM

In Section 3.1, we discuss the time shift problem in traditional RNN where the model has a naive tendency to predict the previous price as the next price. In this section, we explore to see if LSTM's are more capable and if they can provide meaningful insights to produce profits.

4.1 Initial Approach

The initial approach was to predict future prices based on historical data. The rationale was that if the LSTM model could learn price movements and consistently predict future prices or price direction with even accuracy 51%, it could make profits. LSTM's have an edge over traditional RNN's with their long-term memory. Considering this, we hypothesize that they should be able to learn long-term trends that traditional RNN's struggled with, allowing for meaningful forecasts on time-series data.

4.2 Data Preperation

1. **Data Collection:** Historical prices were collected using Yahoo finance.
2. **Scaling and Normalization:** The closing prices were normalized using MinMax scaling to ensure all values were on the same scale.
3. **Sequence Creation:** Since LSTMs rely on sequential patterns, the data was structured into rolling windows of past price movements (e.g., the last 10 days) to predict the next day's price. This would allow the model to recognize trends and dependencies over time.

$$\text{Input: } [P_0, P_1, P_2, P_3, P_4, ...P_n] \text{ where } n = \text{look-back days}$$

$$\text{Output: } [P_{n+1}]$$

4. **Data Split:** The data was split into training, testing, and validation sets with 80% of the data used for training, 10% used for validation, and 10% used for testing.

4.3 Model Architecture

The architecture used follows a structured design with two LSTM layers, dropout for regularization, and dense layers for final prediction.

Model: "sequential"

Layer (type)	Output Shape	Param #
lstm (LSTM)	(None, 10, 50)	10,400
dropout (Dropout)	(None, 10, 50)	0
lstm_1 (LSTM)	(None, 50)	20,200
dropout_1 (Dropout)	(None, 50)	0
dense (Dense)	(None, 25)	1,275
dense_1 (Dense)	(None, 1)	26

Total params: 31,901 (124.61 KB)
Trainable params: 31,901 (124.61 KB)
Non-trainable params: 0 (0.00 B)

Figure 7: Model architecture for a look-back window of 10 days

1. **Multiple LSTM layers:** Multiple LSTM layers were utilized to capture more complex relationships in the historical data. The first LSTM layer processes sequences and outputs a sequence to the next LSTM layer. The next layer processes the sequences and outputs abstracted data to the dense layer.

2. **Dense Layer:** The dense layer refines the information learnt by the LSTM layers into a single price prediction as the output.

3. **Dropout Regularization:** Dropout layers are applied to each LSTM layer as a regularization method to prevent overfitting. Each dropout layer deactivates 20% of the neurons during training.

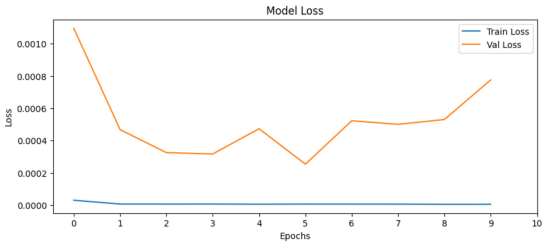
4. **Activation Function:** The tanh activation function was utilized as it is a standard practice to use it in LSTM's and RNN's.

5. **Loss Function:** The model minimizes the MSE (Mean Squared Error) which penalizes heavily on larger deviations compared to smaller ones. This ensures that the model would be able learn trends and price movements.

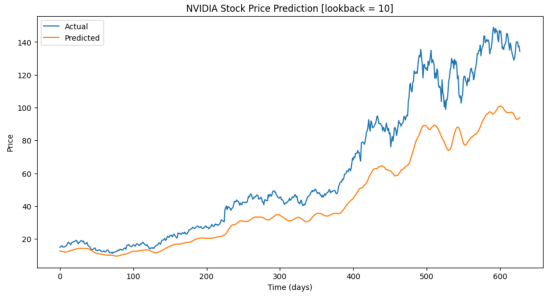
6. **Optimizer:** Adam was selected for adaptive learning rates leading to faster convergence and reducing training times.

4.4 Results

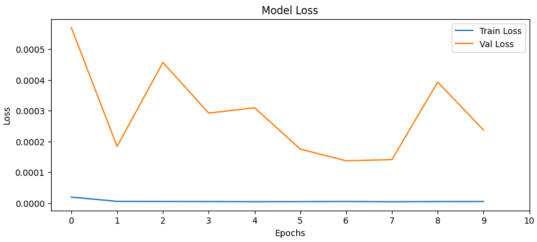
The model was trained on data with varying window sizes of 10, 30, and 100 to see the impact of providing more context for each prediction.



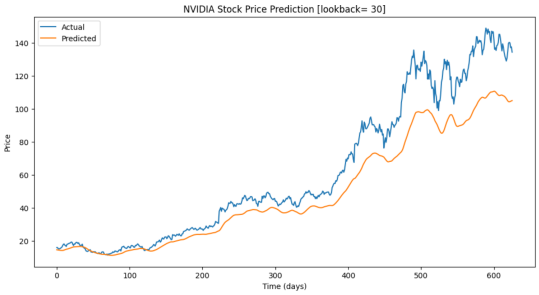
(a) Loss [Window size = 10]



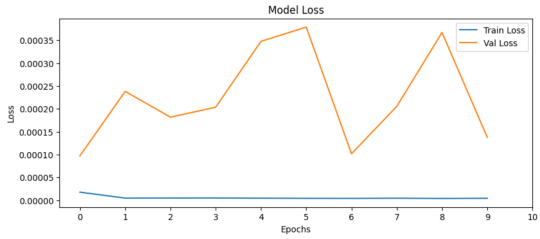
(b) Result [Window size = 10]



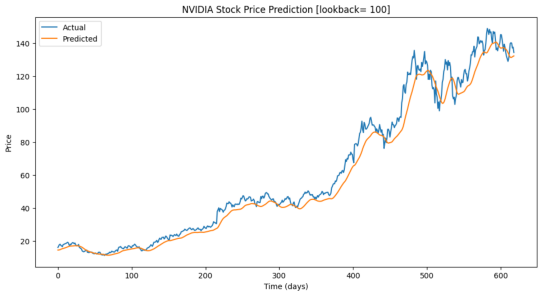
(c) Loss [Window size = 30]



(d) Result [Window size = 30]



(e) Loss [Window size = 100]



(f) Result [Window size = 100]

Figure 8: LSTM model results with different window sizes

Visualizing the loss reveals that the model has overfit. The validation loss is significantly higher than the training loss, indicating that the model knows how to provide accurate outputs on training data it has already seen but fails to do so on unseen data.

The predictions show that using a larger window size and providing more context makes the predictions closer to the actual data. However, this just happens to be a case for one run and there is no real correlation with window sizes and model performance. It seems to be completely randomized due to the stochastic nature of model training. For example figure 9 shows another run with a window size of 10 providing essentially the same results as the [window size = 100] output. A thing to notice however, was the increased time consumption as a result of needing more calculations with a larger window size. Having a window size of 10 took 2-3s per epoch, window size of 30 took 10s per epoch, and window size of 100 took 20s per epoch to train.

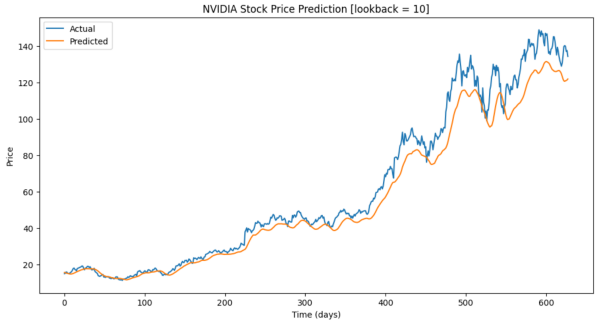
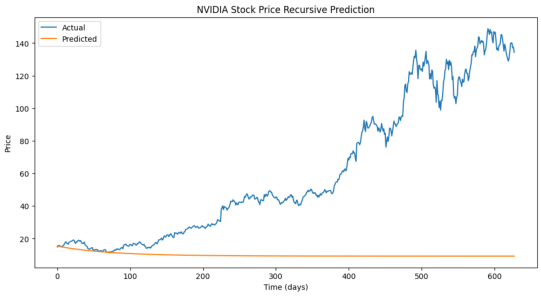
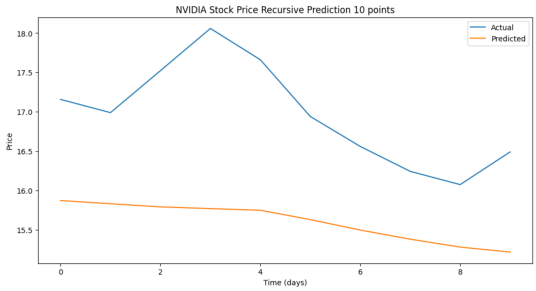


Figure 9: Result [Window size = 10] run 2

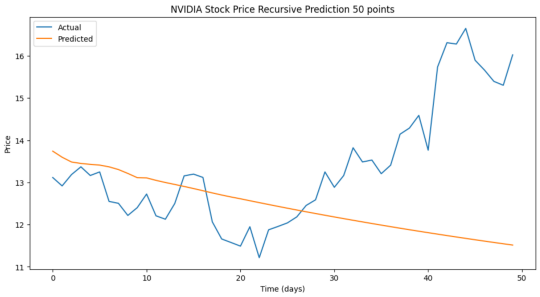
The results show that the LSTM was not able to learn from the provided dataset and faces the same time-shift problem that traditional RNN face. The model learnt the naive trick of predicting the last value as the next to minimize the MSE. In hindsight, this makes complete sense as the most reliable way to keep up with the market trend is to predict more or less the last value seen. If a human was tasked with predicting the price of the next day given only the historical prices, they would most likely assume that it would be a small deviation from the price today. Given historical data alone, they would not be able to predict spikes in price movement. To visualize how the model predicts, a recursive prediction was done. The model was fed one context window and was made to keep repeatedly predicting on its own outputs.



(a) Recursive prediction for all days



(b) Recursive prediction for 10 days



(c) Recursive prediction for 50 days

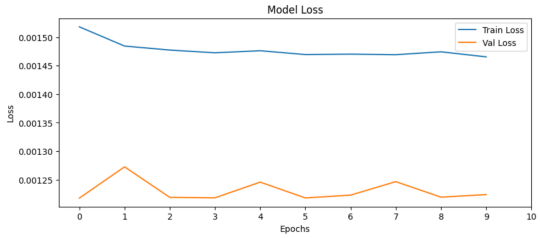
Figure 10: Recursive prediction results for different days

It is now more clear to see exactly what the model is doing to predict future prices. In this instance, it takes the current price and always predicts the next price as a slight decrease from the current price which is highlighted by the downward linear trend of the prediction. This made it clear that predicting price was not the right approach.

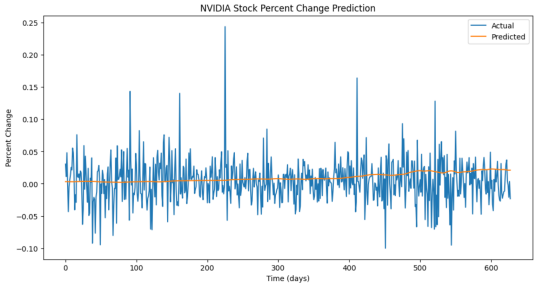
The next approach was trying to predict the price change instead. Instead of predicting the exact price, if the model could predict the price increase or decrease, it could provide valuable insights to make profits. To do this, the model was fed a window of past returns and was asked to predict the future returns. Positive returns indicate if the price increased and similarly negative show price decrease. Two different types of returns were used, percent returns, and log returns.

$$\text{log\_return} = \ln(\frac{P_t}{P_{t-1}})$$

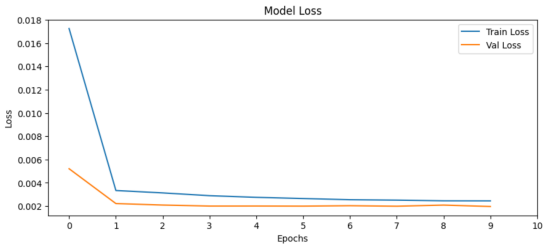
$$\text{percent\_return} = \frac{P_t - P_{t-1}}{P_{t-1}} \times 100$$



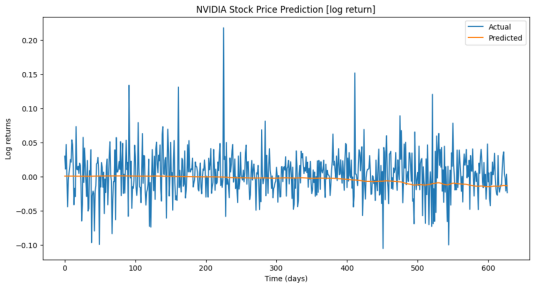
(a) Percent return loss



(b) Percent return result



(c) Log return loss



(d) Log return result

Figure 11: Loss and result plots for percent and log returns

In both cases, the model shows to have difficulty predicting the results. This is mainly due to extreme volatility in the price changes and the heavy noise in the data model is trying to learn off of. Unfortunately, this is the reality of the market and precisely why it is so difficult to predict it.

4.5 Conclusions

The results show that using only historical data and time series analysis is not sufficient enough to make practical market predictions. While these models can capture certain patterns in price movements, they struggle to account for the unpredictable nature of the market. For the next time, incorporating statistical and economic principles directly into the loss function of the model similar to a physics informed model would be interesting to explore. A main reason for the timeshift problem was the MSE loss causing overfitting. Incorporating traditional economic principles into the loss would make it more generalized

and may provide better results. Another thing which could be done differently is trying multimodal models which would incorporate various different factors to provide a more practical prediction.

5 CNN Model for Stock Movement Prediction

This section describes the 1D Convolutional Neural Network (CNN) designed to predict the probability of a stock price increase or decrease for each of the next 10 days.

5.1 Binary Labeling within the Model

For each sample created using a rolling window of 10 days, a binary label is assigned to indicate the movement of the stock price:

$$y_i = \begin{cases} 1, & \text{if } \text{Close}_{i+10} > \text{Close}_i, \\ 0, & \text{otherwise.} \end{cases}$$

This labeling scheme is integral to training the model for binary classification over each day in the prediction horizon.

5.2 Model Architecture

The CNN model architecture is designed as follows:

- **Input Layer:** Accepts time-series data in the format $(\text{sequence\_length}, \text{features})$, where each input represents 10 consecutive days of stock data.
- **Conv1D Layers:**
  - **First Conv1D Layer:** Applies 32 filters with a kernel size of 3 and uses LeakyReLU activation to learn local price patterns.
  - **MaxPooling Layer:** Uses a pool size of 3 to reduce the spatial dimensions of the feature maps.
  - **Second Conv1D Layer:** Utilizes 64 filters with a kernel size of 3 and LeakyReLU activation to extract deeper patterns.
- **Flatten Layer:** Flattens the multi-dimensional output from the convolutional layers into a 1D vector.
- **Dense Layers:**
  - A dense layer with 128 neurons and LeakyReLU activation is used to enhance the feature representation.
  - The **Output Layer** consists of a single neuron with Sigmoid activation, predicting the probability of a stock price increase.
- **Loss Function and Optimizer:**
  - The Binary Cross-Entropy loss function is used since the outputs represent probabilities between 0 and 1.
  - The Adam optimizer is employed to dynamically adjust the learning rate, ensuring faster convergence.
- **Training Process:** The model is trained for 30 epochs with a batch size of 64. Validation data is utilized during training to fine-tune the model parameters, and accuracy is used as the evaluation metric.

5.3 Model Architecture Diagram

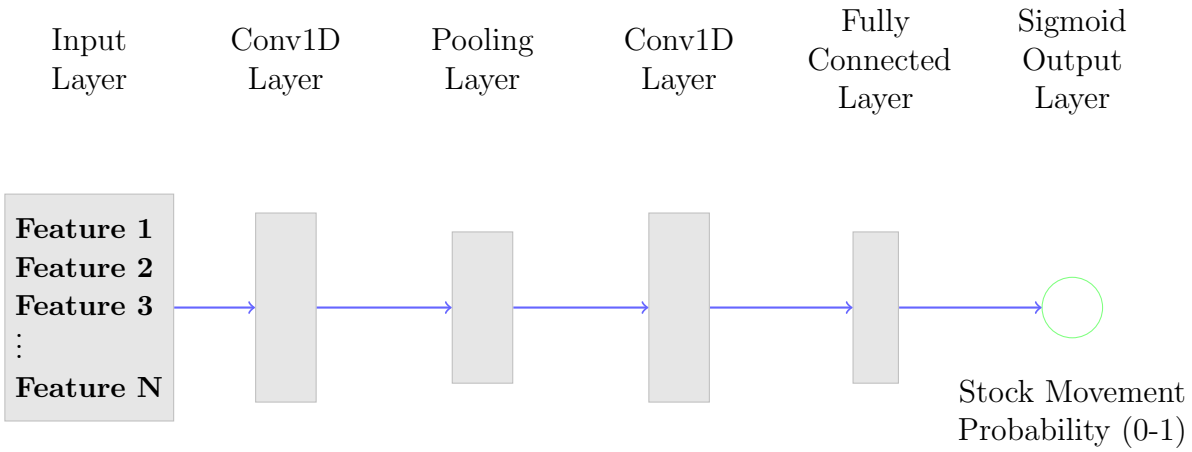


Figure 12: The architecture of our CNN model

5.4 Hyperparameter Tuning

To improve model performance, we experimented with different hyperparameter settings.

Trial	Filters (Conv1D)	Kernel Size	Dropout Rate	Epochs	Batch Size	**Accuracy**
1	128, 256	5	0.4	40	32	0.69
2	128, 256	3	0.4	40	64	0.75
3	64, 128	5	0.3	30	32	0.76
4	64, 128	3	0.3	30	64	0.78
5	32, 64	3	0.2	30	64	0.81

Table 1: Hyperparameter Tuning Results

The best accuracy (81.0%) was achieved with **32, 64 filters, a kernel size of 3, and a dropout rate of 0.2**.

5.5 Results

The performance of the CNN model is summarized by the following classification report:

	precision	recall	f1-score	support
0	0.77	0.71	0.74	383
1	0.83	0.87	0.85	617
accuracy			0.81	1000
macro avg	0.80	0.79	0.79	1000
weighted avg	0.80	0.81	0.80	1000

To better understand the impact of dropout regularization, we compare the training vs. test loss before and after adding dropout. Initially, the model exhibited overfitting, as shown in below, where the test loss increased despite a decreasing training loss.

Figure 13: CNN Training & Test Loss Before Dropout

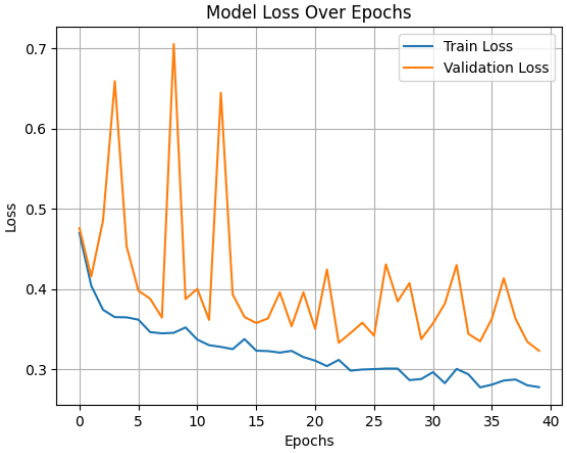


Figure 14: CNN Training & Test Loss After Dropout

5.6 Conclusion

In this study, we implemented a 1D CNN model for stock movement prediction, testing different feature combinations, hyperparameter tuning, and regularization techniques to optimize model performance.Our initial plan focused on using a 1D CNN model to capture temporal stock price patterns and predict stock movement. As we progressed, we conducted feature selection experiments, hyperparameter tuning, and regularization techniques to refine the model. One key finding was that Relative Strength Index (RSI) played a crucial role in improving accuracy. By systematically removing and adding different features, we discovered that including RSI significantly enhanced model performance. Additionally, hyperparameter tuning helped optimize the model's performance. We experimented with different filter sizes, kernel sizes, epochs, and batch sizes to identify the best configuration. Initially, the training loss decreased while test loss increased, indicating overfitting. To mitigate this, we added a dropout rate of 0.2, which improved generalization and reduced the risk of overfitting. The comparison of training vs. test loss before and after dropout confirms this improvement.Through these experiments, we now have a better understanding of how feature selection, model architecture, and hyperparameters impact CNN-based stock movement prediction.

6 BERT for News Sentiment Generation

6.1 Introduction

BERT (Bidirectional Encoder Representations from Transformers) is a transformer-based language model developed by Google that has revolutionized natural language processing. In the attempts to fine-tune this model, the 'BERT-base-uncased' model was selected from Hugging Face. This model is a neural network architecture consisting of 12 transformer encoder layers, with 12 attention heads in each layer, and a hidden size of 768 dimensions. It contains approximately 110 million parameters. The "uncased" designation means that the text input is converted to lowercase before processing, treating "Word" and "word" as the same token. The model's architecture can be broken down into several key components:

Input Embedding Layer: The model accepts tokenized text input and converts it into initial embeddings. It combines three types of embeddings:

- Token embeddings: Represent the meaning of each token in the vocabulary
- Position embeddings: Encode the position of each token in the sequence
- Segment embeddings: Distinguish between pairs of sentences when two sentences are input together

Transformer Encoder Blocks: Each of the 12 encoder layers contains:

- Multi-head self-attention mechanism: Allows the model to focus on different parts of the input sequence simultaneously
- Feed-forward neural networks: Process the attention outputs through two linear transformations with a GELU activation function
- Layer normalization: Applied after each sub-layer
- Residual connections: Help prevent vanishing gradients and maintain information flow

Pre-training Objectives: The model is pre-trained using two main tasks:

1. Masked Language Modeling (MLM): Random tokens in the input are masked, and the model must predict them
2. Next Sentence Prediction (NSP): The model predicts whether two sentences follow each other in the original text

Key Specifications:

- Maximum sequence length: 512 tokens
- Vocabulary size: 30,522 tokens
- Hidden layer size: 768
- Intermediate size in feed-forward layers: 3072
- Attention heads: 12
- Transformer blocks: 12

This architecture enables BERT-base-uncased to capture complex linguistic patterns and relationships in text, making it highly effective for various downstream NLP tasks such as classification, question answering, and named entity recognition. Additionally, the model's bidirectional nature, where it considers context from both directions simultaneously, gives it a significant advantage over previous unidirectional models, allowing for richer language understanding and representation.

6.2 Approach

This model was fine-tuned using the financial_phrasebank dataset, which is a dataset of 4840 sentences annotated by 16 researchers and Master's Graduate Students from the Aalto University School of Business. The 100% agreement corpus was chosen as a subset of this dataset where all annotators agreed on the label (i.e., positive, neutral, negative) of a given sentence. This dataset is the same dataset that was used to fine-tune a high-performing model: 'DistilRoBERTa Fine-Tuned for Financial News Sentiment Analysis,' which is a modified version of BERT that achieves an accuracy of approximately 98% on validation data. With many regarding this particular dataset as one of the gold-standard datasets for financial sentiment analysis fine-tuning, the results of the fine-tuned model were also compared to 'DeBERTa V3 Fine-Tuned for Financial News Sentiment Analysis,' which is a variant of Microsoft's DeBERTa V3 small model that has a top accuracy of 99.9% on validation data.

After developing a fine-tuned BERT model, we filtered a Kaggle Dataset: 'Daily Financial News for 6000+ Stocks' for entries related to Nvidia. We performed inference on the headlines of all such entries (n=3146), calculating an average score per day and exporting the average sentiment scores as a time series. We then compared the average sentiment scores with the daily closing prices of Nvidia's stock at different timeframes to assess if there was any noticeable correlation between news sentiment and stock price.

6.3 Exploratory Data Analysis

Exploration of the fine-tuning data was limited due to the nature of its categorical contents. In total, 2259 unique phrases and annotations were found, with a total of 5 duplicates. No missing entries were found. The most notable feature of the dataset was the class imbalance, with 61.4% of labels marked as 'neutral', 25.2% of labels marked as 'positive', and 13.4% of labels marked as 'negative'. As the data imputation and over-sampling/undersampling was not performed, the F1-score was considered a more robust metric over accuracy in downstream analysis due to the nature of the class imbalances in this dataset.

6.4 Training and Hyperparameter Tuning

The data was initially split into 80% training and 20% validation. The order of the data was randomized, and a sparse categorical cross entropy loss function was chosen based on the desired sentiments the model predicted. The Adam optimizer was chosen with an initial learning rate of $2\text{x}10^{-3}$, but later optimized to $2\text{x}10^{-5}$ (Fig. 25) based on validation accuracy after 3 epochs. Early stopping as a mode of regularization was also implemented with a patience of 2, and with a minimum delta of 0.001. More epochs were implemented later (Fig. 26) demonstrating that training accuracy increased, while validation accuracy

changes were negligible. This is an indication that model is overfitting to the training data.

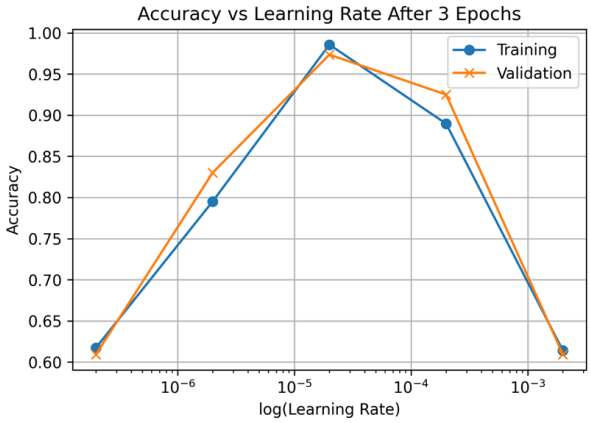


Figure 15: Optimizing Adam learning rate

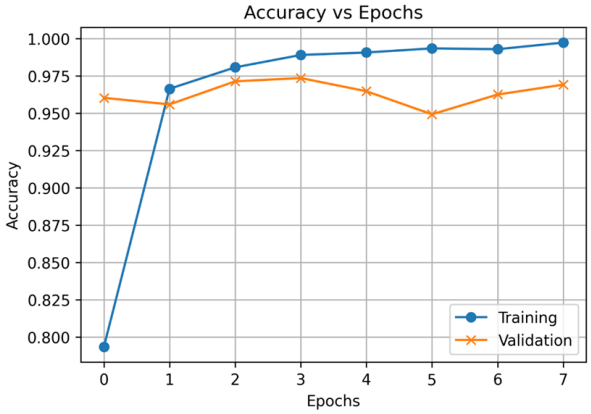
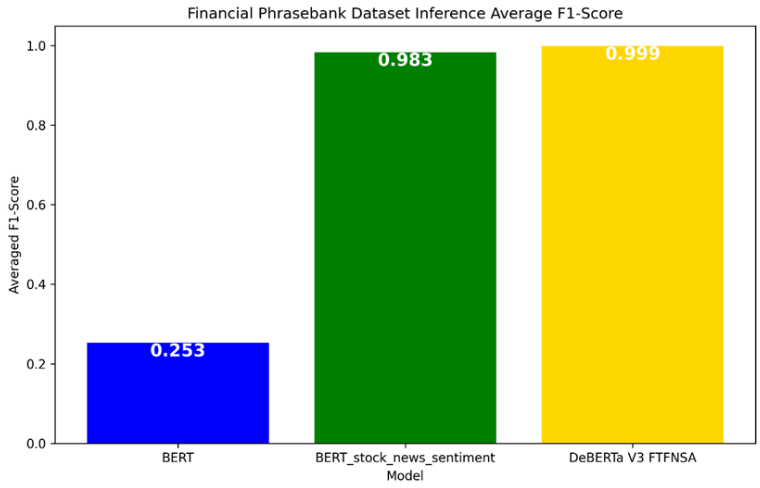


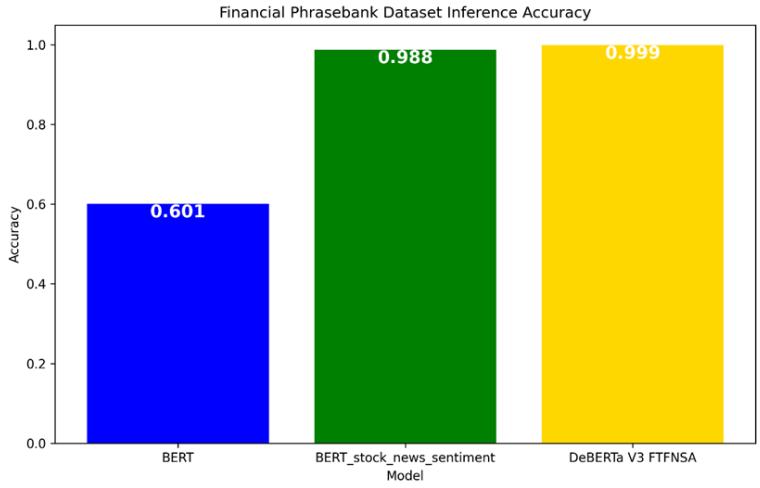
Figure 16: Comparison of epoch accuracy between training and validation data

6.5 Performance

The baseline BERT model achieved the worst inference, as expected, with an averaged F1-score of 0.253 and an accuracy of 0.601 on the validation data (Fig. 27). The fine-tuned BERT model achieved noticeably better performance, with an averaged F1-score of 0.983, and an accuracy of 0.988. In comparison, the gold standard model DeBERTa-V3-FTFNSA achieved a near-perfect F1-score and accuracy.



(a)

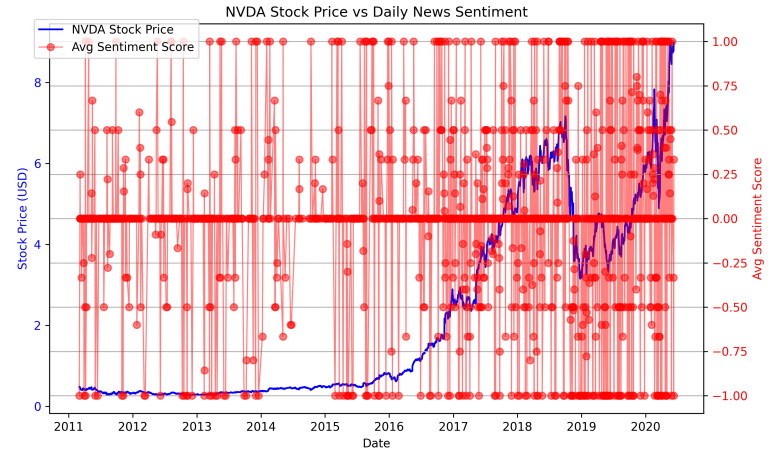


(b)

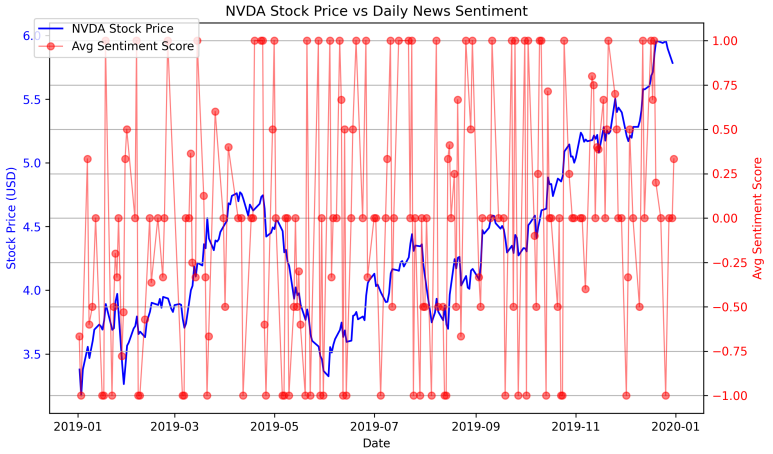
Figure 17: Inference Scores for BERT, fine-tuned BERT, and DeBerTa Models. a) F1-Score, b) Accuracy

6.6 Stock Price vs Sentiment Comparison

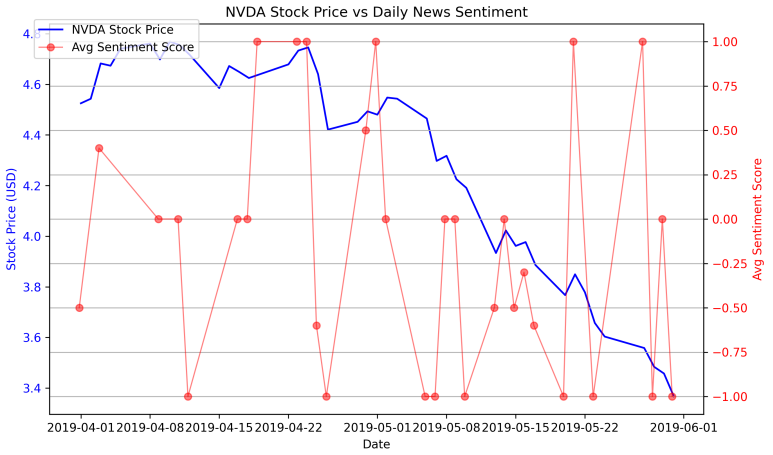
We performed a qualitative analysis of NVDA's stock price with respect to news sentiment scores. Observing the entire span of NVDA's news data from 2011 to 2020, the timeframe was too large to make conclusions. Zooming into the year 2019, the relationship between stock price and average news sentiment is unclear, as certain time periods (Fig. 28) appear to be highly correlated, whereas other periods seem to be random. Thus, no definitive conclusions are made on the nature this relationship but future work will aim to perform a quantitative analysis based on metrics such as Pearson correlation scores.



(a)



(b)



(c)

Figure 18: Comparison of Daily NVDA Stock Price and Averaged News Sentiment Scores. Red lines indicate sentiment, blue lines indicate stock price. a) 2011-2020 time-frame. b) 2019 time-frame. c) April 2019 to June 2019 time-frame.

7 Future Work

In our future work, we will explore Transformer architectures to improve long-range dependencies and contextual understanding. To enhance predictive performance, we will incorporate multimodal learning into our models, combining stock price data with financial news, as well as other financial metrics such as PE ratios. This integration will allow the model to learn both historical price patterns and external market sentiment, leading to more comprehensive and robust forecasting. In addition to news data, data from social media could also be analyzed to analyze the public sentiment regarding the stock for a more robust approach. As our goal is to maximize profits, a different approach with reinforcement learning could also be taken. Implementing reinforcement learning to optimize trading strategies could help to not only predict stock prices but also suggest buy or sell actions based on historical price data and other features. By training the model to maximize rewards (profit) and reducing losses, the model can learn optimal trading strategies instead.

8 References

- Cho, Kyunghyun, et al. "Learning phrase representations using RNN encoder-decoder for statistical machine translation." *arXiv preprint arXiv:1406.1078* (2014).
- Chen, Sheng, and Hongxiang He. "Stock prediction using convolutional neural network." *IOP Conference Series: Materials Science and Engineering*, vol. 435, no. 1, 2018, p. 012026.
- Pekka, Malo, et al. "Good debt or bad debt: Detecting semantic orientations in economic texts." *Journal of the American Society for Information Science and Technology* (2013).