Università di Pisa

Computer Engineering

Intelligent Systems

Stock returns prediction using RNN and LSTM neural networks

Group Project Report

TEAM MEMBERS:
Federico Casu
Daniel Deiana

Academic Year: 2022/2023

Contents

Chapter 1

Introduction

Forecasting stock market behavior is a challenging and highly sought-after task for investors, traders, and researchers in the field of finance. Accurately predicting future stock returns is crucial for making informed investment decisions and maximizing profits. In recent years, recurrent neural networks (RNNs) have emerged as a powerful tool for addressing forecasting problems, thanks to their ability to capture sequential relationships and temporal dependencies present in financial data.

This project aims to explore the application of recurrent neural networks in forecasting stock returns. Recurrent neural networks are a type of artificial neural network that incorporates a memory element, allowing them to process data sequences and model temporal dynamics. This characteristic makes them particularly well-suited for tackling financial forecasting problems, where historical data is organized in time series.

The main objective of this study is to develop a predictive model based on recurrent neural networks to forecast stock returns. The model will be trained with historical stock data, such as closing prices, trading volumes, and other relevant variables. The main goal is to train the model in such a way that it will be able to recognize patterns and relationships that influence future returns.

Various variants of recurrent neural networks will be explored, including simple recurrent neural networks (RNNs), long short-term memory (LSTM) networks. Different architectures and configuration parameters will also be analyzed to achieve the best predictive performance.

The results obtained from this study could provide valuable insights for investors and traders, aiding them in making more informed decisions in the stock market. Additionally, the work may contribute to the existing literature on financial forecasting, demonstrating the effectiveness of recurrent neural networks in tackling the complex problem of predicting stock returns.

Chapter 2

Related work

Over the past few decades, the stock market has experienced significant growth, leading to an increase in the number of researchers focusing on stock price forecasting. Their aim is to analyze and predict fluctuations and price changes within the market. However, stock prices exhibit characteristics of high dynamics, non-linearity, and significant noise. Various factors, including the global economy, politics, government policies, natural or man-made disasters, and investors' behavior, can influence the price of individual stocks. Consequently, forecasting stock prices has become one of the most challenging tasks in the realm of time series forecasting problems. The following is a list of papers that inspired and guided us through the analysis we made.

Gao, Chai & Liu (2017) collected the historical trading data of the Standard & Poor's 500 (S&P 500) from the stock market in the past 20 days as input variables, they were opening price, closing price, highest price, lowest price, adjusted price and transaction volume. They used LSTM neural network as the prediction model, and then evaluated the performance by Mean Absolute Error (MAE), Root Mean Square Error (RMSE), Mean Absolute Error Rate (MSE), and Mean Absolute Percentage Error (MAPE).

Lee & Soo (2017) combined LSTM neural network and CNN to predict the prices of Taiwanese stocks based on historical stock prices and financial news analysis, and found that it could reduce the error of prediction.

Li, Bu & Wu (2017) used the LSTM neural network to predict the ups and downs of the China Securities Index 300 (CSI 300) by inputting the opening price, closing price and trading volume of the past ten days.

Khare et al. (2017) also found that LSTM neural network can successfully predict the ups and downs of stock prices.

Chapter 3

Collecting training data

Training data was obtained by using `yfinance`.

Yfinance is a powerful Python library that facilitates the collection of financial data for time series analysis. It provides a convenient interface to access the Yahoo Finance data service. By importing yfinance into a Python environment, one can specify the desired time range and financial instruments of interest. Yfinance simplifies the process of collecting historical financial data, enabling users to explore and analyze time series in a straightforward and effective manner.

The structure of the dataset extracted using yfinance api is the following for each stock we examined:

- **Date**: This field indicates the date to which the data point in the dataset corresponds. It is typically expressed in the format "yyyy-mm-dd" or a similar format.
- **Open**: It represents the opening price of the stock for the specific day. It indicates the initial trading value for the stock.
- **High** It indicates the highest price reached by the stock during the trading day. This field reflects the highest price point touched by the stock during the day.
- **Low**: It represents the lowest price reached by the stock during the trading day. It indicates the lowest price point touched by the stock during the day.
- **Close**: It indicates the closing price of the stock for the specific day. This field represents the final trading value for the stock at the end of the trading day.
- **Volume** It represents the total number of shares traded during the specific day. This field reflects the amount of trading activity that occurred for the stock.
- **Adj Close (Adjusted Close)**: This field represents the adjusted closing price of the stock for the specific day. It takes into account any corporate actions, such as stock splits or dividends, that may affect the stock's price.

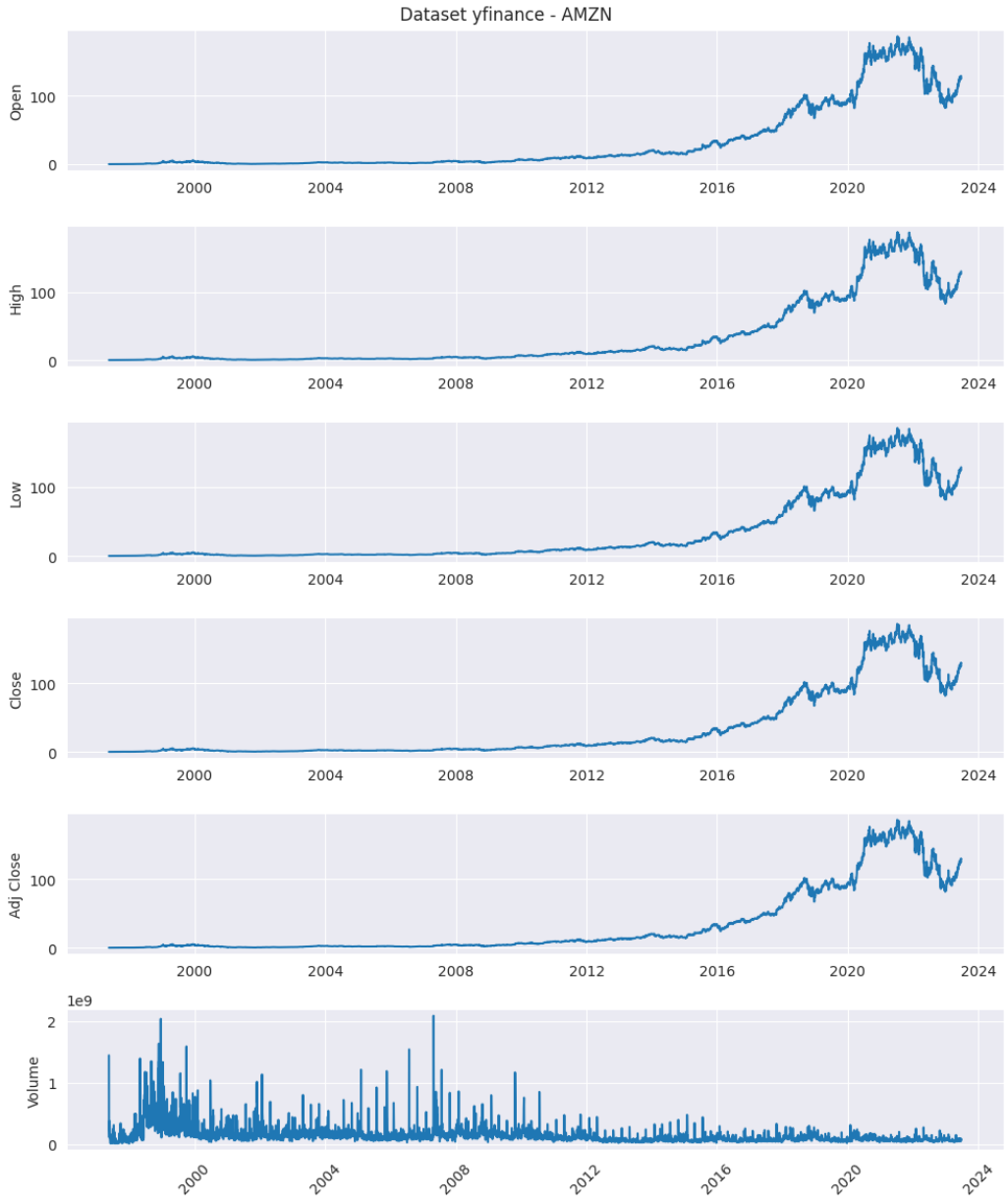


Figure 3.1: Starting dataset.

Chapter 4

Data preprocessing

The pre processing steps taken are the followings:

- **Reducing the time series to a more recent interval** Because of the fact that the initial temporal span of the series was from 1980 to current date we decided to work only from data coming by recent years (2018 and on ..) to obtain only the data that was more impactful for the forecasting.
- **Data Scaling** We used z standardization to allow us to compare and interpret data distributions based on their relative position to the mean. Additionally, it can be advantageous in machine learning algorithms that require normalized data or are sensitive to data scales.

Chapter 5

ARIMA

An autoregressive integrated moving average, or ARIMA, is a statistical analysis model that uses time series data to either better understand the data set or to predict future trends. A statistical model is autoregressive if it predicts future values based on past values. For example, an ARIMA model might seek to predict a stock's future prices based on its past performance.

An ARIMA model can be better understood by outlining each of its components as follows:

- Autoregression (AR). This component focuses on the relationship between a variable and its own past values. The model considers how the variable's behavior changes over time.
- Integrated (I). The integrated component deals with differencing the raw observations in the time series. Differencing is the process of computing the difference between consecutive data points. It is done to transform a non-stationary time series into a stationary one. By differencing, the ARIMA model removes trends or seasonality present in the data, making it easier to model.
- Moving average (MA). The moving average component captures the relationship between an observation and the residual errors from a moving average model applied to lagged observations. A moving average model predicts the current value of a variable based on the weighted average of its past residual errors. The model considers the dependency between an observation and the errors made in the past predictions.

Why we used ARIMA? Since the scope of the project is to test if the deep learning models can be useful in the task of predicting stock market trends, ARIMA is referred as one of the most used methods to forecast time series data in the literature. ARIMA models are also simple to construct only requiring 3 parameters, so we used that as a baseline to confront the neural network models forecasting with.

Parameters and performance of ARIMA model Parameters selection was done by trial and error selecting the best triple of parameters we found. We recall that there is no need to stationarize the data because we are giving as input the returns [1]. We considered as a performance index the **RMSE** and **MAE**. The forecast was done splitting the data in the same way we do for deep learning approach (so 90% for training and 10% for testing the model). The model is defined by $(p, d, q) = (0, 1, 0)$. The results we obtained

[1] Returns are computed differencing input data

are $RMSE = 0.8521$, $MAE = 0.7567$ and they will be used as a baseline for our next experiments.

Chapter 6

RNN vs LSTM univariate approach

6.1 Starting point: predict next day's close price

Our first idea was to predict, directly, the close price for both Apple and Amazon stocks. Before we started building the model, we decided to apply **min-max scaling** to the close price feature. When working with RNN and LSTM models, it is common to apply min-max scaling to the input features.

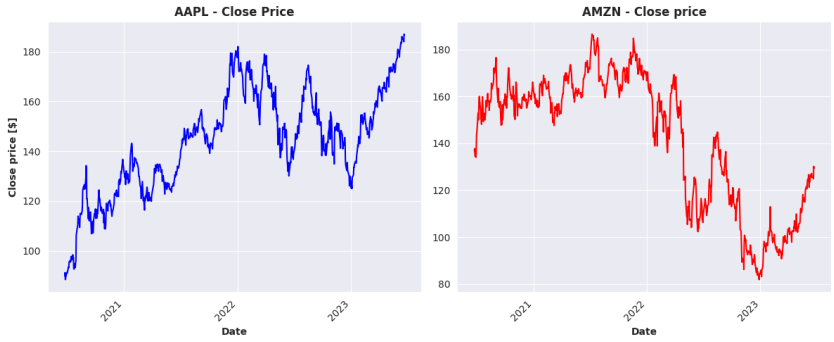


Figure 6.1: Apple and Amazon close prices [2020-06-24, 2023-06-24].

Figure 6.2: Apple and Amazon scaled close prices [2020-06-24, 2023-06-24].

After scaling the input data, we splitted the dataset in 3 subsets:

1. `train_data`, 70% of the entire dataset.
2. `val_data`, 20% of the entire dataset.
3. `test_data`, 10% of the entire dataset.

Since our objective was to predict the next day's close price based on what happened in the last week, `train_data`, `val_data` and `test_data` have the following shape:

- 2 columns:
  1. `Close` prices of the last 7 days.
  2. `Close` price of tomorrow.

The splitting operation was perfomed using the `create_dataset` function (Figure 6.3).

```
import numpy as np

def create_dataset(dataset, lookback):
    X_set = []
    Y_set = []

    for i in range(lookback, len(dataset)):
        X_set.append(dataset[i-lookback:i])
        Y_set.append(dataset[i])

    return np.array(X_set), np.array(Y_set)
```

Figure 6.3: `create_dataset()`.

6.1.1 Architecture

In the first attempt we build a very simple LSTM based network (Figure 6.4):

- One `LSTM` layer with 8 `units`.
- The `LSTM` layer takes 7 input and has 1 output.
- One `Dense` layer with just 1 output (the latter is also the model's output).

```
from keras.layers import LSTM
from keras.layers import Dense
from keras.optimizers import Adam
from keras.models import Sequential


lookback = 7

################### AAPL ###################
AAPL_regressor = Sequential()

AAPL_regressor.add(LSTM(units=8,
                   input_shape=(lookback, 1)))

AAPL_regressor.add(Dense(units=1))
############################################

################### AMZN ###################
AMZN_regressor = Sequential()

AMZN_regressor.add(LSTM(units=8,
                   input_shape=(lookback, 1)))

AMZN_regressor.add(Dense(units=1))
############################################
```

Figure 6.4: LSTM based network architecture.

6.1.2 Training and evaluating the model

During the training process, we used **Mean Squared Error** as loss function and, as optimization algorithm, we used an extension of Stochastic Gradient Descent (SGD): **Adam** optimizer.

To monitor the performance achieved by the model we used **Symmetric Mean Absolute Percentage Error** (SMAPE). SMAPE addresses the lack of symmetry in MAPE (Mean Absolute Percentage Error) by taking the average of the absolute percentage differences between the predicted and actual values, considering both positive and negative errors. We decide to use SMAPE instead of MAPE because SMAPE treats overestimations and underestimations equally.

The starting hyperparameters were the following:

- 1 `LSTM` layer.
- 1 `Dense` layer.
- 8 `units` for the `LSTM` layer.
- 1 `unit` for the `Dense` layer.
- `batch_size` = 16
- `learning_rate` = 0.01
- `epochs` = 100

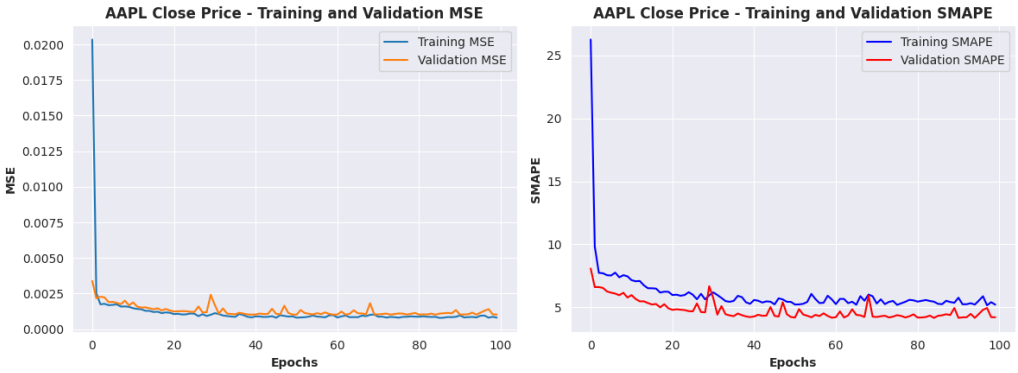


Figure 6.5: `LSTM` - Apple - Training process.

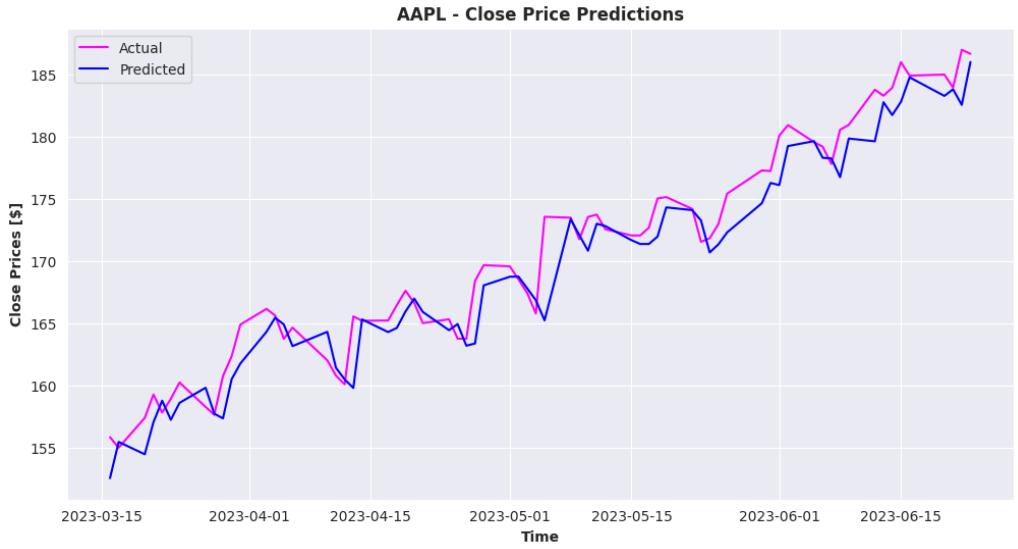


Figure 6.6: `LSTM` - Apple - Predictions.

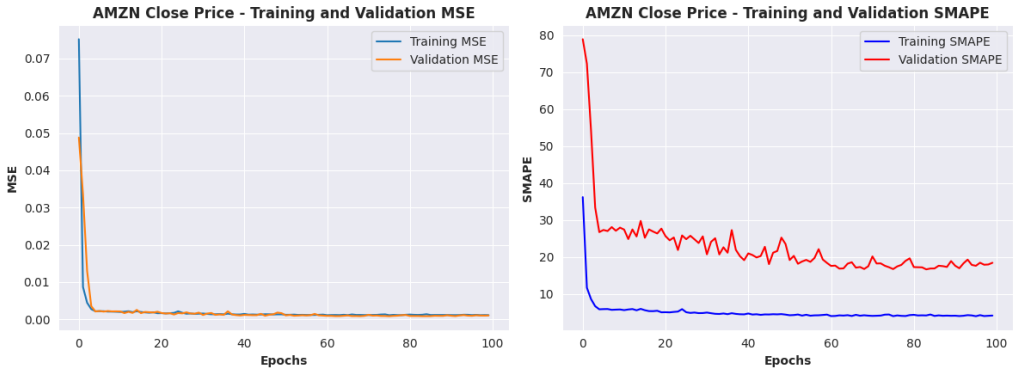


Figure 6.7: `LSTM` - Amazon - Training process.

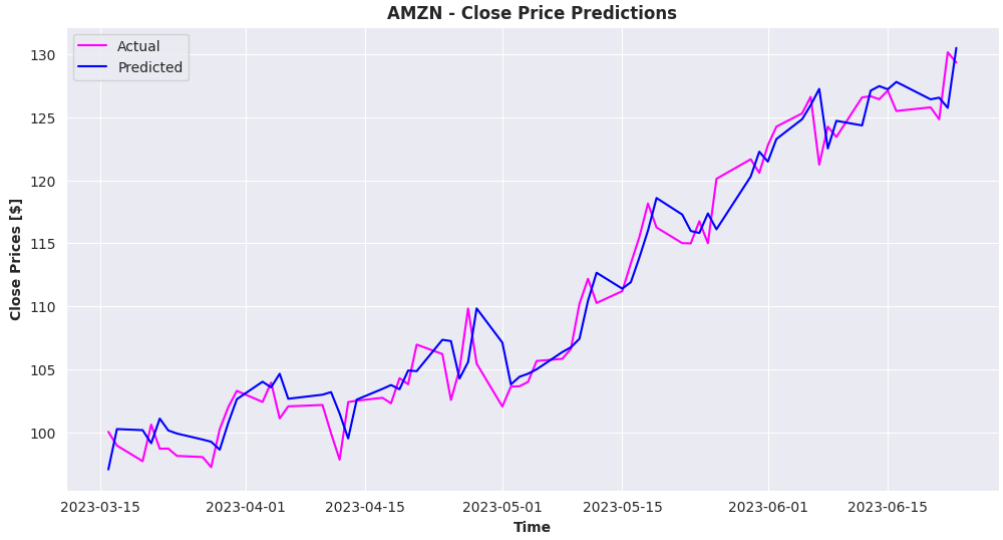


Figure 6.8: `LSTM` - Amazon - Predictions.

After just one attempt, we were able to achieve the following performance on the test set:

- Apple:
  - **MSE**: 4.8243×10^{-4}
  - **SMAPE**: 1.9129
- Amazon:
  - **MSE**: 4.6000×10^{-4}
  - **SMAPE**: 6.3832

Too easy? Stock prices are not necessarily "easy" to predict with LSTM or any other predictive model. Predicting stock prices accurately is a complex and challenging task due to various factors, including market dynamics, economic indicators, news events and investor sentiment, among others. However, RNN networks (LSTM for us) are a popular choice for modeling sequential data, such as stock prices, due to their ability to capture temporal dependencies and patterns in the data.

There are a few reasons why LSTMs may show promise in predicting stock prices:

- **Capturing temporal dependencies**. LSTMs excel at capturing dependencies over time, which can be crucial in stock price prediction.
- **Handling nonlinear relationships**. Stock price movements are (often) nonlinear and influenced by a multitude of factors. LSTMs can learn complex nonlinear relationships and potentially capture hidden patterns that linear models may struggle to capture.

Trying to predict the next close price given the previous T close prices could not be useful. Let's see an example:

- Suppose that our model has a very high "accurancy"[1].

[1] In stock scenario, the accurancy of a model could be defined, for example, as the mean percentage error.

- Also, suppose that the model's predicted direction is always wrong.
- Example: yesterday's price is \$1000 and today's price is \$1000.01. Our model, given yesterday's price as input, predicts \$999.99 (error = \$0.02).
- If I made the decision, based on the predicted price, to sell the stock, I would always lose money!

That being said, we could try to understand the *trend* of a stock, which is often considered more important than accurately predicting the prices of the next day. Here are a few reasons why:

- **Long-term perspective**. Stock investing is typically focused on the long-term. Investors aim to identify stocks that have the potential for sustained growth over an extended period. Understanding the overall trend helps investors make informed decisions about whether to buy, hold, or sell a stock based on its long-term prospects.
- **Risk management**. By focusing on the trend, investors can better manage risk. Short-term price fluctuations are influenced by various factors and can be volatile and unpredictable. Relying solely on short-term price predictions can expose investors to unnecessary risks. Understanding the broader trend provides a more comprehensive view of the stock's performance and helps investors assess risk levels.
- **Avoiding noise and speculation**. Short-term price movements can be influenced by market noise, rumors, or short-term market sentiment. Focusing on the trend helps investors filter out such noise and avoid getting swayed by short-term fluctuations driven by speculation or market sentiment.

What could a valid metric to estimate the trend be? Based on the considerations made above, we decided to focus our analysis to **returns**.

6.2 Return analysis

While stock prices still hold significance, *returns* provide a more meaningful and comprehensive perspective on investment performance. Returns are the difference between the price of an asset at two different times. Mathematically, we can express the return for a period of time T as:

$$RETURN(t) = Price(t) - Price(t - T)$$

A big advantage of using returns is that they are **stationary**. Infact, when a time-series is not stationary, a *differencing* operation is often applied to it. In other words, we take the difference between two "adjacent" samples.

The problem Suppose that we bought a stock on day t. Now the problem we are addressing is the following: given a sequence of returns of the last S trading days, we would like to predict the return on investment (**ROI**) for day $t+T$ ($t+T = S+1$). This means that, for each trading day, we calculate the returns as if the stock were bought T days ago. With the returns of the last S trading days, we aim to predict the return on investment if we were to sell the stock tomorrow.

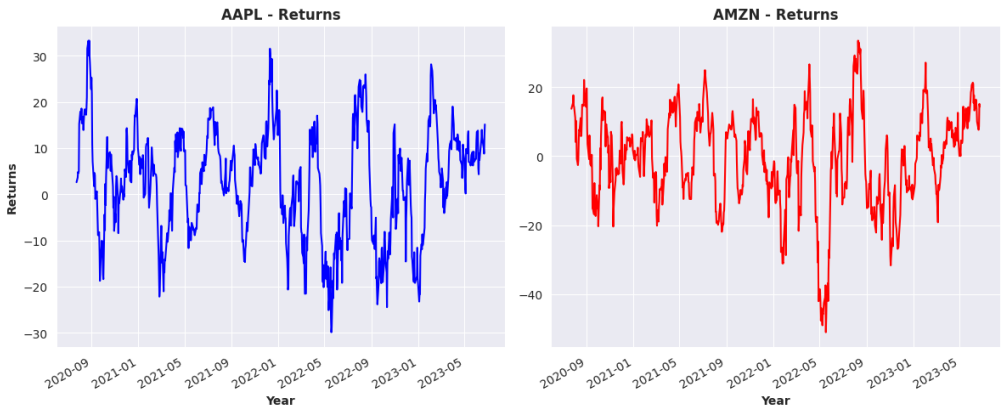


Figure 6.9: Apple and Amazon return. $T = 21$ days.

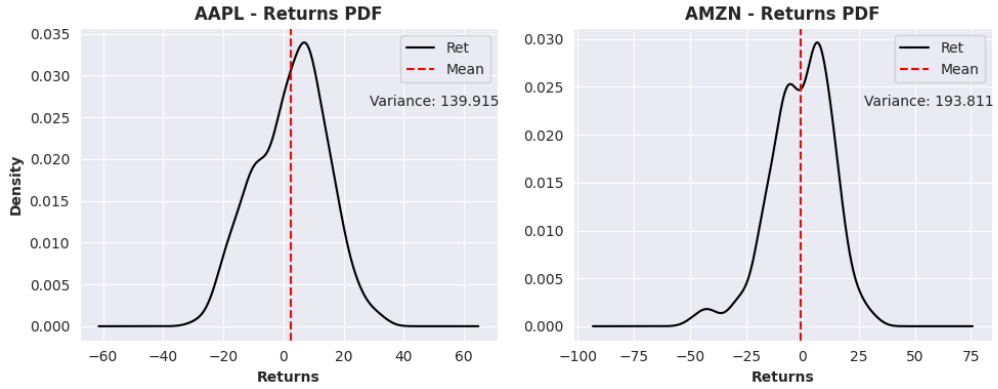


Figure 6.10: PDF of Apple's and Amazon's return.

This time, we decided to use **z-score normalization** instead of applying min-max scaling. The reason behind choosing z-score normalization is that the returns exhibit a distribution similar to a normal distribution. Thus, we opted to preserve the distribution characteristics rather than scaling the values between 0 and 1.

We started with the exact same model used in the close price analysis. Both `RNN` and `LSTM` models have the following architecture:

- One `SimpleRNN/LSTM` layer with 8 `units`.
- The `SimpleRNN/LSTM` layer takes 7 input and has 1 output.
- One `Dense` layer with just 1 output (the latter is also the model's output).

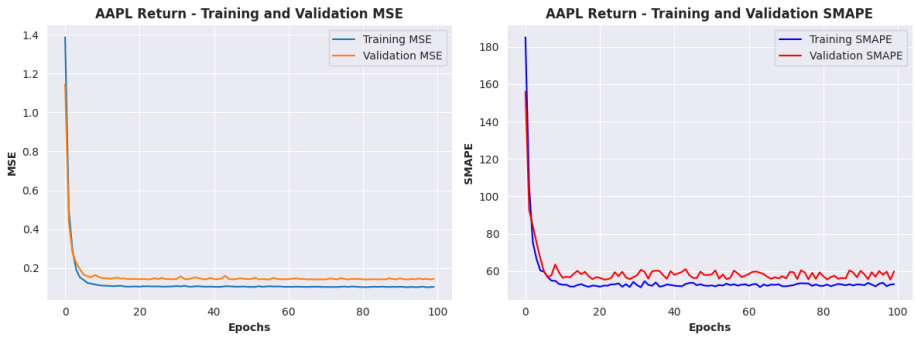
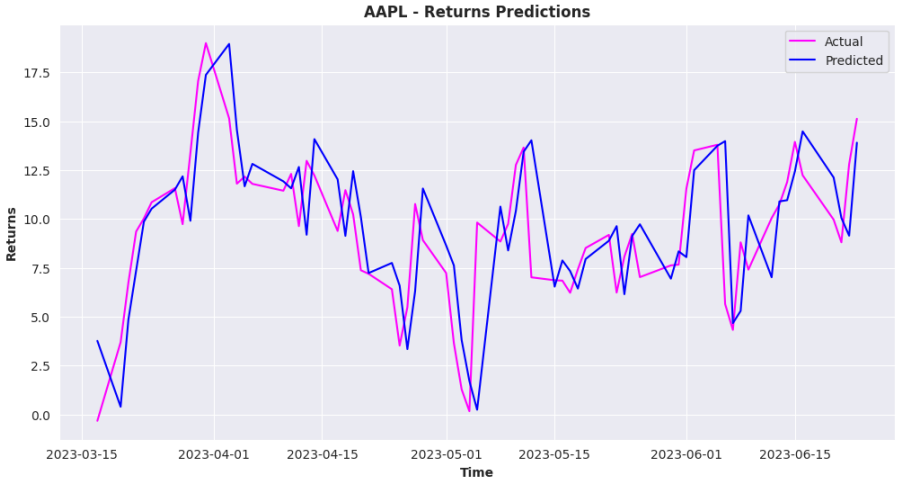
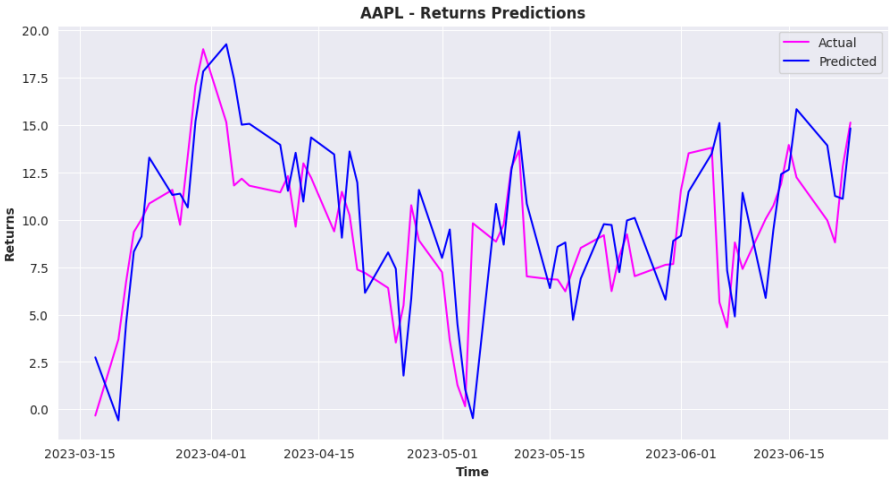


Figure 6.11: `LSTM` - Apple - Training process.



(a) `LSTM` - Apple - Predictions.



(b) `SimpleRNN` - Apple - Predictions.

Figure 6.12: Comparison of prediction results for Apple.

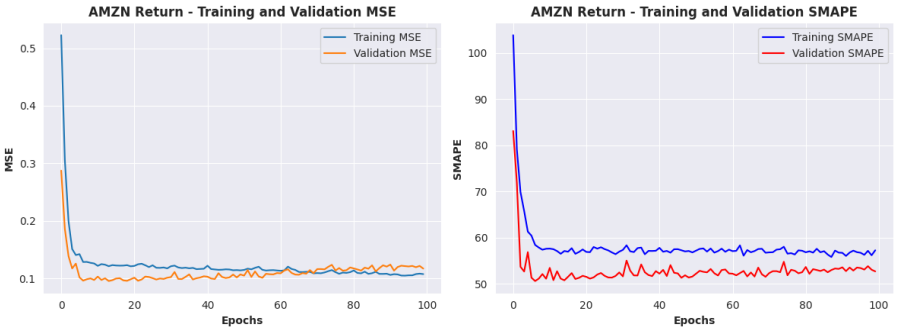
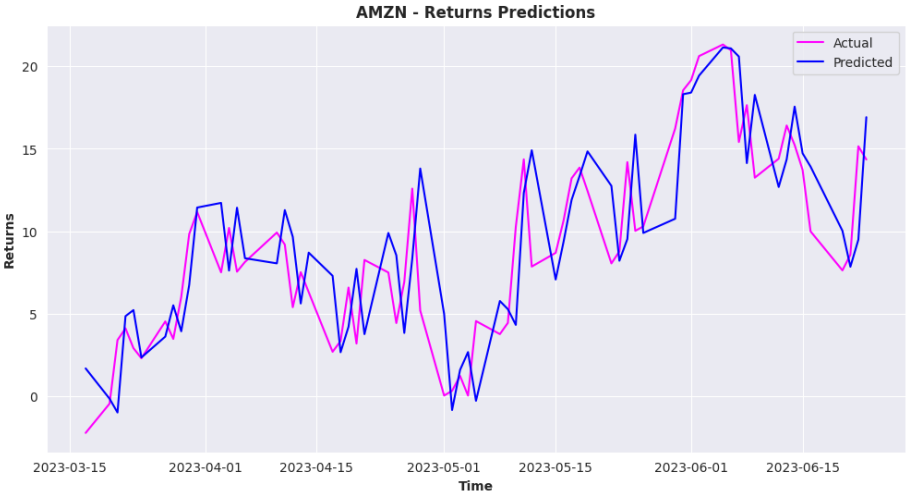
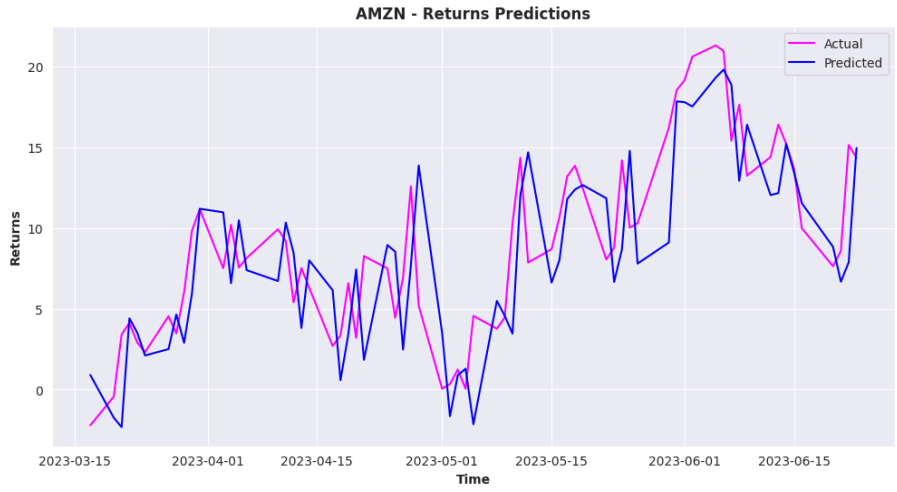


Figure 6.13: `LSTM` - Amazon - Training process.



(a) `SimpleRNN` - Amazon - Predictions.



(b) `LSTM` - Amazon - Predictions.

Figure 6.14: Comparison of prediction results for Amazon.

At the first attempt, we were able to achieve the following performance on the test set:

- Apple:
  - `SimpleRNN` **MSE**: 0.0727, `LSTM` **MSE**: 0.0587
  - `SimpleRNN` **SMAPE**: 41.4139, `LSTM` **SMAPE**: 38.2556
- Amazon:
  - `SimpleRNN` **MSE**: 0.0559, `LSTM` **MSE**: 0.0598
  - `SimpleRNN` **SMAPE**: 36.0006, `LSTM` **SMAPE**: 39.5702

After this first analysis we can conclude that, with respect to ARIMA model,**the deep learning approach using both lstm and rnn's can perform better achieving fewer erros by an order of magnitude lower**. [2]
Looking at the performance of both Apple and Amazon, we observed that the **smape** metric was unsatisfactory. This could be attributed to the high noise present in the dataset. To improve the models' learning ability, we decided to apply a moving average with a `window_size` of $S = 7$ days (Figure 6.15). The moving average is intended to smooth out fluctuations in the return's time series. Importantly, applying the moving average does not compromise the significance of the predictions. Instead, it aids in better understanding the stock's performance by eliminating irrelevant movements that naturally occur due to the unpredictability of the stock market.

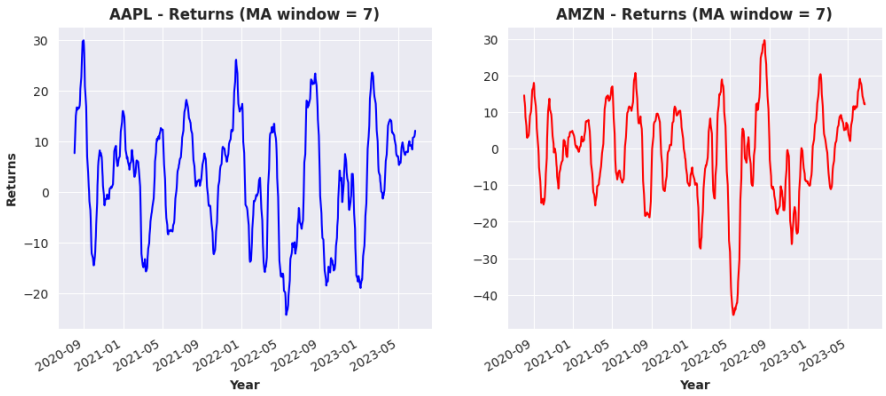


Figure 6.15: `LSTM` - Apple and Amazon returns after moving average (`window_size` = 7).

[2]We specify that the ARIMA parameters may not be the choice that minimizes the MSE but are the result of a trial and error process.

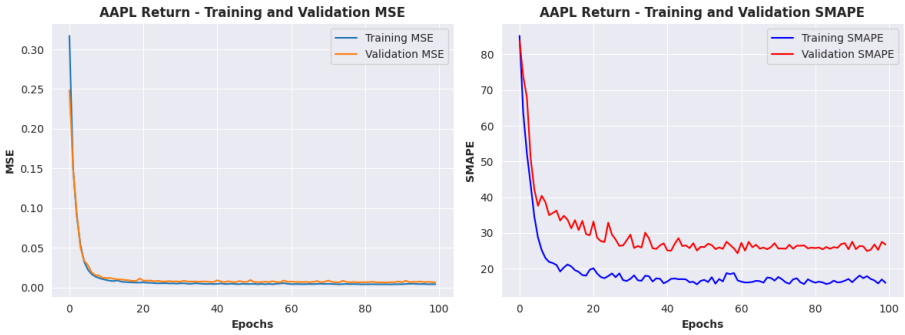
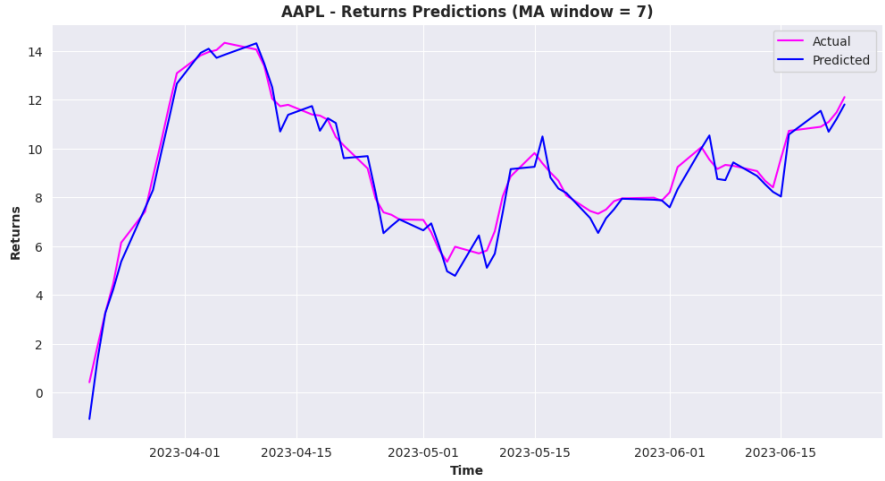
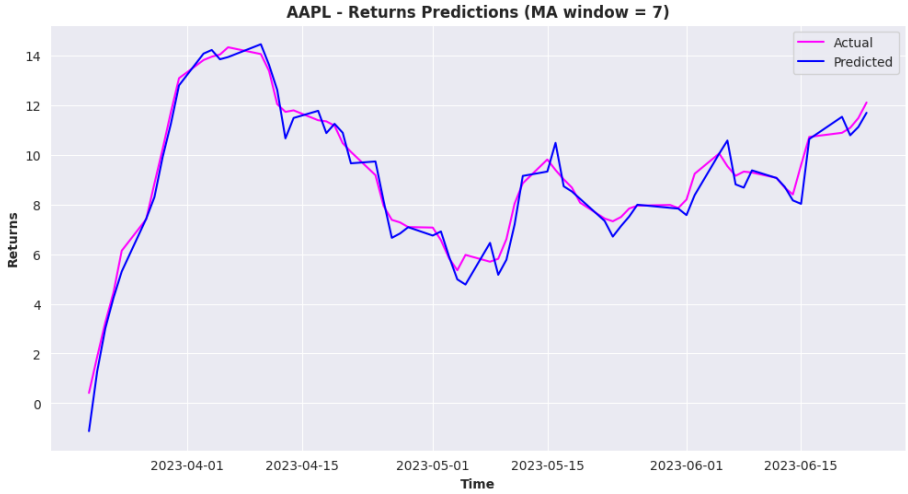


Figure 6.16: `LSTM` - Apple - Training process after applying moving average.



(a) `SimpleRNN` - Apple - Predictions after applying moving average.



(b) `LSTM` - Apple - Predictions after applying moving average.

Figure 6.17: Comparison of prediction figures for Apple.

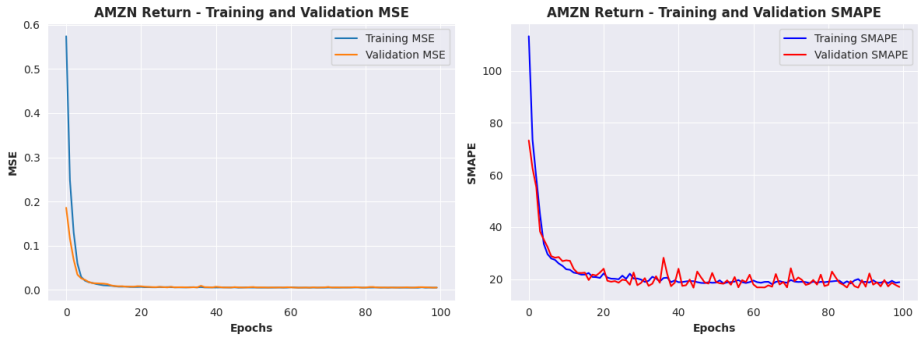
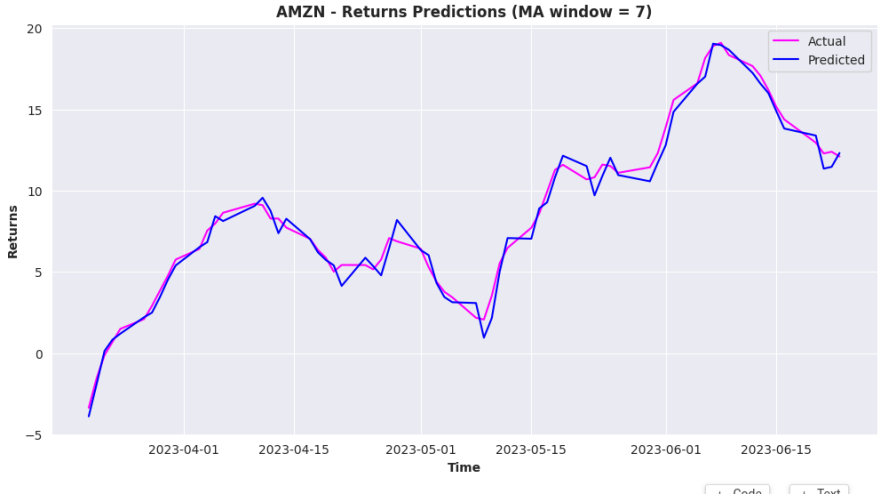
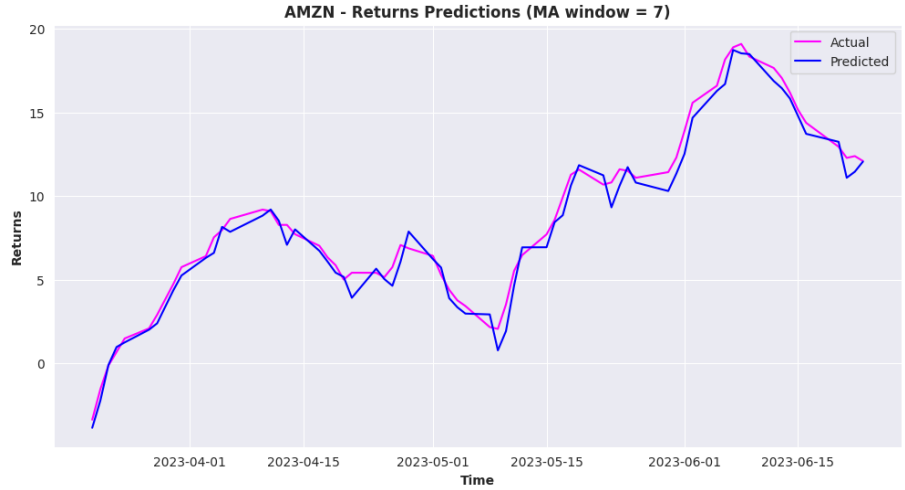


Figure 6.18: `LSTM` - Amazon - Training process after applying moving average.



(a) `SimpleRNN` - Amazon - Predictions after applying moving average.



(b) `LSTM` - Amazon - Predictions after applying moving average.

Figure 6.19: Comparison of prediction for Amazon.

After applying the moving average, we were able to achieve the following perfomance:

- Apple:
  - `SimpleRNN` **MSE**: 0.0026, `LSTM` **MSE**: 0.0027
  - `SimpleRNN` **SMAPE**: 7.6651, `LSTM` **SMAPE**: 8.3605
- Amazon:
  - `SimpleRNN` **MSE**: 0.0023, `LSTM` **MSE**: 0.0031
  - `SimpleRNN` **SMAPE**: 7.4194, `LSTM` **SMAPE**: 8.4921

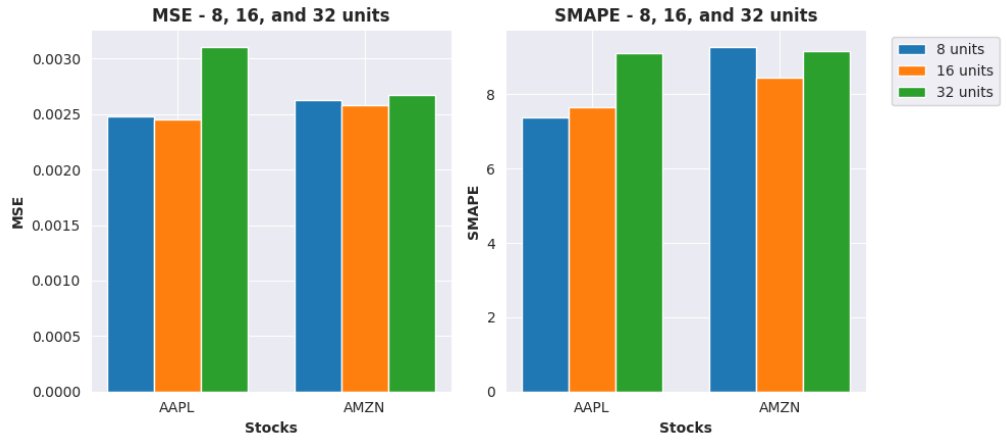
6.2.1 Fine tuning

To better understand which hyperparameters could lead to a better model, we chose to analyze different approaches:

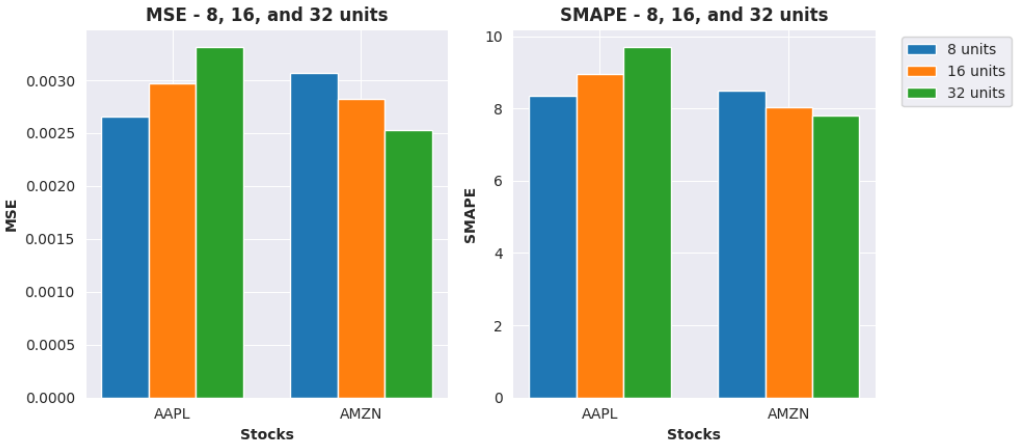
- Increasing the number of `units` in `SimpleRNN/LSTM` layer.
- Trying different training hyperparameters.
- Adding stacked layers.
- Adding dropout.

Observations:

- By increasing the number of `units` in the recurrent layer, we observed that the error (measured both with MSE and SMAPE) does not decrease, but in most cases, it increases (6.20).
- By increasing the number of `epochs` and decreasing the `batch_size`, we were not able to clearly observe a pattern of improvement or worsening (Figure 6.21).
- Adding another recurrent layer with twice (or four times) the number of units compared to the previous case resulted in either equal or worse performance. In particular, the error (both MSE and SMAPE) increased significantly when we introduced `dropout`. This could be due to the fact that the network never suffered from overfitting, suggesting that the addition of this regularization technique reduces the network's ability to learn patterns in the target variable (Figure 6.22).



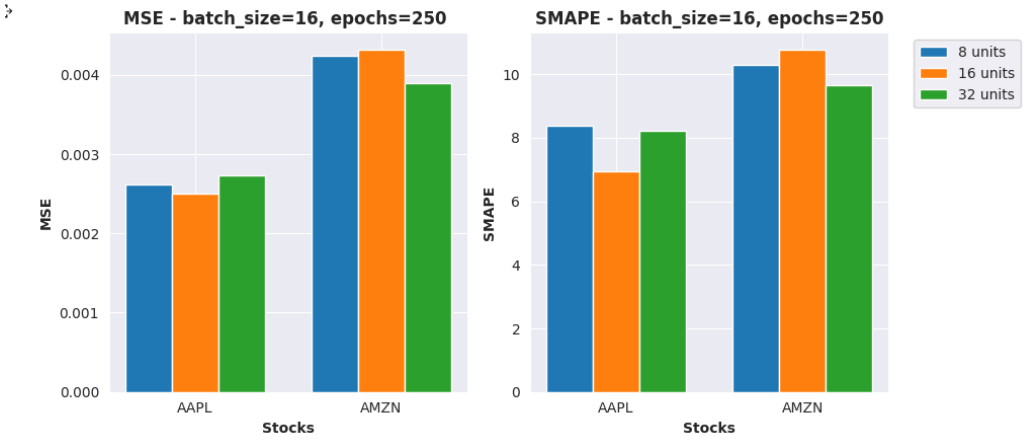
(a) `SimpleRNN`



(b) `LSTM`

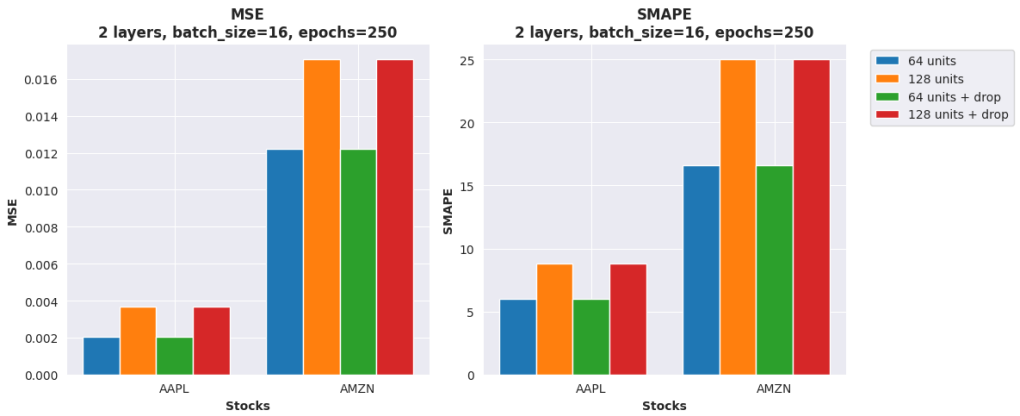
Figure 6.20: Comparison of the benefits of increasing the number of `units`.

(a) `SimpleRNN`

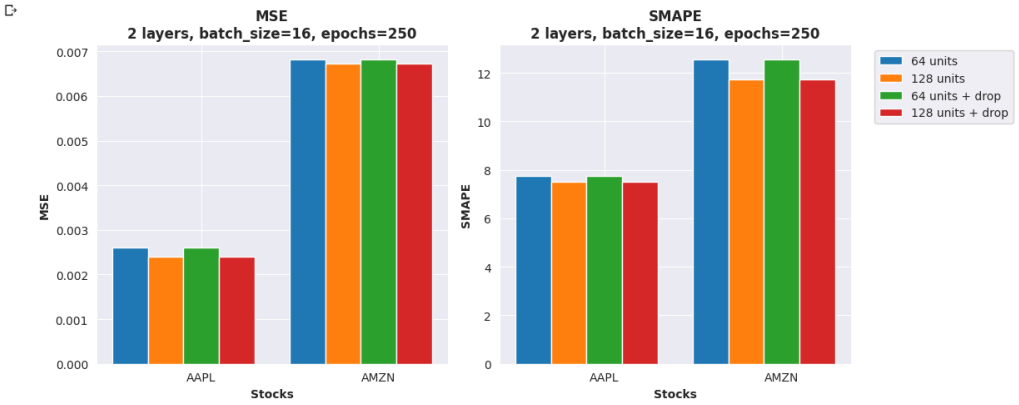


(b) `LSTM`

Figure 6.21: Comparison of the benefits of increasing `epochs` and decreasing `batch_size`.



(a) `SimpleRNN`



(b) `LSTM`

Figure 6.22: Comparison of the benefits of adding one more `layer` and introducing `dropout`.

6.3 `GridSearch`

We decided to use `GridSearch` to find the best configuration of hyperparameters. This approach is particularly suitable because our previous analysis revealed that the model performs better when it is simpler. Therefore, we anticipate that a smaller parameter grid would be sufficient for our search.

To prevent overfitting, we will adopt an *early stopping* approach. The function `compile_and_fit()` creates an `EarlyStopping` callback. This callback monitors the validation `smape` during training and stops the training process early if there is no improvement for a certain number of epochs specified by `_patience` (default: `10`) and if the improvement is less than `_min_delta` (default: `0.0001`).
For both `SimpleRNN` and `LSTM` models we perfomed the search through the following two grids:

```
param_grid = {
      'units': [4, 8, 16, 32],
      'batch_size': [8, 16, 32],
      'epochs': [100, 250],
      'patience': [10, 25],
      'learning_rate': [0.1, 0.01, 0.001]
  }
```

Figure 6.23: Hyperparameters grid used to find the best model (single recurrent layer).

```
param_grid = {
      'units': [32, 64, 128],
      'batch_size': [8, 16, 32],
      'epochs': [100, 250],
      'patience': [10, 25],
      'learning_rate': [0.1, 0.01, 0.001]
  }
```

Figure 6.24: Hyperparameters grid used to find the best model (two recurrent layers).

```
GridSearch('RNN', 'AAPL', layers=1)
[...]
[RNN] AAPL - NEW Best Hyperparameters: {'units': 16,
                                        'batch_size': 16,
                                        'epochs': 100,
                                        'patience': 25,
                                        'learning_rate': 0.01}
[RNN] AAPL - NEW Best MSE Score: 0.002115989683962553
[...]
[RNN] AAPL - NEW Best Hyperparameters: {'units': 32,
                                        'batch_size': 16,
                                        'epochs': 100,
                                        'patience': 25,
                                        'learning_rate': 0.01}
[RNN] AAPL - NEW Best MSE Score: 0.0020845359108380606
[..]
Final result:
[RNN] AAPL - Best Hyperparameters: {'units': 32,
                                   'batch_size': 32,
                                   'epochs': 100,
                                   'patience': 25,
                                   'learning_rate': 0.01}
[RNN] AAPL - Best MSE Score: 0.0019636182023850444

GridSearch('LSTM', 'AAPL', layers=1)
[...]
[LSTM] AAPL - NEW Best Hyperparameters: {'units': 4,
                                         'batch_size': 16,
                                         'epochs': 250,
                                         'patience': 10,
                                         'learning_rate': 0.01}
[LSTM] AAPL - NEW Best MSE Score: 0.002096100017229204
[..]
[LSTM] AAPL - NEW Best Hyperparameters: {'units': 4,
                                         'batch_size': 32,
                                         'epochs': 250,
                                         'patience': 10,
                                         'learning_rate': 0.001}
[LSTM] AAPL - NEW Best MSE Score: 0.0020304527668559493
[...]
[LSTM] AAPL - NEW Best Hyperparameters: {'units': 8,
                                         'batch_size': 32,
                                         'epochs': 250,
                                         'patience': 25,
                                         'learning_rate': 0.01}
[LSTM] AAPL - NEW Best MSE Score: 0.001978851111024825
[...]
Final result:
[LSTM] AAPL - Best Hyperparameters: {'units': 32,
                                    'batch_size': 32,
                                    'epochs': 250,
                                    'patience': 10,
                                    'learning_rate': 0.01}
[LSTM] AAPL - Best MSE Score: 0.0019284040457014013
```


Figure 6.25: Apple - Best hyperparameters (just one recurrent layer).

```
GridSearch('RNN', 'AAPL', layers=2)
[...]
[RNN] AAPL - NEW Best Hyperparameters: {'units': 32,
                                        'batch_size': 8,
                                        'epochs': 100,
                                        'patience': 25,
                                        'learning_rate': 0.01}
[RNN] AAPL - NEW Best MSE Score: 0.01629234696112482
[...]
[RNN] AAPL - NEW Best Hyperparameters: {'units': 32,
                                        'batch_size': 16,
                                        'epochs': 100,
                                        'patience': 10,
                                        'learning_rate': 0.001}
[RNN] AAPL - NEW Best MSE Score: 0.010197396804788249
[...]
Final result:
[RNN] AAPL - NEW Best Hyperparameters: {'units': 64,
                                        'batch_size': 16,
                                        'epochs': 100,
                                        'patience': 10,
                                        'learning_rate': 0.001}
[RNN] AAPL - NEW Best MSE Score: 0.006914621536209187


GridSearch('LSTM', 'AAPL', layers=2)
[...]
[LSTM] AAPL - NEW Best Hyperparameters: {'units': 32,
                                         'batch_size': 8,
                                         'epochs': 100,
                                         'patience': 10,
                                         'learning_rate': 0.1}
[LSTM] AAPL - NEW Best MSE Score: 0.005843650288489823
[...]
Final result:
[LSTM] AAPL - NEW Best Hyperparameters: {'units': 32,
                                         'batch_size': 16,
                                         'epochs': 250,
                                         'patience': 25,
                                         'learning_rate': 0.1}
[LSTM] AAPL - NEW Best MSE Score: 0.005506106377210276
```

Figure 6.26: Apple - Best hyperparameters (two recurrent layers).

```
GridSearch('RNN', 'AMZN', layers=1)
[...]
[RNN] AMZN - NEW Best Hyperparameters: {'units': 4,
                                        'batch_size': 16,
                                        'epochs': 250,
                                        'patience': 25,
                                        'learning_rate': 0.001}
[RNN] AMZN - NEW Best MSE Score: 0.002155339845745889
[...]
[RNN] AMZN - NEW Best Hyperparameters: {'units': 8,
                                        'batch_size': 8,
                                        'epochs': 250,
                                        'patience': 25,
                                        'learning_rate': 0.001}
[RNN] AMZN - NEW Best MSE Score: 0.002010882661660215
Final result:
[RNN] AMZN - Best Hyperparameters: {'units': 8,
                                    'batch_size': 8,
                                    'epochs': 250,
                                    'patience': 25,
                                    'learning_rate': 0.001}
[RNN] AMZN - Best MSE Score: 0.002010882661660215

GridSearch('LSTM', 'AMZN', layers=1)
[...]
[LSTM] AMZN - NEW Best Hyperparameters: {'units': 4,
                                         'batch_size': 8,
                                         'epochs': 100,
                                         'patience': 10,
                                         'learning_rate': 0.01}
[LSTM] AMZN - NEW Best MSE Score: 0.0021412080608309426
[...]
[LSTM] AMZN - NEW Best Hyperparameters: {'units': 8,
                                         'batch_size': 16,
                                         'epochs': 250,
                                         'patience': 10,
                                         'learning_rate': 0.01}
[LSTM] AMZN - NEW Best MSE Score: 0.0020062254086446684
[...]
Final result:
[LSTM] AMZN - Best Hyperparameters: {'units': 8,
                                     'batch_size': 16,
                                     'epochs': 250,
                                     'patience': 10,
                                     'learning_rate': 0.01}
[LSTM] AMZN - Best MSE Score: 0.0020062254086446684
```

Figure 6.27: Amazon - Best hyperparameters (just one recurrent layer).

```
GridSearch('RNN', 'AMZN', layers=2)
[...]
[RNN] AMZN - NEW Best Hyperparameters: {'units': 32,
                                        'batch_size': 8,
                                        'epochs': 250,
                                        'patience': 10,
                                        'learning_rate': 0.01}
[RNN] AMZN - NEW Best MSE Score: 0.019186524950213734
[...]
[RNN] AMZN - NEW Best Hyperparameters: {'units': 32,
                                        'batch_size': 32,
                                        'epochs': 100,
                                        'patience': 10,
                                        'learning_rate': 0.001}
[RNN] AMZN - NEW Best MSE Score: 0.012852416884762187
[...]
Final result:
[RNN] AMZN - Best Hyperparameters: {'units': 128,
                                   'batch_size': 16,
                                   'epochs': 100,
                                   'patience': 25,
                                   'learning_rate': 0.001}
[RNN] AMZN - Best MSE Score: 0.010115268677621563

GridSearch('LSTM', 'AMZN', layers=2)
[...]
[LSTM] AMZN - NEW Best Hyperparameters: {'units': 32,
                                         'batch_size': 32,
                                         'epochs': 250,
                                         'patience': 10,
                                         'learning_rate': 0.001}
[LSTM] AMZN - NEW Best MSE Score: 0.012694279605782956
[...]
[LSTM] AMZN - NEW Best Hyperparameters: {'units': 64,
                                         'batch_size': 8,
                                         'epochs': 250,
                                         'patience': 10,
                                         'learning_rate': 0.01}
[LSTM] AMZN - NEW Best MSE Score: 0.010686980153847097
[...]
Final result:
[LSTM] AMZN - Best Hyperparameters: {'units': 64,
                                    'batch_size': 16,
                                    'epochs': 100,
                                    'patience': 10,
                                    'learning_rate': 0.001}
[LSTM] AMZN - Best MSE Score: 0.01033803305688953
```

Figure 6.28: Amazon - Best hyperparameters (two recurrent layers).

Observation:

- As seen previously, even though the best parameters indicate `units`=32, during the

search for the best hyperparameters, we noticed that the model achieves comparable errors to the best one with low network complexity (`units`=4).

- The model with stacked recurrent layers performs worse (by an order of magnitude) compared to the model with a single layer. A plausible explanation could be the following: adding stacked layers allows the network to learn hidden patterns that are not directly extractable from the shape of the dataset, in most cases. In our case, after performing the MA (Moving Average), the dataset does not seem to exhibit hidden patterns. One could think that the model would have performed better if we had trained it with the original dataset. This could be true but we think that would be not useful: returns shows a very noisy and hard-to-predict shape.

Chapter 7

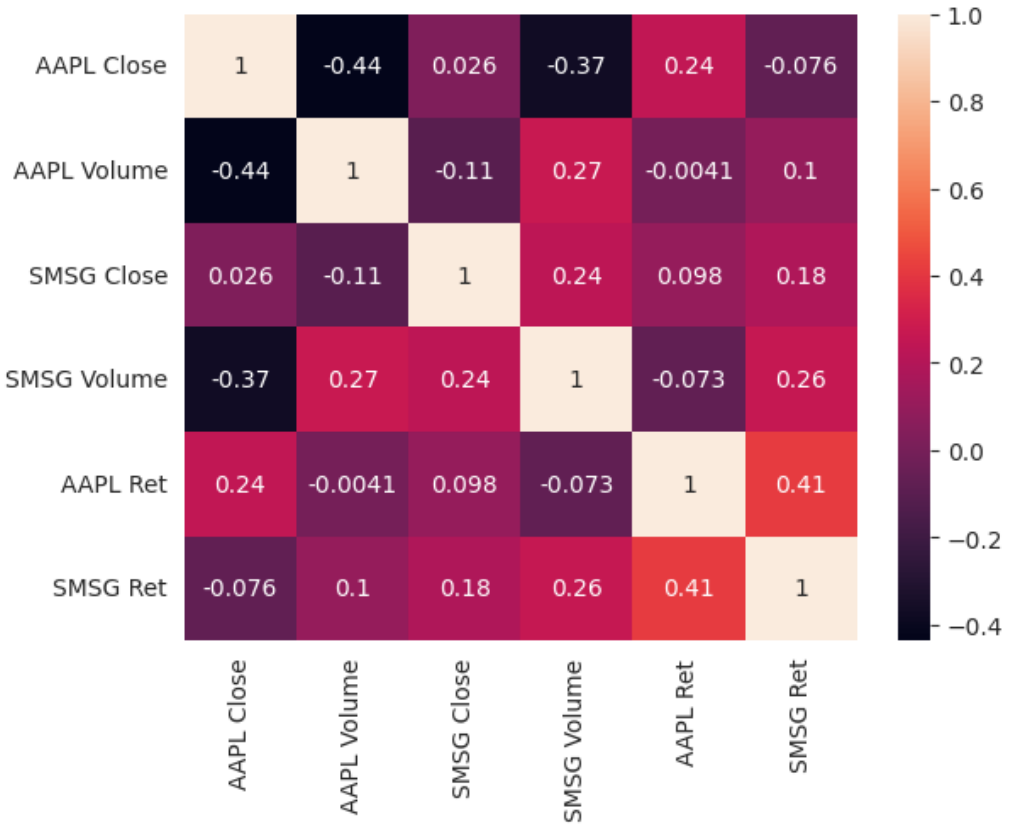
RNN vs LSTM multivariate approach

We also wanted to see if adding more features to our input data was useful to obtain more precise forecasts. The initial approach we followed was to add the following features:

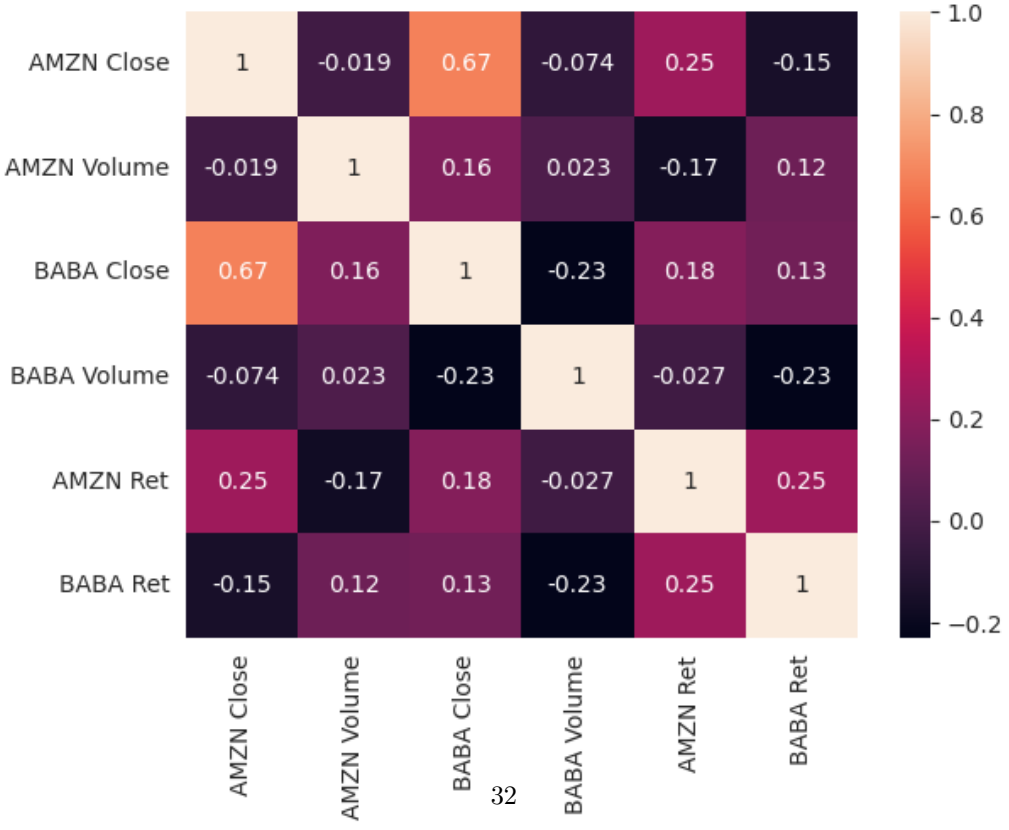
- Adding **Stock Volumes**
- Adding **competitor companies data** as inputs: We simply took the stock closing values of a competitor (we choose Samsung for predicting stock returns of Apple and AliBaba to predict stock returns of Amazon).

To validate the choice of adding the close prices of other companies we computed the $p - value$ for both Samsung and Alibaba features and the values were below a certain significance level, so we rejected the hypothesis that the features were uncorrelated.

To better understand the amplitude of the correlation (both in positive or negative terms) we plotted the correlation matrix between the features for both Amazon and Apple with respect to the competitors.



(a) `Apple Samsung` Correlation Matrix



(b) `Amazon Alibaba` Correlation Matrix

Figure 7.1: Correlation Matrixes

After inspecting the matrixes, for each company we choos the 2 highest correlation features, and so we decided to not use the volumes as first thought. [1], so for the Apple returns we choose the Samsung returns and the Close value. In the case of Amazon we choose the Close value along with Alibaba return values.

Table 7.1: Confronto MSE Multivariate ed Univariate (`SimpleRNN`)

Dataset	Univariate MSE	Multivariate MSE
AAPL 1	0.0019	8.27
AMZN 2	0.0019	6.58e-5

Table 7.2: Confronto MSE Multivariate ed Univariate (`LSTM`)

Dataset	Univariate MSE	Multivariate MSE
AAPL 1	0.0019	8,72e-5
AMZN 2	0.0020	7.10e-5

[1] We tried a model with all starting features but performed worse so we remained with the choice to use the highest 3 features for boht datasets

Chapter 8

Conclusion

Regarding the univariate case, we were able to observe that there are no substantial differences between RNNs and LSTMs. Contrary to what we thought, predicting returns is not a task that seems to leverage the long-term data memorization capability of LSTMs. That being said, it may be convenient to opt for an RNN-based model rather than an LSTM-based one since the cost is lower.

Even in the multivariate case, both models recorded similar errors (on the order of 10^{-5}), and consequently, we can say that the LSTM's memorization capabilities are not exploited by the task.

As expected, the multivariate model performs better than the univariate one. Specifically, with the multivariate model, we were able to achieve, on average, a much lower error (MSE) by two orders of magnitude.

In terms of the usefulness of the work performed, we believe that both models can be used to gain a macro view of short-term investment trends. However, we think that in order to make educated decisions, it is necessary to complement these models not only with analytical information but also by observing market "sentiments." One possible approach could be to develop a model capable of conducting sentiment analysis.