UNIVERSITAT POLITÈCNICA DE VALÈNCIA

WISCONSIN
UNIVERSITY OF WISCONSIN–MADISON

DESIGNING DEMAND FORECASTING MODELS USING DEEP LEARNING

Author: Angel Ortiz Perez

UPV Supervisor: Pedro Gomez Gasquet

UW-Madison Supervisor: Xin Wang

Academic year: 2022-2023

ABSTRACT

This work will explore the use of different deep learning models to predict the future demand of a product, which can be highly useful for the operational management of a company. The development of aggregate production plans (APP), master production schedules (MPS), and material requirements planning (MRP) relies on the forecasted demand of the products included in these plans. Therefore, the more accurate the demand forecast used for these plans, the more efficient the planning and execution will be. The demand for a product can be treated as time series data. Time series data refers to data that is collected over time and arranged based on the moments they were recorded. Generally, this data is collected at regular intervals such as days, weeks, months, or years. Currently, there are various deep learning models that can work with this type of data and enable future predictions of the time series values. This work will utilize several deep learning models, including recurrent neural networks (RNN), LSTM networks, convolutional neural networks (CNN), and Transformers. The initial focus will be on working with unidimensional time series data. An example of such data is stock prices, which will be used in this work due to their easy accessibility and abundance of historical data. Once the models have been programmed and successfully trained, the prediction of multidimensional time series data will be conducted using the aforementioned models. The implementation of the deep learning models, their training, testing, and result analysis, will be carried out using Python and the PyTorch module. The obtained results will allow for the evaluation of each model's effectiveness and determination of which models are most suitable for predicting the future demand of a product. This can be of great value in the decision-making process for the operational management of a company.

TABLE OF CONTENTS

1. Introduction

1.1. Context and motivation

In today's fast-paced and highly competitive business environment, companies are constantly seeking ways to gain a competitive advantage. Accurate demand forecasting is critical to the success of any company, as it enables them to optimize production and inventory levels, reduce costs, and improve customer satisfaction.

Traditionally, companies have relied on statistical models to forecast demand, like ARMA or ARIMA. However, these models have limitations when dealing with complex and dynamic data, such as time series data. Deep learning models, on the other hand, are capable of processing large amounts of data and extracting patterns that can be used to make accurate predictions.

The objective of this work is to explore the use of deep learning models for demand forecasting and to evaluate their effectiveness in predicting demand. The work will focus on time series data, which is a collection of observations made over time.

Several deep learning models will be evaluated, including recurrent neural networks (RNNs), long short-term memory (LSTM) networks, convolutional neural networks (CNNs), and transformers. RNNs and LSTMs have been shown to be effective in processing time series data and making accurate predictions. On the other hand, CNNs and Transformers are typically used for other purposes, but can also be applied to time series data.

To evaluate the effectiveness of these models, root mean square error (RMSE) will be the metric to be used. The models will be trained on historical data and tested on a holdout set to measure their predictive accuracy. This data would consist of stock prices, due to the large amount of available data of this type on the internet and the fact that these are also time series data.

When evaluating this model, it has to be noted that stock prices are much more difficult to predict than some products demand. In fact, stock prices are theoretically impossible to predict. Therefore, this should be something to consider during the evaluation process of these models.

1.2. Objectives

Several objectives will try to be accomplished with this work:

- Assess the performance of different deep learning models that have been stated before and evaluate if some models that are not typically used for time series data like CNN or Transformer can improve the RNN or LSTM performance.
- Assess the impact that the amount of information given to the deep learning models can have on the accuracy of predictions. This is related to using univariate inputs or multivariate inputs and finding which inputs can actually improve the model's performance.
- Explore the influence of various hyperparameters and network architectures on prediction accuracy of the models and techniques for best hyperparameter fitting.
- Analyze performance of different models for several forecast horizons.
- Find an accurate and reliable tool for demand forecasting, which can improve companies' production planning and inventory management. When applied properly, deep learning models can help companies gain a competitive advantage by improving their ability to forecast demand and optimize their operations.

2. Theoretical framework

2.1. Demand forecasting

Demand forecasting is a key element in the process of companies' decision making and planning. Demand forecasting can be understood as the process of estimating the quantity of a product or service that customers will require in a specific future period. Some of the traditional methods used for demand forecasting are:

- **Moving average**: This method calculates the average demand in a given time period. The moving average can be simple or weighted, giving a higher weight to more relevant or recent data. Nevertheless, is difficult to capture complex data patterns or demand shifts using this method.
- **Exponential smoothing:** This approach assigns decreasing exponential weights to past observations. It is based on the assumption that more recent observations have a greater influence on future predictions. However, exponential smoothing may not be suitable for data with nonlinear patterns or seasonality.
- **Regression analysis:** This method uses independent variables, such as prices, promotions, or other relevant metrics, to predict demand. It is based on the linear relationship between the independent variables and the target variable. However, regression analysis may not capture nonlinear relationships or complex interaction effects.
- **Time Series Analysis**: This method focuses on analyzing historical data to identify patterns, trends, and seasonality in demand. Time series models, such as autoregressive integrated moving average (ARIMA) or seasonal decomposition of time series (STL), are commonly employed for forecasting. These models capture the inherent time-dependent structure in data and can be useful for short- to medium-term predictions.
- **Simulation and Scenario Analysis**: Simulation-based approaches involve creating models that simulate different scenarios to estimate future demand. By adjusting various parameters and assumptions, organizations can assess the potential impact of different factors on demand and make informed decisions accordingly.
- **Machine Learning and Data Mining**: Machine learning techniques, such as random forests, gradient boosting, or support vector machines, can be applied to demand forecasting. These algorithms can discover complex patterns and relationships in large

datasets, enabling more accurate predictions. Data mining techniques, such as association rule mining or clustering, can also be used to identify patterns and segment customers based on their purchasing behavior.

- **Neural Networks**: Artificial neural networks, particularly deep learning models like recurrent neural networks (RNNs) and long short-term memory (LSTM) networks, have gained popularity in demand forecasting. These models can handle large volumes of data and capture complex relationships, making them suitable for analyzing non-linear patterns and long-term dependencies. These and other types of models will be the main focus in this work.

2.2. Time series

Time series are groups of data that are collected and recorded at regular intervals over time. In the context of demand forecasting, time series represent observations of the demand for a product or service over time. Having a good understanding of the characteristics and properties of time series is relevant to applying appropriate prediction techniques. Time series data usually have three main components:

- **Trend:** It refers to the overall and persistent direction of the series' behavior in the long term. It can be upward, downward, or remain constant. The trend indicates the direction in which the demand is moving and can be useful in identifying long-term patterns.
- **Seasonality:** This component is related to the regular and predictable fluctuations in demand that occur at specific time intervals, such as annual, monthly, or weekly. Seasonality indicates patterns that repeat in each period and can be helpful in adjusting predictions for recurring cycles.
- **Noise or random component:** It represents random and non-systematic variations in the time series. These variations can be attributed to unpredictable factors or unexpected events that impact demand and do not follow a specific pattern. The noise component can make accurate demand prediction challenging.

Time series also have certain properties and characteristics that need to be considered in their analysis process. These include:

- **Autocorrelation:** Time series tend to exhibit autocorrelation, which means that values observed at close time points are correlated. Autocorrelation can be measured using the

autocorrelation function (ACF) and the partial autocorrelation function (PACF). Autocorrelation indicates the dependence of past values in predicting future values.

- **Non-stationarity:** Time series can be stationary or non-stationary. A stationary time series is one in which statistical properties such as mean, and variance remain constant over time. However, many real-world time series are non-stationary and exhibit trends, seasonal variations, or changes in variance over time, this is the case of stock prices.
- **Structural change:** Time series can experience structural changes, which are significant alterations in the series' behavior at specific points in time. These changes can be due to exceptional events, policy changes, or any other factor that has a notable impact on the time series.
- **Outliers:** Outliers are unusual or extreme values that significantly deviate from the general pattern of the time series. These values can be due to measurement errors, unexpected events, or relevant but unusual information. In this project these values are not going to appear or have to be considered, because data has already been preprocessed before being uploaded to the internet. But when companies measure demand, data would not have been preprocessed, therefore this is an aspect that must be considered and assessed correctly.
- **Persistence:** Persistence refers to the memory of the time series. A highly persistent time series exhibits a strong dependence on past values, indicating that previous observations have a significant impact on future observations. On the other hand, a low persistence time series shows a weaker dependence on past values.

2.3. Recurrent Neural Networks (RNN)

Recurrent Neural Networks (RNNs) are a type of machine learning model widely used in time series analysis. Unlike conventional neural networks, RNNs have recursive connections that allow them to process sequential data and capture long-term dependencies.

In an RNN, neurons not only receive inputs from the current step but also receive information from the previous step. This enables them to maintain an internal memory or hidden state that can influence the current output and is shared across all time steps. This ability to capture temporal dependencies makes them particularly suitable for modeling and predicting time series.

A common architecture of RNN is the Feedback Recurrent Network, where outputs from previous steps are fed back to neurons in the next step. However, traditional RNNs can suffer from difficulties in capturing long-term dependencies due to the vanishing or exploding gradient problem.

In the picture below we can see the structure of an RNN. It can be noted that hidden states from previous time steps are connected to the next time step, providing memory properties to the neural network.

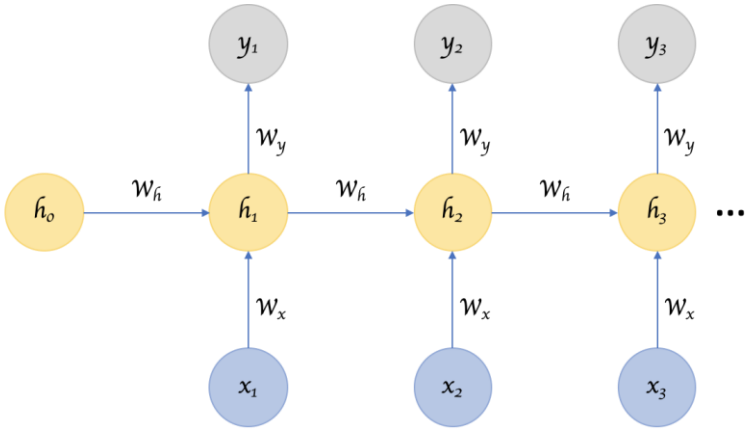

Figure 1. Illustration of RNN internal structure. Source: (VENKATACHALAM, 2019)

To overcome the vanishing or exploding gradient problem, more advanced variants of RNN have been developed, such as Long Short-Term Memory (LSTM) networks and Gated Recurrent Units (GRU). These variants introduce gating mechanisms that control the flow of information and help maintain a balance between short-term memory and long-term memory.

LSTM networks use memory cells and gates to control the writing, reading, and forgetting of information in memory. This allows them to capture patterns of long-term dependency and avoid the vanishing gradient problem. On the other hand, GRU networks also use gates to control the flow of information but with a simpler structure than LSTM networks.

In the following image, the difference between a simple RNN network and LSTM or GRU architecture can be noted. The number of gates and different activation functions in these new networks is much higher compared to RNNs.

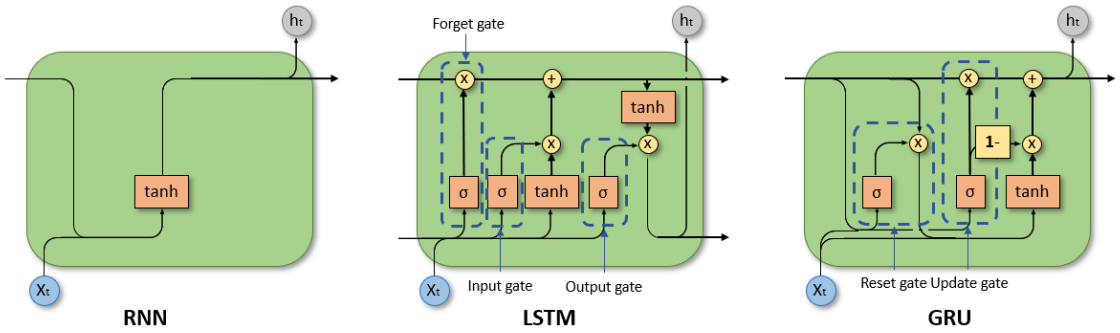

Figure 2. RNN LSTM and GRU internal structure illustration. Source: (Dancker, 2022)

When training an RNN for demand prediction in time series, input data is presented sequentially at each time step, and the network learns to generate a prediction for the next step. The training process involves adjusting the weights of the network's connections using optimization algorithms such as gradient descent.

RNNs have proven to be effective in demand prediction in time series due to their ability to model complex patterns over time. However, their performance can be affected by the choice of hyperparameters, the quantity and quality of the training data, and the proper selection of the network architecture.

2.4. Convolutional Neural Networks (CNN)

Convolutional Neural Networks (CNNs) are another type of deep learning model widely used in the analysis and processing of sequential data, including time series. Although CNNs are primarily known for their applicability in image processing, they have also been shown to be effective in time series prediction and analysis.

Unlike RNNs, CNNs rely on convolution operations to extract relevant features from input data. These convolution operations involve applying filters or kernels across the data, capturing local patterns, and generating feature maps. The resulting feature maps are passed through pooling layers to reduce dimensionality and preserve the most important features.

In the context of time series, CNNs treat the data as an additional dimension, often referred to as a channel dimension. Each time step in the time series is represented as a value in this channel dimension, allowing convolution operations to capture sequential patterns at different scales.

CNNs are especially useful for detecting local patterns and invariant features in time series. For example, in the case of financial data, CNNs can recognize upward or downward patterns in prices within a specific interval. Additionally, pooling layers enable dimensionality reduction and extraction of key features without losing important information.

Like RNNs, training a CNN for demand prediction in time series involves adjusting the weights of the network's connections using optimization algorithms. The choice of CNN architecture, including the number and size of filters, as well as the configuration of pooling layers, is an important factor in achieving optimal performance. The choices to be made and hyperparameters that need to be fitted can be much larger than for RNNs, because CNNs can be more versatile when it comes to the number of layers and neurons in each layer.

It is important to note that CNNs are particularly useful when the patterns of interest in time series are local and repeat over time. However, they may struggle to capture long-term dependencies and more complex sequential relationships. Therefore, the choice of the appropriate model will depend on the nature of the data and the patterns being sought.

2.5. Transformer

The Transformer is a machine learning model architecture that has revolutionized the field of natural language processing and has also proven to be highly effective in time series analysis and prediction. Unlike RNNs and CNNs, the Transformer is based on a completely different approach called attention.

The key concept of the Transformer is attention, which allows the network to focus on specific parts of the input sequence that are relevant for performing computations. Attention is based on the idea of assigning weights to different parts of the sequence based on their relevance to the task at hand. This allows for capturing long-term relationships and modeling dependencies between distant elements in the sequence.

The Transformer consists of a stack of attention layers, known as attention blocks. Each attention block consists of multiple attention heads, which calculate attention weights in parallel and learn higher-quality representations. In addition to the attention layers, the Transformer also includes feed-forward layers, which help capture non-linear relationships in the data.

In the image below the structure of the transformer can be seen in a more visual way:

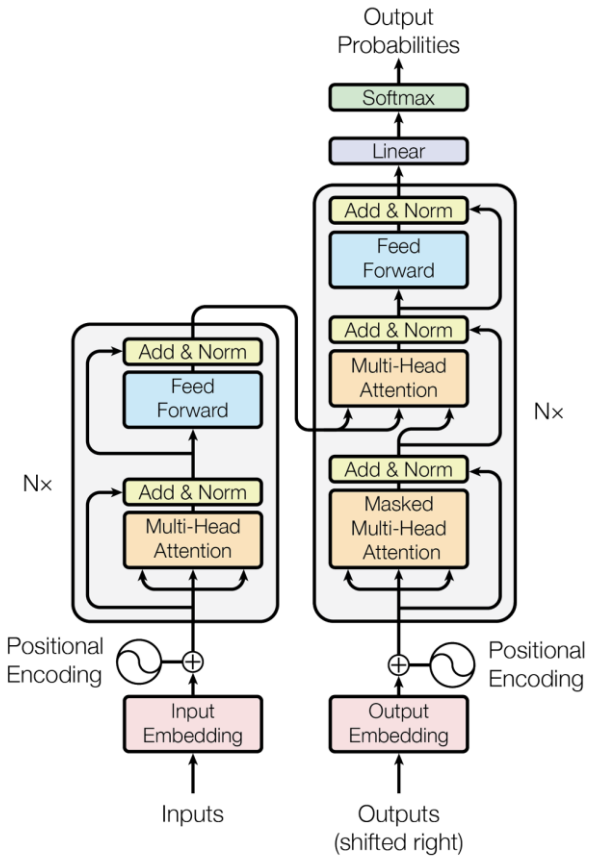

Figure 3. Transformer structure. Source: (Vaswani, et al., 2017)

One of the main advantages of the Transformer is its ability to process entire sequences of data in parallel, making it highly efficient and suitable for tasks involving long time series. Furthermore, the Transformer can capture long-term dependency relationships without suffering from the vanishing or exploding gradient problem that can affect RNNs.

In the context of demand prediction in time series, the Transformer can learn complex patterns and capture long-term dependencies among the input data. It can model non-linear relationships and capture interactions between different components of the time series.

Although the Transformer has proven to be a powerful architecture for time series analysis and prediction, its success largely depends on the quantity and quality of the available training data. Training the Transformer involves adjusting the weights of the network's connections using optimization techniques such as gradient descent. Additionally, the use of techniques such as regularization and hyperparameter tuning is important for achieving optimal model performance.

3. Methodology

3.1. Problem definition

The problem we are trying to solve is achieving the best possible accuracy (or achieving the lowest possible loss) for future predictions of time series data, using historical data related in some way with the predictions we are trying to get.

To be able to measure which predictions might be closer to the ground truth of the data, we would need to define a certain error metric. This one will be the mean square error (MSE).

The formula of the MSE is the following:

$$MSE = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2$$

Where:

-n=Number of predictions or observations

-y=Actual value of the data point (ground truth)

-$\hat{y}$=Forecast

Moreover, the forecast horizon or possible horizons must be defined. These will be one, five, ten, fifteen and twenty days. Each of these horizons will be predicted and compared between different prediction models.

Lastly, we must establish which inputs will be used for the models. These will always be the same for each prediction model, to make fair comparisons between these. Input time series data can be divided into two groups: univariate and multivariate time series. Univariate time series data are a type of data that consists of only one variable recorded at each time step. In this problem case, this data will be the closing price value of the stock that is going to be predicted. On the other hand, for multivariate time series data, several variables (at least more than one) are recorded at each time step. For demand forecasting these could be the demand of various products, their prices or other aspects that could have an impact in the forecasted product demand. For these project other stocks that may be related to the stock that is going to be predicted will also be considered as inputs, thus obtaining a multivariate input for the models.

3.2. Data selection

The ideal case would be using product demand and some more data that might have some correlation with this. But usually, companies do not give open access to their products' demand, therefore more accessible and similar time series data have to be considered for this project. The data that will be used are stock prices, specifically Apple stock closing values. This data has been recorded daily and there is a large amount of historical data accessible on the internet. It was decided to use data from day 1/3/2017 to 3/1/2023, which gives 1550 data points. It is well known that deep learning models need high amounts of data to be trained and be able to generalize for data that may be different to the training data. It was considered that the amount of data used was enough to train these models. The historical data used can be seen in the following graph:

Figure 4. Historical Apple stock daily closing values.

The data was downloaded from Yahoo finance website, and there was no need of using preprocessing tools, because there were no anomalies or missing values. The file obtained from the website is a csv file with the date and opening, high, low, closing, adjusted close and volume values for each day recorded. When using multivariate data, other stocks data will be used. These will also be obtained from the same source and the same amount and timeframe of the data will be used. Some other stocks that will be used are: Microsoft, Amazon, Google, Intel, Nvdia and TSM (Taiwan Semiconductor Manufacturing). These stocks were selected because they might have a certain correlation with the fluctuations and future values of Apple stock. This procedure can be analogously applied to product demand forecasting. The challenge is to find other

external recorded factors or products that may impact the forecasted product demand. By doing this the models have more useful input information and may be able to find patterns in data that could give better demand forecasts.

3.3. Implementation of deep learning models

All deep learning models in this project will be implemented using the Python programming language, with a specific focus on utilizing the PyTorch library for model implementation. PyTorch provides a flexible and efficient interface for building and training neural networks. It is based on the Python programming language. One of the key features of PyTorch is its focus on building models using dynamic computational graphs, which means that the computation graph is built as the code is executed. This allows for greater flexibility in building models and makes debugging and customization easier.

All the models that are going to be used in this work can be implemented using the *nn* module from pytorch. The nn (neural networks) module of PyTorch is a fundamental part of the library that provides a wide range of classes and functions for building and training neural networks. This module is one of the main tools used in PyTorch for constructing deep learning models.

The nn module in PyTorch is based on the concept of a "computational graph" and offers an intuitive and flexible interface for defining and training neural networks. It includes predefined layers that help building the models, classes that make the models definition much easier, loss functions and optimizers and automatic differentiation, which computes gradients efficiently, allowing for easy backpropagation, which is essential for training neural networks.

Next, the implementation of the previously stated models, using Pytorch, will be discussed.

RNN

The main body of the RNN is the class RNN, which inherits from nn.Module class. To define the RNN the nn.RNN class is used. The parameters given to this class are the input size, hidden size of each layer, the number of layers and whether the batch size is the first dimension of the input tensor, which is true in this case.

The input size is always going to be one, because each data point, which is a historical value of the time series, has a dimension of one. After passing the input tensor, which includes

all the batches of input sequences, to the RNN, this gives an output which has the dimension of the hidden size, therefore, a linear layer needs to be applied to this output to obtain the desired output sequence.

When implementing LSTM and GRU, the only change that needs to be made is the nn.RNN class, which has to be now “nn.LSTM” or “nn.GRU”. These classes have the same input parameters and give the same type of output, therefore the forward function and parameters of the RNN do not need to be changed.

The code of these models can be seen in the appendix section 1.1, 1.2 and 1.3.

CNN

Three different CNN models were built and tested, but this work will present and focus on the one that gave the best results. The implementation of CNN has much more “degrees of freedom” when it comes to layers and hyperparameters to choose compared with RNNs. The code of the CNN class can be seen in the appendix, section 1.4. Every layer used in this model is a one-dimensional layer (1d) due to the input data dimensions. Moreover, every convolutional layer has a padding and stride value of 1. These are the layers of the CNN model and the order in which these are applied:

1. Convolutional layer: With number of input channels equal to the input size, which is one, 32 output channels and a kernel size of 3.
2. Batch normalization: With 32 input channels.
3. ReLU activation function.
4. Max Pool layer: With kernel size of 2.
5. Convolutional layer: With 32 input channels, 64 output channels and kernel size of 3.
6. Batch normalization: With 64 input channels.
7. Second ReLU activation function.
8. Max Pool layer: With kernel size of 2.
9. Linear layer: Wich reduces the channels number to 100.
10. Third ReLU activation function.
11. Linear layer: Which connects 100 neurons to the desired size of the output sequence.

Transformer

For the Transformer, only the encoder part was used. Some models for image classification also only use the encoder part of the Transformer. The implementation of the model can be seen in section 1.5 of the appendix.

In the process of building the transformer two classes were used, one was the PositionalEncoder class and the other which is the main class of the model was TransformerModel, both classes inherited from nn module.

Unlike recurrent neural networks (RNNs) or convolutional neural networks (CNNs) that inherently capture sequential information through their architectures, Transformers process input sequences in parallel, making them more efficient for long-range dependencies. However, this parallelism also makes it difficult for Transformers to explicitly encode the positional information of tokens within the sequence. The positional encoder addresses this issue by providing a way to represent the position or order of each token in the input sequence. It assigns unique positional embeddings to each token, which are added to the token embeddings before feeding them into the Transformer model. The most used positional encoder in Transformers is the sine and cosine functions-based positional encoding, which has the following formula:

$$PE_{(pos,2i)} = sin(pos/10000^{2i/d_{\text{model}}})$$
$$PE_{(pos,2i+1)} = cos(pos/10000^{2i/d_{\text{model}}})$$

Where *pos* is the position and *i* is the dimension.

With the positional encoding class defined, the Transformer class could be built. Using this class, the following layers and transformations are applied to the inputs, following this order:

1. Input embedding: A linear layer connects the inputs with the number of neurons corresponding to the dimension of the model specified.
2. Positional encoding: The positional encoder class previously mentioned is applied to the embedded inputs.
3. Encoder layers: The output of the positional encoder is the input of the encoder layers. These layers incorporate the multi-head attention mechanism, feedforward, and normalization layers.
4. Fully connected layer: Lastly, a linear layer connects the encoder layer's output, which has the specified dimension of the model with the desired output sequence length.

3.4. Model training and evaluation

To be able to perform the training and evaluation of the models, the first step is to normalize the historical data. Normalizing data is crucial when using deep learning models for several reasons. It improves convergence, makes the model more stable and robust to variations in the input, can lead to more efficient computation while training and helps the model learn generalizable patterns that can be transferred to unseen examples. There are various normalization methods which are better for specific data types or requirements of the models. The method that is going to be used in this work is the min-max scaling, specifically the data will be scaled between 0 and 1. Min-max scaling can be applied using the following formula:

$$scaled\ value = \frac{value - min}{max - min}$$

Here:

-value: original value of the data.

-min: Minimum value in the dataset considered.

-max: Maximum value in the dataset considered.

Then, historic data is divided into sequences of input data and the expected output predictions that the model should give. The code to do this is in the appendix, section 2.1. To give a visual example of the input and output data, this would be the structure of the two first batches of data used for an input sequence length of 15 and an output length of 5:

Input sequences															Output sequences				
x1	x2	x3	x4	x5	x6	x7	x8	x9	x10	x11	x12	x13	x14	x15	x16	x17	x18	x19	x20
x2	x3	x4	x5	x6	x7	x8	x9	x10	x11	x12	x13	x14	x15	x16	x17	x18	x19	x20	x21

Figure 5. Example of two first batches of input and output data used.

Then these batches of data must be divided into training and testing or evaluation data. It was decided to use 70% of the data to train the model and the other 30% to test it.

After this, the models can be trained and evaluated. To do these, two functions, one for training and another one for evaluating the model, were used. These two functions are inside a loop, which is run for the number of epochs specified, the code can be seen in appendix 2.4.

The training function (appendix section 2.2.) is used to build a model that finds patterns in data and adjusts its weights or parameters, in order to minimize the MSE between predictions and the real value of those predictions (also called ground truth). This can be done using the loss function that calculates the MSE and the optimizer, which in this case was the Adam optimization algorithm. As can be seen, after obtaining the loss between predictions and real values at each epoch, the backward function is called to obtain the gradients and then the optimization algorithm is run.

The evaluation function (appendix section 2.3.) uses the trained model to make predictions and calculates the MSE error of these predictions, which is the metric used to compare models' performances.

4. Results

In the following sections of this work, the results obtained for different models using univariate inputs and multivariate inputs will be presented. It is to be noted that all the mean square errors (MSE) presented were calculated with normalized data between 0 and 1. Therefore, when transforming the data back to the original values, if the MSE is calculated, it will be much higher than the one obtained with normalized data.

4.1. Evaluation of models with univariate time series data

4.1.1. RNN

First the objective was to get predictions for only the next value of the input sequence.

Input sequences of length 30 were used. Using the following hyperparameters we obtained the predictions:

- Hidden size: 51
- Layers: 1
- Epochs: 100
- Learning rate: 0.009

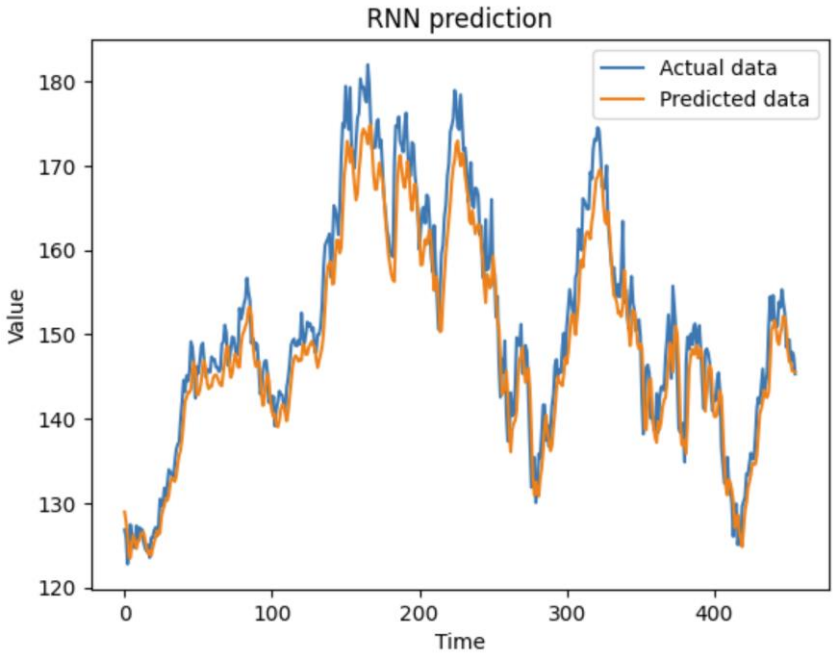

Figure 6. Comparison of RNN predictions and real values.

The loss for the test data was 0.0004.

But predicting only the next value may not be useful in a lot of cases. Typically, predictions of 5, 10 or even 20 values in the future will be needed. An analysis of the RNN was made for those cases. The RNN was trained with different lengths of input sequences and different output sizes, in order to see its performance in each case. As before the MSE was used to compare the results obtained. In the following table it can be seen the MSE for the test data:

	Output size					
Sequence length	**3**	**5**	**8**	**10**	**15**	**20**
20	0.0012	0.0014	0.00167	0.0012	0.00317	0.00326
30	0.00138	0.00144	0.00175	0.00259	0.00282	0.00337
40	0.00113	0.00139	0.00158	0.00193	0.00277	0.00326
50	0.00113	0.00151	0.00168	0.00214	0.00278	0.00345
60	0.00107	0.0016	0.00179	0.00203	0.00285	0.00345

Table 1 . Test data predictions MSE for different input and output lengths using RNN.

The RNN parameters used to obtain these results were:

-Epochs: 100

-Hidden size: 40

-Hidden layers: 1

-Learning rate: 0.01

-Input features (size): 1

The length of the data used was 1550.

It can be observed that for most of the output sizes used, using a sequence length of 40, led to a lower MSE (except for output size of 10). When using an output size of 1, the length sequence that gave the lowest MSE was 30.

Analyzing the MSE for the sequence length of 40 and for the different output sizes:

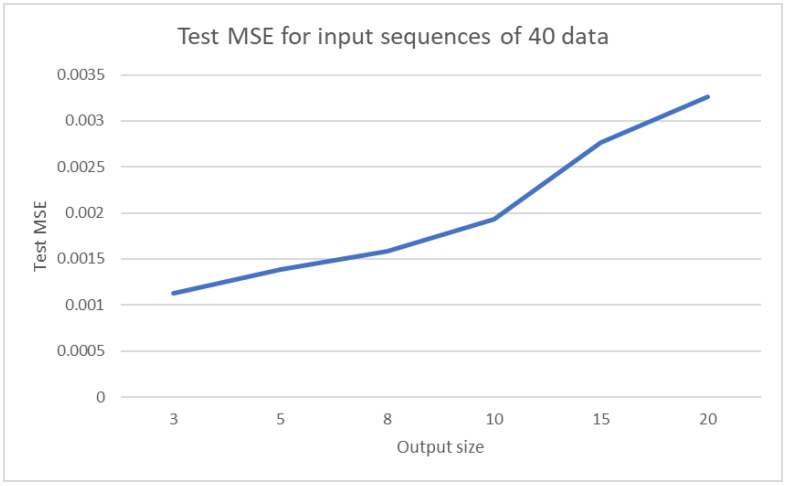

Figure 7. MSE for test data for an input sequence of length 40 and different output sequence length.

As expected, the MSE goes up when the output size is larger.

Now instead of using a simple RNN model to predict, an LSTM will be used. This model has long-short term memory, which allows it to use past data more efficiently when the input sequences used are very long.

For an output size of 1, a simple RNN and an LSTM were trained with: 100 epochs, hidden size of 40, 1 hidden layer, 0.01 learning rate and input sequences of length 40. In the following table we can see the MSE results obtained for the last epoch model:

	Train data	**Test data**
Simple RNN	0.000128	0.000627
LSTM	0.000158	0.000571

Table 2. Comparison of MSE for training and testing data for simple RNN and LSTM model.

In this graph the actual data vs the predictions made by the simple RNN can be seen:

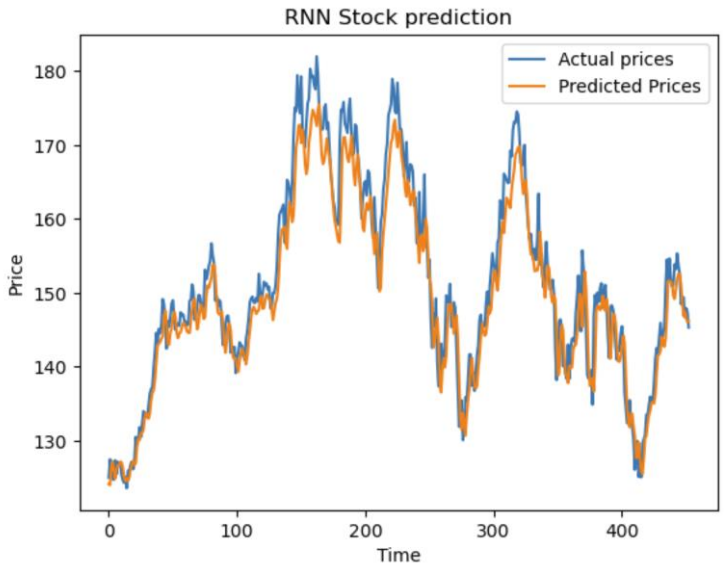

Figure 8. RNN predictions compared with ground truth.

Whereas in this graph the predictions were made using the LSTM model:

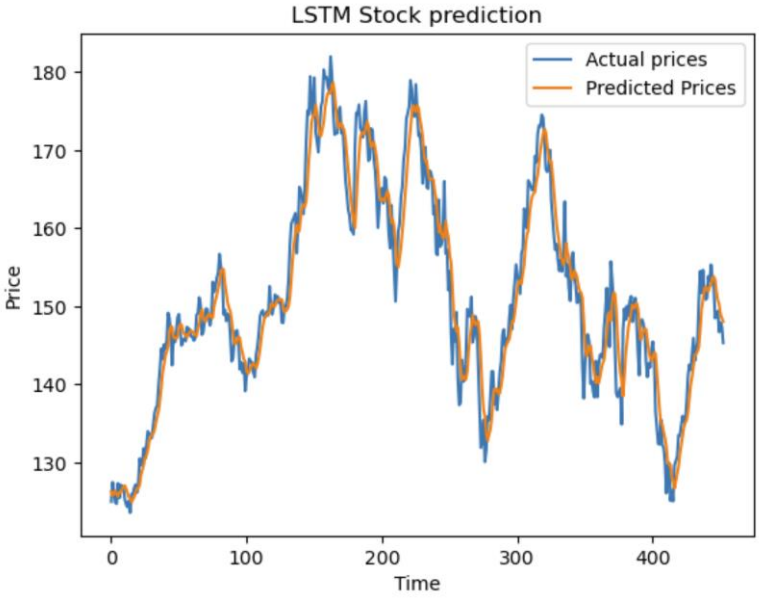

Figure 9. LSTM predictions compared with ground truth.

It can be noted that the LSTM predictions are smoother and more accurate in the high peaks of the data series.

But LSTMs make the difference when the input sequences are very long, to see this larger input length sequences will be tried. If a sequence length of 250 is used, the simple RNN gives a test loss of 0.00858, whereas the LSTM gives a 0.000726 loss.

The improvement from using a LSTM when the amount of historical data is much larger is noticeable. If 3319 data points are used (instead of 1550) the difference between the simple RNN and LSTM is more notorious. The test loss in the last epoch using the simple RNN is 0.00902, however, for the LSTM is 0.00112.

In this graph the predictions for the simple RNN can be seen:

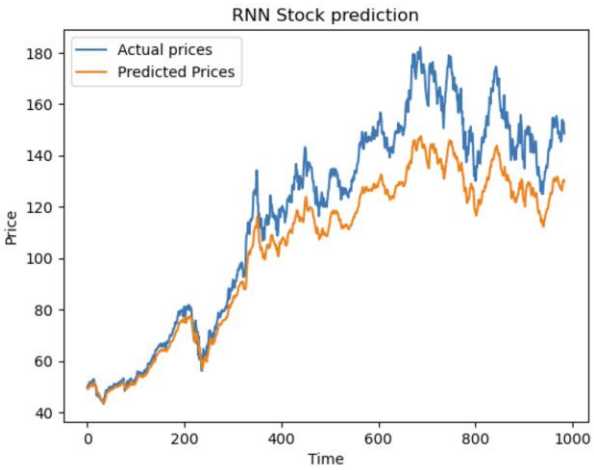

Figure 10. RNN prediction compared with ground truth using more data points.

And these are the predictions for the LSTM:

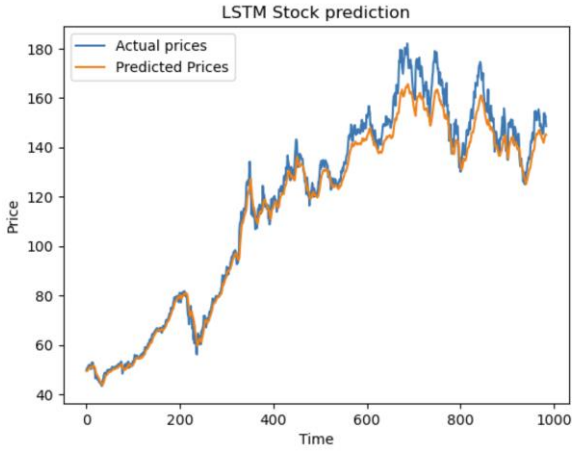

Figure 11. LSTM prediction compared with ground truth using more data points.

When the next 5 values of the input sequence are trying to be predicted, the LSTM has a greater loss for the test data:

-LSTM loss: 0.0031

-Simple RNN loss: 0.0012

This could be happening because the hyperparameters of the LSTM are not optimized for this case. Therefore, an analysis of different hyperparameters would be made for the LSTM, using an output size of 1, and we will suppose that the best parameters and hyperparameters for an output size of 1 would be also the best for greater output sizes. The length of the historical data used to do the analysis was 1550, and the test data length was 456. The results can be seen in the following table:

Sequence lenght	Hidden size	Layers	Learning rate	Epochs	Training loss	Test loss
20	32	1	0.01	100	0.000199	0.00314
30	32	1	0.01	100	0.000198	0.00164
40	32	1	0.01	100	0.000164	0.00108
50	32	1	0.01	100	0.000172	0.0014
40	32	1	0.01	50	0.000294	0.000879
50	32	1	0.01	70	0.000252	0.00063
50	25	1	0.01	70	0.000191	0.00134
50	40	1	0.01	70	0.00024	0.00174
50	50	1	0.01	70	0.000178	0.000668
50	50	1	0.01	100	0.000168	0.000857
50	60	1	0.01	50	0.00023	0.000604
50	60	1	0.01	100	0.000161	0.000771
30	40	1	0.01	100	0.000195	0.000609
30	40	2	0.01	100	0.000275	0.00729
50	50	2	0.01	100	0.000299	0.00783

Table 3. Training and test loss obtained with LSTM for different combinations of hyperparameters.

After seeing these results, the decision was to use a hidden size of 40, 1 layer, 0.01 learning rate and 100 epochs.

Now, using these parameters, the same procedure used for RNN was followed, which was obtaining the MSE loss for the test data, using different input sequence and output lengths. The results obtained can be seen in the table below:

	Output size					
Sequence length	**3**	**5**	**8**	**10**	**15**	**20**
20	0.00351	0.0022	0.00342	0.00526	0.00544	0.00633
30	0.00197	0.0037	0.00381	0.00374	0.00536	0.00577
40	0.00173	0.00487	0.00466	0.00576	0.00596	0.00671
50	0.0039	0.00308	0.00378	0.00574	0.0061	0.00689
60	0.00172	0.00352	0.00491	0.00504	0.00451	0.00672

Table 4. Test data predictions MSE for different input and output lengths using LSTM.

It can be observed that the sequence length that seems to give the lowest loss for the output sizes used is 30. Nevertheless, the MSE values for the LSTM are higher than the ones for the RNN. When the output size increases, this difference is even greater.

In conclusion, the LSTM works better when predicting the next value of the sequence and when the input sequences are large. But, when trying to predict more values in the future time horizon, the RNN makes more accurate predictions.

4.1.2. CNN

For prediction of the next value in the sequence, three different CNN architectures were built, trained, and tested.

For all three CNNs input sequence lengths of 15, 20, 30, 40, 50, 75 and 100 were used and the train loss and test loss were collected in a table.

The first one (CNN1) was formed by the following layers, applied in the following order:

- One-dimension convolutional layer with 1 input channel, 16 output channels, a kernel size of 3 and 0 padding.
- A ReLU activation function.
- Another one-dimension convolutional layer with 16 input channels, 32 output channels, a kernel size of 3 and 0 padding.
- A second ReLu activation function.
- Lastly, a fully contected layer that connects the outputs to one value (which is the output size).

The results obtained using this CNN architecture can be seen in this table:

Seq length	**MSE train loss**	**MSE test loss**
15	0.000373	0.00172
20	0.000473	0.00239
30	0.000298	0.00143
40	0.000432	0.00294
50	0.000236	0.00122
75	0.000322	0.00384
100	0.000218	0.00438

Table 5. Train and test MSE for different input sequence length using CNN1.

The learning rate used was 0.001 and the number of epochs was 500 for every sequence length.

The second CNN (CNN2) was the following:

- A one-dimension convolutional layer with 1 input channel, 16 output channels, a kernel size of 3 and padding of 1.
- A one-dimension batch normalization layer of dimension 16.
- A ReLU activation function.
- A max pool layer of kernel size 2 and stride 2.
- A second one-dimension convolutional layer with 16 input channels, 32 output channels, a kernel size of 3 and padding of 1.
- A one-dimension batch normalization layer of dimension 32.
- A second ReLU activation function.
- Another max pool layer of kernel size 2 and stride 2.
- Finally, a fully contected layer that connects the outputs to one value.

The results obtained using this CNN architecture can be seen in this table:

Seq length	**MSE train loss**	**MSE test loss**
15	0.000183	0.00368
20	0.000184	0.000732
30	0.000334	0.0015
40	0.000207	0.001
50	0.000218	0.00144
75	0.000244	0.00193
100	0.000212	0.00113

Table 6. Train and test MSE for different input sequence length using CNN2.

The learning rate used was 0.001 and the number of epochs was 500 for every sequence length.

The last CNN built (CNN3), has already been escribed in section 3.3. of this project.

The results obtained using this CNN architecture can be seen in this table:

Seq length	**epochs**	**Kernel**	**Padding**	**MSE train loss**	**MSE test loss**
15	200	3	1	0.000149	0.000725
20	500	3	1	0.00009	0.000488
30	400	3	1	0.000076	0.000428
40	450	5	2	0.000134	0.000784
50	500	5	2	7.76E-05	0.000853
75	500	5	2	0.000091	0.00112
100	500	5	2	0.000093	0.00219

Table 7. Train and test MSE for different input sequence length using CNN3.

In this case, for some input sequence lengths, the number of epochs needed to find a good MSE loss was less than 500, moreover with more epochs than the ones used, overfitting would begin to occur. Also, it can be noted that when the kernel size was changed to 5, the padding was increased to 2, this is to guarantee that the shape of the convolutional layers output was the same as the shape of input (the formula used to guarantee this is (kernel size-1)/2). The learning rate used was 0.001 for all length sequences.

Looking at the table, the MSE test loss indicates that the third CNN architecture (CNN3) was the one that obtained better results. The sequence length and parameters that led to the lowest MSE were:

-Sequence length=30

-Epochs=400

-Kernel=3 (in both convolutional layers)

-Padding=1 (in both convolutional layers)

Therefore, we will use these parameters to view next results obtained.

The MSE train and test loss for each epoch can be seen in the following graph:

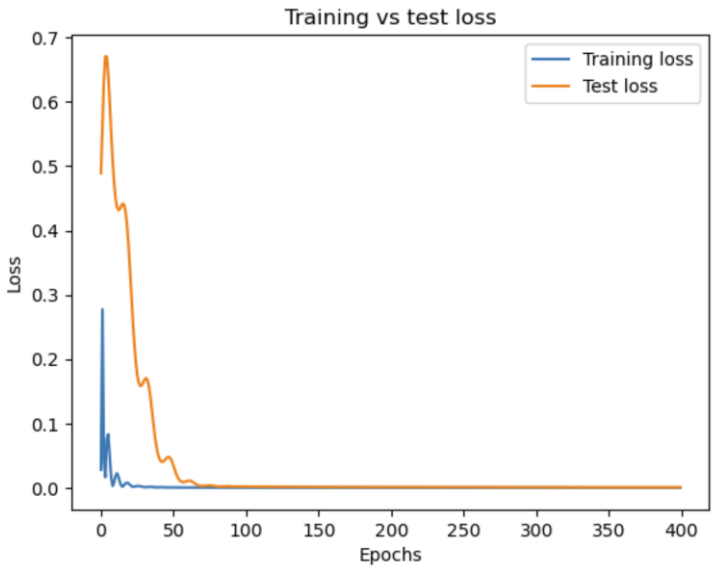

Figure 12. CNN3 test vs training loss at each epoch.

During the first 30 or 40 epochs, the training loss goes down, not in a smooth way, and after epoch 70 the test and training loss become similar, both going down slowly after each epoch.

In the next graph the CNN outputs for the test data can be seen:

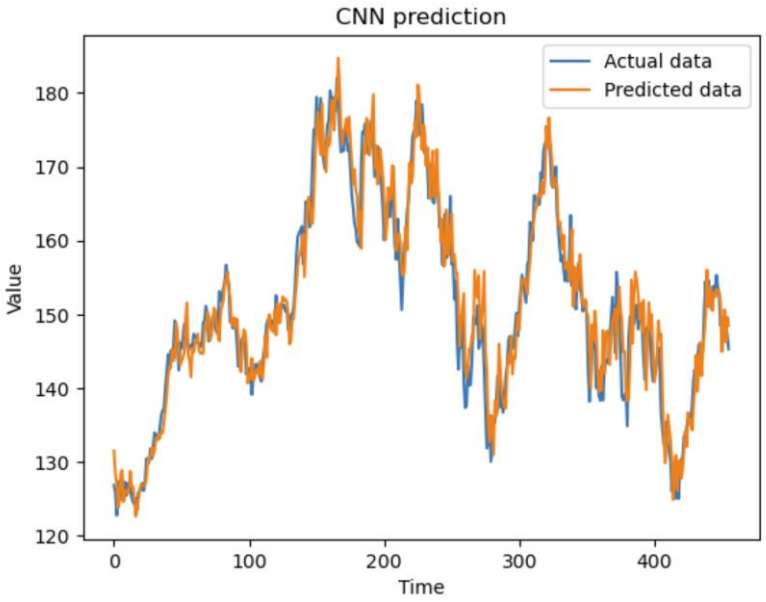

Figure 13. CNN predictions of next time step compared with ground truth.

When predicting for more than one time step into the future, only the third CNN architecture (CNN3) was used. In the table below the test MSE values for different input and output lengths can be seen:

	Output size					
Sequence length	3	5	10	15	20	epochs
15	0.00133	0.00285	0.00448	0.00514	0.00712	350
20	0.0009	0.00223	0.00226	0.00388	0.00588	150
30	0.00152	0.00239	0.00538	0.00631	0.00781	250
40	0.00167	0.00332	0.00461	0.00452	0.00763	250
50	0.00159	0.00256	0.0039	0.00448	0.00499	300
60	0.00157	0.00256	0.00477	0.00577	0.00631	150

Table 8. Test predictions MSE for different input and output lengths using CNN3 model.

The learning rate used for all these different combinations of input and output lengths was 0.001.

4.1.3. Transformer

The next step was using Transformer to see if the error in the predictions could be improved.

A lot of different hyperparameters were tried for predicting the next value of the input sequence, these different parameters can be seen in section 3 of the appendix. One combination of these hyperparameters was selected and used for predicting longer sequences of data. As in previous architectures, different input lengths were used and the mse loss for different output lengths was obtained, the results can be seen in the following table:

	Output size			
Sequence length	5	10	15	20
10	0.00295 (300e)	0.00473 (100e)	0.00594 (90e)	0.00861 (100e)
15	0.00342(370e)	0.00509 (90e)	0.00579 (100e)	0.0082 (100e)
20	0.00325(300e)	0.00542 (100e)	0.00739 (150e)	0.00851 (90e)
30	0.00316 (300 e)	0.00656 (250e)	0.00808 (100e)	0.00887(100e)
40	0.00426(400e)	0.00787 (150e)	0.0092 (100e)	0.0118 (100e)
50	0.00436 (350e)	0.00819 (250e)	0.00879 (110e)	0.0114(100e)

Table 9. Test data MSE for different combinations of input and output lengths using Transformer model.

For an output size of 1, the comparison of predictions and ground truth can be seen in the following graph:

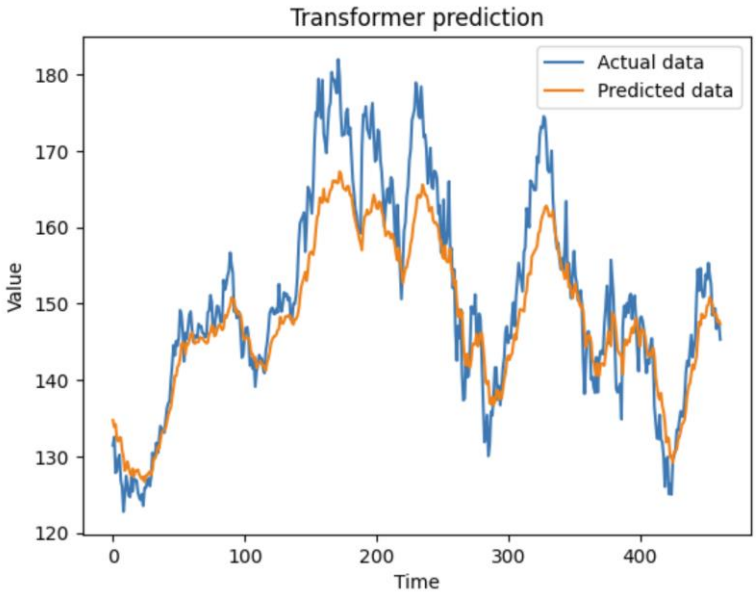

Figure 14. Transformer predictions of next time step compared with ground truth.

4.2. Evaluation of models with multivariate time series data

Until now, only historical data has been used as input for the models. But if more information is given to the model, which in this case could be providing information about other stocks related to the one that is being predicted, maybe improvement in the predictions could be seen.

To do this the RNN, which was the model that gave more consistent and accurate results, was the first tested, using different stocks that may be correlated with apple will be the input data, like Microsoft, Google, Intel, and others.

The mse for the test data changes when using different combinations of these input data can be seen in the following table:

APPL	MSFT	AMZN	GOOGL	INTL	NVDA	TSM	seq_len	test MSE
x	x	x	x	x	x	x	30	0.000337
x	x	x	x	x	x		30	0.000351
x	x	x	x	x			30	0.000231
x	x	x	x				30	0.000236
x	x	x					30	0.000237
x	x						30	0.000227
x							30	0.000966
x	x		x	x		x	30	0.000265
x	x						50	0.000224

Table 10. RNN model MSE for the test data for different combinations of input data and an output length of five.

It can be noted that there is a noticeable decrease in mse, from 0.0097 using just apple historical data, to 0.0002 when we add Microsoft data to the inputs. These values were obtained when predicting the 5 next values into the future.

Nevertheless, when we see some of the predictions, seems that the RNN cannot predict the trend and predictions are not very accurate:

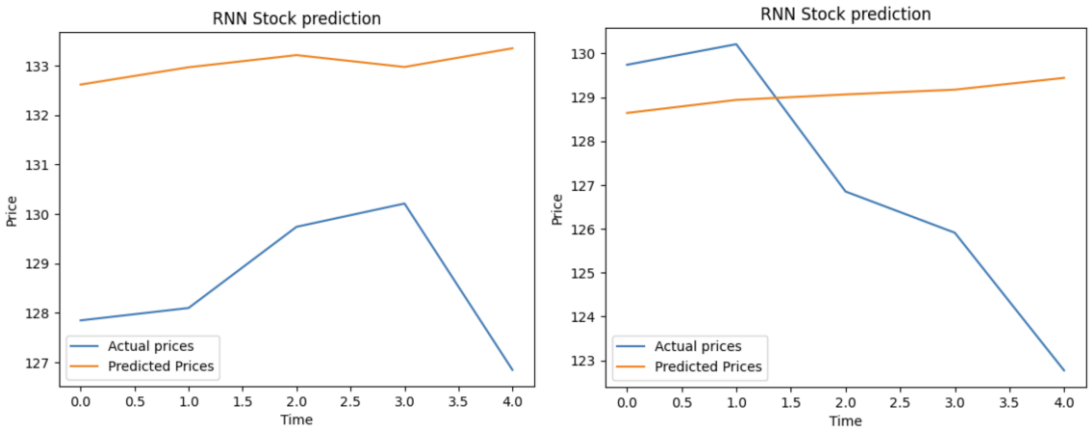

Figure 15. RNN predictions of next five data points using mutlivariate inputs

Besides using information about different stocks, other types of information were also tested. The file that contains the historical data of the stock that is being predicted also contains the opening, highest, lowest and volume of the stock at each day. Without being sure, it could be assumed that giving this information as input to the models could provide better predictions. Therefore, this was tried, and the results can be seen in the following table:

Open	Low	High	Volume	Sequence length	test MSE
x	x	x	x	50	0.001078
	x	x	x	50	0.001042
	x	x		30	0.00106
			x	30	0.001056
x				50	0.001159

Table 11. RNN MSE for test data when using other characteristics of the time series besides from the closing price.

The table below presents the MSE values obtained for the test data, using the marked inputs and the specified sequence length. This sequence length was the one that gave the best results in each particular case. It is to be noted that the MSE obtained using only the closing values of the stock was 0.001087.

Lastly, another approach to the problem was taken. This time the original input data series would be decomposed into trend, seasonality and residual part, and this three time series data would be given to the RNN as inputs. In the graphs below the decomposition of the original time series data can be seen:

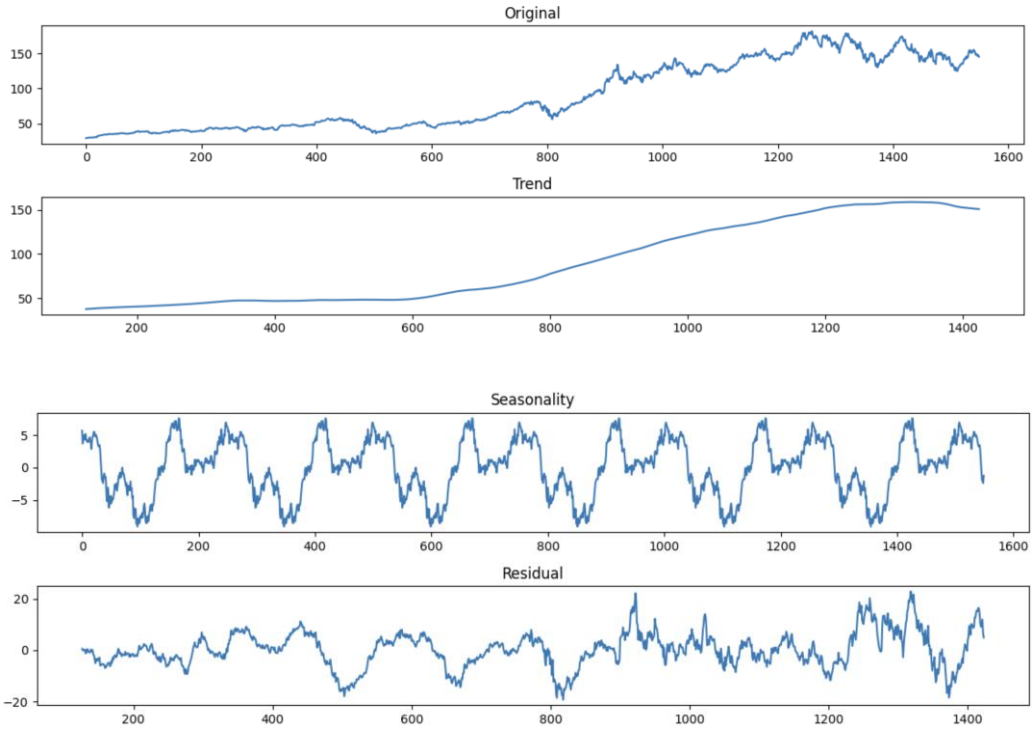

Figure 16. Decomposition of original time series into trend, seasonality and residual part.

Only 5 predictions into the future were made using this new approach, and the error obtained was 0.0042, whereas the error obtained without decomposing the data and for an output size of 5 was 0.00144, therefore no improvement was obtained, in fact, the predictions were worse than before. In the graphs below some of the predictions can be seen:

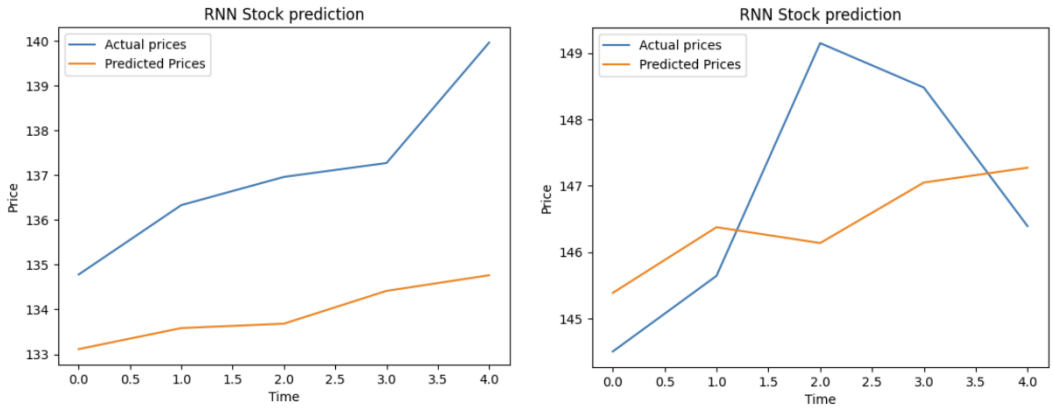

Figure 17. RNN predictions of next five data points using decomposed time series as multivariate input data (1).

Even though the error may be higher, seems that by doing the decomposition, the trend of the predictions is more similar to the trend of the actual data. To compare, this are some of the predictions of the RNN without decomposing the input data, much more predictions were analyzed in order to determine that the trend of the predictions is more similar to the actual data trend:

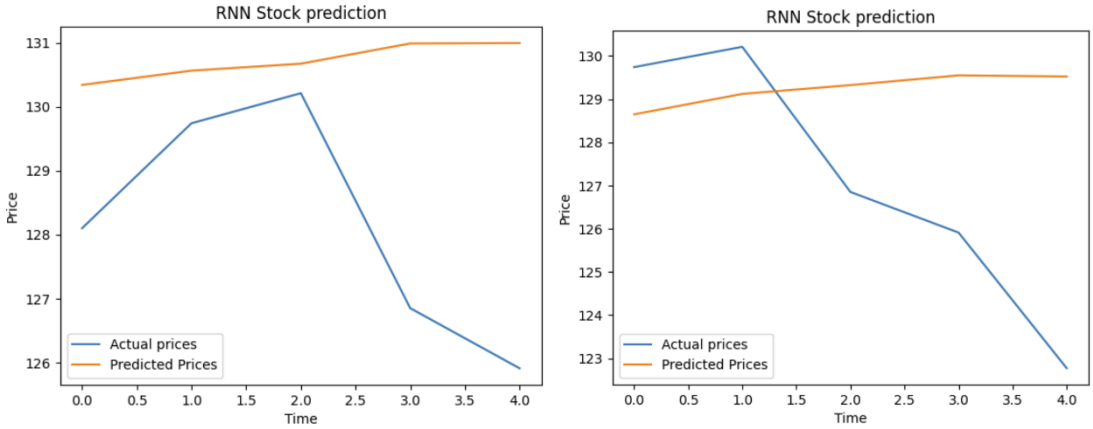

Figure 18. RNN predictions of next five data points using decomposed time series as multivariate input data (2).

After testing the RNN with multivariate data, the same was done using the CNN model (CNN3). In this case only the first approach was used, assuming that it would give the best results, as it did with the RNN model. In the following table the MSE for the test data can be seen for the CNN:

APPL	MSFT	AMZN	GOOGL	INTL	NVDA	TSM	sequence lenght	test MSE
x	x	x	x	x	x	x	30	0.0041
x	x	x	x	x	x		30	0.0071
x	x	x	x	x			20	0.0096
x	x	x	x				20	0.0051
x	x	x					30	0.0043
x	x						20	0.00057
x							20	0.00223

Table 12. CNN3 model MSE for the test data for different combinations of input data and an output length of five.

The parameters and hyperparameters used for each of the different input scenarios were ones that gave the lowest MSE for the test data when using univariate input data. These were:

-Learning rate: 0.001.

-Epochs: 150.

Even though when using Apple and Microsoft the error was significantly reduced, like when using RNN, the predictions do not seem to follow the trend of the actual data. These are some of the predictions:

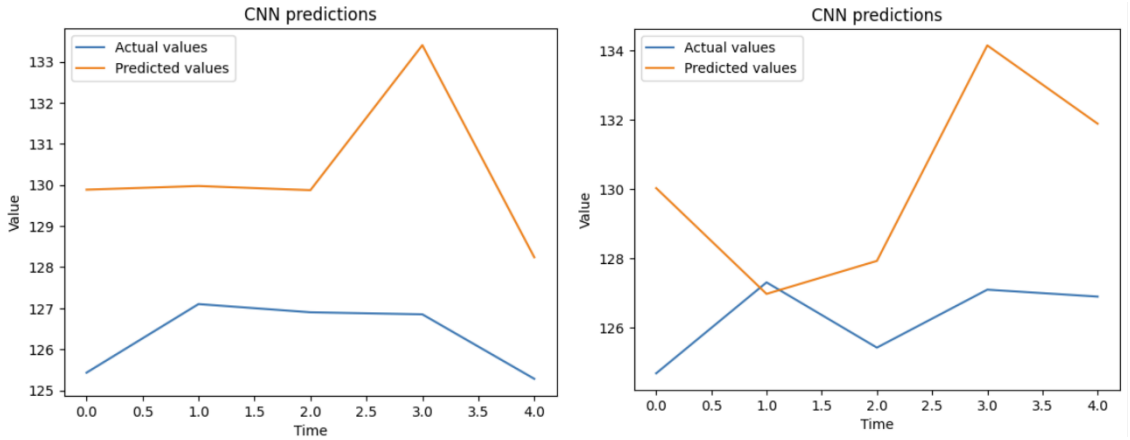

Figure 19. CNN predictions of next five data points using multivariate input data (1).

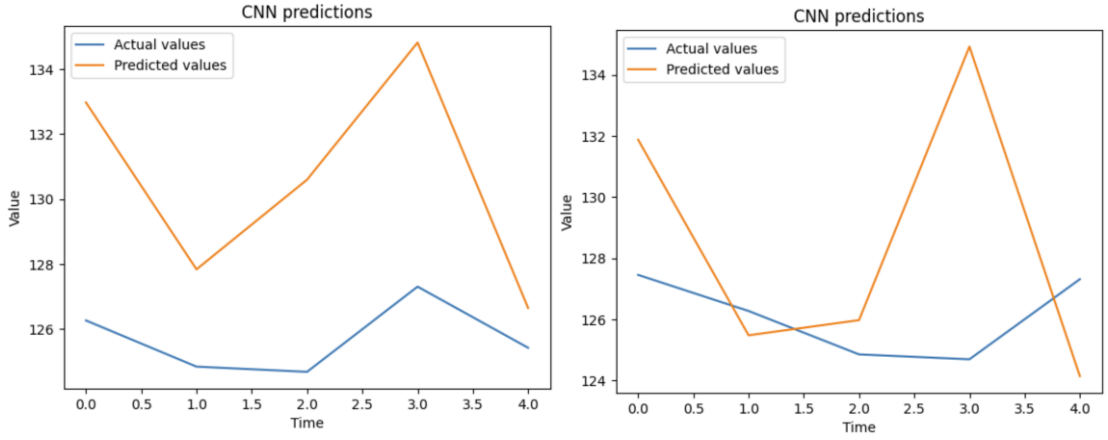

Figure 20. CNN predictions of next five data points using multivariate input data (2).

Lastly, the same was done for the Transformer model, obtaining the following results:

APPL	MSFT	AMZN	GOOGL	INTL	NVDA	TSM	test MSE
x	x	x	x	x	x	x	0.0011
x	x	x	x	x	x		0.00087
x	x	x	x	x			0.0021
x	x	x	x				0.0012
x	x	x					0.0011
x	x						0.0004
x							0.0029

Table 13. Transformer model MSE for the test data for different combinations of input data and an output length of five.

An input sequence length of 10, and 250 epochs were used for each of the different input combinations presented in the table above.

As for the previously analyzed models, the lowest mse is obtained when combining apple and Microsoft input data. Even though the error is low, the predictions do not seem to follow the real trend of the actual data, which also was the issue with CNN and RNN. These are some of the predictions obtained:

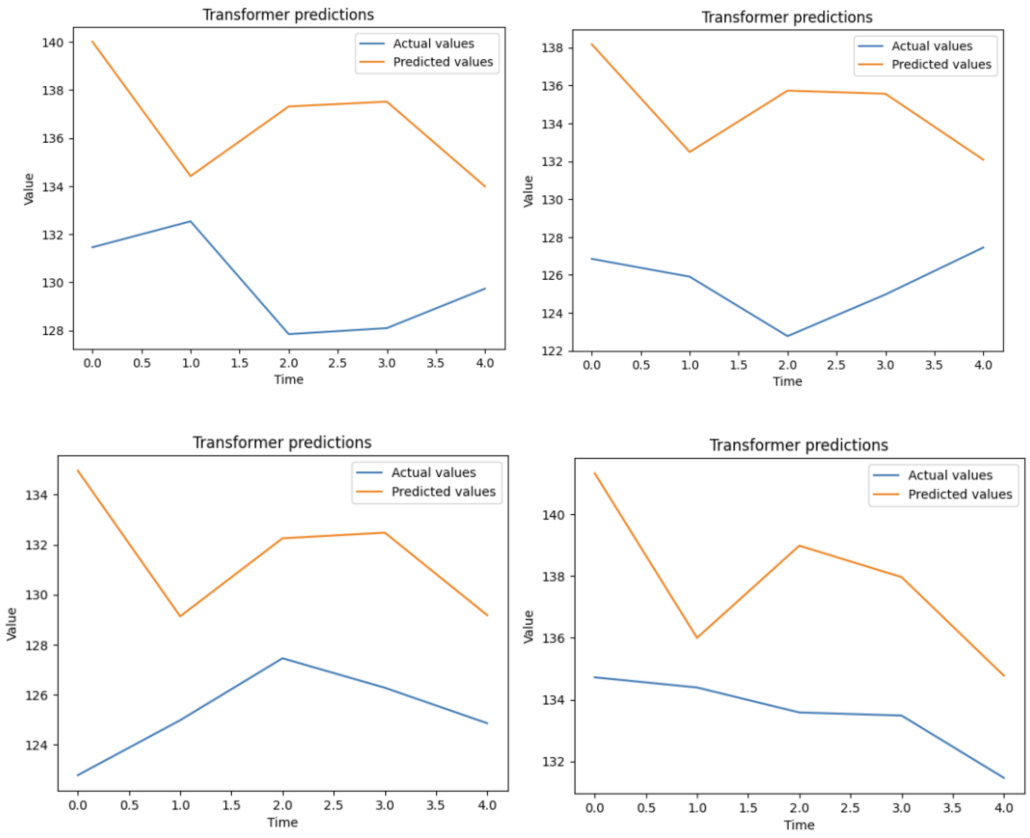

Figure 21. Transformer predictions of next five data points using multivariate input data.

5. Interpretation of results

When the prediction made is only for the next value and univariate input data is used, CNN and RNN have a similar performance. In the next table we can see the test MSE loss for the RNN, LSTM, CNN and Transformer that worked the best for the data used:

Model	**Test MSE loss**
Simple RNN	0.000627
LSTM	0.000571
CNN	0.000428
Transformer	0.00072

Table 14. MSE loss comparison for analyzed models when using univariate input data and predicting the next value of the sequence.

The MSE values are very similar, but the lowest is the one obtained by the CNN and the worst one is obtained by the Transformer.

Nevertheless, when trying to predict more values in the future, results change. The following table presents the MSE loss for the test data for different output sizes. Each model was tested with different input sequences length, but to be able to compare results, the minimum loss obtained for each output size, considering every input sequence length tried, will be the value used in this table:

	Output size				
Model	3	5	10	15	20
Simple RNN	0.00107	0.00139	0.0012	0.00277	0.00326
LSTM	0.00172	0.0022	0.00374	0.00451	0.00577
CNN	0.0009	0.00223	0.00226	0.00388	0.00499
Transformer	-	0.00295	0.00473	0.00579	0.0082

Table 15. MSE loss comparison for analyzed models and different output lengths.

It can be noticed that the RNN was the one that obtained lower MSE for every output size tested, but for the output size of 3.

As mentioned before, when it comes to training time of the models, CNN is the fastest (at least for the architecture made), RNN is also fast but not as much as CNN. LSTM is slower than RNN and CNN, but faster than Transformer. The Transformer model implemented was the slowest, and GPU needed to be used to have a reasonable training time.

It is to be said that even though CNN was faster during the training, more epochs were needed to obtain a good mse error, therefore, it took nearly the same time to train the RNN and the CNN, because less epochs were needed for the RNN training.

When using multivariate input data, in some cases the predictions were more accurate and in other cases not. This illustrates the fact that one of the main challenges when using multivariate input data is finding the data that will improve the accuracy of the models. This is not an obvious task, and some trial and error would be involved in this process.

As described in section 4.2, at first, three different approaches were used for obtaining the multivariate input data. The first was using other stocks closing daily values, the stocks chosen were thought to be correlated to the apple stock, which is the one being predicted. The second was using open, high, low and volume values besides just using the closing value of the stock, and the third was decomposing the original time series into trend, seasonality and residual part and using these time series as inputs for the model.

A summary of the obtained results can be seen in the following table:

Approach	**Test MSE**
Correlated data	0.000224
More information about stock predicted	0.001042
Decomposed data	0.0042

Table 16. RNN MSE for test data when using different multivariate input approaches.

These results were obtained when predicting the next 5 data points of the input sequence. These results were obtained using the RNN model, which was decided to be tested first, because it had a better performance than the other models.

After seeing that using correlated data led to a better performance, it was assumed that this would also happen with the CNN and Transformer model, and this was the only approach taken for using multivariate input data.

The CNN model obtained results much worse than the RNN except for the combination of apple and Microsoft input data, which was the same that led to a lower MSE when using the RNN. In the case of CNN, the difference between the MSE for other combinations of data and the Apple and Microsoft data was much larger than for the RNN case.

Something similar happened for the Transformer model. In this case difference between other combinations MSE and Apple-Microsoft combination was smaller than for the RNN, but larger than for the CNN.

To compare the best results obtained for each model when using multivariate input data, the following table was made:

Model	**Lowest test MSE**
RNN	0.00022
CNN	0.00057
Transformer	0.0004

Table 17. Models lowest MSE for test data when using multivariate input data and predicting five next values of the input sequence.

As mentioned earlier, these MSE were obtained using Apple and Microsoft closing prices as inputs for predicting the next five closing prices of Apple stock. When using univariate input data, the RNN performed the best compared to CNN and Transformer.

6. Conclusions

Deep learning applied to demand forecasting is a very useful tool that allows to take advantage of patterns in data that are difficult to see for humans. Some deep learning models that can be used for this purpose are: RNN, LSTM, GRU, CNN and Transformers. The most used ones for prediction of time series data, like the demand of a product, are RNN, LSTM and GRU.

This independent study tries to give an understanding of each of these models and analysis of which cases they perform the best. RNN seemed to be the best option for the forecasting problem that was trying to be solved. Both when using univariate input data, which were the historical closing prices of the stock that was being predicted and when using multivariate input data, which were other stocks correlated to the one being predicted and this last one itself, RNN gave better results. Nevertheless, results obtained with CNN and Transformer, which were very similar, were not that far from the ones obtained with the RNN.

The fact that the data that was trying to be predicted was stock prices made the forecasting problem really challenging, because this type of data is very difficult to predict, and a lot of factors must be taken into consideration to have a good forecast. This was noticeable when the use of more market information, besides the historical data of the stock that was being predicted, led to a much better performance, and a decrease in the prediction error. But even using this strategy, the forecasts did not seem to be similar or at least have the same trend as the actual values that should have been obtained or “ground truth”. This illustrates the fact that forecasting stock prices is a very challenging task.

The principle of using more information to make forecasts can also be applied to product demand forecasts. The challenge is to figure out which input information could be more useful for the model.

Some interesting future lines of this project could be applying these models to real product demand data, to see if the RNN model would still be the best one and if the models are able to capture patterns and trends in the data better than they did with the data used during this project. In this case as the product demand would be real, considerations about reasonable forecasting horizons and the impact on the accuracy these could have when planning production into the future would be an interesting and necessary task to perform. Moreover, taking into account other products’ demand that may be related to the demand of the forecasted product would be something to analyze, because as it did in this project, it could improve the accuracy

of the forecasts. Lastly other factors like discounts, seasonality, or external economic factors could be given as inputs to the models, in order to study if these reduce the error in the forecasts.

Appendix

1. Models Implementation

1.1. RNN

```
class RNN(nn.Module):
    def __init__(self, input_size, hidden_size, num_layers, output_size):
        super().__init__()
        self.hidden_size=hidden_size
        self.num_layers=num_layers
        self.rnn=nn.RNN(input_size, hidden_size, num_layers, batch_first=True)
        self.linear=nn.Linear(hidden_size, output_size)

    def forward(self, x):
        rnn_out,_=self.rnn(x)
        output=self.linear(rnn_out[:,-1,:])
        return output
```

1.2. LSTM

```
class LSTM(nn.Module):
    def __init__(self, input_size, hidden_size, num_layers, output_size):
        super().__init__()
        self.hidden_size=hidden_size
        self.num_layers=num_layers
        self.rnn=nn.LSTM(input_size, hidden_size, num_layers, batch_first=True)
        self.linear=nn.Linear(hidden_size, output_size)

    def forward(self, x):
        rnn_out,_=self.rnn(x)
        output=self.linear(rnn_out[:,-1,:])
```

```
        return output
```

1.3. GRU

```
class GRU(nn.Module):
    def __init__(self, input_size, hidden_size, num_layers, output_size):
        super().__init__()
        self.hidden_size=hidden_size
        self.num_layers=num_layers
        self.rnn=nn.GRU(input_size, hidden_size, num_layers, batch_first=True)
        self.linear=nn.Linear(hidden_size, output_size)

    def forward(self, x):
        rnn_out,_=self.rnn(x)
        output=self.linear(rnn_out[:,-1,:])
        return output
```

1.4. CNN

```
class CNN(nn.Module):
    def __init__(self, input_size, output_size):
        super().__init__()
        self.conv1=nn.Conv1d(in_channels=input_size,out_channels=32,kernel_size=3, padding=1)
        self.bn1=nn.BatchNorm1d(32)
        self.relu1=nn.ReLU()
        self.pool1=nn.MaxPool1d(kernel_size=2)
        self.conv2=nn.Conv1d(in_channels=32,out_channels=64,kernel_size=3, padding=1)
        self.bn2=nn.BatchNorm1d(64)
        self.relu2=nn.ReLU()
        self.pool2=nn.MaxPool1d(kernel_size=2)
        self.fc1=nn.Linear(320,100)
        self.relu3=nn.ReLU()
```

```
        self.fc2=nn.Linear(100, output_size)

    def forward(self, x):
        x=self.bn1(self.conv1(x))
        x=self.relu1(x)
        x=self.pool1(x)
        x=self.bn2(self.conv2(x))
        x=self.relu2(x)
        x=self.pool2(x)
        x=x.view(x.size(0),-1)
        x=self.fc1(x)
        x=self.relu3(x)
        x=self.fc2(x)
        return x
```

1.5. Transformer

```
class PositionalEncoder(nn.Module):
    def __init__(self, d_model, max_seq_len):
        super().__init__()
        self.d_model=d_model
        self.pos_encoding=self.get_positional_encoding(max_seq_len, d_model).to(device)

    def get_positional_encoding(self, max_seq_len, d_model):
        pos_encoding=torch.zeros(max_seq_len, d_model)
        pos = torch.arange(0, max_seq_len, dtype=torch.float).unsqueeze(1)
        div_term = torch.exp(torch.arange(0, d_model, 2).float()*(-math.log(10000.0)/d_model))
        pos_encoding[:, 0::2]=torch.sin(pos*div_term)
        pos_encoding[:, 1::2]=torch.cos(pos*div_term)
        return pos_encoding.unsqueeze(0)
```

```
    def forward(self, x):
        x=x*math.sqrt(self.d_model)
        x=x+self.pos_encoding[:, :x.size(1), :]
        return x


class TransformerModel(nn.Module): #Only using encoder for the moment
    def __init__(self, input_size, seq_len, d_model, nhead, num_layers, output_size):
        super().__init__()
        self.d_model=d_model
        self.input_emb=nn.Linear(input_size, d_model)
        self.pos_encoder=PositionalEncoder(d_model, seq_len)
        encoder_layers=nn.TransformerEncoderLayer(d_model, nhead, batch_first=True)
#Enconder layer
        self.transformer_encoder=nn.TransformerEncoder(encoder_layers, num_layers,
norm=None) #Stack a number of "num_layers" identical layers
        self.fc=nn.Linear(d_model, output_size) #Linear mapping


    def forward(self, x):
        e=self.input_emb(x)
        e=self.pos_encoder(e)
        x=self.transformer_encoder(e)
        output=self.fc(x)
        return output[:, -1, :]
```

2. Model training and evaluation

2.1. Create input and real output batches of data

```
def create_seq(seq_length, data, output_size):
    x=[]
    y=[]
    for i in range(len(data)-seq_length-output_size):
        x.append(data[i:i+seq_length])
        y.append(data[i+seq_length:i+seq_length+output_size])
    x=np.array(x)
    y=np.array(y)
    return x, y
```

2.2. Model training function

```
def train(model, inputs, real_output, optimizer, criterion):
    model.train()
    optimizer.zero_grad()
    outputs=model(inputs)
    loss=criterion(outputs.squeeze(), real_output)
    keep_loss=loss.item()
    loss.backward()
    optimizer.step()
    return keep_loss
```

2.3. Model evaluation function

```
def predict(model, test_inputs_ev, y_test_ev, criterion):
    model.eval()
    with torch.no_grad():
        predictions=model(test_inputs_ev)
```

```
        loss=criterion(predictions.squeeze(), y_test_ev)
        loss=loss.item()
    return loss, predictions
```

2.4. Training and testing loop

```
for epoch in range(num_epochs):
    loss=train(model, inputs, real_output, optimizer, criterion)
    train_loss.append(loss)
    if epoch%10==0 or epoch==num_epochs-1:
        print("Epoch ",epoch)
        print("Training Loss: ", loss)
    #Predictions
    loss, predictions=predict(model, test_inputs_ev, y_test_ev, criterion)
    test_loss.append(loss)
    if epoch%10==0 or epoch==num_epochs-1:
        print("Test loss: ", loss)
```

3. Transformer hyperparameter evaluation

Input sequence length	d_model	# heads	layers	learning rate	epochs	Train MSE	Test MSE
10	32	1	1	0.0001	500	0.00011	0.00398
10	32	2	1	0.0001	500	0.000104	0.00082
10	32	4	1	0.0001	500	0.000107	0.00272
10	32	8	1	0.0001	500	0.000101	0.00199
10	32	1	2	0.0001	500	0.000109	0.00136
10	32	2	2	0.0001	500	0.0001	0.00237
10	32	4	2	0.0001	500	0.000146	0.00274
10	32	8	2	0.0001	500	0.0001112	0.00137
10	64	1	1	0.0001	500	0.0001	0.00311
10	64	2	1	0.0001	500	0.000102	0.00148
10	64	8	1	0.0001	500	0.000099	0.00216
10	64	1	2	0.0001	500	0.000101	0.00151
10	64	4	2	0.0001	500	0.0000997	0.00109
10	64	8	2	0.0001	500	0.000103	0.00158
10	64	1	4	0.0001	500	0.000101	0.00072
10	64	2	4	0.0001	500	0.000106	0.00085

10	64	8	4	0.0001	500	0.0001	0.00156
10	128	1	1	0.0001	500	0.000099	0.00179
10	128	4	1	0.0001	500	0.000097	0.00101
10	128	1	2	0.0001	500	0.000099	0.00105
10	128	4	2	0.0001	500	0.000105	0.00119
10	128	1	4	0.0001	500	0.0000986	0.00133
10	128	2	4	0.0001	500	0.000103	0.00101
10	128	8	4	0.0001	500	0.0000995	0.0021
10	256	1	1	0.0001	500	0.000099	0.00137
10	256	2	1	0.0001	500	0.000102	0.00257
10	256	4	1	0.0001	500	0.0000975	0.00103
10	256	1	2	0.0001	500	0.000101	0.00133
10	256	2	2	0.0001	500	0.000102	0.00109
10	256	8	2	0.0001	500	0.000098	0.0016
10	256	1	4	0.0001	400	0.000102	0.0009
10	256	4	4	0.0001	400	0.000106	0.00137
10	256	8	4	0.0001	400	0.000114	0.00099
10	512	1	1	0.0001	500	0.000105	0.00182
10	512	2	1	0.0001	500	0.0000989	0.0012
10	512	1	2	0.0001	500	0.000124	0.00126
10	512	4	2	0.0001	400	0.0000994	0.00091
10	512	1	4	0.0001	500	0.000119	0.00181
10	512	2	4	0.0001	500	0.000105	0.00239
10	512	8	4	0.0001	500	0.000149	0.00131
15	64	1	4	0.0001	500	0.00012	0.00165
15	64	2	4	0.0001	400	0.00011	0.00087
15	256	1	4	0.0001	500	0.0000988	0.00139
15	256	8	4	0.0001	500	0.000115	0.00087
20	64	1	4	0.0001	500	0.000103	0.00116
20	64	2	4	0.0001	500	0.000103	0.00187
20	256	1	4	0.0001	500	0.0001	0.00089
20	256	8	4	0.0001	500	0.0000994	0.00189
30	64	1	4	0.0001	500	0.000101	0.00179
30	64	2	4	0.0001	500	0.000106	0.00122
30	128	1	4	0.0001	200	0.000123	0.00096
30	256	1	4	0.0001	500	0.000102	0.00145
30	256	8	4	0.0001	500	0.0000996	0.00165
40	64	1	4	0.0001	500	0.000101	0.00176
40	64	2	4	0.0001	500	0.000102	0.00168
40	128	1	4	0.0001	500	0.000102	0.00225
40	128	2	4	0.0001	500	0.000102	0.00121
50	64	1	4	0.0001	500	0.000103	0.00201
50	64	2	4	0.0001	500	0.000101	0.00182
50	128	1	4	0.0001	500	0.000103	0.00138
50	128	2	4	0.0001	500	0.000103	0.00122

3. Input data processing

3.1. Univariate input data

```
#Load data
data = pd.read_csv('...')
data=data.dropna() #eliminate missing or null values
prices = data['Close'].values
#Show data
print(len(prices))
plt.plot(prices)
plt.show()
#Normalize data
min_price=np.min(prices)
max_price=np.max(prices)
prices=(prices-min_price)/(max_price-min_price)
#Create batches of data
def create_seq(seq_length, data, output_size):
    x=[]
    y=[]
    for i in range(len(data)-seq_length-output_size):
        x.append(data[i:i+seq_length])
        y.append(data[i+seq_length:i+seq_length+output_size])
    x=np.array(x)
    y=np.array(y)
    return x, y
seq_length=20
output_size=5
x1,y=create_seq(seq_length, prices, output_size)
#Divide data into training and test
train_size=int(len(x)*0.7)
x_train=x[:train_size]
```

```
y_train=y[:train_size]
x_test=x[train_size:]
y_test=y[train_size:]
```

3.2. Multivariate input data

```
appl = pd.read_csv('…/AAPL2.csv')
appl=appl.dropna() #eliminate missing or null values
appl= appl['Close'].values

msft=pd.read_csv('… /MSFT.csv')
msft=msft.dropna()
msft=msft['Close'].values

amzn=pd.read_csv('…/AMZN.csv')
amzn=amzn.dropna()
amzn=amzn['Close'].values

googl=pd.read_csv('…/GOOGL.csv')
googl=googl.dropna()
googl=googl['Close'].values

intc=pd.read_csv('…/INTC.csv')
intc=intc.dropna()
intc=intc['Close'].values

nvda=pd.read_csv('…/NVDA.csv')
nvda=nvda.dropna()
nvda=nvda['Close'].values

tsm=pd.read_csv('…/TSM.csv')
```

```
tsm=googl.dropna()
tsm=googl['Close'].values

#Show data
print(len(appl))
plt.plot(appl)
plt.title("APPLE")
plt.show()

print(len(msft))
plt.plot(msft)
plt.title("MICROSOFT")
plt.show()

plt.plot(amzn)
plt.title("AMAZON")
plt.show()

plt.plot(googl)
plt.title("GOOGLE")
plt.show()

plt.plot(intc)
plt.title("INTEL")
plt.show()

plt.plot(nvda)
plt.title("NVDIA")
plt.show()

plt.plot(tsm)
```

```
plt.title("TAIWAN SEMICONDUCTOR")
plt.show()
#Normalize data
min=[]
max=[]
min.append(np.min(appl))
max.append(np.max(appl))
min.append(np.min(msft))
max.append(np.max(msft))
min.append(np.min(amzn))
max.append(np.max(amzn))
min.append(np.min(googl))
max.append(np.max(googl))
min.append(np.min(intc))
max.append(np.max(intc))
min.append(np.min(nvda))
max.append(np.max(nvda))
min.append(np.min(tsm))
max.append(np.max(tsm))

min_price=np.min(min)
max_price=np.max(max)

appl=(appl-min_price)/(max_price-min_price)
msft=(msft-min_price)/(max_price-min_price)
amzn=(amzn-min_price)/(max_price-min_price)
googl=(googl-min_price)/(max_price-min_price)
intc=(intc-min_price)/(max_price-min_price)
nvda=(nvda-min_price)/(max_price-min_price)
tsm=(tsm-min_price)/(max_price-min_price)
#Create batches of data
```

```
def create_seq(seq_length, data, output_size):
    x=[]
    y=[]
    for i in range(len(data)-seq_length-output_size):
        x.append(data[i:i+seq_length])
        y.append(data[i+seq_length:i+seq_length+output_size])
    x=np.array(x)
    y=np.array(y)
    return x, y
seq_length=20
output_size=5
x1,y=create_seq(seq_length, appl, output_size)
x2,_=create_seq(seq_length,msft, output_size)
x3,_=create_seq(seq_length, amzn, output_size)
x4,_=create_seq(seq_length, googl, output_size)
x5,_=create_seq(seq_length, intc, output_size)
x6,_=create_seq(seq_length, nvda, output_size)
x7,_=create_seq(seq_length, tsm, output_size)
x=np.stack([x1, x2, x3, x4, x5, x6, x7], axis=-1)
#Divide data into training and test
train_size=int(len(x)*0.7)
x_train=x[:train_size]
y_train=y[:train_size]
x_test=x[train_size:]
y_test=y[train_size:]
```

4. Plots

4.1. Training vs test loss

```
#Plot train and test loss


plt.plot(range(len(train_loss)), train_loss, label="Training loss")
plt.plot(range(len(test_loss)), test_loss, label="Test loss")
plt.xlabel("Epochs")
plt.ylabel("Loss")
plt.title("Training vs test loss")
plt.legend()
plt.show()
```

4.2. Predictions for output size of 1 or next time step

```
plt.plot(range(len(y_test)), y_test, label="Actual data")
plt.plot(range(len(predictions)), predictions, label="Predicted data")
plt.xlabel("Time")
plt.ylabel("Value")
plt.title("Transformer prediction")
plt.legend()
plt.show()
```

4.3. Predictions for output sizes greater than 1

```
#Plot results for first 50 sequences
for i in range (50):
    print("Sequence ",i+1)
    plt.plot(range(len(y_test[i,:])), y_test[i,:], label="Actual values")
    plt.plot(range(len(predictions[i,:])), predictions[i,:], label="Predicted values")
    plt.xlabel("Time")
```

```
plt.ylabel("Value")
plt.title("Transformer predictions")
plt.legend()
plt.show()
```

References

Ankit, U. (2022, June 28). *Transformer Neural Networks: A Step-by-Step Breakdown*. Retrieved from Built in: https://builtin.com/artificial-intelligence/transformer-neural-network

Brian, L. (2023, April 5). *Can Transformers Improve Time Series Prediction?* Retrieved from applike: https://how.makeanapplike.com/can-transformers-improve-time-series-prediction-256b6696c45a

Brownlee, J. (2018, November 12). *How to Develop Convolutional Neural Network Models for Time Series Forecasting*. Retrieved from Machine Learning Mastery: https://machinelearningmastery.com/how-to-develop-convolutional-neural-network-models-for-time-series-forecasting/

Dancker, J. (2022, December 26). *A Brief Introduction to Recurrent Neural Networks*. Retrieved from Towards Data Science: https://towardsdatascience.com/a-brief-introduction-to-recurrent-neural-networks-638f64a61ff4

Ludvigsen, K. G. (2022, May 12). *How to make a Transformer for time series forecasting with PyTorch.* Retrieved from Towards Data Science: https://towardsdatascience.com/how-to-make-a-pytorch-transformer-for-time-series-forecasting-69e073d4061e

Mishra, M. (2020, August 26). *Convolutional Networks, Explained*. Retrieved from Towards data science: https://towardsdatascience.com/convolutional-neural-networks-explained-9cc5188c4939

Onnen, H. (2021, October 31). *Temporal Loops: Intro to Recurrent Neural Networks for Time Series Forecasting in Python*. Retrieved from Towards Data Science: https://towardsdatascience.com/temporal-loops-intro-to-recurrent-neural-networks-for-time-series-forecasting-in-python-b0398963dc1f

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A., . . . Polosukhin, I. (2017, June 12). *Attention is all you need.* Retrieved from arxiv: https://arxiv.org/abs/1706.03762

VENKATACHALAM, M. (2019, February 28). *gotensor*. Retrieved from Recurrent Neural Networks – Remembering what's important: https://gotensor.com/2019/02/28/recurrent-neural-networks-remembering-whats-important/

Vishwas, B. V., & Patel, A. (2020). *Hands-on Time Series Analysis with Python.* Apress.

Wen, Q., Zhou, T., Zhang, C., Chen, W., Ma, Z., Yan, J., & Sun, L. (2022, February 15). *Transformers in Time Series: A Survey.* Retrieved from arxiv: https://arxiv.org/abs/2202.07125

Wibawa, A. P., Utama, A. B., Elmunsyah, H., Pujianto, U., Dwiyanto, F. A., & Hernandez, L. (2022, April 26). *Time-series analysis with smoothed Convolutional Neural Network*. Retrieved from PMC: https://www.ncbi.nlm.nih.gov/pmc/articles/PMC9040363/#:~:text=CNN%20is%20suitable%20for%20forecasting,time%2Dseries%20%5B14%5D.

Wikipedia. (2008, October 1). *https://en.wikipedia.org/wiki/Recurrent_neural_network*. Retrieved from Wikipedia: https://en.wikipedia.org/wiki/Recurrent_neural_network

Wikipedia. (2014, March 10). *Colvolutional neural network*. Retrieved from Wikipedia: https://en.wikipedia.org/wiki/Convolutional_neural_network

Wikipedia. (2019, January 13). *PyTorch*. Retrieved from Wikipedia: https://en.wikipedia.org/wiki/PyTorch

Wikipedia. (2020, July 9). *Transformer (machine learning modelI)*. Retrieved from Wikipedia: https://en.wikipedia.org/wiki/Transformer_(machine_learning_model)

Zeng, A., Chen, M., Zhang, L., & Xu, Q. (2022, May 26). *Are Transformers Effective for Time Series Forecasting?* Retrieved from arxiv: https://arxiv.org/abs/2205.13504

Figure index

Table index