Wrocław University
of Science and Technology

Faculty of Pure and Applied Mathematics
Field of study: Applied Mathematics
Specialty: Data Engineering

Master's Thesis

OPTIMAL TRADE EXECUTION USING REINFORCEMENT LEARNING

Błażej Jaskólski

keywords:
Reinforcement learning, machine learning, stock market, trading, optimization, linear regression, random forest, recurrent neural networks, agent, environment

short summary:
This paper is devoted to the study of the problem of optimal trade execution. Machine learning and reinforcement learning models were used to analyze and predict prices and trends. The study was conducted on instruments from Warsaw Stock Exchange.

Supervisor	dr inż. Rafał Połoczański	....	....
	Title/degree/name and surname	grade	signature

*For the purposes of archival thesis qualified to:**
- *a) category A (perpetual files)*
- *b) category BE 50 (subject to expertise after 50 years)*

** delete as appropriate*

stamp of the faculty

Wrocław, 2024

Contents

Introduction

The stock market is a key component of the global economy. It is a place where buyers and sellers exchange shares of different public companies. This market sees constant price changes based on supply and demand. Traders aim to buy low and sell high, but due to the market's unpredictability and volatility, it is very difficult to execute profitable trades. These days, many people utilize different models and algorithms to evaluate data and forecast market movements to aid in their trading decisions [9]. There are also many papers written about this topic, exploring it from many different perspectives [21].

Machine Learning (ML) is a subset of artificial intelligence (AI) that aims to build systems that can learn and make decisions based on provided data [2]. It includes many different algorithms, each with specific advantages and uses, like decision trees, neural networks, or Linear Regression.

Reinforcement Learning (RL), on the other hand, is a specialized branch of ML that focuses on training agents to make sequences of decisions. These decisions can be influenced by the reward function, in which a user can specify what behaviour is beneficial to the model. This approach allows the agent to learn optimal strategies over time, making it particularly suited for dynamic and complex environments like the stock market [16].

This thesis explores the application of machine and reinforcement learning in optimizing trade execution in the stock market. The first part will tackle some theoretical concepts and nuances about each machine and reinforcement learning model used, supported by graphs and examples to illustrate their functionality and effectiveness. The next part delves into using these simple machine learning models to predict closing prices and trends, providing a foundation for informed trading decisions. Then we transition to reinforcement learning, aiming to enhance existing trading strategies as well as design an agent that learns how to trade using different models and environments. By integrating advanced learning techniques, the goal is to develop strategies that not only react to market conditions but also adapt and evolve, achieving superior performance in trade execution. Lastly, we will talk about the results and possible future work.

Chapter 1

Theory

In this chapter, we will explore the theory behind all of our models. We will talk about what measures will be used to evaluate the performance of each model, as well as explain how these models work using simple graphs and examples.

1.1 Machine Learning algorithms

These days, more and more people use ML in stock forecasting due to the possibility to analyze large amounts of historical data and identify some complex patterns that might be missed otherwise. The usage of particular ML models depends on the specifics of the problem, the data used and many more factors. The following metrics will be used to evaluate the performance of each model:

1. Mean Squared Error (MSE) - an average squared difference between the estimated values and the actual value:

$$\text{MSE} = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2 \tag{1.1}$$

2. Mean Absolute Error (MAE) - a measure of errors between paired observations expressing the same phenomenon:

$$\text{MAE} = \frac{1}{n}\sum_{i=1}^{n}|y_i - \hat{y}_i| \tag{1.2}$$

3. Coefficient of determination R^2 - relation between the input variable and its prediction.

$$R^2 = 1 - \frac{\sum_{i=1}^{n}(y_i - \hat{y}_i)^2}{\sum_{i=1}^{n}(y_i - \bar{y})^2} \tag{1.3}$$

where:

- y_i: The actual value for the i-th observation.
- $\hat{y}_i$: The predicted value for the i-th observation.
- n: The total number of observations.

1.1.1 Linear Regression

Linear Regression is a statistical model that is used to predict the value of a variable based on the value of another variable or many different variables [10]. In case of the stock forecasting, Linear Regression can be used to predict the closing price using the historical values of 'open', 'volume', or some custom indicators like SMA (Simple Moving Average) or RSI (Relative Strength Index). Linear regression can be expressed by the following equation:

$$Y = \beta_0 + \beta_1 X + \epsilon \tag{1.4}$$

where:

- Y is the dependent variable (the one we try to predict).
- β_0 is the intercept.
- β_1 is the slope coefficient representing the effect of the independent variable X on Y.
- X is the independent variable.
- ϵ is the error term accounting for the variability in Y that cannot be expressed by X.

In case of multiple Linear Regression - if more variables are used to predict a single variable, the model expands, adding additional independent variables:

$$Y = \beta_0 + \sum_{i=1}^{n} \beta_i X_i + \epsilon \tag{1.5}$$

where in this case β_i represents the slope coefficient for its corresponding variable X_i.

While Linear Regression can be a great and simple tool to analyze relations between different variables, it is important to remember to always deeply look into the data and consider not only calculated values and coefficients but also the graphs. A great example of that can be shown using the data sets in Anscombe's quartet [1]:

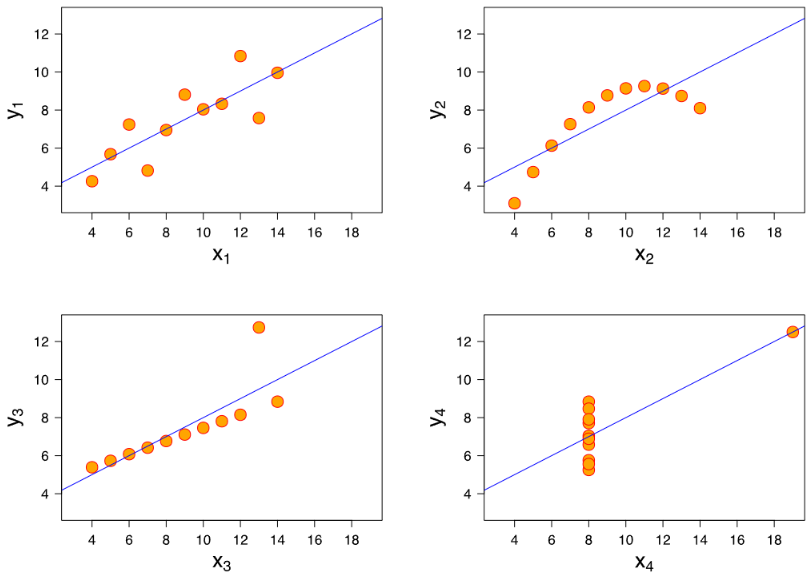


Figure 1.1: Linear Regression's pitfalls shown in Anscombe's quartet data sets. Source: wikipedia.org

These sets are designed to have the same Linear Regression line but have different means and correlations. This shows us that when evaluating the model's performance, we should consider the graphical representation as well as the statistical data, as in this case for different data sets we have an identical regression line.

There is also another very important aspect of Linear Regression: outliers. As shown below, outliers can heavily impact the results of this model.

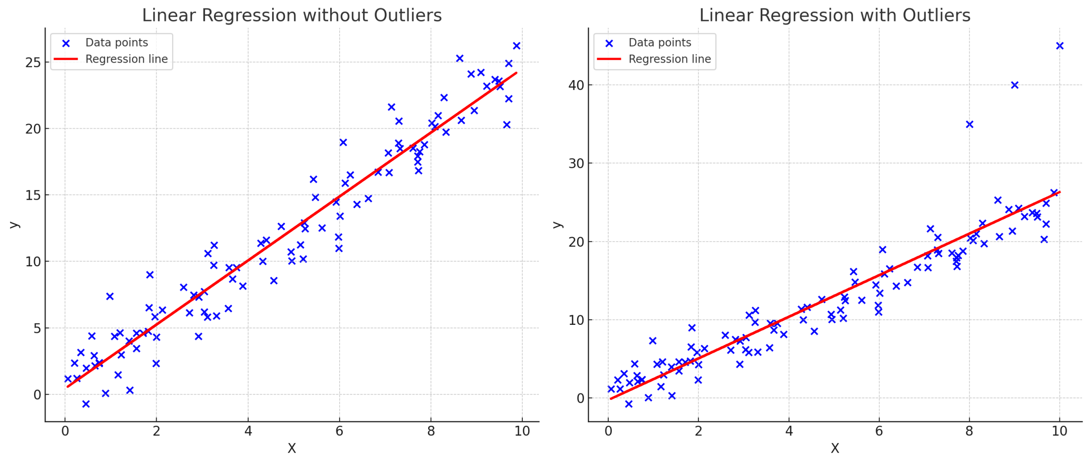


Figure 1.2: Plot showing the impact of outliers on forming the regression line.

1.1.2 Decision Tree Regressor

Decision Tree Regressor is a non-linear ML model used for predictive modeling [3]. It uses recursion to partition the data set into smaller subsets based on the values of the inputs. The partitioning is done through a series of decisions that form a tree-like structure, hence the name. Each node represents a decision based on a feature, each branch represents the outcome of that decision, and each leaf represents the predicted value. We can think of decision trees as a series of 'if-then' rules that guide us to a prediction.

Imagine a decision tree as a series of questions you might ask to make a decision. For example, let's say you want to predict whether it will rain or not. First, we can look at the 'Outlook'; this becomes our root node. From there, we can go to one of three nodes: Sunny, Overcast, and Rain. After choosing the node 'Sunny', we can then ask about the Humidity - if it's 'high', we can say it will not rain, and if the answer is 'normal', then it will. If we choose 'Overcast' when asked about the 'Outlook', we can predict that it will rain, etc. This decision-making process can be visualized using Decision Tree Structure. In this simple example the following tree is made:

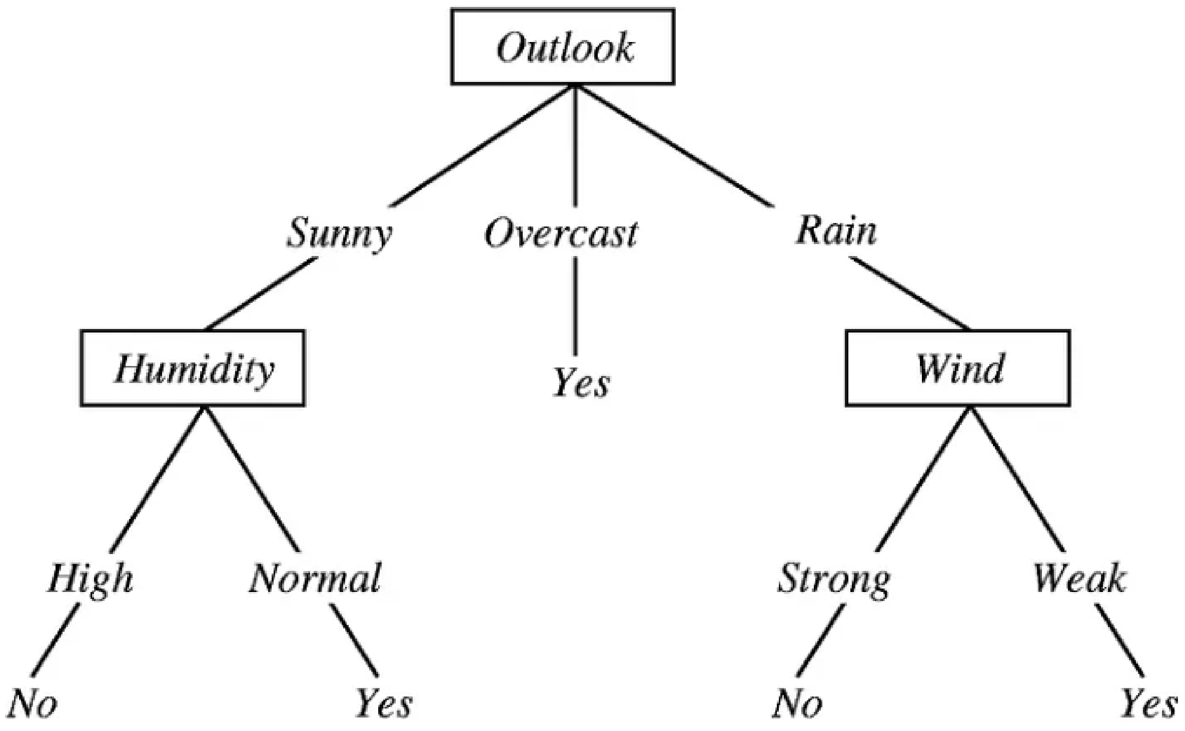


Figure 1.3: Decision Tree Structure of the Decision Tree Regressor model.

Mathematically, a Decision Tree Regressor model can be defined as the following: assume we have a dataset of N observations of the form (x_i, y_i) where y_i is the value of the random variable Y (the output variable), and $x_i = (x_{i1}, x_{i2}, ..., x_{ip})$ is the vector of values of length p, representing the random vector X (the vector of input variables). We denote by $\mathcal{R}$ the $(p+1)$-dimensional space in which the data are located. In the first step, the space $\mathcal{R}$ should be divided into two half-spaces [6]:

$$\mathcal{R}_1(j,s) = \{X \mid X_j \leq s\}, \tag{1.6}$$

$$\mathcal{R}_2(j,s) = \{X \mid X_j > s\}, \tag{1.7}$$

where j is the index of the so-called splitting variable, and s is the splitting point of this variable [6]. This represents the tree branching decision to create new nodes and is done by splitting the observations based on the value of the input variable X_j. To find the optimal split, we need to choose the variable X_j and the point s such that the sum of squared residuals (RSS) is minimized. In other words, we need to find j and s that satisfy the condition [6]:

$$RSS_{min} = \min_{j,s}\left[\min_{c_1}\sum_{x_i\in\mathcal{R}_1(j,s)}(y_i - c_1)^2 + \min_{c_2}\sum_{x_i\in\mathcal{R}_2(j,s)}(y_i - c_2)^2\right], \tag{1.8}$$

where c_k is the estimator of the variable y_i in the k-th node. It is defined as the arithmetic mean of the output variable observations in the k-th node [6]:

$$c_k = \frac{1}{n}\sum_{i=1}^{n} y_i, \tag{1.9}$$

assuming that there are n observations in the k-th node. Equation 1.8 represents the smallest possible sum of errors during the division. The algorithm chooses among all variables X_j (where $j \in \{1, \dots, p\}$) and all splitting points $s \in \mathbb{R}$ the pair (j, s) for which the sum of squared residuals (RSS, eq. 1.8) takes the smallest value. This is equivalent to finding the best division. The above procedure is repeated recursively until the specified stop criterion is met. At this point, new nodes $\mathcal{R}_1$ and $\mathcal{R}_2$ are formed, and the whole group is divided sequentially.

1.1.3 Random Forest

Even though decision trees have many advantages, they have a tendency for overfitting. That is why there can be some situations, in which the model cannot fit perfectly into the test data - even with excellent results on the training data. So new, more advanced models were invented, one of which being the Random Forest model. It is an ensemble learning method that combines multiple decision trees to improve the performance and robustness of the model. The main idea is to create a **forest** of decision trees and use bagging (bootstrap aggregating) on their predictions in order to form more stable and accurate results. It is also less prone to overfitting.

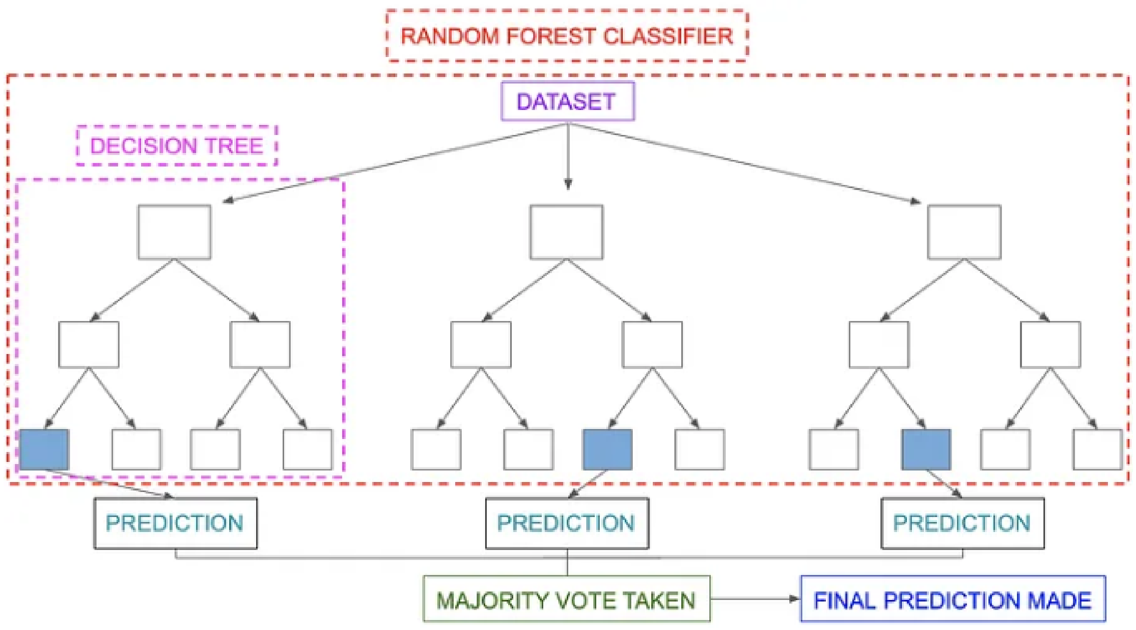


Figure 1.4: Example of a Random Forest. Each of the trees returns an estimator of some variable. Then the values of these estimators are aggregated into one. Source: medium.com

In case of this thesis, the main focus of the Random Forest model will be predicting whether the price of an asset will go up or down, instead of solely focusing on the closing price.

1.2 Recurrent Neural Networks

Recurrent Neural Networks (RNNs) are a class of neural networks designed for processing sequential data. Unlike traditional feedforward neural networks, RNNs have connections that form directed cycles, enabling them to maintain a memory of previous inputs in the sequence [20]. This makes RNNs particularly suitable for tasks where the context provided by prior elements in the sequence is crucial. That is why when working with the stock data and RNNs we will be adding sequences for these models to work correctly. There are many interesting models in this category, but the goal will be to explore three of them.

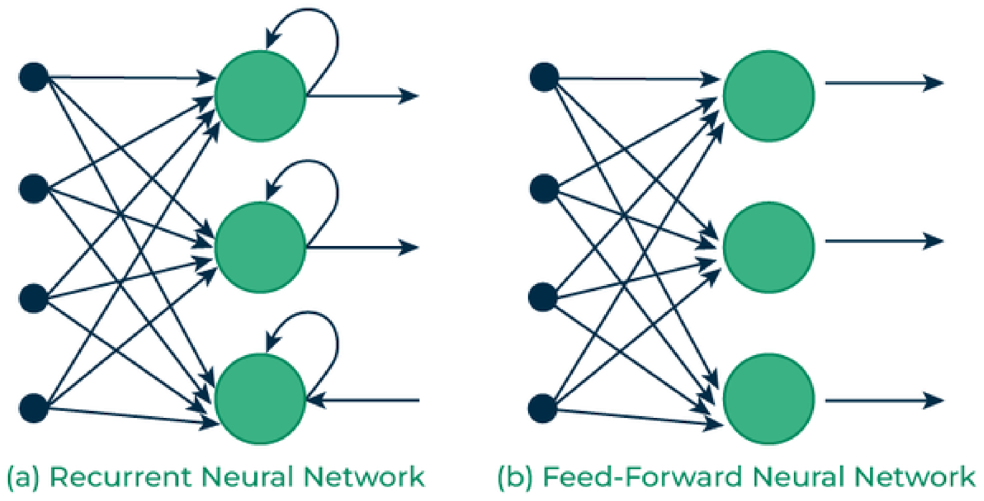


Figure 1.5: Simple diagram showing a comparison between recurrent and feedforward neural networks. Source: geeksforgeeks.org

1.2.1 SimpleRNN

Simple RNNs are often used for basic sequence prediction tasks where the sequences are relatively short and the relationships within the sequence are straightforward. A Simple RNN, also known as a vanilla RNN, maintains a hidden state vector h_t that is updated at each time step based on the current input x_t and the previous hidden state h_{t-1}. The hidden state is calculated using the following equation [20]:

$$h_t = \sigma(W_{xh}x_t + W_{hh}h_{t-1} + b_h) \tag{1.10}$$

where:

- σ is the activation function (e.g., tanh or ReLU),
- W_{xh} is the weight matrix for the input x_t,
- W_{hh} is the weight matrix for the hidden state,
- b_h is the bias term.

The output y_t at time step t is typically computed as:

$$y_t = \phi(W_{hy}h_t + b_y) \tag{1.11}$$

where:

- ϕ is the output activation function (e.g., softmax for classification),
- W_{hy} is the weight matrix for the output,
- b_y is the bias term.

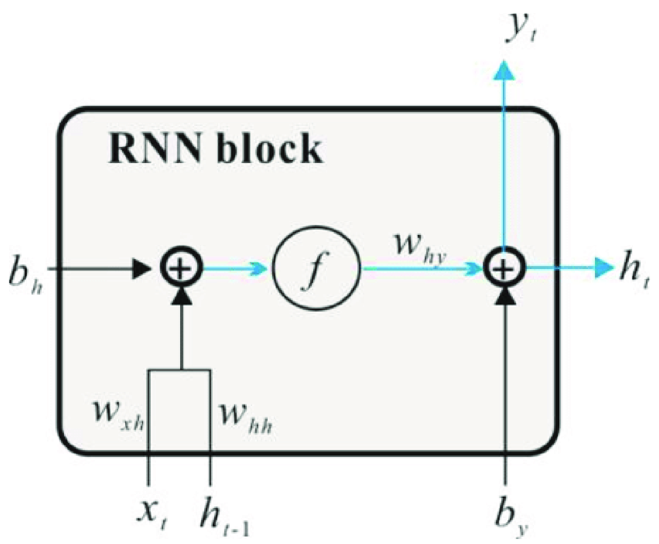


Figure 1.6: SimpleRNN cell structure in hidden layer[20].

1.2.2 LSTM

LSTM networks are a type of RNN specifically designed to address the vanishing gradient problem, which makes training traditional RNNs challenging for long sequences. LSTMs introduce a memory cell c_t and three gates (input gate, forget gate, and output gate) to control the flow of information [20]:

1. **Input gate** i_t:
$$i_t = \sigma(W_{xi}x_t + W_{hi}h_{t-1} + b_i) \tag{1.12}$$
2. **Forget gate** f_t:
$$f_t = \sigma(W_{xf}x_t + W_{hf}h_{t-1} + b_f) \tag{1.13}$$
3. **Output gate** o_t:
$$o_t = \sigma(W_{xo}x_t + W_{ho}h_{t-1} + b_o) \tag{1.14}$$
4. **Memory cell update**:
$$\tilde{c}_t = \tanh(W_{xc}x_t + W_{hc}h_{t-1} + b_c) \tag{1.15}$$
$$c_t = f_t \odot c_{t-1} + i_t \odot \tilde{c}_t \tag{1.16}$$
5. **Hidden state update**:
$$h_t = o_t \odot \tanh(c_t) \tag{1.17}$$

where $\odot$ denotes element-wise multiplication.

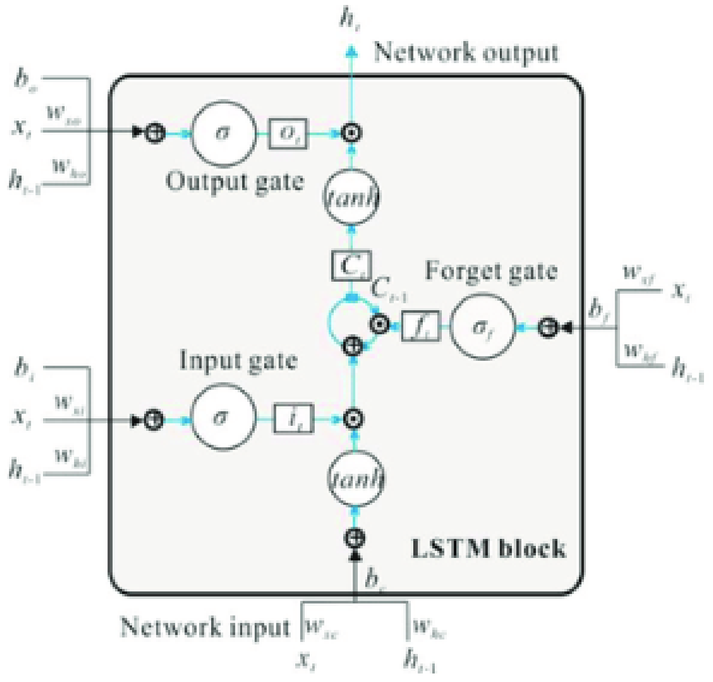


Figure 1.7: LSTM block structure [20].

LSTMs are widely used for tasks that require learning long-term dependencies in the data. Common applications include speech recognition, language translation, text generation, and more complex time series forecasting.

1.2.3 GRU

GRUs are a simpler variant of LSTMs that combine the forget and input gates into a single update gate and merge the cell state and hidden state. This reduces the computational complexity while retaining the ability to capture long-term dependencies [17]:

1. **Update gate** z_t:
$$z_t = \sigma(W_{xz}x_t + W_{hz}h_{t-1} + b_z) \tag{1.18}$$

2. **Reset gate** r_t:
$$r_t = \sigma(W_{xr}x_t + W_{hr}h_{t-1} + b_r) \tag{1.19}$$

3. **Candidate activation**:
$$\tilde{h}_t = \tanh(W_{xh}x_t + r_t \odot (W_{hh}h_{t-1}) + b_h) \tag{1.20}$$

4. **Hidden state update**:
$$h_t = (1 - z_t) \odot h_{t-1} + z_t \odot \tilde{h}_t \tag{1.21}$$

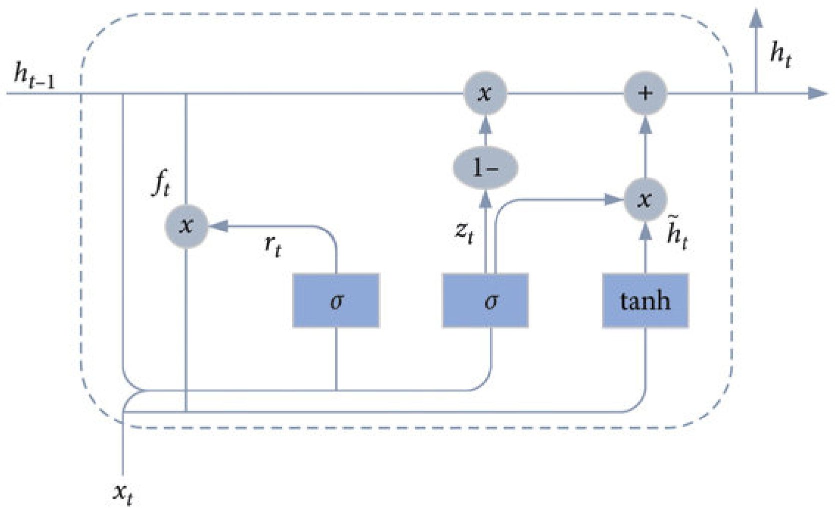


Figure 1.8: GRU block structure [17].

1.3 Reinforcement Learning models

Reinforcement Learning (RL) is a type of machine learning where an agent learns to make decisions by interacting with an environment. By taking actions inside the environment the agent receives feedback in the form of rewards defined by a reward function. This function is used to further improve agent's performance. The goal of the agent is to maximize the cumulative reward over time. A simple RL diagram looks like this:

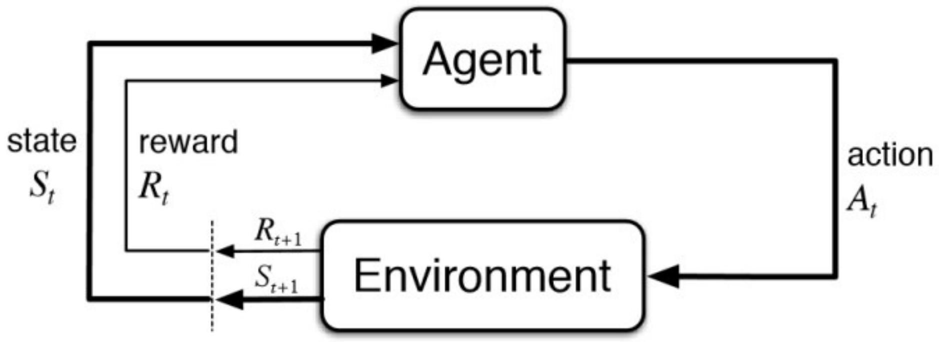


Figure 1.9: Simple RL diagram. Source: analyticsvidhya.com

where:

- State S_t: The current situation or configuration of the environment at time t.
- Action A_t: The decision or move made by the agent at time t.
- Reward R_t: The feedback received from the environment after the agent takes an action at time t.
- Next State S_{t+1}: The state of the environment after the agent takes the action A_t.
- Next Reward R_{t+1}: The feedback received after transitioning to the next state S_{t+1}.

The agent's action selection are modeled as a map called *policy* denoted as π:

$$\pi : \mathcal{S} \times \mathcal{A} \to [0, 1],$$

$$\pi(s, a) = \Pr(A_t = a \mid S_t = s)$$

There are many policies available, each specializing in different types of data and problems. For instance, some policies use Convolutional Neural Networks (CNNs) for spatial data, while others use Multi-Layer Perception (MLP). Due to the nature of the stock data, the 'MlpPolicy' will be used.

1.3.1 PPO

Proximal Policy Optimization (PPO) is an advanced RL algorithm that aims to achieve stable and efficient policy updates. Because it is a policy gradient method, it optimizes policies directly [14]. Here is the general algorithm for PPO:

1. Collect trajectories: The agent interacts with the environment to collect sequences of states, actions, rewards, etc.

2. Compute Advantages Calculate the advantage function for the collected trajectories. The advantage function measures how much better an action a is compared to the average action in state s.

3. Update Policy: Adjust the policy to make actions that had higher advantages more likely in the future, but restrict how much it can change in one update (using clipping).

4. Repeat: Repeat the process of collecting data and updating the policy until the agent performs well.

1.3.2 DQN

Deep Q-Network (DQN) is used to train an agent to make decisions by learning the value of taking certain actions in given states. It combines Q-Learning with deep neural networks to handle environments with large state spaces. The main idea of this model is to estimate the value of taking a specific action in a given state, aiming to maximize future rewards. It also uses a replay buffer to store the agent's experiences in order to train the network, improving learning stability and efficiency [19]. Here is how DQN works step by step:

1. Initialize Networks: Initialize two neural networks, one for estimating Q-values (Q-network) and one for providing target Q-values (Target network) as well as the replay buffer to store experiences.

2. Interaction: The agent interacts with the environment, taking actions based on the current policy.

3. Store Experiences: Store the experiences (state, action, reward, next state) in the replay buffer.

4. Sample from buffer and compute targets: Randomly sample a batch of experiences from the replay buffer and se the target network to compute the target Q-values.

5. Update both Q-network and the taget network: Train the Q-network to minimize the difference between the predicted Q-values and the target Q-values and then periodically update the target network to match the Q-network.

1.3.3 A2C

Advantage Actor-Critic (A2C) is a reinforcement learning algorithm that combines the benefits of both policy-based and value-based methods. It uses two neural networks: one to learn the policy (actor) and another to learn the value function (critic). The actor network outputs a probability distribution over actions given the current state. The critic network estimates the value function, which predicts the expected return from a given state [7]. Simple A2C algorithm would look like this:

1. Initialize Networks: Initialize the actor and critic networks.

2. Collect Data: The agent interacts with the environment to collect data about the states, actions, and rewards.

3. Compute Advantages: Calculate the advantage estimates, which are the difference between the actual rewards and the predicted values from the critic.

4. Update actor and critic: Update the actor network to maximize the advantage-weighted log probability of actions and the critic network to minimize the mean squared error between the predicted value and the actual returns.

5. Repeat: Repeat the process of collecting data and updating the networks until the agent performs well.

1.4 Custom indicators

Because both ML and RL models base their predictions on the historical data it would be a good idea to implement a wide variety of custom indicators to improve the results of simulations by providing additional information:

1. Order book imbalance (OBI) - is a measure that indicates whether there is more buying or selling pressure on the market [11]:

$$\mathrm{OBI} = \frac{\mathrm{V}_{\mathrm{buy}} - \mathrm{V}_{\mathrm{sell}}}{\mathrm{V}_{\mathrm{buy}} + \mathrm{V}_{\mathrm{sell}}} \tag{1.22}$$

   where:

   - $\mathrm{V}_{\mathrm{buy}}$ is the total volume of buy orders.
   - $\mathrm{V}_{\mathrm{sell}}$ is the total volume of sell orders.

2. Simple moving average (SMA) - calculates the average of the closing prices over the last n periods. It is used to smooth out price data in order to identify trends [4]:

$$\text{SMA}_n = \frac{1}{n}\sum_{i=0}^{n-1} P_{t-i} \tag{1.23}$$

where:

- P_{t-1} is the price at time $t-1$.
- n is the number of periods.

3. Volatility - is another measure used commonly in trading. One of the approaches when calculating volatility (σ) is to use standard deviation of the returns [12]:

$$\sigma = \sqrt{\frac{1}{n-1}\sum_{i=1}^{n}(R_i - \bar{R})^2} \tag{1.24}$$

where:

- R_i is the return on day i.
- $\bar{R}$ is the average return over the n days.

Chapter 2

Data preparation

Before any data analysis can be performed, the data used needs to be properly described, understood and most importantly prepared. Some models cannot handle NaN (not a number) values, others are heavily impacted by the outliers. Sometimes it is also a good idea to normalize the data as well.

2.1 Used tools

Data analysis was performed in Python using many libraries. Here are the most important ones:

- numpy: Arrays and mathematical functions,
- pandas: Data manipulation and analysis,
- stablebaselines3: Reinforcement learning algorithms,
- scikit-learn: Machine learning tools,
- matplotlib: Plots and visualizations.
- scipy-stats: Statistical functions and distributions,
- gym: Standardized environments for RL,
- gym_anytrading: Trading environments for RL,
- finta: Technical indicators for financial analysis.

2.2 Data description and visualization

Data that will be used in the analysis comes from the Warsaw Stock Exchange (WSE) and consist of almost 50'000 intraday observations from 2021 to 2023 collected every 5 minutes for almost 8 hours a day. For both Machine and Reinforcement Learning purposes we will be using KGHM as our testing stock.

The easiest way to understand our data is to look at the plots of 'close' and 'volume' values, as well as the logarithmic returns and the summary statistics.

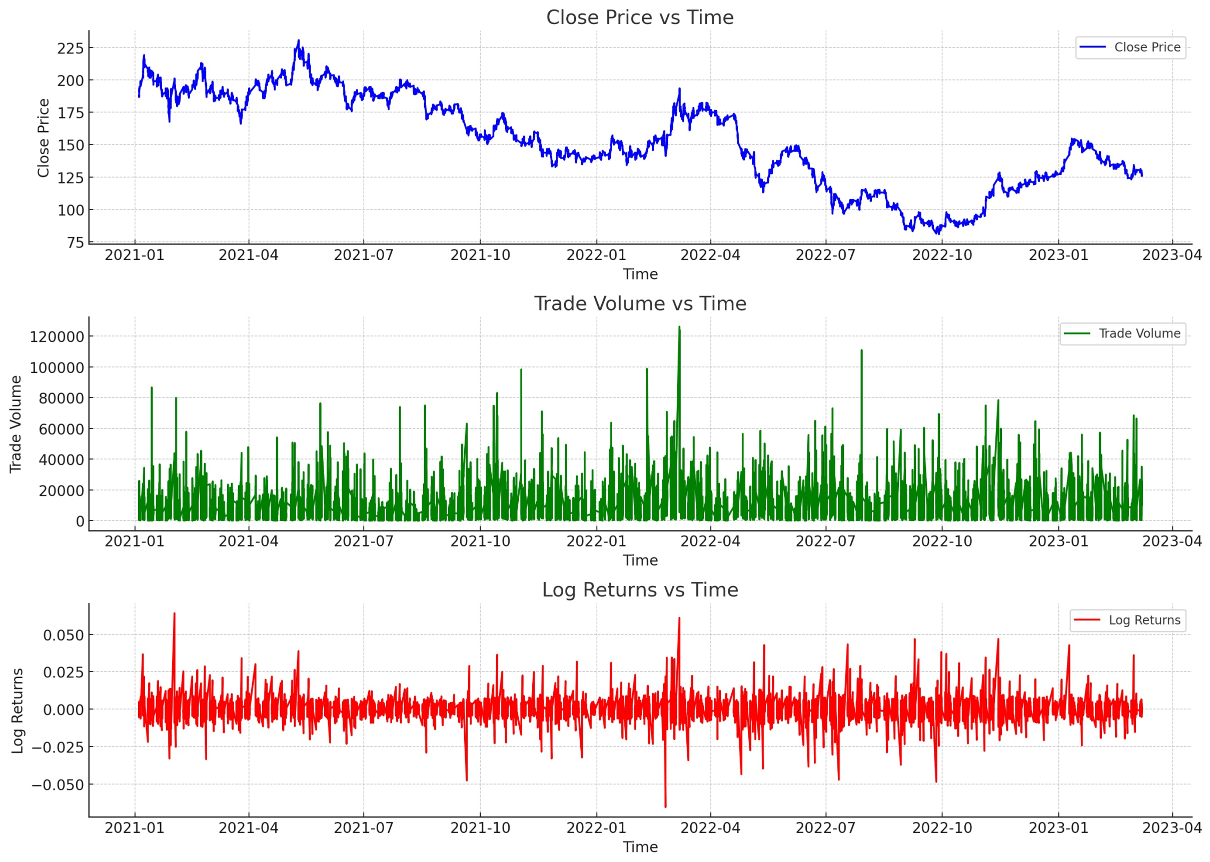


Figure 2.1: Closing price, volume and log returns of KGHM stock.

Table 2.1: Summary Statistics - Part 1.

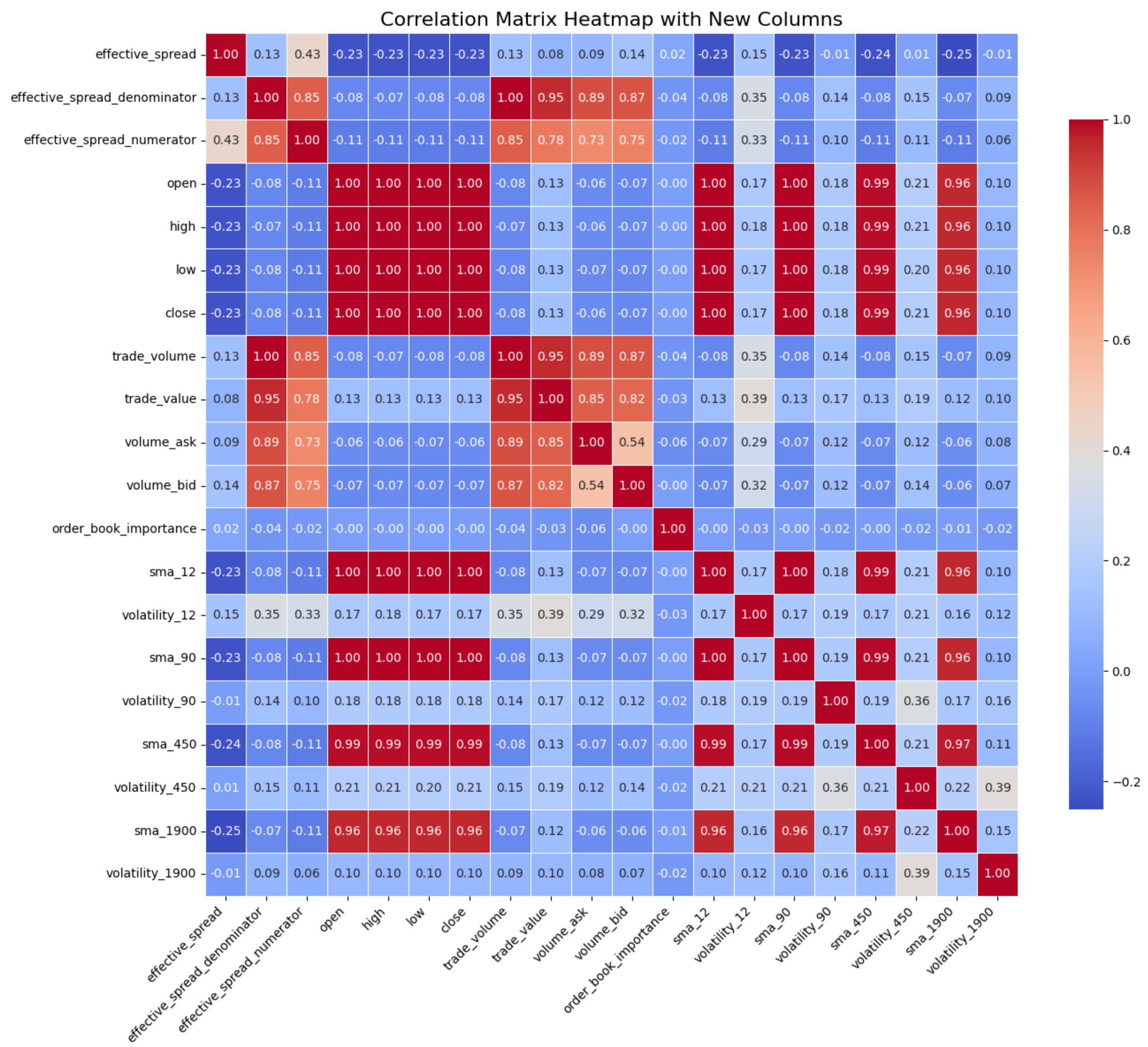
	Eff__spread	**Eff__spread__denom**	**Eff__spread__num**	**Open**	**High**
Count	51467	51467	51467	51467	51467
Mean	8.90	5515.50	52171.70	152.52	152.75
Std. Dev.	3.84	6022.01	84354.40	35.17	35.21
Min	0.00	0.00	0.00	80.90	81.00
25%	6.23	2004.00	15094.71	126.40	126.60
50%	8.10	3761.00	29534.56	150.45	150.65
75%	10.64	6825.00	57645.89	185.05	185.35
Max	50.44	124181.00	2935469.00	230.60	230.80

Table 2.2: Summary Statistics - Part 2.

	Close	**Volume**	**Trade__value**	**Volume__ask**	**Volume__bid**	**Low**
Count	51467	51467	51467	51467	51467	51467
Mean	152.51	5540.64	828610.80	2788.62	2726.88	152.26
Std. Dev.	35.17	6045.02	924445.80	3540.86	3316.97	35.13
Min	80.86	1.00	91.70	0.00	0.00	80.68
25%	126.40	2015.00	294459.80	858.00	860.00	126.20
50%	150.40	3781.00	561974.70	1796.00	1766.00	150.20
75%	185.05	6860.50	1026762.00	3417.00	3349.00	184.75
Max	230.60	126175.00	23577140.00	113267.00	76586.00	230.20

2.3 Feature selection

One of the better ways to find out which features to use in the models is to look at the correlation matrix.



	effective_spread	effective_spread_denominator	effective_spread_numerator	open	high	low	close	trade_volume	trade_value	volume_ask	volume_bid	order_book_importance	sma_12	volatility_12	sma_90	volatility_90	sma_450	volatility_450	sma_1900	volatility_1900
effective_spread	1.00	0.13	0.43	-0.23	-0.23	-0.23	-0.23	0.13	0.08	0.09	0.14	0.02	-0.23	0.15	-0.23	-0.01	-0.24	0.01	-0.25	-0.01
effective_spread_denominator	0.13	1.00	0.85	-0.08	-0.07	-0.08	-0.08	1.00	0.95	0.89	0.87	-0.04	-0.08	0.35	-0.08	0.14	-0.08	0.15	-0.07	0.09
effective_spread_numerator	0.43	0.85	1.00	-0.11	-0.11	-0.11	-0.11	0.85	0.78	0.73	0.75	-0.02	-0.11	0.33	-0.11	0.10	-0.11	0.11	-0.11	0.06
open	-0.23	-0.08	-0.11	1.00	1.00	1.00	1.00	-0.08	0.13	-0.06	-0.07	-0.00	1.00	0.17	1.00	0.18	0.99	0.21	0.96	0.10
high	-0.23	-0.07	-0.11	1.00	1.00	1.00	1.00	-0.07	0.13	-0.06	-0.07	-0.00	1.00	0.18	1.00	0.18	0.99	0.21	0.96	0.10
low	-0.23	-0.08	-0.11	1.00	1.00	1.00	1.00	-0.08	0.13	-0.07	-0.07	-0.00	1.00	0.17	1.00	0.18	0.99	0.20	0.96	0.10
close	-0.23	-0.08	-0.11	1.00	1.00	1.00	1.00	-0.08	0.13	-0.06	-0.07	-0.00	1.00	0.17	1.00	0.18	0.99	0.21	0.96	0.10
trade_volume	0.13	1.00	0.85	-0.08	-0.07	-0.08	-0.08	1.00	0.95	0.89	0.87	-0.04	-0.08	0.35	-0.08	0.14	-0.08	0.15	-0.07	0.09
trade_value	0.08	0.95	0.78	0.13	0.13	0.13	0.13	0.95	1.00	0.85	0.82	-0.03	0.13	0.39	0.13	0.17	0.13	0.19	0.12	0.10
volume_ask	0.09	0.89	0.73	-0.06	-0.06	-0.07	-0.06	0.89	0.85	1.00	0.54	-0.06	-0.07	0.29	-0.07	0.12	-0.07	0.12	-0.06	0.08
volume_bid	0.14	0.87	0.75	-0.07	-0.07	-0.07	-0.07	0.87	0.82	0.54	1.00	-0.00	-0.07	0.32	-0.07	0.12	-0.07	0.14	-0.06	0.07
order_book_importance	0.02	-0.04	-0.02	-0.00	-0.00	-0.00	-0.00	-0.04	-0.03	-0.06	-0.00	1.00	-0.00	-0.03	-0.00	-0.02	-0.00	-0.02	-0.01	-0.02
sma_12	-0.23	-0.08	-0.11	1.00	1.00	1.00	1.00	-0.08	0.13	-0.07	-0.07	-0.00	1.00	0.17	1.00	0.18	0.99	0.21	0.96	0.10
volatility_12	0.15	0.35	0.33	0.17	0.18	0.17	0.17	0.35	0.39	0.29	0.32	-0.03	0.17	1.00	0.17	0.19	0.17	0.21	0.16	0.12
sma_90	-0.23	-0.08	-0.11	1.00	1.00	1.00	1.00	-0.08	0.13	-0.07	-0.07	-0.00	1.00	0.17	1.00	0.19	0.99	0.21	0.96	0.10
volatility_90	-0.01	0.14	0.10	0.18	0.18	0.18	0.18	0.14	0.17	0.12	0.12	-0.02	0.18	0.19	0.19	1.00	0.19	0.36	0.17	0.16
sma_450	-0.24	-0.08	-0.11	0.99	0.99	0.99	0.99	-0.08	0.13	-0.07	-0.07	-0.00	0.99	0.17	0.99	0.19	1.00	0.21	0.97	0.11
volatility_450	0.01	0.15	0.11	0.21	0.21	0.20	0.21	0.15	0.19	0.12	0.14	-0.02	0.21	0.21	0.21	0.36	0.21	1.00	0.22	0.39
sma_1900	-0.25	-0.07	-0.11	0.96	0.96	0.96	0.96	-0.07	0.12	-0.06	-0.06	-0.01	0.96	0.16	0.96	0.17	0.97	0.22	1.00	0.15
volatility_1900	-0.01	0.09	0.06	0.10	0.10	0.10	0.10	0.09	0.10	0.08	0.07	-0.02	0.10	0.12	0.10	0.16	0.11	0.39	0.15	1.00

Figure 2.2: Correlation matrix of all variables in the dataset.

From the correlation matrix alone we can clearly see that we can replace some of the variables with our custom indicators because of the really high correlation between them. For example we can substitute 'open', 'high' or 'low' with simple moving averages. We can of course use both options and see how that impacts the performance of each model.

Additionally for each model we can perform something called **feature selection**. As an example we can use it in a Random Forest model for determining which variables have the most impact.

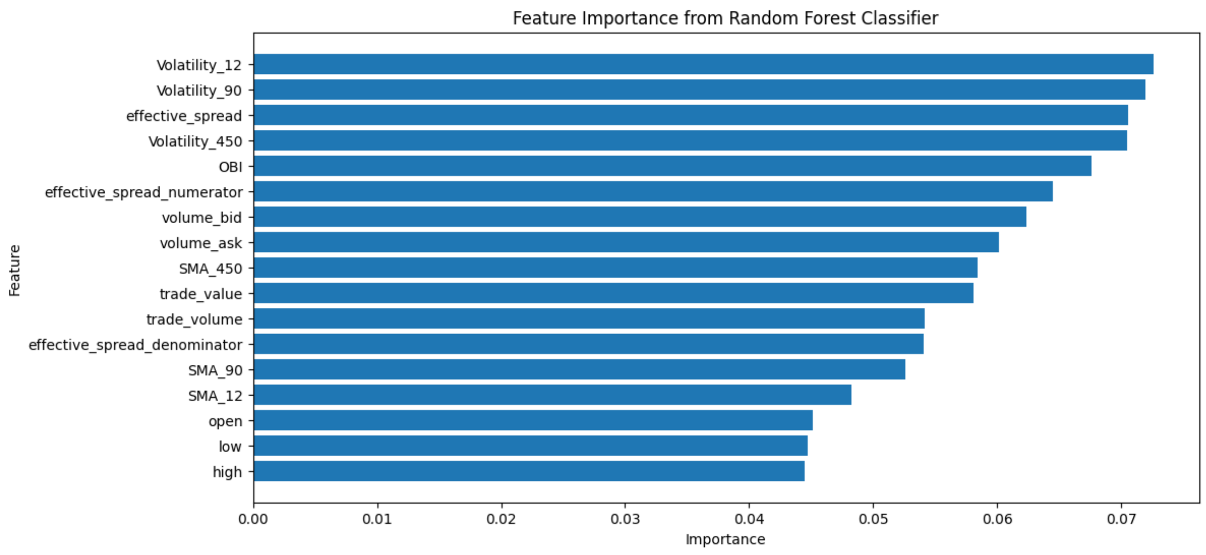


Figure 2.3: Feature selection in Random Forest.

Features such as 'Volatility' and 'effective_spread' have the most impact on the model so they should definitely be included in the analysis. We should also experiment with different combinations. We can disregard features such as 'volume_ask' and 'volume_bid' because they are used to create 'OBI' feature.

Chapter 3

Machine Learning

In this chapter we will use the models introduced in Chapter 1 using the data prepared in Chapter 2 and see how these models perform when predicting stock values or trends. We will talk about their strengths, weaknesses and compare their results. The models that will used to predict the closing price are Linear Regression, Decision Tree Regressor and Recurrent Neural Network models - SimpleRNN, LSTM and GRU.

3.1 Linear Regression and Decision Tree Regressor in price prediction

Using the feature selection we talked about in 2.3 the following attributes were chosen

- Open price.
- High price.
- Low price.

As well as the custom indicators introduced in 1.4:

- OBI.
- SMA.
- Volatility.

When it comes to predicting the closing prices of a stock using Linear Regression or Decision Tree Regressor, one thing immediately comes to mind. In order to get any satisfying results (low MSE or MAE) we need to include some highly correlated feature like ’open’, ’high’, ’low’, or some kind of SMA. Tables below show the results for features: ’order_book_importance’, ’sma_90’, ’volatility_90’, ’sma_450’, ’volatility_450’.

Table 3.1: Model Evaluation Metrics

Model	**MSE**	**MAE**	**R-squared**
Decision Tree	0.443	0.339	0.998
Linear Regression	2.725	1.286	0.997

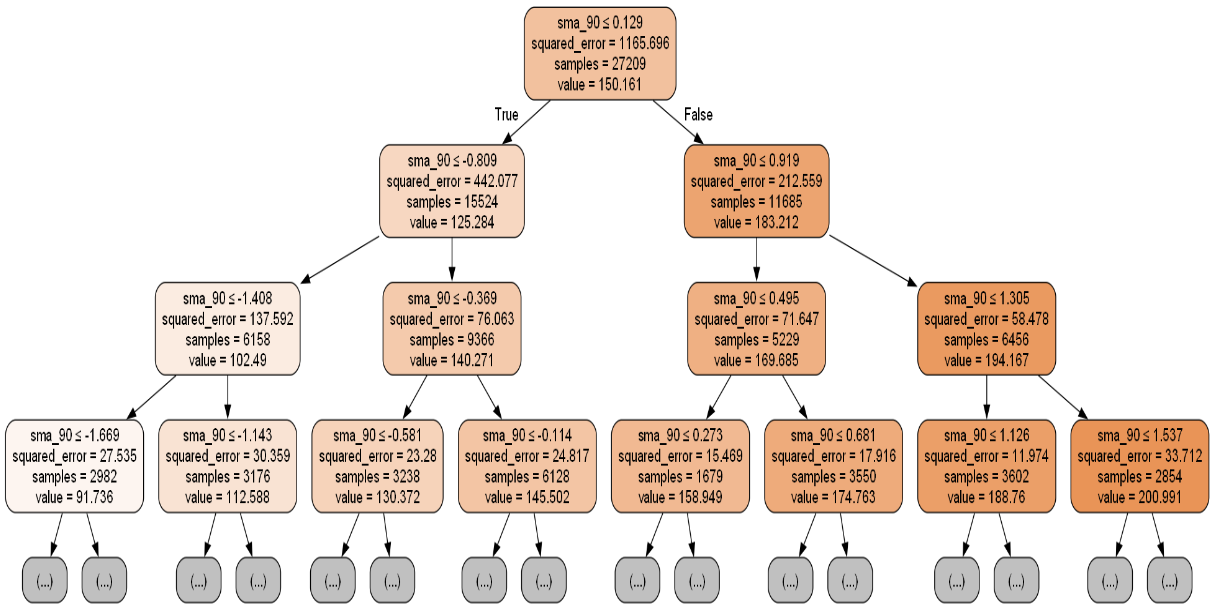


Figure 3.1: Feature selection in Random Forest.

Decision Tree Regressor performed better than linear regression. It can be further tuned by optimizing the choice of hyperparameters [18] but in this case this is not necessary as the results are already great.

3.2 Recurrent Neural Networks in price prediction

The same set of features was used to predict closing price using RNN's. Each model had an early stopping implemented to avoid overfitting meaning that if the model has not improved for the next ten epochs, the simulation stops. Below is the scheme for all the models as well as the plots for each RNN with a table comparing different metrics for each model. For each model the following attributes were used for prediction: 'open', 'trade_volume', 'OBI', 'SMA_90', 'volatility_90'.

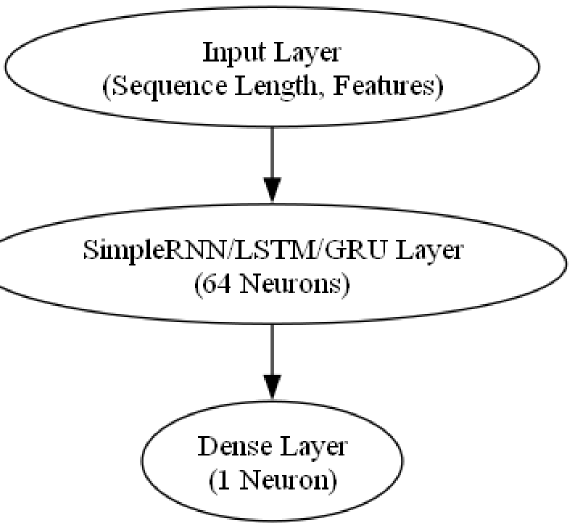


Figure 3.2: RNN scheme.

- SimpleRNN: the basic form of RNN, which can suffer from vanishing gradient problems, making it less effective for long sequences.

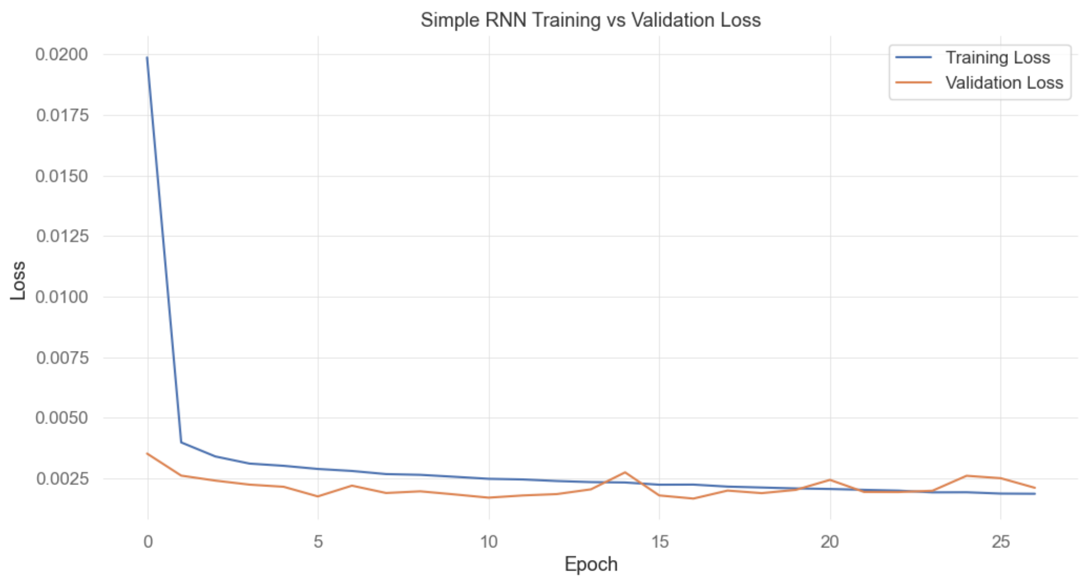


Figure 3.3: Traning versus validation in the SimpleRNN model.

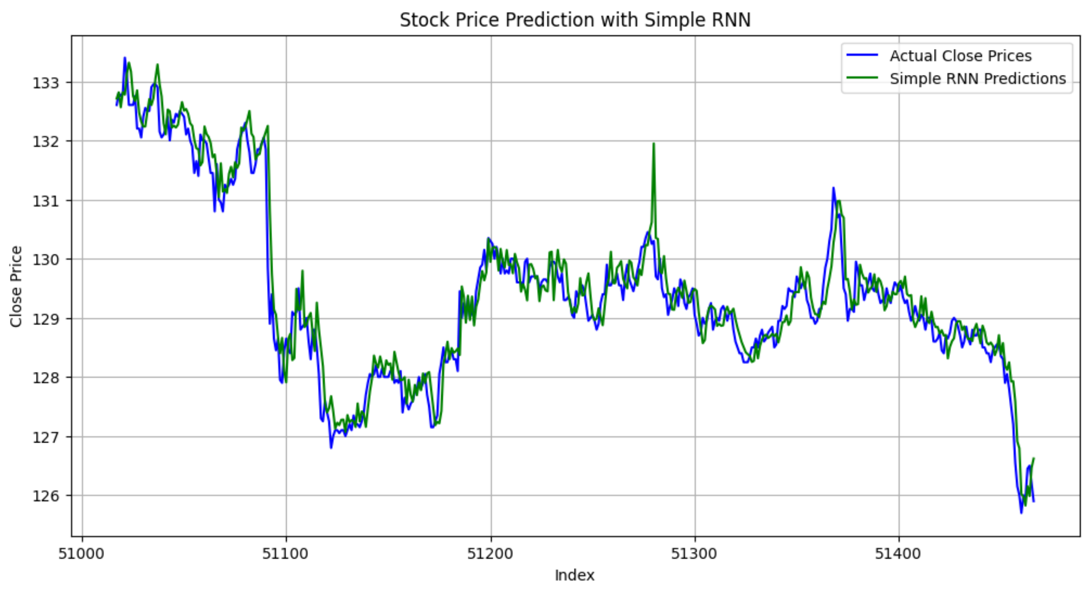


Figure 3.4: Comparison between actual prices and the ones predicted by the SimpleRNN.

- LSTM: these models use memory cells and gates to maintain long-term dependencies, making them highly effective for time-series prediction.

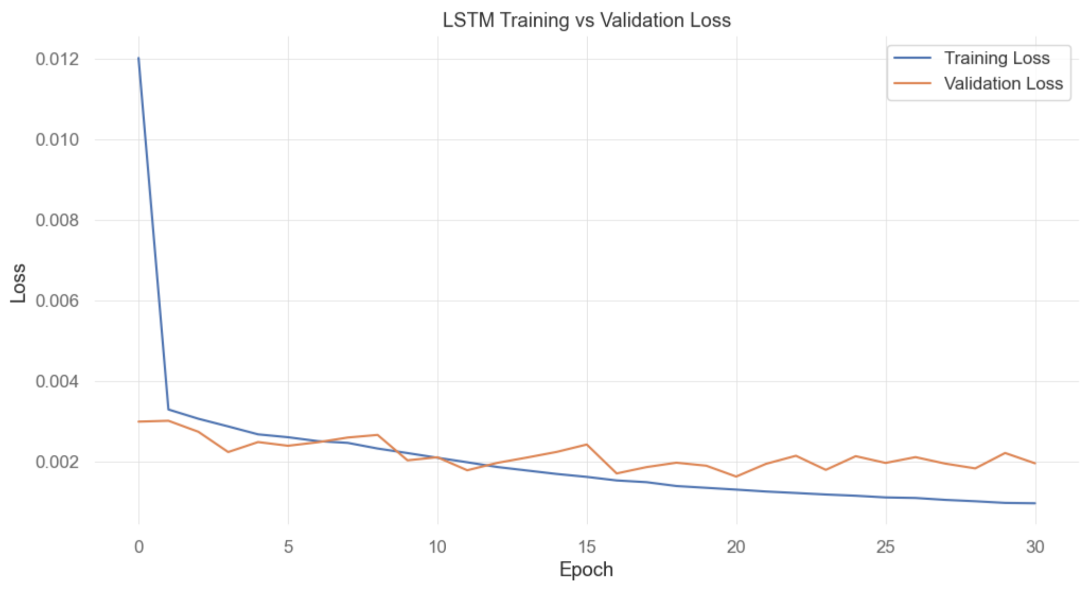


Figure 3.5: Traning versus validation in the LSTM model.

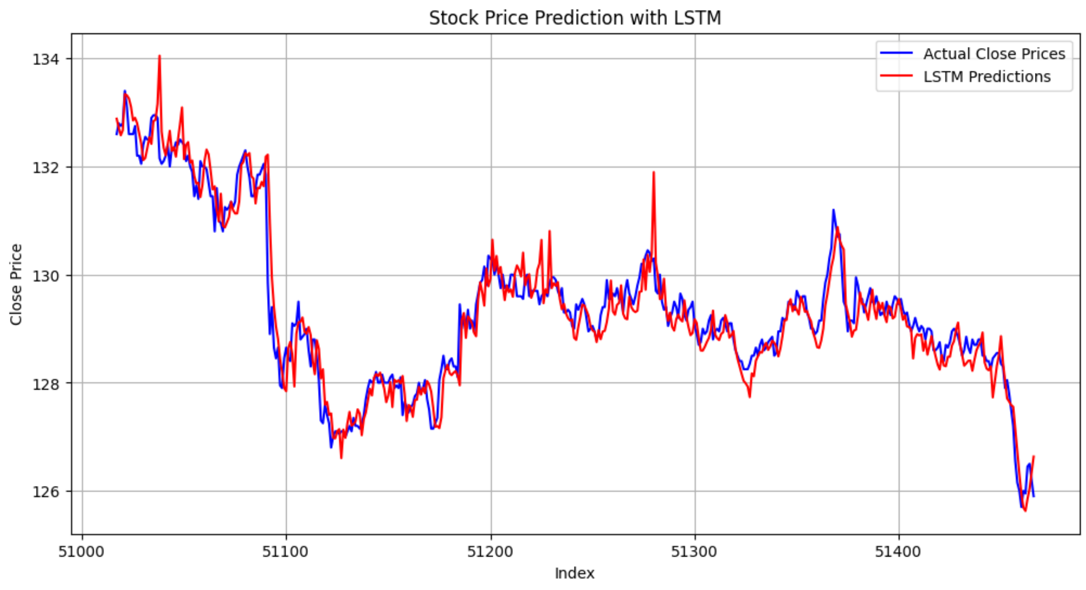


Figure 3.6: Comparison between actual prices and the ones predicted by the LSTM model.

- GRU: a simpler alternative to LSTM that is computationally more efficient.

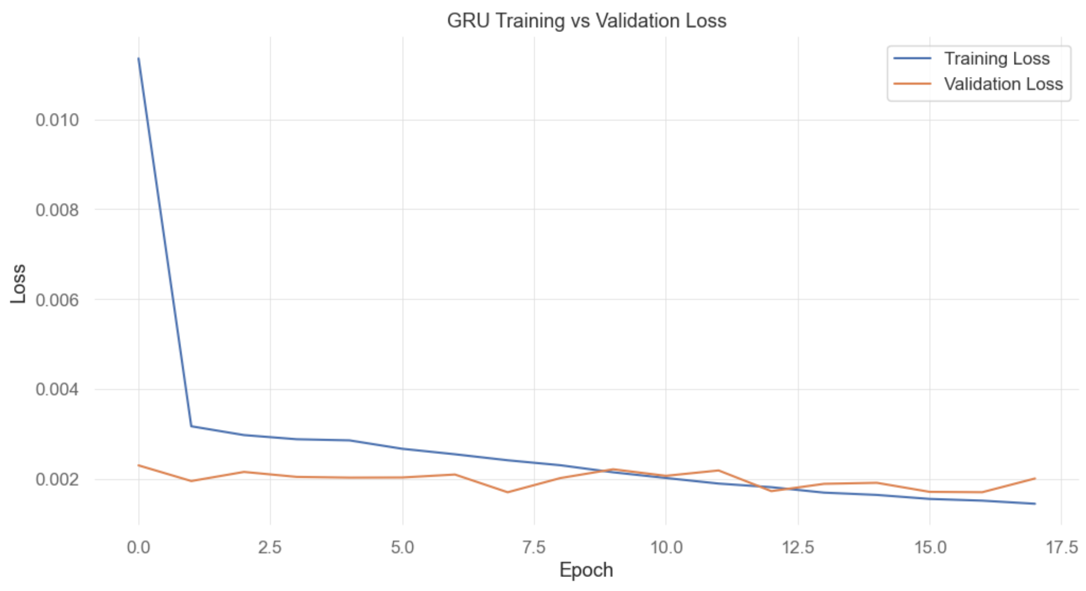


Figure 3.7: Traning versus validation in for the GRU model.

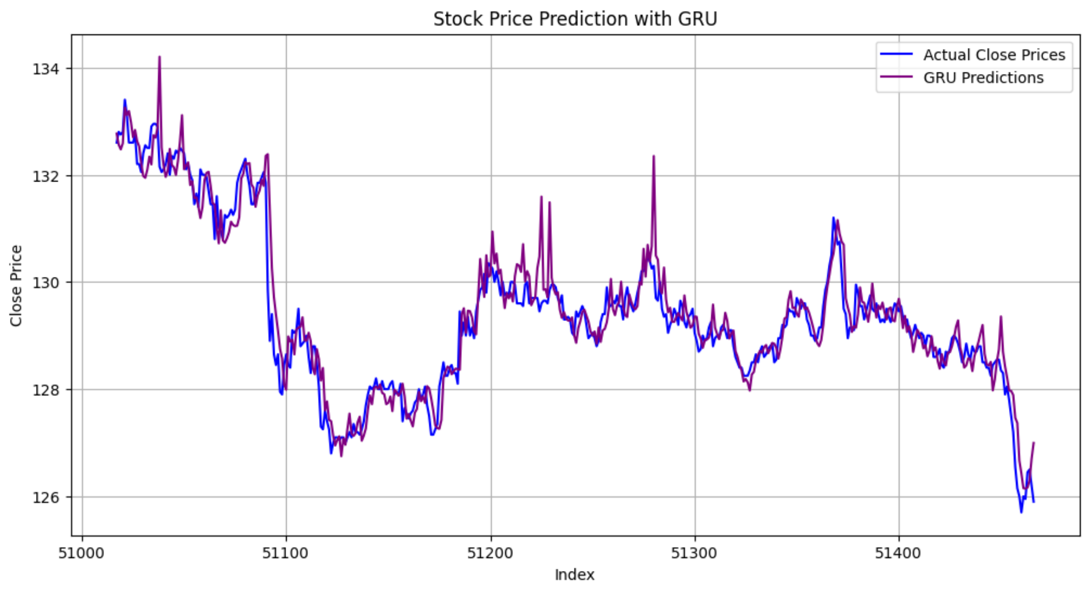


Figure 3.8: Comparison between actual prices and the ones predicted by the GRU model.

Table 3.2: Model Performance Metrics.

Model	**MSE**	**MAE**	**R²**
Simple RNN	0.1483	0.2810	0.9314
LSTM	0.1563	0.2836	0.9281
GRU	0.1882	0.2979	0.9134

All the plots with training and validation losses look correct - training losses keep decreasing, while the validation losses are stable. The SimpleRNN model turned out to be the best from the three with the lowest errors and the highest R^2 metric.

3.3 Trend prediction using Random Forest

Using models like LSTM or linear regression, while good in theory, might not be the best idea in the real world. Even though their predictions are very close to the actual prices, it is not a realistic approach to use these models to make money on trading stocks. Much more important thing than predicting the closing price of a stock is to predict the trend. Will the price go up or down? For this approach, Random Forest will be used.

We will check whether we can accurately tell what will happen with the price by introducing some new variables into our data frame.

- ’Tomorrow’ - we can shift the ’close’ column by a day in order to create a new column that will show us the next days’ price.
- ’Target’ - this one is boolean function which checks whether the tomorrow’s price went up (returning ’1’) or down (returning ’0’).

We will also measure the model’s performance using the accuracy metric. It will tell us how many of our predictions were correct (percentage-wise). Models were backtested using 10’000 observations to learn from and than after the initial training, we introduce a sliding window of 1000 observations, meaning each iteration of the backtest, the model had 1000 new observations in the training pool.

First we will look at the model with just ’Open’, ’High’, ’Low’ and ’Trade_volume’ as features:

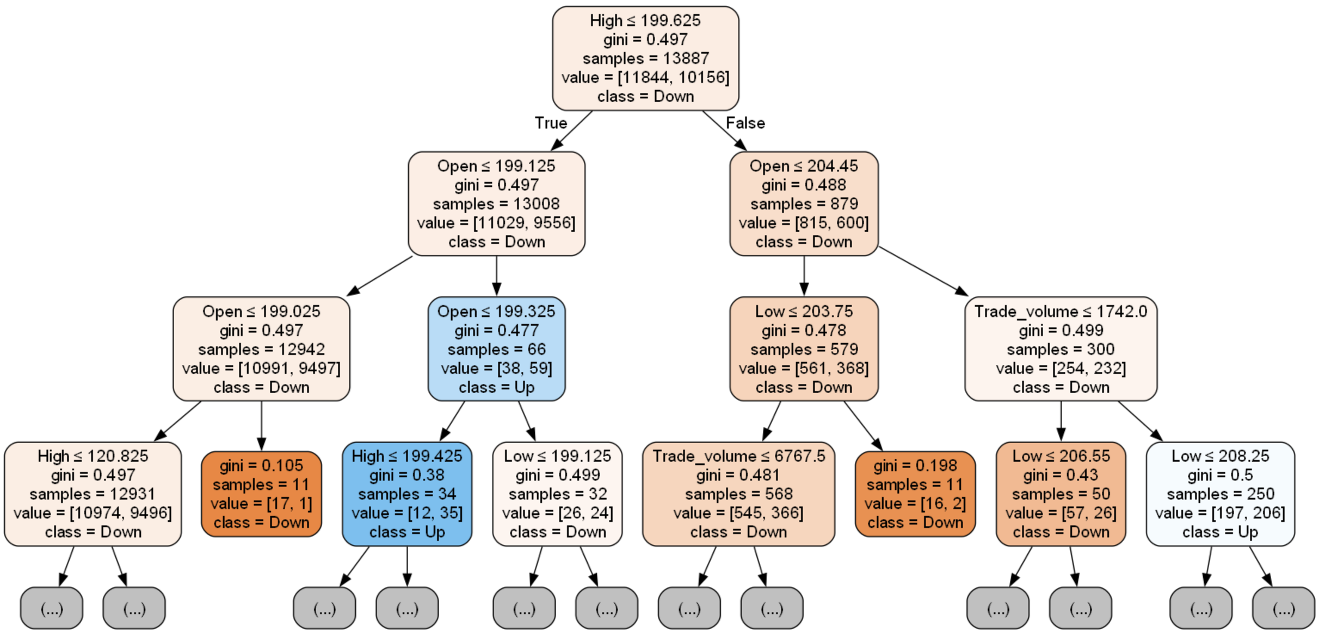


Figure 3.9: Fragment of a decision tree from Random Forest model with base parameters.

Despite previous models’ results that estimated the closing price so perfectly (in theory), this time when predicting the trend, Random Forest with base features could not reach above 46% accuracy.

Table 3.3: Precision Scores using base indicators.

Max Depth	**Precision Score**
2	0.4499
3, 4, 5	0.4535
10	0.4559
20	0.4512

Even with tuning the parameters of the model like experimenting with max_depth, the results have not improved much as shown in the table 2.4.

In the second version of the model custom indicators like SMA, volatility, OBI and rolling ratios were used instead. This time the results are a lot more promising.

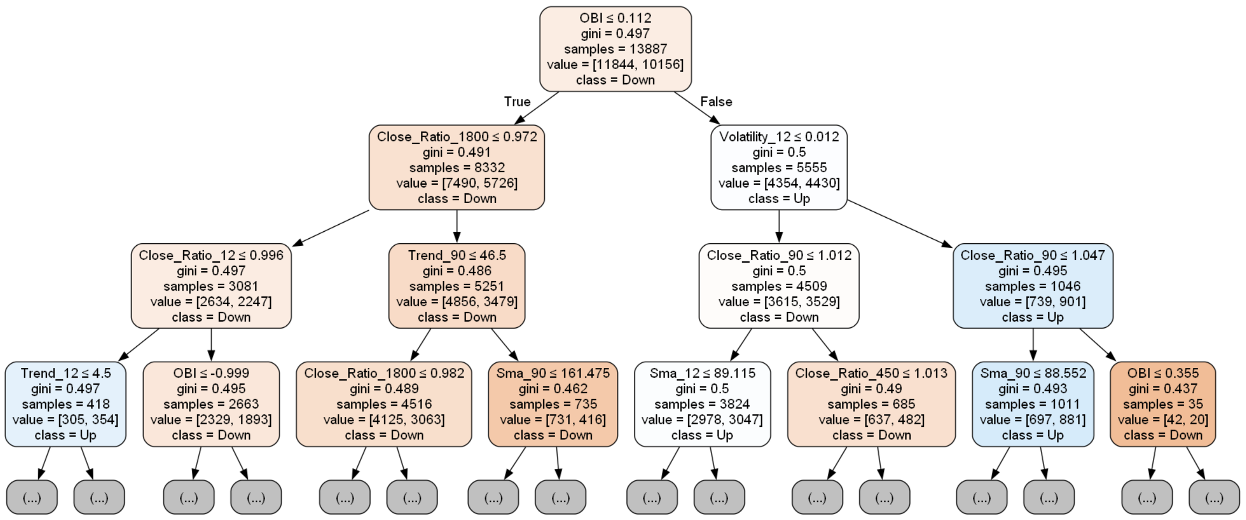


Figure 3.10: Fragment of a decision tree from Random Forest model with custom indicators.

With the improved model the accuracy went up to almost 49%. After tuning the parameters with GridSearch function [8], exploring different combinations of parameter values, the tuned model reached 52.7% accuracy. This means if we bought/sold our stocks base on the predictions, we would have actually made money. To go a step further we need to explore reinforcement learning territory and see, if we can train the agent to perform similar analysis in order to buy/sell stocks.

Chapter 4

Trading with RL using price prediction

This chapter will tackle the problem of creating an agent that learns by interacting with the trading environment, as well as using various RL models to optimize its trading performance. For this section we will use 'StocksEnv' environment from gym_anytrading. The agent will explore and learn during the first 50000 observations and than will be tested on the next 450 (which is about one trading week).

4.1 Base environment

First it will be necessary to introduce the base version of the environment. Here, the agent will only use the 'Close' column from the input data, as well as the differences in prices, which could help him better understand trends and volatility. Here is how the agent's learning process inside the 'StocksEnv' environment looks like:

1. State Representation: The state is represented by 'signal_features', which includes the close price and the difference in those prices within a specified window. The window size used in the simulations is 90 which is almost exactly one trading day as we use data with 5 minute time intervals.

2. Actions: The agent can take one of the following actions:

   - Buy: Enter or maintain a long position.
   - Sell: Enter or maintain a short position.
   - Hold: Do nothing and maintain the current position.

3. Rewards: Rewards are calculated based on the change in prices and the actions taken. For instance:

   - If the agent buys and the price goes up, it receives a positive reward.
   - If the agent sells and the price goes down, it receives a positive reward.
   - The agent also takes into account the trading fees, which are factored into the reward calculation.

4. Trading fees: For both selling and buying the stock the are trading fees (1%) in order to simulate real life stock market.

After initializing the environment we have the access to information about the shape of the input data or the size of the environment. The most important thing available from this data is the 'max_possible_profit' metric. It signifies that if the market didn't have trade fees, one could have earned 1.4341863669137915 units of currency by starting with 1.0. **Long** and **Short** positions are shown in green and red colors. Note that the agent buys and sells the maximum possible amount of stock every time. The starting position of the environment is always **Short**. Even before training the agent using any of the models mentioned in 1.3 we can take a look on what will happen if we tell the agent to perform random actions.

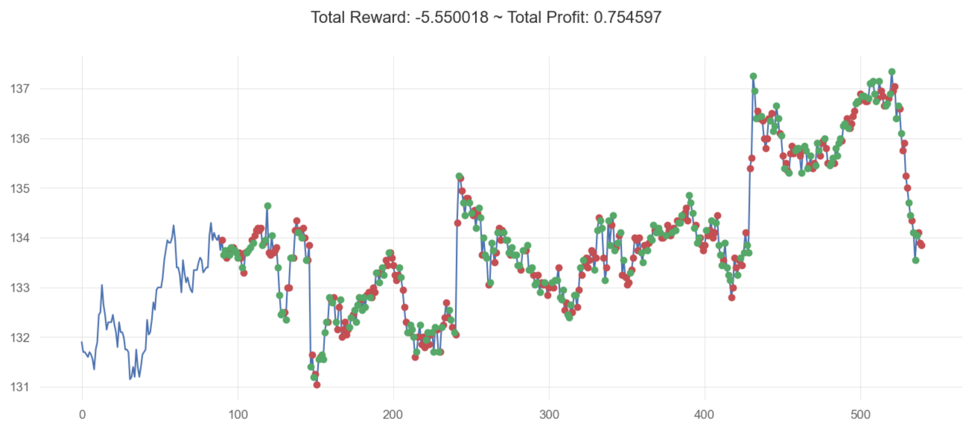


Figure 4.1: Random actions taken by the agent in the base environment. The x-axis represents indexes with the y-axis being the price of the stock.

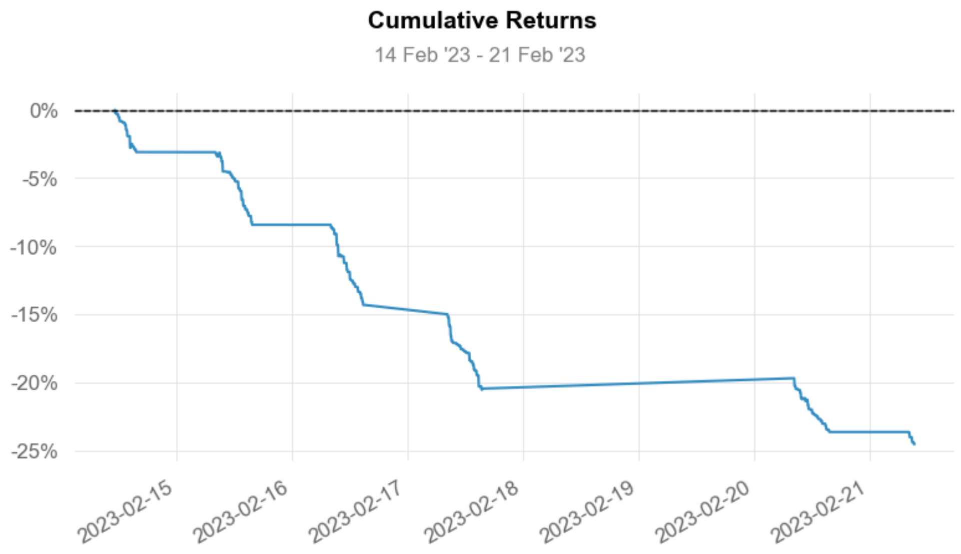


Figure 4.2: Cumulative returns of the strategy taken by the agent using random actions.

During the specified window of time the agent performing random action lost about 25% of the initial capital. Let us compare that to each of the results from training the agent on each of the previously mentioned models.

- **PPO:** The PPO model achieved much better result than random actions but ended up losing about 7% of the money. Generally without much tuning these models do not perform very well. On its best day the agent made 0.51% profit.

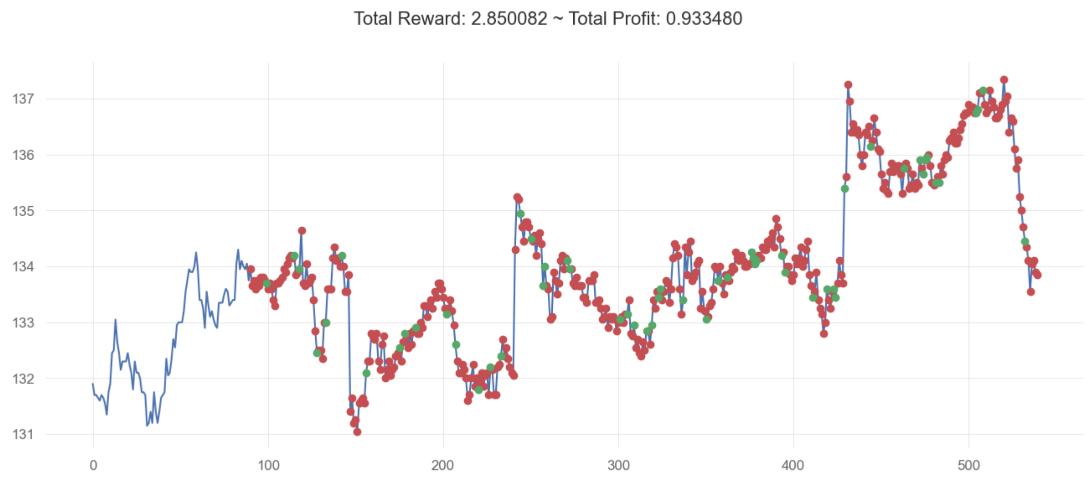


Figure 4.3: Actions taken by an agent that learned using PPO. The x-axis represents indexes with the y-axis being the price of the stock.

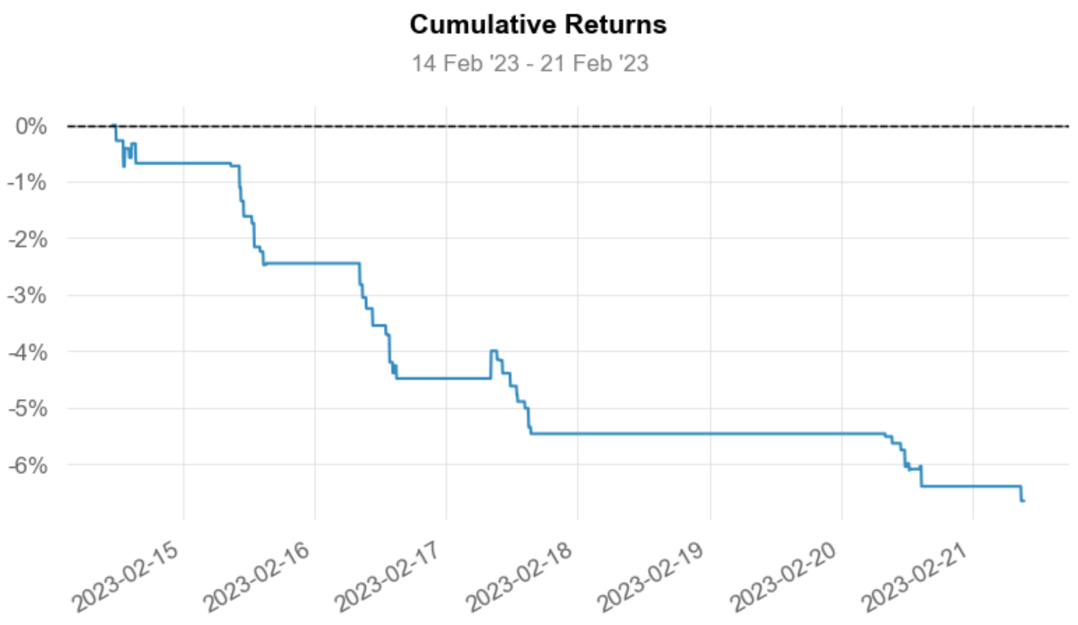


Figure 4.4: Cumulative returns of the strategy taken by the agent that learned using PPO.

As seen from the Actions plot, the agent chose to take the **Short** position much more often than **Long**. Looking at the cumulative returns plot we can that the agent slowly lost more and more of its starting capital.

- **A2C:** Actor-Critic model performed better than taking random actions but worse than the PPO. On average this model was gaining and losing more money than the more stable PPO.

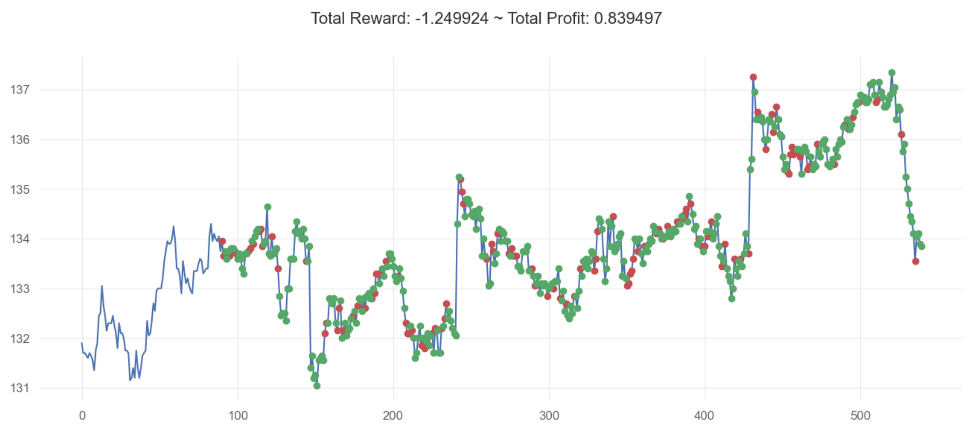


Figure 4.5: Actions taken by an agent that learned using A2C. The x-axis represents indexes with the y-axis being the price of the stock.

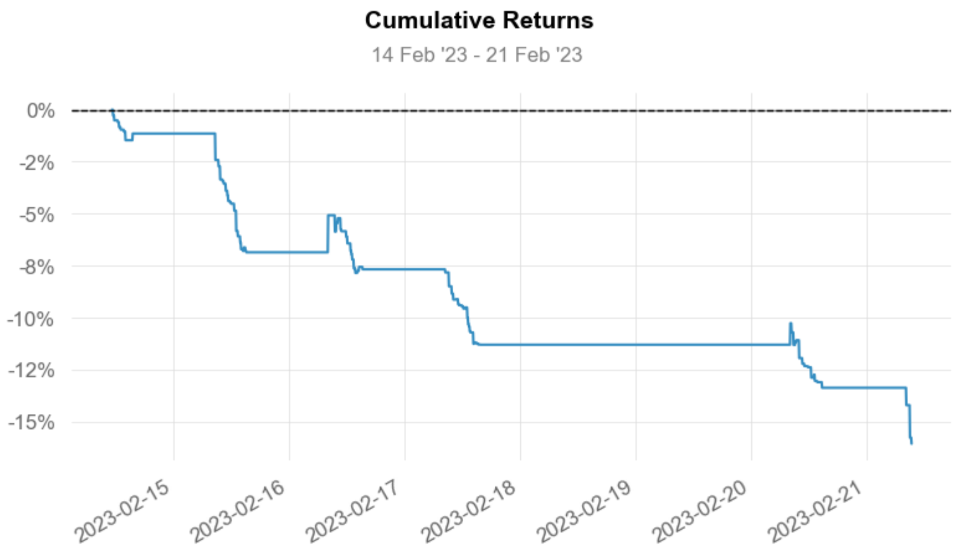


Figure 4.6: Cumulative returns of the strategy taken by the agent that learned using A2C.

When learning using A2C model, the agent much more likely chose to take **Long** positions. There were two moments were he was making back the lost capital but in the end he still ended up losing almost 16%. The return plot looks more volatile when compared to the one from PPO.

- **DQN:** Deep Q-Network behaved similarly to A2C in terms of total profit as well as stability. On the best day it gained more than 2% profit.

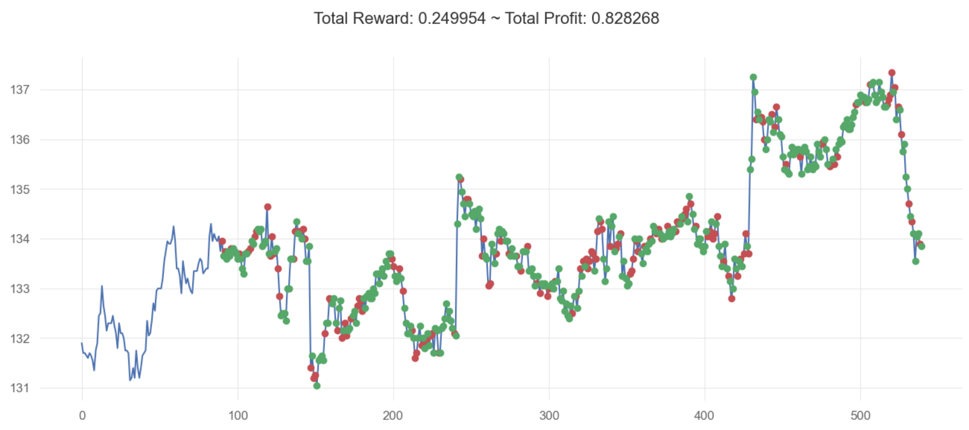


Figure 4.7: Actions taken by an agent that learned using DQN. The x-axis represents indexes with the y-axis being the price of the stock.

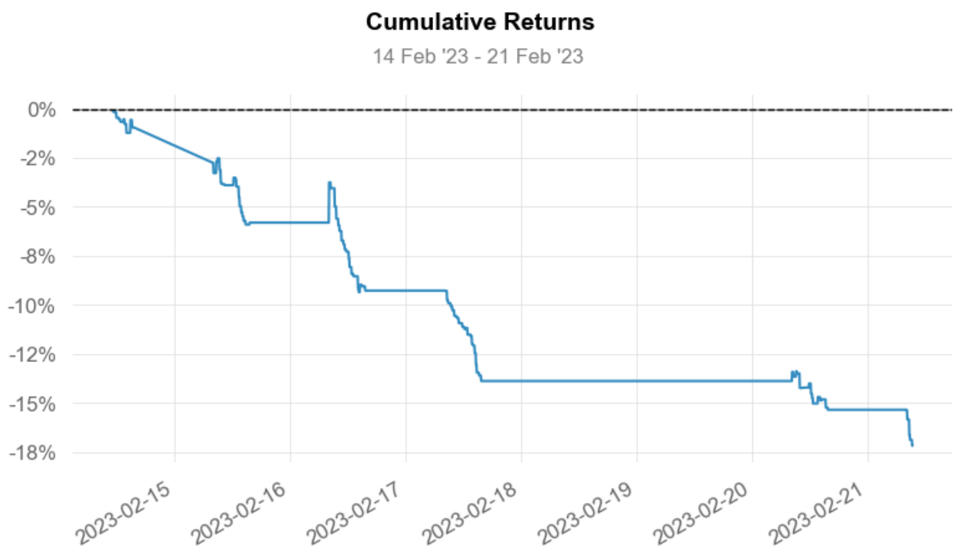


Figure 4.8: Cumulative returns of the strategy taken by the agent that learned using DQN.

Both actions and cumulative returns plots closely resemble their A2C versions. The gent was more likely to take **Long** positions and lost about 18% of its initial capital making it the worst performing model of the three.

4.2 Extended environment

The results of the base models were mostly as expected. All of them outperformed random action selection with the PPO demonstrating greater stability compared to A2C and DQN. Given that these models were only provided with the previous day's closing price as an input, their performance can still be improved by introducing more features as inputs. Notably the custom indicators like SMA or OBI. This in theory should improve the results as the models will have more information and data to work with.

- **Extended PPO:** The agent learning with the extended version of PPO improved the least from the three while still maintaining its characteristic features including stability and choosing to **Short** more often than **Long**.

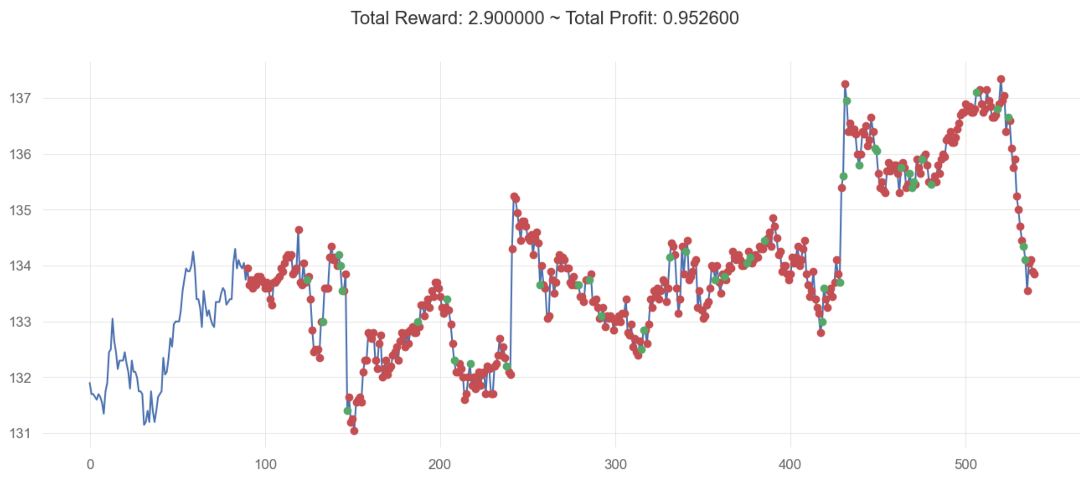


Figure 4.9: Actions taken by an agent that learned using extended PPO. The x-axis represents indexes with the y-axis being the price of the stock.

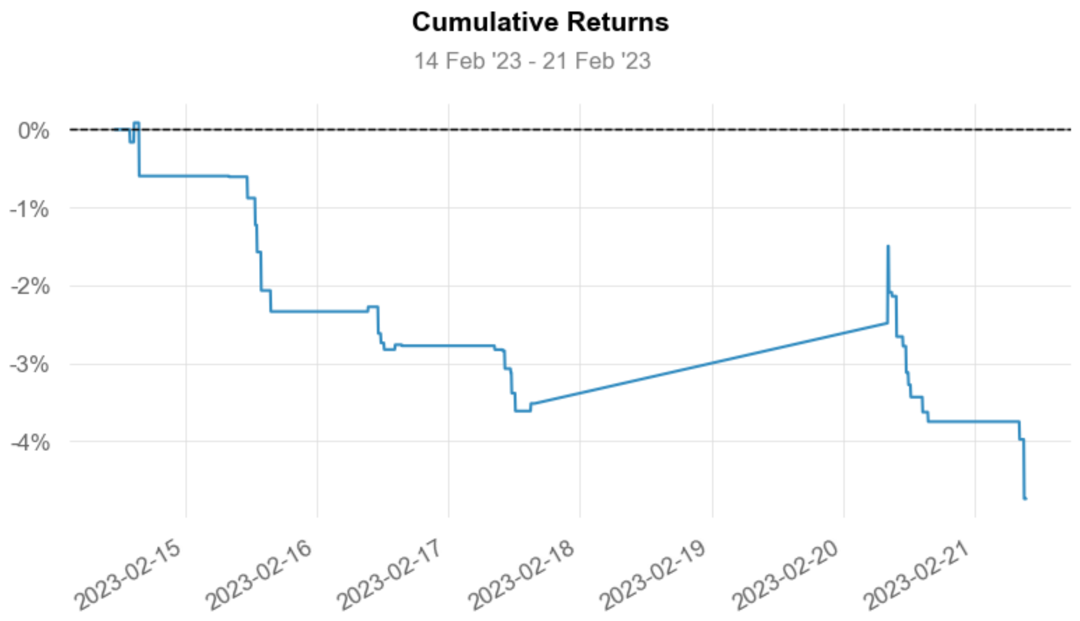


Figure 4.10: Cumulative returns of the strategy taken by the agent that learned using extended PPO.

- **A2C:** The agent's performance using the extended A2C became a lot better. Cumulative returns plot shows that there are times the agent had more money then it started with. With the improvements to the model, the agent lost only $\tilde{4}$% of its initial capital making it the best performing model so far.

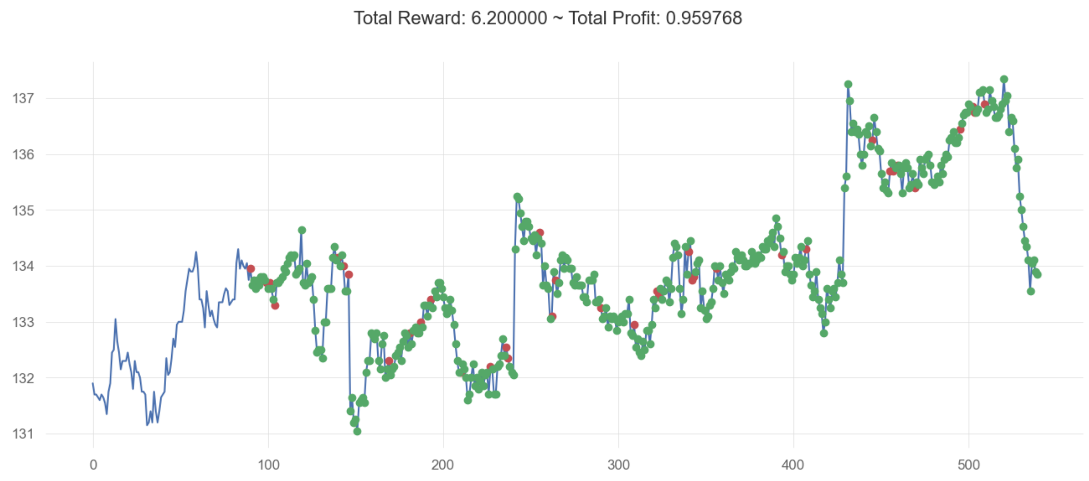


Figure 4.11: Actions taken by an agent that learned using A2C. The x-axis represents indexes with the y-axis being the price of the stock.

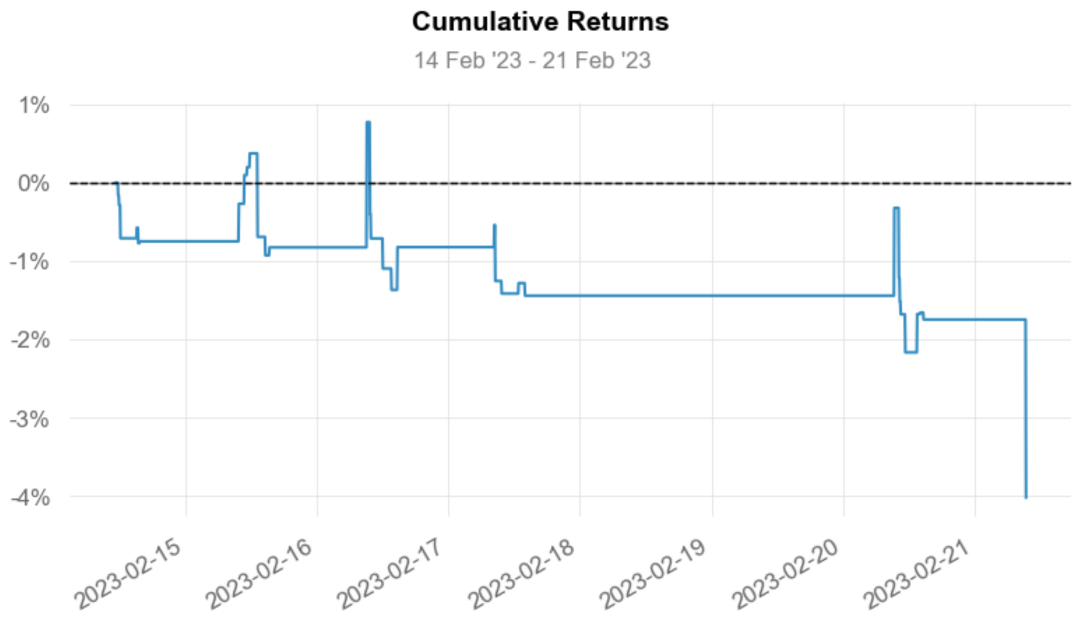


Figure 4.12: Cumulative returns of the strategy taken by the agent that learned using A2C.

- **DQN:** In case of DQN the agent also improved compared to the base version of the model. Similarly to A2C there was a time when the agent was actually making money.

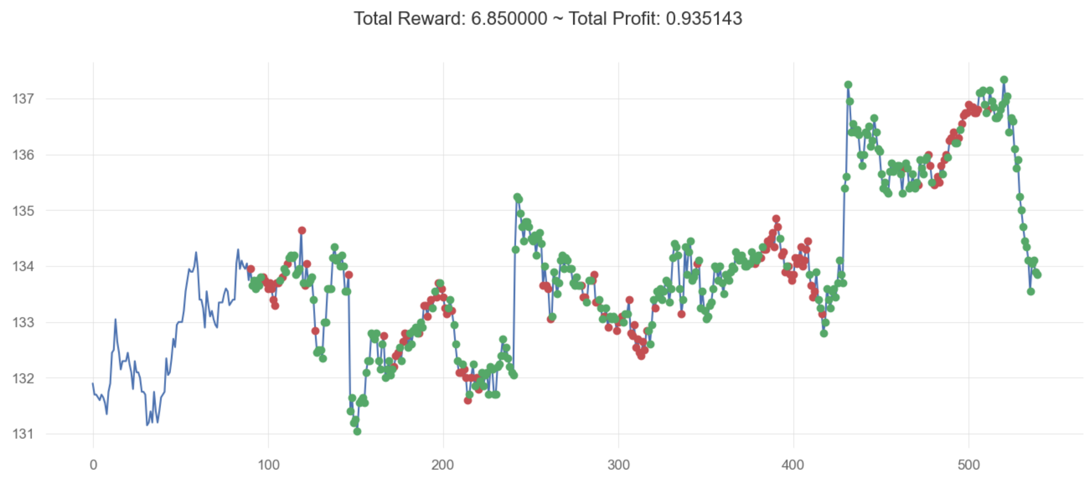


Figure 4.13: Actions taken by an agent that learned using DQN. The x-axis represents indexes with the y-axis being the price of the stock.

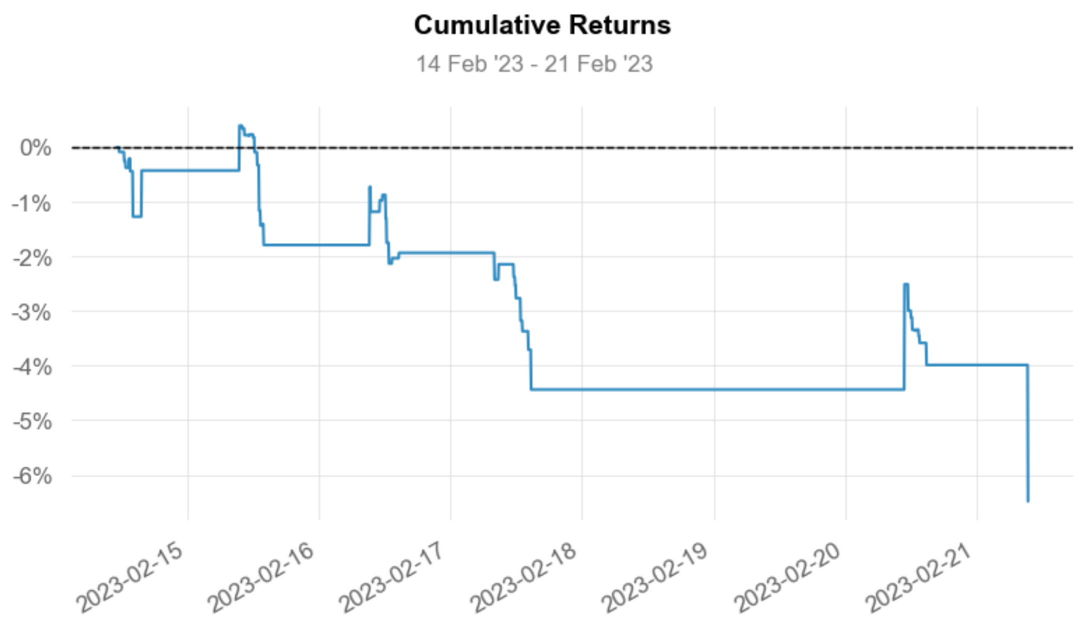


Figure 4.14: Cumulative returns of the strategy taken by the agent that learned using DQN.

4.3 No trading fees extended environment

Trading fees had a large impact on previous RL models' results. Despite the agent not making a profit, this outcome is not entirely negative because it was a simulation of a real-life market, where trading fees and market conditions can significantly influence returns. The best model (PPO), even in the presence of trading fees, was only losing 4%, indicating that it performed relatively well under realistic conditions. Therefore, it would be beneficial to analyze the models' performance without trading fees to understand their true potential and decision-making effectiveness in an ideal scenario.

- **Ideal PPO:** The best performing models of the three. Here, the agent was able to make 3% profit during the specified trading week. Without trading fees was able to perform a lot more trades then before.

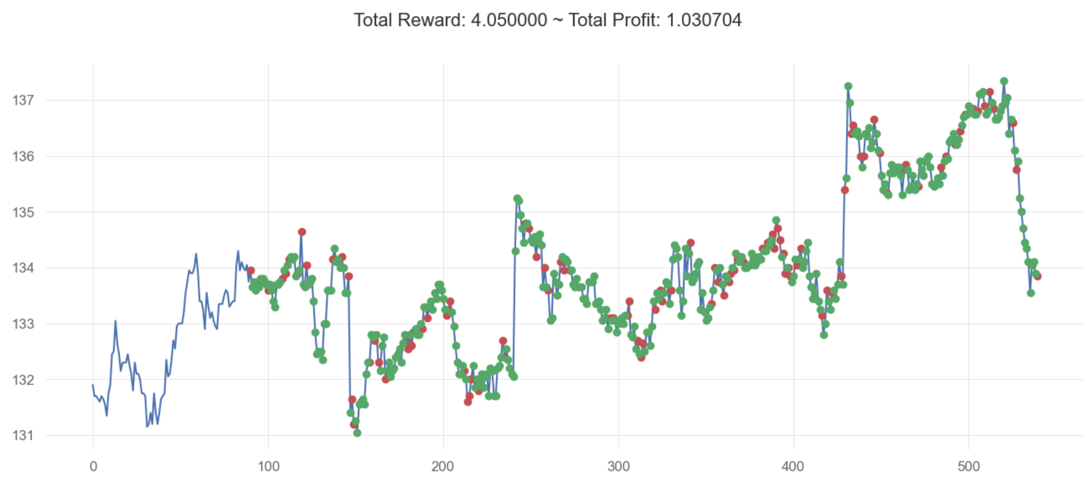


Figure 4.15: Actions taken by an agent that learned using PPO without trading fees. The x-axis represents indexes with the y-axis being the price of the stock.

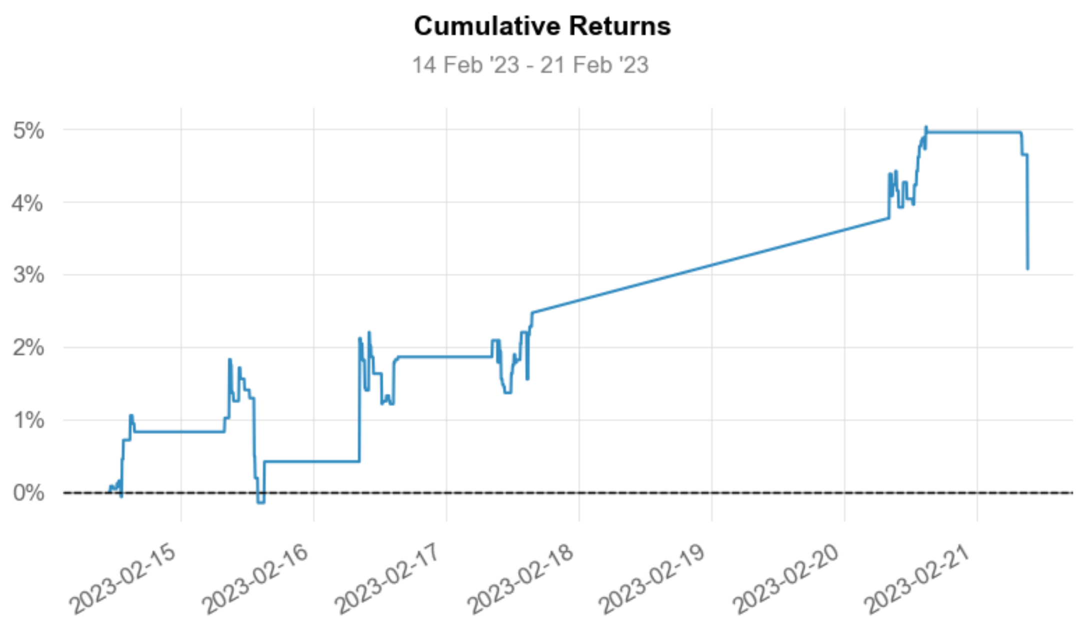


Figure 4.16: Cumulative returns of the strategy taken by the agent that learned using PPO without trading fees.

- **Ideal A2C:** The agent's performance using the ideal version of extended A2C also improved. When we removed the trading fees, the agent was able to make over 2% in profit. The number of trades has also improved.

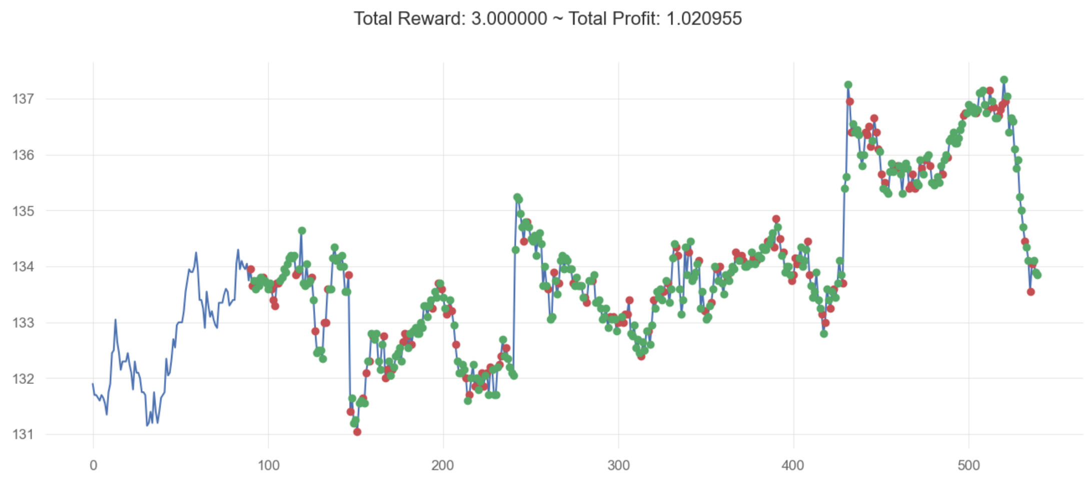


Figure 4.17: Actions taken by an agent that learned using A2C without trading fees. The x-axis represents indexes with the y-axis being the price of the stock.

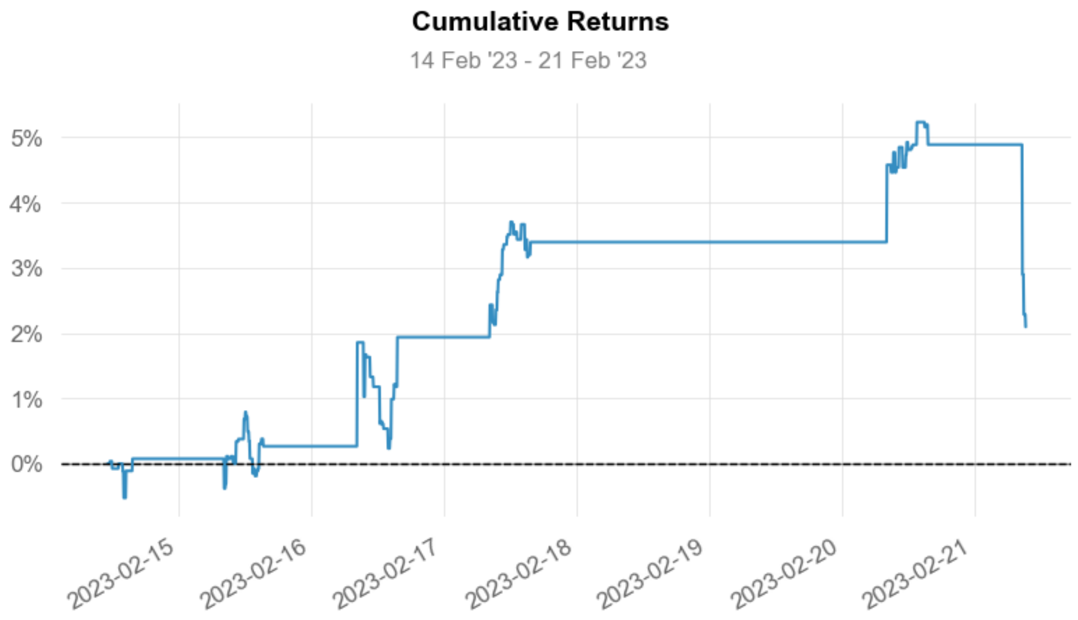


Figure 4.18: Cumulative returns of the strategy taken by the agent that learned using A2C without trading fees.

- **Ideal DQN:** While also bringing profit, the agent that learned using Deep Q-network performed the worst of the three models (1.7% profit).

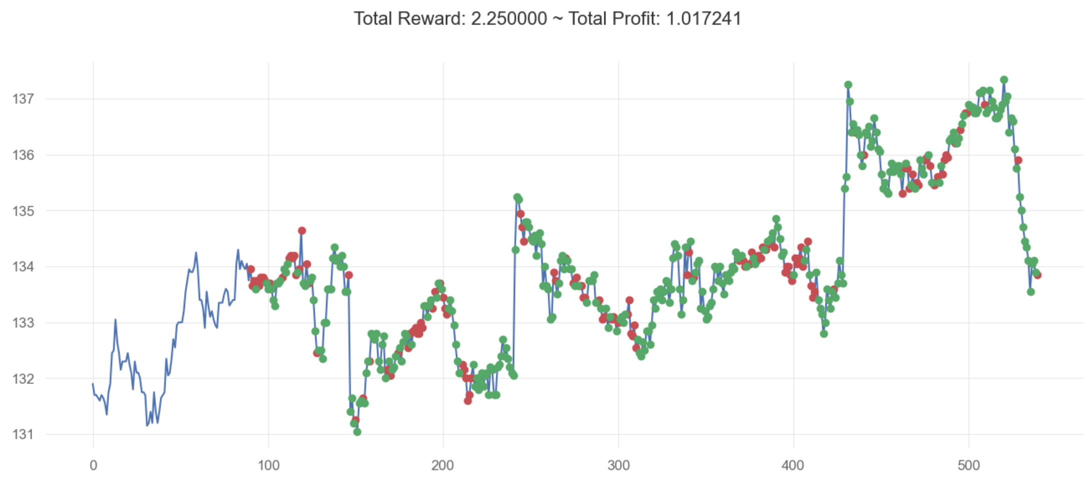


Figure 4.19: Actions taken by an agent that learned using DQN without trading fees. The x-axis represents indexes with the y-axis being the price of the stock.

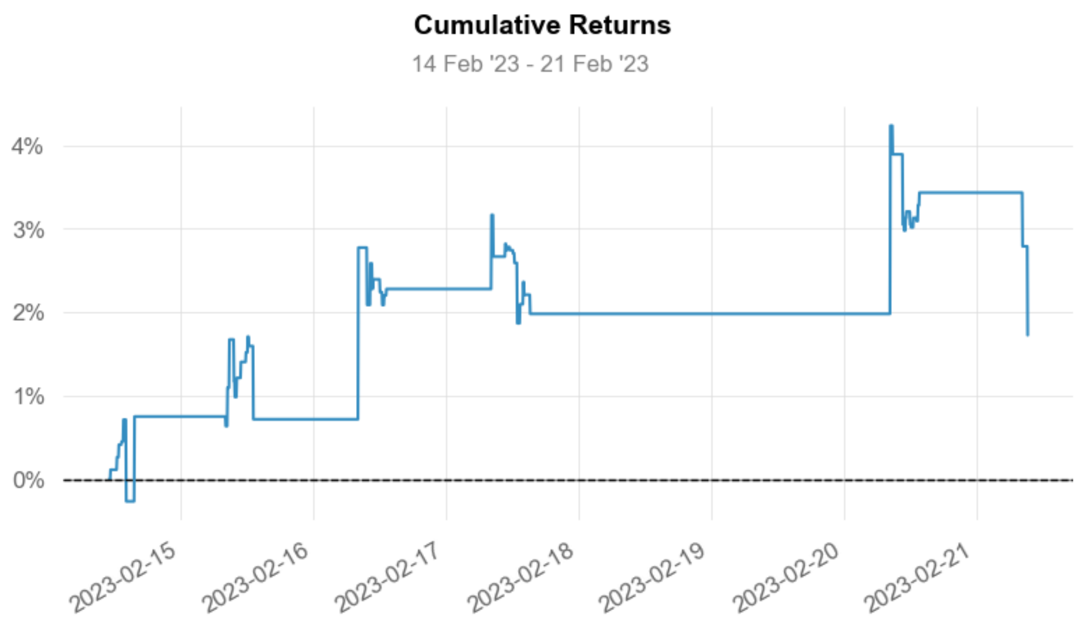


Figure 4.20: Cumulative returns of the strategy taken by the agent that learned using DQN without trading fees.

Table 4.1: Total_profit metric for all of the models in different scenarios.

Models	**Base**	**Extended**	**Ideal**
PPO	0.933	0.952	1.030
A2C	0.839	0.959	1.020
DQN	0.828	0.935	1.017

The PPO model was the most stable and consistent with the best results in all of the testing scenarios. Both A2C and DQN improved significantly when the models were extended with additional input features. With the trading fees removed, all of the strategies learned by the agent became profitable.

4.4 Trading using Q-learner

This time instead of using RL models from stable_baselines3 library a simple Q-learner model was implemented. We still want to create an agent that learns to make optimal trading decisions based on historical price data. We will assumme no trading fees for this scenario. The idea behind Q-learning is to make the agent to learn a policy that maximizes the cumulative reward. The model works as follows:

1. Load historical intraday price data from a CSV file.
2. There are three states that represent the price movement between consecutive time steps:
   - "up",
   - "down",
   - "stable".
3. As well as three actions representing possible trading decisions:
   - "buy",
   - "sell",
   - "hold".
4. Split the data into training and testing set. Set up model's parameters
5. For each training episode, the agent will make actions using an epsilon-greedy policy. That means that with probability ϵ, the agent will try to explore new actions, and with probability $1 - \epsilon$ he will choose an action based on what he has already learned.

Compared to the previous models, the results are quite remarkable.

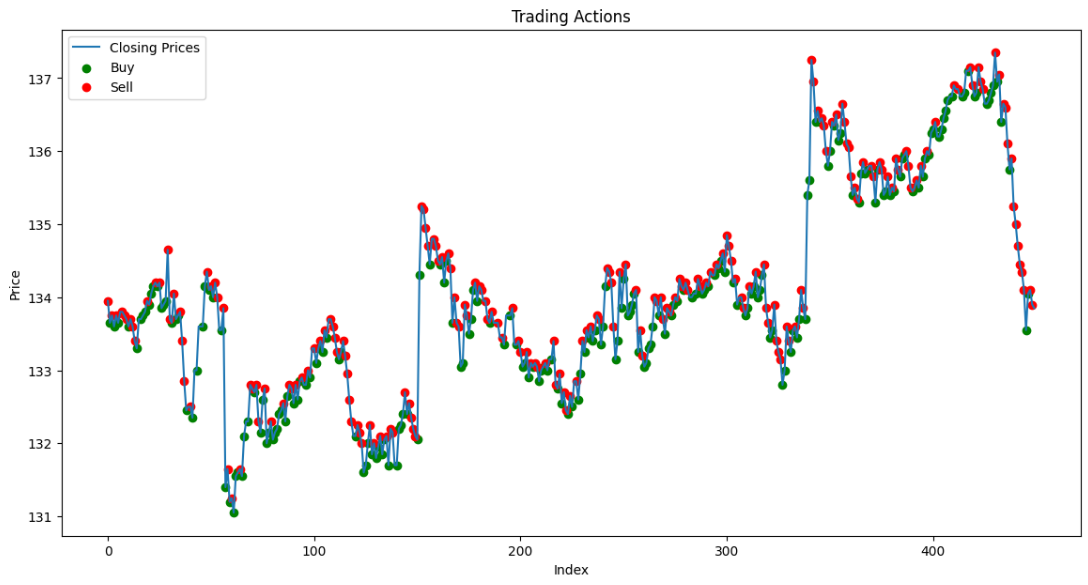


Figure 4.21: Actions taken by an agent that learned using Q-learner model.

From the actions plot we can clearly see that the agent bought low and sold high. The cumulative return plot shows that during the specified window of time, the agent was able to make about 9% profit indicating that Q-learning might be the best option for price prediction going forward.

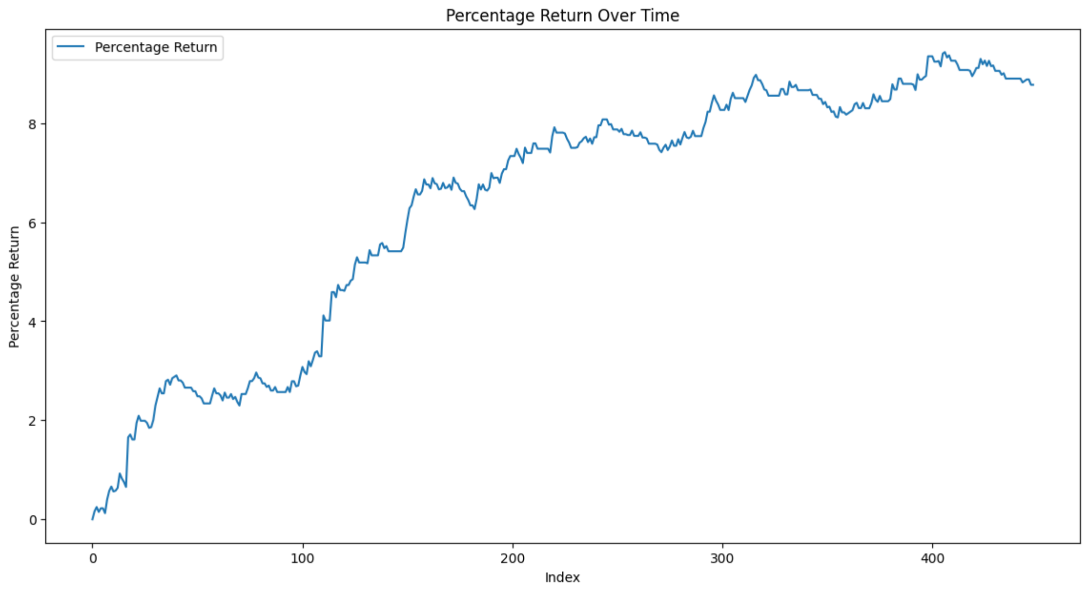


Figure 4.22: Cumulative returns of the strategy taken by agent that learned using Q-learner model.

Extending this model with custom indicators and improved actions/states would be an interesting topic for future research as the results with the base version already outperform other RL models.

Table 4.2: Total_profit metric comparison between previous RL models and the Q-learner.

Model	**No fees environment**
Q-learner	1.089
PPO	1.030
A2C	1.020
DQN	1.017

Conclusion

In summary, this thesis demonstrated the application of machine learning and reinforcement learning techniques to optimize trade execution in the stock market. The models, including Linear Regression, Decision Tree Regressor, Random Forest, SimpleRNN, LSTM, GRU, PPO, A2C, DQN, and Q-learner, were evaluated for their predictive performance and ability to enhance trading strategies.

For simple forecasting problems, Decision Tree Regressor performed better than Linear Regression. Recurrent neural networks performed even better and showed the highest potential for this task as shown by Dutta [5]. Some focus should also be put into predicting trends - showed by the Random Forest example. An important aspect of training ML models was hyperparameter tuning. Results of each model were or could have been improved by using proper parameters. In case of price prediction we were able to increase the percentage from 49% to almost 53% when the parameters of the model were tuned properly.

The results from reinforcement learning models indicate that incorporating advanced indicators and extending the input features can significantly improve the performance of trading agents. We also can clearly see that using RL models to simply predict prices is not working flawlessly and the simplest of them all - Q-learner achieved the best results, indicating the need for further exploration. By including the scenario with no trading fees we can observe models' performances under ideal circumstances.

Future work could explore further refinement of RL models and their application beyond price prediction. Instead of solely relying on predicting the prices, RL models could be use to enhance already existing trading strategies like Time Weighted Average Price (TWAP) [15] or Volume Weighted average price (VWAP) [13]. Further testing needs to be done on Q-learning as it showed the best potential for price prediction. More direct comparison between models with and without trading fees would also bring some more insight into the usage of RL in predicting prices.

Bibliography

[1] Anscombe, F. J. Graphs in statistical analysis. *The American Statistician 27*, 1 (1973), 17–21.

[2] Bishop, C. M. *Pattern Recognition and Machine Learning.* Springer, 2006.

[3] Breiman, L., Friedman, J., Stone, C. J., Olshen, R. *Classification and Regression Trees.* Chapman and Hall/CRC, 1984.

[4] Doe, J., Smith, J. Simple moving average strategies in stock trading. *Journal of Financial Trading 34* (2019), 23–45.

[5] Dutta, A., Kumar, S., Basu, M. A gated recurrent unit approach to bitcoin price prediction. *Journal of Risk and Financial Management 13*, 2 (2020), 23.

[6] Hastie, T., Tibshirani, R., Friedman, J. *The elements of statistical learning: data mining, inference and prediction*, 2 ed. Springer, 2009.

[7] Henderson, P., Islam, R., Bachman, P., Pineau, J., Precup, D., Meger, D. A2c is a special case of ppo. *arXiv preprint arXiv:2205.09123* (2020).

[8] Liashchynskyi, P., Liashchynskyi, P. *Grid Search, Random Search, Genetic Algorithm: A Big Comparison for NAS.* 2019.

[9] Moallemi, C. C., Wang, M. A reinforcement learning approach to optimal execution. *SSRN Electronic Journal* (2021).

[10] Montgomery, D. C., Peck, E. A., Vining, G. G. *Introduction to Linear Regression Analysis*, 6th ed. John Wiley & Sons, 2021.

[11] OsQuant. Key insights: Imbalance in the order book, 2022.

[12] Poon, S.-H., Granger, C. W. J. *Forecasting Volatility in Financial Markets*, 2nd ed. Wiley, New York, 2005.

[13] QuantInsti. Algorithmic trading and vwap. *Quantitative Finance Journal* (2020).

[14] Schulman, J., Wolski, F., Dhariwal, P., Radford, A., Klimov, O. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347* (2017).

[15] Springer, X. Price impact equilibrium with transaction costs and twap trading. *Springer* (2021).

[16] Sutton, R. S., Barto, A. G. *Reinforcement Learning: An Introduction*, 2 ed. MIT Press, 2018.

[17] Xiong, L., Zhang, L., Huang, X., Yang, X., Huang, W., Zeng, H., Tang, H. Dcast: A spatiotemporal model with densenet and gru based on attention mechanism. *Mathematical Problems in Engineering 2021* (02 2021).

[18] Yang, L., Shami, A. On hyperparameter optimization of machine learning algorithms: Theory and practice. *Neurocomputing 415* (2020), 295–316.

[19] Yoon, C. Deep q networks: From theory to implementation. *Towards Data Science* (2023).

[20] Zhang, D., Peng, Q., Lin, J., Wang, D., Liu, X., Zhuang, J. Simulating reservoir operation using a recurrent neural network algorithm. *Water* (04 2019).

[21] Zheng, C., He, J., Yang, C. Optimal execution using reinforcement learning. *arXiv preprint arXiv:2306.17178* (2023).