Reinforcement Learning for Stock Trading

Rophence Ojiambo & Anusha Kumar

Contents

1 Introduction

1.1 Learning by Reward

It may accidentally mix whites and colors or add too much detergent, resulting in negative rewards. However, through trial and error, the robot gradually learns the correct sequence of actions to take when doing laundry. As the robot continues to do laundry over time, the robot starts to associate successfully washing and drying laundry with receiving a high-five and starts to optimize its laundry skills to maximize the chances of getting a high-five. It refines its strategy and learns to adapt to several types of fabrics and stains. It may adjust the water temperature, or the amount of detergent based on the type of fabric being washed or selecting the correct cycle.

Over time, the robot becomes your skilled laundry assistant, capable of doing laundry efficiently and effectively. Now imagine that you change the scenario by adding a new type of fabric or stain that the robot has never encountered before. The robot may initially struggle to determine the correct cycle or water temperature or detergent amount to use. However, by experimenting with different settings and observing the results, the robot can learn to adjust its strategy to handle the new fabric or stain type. These are just a few examples of reinforcement learning. The robot/monkeys are the agents, the shot/laundry task are the actions, the banana/high-five are the rewards, and the association between the action and the reward is the policy. By using a reward signal to guide the agents' behavior, you can train them to learn a specific task or behavior. This also demonstrates how reinforcement learning algorithms can adapt to new and unexpected situations, even in complex tasks like doing laundry.

1.2 Stock Trading

Now, here's where things get tricky. RL isn't like other games where they can just hit "reset" and start over if you mess up. They have to learn from their mistakes and make smarter choices next time. So, one should do their research, pay attention to trends, and don't be afraid to take risks (but not too many!). It's all about balancing risk and reward, just like in RL.

In the world of stock trading, timing is everything. It is no secret that the stock market is unpredictable, with prices rising and falling based on everything from global events to social media trends. Recently, stocks like Bitcoin, Dogecoin, GameStop and Tesla have captured the attention of investors and the public alike[1]. Whether it is buying low and selling high or knowing when to hold onto a stock for the long term, making the right decision can mean the difference between profit and loss. But how can traders stay on top of market trends and make informed decisions in such a volatile environment? Is it possible to automate trading entirely for traders who want to trade 24/7 but do not want to be glued to their screens all the time? One answer is reinforcement learning, a form of machine learning that trains algorithms to learn from past actions and outcomes and make better decisions over time[2,3]. In this project, we will explore the application of reinforcement learning in stock trading and how this can potentially change the way we think about financial investment decisions.

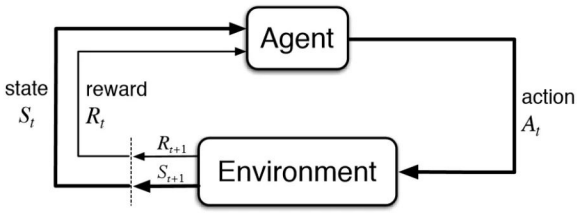
2 What is Reinforcement Learning?

Reinforcement learning (RL) is a type of machine learning algorithm that enables an agent to learn how to make decisions based on rewards and punishments. It involves an agent interacting with an environment, taking actions, receiving feedback, and adjusting its behavior based on that feedback[3]. The goal of RL is to optimize the agent's behavior to maximize the cumulative reward it receives over time. RL algorithms are used in various applications such as robotics, gaming, finance, and natural language processing[4–7]. RL is based on the concept of trial and error. We can consider an individual placed in an unfamiliar environment. Initially, mistakes may occur, but through learning from them, they can avoid repeating them in the future when faced with similar circumstances. RL employs a similar approach in training its model, whereby the

agent tries different actions and observes the resulting rewards or punishments, and then adjusts its behavior to maximize the expected reward in the future.

2.1 Mathematical Nature of Reinforcement Learning

We can illustrate the nature of reinforcement learning as follows: at each sequence of discrete time points (or steps), with t = 0, 1, 2, 3, etc., the agent and environment interact. The agent receives information about the environment's state, s_t within S, at each time point/step, t, where S represents the set of possible states, which on this basis, selects an action from A(s_t), the set of actions to choose from. The agent receives the numerical reward within R, and then finds itself under a new state[8]. The diagram below illustrates this process:



The agent-environment interaction in RL framework (Source: Sutton and Barto, 2018)

The goal of reinforcement learning (RL) can be summarized as finding a policy that maximizes the expected cumulative reward over a sequence of interactions between an agent and its environment. Mathematically, this can be expressed as:

$$\boldsymbol{maximize}\ \boldsymbol{E}[\boldsymbol{R}] = \max_{\boldsymbol{\pi}} \boldsymbol{E}\left[\sum_{\boldsymbol{t=0}}^{\boldsymbol{\infty}} \boldsymbol{\gamma^t r_t}\right]$$

where $\boldsymbol{E}[\boldsymbol{R}]$ is the expected cumulative reward, $\boldsymbol{\gamma}$ is the discount factor ($0 \leq \gamma \leq 1$), $\boldsymbol{r_t}$ is the reward obtained at time $\boldsymbol{t}$, and the summation is taken over all time steps from $\boldsymbol{t = 0}$ to $\boldsymbol{t = \infty}$.

The objective function in RL is typically the expected cumulative reward, which is also known as the return. This function is used to evaluate the performance of a policy, which maps states to actions. The objective is to find the policy that maximizes the expected return over a given time horizon.

The objective function can be expressed as:

$$\boldsymbol{J(\pi)} = \boldsymbol{E_\pi}\left[\sum_{\boldsymbol{t=0}}^{\boldsymbol{\infty}} \boldsymbol{\gamma^t r_t}\right]$$

where $\boldsymbol{J(\pi)}$ is the expected return for policy $\boldsymbol{\pi}$, $\boldsymbol{E_\pi}$ is the expectation over the distribution of trajectories generated by following policy $\boldsymbol{\pi}$, and the summation is taken over all time steps from $\boldsymbol{t = 0}$ to $\boldsymbol{t = \infty}$.

The goal of RL is to find the policy $\boldsymbol{\pi^*}$ that maximizes the objective function $\boldsymbol{J(\pi)}$, i.e.,

$$\boldsymbol{\pi^* = \arg\max_{\pi} J(\pi)}$$

Since the challenge that remains is to nudge the policy parameters such that the return can be maximized, we can meet this goal by using the gradient of the objective function. This is known as vanilla gradient ascent. We should note that gradient descent and gradient ascent are synonymous, with only a sign change being the notable difference (sign change is moved to downhill for gradient descent)[9].

Policy Gradient Theorem

$$
\begin{aligned}
\nabla J(\theta) &\doteq \nabla \mathbb{E}_{\pi}\left[G \mid s, a\right] && (1)\\
&= \nabla \sum_{a \in \mathscr{A}}\left[Q(s, a)\pi(a \mid s)\right] && (2)\\
&\propto \sum_{a \in \mathscr{A}}\left[Q(s, a)\nabla\pi(a \mid s)\right] && (3)\\
&\propto \sum_{a \in \mathscr{A}}\left[Q(s, a)\pi(a \mid s)\frac{\nabla\pi(a \mid s)}{\pi(a \mid s)}\right] && (4)\\
&\propto \sum_{a \in \mathscr{A}}\left[Q(s, a)\pi(a \mid s)\nabla \ln \pi(a \mid s)\right] && (5)\\
&\propto \mathbb{E}_{\pi}\left[G\nabla \ln \pi(a \mid s)\right] && (6)
\end{aligned}
$$

The gradient of a policy (Source: D, P.W.P., 2020[9])

- In step 1, we observe that our goal is to find the gradient of the objective function.
- Then, within step 2, the expected return is replaced with an implementation (can be any of your choice), however it is usually easier to proceed with the action-value function. We denote the expected return as equivalent to each action's (Q) value multiplied by the probability that the particular action is visited (which is defined by the policy, π).
- We take the gradient with respect to the parameters of the policy. Therefore, we can move Q, the action value function, outside of the calculation of the gradient (as defined in step 3).
- Proceeding with our steps, for step 4, we multiply the entire equation by $\pi(\text{a} \mid \text{s})/\pi(\text{a} \mid \text{s})$, which is equivalent to one. Therefore, the equation is not altered.
- For step 5, we proceed with a typical mathematical identity, which replaces the fraction with a natural logarithm. This is a fairly standard computational trick, where we prevent the multiplication of small probabilities, which can make the gradients zero (also known as the vanishing gradient problem).
- Finally, for step 6, we can remember that the summation of action values, which are multiplied by the probabilities for the action, are what we started with in step 2. However, as a substitute to summing over the actions, we instruct the agent to adhere by the policy and return the expectation of the specified action over time[9].

2.2 Key Elements of Reinforcement Learning

- **Agent:** The agent in RL is the decision-making entity that interacts with the environment. In the stock trading analogy, the agent could be the trader who decides when or which stocks to buy and sell.
- **Environment:** The environment in RL is the world in which the agent operates and interacts with. In the stock trading analogy, the environment would be the stock market, which includes all the stocks, their prices, and the various events that affect the market.
- **State (S):** A state in RL refers to the current situation of the environment, as observed by the agent. In the stock trading analogy, a state would include the current prices of the stocks the trader is interested in, the current state of the economy, and any other relevant factors that may influence the decision to buy or sell.
- **Policy (π):** A policy in RL is a set of rules or instructions that the agent uses to determine its actions in a given state. In the stock trading analogy, a policy could be the algorithm that the agent uses to determine which stocks to buy and sell based on the current state.
- **Action (A):** In RL, the action is the decision made by the agent based on the current state and the policy. In the stock trading analogy, the action would be the trader's decision to hold, buy or sell a specific stock.
- **Reward (R):** A reward in RL is a feedback signal that the agent receives after taking an action. It indicates how good or bad the action was in achieving the agent's objective. In the stock trading analogy, the reward would be the profit or loss made from the trade.

- **Penalty:** A penalty in RL is a negative feedback signal that the agent receives after taking an action that is not desired or expected. In the stock trading analogy, the penalty would be the loss incurred from a bad trade or missed opportunity..

2.3 Advantages of Reinforcement Learning

- Learning through trial and error: Unlike supervised learning, where the algorithm is provided with labeled data, RL algorithms learn by trial and error, without explicit supervision thus making it well-suited for applications where it is difficult or impractical to provide labeled data.
- Flexibility: RL can be used in both single-agent and multi-agent settings. In a single-agent setting, the agent learns to optimize its behavior by interacting with the environment while in a multi-agent setting, multiple agents learn to interact with each other and optimize their behavior collectively.
- Adaptability: RL algorithms can adapt to changes in the environment and adjust their behavior accordingly. This makes RL well-suited for dynamic and uncertain environments where traditional machine learning approaches may struggle.
- Exploration: RL algorithms are designed to explore new strategies and actions in order to maximize their rewards. This can lead to the discovery of novel solutions and approaches that may not have been considered otherwise.

2.4 Challenges of Reinforcement Learning

- Exploration-exploitation tradeoff: RL algorithms need to balance between exploring the environment to find new and potentially better actions, and exploiting the knowledge they already have to maximize rewards in the past. This tradeoff can be challenging, especially in complex environments with many possible actions.
- Generalization: RL algorithms may struggle to generalize their learned policies to new, unseen environments or tasks that were not encountered during training, which can limit their usefulness in practice. Data efficiency: RL algorithms typically require a large amount of data to learn a good policy, which can be time-consuming and expensive to obtain.
- Reward engineering: The quality of the learned policy depends heavily on the reward function used, which can be difficult to design in a way that accurately reflects the desired behavior.
- Safety and ethics: RL agents can learn to take actions that are harmful or unethical, especially if the reward function is not carefully designed or the agent's behavior is not appropriately constrained.
- Interpretability: RL algorithms can be difficult to interpret and explain, especially when they use complex models or operate in high-dimensional state and action spaces.

3 Reinforcement Learning Algorithms

There are several categories of reinforcement learning algorithms that can be used for stock trading, including:

- **Model-Based Reinforcement Learning Algorithms:** These algorithms learn the model from data and use it to optimize the agent's behavior and plan actions that maximize the expected cumulative reward to predict the outcomes of actions. Model-based algorithms are computationally efficient and require less data to learn than model-free algorithms. However, they may suffer from errors in the learned model, which can lead to sub optimal behavior. Examples of model-based reinforcement learning algorithms for stock trading include: Dynamic Programming, Monte Carlo methods and Temporal Difference Learning

- **Value-Based Reinforcement Learning Algorithms:** These are model-free that learn an estimate of the optimal value function and use it to derive an optimal policy. These algorithms are more robust to errors in the environment model, but they require more data to learn and can be computationally expensive. Examples of value-based reinforcement learning algorithms for stock trading include: Q-Learning, Deep Q-Networks (DQNs), and Double DQNs.
- **Policy-Based Reinforcement Learning Algorithms:** These algorithms learn the optimal policy directly, without estimating the value function. Examples of policy-based reinforcement learning algorithms for stock trading include:REINFORCE, SARSA, Proximal Policy Optimization (PPO) and Actor-Critic
- **Hybrid Reinforcement Learning Algorithms:** These algorithms combine elements of value-based and policy-based reinforcement learning. Examples of hybrid reinforcement learning algorithms for stock trading include: Trust Region Policy Optimization (TRPO) and Asynchronous Advantage Actor-Critic (A3C)

  In practice, the choice of reinforcement learning algorithm for stock trading depends on the specific task and the characteristics of the environment. For example, a value-based algorithm like DQN may be well-suited for a simple trading environment with discrete actions, while a policy-based algorithm like PPO may be better for a more complex environment with continuous actions.

  Each of these algorithms has different advantages and disadvantages, and their performance can vary depending on the specific problem being addressed and on several factors, such as: (1)Efficiency and speed of convergence, (2) Performance on historical data, (3) Robustness to market changes, (4) Ability to handle high-dimensional state and action space, and (5) Interpretability: how easily the algorithm's decisions can be interpreted and understood. Ultimately, the choice of algorithm will depend on the specific goals of the trader, the particular characteristics of the stock market being traded, and the desired performance metrics.

Next, we expand more on the algorithms we will use for our data

3.1 1. Q-Learning

Q-learning[10] is a popular value-based reinforcement learning algorithm based on the well known Bellman equation:

$$\boldsymbol{V(s) = \mathbb{E}[R_{t+1} + \gamma V(s_{t+1}) | S_t = s]}$$

Where; $\boldsymbol{V(s)}$ is the value of the current state s, $\mathbb{E}$ refers to the expectation, while γ refers to the discount factor that determines the importance of future rewards. From the above definition of the Bellman's equation, the action value function can be expressed as:

$$Q^{\pi}(s,a) = \mathbb{E}[r_{t+1} + \lambda r_{t+2} + \lambda^2 r_{t+3} + \ldots | s, a]$$

$$= \mathbb{E}_{s'}[r + \lambda Q^{\pi}(s', a') | s, a]$$

In Q-learning, the agent uses an iterative approach to update the Q-function estimates and learn an estimate of the optimal action-value based on the rewards obtained from each function. The optimal Q-value, denoted as Q* can be expressed using the law of total probability as:

$$\boldsymbol{Q^*(s,a) = \mathbb{E}s' \sim p(\cdot|s,a)\Big[r + \gamma \max_{a'} Q^*(s', a')\Big| s, a\Big]}$$

where $\boldsymbol{s' \sim p(\cdot|s,a)}$ denotes the next state sampled from the transition probability distribution, $\boldsymbol{p(\cdot|s,a)}$, $\boldsymbol{r}$ is the immediate reward obtained after taking action $\boldsymbol{a}$ in state $\boldsymbol{s}$.

In the context of stock trading, Q-learning can be used to learn the optimal buying and selling decisions for a given stock, based on historical price data and other market indicators. The agent can learn to maximize its expected profit over a given time horizon, taking into account the risks and uncertainties associated with stock trading.

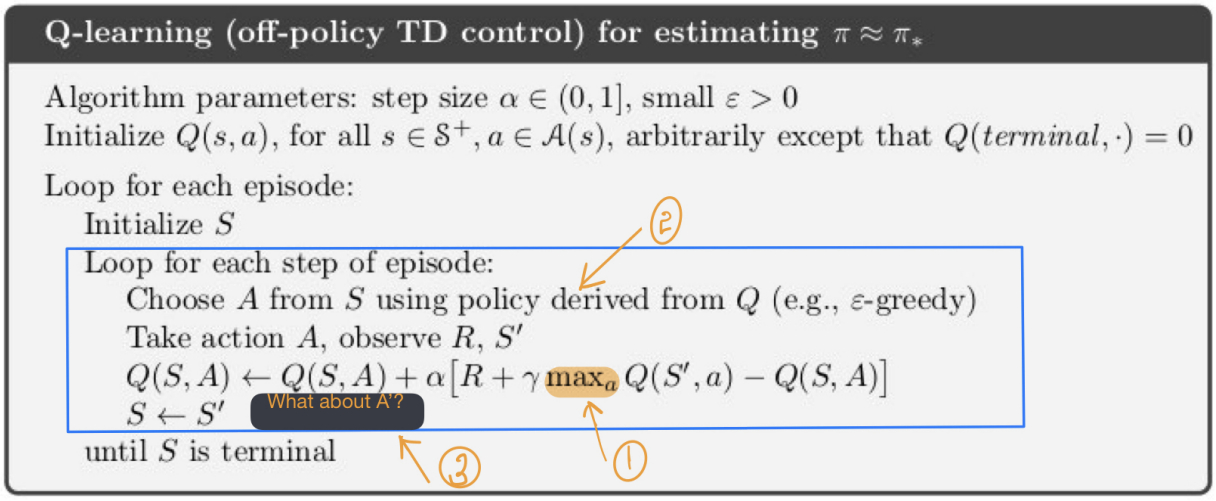
The Q-learning algorithm can be described by the following update rule:

$$\boldsymbol{Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha\Big[R_{t+1} + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, A_t)\Big]}$$

where:

- $\boldsymbol{Q(S_t, A_t)}$ is the current estimate of the Q-value for state-action pair - $(\boldsymbol{S_t}, \boldsymbol{A_t})$,
- $\boldsymbol{R_{t+1}}$ is the immediate reward obtained after taking action
- $\boldsymbol{A_t}$ in the next state $\boldsymbol{S_{t+1}}$,
- $\boldsymbol{\alpha}$ is the learning rate,
- $\boldsymbol{\gamma}$ is the discount factor, and
- $\boldsymbol{\max_a Q(S_{t+1}, a)}$ is the maximum Q-value over all possible actions in the next state $\boldsymbol{S_{t+1}}$.

The algorithm is summarized as below:



3.2 2. State-Action-Reward-State-Action (SARSA)

SARSA (State-Action-Reward-State-Action) is a RL algorithm that is used for online and on-policy learning. It is similar to Q-learning, but instead of updating the Q-value of the current state-action pair using the maximum Q-value of the next state, SARSA updates the Q-value using the Q-value of the next state-action pair, that is, it learns the Q-value of the policy being followed[2]. The SARSA algorithm can be described by the following update rule:

$$\boldsymbol{Q(S_t, A_t) \leftarrow Q(S_t, a_t) + \alpha[r_{t+1} + \gamma Q(S_{t+1}, a_{t+1}) - Q(S_t, a_t)]}$$

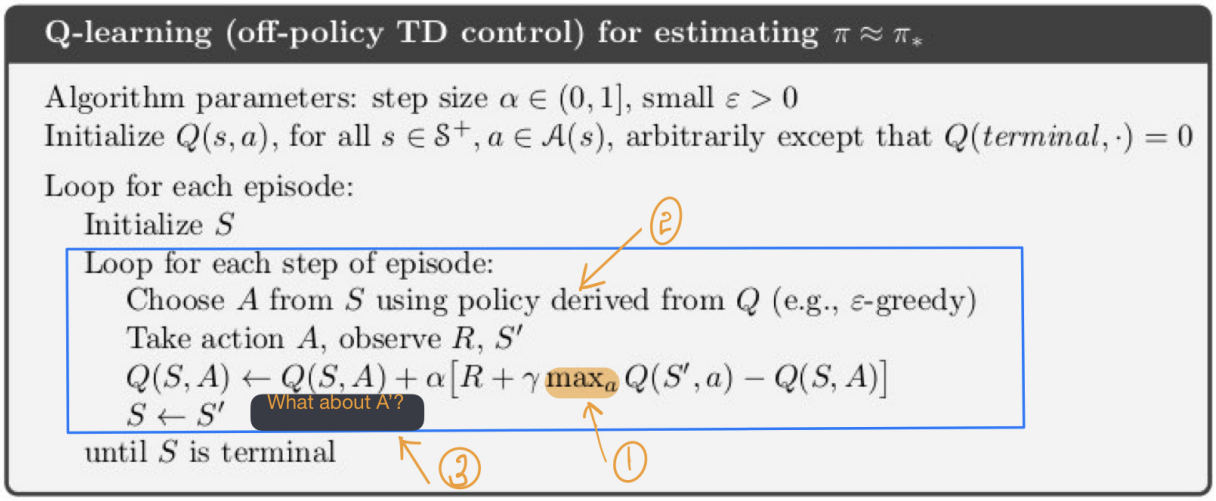
Where:

- $\boldsymbol{Q(S_t, a_t)}$ is the Q-value of the current state-action pair (S_t, A_t)
- $\boldsymbol{\alpha}$ is the learning rate that determines the impact of the new information on the existing Q-value

- $\boldsymbol{r_{t+1}}$ is the immediate reward obtained after taking action A_t in state S_t
- γ is the discount factor that determines the importance of future rewards
- $\boldsymbol{Q(S_{t+1}, a_{t+1})}$ is the Q-value of the next state-action pair (S_{t+1}, a_{t+1}) under the current policy. This is the value of the action selected by the agent in the next state.
- $\boldsymbol{Q(S_t, a_t)}$ is the current estimate of the Q-value of state-action pair (S_t, a_t).

This equation updates the estimate of the Q-value of the current state-action pair by adding the difference between the observed reward and the estimate of the next state-action pair's Q-value, multiplied by the learning rate α.

The algorithm is summarized as below:



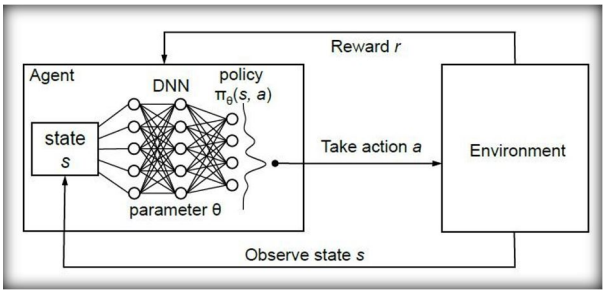
3.2.1 What then is the difference between Q-Learning and SARSA?

From the Q-Learning and SARSA pseudo codes illustrated in the previous sections, the blue boxes emphasize the portion where the two algorithms exhibit a distinction, while the numbers highlight a more intricate difference as elaborated below.

1. The key difference between SARSA and Q-learning lies in how the Q-value is updated after each action. In SARSA, the Q-value is updated based on the Q-value of the action chosen using an epsilon-greedy policy. In contrast, Q-learning selects the maximum Q-value over all possible actions for the next step, which effectively means that no exploration is performed at this stage.
2. Despite this difference, Q-learning still selects the action based on an epsilon-greedy policy when taking an actual action. Therefore, the "Choose A ..." step is included in the repeat loop.
3. In Q-learning, the action for the next step is still selected using an epsilon-greedy policy, following the logic of the loop.

3.3 3. Deep Q-network (DQN)

Deep Q-Network (DQN), is a type of RL algorithm that was introduced in 2015 and uses deep neural networks to approximate the Q-values of state-action pairs in a Markov decision process[11].



The agent-environment interaction in Deep Reinforcement Learning[12]

The DQN algorithm uses experience replay and target networks to improve the stability and efficiency of the learning process. Experience replay involves storing the agent's experiences (i.e., state, action, reward, next state) in a replay buffer and sampling mini-batches of experiences randomly from the buffer to train the neural network.

Target networks involve creating a separate neural network with the same architecture as the Q-network but with frozen weights to estimate the target Q-values used in the Bellman equation. This reduces the correlation between the target and predicted Q-values, which improves the stability of the learning process.

The update equation for Deep Q-Network (DQN) algorithm can be written as follows:

$$\boldsymbol{Q(s,a) = Q(s,a) + \alpha[r + \gamma \max_{a'} Q'(s',a') - Q(s,a)]}$$

Where:

- $\boldsymbol{Q(s,a)}$ is the estimated Q-value for state s and action a;
- $\boldsymbol{r}$ is the reward obtained after taking action a in state s;
- $\boldsymbol{s'}$ is the next state after taking action a in state \$s;
- a' is the action with the highest Q-value in state s';
- $\boldsymbol{Q'(s',a')}$ is the target Q-value for the next state-action pair, computed using a separate target network that is periodically updated with the main network;
- $\boldsymbol{\alpha}$ is the learning rate;
- $\boldsymbol{\gamma}$ is the discount factor, which determines the weight given to future rewards.

The algorithm is summarized as below:

Algorithm 1 Deep Q-learning with Experience Replay

Initialize replay memory $\mathcal{D}$ to capacity N
Initialize action-value function Q with random weights
for episode $= 1, M$ **do**
  Initialise sequence $s_1 = \{x_1\}$ and preprocessed sequenced $\phi_1 = \phi(s_1)$
  for $t = 1, T$ **do**
    With probability ϵ select a random action a_t
    otherwise select $a_t = \max_a Q^*(\phi(s_t), a; \theta)$
    Execute action a_t in emulator and observe reward r_t and image x_{t+1}
    Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$
    Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in $\mathcal{D}$
    Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from $\mathcal{D}$
    Set $y_j = \begin{cases} r_j & \text{for terminal } \phi_{j+1} \\ r_j + \gamma \max_{a'} Q(\phi_{j+1}, a'; \theta) & \text{for non-terminal } \phi_{j+1} \end{cases}$
    Perform a gradient descent step on $(y_j - Q(\phi_j, a_j; \theta))^2$
  end for
end for

3.3.1 Exploration vs Exploitation

The process of learning in RL is through exploration and exploitation[13].

Exploration involves selecting actions that the agent has not taken before and is a crucial aspect of RL as it helps the agent to enhance its current knowledge of each action by discovering new states and actions that can potentially lead to a higher rewards. By improving the precision of the estimated action-values, the agent can make more informed decisions in the future, leading to better performance.

On the other hand, exploitation selects the action that appears to offer the greatest reward by utilizing the agent's current action-value estimates. However, pursuing greedy actions based on action-value estimates may not necessarily lead to the optimal solution, resulting in sub-optimal behavior. Exploration provides the agent with more accurate estimates of action-values, whereas exploitation may yield more rewards. However, the agent cannot pursue both simultaneously, creating a dilemma known as the exploration-exploitation tradeoff.

- **Greedy Policy:** The agent takes the action with the maximum Q value at any state in the environment in the greedy policy exploration. This approach ensures the agent takes the optimal action at each step, but it has the obvious shortcoming of never exploring any other action other than the optimal one, leading to sub-optimal solutions.
- **Epsilon-greedy** exploration is another popular exploration strategy in RL, which involves selecting a random action with a certain probability (epsilon) and selecting the action with the highest expected reward with a probability of 1-epsilon[13,14]. Epsilon-greedy exploration strikes a balance between exploration and exploitation, and it is particularly useful when the agent has some prior knowledge of the environment. However, this strategy can lead to sub optimal solutions if the agent gets stuck in a local maximum.

- **Thompson sampling**[15] is a more sophisticated exploration strategy that involves selecting actions based on the probability of their being optimal. This strategy involves maintaining a probability distribution over the possible rewards of each action and then selecting the action with the highest expected reward according to this distribution. Thompson sampling is particularly useful in situations where the environment is complex and dynamic and where it is not clear which actions will lead to a higher reward. However, this strategy can be computationally expensive and difficult to implement in practice.

When implementing the above mentioned algorithms, we mainly apply the `Epsilon-greedy` policy.

3.4 Previous Work

Reinforcement learning (RL) is an effective approach for building intelligent trading agents in stock markets. Among various RL algorithms, Q-learning, SARSA, and Deep Q-Network (DQN) are the most popular ones. Q-learning is a model-free RL algorithm that uses the Q-function to approximate the optimal policy. The Q-value of each action-state pair is updated using the Bellman equation. A recent study conducted by Chakole et. al[16] used the Q-learning algorithm of Reinforcement Learning to train a trading agent for discovering optimal dynamic trading strategies. They conducted experiments using the proposed models on real stock market data from the Indian and American stock markets. The results showed that the proposed models were more profitable compared to the Buy-and-Hold and Decision-Tree based trading strategies.

SARSA is another popular RL algorithm that is used for trading in stock markets. Unlike Q-learning, SARSA is an on-policy algorithm that uses the SARSA Q-function to estimate the expected future rewards. A recent study[17] involved implementing a RL agent with the SARSA algorithm and testing it on 10 stocks from the Brazilian B3 stock market. The experiments indicated that the RL agent was able to generate high profits with lower risks as compared to a supervised learning agent that employed a LSTM neural network.

Deep Q-Network (DQN) is a popular RL algorithm that uses deep neural networks to approximate the Q-function. Another study[18] aimed to create an end-to-end daily stock trading system using Deep Q-network (DQN) and Deep Recurrent Q-network (DRQN) algorithms to automatically decide whether to buy or sell stocks. The S&P500 ETF was used as the trading asset and daily trading data as the state of the trading environment. The performance of the system was compared to benchmarks of Buy and Hold (BH) and Random action-selected DQN trader, and the results showed that the DQN trader outperformed both benchmarks.

When using RL for trading, the figure below shows a general expectation of the trading decisions made by the interaction of the agent and states in the stock trading environment.

Image Source:QUANT INSTI

4 Application to Stock Data

In this section, we will explore data from the stock market, downloaded from yahoo finance (2010-2022) and show implementation of the RL algorithms discussed in the previous section. Below are the variable descriptions for stock data:

symbol: The name or ticker symbol of the stock, Apple (AAPL), Amazon (AMZN), Johnson & Johnson stock (JNJ), and NFLX.

date: The date of the stock price.

open: The opening price of the stock on a given day.

high: The highest price that the stock traded at during the day.

low: The lowest price that the stock traded at during the day.

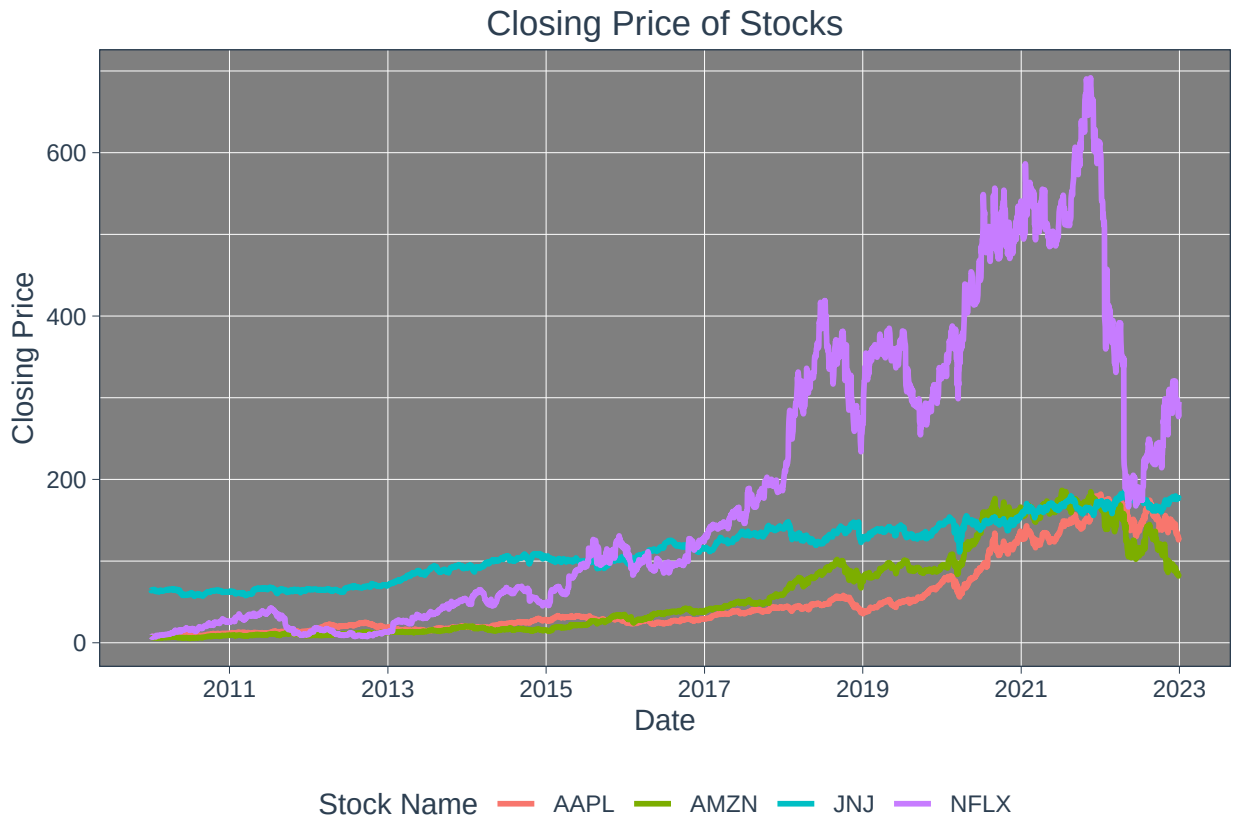
close The price at which the stock closed for trading on that particular day.

volume: The number of shares of the stock that were traded on a given day.

adjusted: The adjusted closing price of the stock on a given day, which takes into account any corporate actions, such as stock splits or dividends, that affect the stock price.

To download stock data for Apple, Amazon, Johnson & Johnson, and Netflix in R, we can use the `tidyquant` package.

```r
library(tidyquant) # for stock data

# Define a vector with the stock symbols of interest
symbols <- c("AAPL", "AMZN", "JNJ", "NFLX")

# Use the tq_get() function to download the stock data for each symbol and bind them into
↪  one data frame
stocks_data <- symbols %>%
  tq_get(get = "stock.prices", from = "2010-01-01", to = "2022-12-31") %>%
  group_by(symbol) %>%
  mutate(date = as.Date(date)) %>%
  ungroup() %>%
  select(symbol, date, everything())
```



4.1 Implementation of Q-learning

Our code implements the Q-learning algorithm to create a simple trading strategy for stocks. The Q-learning algorithm is a model-free reinforcement learning technique used to learn an optimal policy based on trial and error. It involves building a `Q-table` that maps the current state and action to the expected future reward. In this case, the state is the current stock price, and the action is either to buy or hold/sell.

The function `Q_learning` takes a data frame of stock prices and applies the Q-learning algorithm to each stock to create a Q-matrix that stores the expected future rewards for each state-action pair. It also creates a list of Boolean Q-tables for each stock, which specify whether to buy or hold/sell at each time step based on the Q-matrix. Finally, it returns the Q-matrix, actions taken, and Q-table for each stock.

Our code then loads stock data for four companies `AAPL`, `AMZN`, `JNJ`, `NFLX`, extracts the adjusted prices, and combines them into a matrix. It then applies the `Q_learning` function to each stock to create a list of Q-matrices, actions taken, and Q-tables.

The `Q_learning_summary` function generates a summary table of the actions taken for each stock, converts the action codes to action names, and splits the summary table by stock. It then creates tables for action for each stock and access the Q matrix for each stock. The `predict_prices` function predicts the prices of a stock using the Q matrix and the current prices of the stock. All the functions mentioned can be found in Q-learning file.

```
# Q_learning function

Q_learning <- function(prices) {
```

```r
  # Define the initial values of the Q-matrix and the learning parameters
  n <- nrow(prices)
  n_stocks <- ncol(prices)
  Q_list <- vector("list", n_stocks)
  for (j in 1:n_stocks) {
    Q_list[[j]] <- matrix(0, nrow = n, ncol = 3) # Q-matrix for stock j with 3 actions
  }

  alpha <- 0.2 # Learning rate
  gamma <- 0.9 # Discount factor

  # Initialize an empty vector to store the actions taken
  actions_taken <- vector("numeric", n-1)

  # Loop through each time step
  for (i in 2:n) {
      # Loop through each stock
    for (j in 1:n_stocks) {
      # Define the state at time t-1
      state <- prices[i-1, j]

      # Select the action with the highest Q-value
      q_values <- Q_list[[j]][i-1, ]
      action <- sample(which(q_values == max(q_values)), 1)

      # Store the action taken at time t-1
      actions_taken[i-1] <- action

      # Define the reward and the next state at time t
      reward <- prices[i, j] - prices[i-1, j] # The reward is the change in the price of
↪  stock j
      next_state <- prices[i, j]

      # Update the Q-matrix using the Q-learning algorithm
      Q_list[[j]][i, action] <- Q_list[[j]][i-1, action] + alpha * (reward + gamma *
↪  max(Q_list[[j]][i-1, ]) - Q_list[[j]][i-1, action])
    }
  }

  # Create a list of Q-tables for each stock
  Q_table <- lapply(Q_list, function(x) {
    buy_action <- x[, 1] > x[, 2] & x[, 1] > x[, 3] # Buy if Q-value of buy is greater
↪  than Q-value of hold/sell
    sell_action <- x[, 2] > x[, 1] & x[, 2] > x[, 3] # Sell if Q-value of sell is greater
↪  than Q-value of hold/buy
    hold_action <- !buy_action & !sell_action # Hold if none of the above is true
    data.frame(Buy = buy_action, Sell = sell_action, Hold = hold_action)
  })

  # Return the list of Q-matrices (one for each stock), actions taken, and Q-tables
  return(list(Q_list = Q_list, actions_taken = actions_taken, Q_table = Q_table))
}
```

Table 1: Summary of Actions taken

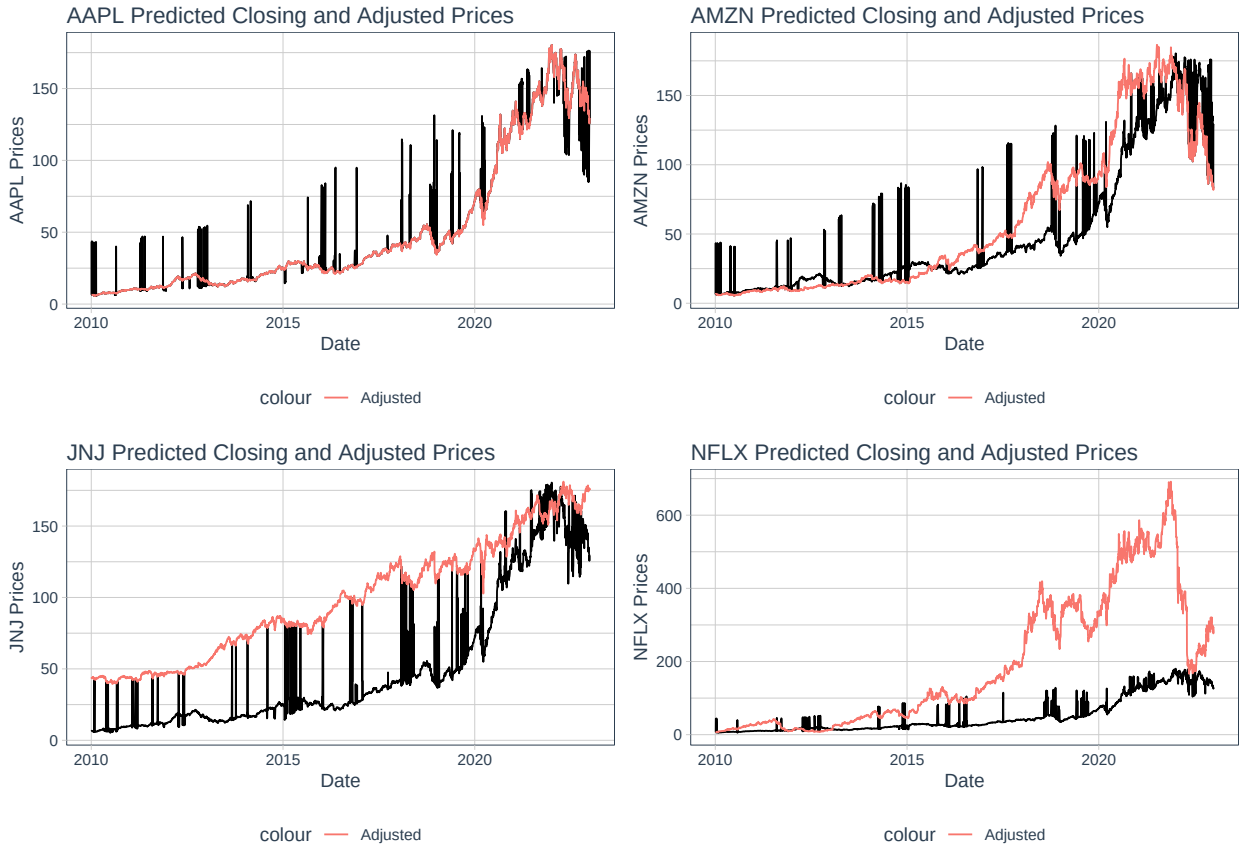
	Stocks			
	AAPL	**AMZN**	**JNJ**	**NFLX**
Buy	2958	2958	2958	2958
Sell	157	157	157	157
Hold	156	156	156	156

4.1.1 Summary tables for action taken

The table shows the frequency of each action `Buy`, `Sell`, `Hold` taken for each stock. For all the mentioned stocks, the algorithm took the action `Buy` 2958 times, `Sell` 157 times, and `Hold` 156 times during the trading.

4.1.2 Plot for predicted closing prices and Adjusted prices

The plots shows that the algorithm was able to predict closing prices with significant accuracy since the predicted closing prices are not deviating too much from the adjusted prices.



4.1.3 Performance Check

We show the performance of the predictions using mean squared error for four stocks (AAPL, AMZN, JNJ, and NFLX). For each stock, the code first predicts the prices using the `Q matrices` trained on the training

	AAPL	AMZN	JNJ	NFLX
MSE	161.2266	1006.65	3317.487	91061.72

data (2010 to 2016) and then calculates the mean squared error between the predicted prices and the actual prices in the testing data (2017 to 2022).

The mean squared error for AAPL is 161.2266, for AMZN it is 1006.65, for JNJ it is 3317.487, and for NFLX it is 91061.72. A lower mean squared error indicates better performance, therefore the predictions for AAPL are the most accurate among the four stocks.

4.2 Implementation of SARSA

The SARSA code provided is an implementation of the SARSA (State-Action-Reward-State-Action) reinforcement learning algorithm to train an agent to make buy, sell, or hold decisions for the stocks used in this project. The code is set up to train the agent on a subset of the data (2010-2016) and test the agent on the remaining data (2017-2022).

```
# extract closing prices from data
closing_prices <- train$close

# define state based on previous 3 closing prices
define_state <- function(closing_prices, i) {
  if (i < 4) {
    # if not enough data points yet, use the first one as the state
    state <- closing_prices[1]
  } else {
    # use previous 3 closing prices as the state
    state <- closing_prices[(i-3):i]
  }
  return(state)
}
```

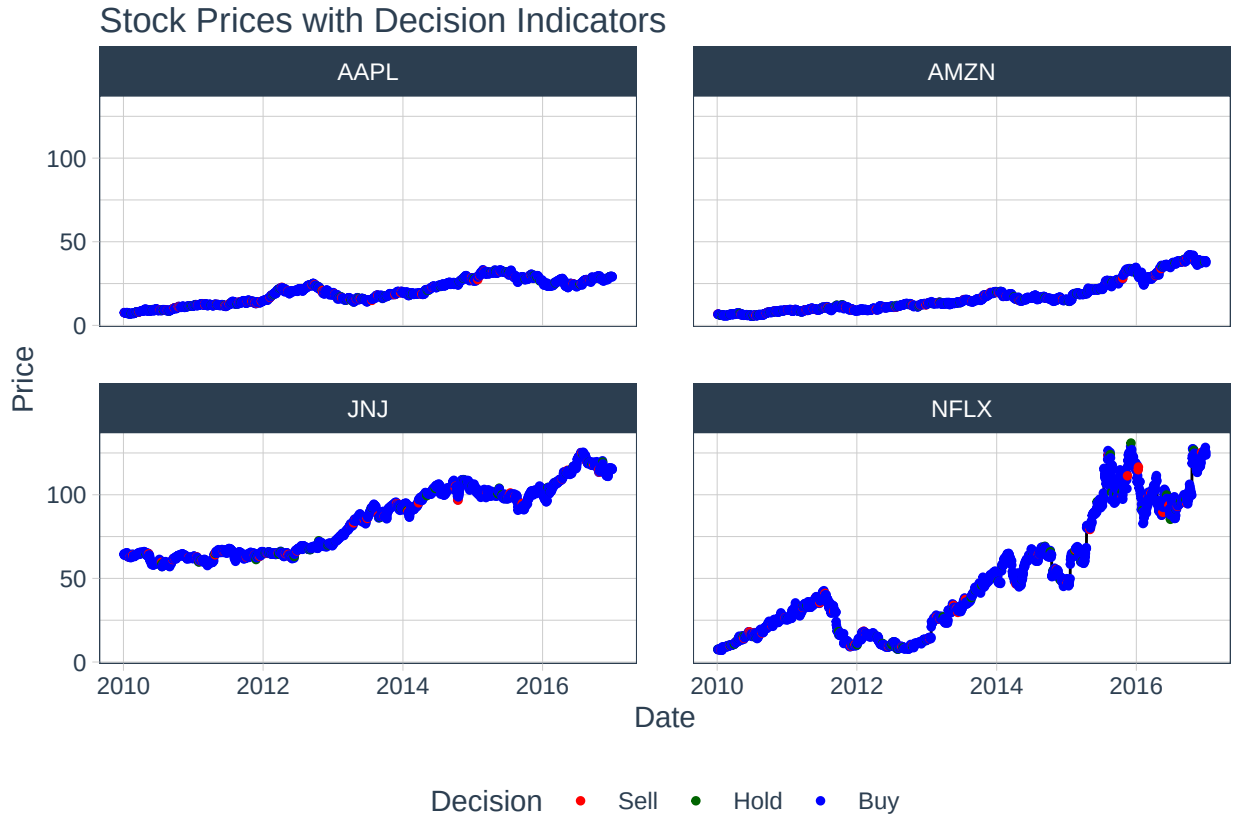
Next, we defined the `tune` function that takes in a dataset, a range of values for the learning rate (alpha), discount factor (gamma), and epsilon-greedy policy (epsilon). The function uses nested loops to iterate over each combination of alpha, gamma, and epsilon values and runs the SARSA algorithm on the dataset using those hyperparameters. The function then calculates the total reward obtained by the `SARSA` algorithm and stores the hyperparameters and the total reward as a list in the "results" variable.

After performing the hyperparameter tuning, the optimal combination of hyperparameters that yielded the highest total reward for the `SARSA` algorithm were; $\alpha = 0.8$, $\gamma = 0.3$, and $\epsilon = 0.1$. These were associated with the highest total reward of 21069.54.

```
# function that extracts the hyperparameters associated with the best result and returns
↪  them as a list

tune <- function(data, alpha_range, gamma_range, epsilon_range) {
  # initialize results list
  results <- list()

  # loop through alpha, gamma, and epsilon values
  for (alpha in alpha_range) {
    for (gamma in gamma_range) {
```

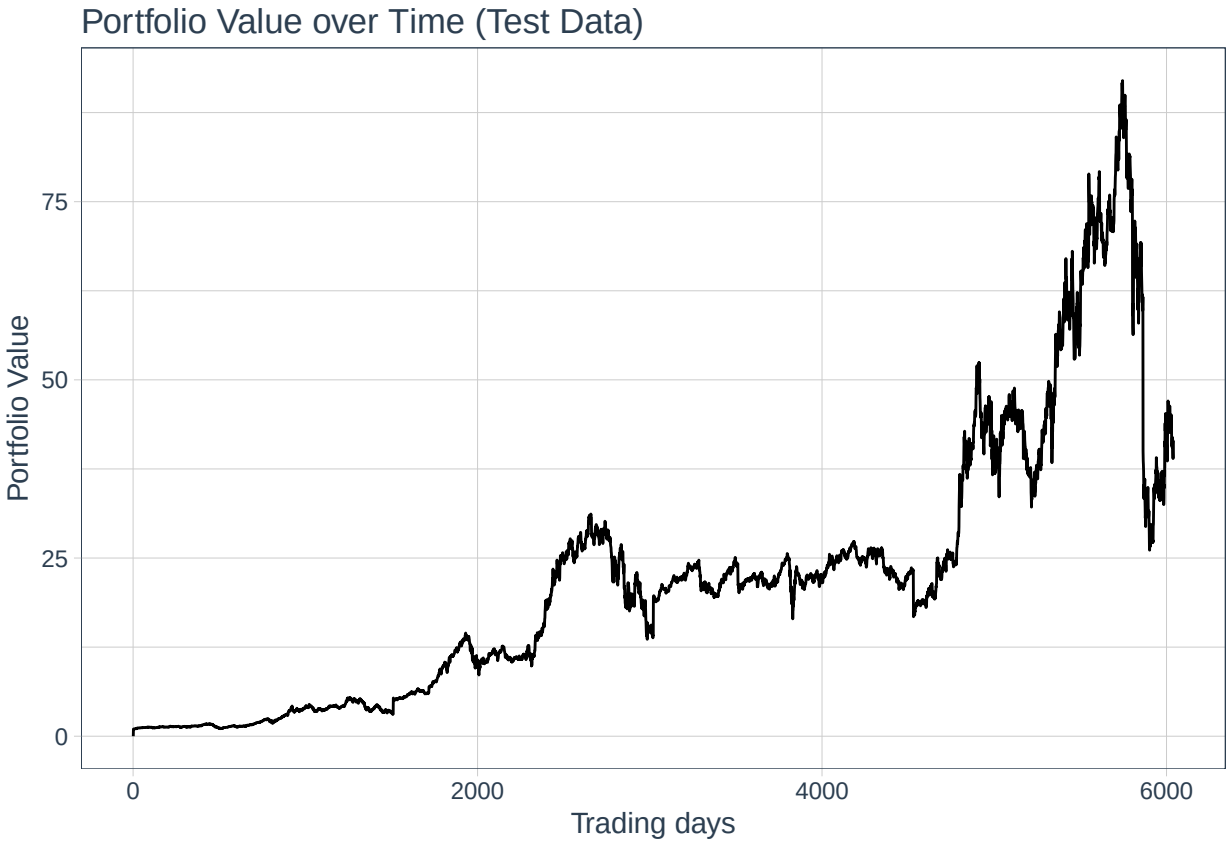
```
      for (epsilon in epsilon_range) {

        # run SARSA algorithm with current hyperparameters
        trades_and_Q <- sarsa(data, alpha = alpha, gamma = gamma, epsilon = epsilon)

        # calculate total reward and store results
        total_reward <- sum(unlist(lapply(trades_and_Q$trades, function(x) x$reward)))
        results[[length(results)+1]] <- list(alpha = alpha, gamma = gamma, epsilon =
↪   epsilon, total_reward = total_reward)
      }
    }
  }

  # return hyperparameters with highest total reward
  best_result_idx <- which.max(sapply(results, function(x) x$total_reward))
  best_result <- results[[best_result_idx]]

  return(list(alpha = best_result$alpha, gamma = best_result$gamma, epsilon =
  ↪  best_result$epsilon, total_reward = best_result$total_reward))
}
```

We then used the combination of the best hyperparameters to train the algorithm on the train test. The figure below shows the algorithm made a higher number of `buy` trading actions compared to the `sell` actions.

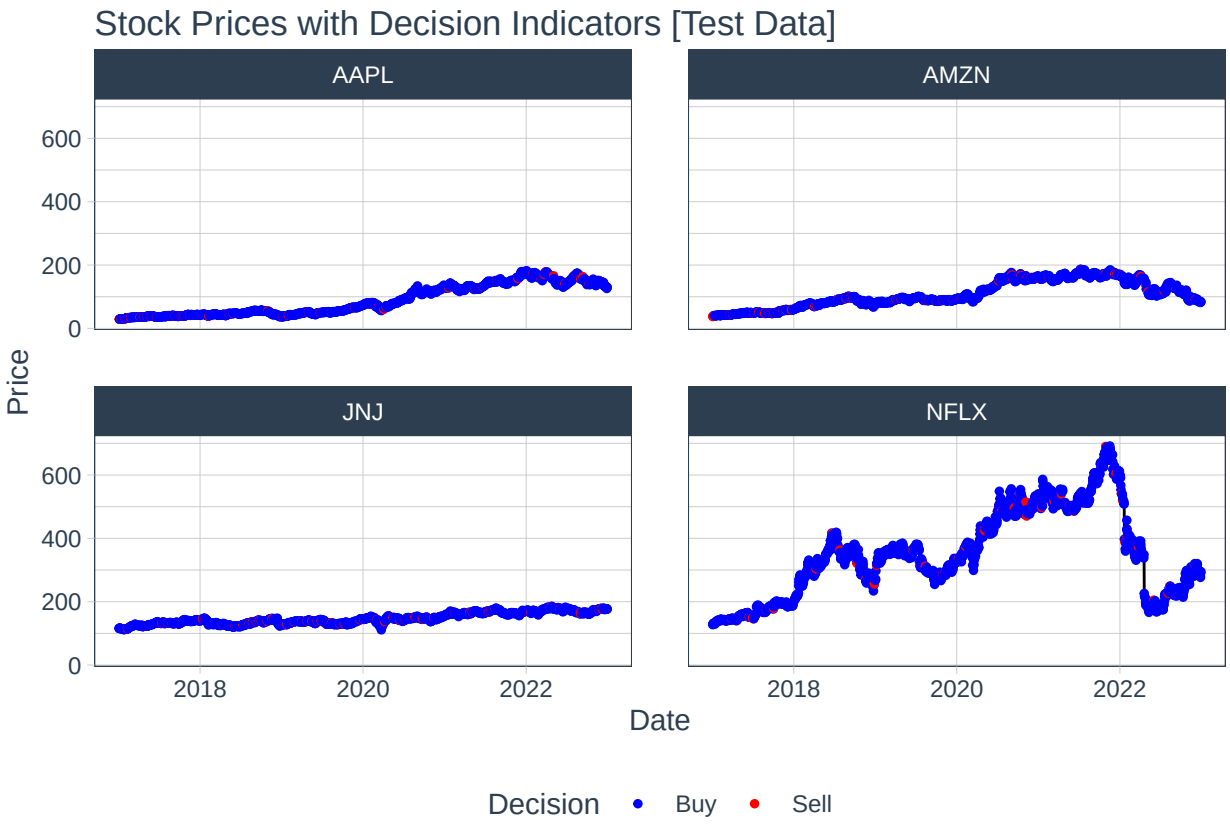


To test the `SARSA` algorithm on the test data, we modified the `sarsa` function to use the learned Q-table to choose actions based on the current state instead of using the `epsilon-greedy policy`. Here, we removed

the Q-table initialization and epsilon-greedy policy from the original `sarsa` function, and added an additional input parameter Q to the `sarsa_test` function to take in the learned Q-table. To test the `SARSA` algorithm on the test data, we can call the `sarsa_test` function with the test data and the learned Q-table as input:



The portfolio value over time represents the value of the portfolio as it changes over the period of the test data. It shows how the agent's decisions based on the SARSA algorithm impacted the portfolio's value. A higher portfolio value means that the agent's decisions led to more profitable trades and a better overall outcome.



From the above results; while the code may be able to generate some predictions on the test data, it is important to keep in mind that SARSA is a model-free algorithm, meaning that it does not explicitly model the underlying dynamics of the stock market. Instead, it learns through trial and error by exploring different actions and observing the resulting rewards.

Limitation: It is difficult to make definitive predictions on the test data using this code alone. It would be more appropriate to evaluate the performance of the agent on the test data by analyzing its trades and portfolio value over time and comparing it to a benchmark such as a buy-and-hold strategy. Additionally, it would be useful to further analyze the agent's behavior and performance by adjusting the hyperparameters (alpha, gamma, and epsilon) and potentially using alternative reinforcement learning algorithms or additional features.

4.3 Implementation of DQN

Next, we used `tensorflow` package in python to implement the DQN on our stock data. However, since training on the full train data set was not feasible due to computational time and memory capacity, we trained the DQN on one stock, NFLX. To reproduce the same using other stocks, all one has to do is just select the stock they want from the drop down widget created in the DQN file. We used the same train/test splits as in the previous sections. We build some of the code used in this section from Analytics Vidhya, but tailored it to serve the needs of our project.

The code in the DQN file defines a class `Agent` which implements a Q-learning algorithm for trading in financial markets.The `__init__ method` initializes the hyperparameters and the neural network architecture. `act` method selects an action based on the current state and `get_state` method returns the current state.

`replay` method trains the Q-network. `buy` method buys or sells shares according to the action selected by the Q-network.

The main logic of the code is in the `buy` method, which takes the initial money and tries to buy and sell shares at different time steps based on the Q-network's decisions. The actions can be to `buy`, `sell`, or `do nothing`. The shares are bought if the Q-network selects action 1 and if the price is less than the available initial money. The shares are sold if the Q-network selects action 2 and if there are shares in the inventory. The profit or loss is calculated based on the price at which the shares are bought and sold. The overall rewards are stored in the total profit variable.

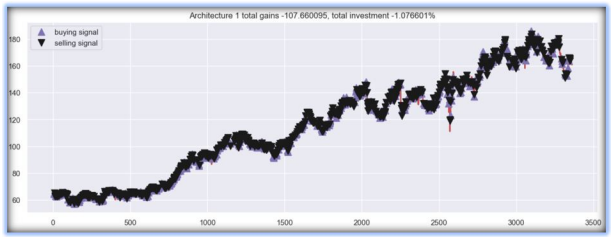
The code also includes a neural network architecture with several hidden layers and uses the `TensorFlow` library for deep learning. We tested the performance of 4 different architectures, with Adam chosen as our chosen gradient descent algorithm as shown in the snip below, with a default batch size of 32. Results of our evaluation are presented below for our four network architectures:

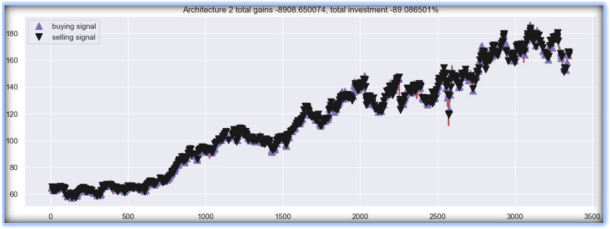
```
        if architecture == 1:
            hidden1 = tf.keras.layers.Dense(256, activation=tf.nn.relu)(self.X)
            hidden2 = tf.keras.layers.Dense(32, activation=tf.nn.relu)(hidden1)
        elif architecture == 2:
            hidden1 = tf.keras.layers.Dense(128, activation=tf.nn.relu)(self.X)
            hidden2 = tf.keras.layers.Dense(32, activation=tf.nn.relu)(hidden1)
        elif architecture == 3:
            hidden1 = tf.keras.layers.Dense(256, activation=tf.nn.relu)(self.X)
            hidden2 = tf.keras.layers.Dense(128, activation=tf.nn.relu)(hidden1)
            hidden3 = tf.keras.layers.Dense(64, activation=tf.nn.relu)(hidden2)
        elif architecture == 4:
            hidden1 = tf.keras.layers.Dense(512, activation=tf.nn.relu)(self.X)
            hidden2 = tf.keras.layers.Dense(256, activation=tf.nn.relu)(hidden1)


        self.logits = tf.keras.layers.Dense(self.action_size)(hidden2)
        self.cost = tf.reduce_mean(tf.square(self.Y - self.logits))
        self.optimizer = tf.train.GradientDescentOptimizer(1e-5).minimize(self.cost)
```

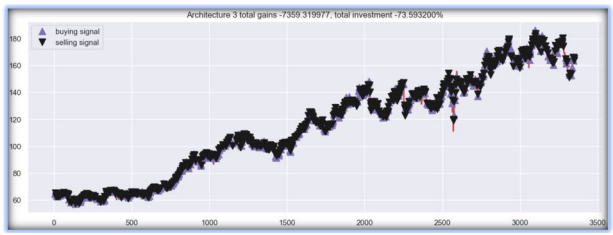
The different network architectures tested

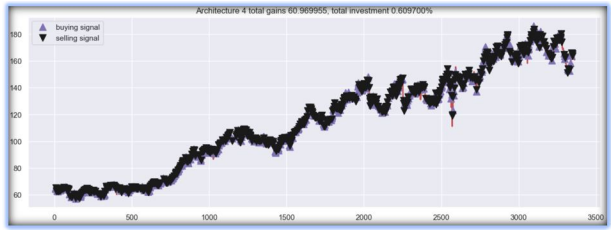
Below are the performance results for the different architectures;

```
Architecture 1:
Total gains: -107.66009521484375
Investment: -1.0766009521484374
Architecture 2:
Total gains: -8908.650074005127
Investment: -89.08650074005126
Architecture 3:
Total gains: -7359.319976806641
Investment: -73.59319976806641
Architecture 4:
Total gains: 60.96995544433594
Investment: 0.6096995544433593
```









We found the best architecture to be the fourth one; the one with two neural networks, with 512 neurons in the first layer and 256 in the second layer. The metrics used to compare our models were the total gains in terms of total investment which were observed over the trading days across each 50 Episode period during training. The fourth architecture had 60.97% gains in investment during training.

Next, we tested the best network architecture on the test data. The buy function will return the buy, sell, profit, and investment figures.

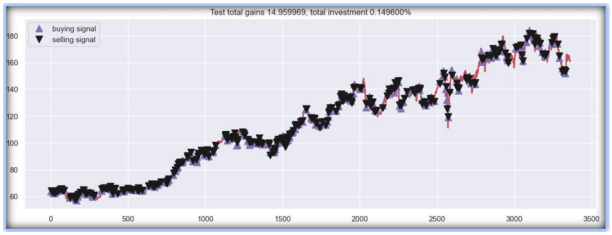
```
# test the agent
states_buy_test, states_sell_test, total_gains_test, invest_test =
↪  test_agent.buy(initial_money = initial_money)
```

Below is a snip of how the agent was trading in the test data for the first 35 days.

```
day 1: buy 1 unit at price 63.930000, total balance 9936.070000
day 4, sell 1 unit at price 64.209999, investment 0.437977 %, total balance 10000.279999,
day 12: buy 1 unit at price 63.970001, total balance 9936.309998
day 13, sell 1 unit at price 63.200001, investment -1.203690 %, total balance 9999.509998,
day 15: buy 1 unit at price 62.790001, total balance 9936.719997
day 19, sell 1 unit at price 63.090000, investment 0.477782 %, total balance 9999.809998,
day 24: buy 1 unit at price 62.369999, total balance 9937.439999
day 25: buy 1 unit at price 62.759998, total balance 9874.680000
day 26, sell 1 unit at price 62.730000, investment 0.577202 %, total balance 9937.410000,
day 28, sell 1 unit at price 62.720001, investment -0.063730 %, total balance 10000.130001,
day 32: buy 1 unit at price 63.810001, total balance 9936.320000
day 33: buy 1 unit at price 63.490002, total balance 9872.829998
day 34, sell 1 unit at price 63.310001, investment -0.783576 %, total balance 9936.139999,
day 35, sell 1 unit at price 63.450001, investment -0.063003 %, total balance 9999.590000,
```

Trading decisions and performance of architecture 4 in the test data

For the Netflix stocks, the figure below shows the trading decisions made by the agent in the test data.



The trading calls of best architecture in the testd data

We observe that in the test data, the agent achieved a 14.96% gains in total investments, which is good performance.

5 Discussion

In this project, we have explored the use of reinforcement learning algorithms for making trading decisions in the stock market. Specifically, we implemented Q-learning and SARSA algorithms in R and a deep Q-network (DQN) algorithm in Python to develop trading strategies based on historical stock data. For the DQN, we trained the agent for 50 iterations, during which it made buy/sell decisions based on the current state and the Q-values of the available actions. The agent stored the experience tuple (state, action, reward, next_state, done) in its memory buffer, and periodically performed a batch update on the Q-values using the experiences in the buffer. At each checkpoint (100), the total rewards earned by the agent, the cost (i.e., the loss during training), and the current amount of money held by the agent were printed.

5.1 Conclusion

It's worth noting that developing an effective trading strategy using Q-learning, SARSA, and DQN RL algorithms is a challenging task that requires careful attention to the choice of input features, reward function, and hyperparameters. Also, keep in mind that trading stocks is a complex task and that the code in this project is only a starting point for a basic implementation and can be improved. It is unlikely to produce profitable trading strategies without further modification, that is, by (1) adding more features (such as technical indicators,) (2) using a more sophisticated state representation (such as sentiment scores, market indices, company fundamentals), (3) running on more episodes and (5) extensive tuning the hyperparameters.

All codes for this project can be found at our GitHub Repository, which also has the link to Project Website

6 References

1. Mishra, A., Gupta, V., Srivastava, S., Pandey, A. K., Kumar, L., & Choudhury, T. (2022). Social media role in pricing value of cryptocurrency. In *Machine intelligence and data science applications: Proceedings of MIDAS 2021* (pp. 819–829). Springer.
2. Sutton, R. S., Barto, A. G., et al. (1998). *Introduction to reinforcement learning. Vol. 135*. MIT press Cambridge.
3. Sutton, R. S., & Barto, A. G. (2018). *Reinforcement learning: An introduction*. MIT press.
4. Ibarz, J., Tan, J., Finn, C., Kalakrishnan, M., Pastor, P., & Levine, S. (2021). How to train your robot with deep reinforcement learning: Lessons we have learned. *The International Journal of Robotics Research*, *40*(4-5), 698–721.
5. Kaiser, L., Babaeizadeh, M., Milos, P., Osinski, B., Campbell, R. H., Czechowski, K., Erhan, D., Finn, C., Kozakowski, P., Levine, S., et al. (2019). Model-based reinforcement learning for atari. *arXiv Preprint arXiv:1903.00374*.
6. Charpentier, A., Elie, R., & Remlinger, C. (2021). Reinforcement learning in economics and finance. *Computational Economics*, 1–38.
7. Li, Y. (2017). Deep reinforcement learning: An overview. *arXiv Preprint arXiv:1701.07274*.
8. Wiering, M. A., & Van Otterlo, M. (2012). Reinforcement learning. *Adaptation, Learning, and Optimization*, *12*(3), 729.
9. D, P. W. P. (2020). *Reinforcement learning*. O'Reilly Media. https://books.google.com/books?id=R9cHEAAAQBAJ
10. Watkins, C. J., & Dayan, P. (1992). Q-learning. *Machine Learning*, *8*, 279–292.
11. Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A. A., Veness, J., Bellemare, M. G., Graves, A., Riedmiller, M., Fidjeland, A. K., Ostrovski, G., et al. (2015). Human-level control through deep reinforcement learning. *Nature*, *518*(7540), 529–533.
12. Mao, H., Alizadeh, M., Menache, I., & Kandula, S. (2016). Resource management with deep reinforcement learning. *Proceedings of the 15th ACM Workshop on Hot Topics in Networks*, 50–56.
13. Powell, W. B., & Frazier, P. (2008). Optimal learning. In *State-of-the-art decision-making tools in the information-intensive age* (pp. 213–246). Informs.

14. Singh, S., Jaakkola, T., Littman, M. L., & Szepesvári, C. (2000). Convergence results for single-step on-policy reinforcement-learning algorithms. *Machine Learning*, *38*, 287–308.
15. Agrawal, S., & Goyal, N. (2013). Thompson sampling for contextual bandits with linear payoffs. *International Conference on Machine Learning*, 127–135.
16. Chakole, J. B., Kolhe, M. S., Mahapurush, G. D., Yadav, A., & Kurhekar, M. P. (2021). A q-learning agent for automated trading in equity stock markets. *Expert Systems with Applications*, *163*, 113761.
17. Oliveira, R. A. de, Ramos, H. S., Dalip, D. H., & Pereira, A. C. M. (2020). A tabular sarsa-based stock market agent. *Proceedings of the First ACM International Conference on AI in Finance*, 1–8.
18. Chen, L., & Gao, Q. (2019). Application of deep reinforcement learning on automated stock trading. *2019 IEEE 10th International Conference on Software Engineering and Service Science (ICSESS)*, 29–33.

7 Session Info

```
## R version 4.2.0 (2022-04-22 ucrt)
## Platform: x86_64-w64-mingw32/x64 (64-bit)
## Running under: Windows 10 x64 (build 19045)
##
## Matrix products: default
##
## locale:
## [1] LC_COLLATE=English_United States.utf8
## [2] LC_CTYPE=English_United States.utf8
## [3] LC_MONETARY=English_United States.utf8
## [4] LC_NUMERIC=C
## [5] LC_TIME=English_United States.utf8
##
## attached base packages:
## [1] stats     graphics  grDevices utils     datasets  methods   base
##
## other attached packages:
##  [1] forcats_1.0.0               stringr_1.5.0
##  [3] purrr_1.0.1                 readr_2.1.4
##  [5] tidyr_1.3.0                 tibble_3.2.0
##  [7] tidyverse_2.0.0             kableExtra_1.3.4
##  [9] tidyquant_1.0.7             PerformanceAnalytics_2.0.4
## [11] lubridate_1.9.2             cowplot_1.1.1
## [13] knitr_1.42                  reshape2_1.4.4
## [15] gridExtra_2.3               fBasics_4022.94
## [17] htmltools_0.5.4             DT_0.27
## [19] dplyr_1.1.0                 ggplot2_3.4.1
## [21] ReinforcementLearning_1.0.5 quantmod_0.4.20
## [23] TTR_0.24.3                  xts_0.13.0
## [25] zoo_1.8-11
##
## loaded via a namespace (and not attached):
##  [1] httr_1.4.5          jsonlite_1.8.4      viridisLite_0.4.1
##  [4] yaml_2.3.7          pillar_1.8.1        lattice_0.20-45
##  [7] glue_1.6.2          quadprog_1.5-8      digest_0.6.31
## [10] rvest_1.0.3         colorspace_2.1-0    plyr_1.8.8
## [13] timeDate_4022.108   pkgconfig_2.0.3     bookdown_0.33
## [16] scales_1.2.1        webshot_0.5.4       svglite_2.1.1
## [19] jpeg_0.1-10         tzdb_0.3.0          timechange_0.2.0
## [22] generics_0.1.3      spatial_7.3-15      ellipsis_0.3.2
```

```
## [25] withr_2.5.0         cli_3.6.0           magrittr_2.0.3
## [28] evaluate_0.20       fansi_1.0.4         xml2_1.3.3
## [31] tools_4.2.0         hash_2.2.6.2        hms_1.1.2
## [34] lifecycle_1.0.3     munsell_0.5.0       compiler_4.2.0
## [37] timeSeries_4021.105 systemfonts_1.0.4   rlang_1.1.0
## [40] grid_4.2.0          rstudioapi_0.14     htmlwidgets_1.6.1
## [43] rmarkdown_2.20      codetools_0.2-18    gtable_0.3.1
## [46] curl_5.0.0          R6_2.5.1            fastmap_1.1.1
## [49] utf8_1.2.3          Quandl_2.11.0       stringi_1.7.12
## [52] Rcpp_1.0.10         vctrs_0.6.0         tidyselect_1.2.0
## [55] xfun_0.37
```