FinSearch

Mid-Term Report

Topic: **Deep RL to optimize stock trading strategy and maximizing returns**

Team participants:

SHRESTH VERMA
RAHUL GUPTA
SHREYAS SINHA
PRANAV GUPTA

Contents

Roadmap 1-

Introduction

Traditional Algorithms and RL Based Algorithms

Examples of Algorithms

Advantages and Disadvantages

Roadmap 2-

Introduction

Markov decision Process (MDP)

Terminologies

Steps followed

Important necessary Libraries

Dataset

Necessary Data Processing

Environment with explanation

Training and Testing the Algorithm

Results

Contents

Roadmap 3-

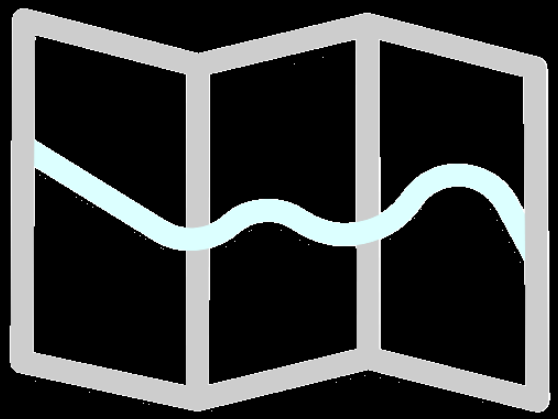
Environment with explanation

Training and Testing the Algorithm

Results

Roadmap 4-

References



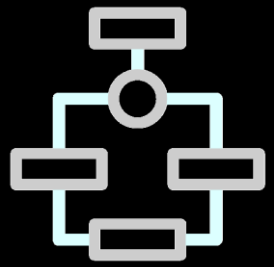
Roadmap 1

Introduction:

This note aims to provide a comparative analysis of traditional algorithms and reinforcement learning (RL)-based algorithms. Traditional algorithms have been widely used in various fields for decades, while RL-based algorithms leverage machine learning techniques to optimize decision-making processes. We will explore the relative advantages and disadvantages of these two approaches, considering factors such as interpretability, generalizability, training data requirements, and computational complexity. Understanding the trade-offs between traditional algorithms and RL-based algorithms is crucial for selecting the most appropriate approach for a given problem domain.

Algorithms

Traditional Algorithms:



Traditional algorithms, also known as classical algorithms or deterministic algorithms, refer to a set of well-defined procedures or step-by-step instructions designed to solve specific problems or perform specific tasks. These algorithms are based on established mathematical and computational principles and have been used for decades in various fields, including computer science, mathematics, engineering, and operations research.

Traditional algorithms are typically rule-based and follow a deterministic approach, meaning that for a given input, they produce the same output consistently. They rely on predefined rules, logical conditions, and mathematical operations to process input data and generate desired results. These algorithms often involve explicit programming and use structured, procedural, or imperative programming languages.

The design and implementation of traditional algorithms rely on formal mathematical models and established algorithms that have been extensively studied and analysed. They leverage well-established theories and principles, such as sorting algorithms (e.g., bubble sort, quicksort), search algorithms (e.g., binary search, depth-first search), and optimization algorithms (e.g., linear programming, greedy algorithms).

Traditional algorithms are known for their simplicity, transparency, and interpretability. Examples of traditional algorithms include algorithms for sorting, searching, graph traversal, numerical computations, encryption, and many other specific problem-solving techniques. They have been widely used in areas such as data analysis, image processing, pattern recognition, optimization problems, and more

RL Based Algorithms

Reinforcement learning (RL) is a subfield of machine learning that focuses on the development of algorithms capable of learning optimal decision-making policies through interaction with an environment. RL-based algorithms aim to solve sequential decision-making problems by maximizing a cumulative reward signal.

RL algorithms learn from experience by iteratively interacting with an environment. They operate in an agent-environment framework, where an agent takes actions based on its observations and receives feedback in the form of rewards or penalties. The agent's objective is to learn an optimal policy that maximizes the long-term expected reward.

In RL-based algorithms, the environment is typically modelled as a Markov Decision Process (MDP), which is a mathematical framework that captures the dynamics of sequential decision-making problems. The MDP consists of a set of states, actions, transition probabilities, rewards, and a discount factor. The RL algorithm learns to estimate the value or utility of each state or state-action pair and uses this information to make decisions.

RL algorithms employ exploration and exploitation strategies to balance the trade-off between gathering more information about the environment and exploiting already learned knowledge. Common RL algorithms include Q-learning, SARSA, Deep Q-Networks (DQN), Policy Gradient methods, and Proximal Policy Optimization (PPO), among others.

One of the primary advantages of RL-based algorithms is their ability to handle complex problems with high dimensional state and action spaces. RL algorithms can optimize decisions by considering long-term consequences, allowing them to navigate through environments with numerous variables and interactions. RL-based algorithms also exhibit adaptability and generalization. They can learn from experience and adapt to changing environments, making them suitable for dynamic and uncertain scenarios. RL algorithms can handle situations where the optimal policy may change over time or when the environment dynamics are not fully known.

Examples of Traditional algorithms used in stock analysis

Linear Regression: This algorithm assumes a linear relationship between the independent variables (such as historical stock prices, trading volumes, or economic indicators) and the dependent variable (the stock price). It fits a line to the data and uses it to make predictions.

Bayesian Networks: Bayesian Networks model the probabilistic relationships between variables. They can incorporate both historical stock price data and other relevant factors (e.g., news sentiment or macroeconomic indicators) to make predictions.

Autoregressive Integrated Moving Average (ARIMA): ARIMA models capture the temporal dependencies and trends in the stock price data. It considers the past values of the stock prices and their differences to forecast future values.

Hidden Markov Models (HMM): HMMs are used to model stock prices as a sequence of hidden states, where each state represents a different market regime (e.g., bull or bear market). HMMs can capture the underlying dynamics of the market and make prediction based on the current state.

GARCH (Generalized Autoregressive Conditional Heteroskedasticity): GARCH models are used to capture volatility clustering and time-varying variances in stock prices. They take into account the conditional volatility of previous periods to predict future volatility.

Kalman Filters: Kalman Filters are recursive estimation algorithms that can be used to predict stock prices. They update their predictions as new data becomes available, incorporating both the observed data and the model's predictions

Support Vector Machines (SVM): SVM is a supervised learning algorithm that can be used for stock price prediction. It finds a hyperplane that separates the data into different classes (e.g., price increase or decrease). SVM can handle non-linear relationships through the use of kernel functions.

Random Forest: Random Forest is an ensemble learning algorithm that combines multiple decision trees to make predictions. It can handle non-linear relationships and capture complex interactions between variables.

Examples of RL Based Algorithms used for Stock Analysis

Q-Learning is a popular RL algorithm used for sequential decision-making problems. In the context of stock price prediction, it can be used to determine the optimal trading actions (e.g., buy, sell, or hold) based on the observed market conditions. The algorithm learns a Q-value function that estimates the expected cumulative reward for each state-action pair and uses it to make decisions.

Policy Gradient Methods: Policy Gradient methods directly learn a policy that maps states to actions without explicitly estimating the value function. These methods optimize the policy parameters to maximize the cumulative reward. They can be applied to stock price prediction by learning a policy that determines the trading actions based on historical data and possibly other relevant features.

Deep Q-Networks (DQN): DQN combines Q-Learning with deep neural networks. It uses a deep neural network, known as the Q-network, to approximate the Q-value function. This allows for more complex representations and can capture non-linear relationships between market data and trading actions. DQN has been applied to stock price prediction by considering historical stock prices and technical indicators as input features

Actor-Critic Methods: Actor-Critic methods combine aspects of both value-based and policy-based RL.

Advantages of Traditional Algorithms

- Simplicity and Interpretability: Traditional algorithms are often simpler in terms of their design and implementation compared to RL-based algorithms. They follow a deterministic approach, which means that given the same input, they produce the same output consistently. This simplicity contributes to their interpretability, making them easier to understand and reason about. Example: Sorting Algorithms Sorting algorithms such as Bubble Sort or Insertion Sort are classic examples of traditional algorithms. They follow a step-by-step procedure to arrange elements in a specific order, such as ascending or descending. These algorithms are straightforward to understand and interpret, as they involve comparing and swapping elements based on predefined rules.

- Well-Established Theory and Implementation: Traditional algorithms have been studied and researched for many years, resulting in a vast body of knowledge and well-established theories. They have mature implementation frameworks and libraries, making them readily available for use. Extensive documentation, textbooks, and online resources are also available, making it easier to learn and apply traditional algorithms. Example: Dijkstra's Algorithm Dijkstra's algorithm is a widely used traditional algorithm for finding the shortest path in a graph. It is based on graph theory and has been extensively studied and implemented in various programming languages. Due to its well-established theory and implementation, it is widely used in navigation systems, network routing, and other applications that require pathfinding.

- Less Computational Complexity: Traditional algorithms often have lower computational complexity, meaning they require fewer computational resources and have faster execution times compared to some complex machine learning algorithms. They are suitable for resource-limited environments or situations where real-time processing is crucial. Example: Binary Search Binary search is a classic example of a traditional algorithm with low computational complexity. It efficiently searches for a target element in a sorted array by repeatedly dividing the search space in half. Binary search has a time complexity of O(log n), making it faster than linear search algorithms when dealing with large datasets.

Disadvantages of Traditional Algorithms

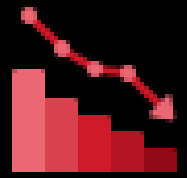
- Lack of Adaptability: Traditional algorithms are often designed with predefined rules and assumptions based on static problem settings. They lack the ability to adapt and learn from experience or adjust their behaviour in response to changes in the environment. This makes them less suitable for dynamic or uncertain scenarios where the optimal solution may vary over time. Example: Fixed-Rule Decision Systems Fixed-rule decision systems, such as rule-based expert systems, are traditional algorithms that rely on predetermined rules to make decisions. These systems are limited in their adaptability since the rules need to be manually programmed and do not learn or update based on new information. As a result, they may not handle situations where the rules need to be modified or updated based on changing conditions.

- Limited Optimization: Traditional algorithms may not be capable of achieving optimal solutions in complex problems with a large number of variables and interactions. They often operate based on local optimization or heuristic approaches and do not consider long-term consequences or learn from experience. This limits their ability to handle problems that require sophisticated optimization strategies. Example: Greedy Algorithms Greedy algorithms are traditional algorithms that make locally optimal choices at each step in the hope of finding a global optimum. However, they do not consider the overall consequences of their choices. While greedy algorithms can be efficient and provide good approximate solutions in some cases, they may not guarantee the optimal solution for all problem instances.

- Sensitivity to Input Quality: Traditional algorithms heavily rely on the quality and reliability of the input data. They assume that the input data is accurate, consistent, and free from noise or biases. However, in real-world scenarios, data can be noisy, incomplete, or contain outliers, which can significantly affect the performance and accuracy of traditional algorithms. Example: Linear Regression with Outliers Linear regression is a traditional algorithm used for predicting a continuous output variable based on input features. However, if the dataset contains outliers, which are extreme values significantly deviating from the expected pattern, linear regression can be sensitive to these outliers and may produce inaccurate or unreliable predictions

Advantages of RL Based Algorithms

- **Adaptability and Generalization:** RL-based algorithms excel at adapting to changing environments and learning from experience. They can dynamically adjust their decision-making policies based on the feedback received from the environment. This adaptability allows RL algorithms to handle complex and dynamic problem domains where the optimal solution may change over time. Example: Autonomous Driving RL-based algorithms can be used in autonomous driving systems to learn optimal driving policies. By interacting with the environment (e.g., simulators or real-world driving scenarios), an RL agent can learn to navigate roads, make appropriate decisions at intersections, and respond to dynamic traffic conditions. The agent continually adapts its driving behaviour based on the rewards and penalties received, leading to improved performance over time.
- **Complex Decision-Making:** RL-based algorithms are well-suited for problems with high-dimensional state and action spaces, where traditional algorithms may struggle due to the exponential growth of possibilities. RL algorithms can optimize decisions by considering long-term consequences and interactions between states and actions, allowing them to handle complex problem domains effectively. Example: Game Playing RL-based algorithms have achieved remarkable success in playing complex games. For instance, AlphaGo, an RL-based algorithm developed by DeepMind, achieved groundbreaking performance by defeating world champion Go players. Through extensive self-play and reinforcement learning, AlphaGo learned complex strategies, positional evaluations, and long-term planning, surpassing traditional approaches that relied on handcrafted heuristics.
- **Data Efficiency:** RL-based algorithms can learn from sparse reward signals, meaning they can extract valuable information from limited feedback or interactions with the environment. They are capable of learning with minimal supervision, reducing the reliance on large labelled datasets. Example: Robotics Manipulation RL-based algorithms have been employed in robotics manipulation tasks, where a robotic arm learns to manipulate objects in its environment. By iteratively interacting with the environment, the RL agent can learn complex grasping and manipulation policies using sparse rewards. The agent gradually refines its actions based on the obtained rewards, eventually achieving fine-grained control and dexterity.



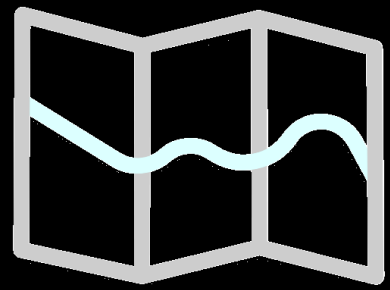
Disadvantages of RL Based Algorithms

- **Computational Complexity:** RL-based algorithms often require significant computational resources and time for training. The learning process involves iteratively exploring and interacting with the environment, which can be computationally intensive. This limitation makes RL algorithms less suitable for real-time or resource constrained applications. Example: Deep Reinforcement Learning Deep Reinforcement Learning algorithms, such as Deep Q-Networks (DQN) or Proximal Policy Optimization (PPO), utilize deep neural networks to approximate value functions or policies. Training these networks often requires large amounts of data and computationally expensive operations, making them resource-intensive and time-consuming.

- **Lack of Interpretability:** RL-based algorithms are often considered as "black-box" approaches since the decision-making process can be complex and difficult to interpret. The learned policies or value functions may not provide clear explanations or insights into the underlying reasoning. This lack of interpretability can be a limitation in domains where transparency and explain ability are essential. Example: Deep Reinforcement Learning for Image Classification Applying RL algorithms directly to image classification tasks can be challenging in terms of interpretability. When an RL agent learns to classify images using deep neural networks, understanding the specific features or patterns that contribute to its decision-making can be challenging. The network's internal representations might not be easily interpretable, making it difficult to trace decisions back to human-understandable rules.

- **Training Data Requirements:** RL-based algorithms typically require substantial amounts of training data to converge to optimal policies. The learning process relies on interactions with the environment to update the value functions or policies, and this can be data intensive. Obtaining large amounts of labelled or interaction data can be impractical or costly in certain domains. Example: Simulated Environments for RL Training RL algorithms are often trained in simulated environments that mimic real-world scenarios. However, creating accurate and representative simulations can be challenging and might not fully capture the complexity and diversity of real-world situations. The quality and availability of training data play a significant role in the performance and generalization capabilities of RL algorithms. These examples illustrate the disadvantages of RL-based algorithms in terms of computational complexity, interpretability, and training data requirements. While RL algorithms offer remarkable adaptability, complex decision-making, and data efficiency, these limitations should be considered when applying them to real-world problems. Understanding the computational and interpretability trade-offs is crucial, and alternative approaches or hybrid solutions may be more suitable depending on the specific requirements of the problem domain

Roadmap 2

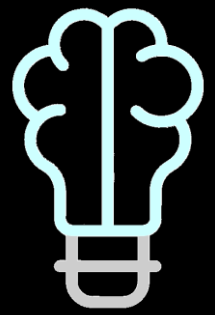


Aim

The main aim of the project is to build a RL based model to optimise and automate stock trading and bring greater returns from the market.

Abstract

Reinforcement learning is all about taking the right steps to maximise your reward in a given situation. It is used by a variety of software and computers to determine the best feasible action or path in a given situation. The Reinforcement Machine Learning model is employed in this work to forecast the closure price using past data. We develop a predictor for multiple firms using these trained models that forecasts the every day close stock prices. By providing inputs such as open, high, and low prices, stock volume, and the latest events about each firm, this predictor may be used to determine the price at which the stock value will close for a certain day. The goal of this project is to learn and get hands-on experience in Data Analytics and Machine Learning.



Roadmap 2: Introduction

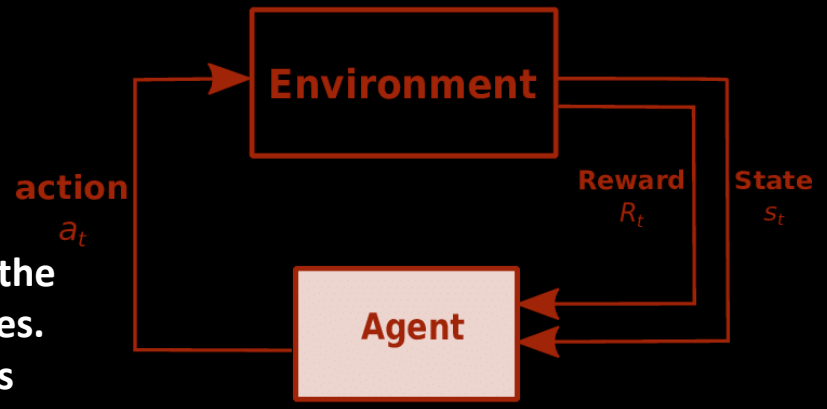
We've always been captivated by the stock market's seeming unpredictability. There are hundreds of stocks to select from, and day traders can trade almost any of them. So, for a day trader, deciding what to trade is the first and most important step. The following stage is to come up with some ways to benefit from the trading opportunity (one stock, several stocks, exchange traded funds, ETFs, etc.). Intraday traders use a number of ways to profit from price movements in a particular asset. Day traders should look for equities with plenty of liquidity, moderate to high volatility, and a large number of followers. Isolating the present market trend from any surrounding noise and then capitalising on that trend is the key to finding the appropriate stocks for intraday trading. This has traditionally been done in conjunction with the trade plan and current events. Various research techniques have been explored to automate this laborious procedure since the emergence of Data Science and Machine Learning. This automated trading method will assist in providing recommendations at the appropriate moment and with more accurate estimates. Mutual funds and hedge funds would benefit greatly from an automated trading approach that maximises profits. The type of profitable returns that may be expected will be accompanied by some risk. It's difficult to come up with a lucrative automated trading technique. Every human being aspires to make as much money as possible in the stock market. It's critical to devise a well-balanced, low-risk plan that will benefit the majority of individuals. One such technique proposes the use of reinforcement learning agents to generate automated trading strategies based on previous data. This project was made because we were intrigued and we wanted to gain hands-on experience with the Machine Learning Project.

Markov decision process (MDP)

It is a mathematical framework used for modelling decision-making problems where the outcomes are partly random and partly controllable. A model of anticipating outcomes is the Markov decision process. The model, like a Markov chain, tries to predict a result based solely on the present state's information. The Markov decision process, on the other hand, takes into account the features of actions and motivations.

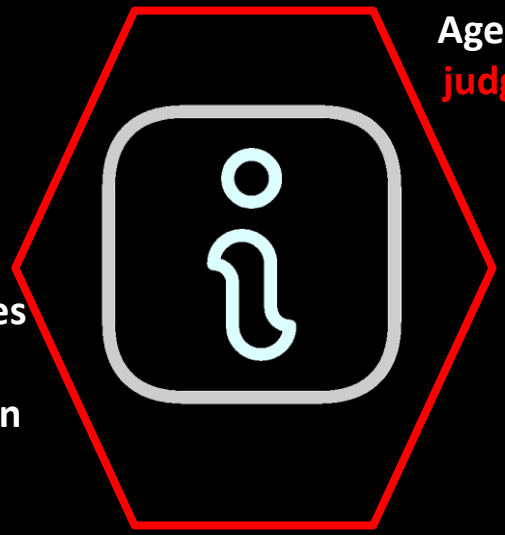
The decision-maker may choose an action accessible in the current state at each phase of the process, causing the model to advance to the next step and rewarding the decision-maker. At each step during the process, the decision-maker may choose to take any action available in the current state, resulting in the model moving to the next step and offering the decisionmaker a reward.

Termindologies



Environment - The agent's environment is the setting in which the agent interacts. For example, the house where the Robot moves. The agent has no influence over the surroundings; all it has is control over its own actions. For instance, Although the Robot cannot control where a table is kept in the house, it can walk around it to avoid colliding with it.

Action - The decision made by the agent in the current time step. For instance: it can move its right or left leg, or raise its arm, or lift an object, turn right or left, etc. We know ahead of time what actions or decisions the agent can take.



Agent - An RL agent is a machine that we're teaching to make good judgments. For instance, a robot is being taught how to navigate a house without colliding.

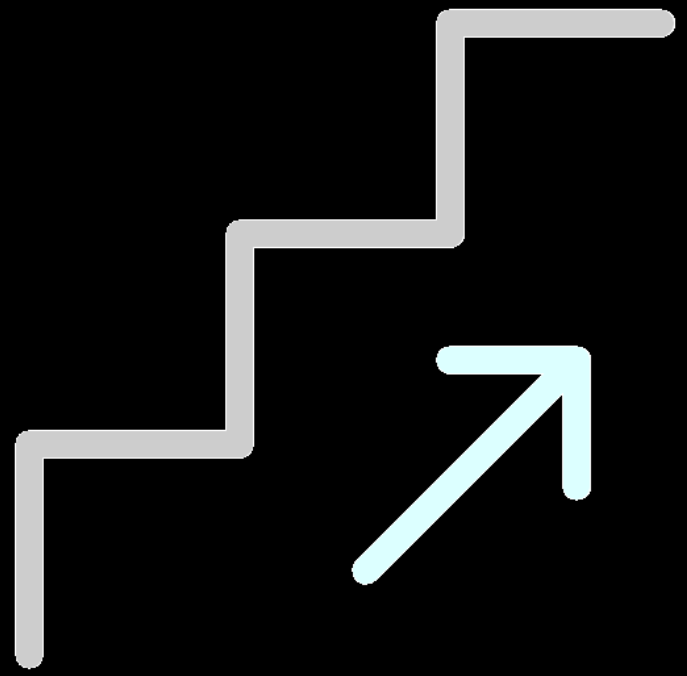
Policy - A policy is the thinking process that goes into deciding on a course of action. It is a probability distribution applied to the collection of activities in practice. Actions that are very rewarding will have a high likelihood, and vice versa.

State - The state defines the current situation of the agent, for example, It may be the Robot's specific position in the house, the alignment of its two legs, or its current posture, depending on how you approach the issue.

Note: Even if an action has a low probability, that does not mean it will not be chosen. It's only that it has a lower chance of being chosen.

Steps followed to solve Roadmap 2

- Import necessary Libraries.
- Load Dataset.
- Perform Exploratory Data Analysis.
- Create an Environment.
  a)define the state of the environment
  b)define the action space
  c) define the reward function to train the agent
- Prepare Data.
- Train Data.
- Test Data.
- Evaluate Model.



Import necessary Libraries

jupyter Untitled2 Last Checkpoint: 5 minutes ago (autosaved)

Logout

File Edit View Insert Cell Kernel Widgets Help

Trusted | Python 3 (ipykernel)

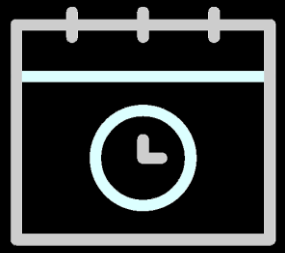
Run Code

In [4]:
```
!pip install chainer
!pip install plotly
```

```
Requirement already satisfied: chainer in c:\users\shres\anaconda3\lib\site-packages (7.8.1)
Requirement already satisfied: filelock in c:\users\shres\anaconda3\lib\site-packages (from chainer) (3.9.0)
Requirement already satisfied: typing-extensions in c:\users\shres\anaconda3\lib\site-packages (from chainer) (4.4.0)
Requirement already satisfied: setuptools in c:\users\shres\anaconda3\lib\site-packages (from chainer) (65.6.3)
Requirement already satisfied: six>=1.9.0 in c:\users\shres\anaconda3\lib\site-packages (from chainer) (1.16.0)
Requirement already satisfied: protobuf>=3.0.0 in c:\users\shres\anaconda3\lib\site-packages (from chainer) (4.23.4)
Requirement already satisfied: numpy>=1.9.0 in c:\users\shres\anaconda3\lib\site-packages (from chainer) (1.23.5)
Requirement already satisfied: plotly in c:\users\shres\anaconda3\lib\site-packages (5.9.0)
Requirement already satisfied: tenacity>=6.2.0 in c:\users\shres\anaconda3\lib\site-packages (from plotly) (8.0.1)
```

In [3]:
```
import time
import copy
import numpy as np
import pandas as pd
import chainer
import chainer.functions as F
import chainer.links as L
from plotly import tools, subplots
from plotly.graph_objs import *
from plotly.offline import init_notebook_mode, iplot, iplot_mpl
init_notebook_mode()
```

All the following libraries are necessary for defining the time stamps, interpret the data, plot them, use them to train the RI agent and create the reward function.



DATASET

The data used is as follows:
(from 2013 to 2018)

```
In [6]: data
```

Out[6]:

	date	open	high	low	close	volume	Name
0	08-02-2013	15.07	15.12	14.63	14.75	8407500	AAL
1	11-02-2013	14.89	15.01	14.26	14.46	8882000	AAL
2	12-02-2013	14.45	14.51	14.10	14.27	8126000	AAL
3	13-02-2013	14.30	14.94	14.25	14.66	10259500	AAL
4	14-02-2013	14.94	14.96	13.16	13.99	31879900	AAL
...	...	...	...	...	...	...	...
619035	01-02-2018	76.84	78.27	76.69	77.82	2982259	ZTS
619036	02-02-2018	77.53	78.12	76.73	76.78	2595187	ZTS
619037	05-02-2018	76.64	76.92	73.18	73.83	2962031	ZTS
619038	06-02-2018	72.74	74.56	72.13	73.27	4924323	ZTS
619039	07-02-2018	72.70	75.00	72.69	73.86	4534912	ZTS

619040 rows × 7 columns

```
In [7]: data.head()
```

Out[7]:

	date	open	high	low	close	volume	Name
0	08-02-2013	15.07	15.12	14.63	14.75	8407500	AAL
1	11-02-2013	14.89	15.01	14.26	14.46	8882000	AAL
2	12-02-2013	14.45	14.51	14.10	14.27	8126000	AAL
3	13-02-2013	14.30	14.94	14.25	14.66	10259500	AAL
4	14-02-2013	14.94	14.96	13.16	13.99	31879900	AAL

```
In [8]: data.tail()
```

Out[8]:

	date	open	high	low	close	volume	Name
619035	01-02-2018	76.84	78.27	76.69	77.82	2982259	ZTS
619036	02-02-2018	77.53	78.12	76.73	76.78	2595187	ZTS
619037	05-02-2018	76.64	76.92	73.18	73.83	2962031	ZTS
619038	06-02-2018	72.74	74.56	72.13	73.27	4924323	ZTS
619039	07-02-2018	72.70	75.00	72.69	73.86	4534912	ZTS

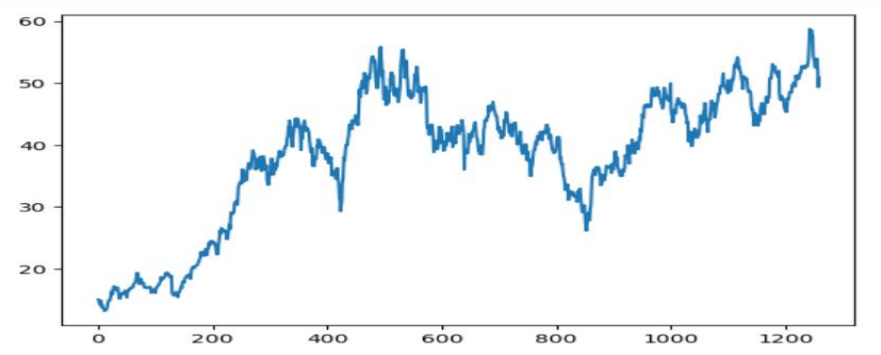
Necessary Data Processing

1. Setting the date column as standard date time format:

	date	open	high	low	close	volume	Name
619035	01-02-2018	76.84	78.27	76.69	77.82	2982259	ZTS
619036	02-02-2018	77.53	78.12	76.73	76.78	2595187	ZTS
619037	05-02-2018	76.64	76.92	73.18	73.83	2962031	ZTS
619038	06-02-2018	72.74	74.56	72.13	73.27	4924323	ZTS
619039	07-02-2018	72.70	75.00	72.69	73.86	4534912	ZTS

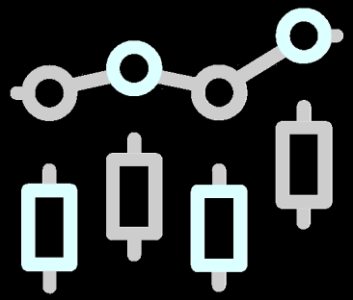
2. Plotting a line chart of AAL stocks against closing price:

```
In [9]: import matplotlib.pyplot as plt
        plt.plot(data[data["Name"]=="AAL"]["open"])
```



4. Resolving the missing values of dataset:

```
In [15]: df.isnull().sum()

Out[15]: open      0
         high      0
         low       0
         close     0
         volume    0
         Name      0
         dtype: int64
```

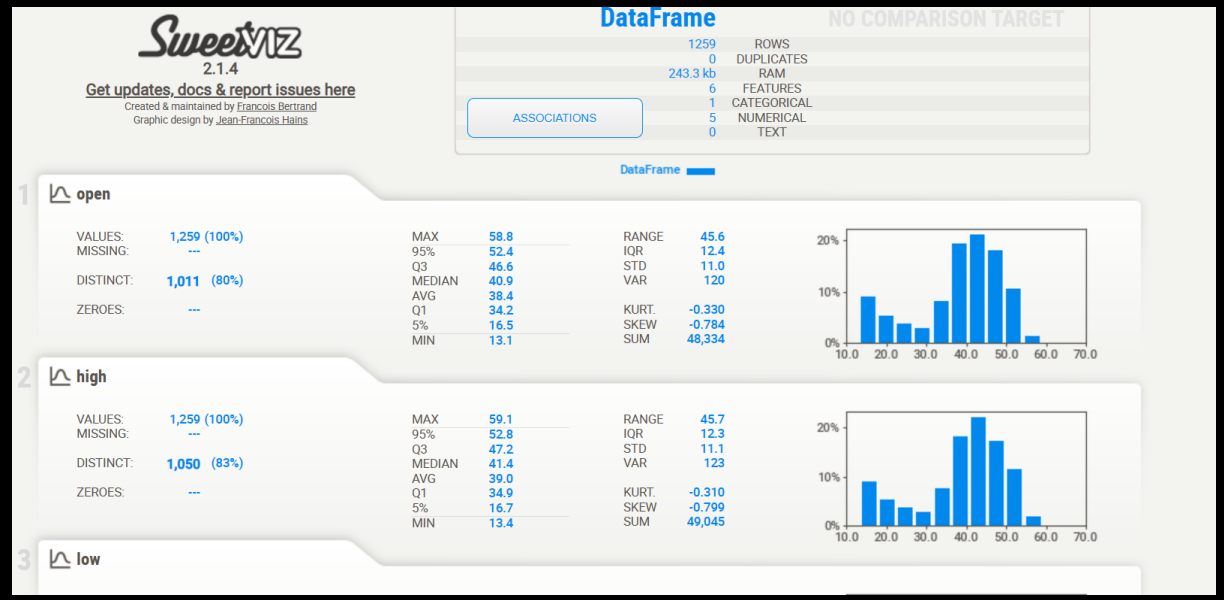


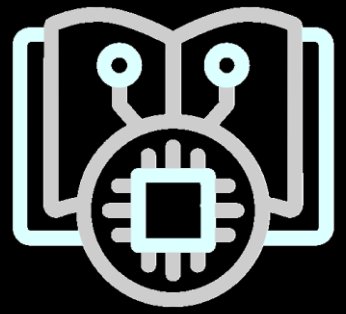
3. To automate our tasks in exploratory data analysis, we may utilise python packages like dtale, pandas profiling, sweetviz, and autoviz.

```
import sweetviz as sv
advert_report=sv.analyze(df)
advert_report.show_html('EDA.html')
```

```
C:\Users\shres\anaconda3\lib\site-packages\sweetviz\dataframe_report.py:74: FutureWarning:

iteritems is deprecated and will be removed in a future version. Use .items instead.

C:\Users\shres\anaconda3\lib\site-packages\sweetviz\dataframe_report.py:109: FutureWarning:

iteritems is deprecated and will be removed in a future version. Use .items instead.
```

Done! Use 'show' commands to display/save. [100%] 00:01 -> (00:00 left)

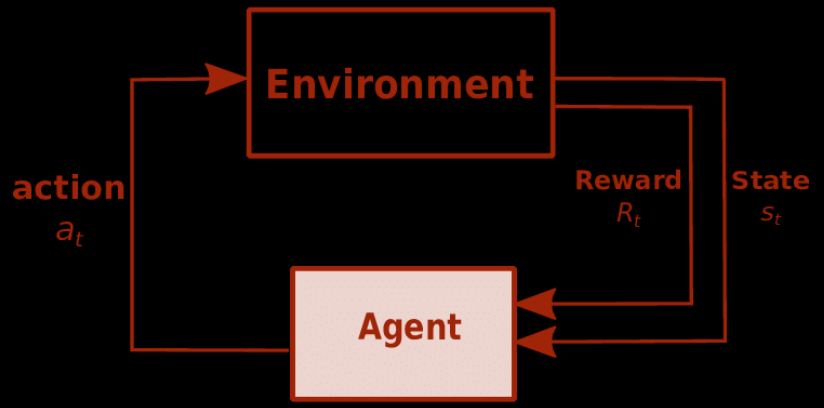




Roadmap 3

Environment with Explanation

CREATE AN ENVIRONMENT



In the reinforcement learning problem, the environment is a critical component. It's critical to have a solid grasp of the underlying world with which the RL agent will interact. This aids us in developing the best design and learning strategy for the agent.

The environment creation process involves the following path:

- **Defining a state**
- **Defining an action space**
- **Defining a reward function**
- **Explanation**

Defining a state

When it comes to capturing market conditions, there are several common state representations that are used in financial analysis and modelling. Here are a few examples:

1. Price-based indicators: These state representations utilize price data to capture market conditions.

2. Moving averages: Simple Moving Average (SMA), Exponential Moving Average (EMA)

- Bollinger Bands
- Relative Strength Index (RSI)
- Stochastic Oscillator

3. MACD (Moving Average Convergence Divergence)

Volume-based indicators: These state representations focus on trading volume to capture market conditions.

4. On-Balance Volume (OBV)

- Volume Weighted Average Price (VWAP)
- Accumulation/Distribution Line (ADL)
- Volatility-based indicators: These state representations aim to measure market volatility.

5. Average True Range (ATR)
Bollinger Bands (which can also be considered a volatility indicator)
Market breadth indicators: These state representations provide an overview of the market's overall health and participation.

6. Advance-Decline Line (AD Line)
McClellan Oscillator
Sentiment indicators: These state representations attempt to capture market sentiment and investor psychology.
Example: Put/Call Ratio

7. Fear and Greed Index

8. Indicators: These state representations utilize fundamental data such as earnings, revenue, or economic indicators to capture market conditions. Examples include:
Price-to-Earnings (P/E) ratio
Price-to-Sales (P/S) ratio
Gross

Note: It's important to note that these state representations are not exhaustive, and the choice of representations may vary depending on the specific analysis or modelling task. Additionally, combining multiple indicators or creating custom representations can provide a more comprehensive view of market conditions.

Defining an Action space

The action space for buying, selling, and holding stocks can be represented using discrete actions. Here's an example of designing the action space:

Buy: This action represents buying stocks. The agent decides to allocate a portion of its available funds to purchase a specific quantity of stocks.

Hold: This action represents holding onto the current position or taking no action. It indicates that no buying or selling of stocks should be performed.

Sell: This action represents selling stocks. The agent decides to sell a portion or all of its current stock holdings.

These actions can be encoded as integers or categorical variables to represent the agent's decision-making process.

- Hold: Action 0
- Buy: Action 1
- Sell: Action 2

The agent, based on its observation of the market conditions and state variables, will select one of these actions at each time step. The chosen action will determine the agent's interaction with the stock market.

Defining a Reward function

To create a reward function based on desired stock return optimization, you need to define a metric that quantifies the performance or profitability of the agent's actions. Here's an example of a reward function that encourages maximizing stock returns:

- For the "buy" action, the reward is set to the stock return, encouraging the agent to make profitable buying decisions.
- For the "sell" action, the reward is the negative of the stock return, penalizing the agent for selling at a lower price.
- For the "hold" action, the reward is set to zero, as no profit or loss is incurred from holding stocks.

The reinforcement learning task is intended to be a simple framing of the challenge of learning from interaction in order to accomplish a goal. The agent is the learner and decision-maker. The environment is the object it interacts with, and it consists of everything outside the agent. The agent chooses actions, and the environment reacts to those actions, presenting the agent with new scenarios. The environment also produces rewards, which are unique numerical values that the agent attempts to maximise over time.

Following is the example of a simple environment created for agent , by defining the state, the action space and the reward function

```python
class Environment1:

    def __init__(self, df, history_t=90):
        self.data = df
        self.history_t = history_t
        self.reset()

    def reset(self):
        self.t = 0
        self.done = False
        self.profits = 0
        self.positions = []
        self.position_value = 0
        self.history = [0 for _ in range(self.history_t)]
        return [self.position_value] + self.history # obs

    def step(self, act):
        reward = 0

        # act = 0: stay, 1: buy, 2: sell
        if act == 1:
            self.positions.append(self.data.iloc[self.t, :]['close'])
        elif act == 2: # sell
            if len(self.positions) == 0:
                reward = -1
            else:
                profits = 0
                for p in self.positions:
                    profits += (self.data.iloc[self.t, :]['close'] - p)
                reward += profits
                self.profits += profits
                self.positions = []

      # set next time
      self.t += 1
      self.position_value = 0
      for p in self.positions:
          self.position_value += (self.data.iloc[self.t, :]['close'] - p)
      self.history.pop(0)
      self.history.append(self.data.iloc[self.t, :]['close'] - self.data.iloc[(self.t-1), :]['close'])

      # clipping reward
      if reward > 0:
          reward = 1
      elif reward < 0:
          reward = -1

      return [self.position_value] + self.history, reward, self.done # obs, reward, done
```

➤ Explanation of the above code

In the provided code, the state is defined by combining the position_value and the history of price changes.

position_value represents the value of the current stock position held by the agent. It is calculated based on the price changes from the previous time step to the current time step.

history is a list that contains the price changes over a fixed historical period (history_t). It keeps track of the changes in stock prices over time.

The state representation combines these two components: [position_value] + history. It represents the current position value and the historical price changes, allowing the agent to observe its current stock position and the recent market trends. The action space in the code consists of three actions:

Action 0: "stay" or "hold" - the agent maintains its current stock position.

Action 1: "buy" - the agent buys stocks at the current price.

Action 2: "sell" - the agent sells the stocks it currently holds.

The agent takes an action at each time step, and based on that action, the environment updates the agent's position and computes the reward.

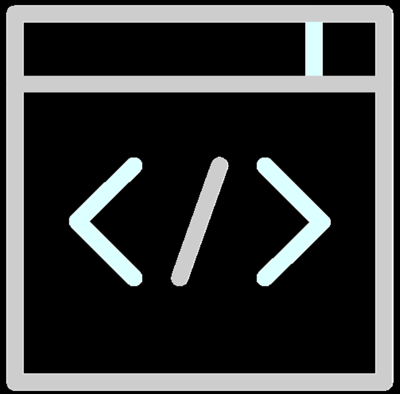
The reward function in this code is relatively simple. It computes the reward based on the agent's action and the resulting change in the position value.

The rewards are assigned as follows:

For the "buy" action (action == 1), the reward is 0 since the agent incurs no immediate profit or loss.

For the "sell" action (action == 2), the reward is equal to the profits earned from selling the stocks held by the agent.

For the "hold" action (action == 0), the reward is 0 since no buying or selling occurs.



Use example as follows:

```
In [19]: env = Environment1(train)
         print(env.reset())
         for _ in range(3):
             pact = np.random.randint(3)
             print(env.step(pact))

[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0]
([14.79, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 14.79], 0, False)
([5.6100000000000065, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 14.79, -4.589999999999996], 0, False)
([10.21, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 14.79, -4.589999999999996, 2.299999999999997], 0, False)
```

...Explanation continued

The code snippet provided demonstrates the usage of the Environment1 class and showcases a random agent's interaction with the environment. Here's a breakdown of the code and its output:

env = Environment1(train): This line initializes an instance of the Environment1 class using the train dataset.

print(env.reset()): The reset() method is called to reset the environment and obtain the initial state. The output is a list containing the initial position value and the historical price changes.

for _ in range(3): This loop iterates three times to simulate three time steps.

pact = np.random.randint(3): A random action (pact) is selected using np.random.randint(3), which generates a random integer between 0 and 2 inclusive. This action represents the agent's decision of "stay" (0), "buy" (1), or "sell" (2).

print(env.step(pact)): The step() method is called with the chosen action pact. It updates the environment based on the action and returns the updated state, reward, and the "done" flag indicating whether the episode is complete.

The output will vary with each run due to the random nature of the agent's actions. It will display the updated state, reward, and the "done" flag for each time step.

- Now creating an environment based on RL algorithm

1.

```python
# DQN

def train_dqn(env):

    class Q_Network(chainer.Chain):

        def __init__(self, input_size, hidden_size, output_size):
            super(Q_Network, self).__init__(
                fc1 = L.Linear(input_size, hidden_size),
                fc2 = L.Linear(hidden_size, hidden_size),
                fc3 = L.Linear(hidden_size, output_size)
            )

        def __call__(self, x):
            h = F.relu(self.fc1(x))
            h = F.relu(self.fc2(h))
            y = self.fc3(h)
            return y

        def reset(self):
            self.zerograds()

    Q = Q_Network(input_size=env.history_t+1, hidden_size=100, output_size=3)
    Q_ast = copy.deepcopy(Q)
    optimizer = chainer.optimizers.Adam()
    optimizer.setup(Q)

    epoch_num = 60
    step_max = len(env.data)-1
    memory_size = 200
    batch_size = 20
    epsilon = 1.0
    epsilon_decrease = 1e-3
    epsilon_min = 0.1
    start_reduce_epsilon = 200
    train_freq = 10
    update_q_freq = 20
    gamma = 0.97
    show_log_freq = 5

    memory = []
    total_step = 0
    total_rewards = []
    total_losses = []

    start = time.time()
    for epoch in range(epoch_num):

        pobs = env.reset()
        step = 0
        done = False
        total_reward = 0
        total_loss = 0
```

2.

```python
        while not done and step < step_max:

            # select act
            pact = np.random.randint(3)
            if np.random.rand() > epsilon:
                pact = Q(np.array(pobs, dtype=np.float32).reshape(1, -1))
                pact = np.argmax(pact.data)

            # act
            obs, reward, done = env.step(pact)

            # add memory
            memory.append((pobs, pact, reward, obs, done))
            if len(memory) > memory_size:
                memory.pop(0)

            # train or update q
            if len(memory) == memory_size:
                if total_step % train_freq == 0:
                    shuffled_memory = np.random.permutation(memory)
                    memory_idx = range(len(shuffled_memory))
                    for i in memory_idx[::batch_size]:
                        batch = np.array(shuffled_memory[i:i+batch_size])
                        b_pobs = np.array(batch[:, 0].tolist(), dtype=np.float32).reshape(batch_size, -1)
                        b_pact = np.array(batch[:, 1].tolist(), dtype=np.int32)
                        b_reward = np.array(batch[:, 2].tolist(), dtype=np.int32)
                        b_obs = np.array(batch[:, 3].tolist(), dtype=np.float32).reshape(batch_size, -1)
                        b_done = np.array(batch[:, 4].tolist(), dtype=bool)

                        q = Q(b_pobs)
                        maxq = np.max(Q_ast(b_obs).data, axis=1)
                        target = copy.deepcopy(q.data)
                        for j in range(batch_size):
                            target[j, b_pact[j]] = b_reward[j]+gamma*maxq[j]*(not b_done[j])
                        Q.reset()
                        loss = F.mean_squared_error(q, target)
                        total_loss += loss.data
                        loss.backward()
                        optimizer.update()

                    if total_step % update_q_freq == 0:
                        Q_ast = copy.deepcopy(Q)

            # epsilon
            if epsilon > epsilon_min and total_step > start_reduce_epsilon:
                epsilon -= epsilon_decrease
            # next step
            total_reward += reward
            pobs = obs
            step += 1
            total_step += 1

        total_rewards.append(total_reward)
        total_losses.append(total_loss)

        if (epoch+1) % show_log_freq == 0:
            log_reward = sum(total_rewards[((epoch+1)-show_log_freq):])/show_log_freq
            log_loss = sum(total_losses[((epoch+1)-show_log_freq):])/show_log_freq
            elapsed_time = time.time()-start
            print('\t'.join(map(str, [epoch+1, epsilon, total_step, log_reward, log_loss, elapsed_time])))
            start = time.time()


    return Q, total_losses, total_rewards
```

Explanation of the above code

In the given code, the state representation used to define the environment's state includes two main components:

position_value: It represents the current value of the position (stocks) held by the agent.

history: It is a list of historical price changes of the stocks.

The state representation is constructed by combining these two components: [position_value] + history.

The action space in this code consists of three possible actions:

0 (Stay): The agent chooses to hold its current position and take no action.

1 (Buy): The agent decides to buy stocks at the current price.

2 (Sell): The agent chooses to sell its existing stocks at the current price.

The agent selects actions randomly (np.random.randint(3)) unless its exploration rate (epsilon) allows it to choose actions based on the Q-network's predictions (np.argmax(pact.data)).

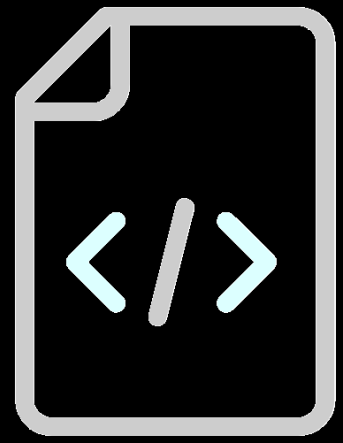
The reward function in the code is based on the profit or loss made by the agent. Here's how it works:

If the agent chooses to sell (act == 2) but has no stocks in its possession (len(self.positions) == 0), it receives a reward of -1 as a penalty.

If the agent sells stocks (act == 2) and has stocks in its possession, it calculates the profit made by selling the stocks at the current price compared to their purchase price. The reward is the total profit obtained from selling all the stocks.

The reward is clipped to be either -1 (for losses) or 1 (for profits) to limit its range.

The goal of the agent is to learn a policy that maximizes its cumulative reward (profit) over time by making optimal decisions on when to buy and sell stocks.

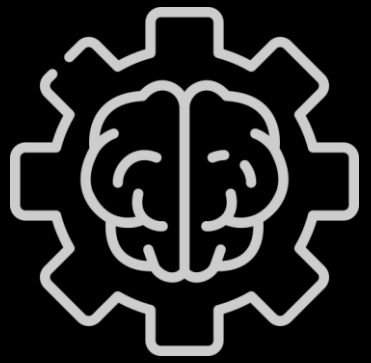


USE:

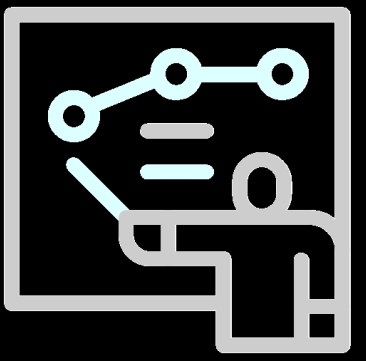
```
Q, total_losses, total_rewards = train_dqn(Environment1(train))
```

```
/opt/conda/lib/python3.7/site-packages/numpy/core/_asarray.py:83: VisibleDeprecationWarning:

Creating an ndarray from ragged nested sequences (which is a list-or-tuple of lists-or-tuple
s-or ndarrays with different lengths or shapes) is deprecated. If you meant to do this, you
must specify 'dtype=object' when creating the ndarray
```

5	0.0999999999999992	4715	-36.4	1269.1880589734762	58.75414991378784
10	0.0999999999999992	9430	68.8	106.02257219804451	62.61797070503235
15	0.0999999999999992	14145	92.4	88.00889271469786	63.70128417015076
20	0.0999999999999992	18860	105.2	81.62211509905755	64.25300574302673
25	0.0999999999999992	23575	130.8	67.72250651605427	65.3246157169342
30	0.0999999999999992	28290	128.8	64.84192206077277	64.26934337615967
35	0.0999999999999992	33005	109.2	37.99149908022955	65.57610964775085
40	0.0999999999999992	37720	102.0	30.748561142757534	66.24323463439941
45	0.0999999999999992	42435	100.0	30.852891511656345	67.29631352424622
50	0.0999999999999992	47150	132.0	39.443355267494915	65.53461694717407
55	0.0999999999999992	51865	137.6	38.289829121623185	65.3227026462555
60	0.0999999999999992	56580	161.4	42.372054285090414	64.54252576828003



Training and Testing the Algorithm



Model evaluation is a step in the model creation process that is often overlooked. This is the stage where the model's performance is determined. As a result, it's important to think about the model's results in terms of every conceivable assessment technique. Using various approaches can result in a variety of viewpoints. The RI agents needs to be continuously trained and tested to make it better and better

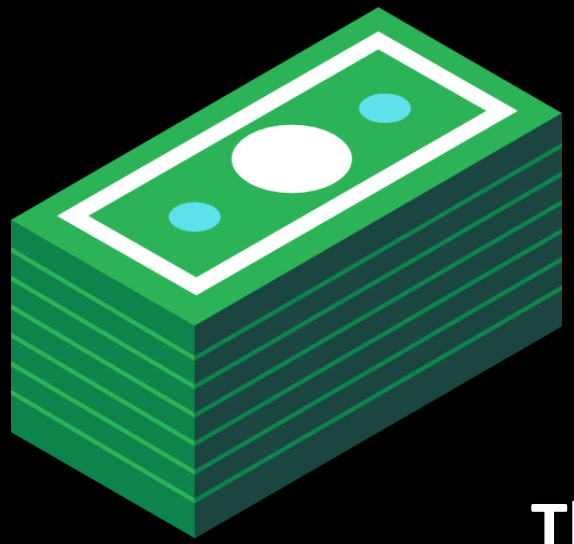
Code:

```python
def plot_train_test_by_q(train_env, test_env, Q, algorithm_name):

    # train
    pobs = train_env.reset()
    train_acts = []
    train_rewards = []

    for _ in range(len(train_env.data)-1):

        pact = Q(np.array(pobs, dtype=np.float32).reshape(1, -1))
        pact = np.argmax(pact.data)
        train_acts.append(pact)

        obs, reward, done = train_env.step(pact)
        train_rewards.append(reward)

        pobs = obs

    train_profits = train_env.profits

    # test
    pobs = test_env.reset()
    test_acts = []
    test_rewards = []

    for _ in range(len(test_env.data)-1):

        pact = Q(np.array(pobs, dtype=np.float32).reshape(1, -1))
        pact = np.argmax(pact.data)
        test_acts.append(pact)

        obs, reward, done = test_env.step(pact)
        test_rewards.append(reward)

        pobs = obs

    test_profits = test_env.profits
```

```python
# plot
train_copy = train_env.data.copy()
test_copy = test_env.data.copy()
train_copy['act'] = train_acts + [np.nan]
train_copy['reward'] = train_rewards + [np.nan]
test_copy['act'] = test_acts + [np.nan]
test_copy['reward'] = test_rewards + [np.nan]
train0 = train_copy[train_copy['act'] == 0]
train1 = train_copy[train_copy['act'] == 1]
train2 = train_copy[train_copy['act'] == 2]
test0 = test_copy[test_copy['act'] == 0]
test1 = test_copy[test_copy['act'] == 1]
test2 = test_copy[test_copy['act'] == 2]
act_color0, act_color1, act_color2 = 'gray', 'cyan', 'magenta'

data = [
    Candlestick(x=train0.index, open=train0['open'], high=train0['high'], low=train0['low'], close=train0['close'], increasing=dict(line=dict(col
    Candlestick(x=train1.index, open=train1['open'], high=train1['high'], low=train1['low'], close=train1['close'], increasing=dict(line=dict(col
    Candlestick(x=train2.index, open=train2['open'], high=train2['high'], low=train2['low'], close=train2['close'], increasing=dict(line=dict(col
    Candlestick(x=test0.index, open=test0['open'], high=test0['high'], low=test0['low'], close=test0['close'], increasing=dict(line=dict(color=ac
    Candlestick(x=test1.index, open=test1['open'], high=test1['high'], low=test1['low'], close=test1['close'], increasing=dict(line=dict(color=ac
    Candlestick(x=test2.index, open=test2['open'], high=test2['high'], low=test2['low'], close=test2['close'], increasing=dict(line=dict(color=ac
]
title = '{}: train s-reward {}, profits {}, test s-reward {}, profits {}'.format(
    algorithm_name,
    int(sum(train_rewards)),
    int(train_profits),
    int(sum(test_rewards)),
    int(test_profits)
)
layout = {
    'title': title,
    'showlegend': False,
     'shapes': [
        {'x0': date_split, 'x1': date_split, 'y0': 0, 'y1': 1, 'xref': 'x', 'yref': 'paper', 'line': {'color': 'rgb(0,0,0)', 'width': 1}}
     ],
    'annotations': [
       {'x': date_split, 'y': 1.0, 'xref': 'x', 'yref': 'paper', 'showarrow': False, 'xanchor': 'left', 'text': ' test data'},
       {'x': date_split, 'y': 1.0, 'xref': 'x', 'yref': 'paper', 'showarrow': False, 'xanchor': 'right', 'text': 'train data '}
    ]
}
figure = Figure(data=data, layout=layout)
iplot(figure)
```

Explanation of the above Code

The plot_train_test_by_q function is used to create a visualization of the actions taken by an agent during the training and testing phases of a trading algorithm. It generates a candlestick chart where different colors represent different actions: holding (gray), buying (cyan), and selling (magenta).



The function takes the following inputs:

- train_env: The training environment object.
- test_env: The testing environment object.
- Q: The trained Q-network model.
- algorithm_name: The name of the algorithm used.

The function performs the following steps:

- Training Phase Visualization
- Testing Phase Visualization

Training Phase Visualization

It resets the training environment (train_env) and initializes empty lists to store the actions and rewards.

For each time step in the training data:

It uses the trained Q-network (Q) to select an action (pact) based on the current observation (pobs).

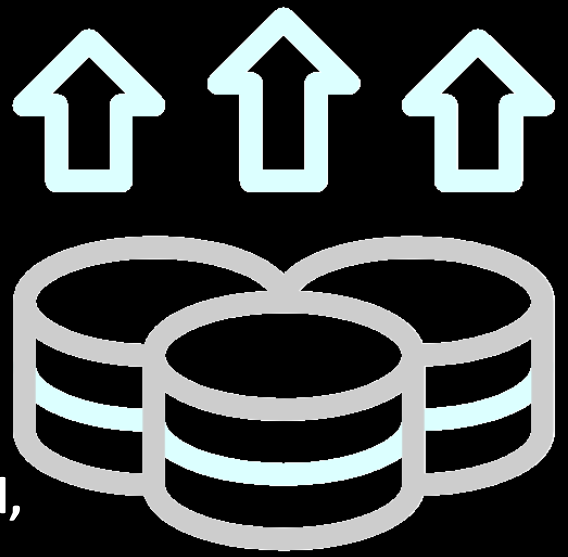
It appends the selected action to the train_acts list.

It executes the action in the environment and receives the next observation, reward, and done flag.

It appends the received reward to the train_rewards list.

It updates the current observation (pobs) with the next observation.

It calculates the total profits (train_profits) obtained during the training phase.

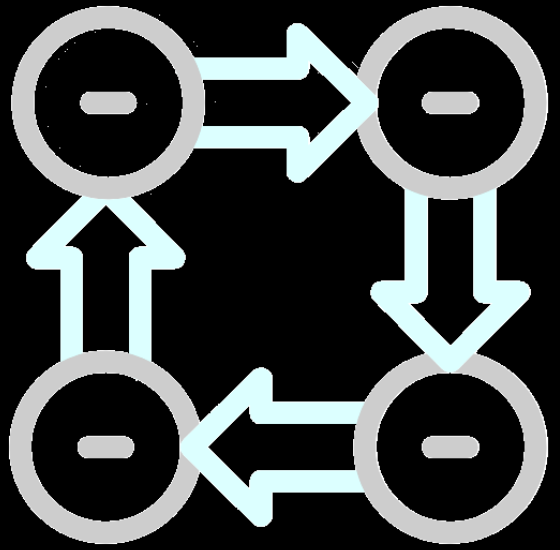


Testing Phase Visualization

It resets the testing environment (test_env) and initializes empty lists to store the actions and rewards.

For each time step in the testing data:

- It uses the trained Q-network (Q) to select an action (pact) based on the current observation (pobs).
- It appends the selected action to the test_acts list.
- It executes the action in the environment and receives the next observation, reward, and done flag.
- It appends the received reward to the test_rewards list.
- It updates the current observation (pobs) with the next observation.
- It calculates the total profits (test_profits) obtained during the testing phase.



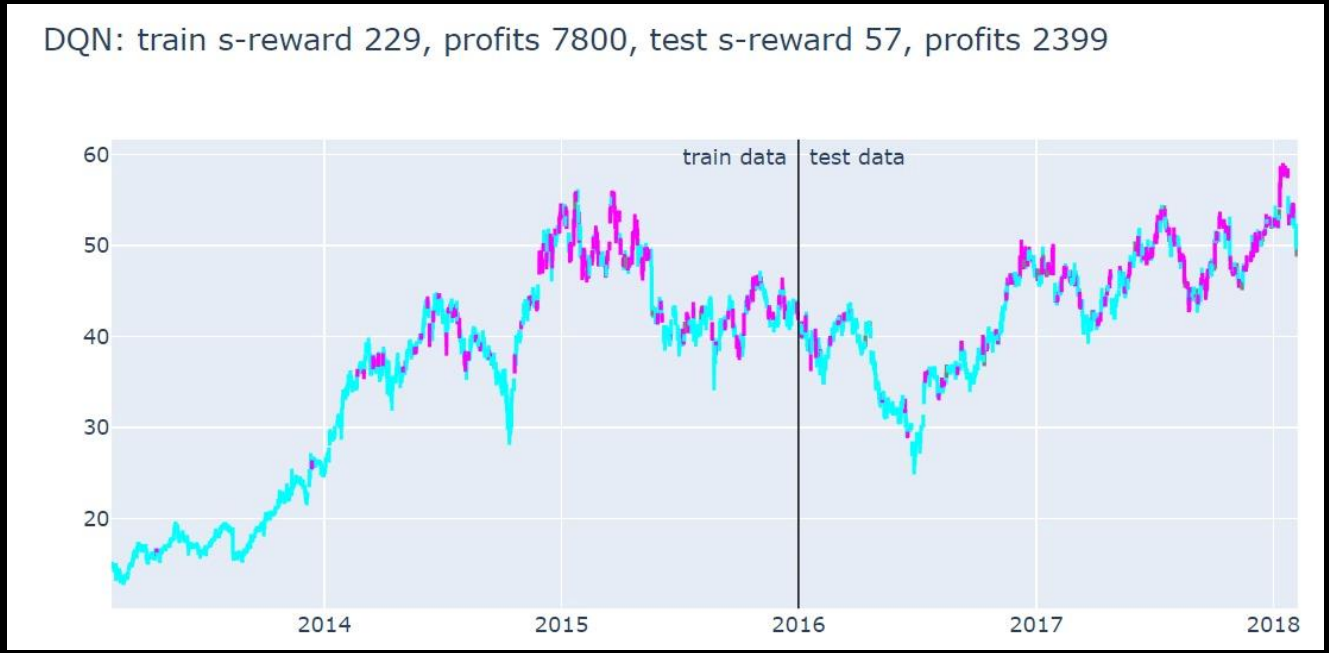
Plotting the Candlestick Chart:

- It creates separate data sets for different actions (hold, buy, sell) in the training and testing phases.
- It defines the colours for each action.
- It creates a candlestick chart with the training and testing data, using the defined colours.
- It adds a vertical line at the split point between training and testing data.
- It adds annotations to indicate the training and testing data regions.

Results

The resulting candlestick chart provides a visual representation of the actions taken by the trading algorithm during the training and testing phases, along with the associated rewards and profits

```
plot_train_test_by_q(Environment1(train), Environment1(test), Q, 'DQN')
```



Roadmap 4

References:

- https://developers.google.com/machine-learning/crash-course/ml-intro
- https://www.youtube.com/live/N5fSpaaxoZc?feature=share
- https://youtube.com/playlist?list=PLZbbT5o_s2xoWNVdDudn51XM8lOuZ_Njv
- https://youtu.be/D9sU1hLT0QY
- https://towardsdatascience.com/deep-reinforcement-learning-for-automated-stock-trading-f1dad0126a02
- https://youtu.be/gtjxAH8uaP0?si=ZRWA-589N0l0Is6M
- https://youtu.be/tPYj3fFJGjk?si=7h2BlG5ZY6YLEUU9

Thank You