<u>**CAPSTONE PROJECT REPORT**</u>

REINFORCEMENT LEARNING-BASED TRADING STRATEGY

By,

Ally Saha

Antarlin Chanda

Gulivindala Naveen Kumar

Harendra Sai Nath Lella

Shramana Bhattacharya

Tamosa Sur

Under the Guidance of,

Prof. Jaydip Sen

Praxis Business School, Kolkata, INDIA

PGPDS 2022

Acknowledgments

We would like to express our gratitude and special thanks to our professor Jaydip Sen who gave us the possibility to complete this report and encouraged us in all time of coding challenges and helped us in writing this report. We also sincerely thank him for the time spent proofreading and correcting our mistakes.

We would also thank all our professors for guiding us all the way in making all the concepts very clear and a special thanks for their utmost effort in guiding us to achieve our goal as well as their encouragement to maintain our progress on track. We would also like to acknowledge with much appreciation the crucial role of the college staff, who gave us access to the library and all necessary materials as and when required.

Our thanks to all our classmates and all our friends for spending their time and helping us by extending their support whenever needed.

Abstract

The trading strategy is the task of identifying a set of capital assets with respect to their weights of allocation and developing a policy that determines the best action to take based on the current state of the market and also by considering past market activity. Preparing a trading strategy is computationally hard as it involves identifying parameters and hyperparameters and optimizing them. Finally, backtesting the viability of the trading strategy with the help of historical data.

Reinforcement Learning is a non-policy neural network algorithm where it builds a policy on its own based on the historical data present and the parameters and the hyperparameters present in the model. Their ability to decide a policy on their own makes RL models to be suitable machine learning algorithms for creating trading bots.

In our project, we have created an end-to-end trading strategy based on Reinforcement Learning. We used the stock price data of SP500, NIFTY50, Adani Enterprises and Reliance Industries from yahoo finance. We used a Q-learning approach with Deep Q-Network (DQN) to come up with a policy and then implement the trading strategy. The architecture of RL is adapted based on the historical stock prices of a given sector (i.e., on the training data) and the reward function determines whether to buy, sell or hold the stocks.

Introduction

Reinforcement learning; one of the most popular machine learning methods used today; enables a computer system to learn how to make choices by being rewarded for its successes [1]. RL can be considered to be an extremely powerful tool for optimization and decision-making. It is an approach to machine learning in which the agents are trained to make a sequence of decisions. An agent can learn a sequence of actions to maximize rewards and thereby achieve the desired goal.

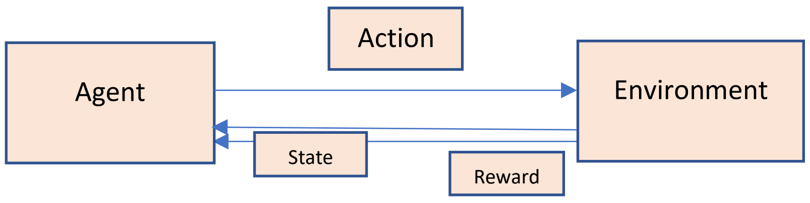
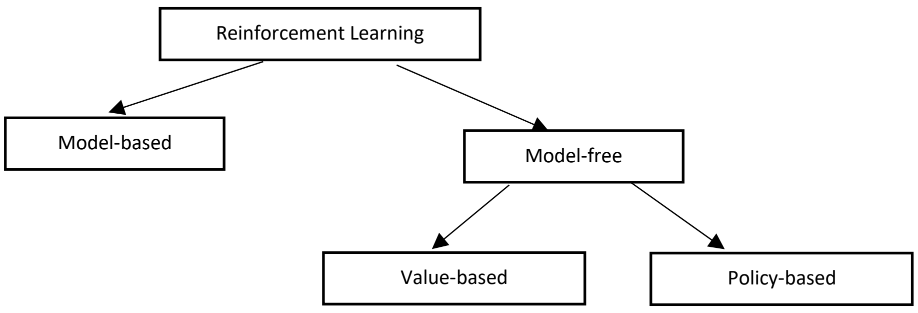


Fig: Typical RL scenario

In the above diagram, the agent takes an action in the given environment in a state. A response is sent by the to the agent in form of reward and the new state information. Change of state takes place as a result of action taken by the agent. [2]

Reinforcement learning broad classification [3]:



- **Model - based:** A virtual model is created for the agent to learn and perform in each specific environment.
- **Value – based:** Main goal here is to maximize the value function.

- **Policy – based:** Aim is to gain maximum rewards in future through a proper strategy attained by possible actions in each state. It can be deterministic or stochastic.

Learning Models of Reinforcement [4]

- **Markov Decision Process (MDP)**
- **Q learning**

Reinforcement learning has various applications in the real world like robotics for industrial automation, machine learning, data preprocessing, aircraft control, robot motion control, etc. RL helps to find out which situations need action or which action will lead to the highest reward over a longer period and also obtain the best method for larger rewards.

However, when there is enough data to solve the problem using supervised machine learning, then RL is not recommended since it requires heavy computing, huge time, and large action space.

In this project, we are building an RL agent that would automatically take decisions of buy, sell or hold in stock trading. Simple buy and sell strategies can be outperformed by RL agents with the help of MDP and model using DQN. Humans cannot consider such wide range of possible states and take actions as efficiently as an RL agent. Thus, with proper training, an RL agent would easily take optimum buy, sell or hold decisions which would lead to better trading strategies in the long run.

Literature Review

Researchers in a paper; applied high-dimensional sensory input using reinforcement learning to seven Atari 2600 games from the Arcade Learning Environment, with no adjustment of the architecture or learning algorithm, which outperformed all previous approaches on six of the games and surpassed a human expert on three of them [5]. Another article investigates the effects of adding recurrency to a Deep Q-Network (DQN) by replacing the first post-convolutional fully-connected layer with a recurrent LSTM [6]. In a paper, scholars show that the recent DQN algorithm, which combines Q-learning with a deep neural network, suffers from substantial overestimations in some games in the Atari 2600 domain and that the idea behind the Double Q-learning algorithm, which was introduced in a tabular setting, can be generalized to work with large-scale function approximation. They propose a specific adaptation to the DQN algorithm and show that the resulting algorithm not only reduces the observed overestimations, as hypothesized but also leads to much better performance on several games [7]. In an article, it has been investigated to find out whether a simple model of hippocampal episodic control can learn to solve difficult sequential decision-making tasks and demonstrate that it not only attains a highly rewarding strategy significantly faster than state-of-the-art deep reinforcement learning algorithms but also achieves a higher overall reward on some of the more challenging domains [8]. In another paper, the generalization of the approach of the game into a single AlphaZero algorithm that can achieve, tabula rasa, superhuman performance in many challenging domains has been done. Starting from random play, and given no domain knowledge except the game rules, AlphaZero achieved within 24 hours a superhuman level of play in the games of chess and shogi (Japanese chess) as well as Go, and convincingly defeated a world-champion program in each case [9]. A novel reinforcement learning algorithm that decomposes the problem into separate planning and generalization tasks has been proposed by Thomas Anthony et al in their research work [10]. One of the notable research works introduces several reinforcement learning tasks with multiple commercially available robots that present varying levels of learning difficulty, setup, and repeatability. On these tasks, they test the learning performance of off-the-shelf implementations of four reinforcement learning algorithms and analyze sensitivity to their hyper-parameters to determine their readiness for applications in various real-world tasks. The results show that with a careful setup of the task interface and computations, some of these implementations can be readily applicable to physical robots [11]. An article presented Horizon, Facebook's open-source applied reinforcement learning (RL) platform. Horizon is an end-to-end platform designed to solve industry-applied RL problems where datasets are large (millions to billions of observations), the feedback loop is slow (vs. a simulator), and experiments must be done with care because they don't run in a simulator [12].

Methodology

We have built reinforcement learning modal based agents for the stock prices of SP500, NIFTY50, Reliance and Adani using the historical data of 2010-2019. The historical data has been extracted from yahoo finance. In order to build an RL agent, we have used an open-source software library called Keras, which provides a python interface for artificial neural networks. Keras acts as an interface for the TensorFlow library.

Basic Exploratory Data Analysis (EDA) is done on the dataset in order to check for missing values. As the dataset is about the stock price data which is updated daily, if at there are any missing values in the dataset, we fill those missing values by using a method forward fill, which fills the subsequent missing values with the previous value.

Since our dataset is ready, we now need to split it. We split the dataset into training and testing in the ratio 80:20 respectively. The validation size of the dataset is kept at 50%. Since our dataset is ready, we now need to define the agent class.

The agent class defines important parameters like gamma, epsilon, epsilon_min, epsilon_decay, state_size, window_size, action_size, memory, and also is_eval parameter as well. As the agent class has been defined, now we need to define the model function.

In the model function, we define it to be sequential in nature and define Deep Neural Network architecture here, which will be used to estimate the Q values. We also define the number of input, hidden and the outout layers. The next thing we do is define the function called act, which determines whether to take action or not based on the Q value.

The expReplay function is defined before testing the dataset. This function stores the history of the state, action, reward, and next state transition in the memory and the new experiences are added to the replay buffer memory using a for loop. The target variable for the Q-table is updated based on the   Bellman equation. The update happens if the current state is the terminal state or the end of the episode. This is represented by the variable done and is defined further in the training function. If it is not done, the target is just set to reward. Predict the Q-value of the next state using a deep learning model. The Q-value of this state for the action in the current replay buffer is set to the target. The deep learning model weights are updated by using the model.fit function. The epsilon greedy approach is implemented. Recall that this approach selects an action randomly with a probability of e or the best action, according to the Q- value function, with probability l - e.

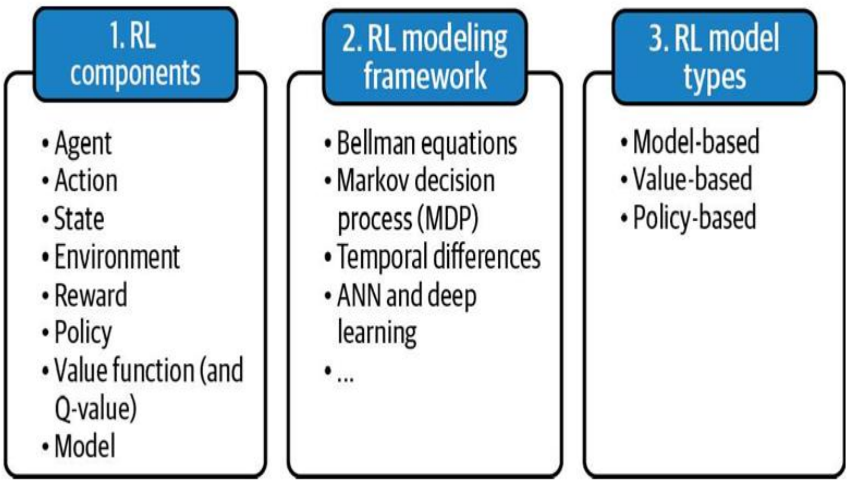
Helper functions are defined for further aiding the training process and the plot behavior function prints the corresponding output graphs.

<u>REINFORCEMENT LEARNING-BASED TRADING STRATEGY</u>

A stock market is a place there is immense opportunity to make profits. But stock market trading is a complex problem with many internal and external factors and unpredictability associated with it.

<u>Design</u>

All Basic components of RL was used to develop this project. Understanding these basic components of RL are essential to understanding the project.



Agent-The entity we create which will perform the actions.

Actions-The activity that the agent performs on the environment

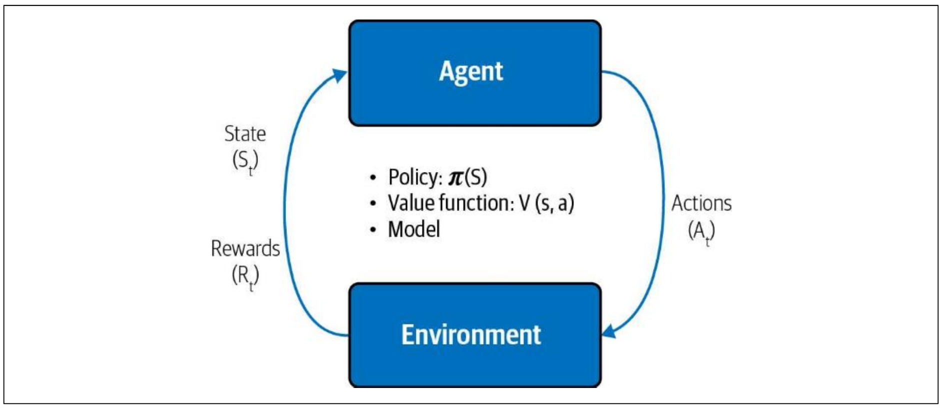
Environment-The space where the agent is located and will perform actions

State-The current situation

Reward-Immediate return sent by the environment to evaluate the last action by the agent.

The goal of our RL will be to generate optimal strategy through experimental trials and a simple feedback loop.

With a strategy that the RL algorithm formulates with learning, the agent is actively able to take decisions that will maximize its rewards.



The sequence of steps is explained below:

The agent is fed a state and based on that the agent takes an action so that it wants to maximize its rewards in the next steps .Now the above diagram explains this decision-making process.

The agent based on the optimization of maximization of rewards/value got decides the rules system(policy) it should use to make a decision regarding what actions the agent should take to maximize future rewards.

Value function-The reward concept is explained through the value function.

This give us the potential immediate reward along with all the potential future rewards. This value has to be discounted due to the fact that future rewards inherently have uncertainity associated with them in the sense they may not be realized.So we have a discounting factor of γ for this purpose.

$Gt = Rt+1 + \gamma Rt+2 + \ldots = \sum ykRt+k+1$

Policy- The rule followed by RL Agent to decide the action it will take at a particular state

$at = \pi(st)$

Value Function(or state Value)- Measures attractiveness of a state through prediction of future reward G for being at that state

$V(s) = E[Gt|St=s]$

Action-Value Function (Q-value) of a state-action pair (s, a)-Expected Reward given a state and action

$Q(s, a) = E[Gt|St = s, At = a]$

Value Function and Q value interaction- The product of probability distribution of action, state pair and the q value associated with that state and action summed over all pairs gives us back the value function.

This is intuitive as product of probability of taking an action multiplied by the Q value associated with taking that action at that particular state and then summed up over all possible states should always give us the net expected value for being in some particular state.

$V(s) = \sum_{a \in A} Q(s,a)\pi(a|s)$

Ultimately all of the above are used to optimize the Bellman Equation wich is ultimately what we use to improve our algorithm iteratively.

Bellman Equation:- A set of equations that decompose the value function and Q-value into the immediate reward plus the discounted future values.

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

$$\mathbf{q^*(s,a) = E[R_{t+1} + \gamma maxa'q^*(s',a')]}$$

It is used to find optimal q∗ in order to find optimal policy π and thus a reinforcement learning algorithm can find the action a that maximizes q∗(s, a). That is why this equation has its importance.

The simplified iteration algorithms for the value function and Q-value are:

Iteration algorithm for value function

$$V_{k+1}(s) = \max_a \sum_{s'} P_{ss'}\left(R^a_{ss'} + \gamma V_k(s')\right)$$

Iteration algorithm/or Q-value

$$Q_{k+1}(s,\, a) = \sum P^a_{ss'}[R^a_{ss'} + \gamma * max * Q_k(s',\, a')]$$

where,

$P_{ss'}^{a}$ is the transition probability from state s to state s', given that action a was chosen.

$R_{ss'}^{a}$ is the reward that the agent gets when it goes from state *s* to state s', given that action a was chosen.

If we have the above values we can easily calculate the value associated with each state but in real life scenarios we do not have idea about the transition probabilities and rewards for state to state transition. Thus we cant apply bellman eqn s directly.

Markov Decision Process

The above MDP diagram can be easily used to find the optimal policy or strategy to achieve the most required over time. For example, in state S_0 hold is the best option, in state S_2 buy is the best option but in the stagnant market, it's not clear whether hold or sell.

For MDP we can use the following Bellman Equation:

$$\mathbf{Q_{k+1}(s,a)=\sum_{s'} P_{ss'}^{a}[R_{ss'}^{a} + \gamma * max Q_k(s',a')]}$$

From this, we can calculate the Q value.

But in the real world, we have no knowledge abt the Rewards and the transition probabilities so we can't use the above MDP and Bellman Eqn to devise an optimum strategy.

To solve the above problem we resort to Temporal Difference Learning.

<u>Temporal Difference Learning</u>

Similar to the value iteration algorithm got above but tweaked to account that agent have partial knowledge of MDP.

Here the agent only knows the possible states and actions. For example, the agent uses an exploration policy, a purely random policy, to explore the MDP, and as it progresses, the TD learning algorithm updates the estimates of the state values based on the transitions and rewards that are actually observed.

The key idea in TD learning is to update the value function V(S,) toward an estimated return (known as the *TD target).*

$$\mathbf{V(st) \leftarrow V(st) + \alpha(Rt+1 + \gamma V(st+1) - V(st))}$$

Similarly, for Q-value estimation:

$$\mathbf{Q(st, at) \leftarrow Q(st, at) + \alpha(Rt+1 + \gamma Q(st+1, at+1) - Q(st, at))}$$

<u>Q-Learning</u>

Q-learning is an adaptation of TD learning. The algorithm evaluates which action to take based on a Q-value (or action-value) function that determines the value of being in a certain state and taking a certain action at that state. For each state-action pair *(s, a)*, this algorithm keeps track of a running average of the rewards, R, which the agent gets upon leaving the state s with action *a,* plus the rewards it expects to earn later.

Since the target policy would act optimally, we take the maximum of the Q-value estimates for the next state.

The learning proceeds off policy—that is, the algorithm does *not* need to select actions based on the policy that is implied by the value function alone. However, convergence requires that all state-action pairs continue to be updated throughout the training process, and a straightforward way to ensure that this occurs is to use an ε- *greedy* policy, which is defined further in the following section.

The steps of Q-learning are:

1. At time step f, we start from **state *s,*** and pick an action according to Q-values, a_t=max Q(s_t,*a).*

2. We apply an ε *-greedy* approach that selects an action randomly with a probability of e or otherwise chooses the best action according to the Q-value function. This ensures the exploration of new actions in a given state while also exploiting the learning experience.'
3. With action a„ we observe reward R_{t+1} and get into the next state S_{t+1}.
4. We update the action-value function:
   Q(st, at) ← Q(st, at) + α(Rt+1 + γmaxQ(st+1, at+1) – Q(st, at))

5. We increment the time step,t=t+1, and repeat the steps.

Given enough iterations of the steps above, this algorithm will converge to the optimal Q-value.

<u>Deep Q Network</u>

Though Q learning is very much suited to our needs but it had some problems like

- In cases where the state and action space are large, the optimal Q-value table quickly becomes computationally infeasible.
- Q-learning may suffer from instability and divergence.

To address these shortcomings, we use ANNs to approximate Q-values. For example, if we use a function with parameter 8 to calculate Q-values, we can label the Q-value function as Q(s,oi8). The deep Q-learning algorithm approximates the Q-values by learning a set of weights, 8, of a multilayered deep Q-network that maps states to actions. The algorithm aims to greatly improve and stabilize the training procedure of Q-learning through two innovative mechanisms:

Experience replay

Instead of running Q-learning on state-action pairs as they occur during simulation or actual experience, the algorithm stores the history of state, action, reward, and next state transitions that are experienced by the agent in one large *replay memory*. This can be referred to as **a *mini-batch of*** observations. During Q-learning updates, samples are drawn at random from the replay memory, and thus one sample could be used multiple times. Experience replay improves data efficiency, removes correlations in the observation sequences, and smooths over changes in the data distribution.

Periodically updated target

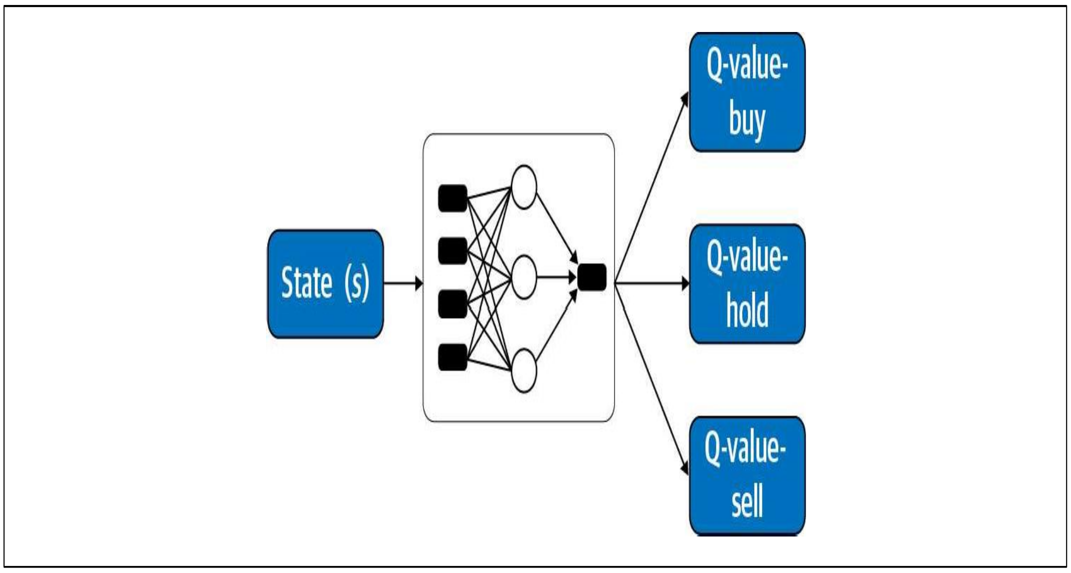
Q is optimized toward target values that are only periodically updated. The Q-network is cloned and kept frozen as the optimization targets every C step (C is a hyperparameter). This modification makes the training more stable as it overcomes the short-term oscillations. To learn the network parameters, the algorithm applies *gradient descent'* to a loss function defined as the squared difference between the DQN's estimate of the target and its estimate of the Q- value of the current state-action pair, ***Q(s,a:θ).*** The loss function is as follows:

$$\mathbf{L(\theta i) = E[(r + \gamma maxQ(s', a'; \theta i\text{-}1) - Q(s, a; \theta i))^2]}$$

The loss function is essentially a mean squared error (MSE) function, where (r + ymaxq Q(s ', a', θ_{i-1})) represents the target value and Q[s,a;θ_i] represents the predicted value. θ are the weights of the network, which are computed when the loss function is minimized. Both the target and the current estimate depend on the set of weights, underlining the distinction from supervised learning, in which targets are fixed prior to training.

An example of the DQN for the trading example containing buy, sell, and hold actions is represented below. Here, we provide the network only the state (s) as input, and we receive Q-values for all possible actions (i.e., buy, sell, and hold) at once.

The theoretical setup needed is complete:



Now we go into program design:

We setup the following:

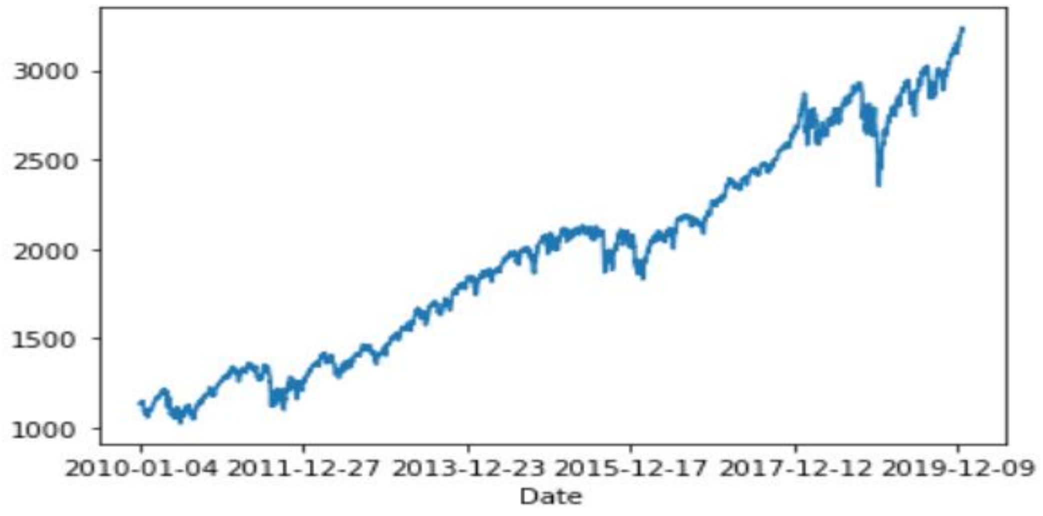
- Agent
- Action
- Reward Function
- State- A sigmoid function" of the differences of past stock prices for a given time window is used as the state. State S_t is described as $(d_{t-\tau+1}, d_{t-1}, d_t)$, where

  $D_t = sigmoid(p_t, p_{t-1})$ p_t is the price at time t, and τ is the time window size. A sigmoid function converts the differences of past stock prices into a number between zero and one, which helps to normalize the values to probabilities and makes the state simpler to interpret.

- Environment
- Reward Function

<u>Program Design</u>

1. Loading necessary python packages
2. Loading Dataset
3. EDA
4. Data Preparation through checking for null values and replacing them with the previous values. Train Test Split
5. Model Development



<u>S&P500 Data (2010-2019)</u>

<u>Modules and functions</u>

Implementing this DQN algorithm requires the implementation of several functions and modules that interact with each other during the model training. Here is a summary of the modules and functions:

Agent class

The agent is defined as an "Agent" class. This holds the variables and member functions that perform the Q-learning. An object of the Agent class is created using the training phase and is used for training the model.

Helper function

In this module, we create additional functions that are helpful for training.

Training module

In this step, we perform the training of the data using the variables and the functions defined in the agent and helper methods. During training, the prescribed action for each day is predicted, the rewards are computed, and the deep learning—based Q-learning model weights are updated iteratively over a number of episodes. Additionally, the profit and loss of each action are summed to determine whether an overall profit has occurred. The aim is to maximize the total profit.
Let us look at each of these in detail.

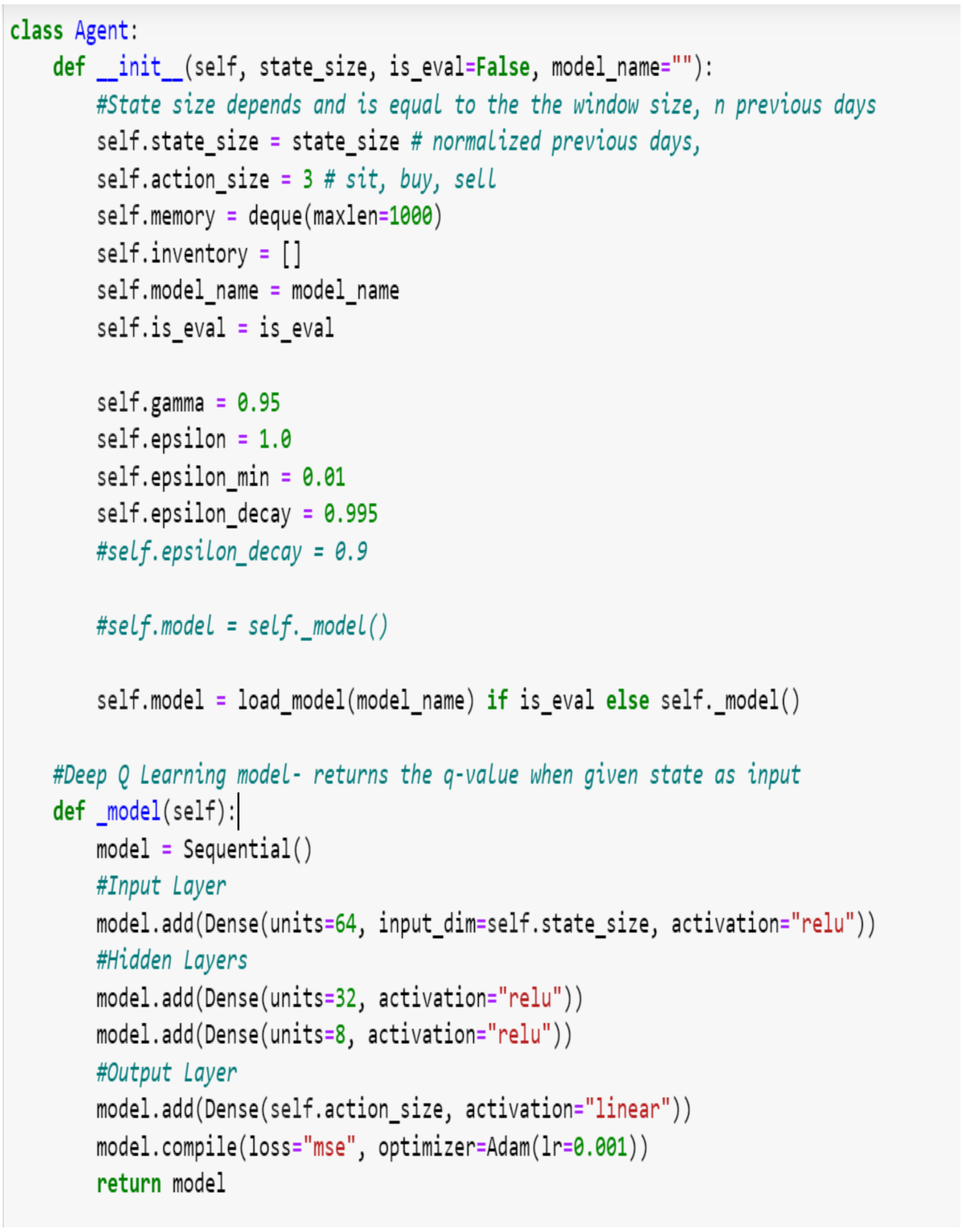
Agent script

The definition of the Agent script is the key step, as it consists of the In this section, we will train an agent that will perform reinforcement learning based on the Q-Learning. We will perform the following steps to achieve this:

- Create an agent class whose initial function takes in the batch size, state size, and an evaluation Boolean function, to check whether the training is ongoing.
- In the agent class, create the following methods:
    - Constructor: The constructor initializes all the parameters.
    - Model: This function has a deep learning model to map the state to action.
    - Act function: Returns an action, given a state, using the output of the model function. The number of actions is defined as 3: sit, buy, sell
    - expReplay: Create a Replay function that adds, samples, and evaluates a buffer. Add a new experience to the replay buffer memory. Randomly sample a batch of experienced tuples from the memory. In the following function, we randomly sample states from a memory buffer. Experience replay stores a history of the state, action, reward, and next-state transitions that are experienced by the agent. It randomly samples mini-batches from this experience to update the network weights at each time step before the agent selects an ε-greedy action.

Experience replay increases sample efficiency, reduces the autocorrelation of samples that are collected during online learning, and limits the feedback due to the current weights producing training samples that can lead to local minima or divergence.

Screenshots taken from Python program



```python
class Agent:
    def __init__(self, state_size, is_eval=False, model_name=""):
        #State size depends and is equal to the the window size, n previous days
        self.state_size = state_size # normalized previous days,
        self.action_size = 3 # sit, buy, sell
        self.memory = deque(maxlen=1000)
        self.inventory = []
        self.model_name = model_name
        self.is_eval = is_eval

        self.gamma = 0.95
        self.epsilon = 1.0
        self.epsilon_min = 0.01
        self.epsilon_decay = 0.995
        #self.epsilon_decay = 0.9

        #self.model = self._model()

        self.model = load_model(model_name) if is_eval else self._model()

    #Deep Q Learning model- returns the q-value when given state as input
    def _model(self):
        model = Sequential()
        #Input Layer
        model.add(Dense(units=64, input_dim=self.state_size, activation="relu"))
        #Hidden Layers
        model.add(Dense(units=32, activation="relu"))
        model.add(Dense(units=8, activation="relu"))
        #Output Layer
        model.add(Dense(self.action_size, activation="linear"))
        model.compile(loss="mse", optimizer=Adam(lr=0.001))
        return model
```

```
def act(self, state):
    #If it is test and self.epsilon is still very high, once the epsilon become low, there are no random
    #actions suggested.
    if not self.is_eval and random.random() <= self.epsilon:
        return random.randrange(self.action_size)
    options = self.model.predict(state)
    #set_trace()
    #action is based on the action that has the highest value from the q-value function.
    return np.argmax(options[0])

def expReplay(self, batch_size):
    mini_batch = []
    l = len(self.memory)
    for i in range(l - batch_size + 1, l):
        mini_batch.append(self.memory[i])

    # the memory during the training phase.
    for state, action, reward, next_state, done in mini_batch:
        target = reward # reward or Q at time t
        #update the Q table based on Q table equation
        #set_trace()
        if not done:
            #set_trace()
            #max of the array of the predicted.
            target = reward + self.gamma * np.amax(self.model.predict(next_state)[0])

        # Q-value of the state currently from the table
        target_f = self.model.predict(state)
        # Update the output Q table for the given action in the table
        target_f[0][action] = target
        #train and fit the model where state is X and target_f is Y, where the target is updated.
        self.model.fit(state, target_f, epochs=1, verbose=0)

    if self.epsilon > self.epsilon_min:
        self.epsilon *= self.epsilon_decay
```

Helper Functions

In this script, we will create functions that will be helpful for training. We create the following functions:

1) format price: format the price to two decimal places, to reduce the ambiguity of the data:

2) getStockData: Return a vector of stock data from the CSV file. Convert the closing stock prices from the data to vectors, and return a vector of all stock prices.

3) getState: Define a function to generate states from the input vector. Create the time series by generating the states from the vectors created in the previous step. The function for this takes three parameters: the data; a time, t (the day that you want to predict); and a window (how many days to go back in time). The rate of change between these vectors will then be measured and based on the sigmoid function.

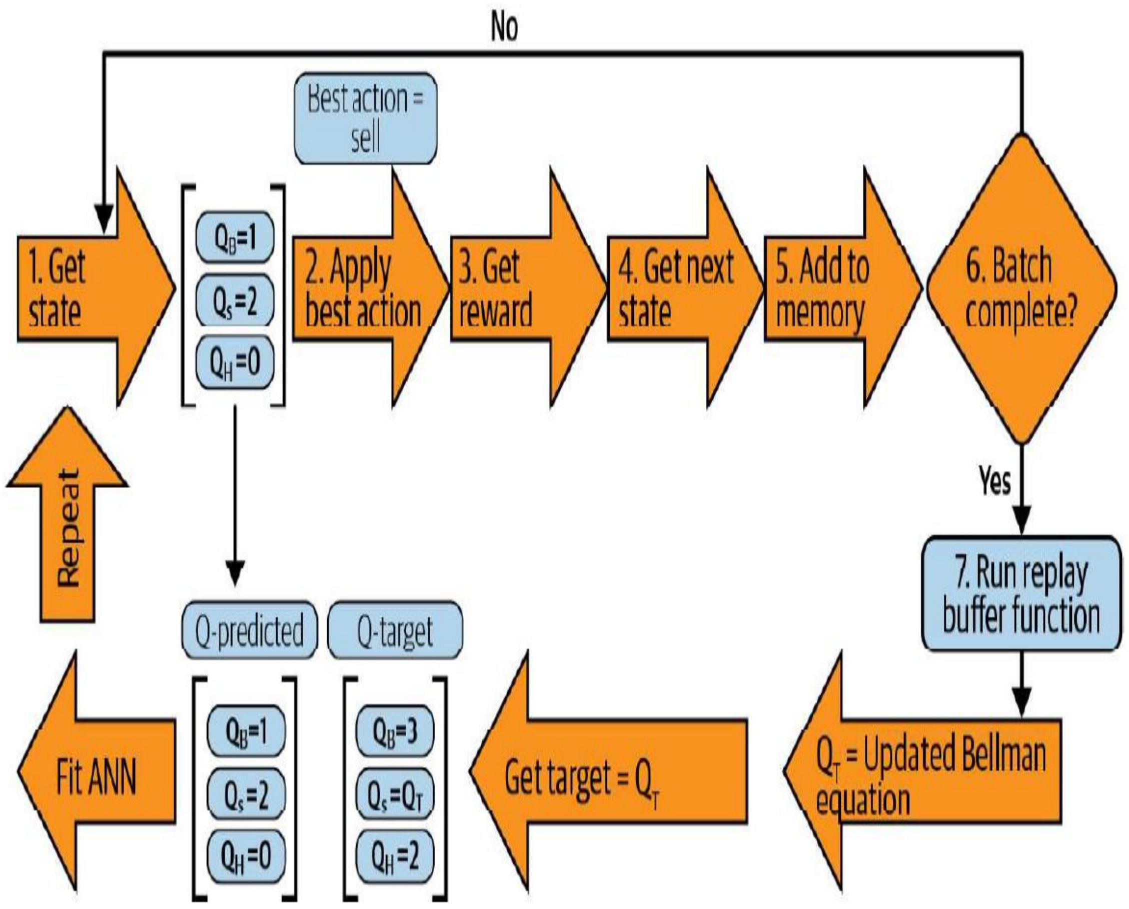
```python
import numpy as np
import math

# prints formatted price
def formatPrice(n):
    return ("-$" if n < 0 else "$") + "{0:.2f}".format(abs(n))

# # returns the vector containing stock data from a fixed file
# def getStockData(key):
#     vec = []
#     lines = open("data/" + key + ".csv", "r").read().splitlines()

#     for line in lines[1:]:
#         vec.append(float(line.split(",")[4])) #Only Close column

#     return vec

# returns the sigmoid
def sigmoid(x):
    return 1 / (1 + math.exp(-x))

# returns an an n-day state representation ending at time t

def getState(data, t, n):
    d = t - n + 1
    block = data[d:t + 1] if d >= 0 else -d * [data[0]] + data[0:t + 1] # pad with t0
    #block is which is the for [1283.27002, 1283.27002]
    res = []
    for i in range(n - 1):
        res.append(sigmoid(block[i + 1] - block[i]))
    return np.array([res])

# Plots the behavior of the output
def plot_behavior(data_input, states_buy, states_sell, profit):
    fig = plt.figure(figsize = (15,5))
    plt.plot(data_input, color='r', lw=2.)
    plt.plot(data_input, '^', markersize=10, color='m', label = 'Buying signal', markevery = states_buy)
    plt.plot(data_input, 'v', markersize=10, color='k', label = 'Selling signal', markevery = states_sell)
    plt.title('Total gains: %f'%(profit))
    plt.legend()
    #plt.savefig('output/'+name+'.png')
    plt.show()
```

<u>**Training the data**</u>

We will proceed to train the data, based on our agent and helper methods. This will provide us with one of three actions, based on the states of the stock prices at the end of the day. These states can be to buy, sell, or hold. During training, the prescribed action for each day is predicted, and the price (profit, loss, or unchanged) of the action is calculated. The cumulative sum will be calculated at the end of the training period, and we will see whether there has been a profit or a loss. The aim is to maximize the total profit.

Steps:

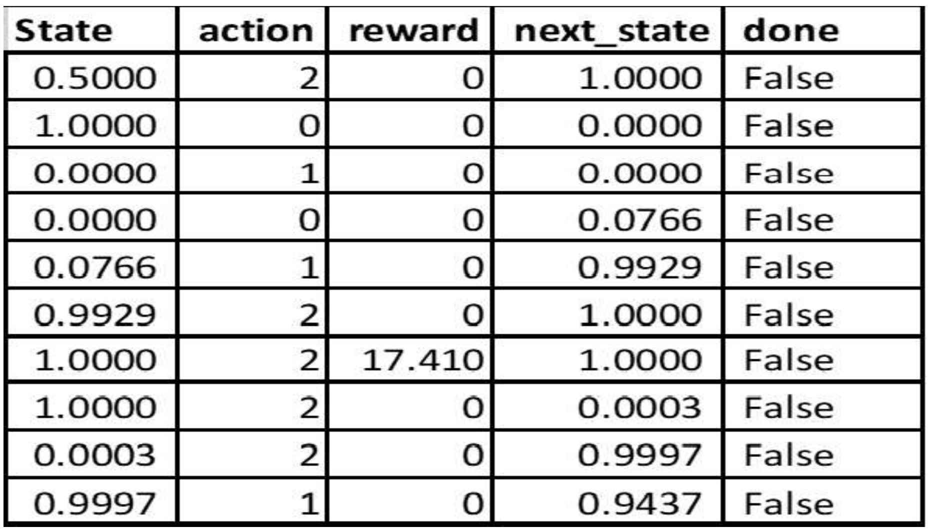
- Define the number of market days to consider as the window size and define the batch size with which the neural network will be trained.
- Instantiate the stock agent with the window size and batch size.
- Read the training data from the CSV file, using the helper function.
- The episode count is defined. The agent will look at the data for so many numbers of times. An episode represents a complete pass over the data.
- We can start to iterate through the episodes.
- Each episode has to be started with a state based on the data and window size. The inventory of stocks is initialized before going through the data.
- Start to iterate over every day of the stock data. The action probability is predicted by the agent**.**
- Next, every day of trading is iterated, and the agent can act upon the data. Every day, the agent decides on an action. Based on the action, the stock is held, sold, or bought.
- If the action is 1, then the agent buys the stock.
- If the action is 2, the agent sells the stocks and removes it from the inventory. Based on the sale, the profit (or loss) is calculated.
- If the action is 0, then there is no trade. The state can be called holding during that period.
- The details of the state, next state, action, etc is saved in the memory of the agent object, which is used further by the exeReply function.



Steps 1 to 6 shown above are numbered in the following Python code and are described as follows:

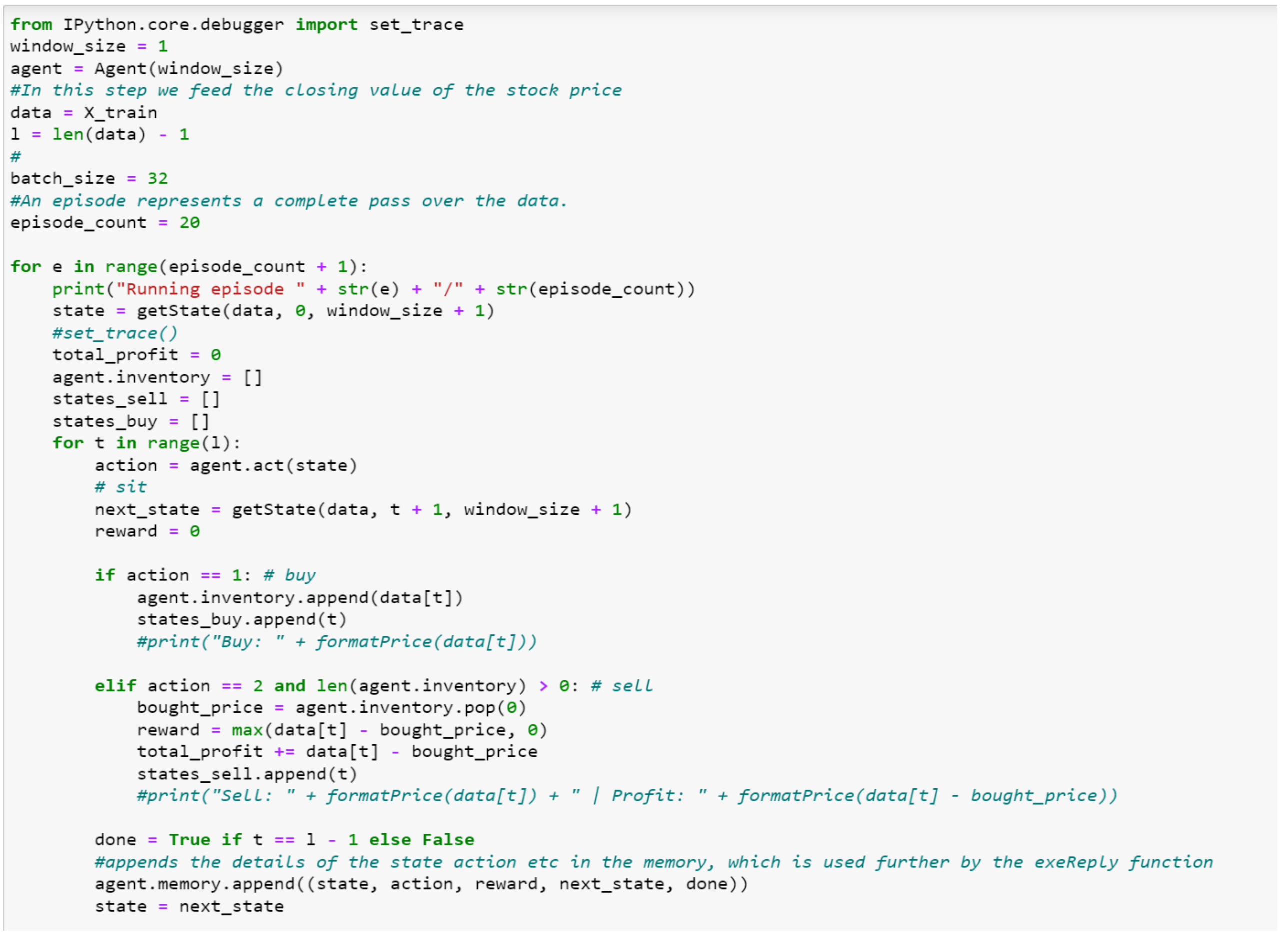
1. Get the current state using the helper function get state. It returns a vector of states, where the length of the vector is defined by windows size and the values of the states are between zero and one.
2. Get the action for the given state using the act function of the agent class.
3. Get the reward for the given action. The mapping of the action and reward is described in the problem definition section of this case study.

4. Get the next state using the get State function. The detail of the next state is fur- ther used in the Bellman equation for updating the Q-function
5. The details of the state, next state, action, etc., are saved in the memory of the agent object, which is used further by the exeReply function. A sample mini- batch is as follows:

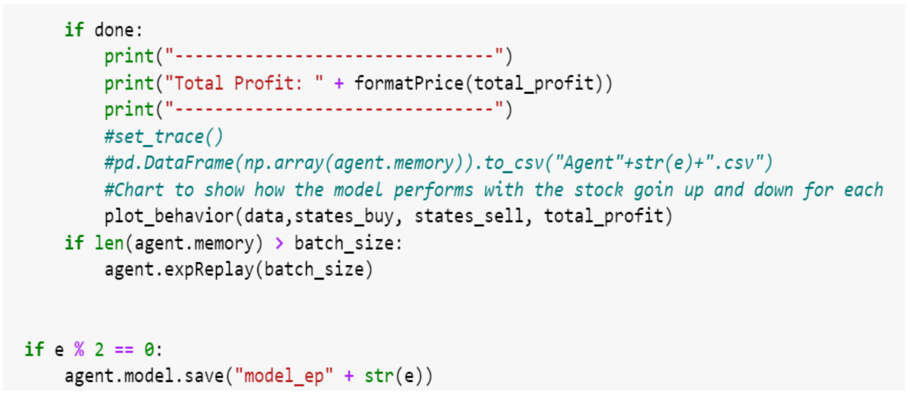


State	action	reward	next_state	done
0.5000	2	0	1.0000	False
1.0000	0	0	0.0000	False
0.0000	1	0	0.0000	False
0.0000	0	0	0.0766	False
0.0766	1	0	0.9929	False
0.9929	2	0	1.0000	False
1.0000	2	17.410	1.0000	False
1.0000	2	0	0.0003	False
0.0003	2	0	0.9997	False
0.9997	1	0	0.9437	False

6. Check if the batch is complete. The size of a batch is defined by the batch size variable. If the batch is complete, then we move to the Replay buffer function and update the Q-function by minimizing the MSE between the Q-predicted and the Q-target. If not, then we move to the next time step.



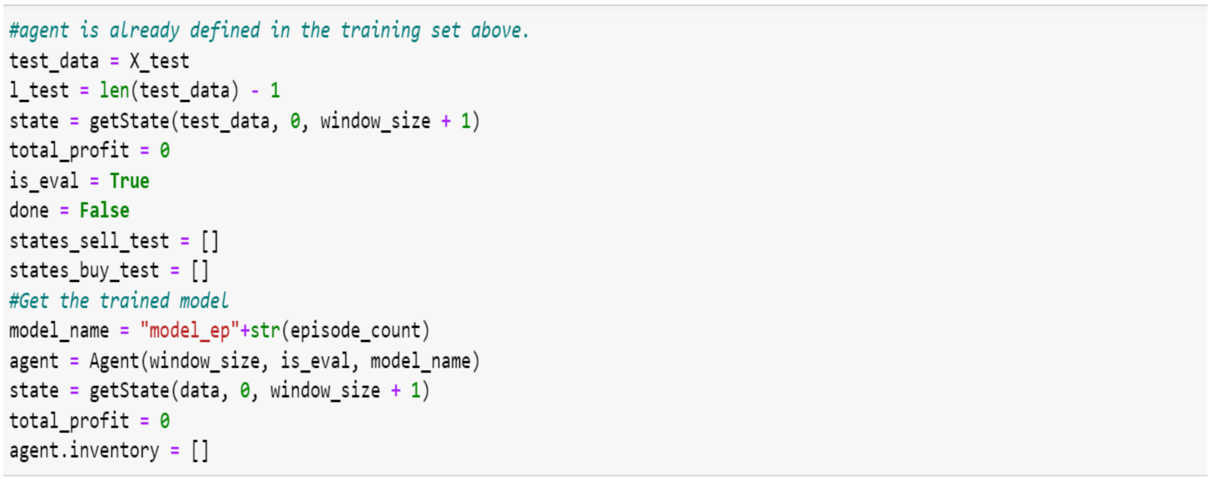
```python
from IPython.core.debugger import set_trace
window_size = 1
agent = Agent(window_size)
#In this step we feed the closing value of the stock price
data = X_train
l = len(data) - 1
#
batch_size = 32
#An episode represents a complete pass over the data.
episode_count = 20

for e in range(episode_count + 1):
    print("Running episode " + str(e) + "/" + str(episode_count))
    state = getState(data, 0, window_size + 1)
    #set_trace()
    total_profit = 0
    agent.inventory = []
    states_sell = []
    states_buy = []
    for t in range(l):
        action = agent.act(state)
        # sit
        next_state = getState(data, t + 1, window_size + 1)
        reward = 0

        if action == 1: # buy
            agent.inventory.append(data[t])
            states_buy.append(t)
            #print("Buy: " + formatPrice(data[t]))

        elif action == 2 and len(agent.inventory) > 0: # sell
            bought_price = agent.inventory.pop(0)
            reward = max(data[t] - bought_price, 0)
            total_profit += data[t] - bought_price
            states_sell.append(t)
            #print("Sell: " + formatPrice(data[t]) + " | Profit: " + formatPrice(data[t] - bought_price))

        done = True if t == l - 1 else False
        #appends the details of the state action etc in the memory, which is used further by the exeReply function
        agent.memory.append((state, action, reward, next_state, done))
        state = next_state
```



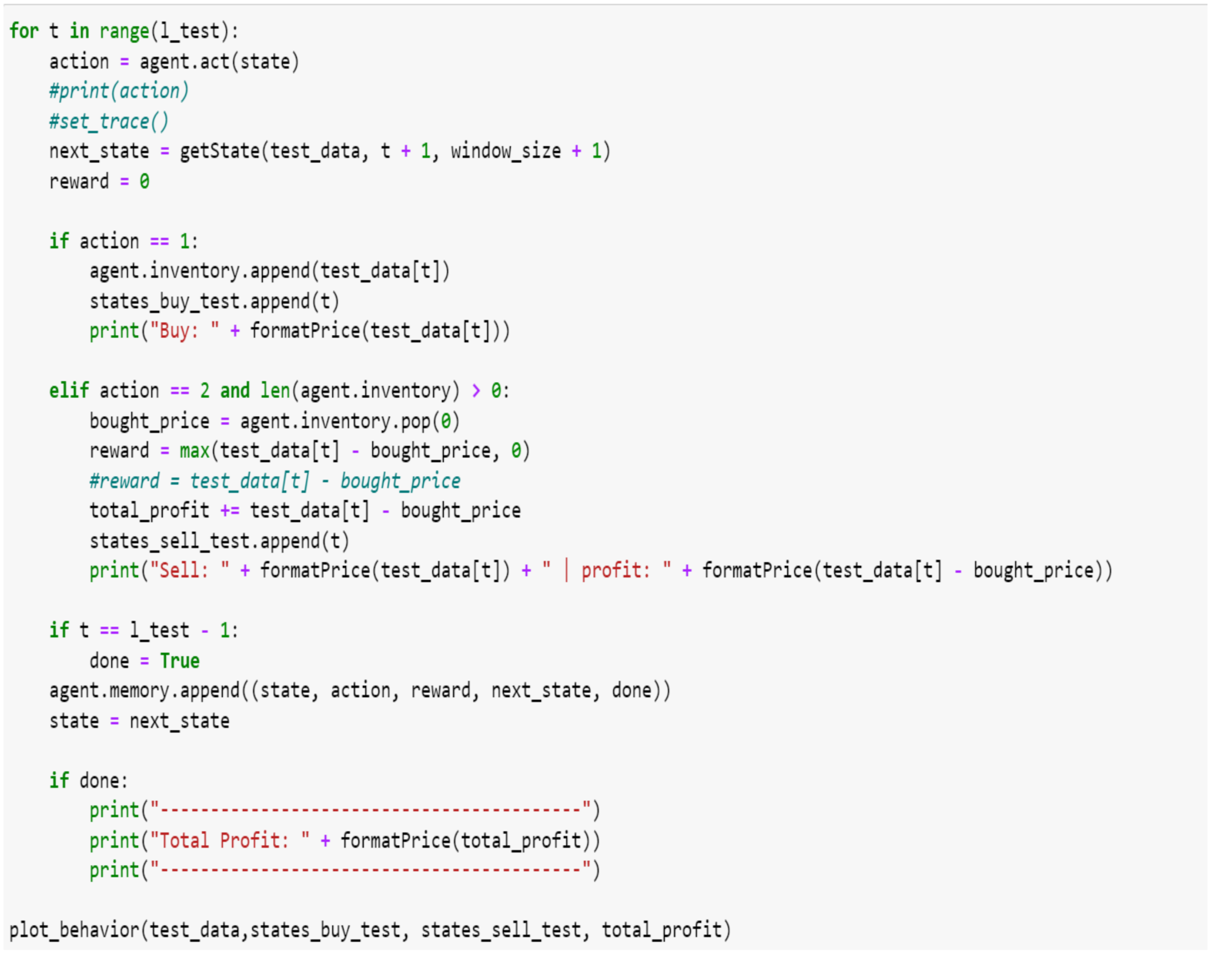
```python
        if done:
            print("--------------------------------")
            print("Total Profit: " + formatPrice(total_profit))
            print("--------------------------------")
            #set_trace()
            #pd.DataFrame(np.array(agent.memory)).to_csv("Agent"+str(e)+".csv")
            #Chart to show how the model performs with the stock goin up and down for each
            plot_behavior(data,states_buy, states_sell, total_profit)
        if len(agent.memory) > batch_size:
            agent.expReplay(batch_size)


    if e % 2 == 0:
        agent.model.save("model_ep" + str(e))
```

Testing the Data

After training the data, it is tested it against the test dataset. Our model resulted in an overall profit. The best thing about the model was that the profits kept improving over time, indicating that it was learning well and taking better actions.



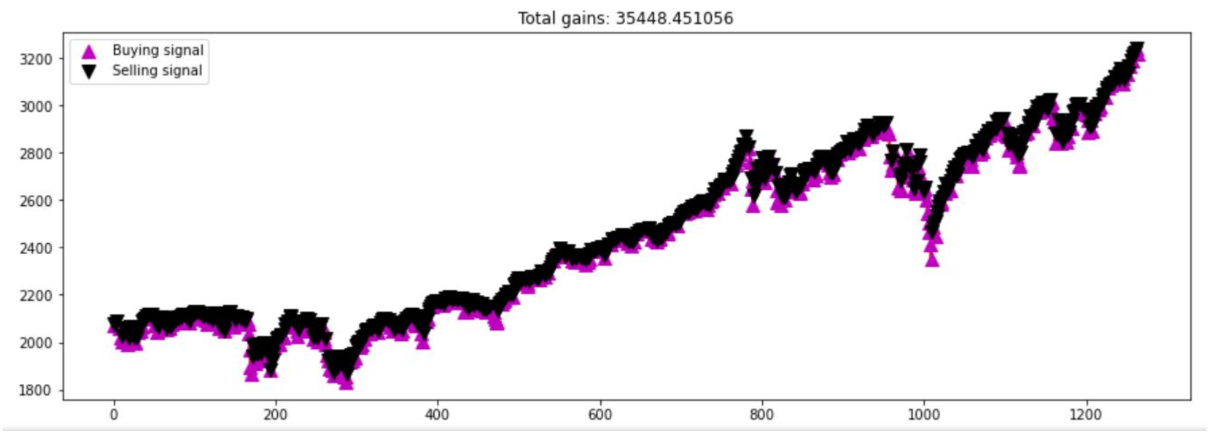
```
#agent is already defined in the training set above.
test_data = X_test
l_test = len(test_data) - 1
state = getState(test_data, 0, window_size + 1)
total_profit = 0
is_eval = True
done = False
states_sell_test = []
states_buy_test = []
#Get the trained model
model_name = "model_ep"+str(episode_count)
agent = Agent(window_size, is_eval, model_name)
state = getState(data, 0, window_size + 1)
total_profit = 0
agent.inventory = []
```



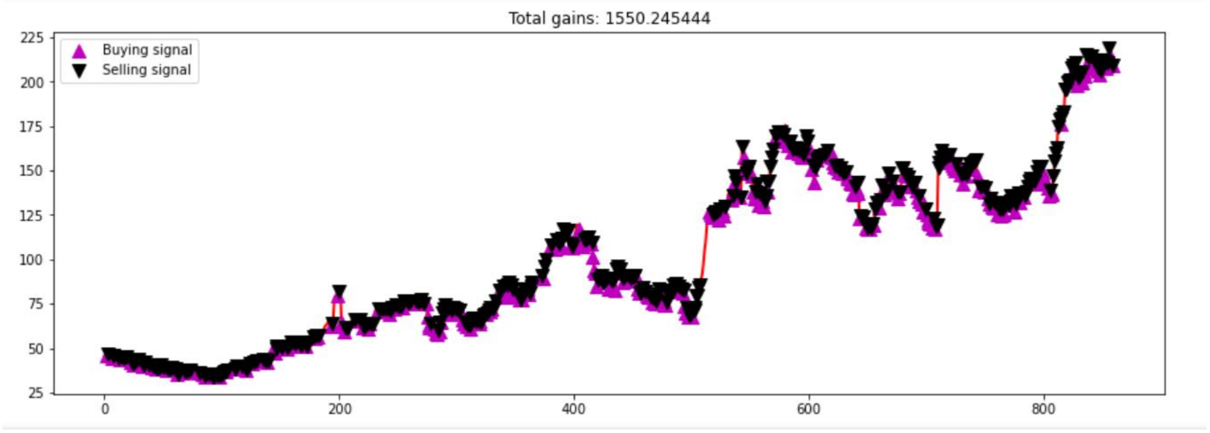
```
for t in range(l_test):
    action = agent.act(state)
    #print(action)
    #set_trace()
    next_state = getState(test_data, t + 1, window_size + 1)
    reward = 0

    if action == 1:
        agent.inventory.append(test_data[t])
        states_buy_test.append(t)
        print("Buy: " + formatPrice(test_data[t]))

    elif action == 2 and len(agent.inventory) > 0:
        bought_price = agent.inventory.pop(0)
        reward = max(test_data[t] - bought_price, 0)
        #reward = test_data[t] - bought_price
        total_profit += test_data[t] - bought_price
        states_sell_test.append(t)
        print("Sell: " + formatPrice(test_data[t]) + " | profit: " + formatPrice(test_data[t] - bought_price))

    if t == l_test - 1:
        done = True
    agent.memory.append((state, action, reward, next_state, done))
    state = next_state

    if done:
        print("------------------------------------------")
        print("Total Profit: " + formatPrice(total_profit))
        print("------------------------------------------")

plot_behavior(test_data,states_buy_test, states_sell_test, total_profit)
```

Results and Findings

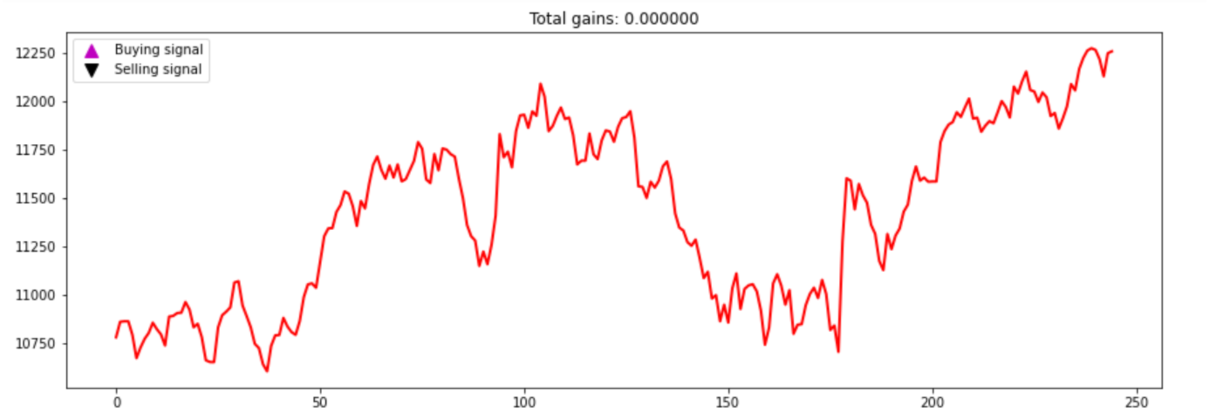
S&P500



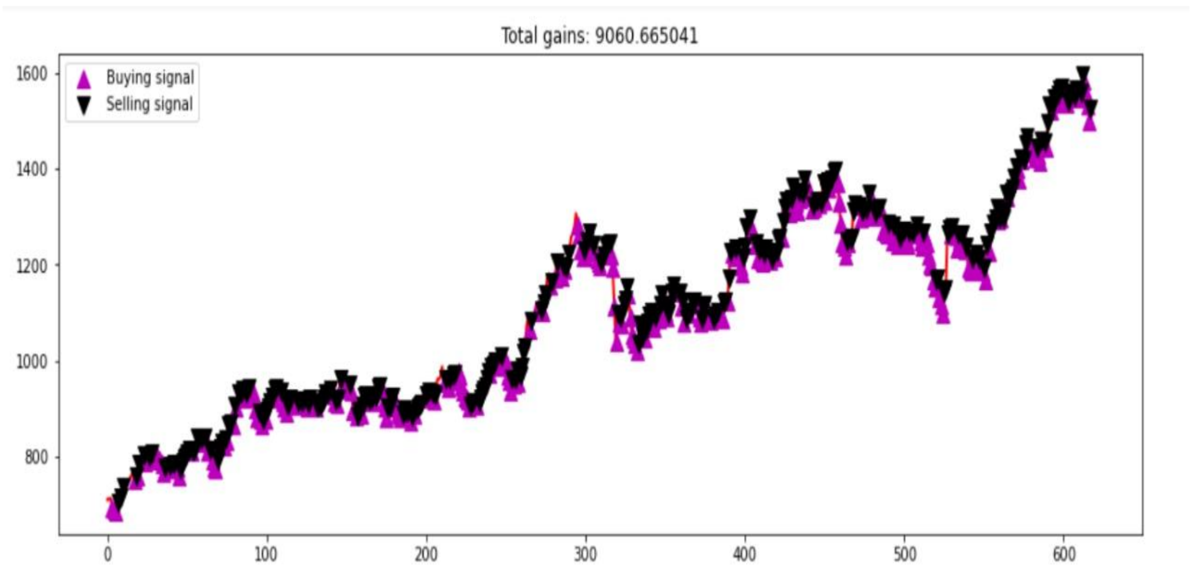
ADANI ENTERPISE LTD.



NIFTY50

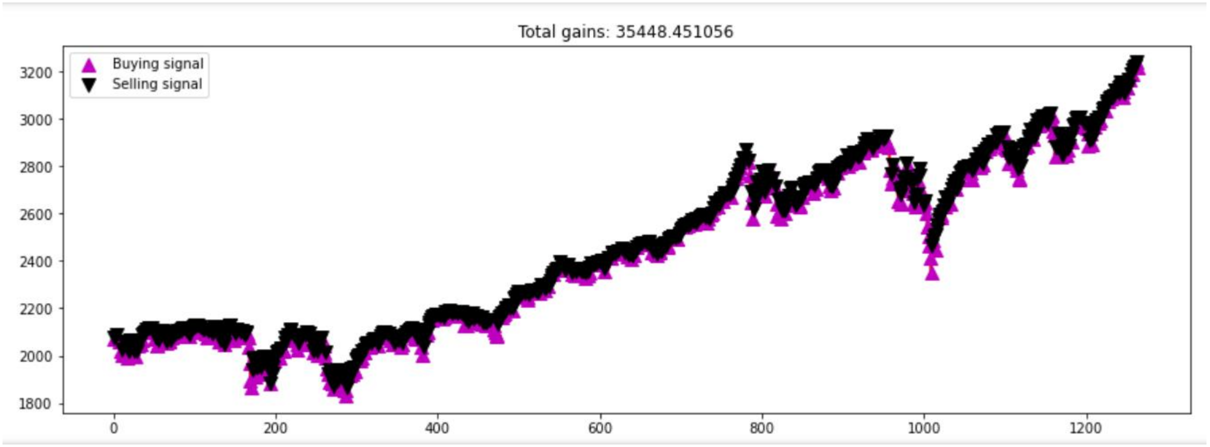


RELIANCE INDUSTRIES LTD.

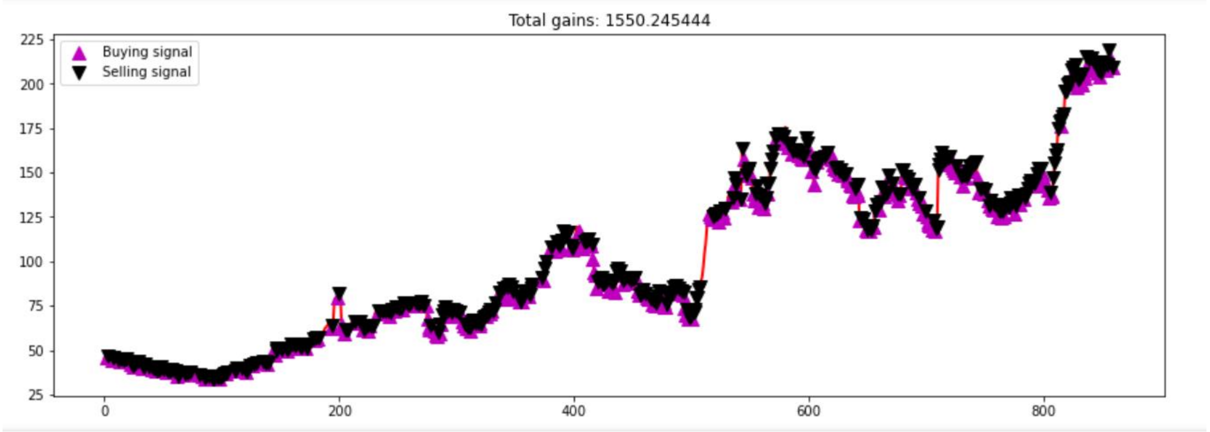


Discussion of Results and Interpretation

We tried out stocks of S&P500, NIFTY50, Adani Enterprise Ltd.,Reliance Industries Ltd. We were able to generate good returns on S&P500 multiple times, and also on Adani Stocks.



Return on S&P500 was around 35000 dollars



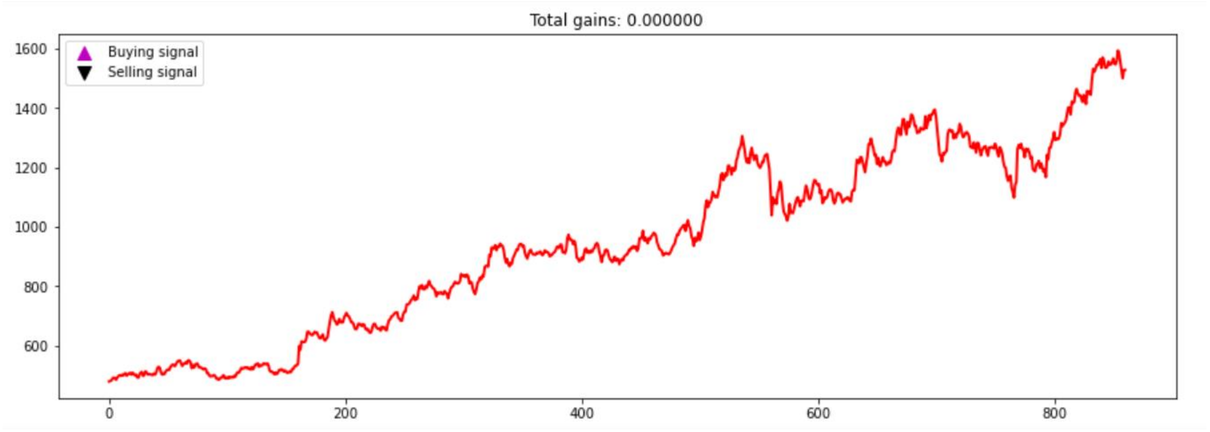
Return on Adani Stocks was about 1550 dollars and on Reliance about 9000 dollars.

These are very good numbers since if an automated agent can play in the stock market and generate profits of the order of thousands of dollars by itself it is a win-win situation for me.

For the S&P500 returns were expected as we in general S&P500 show a clear upwards trend in stock prices so the possibility of making profits is very large.

But Adani in spite of showing a lot of volatility in prices we were able to make reasonably good profits.

This show a proper hyperparameter tunes RL model over long no of episodes will be able to approximate almost any problem and find out an optimal solution no matter how complex the environment may be.



We were unable to generate profits and our algorithm went into total hold out or playing it safe mode for NIFTY50 and Reliance which in general are much more volatile than S&P500 stocks.

But as Adani had shown if we are able to guesstimate some appropriate hyperparameters for the model and use a large no. of episodes and use better computers where we can run for more episodes and larger memory sizes, and run it then there is a high probability that we will be able to reproduce good results for those also.

Conclusion

In this project, we have designed an RL agent using Q and DQ learning and deployed it with several real-world index series and stock price series. The framework has been fine-tuned, optimized, and trained, and its performance is evaluated on the hold-out test dataset. While there were occasional hiccups, overall, the performance of the agent looked quite satisfactory.

Future works and scope

Future work includes further optimizing and fine-tuning the current RL framework. Another direction of work will certainly be looking for other variants of the RL framework like policy gradient-based models and a comparative analysis of those models with the current one will be an interesting case.

References

[1] P. Bajaj, "Reinforcement Learning," [Online]. Available: https://www.geeksforgeeks.org/what-is-reinforcement-learning/.

[2] D. Johnson, "Reinforcement Learning: What is, Algorithms, Types & Examples," [Online]. Available: https://www.guru99.com/reinforcement-learning-tutorial.html.

[3] P. Kadari, "Introduction to Reinforcement Learning for Beginners," [Online]. Available: https://www.analyticsvidhya.com/blog/2021/02/introduction-to-reinforcement-learning-for-beginners/.

[4] Daniel, "Reinforcement Learning," [Online]. Available: https://www.guru99.com/reinforcement-learning-tutorial.html.

[5] K. K. D. S. A. G. I. A. Volodymyr Mnih, "Playing Atari with Deep Reinforcement Learning".

[6] P. S. Matthew Hausknecht, "Deep Recurrent Q-Learning for Partially Observable MDPs," [Online]. Available: https://arxiv.org/abs/1507.06527.

[7] A. G. D. S. Hado van Hasselt, "Deep Reinforcement Learning with Double Q-learning," *Computer Science,* 22 Sep 2015.

[8] B. U. A. P. Y. L. A. R. J. Z. L. J. R. D. W. D. H. Charles Blundell, "Model-Free Episodic Control," *Statistics,* 14 Jun 2016.

[9] [Online]. Available: https://arxiv.org/abs/1712.01815.

[10] Z. T. D. B. Thomas Anthony, "Thinking Fast and Slow with Deep Learning and Tree Search," *Computer Science,* 23 May 2017.

[11] D. K. G. V. W. M. J. B. A. Rupam Mahmood, "Benchmarking Reinforcement Learning Algorithms on Real-World Robots," *Computer Science,* 20 Sep 2018.

[12] E. C. Y. L. K. V. Y. H. Z. K. V. N. X. Y. Z. C. S. F. Jason Gauci, "Horizon: Facebook's Open Source Applied Reinforcement Learning Platform," *Computer Science,* 1 Nov 2018.