Intro To Machine Learning – Assignment-3

UB PERSON NUMBER: 50425014
UB IT NAME: PARAVAMU

TASK: The goal of the assignment is to learn the trends in stock price and perform a series of trades over a period and end with a profit. In each trade you can either buy/sell/hold. You will start with an investment capital of $100,000 and your performance is measured as a percentage of the return on investment.
You will use the Q-Learning algorithm for reinforcement learning to train an agent to learn the trends in stock price and perform a series of trades. You will implement Q-learning algorithm from scratch. The purpose of this assignment is to understand the benefits of using reinforcement learning to solve the real-world problem of stock trading.

STEPS INVOLVED:

1.) IMPORTING NECESSARY LIBRARIES

- ⇨ Importing numpy and pandas for data processing
- ⇨ Importing Gym for developing our environment for the reinforcement problem.
- ⇨ Importing matplotlib to show images
- ⇨ Importing random for randomizing actions

```
[6]
# Imports
import gym
from gym import spaces
import math
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
import random
```

2.) IMPORTING THE DATASET

```
stock_trading_environment = StockTradingEnvironment('./NVDA.csv', number_of_days_to_consider=10)

```

- ⇨ The dataset given to our environment is the historical stock price for NVIDIA for the last 5 years
- ⇨ The features include information such as: Date, Open, High, Low, Close, Adj Close, Volume.
- ⇨ We split the data into train and test data
- ⇨ We use the train set for agent training and we use the test set for agent evaluation

3.) STOCK TRADINDG ENVIRONMENT

- Our stock trading environment has three main functions.
- They are:
    - Reset ()
    - Step ()
    - Render ()

- Reset ()
    - This method resets the environment and returns the initial observation
    - There are 4 types of states or Observations ranging from 0 to 3
    - Based on the observation vector it returns the observation

```
        # Observation vector.
        observation = [price_increase, price_decrease, 0, 1]
    if np.array_equal(observation, [1, 0, 0, 1]):
        observation = 0
    if np.array_equal(observation, [1, 0, 1, 0]):
        observation = 1
    if np.array_equal(observation, [0, 1, 0, 1]):
        observation = 2
    if np.array_equal(observation, [0, 1, 1, 0]):
        observation = 3

    return observation
```

- Step ()
    - The step method takes an action as the input.
    - There are three actions for us to consider
    - Buy, Sell and Hold
    - Based on the action given it returns the next Observation, Reward, If the episode is done or not and some info

```
        return observation, reward, done, info
```

- Render ()
    - The render method is a function which uses matplotlib to plot the total account value over time.

```
plt.figure(figsize=(15, 10))
plt.plot(self.total_account_value_list, color='lightseagreen', linewidth=7)
plt.xlabel('Days', fontsize=32)
plt.ylabel('Total Account Value', fontsize=32)
plt.title('Total Account Value over Time', fontsize=38)
plt.grid()
plt.show()
```

4.) QLEARNING IMPLEMENTATION

- ⇨ QLearning learns the underlying value of the possible actions in a particular observation or state.
- ⇨ Its model free unlike other reinforcement algorithms.
- ⇨ It also takes a discount factor so it can determine the importance of the future reward.

5.) INITIALIZING THE PARAMETERS

- ⇨ We'll pass the stock trading environment as a parameter, so we can use all the methods provided by the environment

```
Agent = QLearning(stock_trading_environment)
```

- ⇨ Number of episodes denotes how many epochs the agent should train for.
- ⇨ We'll need the Qtable initialized with zeros so we can use it to fill the rewards and predict our actions.
- ⇨ We'll need all the parameters for making our epsilon decay algorithm.
- ⇨ Also, we'll need two lists to track the reward dynamics and epsilon decay.
- ⇨ We'll need to also initialize the possible actions, so we can later randomly pick an action to feed to our step function.

```
        self.environment = environment
        #no of epochs it should run for
        self.numOfEpisodes = 500
        #stores the epsilon decay over time
        self.exploreProbabilityArray = []
        #stores the reward over time
        self.rewardsArray = []
        #initializing the Qtable with zeros states x actions
        self.Qtable = np.zeros([4,3])
        self.epsilon = 0.9
        self.epsilonMin = 0.00001  # minimum exploration probability
        self.epsilonDecay = 0.000005  # exponential decay rate for exploration prob
        self.initialState = self.environment.reset()
        self.action = [0,1,2]
        self.decayStep = 0
        self.exploreProbability = 1;
```

6.) TRAINING OUR AGENT

- ⇨ The training runs the number of episodes given above.

```

        for episode in range(self.numOfEpisodes):

```

- ⇨ We get the initial state with the help of environment reset function.

```
            #gets the initial state from the environment
            currentState = self.environment.reset()
```

- ⇨ To denote an episode completion, the step function returns done.
- ⇨ So, till we get a done, we will run one episode.
- ⇨ About the Qtable, the Qtable can be formed with the help of Bellman Equation

```
#we use bellmans equation to form the qtable. with this we can get the actions
self.Qtable[currentState,possibleAction] =  nextReward + (self.epsilon*np.max(self.Qtable[nextState,: ]))
```

- ⇨ The Qtable will help us predict the actions.
- ⇨ But we don't know what action to start from (The action to be passed to the step function to get our next observation and its corresponding reward).
- ⇨ So we randomize the actions initially with the help of random.choice.
- ⇨ With the help of this we complete every episode and complete our Qtable.
- ⇨ But this is not enough. We are exploring 100% to complete the Qtable and this is not very efficient and not the expected result.
- ⇨ To make this efficient we use the Epsilon decay algorithm.
- ⇨ We make the nextState as the current State and run the algorithm till we get a done from the step function and complete all our episodes.
- ⇨ We append the cumulative reward for each episode also append the epsilon decay change over time.

```
self.rewardsArray.append(cumulativeReward)
self.exploreProbabilityArray.append(self.exploreProbability)
```

- ⇨ The training iterations for few episodes are shown below:

```
[43] Agent.train()

Episode Number: 0
Epsilon decay: 0.8959752709752474
Rewards: -2747.9922532831242

Episode Number: 1
Epsilon decay: 0.8919685404212803
Rewards: -2455.783088165446

Episode Number: 2
Epsilon decay: 0.8879797278494629
Rewards: -2506.967828303273

Episode Number: 3
Epsilon decay: 0.8840087531311016
Rewards: -2660.7703487069853

Episode Number: 4
Epsilon decay: 0.8800555364958373
Rewards: -2198.214010777258 4

Episode Number: 5
Epsilon decay: 0.8761199985300405
Rewards: -2135.4274699228527
```

7.) <u>EPSILON DECAY ALGORITHM</u>

- ⇨ The epsilon decay algorithm controls two things:
  - o How much should the algorithm explore.
  - o How much should the algorithm exploit.
- ⇨ Exploitation is possible only if the exploration is done and tracked properly.
- ⇨ The act method contains the epsilon decay algorithm.
- ⇨ It has two choices based on the constraint.

⇨ It checks if the probability which slowly decreases over time is greater than randon.rand value (between 0 to 1).

⇨ We usually start with 1: which means that we ask the algorithm to explore 100% times.

⇨ But this slowly decreases over time and the probability of exploration decreases and we pick the argmax reward from the qtable [nextState row] to get the best possible action.

```
def act(self,decayStep,currentState,episodeNumber):
    self.exploreProbability = self.epsilonMin + (self.epsilon - self.epsilonMin) * np.exp(-self.epsilonDecay * decayStep)
    # print("exploreProbability:",self.exploreProbability)
    randInt = np.random.rand();
    if self.exploreProbability > randInt:
        # print("rand called")
        return random.choice(self.action), self.exploreProbability
    else:
        # print("arg called")
        return np.argmax(self.Qtable[currentState,:]), self.exploreProbability

```

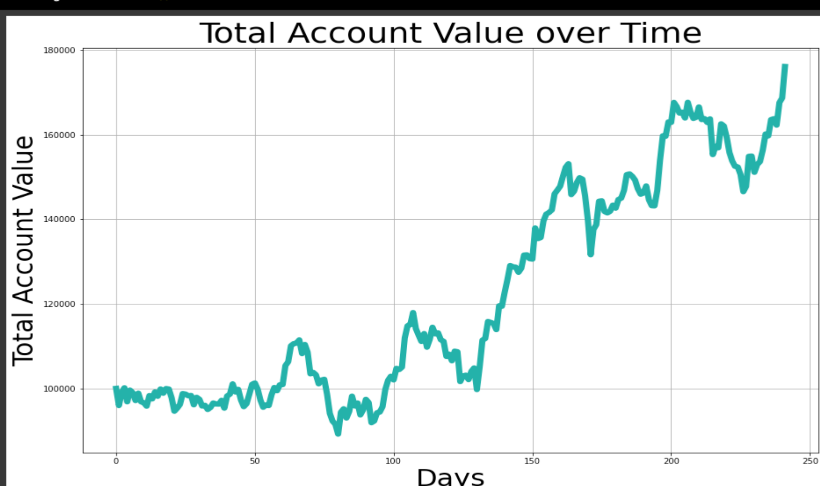
8.) <u>EVALUATING OUR ALGORITHM</u>

⇨ Now that the Qtable is completed, we can pick the best actions, observations and get the maximum total account value over time.

```
def evaluate(self):
    """This method evaluate the trained agent's performance."""
    """"TO DO: Evaluate the trained agent's performance by selecting only the greedy/best action in each state."""
    self.environment.train = False
    #running it over test data
    self.environment.reset()
    for episode in range(1):
      currentState = self.environment.reset()
      done = False
      while not done:
        #picking actions based on qtable
        possibleAction = np.argmax(self.Qtable[currentState,:])
        nextState,nextReward,done,info = self.environment.step(possibleAction)
        currentState = nextState
        #to display the total account value over time
    self.environment.render()
```

⇨ We call the render method to view our results.

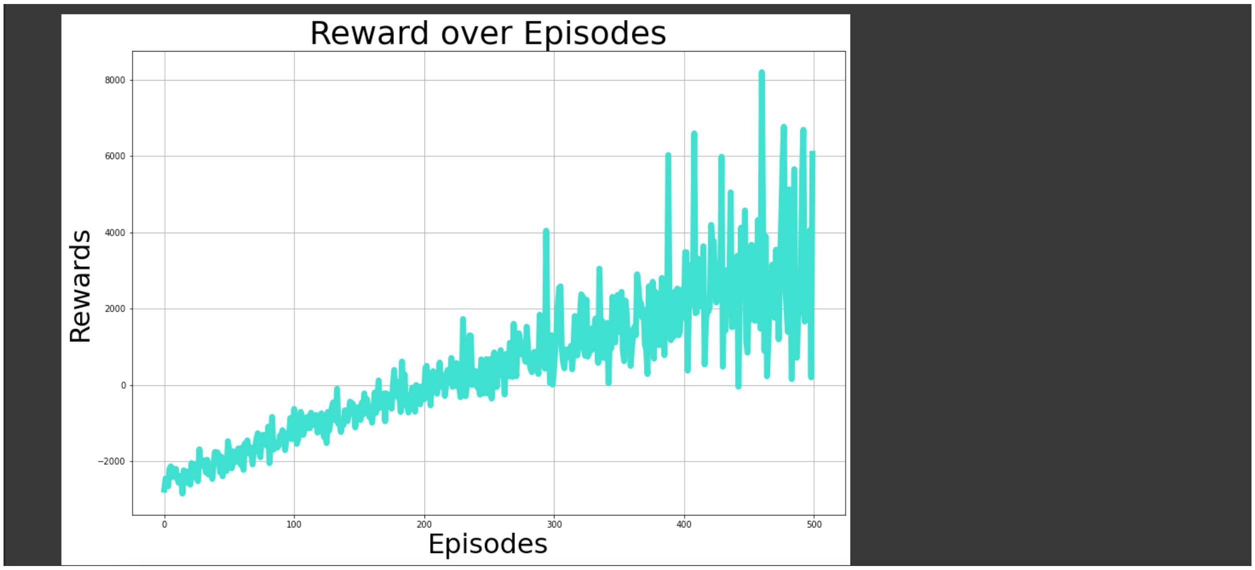
```
Agent.evaluate()
```



9.) <u>PLOTTING OUR RESULTS</u>

⇨ With the help of the plot function, we can visualize the reward dynamics over time

```
plt.figure(figsize=(15, 10))
plt.plot(self.rewardsArray, color='turquoise', linewidth=7)
plt.xlabel('Episodes', fontsize=32)
plt.ylabel('Rewards', fontsize=32)
plt.title('Reward over Episodes', fontsize=38)
plt.grid()
plt.show()
```



⇨ We also plot the epsilon decay over the episodes