Deep Q- learning Agent- Stock Trading

Udacity ML Nanodegree Capstone Project

Abstract

Every day, millions of traders around the world are trying to make money by trading stocks. These days, physical traders are also being replaced by automated trading robots. Algorithmic trading market has experienced significant growth rate and large number of firms are using it. I have tried to build a Deep Q-learning reinforcement agent model to do automated stock trading.

Sachin Kulkarni

Sachin.Kulkarni.iimc@gmail.com

Table of Contents

1. Definition

1.1 Project Overview

Every day, millions of traders around the world are trying to make money by trading stocks. However, it has never been easy to be a good trader. There are many questions a trader needs to answer to maximize his or her profit. When to buy? When to sell? What is the target price? And how long to target?

Moreover, since all of the market variables keep changing, the target price is also adjusted continuously. Supposed that you derive the target price of a stock with a lot of inputs such as interest rate, trading volume, and stock price. All of these variables are real-time variables that changes every second. Hence your target price will change every single second.

Reinforcement learning can be used to create a Trader like Agent and this can be an interesting problem to solve.

Dataset for this will be taken from Kaggle with almost 10+ years of data on various stocks.

1.2 Problem Statement

Since building a trading agent that can choose from all the available stocks is a difficult problem, I have chosen to start with a smaller number of stocks- 2 stocks.

I will use reinforcement learning to train an agent that will buy/sell/hold 2 stocks every-day, once a day, so that long term capital increases.

- Agent has an initial capital of $ X.
- Environment has 2 equities that agent can trade A and B.
- Price of these equities are defined as P_A^t and P_B^t.
- Agent can perform tasks: Buy-A, Sell-A, Do Nothing, Buy-B and Sell-B

To make the problem simpler, there are no transaction costs considered in buying/selling stocks.

1.3 Metrics

A basic metric used in stock trading is Profit and Loss. I will also use the same metric. Agent starts with a capital of $100,000 and stocks of Apple and Amazon with $50,000 each.
At the end of episode, we calculate net PnL as follows:

- Net PnL (Net Profit and Loss) for the whole period = (Portfolio Value) at the end of period - (Portfolio Value) at the start of the period
- Portfolio Value= (Quantity of Apple Stock* Price of Apple Stock) + (Quantity of Amazon Stock* Price of Amazon Stock) + Open Cash

The problem that we are trying to solve here is for a Day Trader and not a long-term Warren Buffet like investment. Hence in this problem:

- We are not looking at fundamental analysis of stock price, but at a daily small variation in stock price.
- We are also not taking the risk that is involved in taking a big position in a stock (so buying/selling in lots of say 100, 1000, 500 stocks at the same time). We are trying to find a daily buy/sell/hold strategy, with buying 1 stock a day out of the two available stocks and then trying to make profit from this.

Therefore, this metric Profit and Loss is the best for this problem.

Past Work Done in this area is given in the reference section. Reinforcement learning will be used to solve above problem. I will be using **Deep Q-learning algorithm** to solve the problem. Detailed algorithm defined in later section.

2. Analysis

2.1 Data Exploration

Dataset is gathered from Kaggle:

https://www.kaggle.com/borismarjanovic/price-volume-data-for-all-us-stocks-etfs/home

The data is presented in CSV format as follows: Date, Open, High, Low, Close, Volume, OpenInt.

I will use following columns from data set: Date, Open, Close and Volume

The data is for a period of more than 15 years (various stocks have various data points).

I will be planning to use 2 equities from the data set- Apple (aapl.us.txt) and Amazon (amzn.us.txt)

Descriptive Statistics

Apple Stock Data

Out[7]:

	Open	High	Low	Close	Volume	OpenInt
count	5155.000000	5155.000000	5155.000000	5155.000000	5.155000e+03	5155.0
mean	35.458280	35.789505	35.097373	35.453219	1.367482e+08	0.0
std	43.144049	43.466721	42.799443	43.146934	1.119160e+08	0.0
min	0.412350	0.423880	0.408530	0.413660	0.000000e+00	0.0
25%	1.517500	1.552100	1.485600	1.521300	6.330228e+07	0.0
50%	12.334000	12.650000	12.013000	12.396000	1.075093e+08	0.0
75%	64.033500	64.795500	63.588500	64.278500	1.764828e+08	0.0
max	175.110000	175.610000	174.270000	175.610000	2.069770e+09	0.0

Amazon Stock Data

Out[8]:

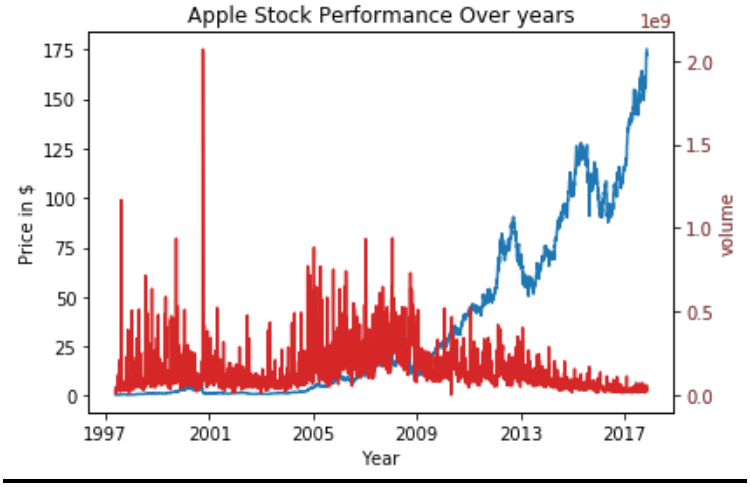
	Open	High	Low	Close	Volume	OpenInt
count	5153.000000	5153.000000	5153.000000	5153.000000	5.153000e+03	5153.0
mean	181.747357	183.880652	179.466684	181.769343	7.837325e+06	0.0
std	239.611052	241.226109	237.638139	239.548391	7.594745e+06	0.0
min	1.410000	1.450000	1.310000	1.400000	0.000000e+00	0.0
25%	35.500000	36.130000	35.000000	35.550000	3.779449e+06	0.0
50%	70.900000	72.750000	69.020000	70.700000	5.902992e+06	0.0
75%	242.850000	245.770000	240.670000	243.880000	8.888949e+06	0.0
max	1126.100000	1135.540000	1124.060000	1132.880000	1.043288e+08	0.0

Observations:

- Data from both stocks does not have same time period, Apple stock has 5155 records while Amazon has 5153 records.
- Open Interest column has no data in both stocks, so this column can be discarded from analysis
- Standard deviation of Amazon stock is more than Apple. Volume of Apple is more than Amazon

2.2 Exploratory Visualization

Graph of Apple stock



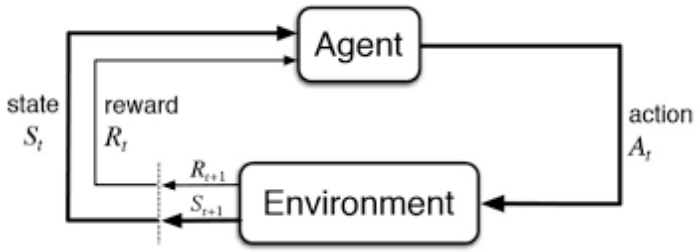
Graph of Amazon stock

Above are graphs of Apple and Amazon stocks from Year 1997 till 2017. From the graph, we can see the following:

1. If agent buys the stocks at the start of 1997 and just holds it for the whole period and sells it at the end of period, it will be lot of profit. Our benchmark model does the same. So, beating this benchmark model in itself will be a challenge for the agent
2. Agent thus, has to be trained to be in the game till end of the period and sell when the prices are high and buy when they get low, so that it beats the benchmark model.
3. There is not much correlation of prices and volume over the whole period. For e.g. during Year 2000, volume was too high, but price was not that high.

2.3 Algorithms and Techniques

In our case, reinforcement learning agent can:



- At every state S_t of environment (i.e. every day, with prices of Apple and Amazon stock given)
- Take 1 of the actions from possible actions A_t (Buy Apple, Buy Amazon, Buy Amazon, sell Amazon, Do nothing).
- And Based on action taken, agent gets a reward R_t
- Next State S_{t+1} gets defined (i.e. new prices for next day)

Question is how does agent know the right action to take. This action should be such that it maximizes not just current reward, but overall reward for the whole episodes, so the reward for not just S_{t+1}, but also for all further states (S_{t+2}, S_{t+3}, S_{t+4}.... S_{t+n}). We use Q-learning to find the right action to take.

Q-Learning:

In Q-learning we define a function Q(s,a) representing the discounted future reward when we perform action a in state s, and continue optimally from that point on.

$$Q(S_t, a_t)=\max_{\pi} R_{t+1}$$

The way to think about Q(s,a) is that it is "the best possible score at the end of game after performing action a in state s". It is called Q-function, because it represents the "quality" of certain action in given state.

This may sound quite a puzzling definition. How can we estimate the score at the end of game, if we know just current state and action, and not the actions and rewards coming after that? We really can't. But as a theoretical construct we can assume existence of such a function.

Suppose you are in state s and pondering whether you should act a or b. You want to select the action, that results in the highest score at the end of game. Once you have the magical Q-function, the answer becomes really simple – pick the action with the highest Q-value!

$$\pi(s)=\text{argmax}\, Q(s,a)$$

Here π represents the policy, the rule how we choose an action in each state.

OK, how do we get that Q-function then? Let's focus on just one transition <s,a,r,s'>. We can express Q-value of state s and action a in terms of Q-value of next state s'.

$$Q(s,a) = r + \gamma max_{a'} Q(s',a')$$

This is called the **Bellman equation**. If you think about it, it is quite logical – maximum future reward for this state and action is the immediate reward plus maximum future reward for the next state.

The main idea in Q-learning is that **we can iteratively approximate the Q-function using the Bellman equation**. In the simplest case the Q-function is implemented as a table, with states as rows and actions as columns. The gist of Q-learning algorithm is as simple as the following:

```
initialize Q[numstates,numactions] arbitrarily
observe initial state s
repeat
    select and carry out an action a
    observe reward r and new state s'
    Q[s,a] = Q[s,a] + α(r + γmaxa' Q[s',a'] - Q[s,a])
    s = s'
until terminated
```

α in the algorithm is a learning rate that controls how much of the difference between previous Q-value and newly proposed Q-value is taken into account. In particular, when α=1, then two Q[s,a]-s cancel and the update is exactly the same as Bellman equation.

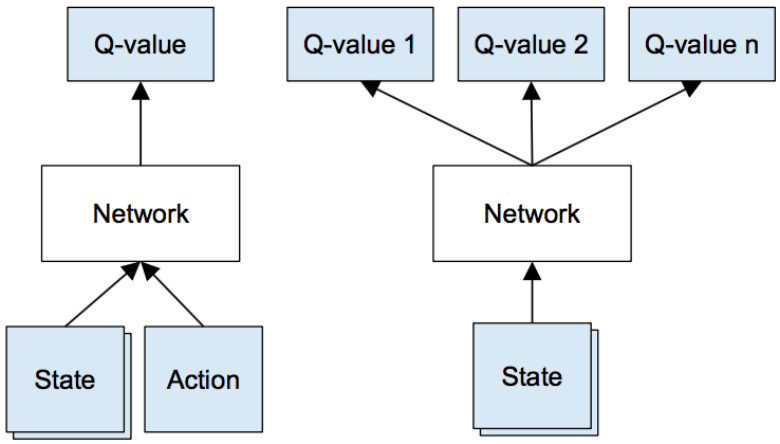
$\max_{a'} Q[s',a']$ that we use to update Q[s,a] is only an estimation and in early stages of learning it may be completely wrong. However the estimations get more and more accurate with every iteration and it has been shown, that if we perform this update enough times, then the Q-function will converge and represent the true Q-value.

Deep Q Network

Neural networks are exceptionally good in coming up with good features for highly structured data.

We could represent our Q-function with a neural network, that takes the state (Stock Prices and other stte variables defined earlier) and action as input and outputs the corresponding Q-value. Alternatively, we could take only state as input and output the Q-value for each possible action. This approach has the advantage, that if we want to perform a Q-value update or pick the action with highest Q-value, we only have to do one forward pass through the network and have all Q-values for all actions immediately available.

Optimal Action to take at any state can be found using deep Q-network.



We will store values of <s,a,r,s′> in memory. Given a transition <s,a,r,s′>, the Q-table is updated by:

1. Do a feedforward pass for the current state s to get predicted Q-values for all actions.
2. Do a feedforward pass for the next state s′ and calculate maximum over all network outputs maxa′Q(s′,a′).

3. Set Q-value target for action a to r+γmaxa′Q(s′,a′) (use the max calculated in step 2). For all other actions, set the Q-value target to the same as originally returned from step 1, making the error 0 for those outputs.
4. Update the weights using backpropagation

Here we also use other concepts like Experience Replay and Exploration vs Exploitation, so that Deep Q network converges quickly. These concepts are explained in methodology section.

2.4 Benchmark Model

Benchmark model is a model where there is no agent, no reinforcement learning. So, the idea is that I would like to compare my agent with a simple benchmark where there is no intelligence- just buy the stocks at the start of period and sell it periodically over the whole period, with 10% sell every time.

Actually, is a good strategy for a stock like Apple and Amazon. If I would have bought these stocks at start of 1997 and hold it till now, I would have been a multi-millionaire ☺

Thus, the benchmark model is a model where:

- At the start of period: We buy stocks Apple and Stock Amazon, with half amount invested in each stock. Open Cash remains the same through-out the period
- Sell 10% of stocks in 10 intervals, thus increasing open cash
- At the end of period calculate portfolio value as follows:
    - Portfolio Value= (Quantity of Apple Stock* Price of Apple Stock) + (Quantity of Amazon Stock* Price of Amazon Stock) + Open Cash

3. Methodology

3. 1 Data Pre-processing:

First step in data pre-processing was to convert "Date" column to date format.

Next step was to make sure that data for both Apple and Amazon are for the same dates. There are many days where either Apple or Amazon data are not present. Where data is not present for a particular day, I had 2 options:

1. I could either populate the data from previous day to the missing day or
2. I could delete the record from other stock.

I have taken the option-2 i.e. of deleting the record from other stock for missing stock date. So, for e.g. if there is no data present for Apple stock on date 12th Jan 1998, then I would delete corresponding day's record from Amazon stock. This way the days in Apple and Amazon stock would be the same set.

Some records from Apple and Amazon Stock data have been deleted to make sure that the data is of the same size. This is not a big number of records- overall 6 records from 5155 records.

Other than this, data is already clean, prices are adjusted for splits etc.

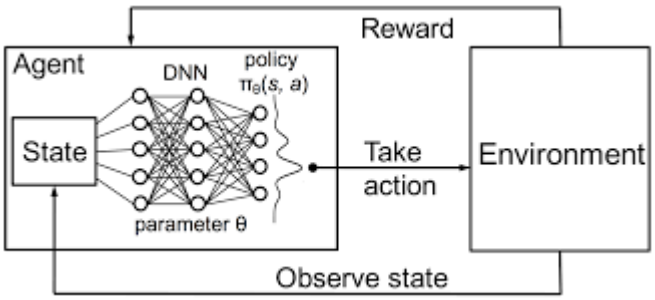
Next step is to divide data into training and test set, in chronological order.

When I decide on the number of episodes to use (say 100), I will divide the dataset in chronological order from a start date D. So:

- Training data will be from Day D to Day D+1000.
- Then I will do a test from day D+1001 to D+1500

3.2 Implementation

As discussed in algorithm section, our solution is to build a Deep Q-Network which has below architecture



3.2.1 State/Environment (state.py)

It was not an easy task to define what State should be. Input data had parameters like Date, Open, High, Low, Close, Volume, Open Interest. Not all of these variables were enough to train the agent well:

- Some of the variables like Volume and Open Interest were not useful for learning

- For Price, we have 3 variables: Open, High and Close Price. I chose Open price, as the agent trades only once in a day, so start of day. Other 2 prices were discarded from state
- We would also need balance of Apple and Amazon stock on every day
- We would also need Open Cash available for agent to buy stocks
- Portfolio value at any point of time

Based on various trial/error, I found that 5-day trailing price of stocks will help agent in making right choice about when it is a good action to buy/sell/do-nothing. This is a typical parameter that even traders use to make a buy/sell decision.

```
class State:
    def __init__(self, data1, data2, Bal_stock1, Bal_stock2, open_cash, timestep):
        self.Stock1Price=data1[timestep] #stock 1 open price
        self.Stock2Price=data2[timestep] #stock 2 open price
        self.Stock1Blnc=Bal_stock1 #stock 1 balance
        self.Stock2Blnc=Bal_stock2 #stock 2 balance
        self.open_cash=open_cash #cash balance
        self.fiveday_stock1=self.five_day_window(data1, timestep)
        self.fiveday_stock2=self.five_day_window(data2, timestep)
        #self.volume1=volume1[timestep]
        #self.volume2=volume2[timestep]
        self.portfolio_value=self.portfolio_value()
```

3.2.2 Agent (Agent.py):

Agent-Initialization:

Agent class has agent initialization method, where parameter like Gamma, epsilon, memory size, action_size and state_size are defined:

```
13 class Agent:
14     def __init__(self, state_size, is_eval=False, model_name=""):
15         self.state_size = state_size # normalized previous days
16         self.action_size = 5 #  buy_1, sell_1,DO Nothing, buy2, sell2
17         self.memory = deque(maxlen=2000)
18         self.inventory1 = []
19         self.inventory2 = []
20         self.model_name = model_name
21         self.is_eval = is_eval
22         self.gamma = 0.95 # gamma is the discount factor. It quantifies how much importance we give for future rewards.
23         self.epsilon = 1.0 # Exploration and Exploitation – Epsilon (ε)
24         self.epsilon_min = 0.01
25         self.epsilon_decay = 0.995
26         self.model = load_model("models/" + model_name) if is_eval else self._model()
27
```

Agent- Act Method:
Agent can take an action based on the state of the environment. The method for this is “Act”.
In this method, I have implemented Exploration vs Exploitation logic. Initially agent will explore more and later will exploit what it has learnt. This is done using epsilon, which decays as the episode goes along.

1. Exploration: Acts randomly from action space. This is based on epsilon value. It starts with value 1 and decays as the agent gets trained everyday
2. Exploitation: Use neural network to find optimal action

```
def act(self, state):
    if not self.is_eval and random.random() <= self.epsilon:
        print("random action")
        return random.randrange(self.action_size)
    print("Calculating using model")
    options = self.model.predict(state)
    print(str(options))
    return np.argmax(options[0])
```

Thus, agent initially chooses random action and as time increases, epsilon decays, and agent will use neural network below for deciding at action to take.

Agent- Model Method: Neural network is modelled with input as state_size and output as action_size. So, it takes:

- Input as State: Stock1Price, Stock2Price, Stock1Balance, Stock2Balance, OpenCash, Stock1_5day_price, Stock2_5day_price, PortfolioValue
- Output: One of the following possible actions: BuyStock1, BuyStock2, DoNothing, SellStock1, SellStock2

```
def _model(self):
    model = Sequential()
    model.add(Dense(units=64, input_dim=self.state_size, activation="relu"))
    model.add(Dense(units=32, activation="relu"))
    model.add(Dense(units=8, activation="relu"))
    model.add(Dense(self.action_size, activation="linear"))
    model.compile(loss="mse", optimizer=Adam(lr=0.001))
    return model
```

Experience-replay:

Our problem is that we have sequential samples from interactions with the environment to our neural network. And it tends to forget the previous experiences as it overwrites with new experiences.

For instance, if we are on Day 400, where stock is going up continuously for last 30 days, it will forget the experience from day 250 to day 300 where stock had a very low valuation because of poor financial performance.

Correlation Problem: If our agent is learning only from experiences of past few days then in short term agent can think that Amazon stock will go down, hence will sell.

But if you look at long term view, stock might be a BUY.

As a consequence, it will be more efficient to make use of previous experience, by learning with it multiple times.

So the solution is to create a “replay buffer.” This stores experience tuples while interacting with the environment, and then we sample a small batch of tuple to feed our neural network.

Experience replay will help us to handle two things:

- Avoid forgetting previous experiences.
- Reduce correlations between experiences

Deep Q-Networks (DQN): Experience Replay

To remove correlations, build data-set from agent's own experience

$$s_t, a_t, r_{t+1}, s_{t+1} \rightarrow \begin{array}{|c|}\hline s_1, a_1, r_2, s_2 \\ \hline s_2, a_2, r_3, s_3 \\ \hline s_3, a_3, r_4, s_4 \\ \hline \ldots \\ \hline s_t, a_t, r_{t+1}, s_{t+1} \\ \hline \end{array} \rightarrow s, a, r, s'$$

Sample experiences from data-set and apply update

$$l = \left(r + \gamma \max_{a'} Q(s', a', \mathbf{w}^-) - Q(s, a, \mathbf{w}) \right)^2$$

To deal with non-stationarity, target parameters $\mathbf{w}^-$ are held fixed

At each time step, we receive a tuple (state, action, reward, new_state). We learn from it (we feed the tuple in our neural network), and then throw this experience.

```
    def expReplay(self, batch_size):
        mini_batch = []
        l = len(self.memory)

        minibatch = random.sample(self.memory, batch_size)

        for state, action, reward, next_state, done in mini_batch:
            target = reward

            if not done:
                target = reward + self.gamma * np.amax(self.model.predict(next_state)[0])

            target_f = self.model.predict(state)
            target_f[0][action] = target
            self.model.fit(state, target_f, epochs=1, verbose=0)

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

In the above function, the line:

target = reward + self.gamma * np.amax(self.model.predict(next_state)[0])

is Bellman's equation below:

$$Q(s, a) \sim r + \gamma * Q'(s', a')$$

s: Agent's current state
a: current optimal action
γ: discount factor
r: Immediate reward from Q->Q'
s': next *optimal* state
a': *optimal* action in the next state

We will use a Deep Neural Network model defined above to approximate the Q(s, a) function through backward propagation. Over multiple iterations, the Q(s, a) function will converge to find the optimal action in every possible state it has explored.

3.2.3 Main Program

Main program is used for train the agent with say 100 Episode. Every 10 episode, we are storing a model that we can use to use during Test Run. This model is stored in Models directory.

We start with data preprocessing, then define a benchmark model and then go to episode run.

In each episode, we run the program for all days in the episode. Every day w new state is defined

- State: Holds the price of Apple and Amazon at any given time = t
  state_class_obj= State(data1_train, data2_train, Bal_stock1, Bal_stock2, open_cash,t)
  state_array_obj=state_class_obj.getState()

- Agent is initialized with available balance of stock and other parameters every day. Agent Acts based on state:
  action = agent.act(state_array_obj)

- There are 2 variables defined, changepercentageStock1 and changepercentageStock2. These variables are used to find out how much is stock price of today more or less than last 5-day stock price. Rewards for various actions are based on the value of this variable.
- Based on the Action, Agent gets a reward:
    - Action 0: this action will buy the Apple stock if Cash is available:
        - If Agent tries to buy Apple stock when it does not have enough Open Cash, then agent gets bankrupt, episode ends and reward is a big negative number, -200000
        - If Open cash is less than 500 and agent tries to sell it, then give negative reward -10000, this is to train agent to not go bankrupt
        - Also, if the stock price is only 2% less than last 5-day trading price, then the reward is -10,000. This is to ensure that agent does not do trading for small gains
        - Otherwise, agent gets reward based on how much is the buying price less than 5-day trailing price
    - Action1: in this action program will sell the Apple stock
        - If Agent has 0 Apple Stocks and it tries to buy it, then agent gets bankrupt, episode ends and reward is big negative number -200000
        - If Apple stocks are less than 10 and agent tries to sell it, then give negative reward -100000, this is to train agent to not go bankrupt
        - Also, if the stock price is only 2% less than last 5-day trading price, then the reward is -10,000. This is to ensure that agent does not do trading for small gains
        - Otherwise, agent gets reward based on how much is the buying price less than 5-day trailing price

    - Action2: Agent will do nothing
        - If agent has less than 2 Apple and Amazon stocks, and it does nothing, then it is good and it gets reward of 10000
        - Else, it gets reward of -100000 for not doing anything. This is to make agent take an action
    - Action3: this action will buy the Amazon stock if Cash is available
        - If Agent tries to buy Amazon stock when it does not have enough Open Cash, then agent gets bankrupt, episode ends and reward is big negative number -200000
        - If Open cash is less than 500o and agent tries to sell it, then give negative reward -10000, this is to train agent to not go bankrupt
        - Also, if the stock price is only 2% less than last 5-day trading price, then the reward is -10,000. This is to ensure that agent does not do trading for small gains
        - Otherwise, agent gets reward based on how much is the buying price less than 5-day trailing price
    - Action4: this action will sell the Amazon stock if Cash is available
        - If Agent has 0 Amazon Stocks and it tries to buy it, then agent gets bankrupt, episode ends and reward is big negative number -200000
        - If Apple stocks are less than 10 and agent tries to sell it, then give negative reward -10000, this is to train agent to not go bankrupt
        - Also, if the stock price is only 2% less than last 5-day trading price, then the reward is -10,000. This is to ensure that agent does not do trading for small gains
        - Otherwise, agent gets reward based on how much is the buying price less than 5-day trailing price

Reward is really important to take the next action, so agent can learn very well and take right decision.

After every day run, the state

```
agent.memory.append((state_array_obj, action, reward, next_state_array_obj, done))
```

Experience replay will take records from this memory and use it in the neural network.

After the agent is trained in episodes and models are saved in Models directory, we will run the Test Run.

In the test run, model is pre-loaded from the Models directory and then used to find out optimal action for the agent.

3.3 Challenges Faced

There were lots of challenges faced during developing this project. I will try to list down these in short:

1. Selecting 2 stocks for the algorithm: Here it is important that the 2 stocks chosen were of similar companies, preferably in the same industry, so same forces determine their price changes. Initially I had taken 2 un-correlated stocks and training the agent with them was difficult
2. Defining State: Since we are talking here about a Markov Decision Process problem, the next state is dependent only on the current state and not all the data from the past. But typically, stock prices change, over a period of time, so past stock prices (so not just today, but also past days) do have influence on what the price of stock will be tomorrow. So, in state I have included 5-day price. This could have been more days in the past, but based on training agent, I found that 5-days price is a good proxy for past prices
3. Defining Actions: Actions that agent could take could be many. Right now, I have only 5 actions (as defined earlier). I also tried actions like buyStock1_sellStock 2, buyStock2_sellStock 1 at the same time, but this did not lead to much increase in profit for the agent. So, then these actions were discarded.
4. Defining rewards: There was lot of fine tuning needed in defining the rewards. When I tried a scenario where agent had less open-cash, the agent went bankrupt, as it tried to buy stocks when prices went lower. So, then I had to modify reward function to include scenario where agent would be rewarded negatively, if it tries to buy when it does not have cash, or when it tries to sell, when it does not have enough stocks.
5. Converting State to vector: State was input parameter for Deep Q neural network. Initially I was passing state object (as defined in State class), but it was leading to issues in neural network with input dimensions. So, I created a getState method that returns a vector for State.

3.4 Refinement

Parameters that were tuned are:

- Exploration vs Exploitation parameter- Epsilon and its decay rate
- State definition: There are various values that can go in state and I will try with various inputs like 5 days stock price, volume etc.
- Memory size
- Action Size: What actions can agent do? I decided to use 5 actions Buy1, Sell1, Buy2, Sell2, Do nothing. I also evaluated model using other actions like Buy1Sell2 and Buy2Sell1, but found that model performance did not change much with it
- Reward for various actions: Reward for not going bankrupt and to not go near bankruptcy were defined.
- Neural network architecture:
    - Number of layers
    - Layer types (relu, Linear)
    - Layer parameters

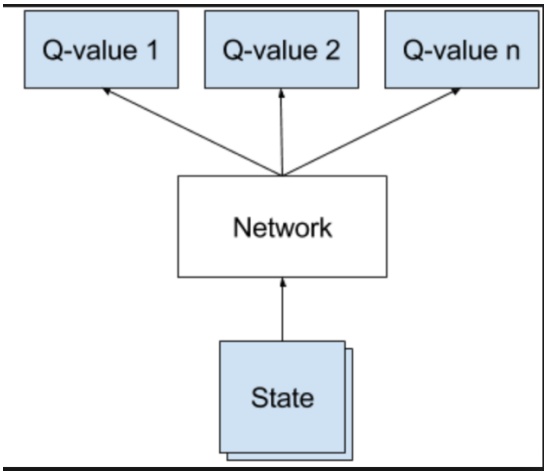
4. Result

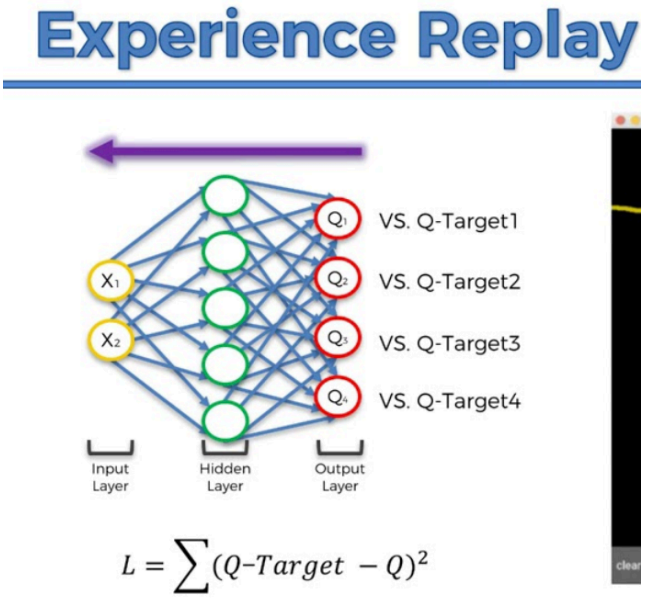
4.1 Model Evaluation and Validation

Agent was trained over 100 episodes and the corresponding models ep10, ep20…ep100 were stored in model's directory.

In the Training section, we load these models and test how good agent has learned.

Final Architecture of the model is, a deep Q-network with following layers in neural network, Experience replay is used to help the neural network to converge better.

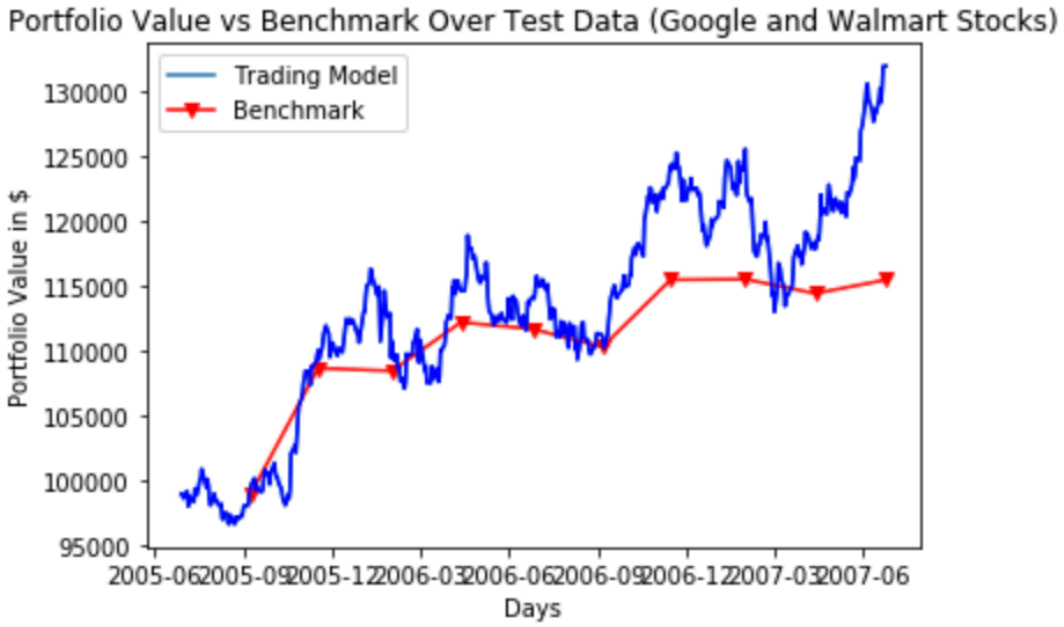




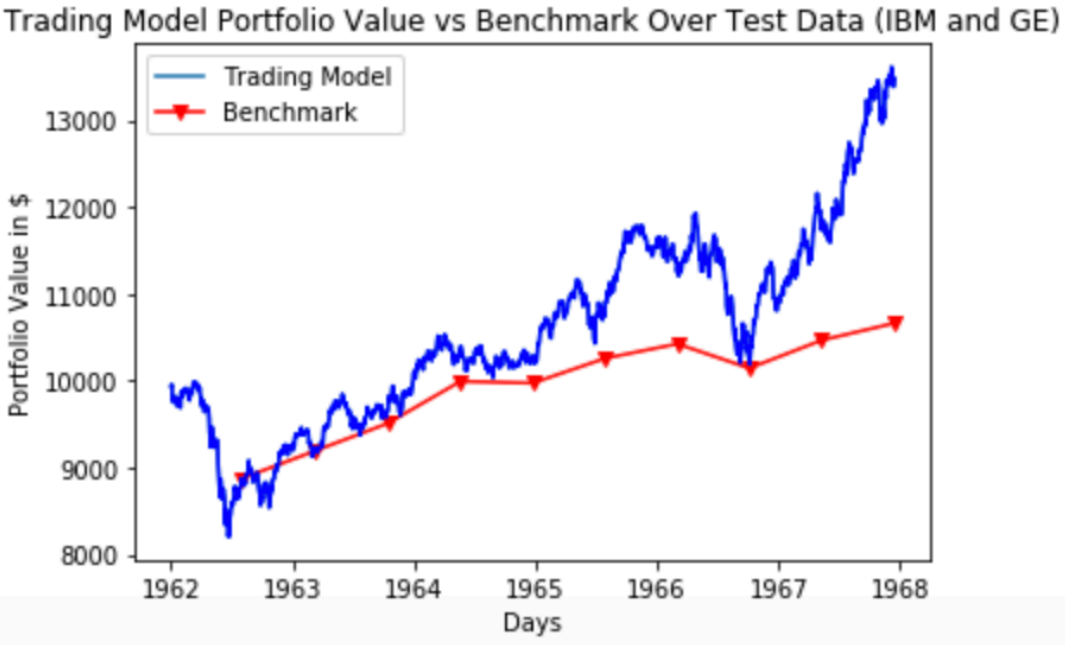
Input to Neural network is states and the output is the Q-Values. Neural network has 3 layers with 64, 32 and 8 units. Output layer is the Q layer with units=action size (i.e. 5)

```
def _model(self):
    model = Sequential()
    model.add(Dense(units=64, input_dim=self.state_size, activation="relu"))
    model.add(Dense(units=32, activation="relu"))
    model.add(Dense(units=8, activation="relu"))
    model.add(Dense(self.action_size, activation="linear"))
    model.compile(loss="mse", optimizer=Adam(lr=0.0001))
    return model
```

The model is robust. It was tested with unseen data, using the file Testing_5_Jan Google n Walmart.ipynb. In this file, the model generated using Amazon and Apple data was pre-loaded to test the robustness with another pair of stock: Google and Walmart. The result of the model was quite good as compared to Benchmark model:



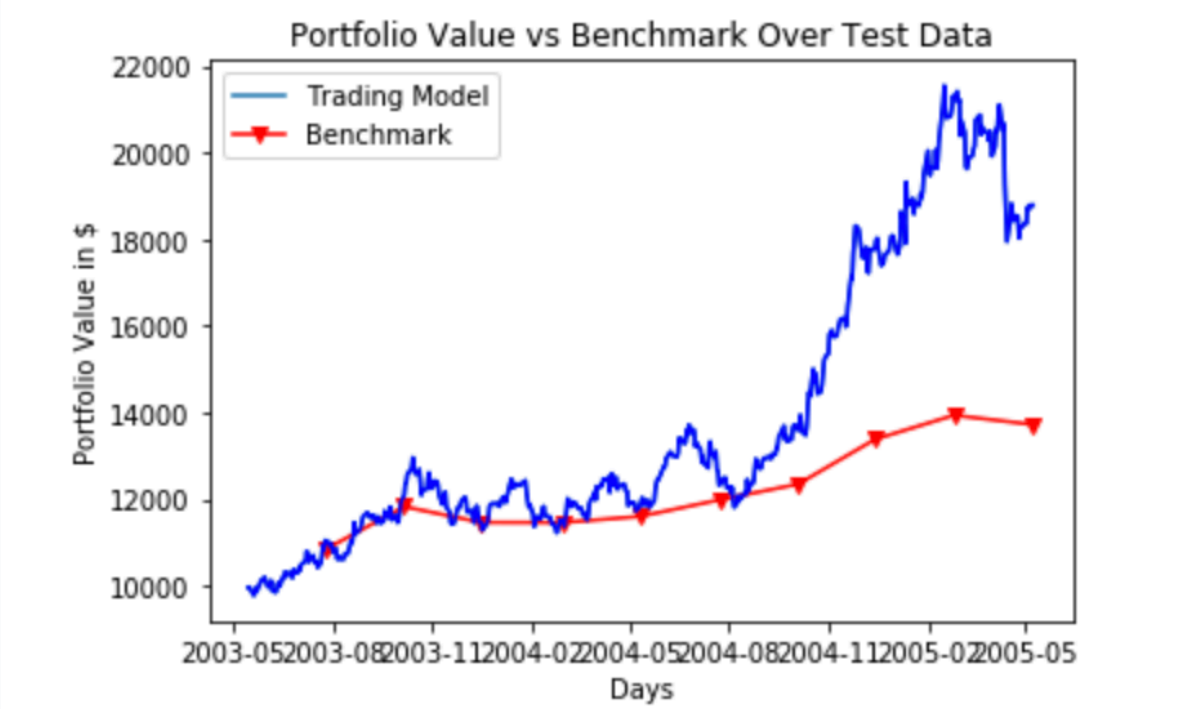
Same test was done with other pair of stocks- IBM and GE using file Testing_5_Jan IBM n GE.ipynb.



In both of above unseen data, model performed at least **30% better than benchmark model**.

4.2 Justification

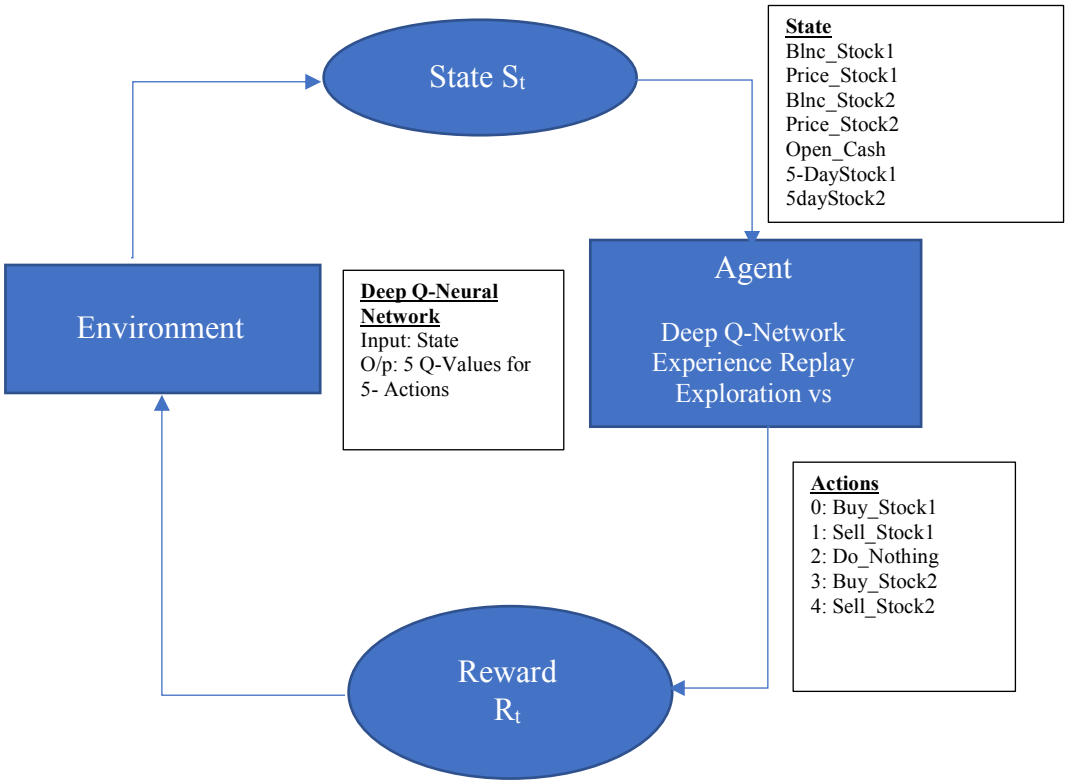
We compared performance of agent w.r.t. benchmark portfolio. Agent has produced more profits than benchmark portfolio, for both training run and actual test run.



Performance of agent w.r.t. benchmark portfolio is almost 30% better. This performance is not by chance, the same performance is also seen in case of unseen data (as shown in previous section). Over there as well, the agent performed 30% more than the benchmark model. This justifies that the model is robust and gives good performance.

5. Conclusion

5.1 End-to-end problem solution and Visualization



Below is step wise process of how I went about solving the problem:

1. Decide on the problem statement: Restrict the problem to reinforcement learning agent for 2 stocks with set of actions that would be performed daily, so a daily trading stock agent.
2. Get the data from the right site: Kaggle
3. Pre-process the data, analyze important features from the data and visualize data
4. Analyze various algorithms that could be used to solve the problem: Actor Critic models vs Deep Q- Network. After some initial analysis and reading research papers on the topic, I found that Deep Q- network is better suited for solving the problem
5. Define State Parameters: It was important to have state defined as a Markov Decision process, so that next state depends only on current state and not from the states in the past. So, here I included last 5-day price of stock, to take care of past prices.
6. Define Agent: Here I defined various parameters for the agent.
    a. Agent can do various actions (5 actions)
    b. In Agent class, a Deep Q-neural network model was defined. This model takes State as Input and gives Output as Q-Values (same number as actions, so 5 output nodes)
    c. Agent uses Exploration vs Exploitation to find the right action. Initially agent will do more exploration (choose random actions) and later, agent will use model above to find the right action
    d. To fine-tune the neural network, agent uses Experience replay to train agent from the past actions
7. Defining rewards for each of the actions: Here I used %change from last 5 days to decide on the right reward. Also, rewards were designed to ensure agent does not go bankrupt

8. Later parameters of Deep Q-network were fine tuned
9. Benchmark model was defined that matches a typical trader, selling stocks in 10 equally spaced intervals, over the period
10. Agent was trained for multiple episodes over a period X, and the model generated was saved
11. Agent was tested over period X+Y, using the model generated during trading period and the performance of agent was compared to benchmark portfolio.
12. Agent consistently performed more than benchmark model, also on unseen data

I found the problem quite interesting and can be scaled to do many more things:

1. Build a Capital Asset Pricing Model (CAPM) optimal portfolio based on this. So instead of 2 stocks, we can build a complete portfolio
2. We can include risk/volatility into the model, so base reward not just on the Profit and Loss, but on Sharpe ratio
3. Increase action space to more actions like buying/selling in lots of 10, 100, 200, 300, 500 stocks

Overall, it was a good learning experience to develop a working model. Also, reinforcement learning was one of the difficult topics in this Nanodegree and doing this project helped me to learn it well.

5.2 Improvement

Improvements that can be done in the model are as follows:

1. Train with more data. Current data is not enough to train. One of the approaches could be the same as what Gordon Ritter used in his project. It will be to simulate data with mean price for a particular period
2. Some research scientists have used Double DQN instead and this might lead to better performance.

5.3 References

1. MACHINE LEARNING FOR TRADING: GORDON RITTER: https://cims.nyu.edu/~ritter/ritter2017machine.pdf
2. Financial Trading as a Game: A Deep Reinforcement Learning Approach: Huang, Chien-Yi https://arxiv.org/pdf/1807.02787.pdf
3. Convergence of Q-learning: a simple proof: Francisco S. Melo: http://users.isr.ist.utl.pt/~mtjspaan/readingGroup/ProofQlearning.pdf
4. https://medium.com/@chinmaya.mishra1/deep-dive-in-to-reinforcement-learning-10fa30b418f9
5. David Silver's lectures about deep reinforcement learning