Automated Trading using Reinforcement Learning (Double Deep Q Network)

S. A. M. Saddam Chowdhury

Capstone Project for Machine Learning Engineer Nanodegree (Udacity)

Submitted: 1 January 2019

1 DEFINITION

1.1 Project Overview

An automated trading system that makes investment decisions in a fully automatized way and generates constant profit from the financial market is lucrative for every market practitioner. Reinforcement Learning (RL) is a branch of Machine Learning (ML) that allows to find an optimal strategy for a sequential decision problem by directly interacting with the environment. In this project, I develop an automated trading algorithm based on RL. The project was inspired by Udacity's "Machine Learning for Trading" course [1].

1.2 Problem Statement

The RL agent is trained to interact with the market to achieve some intrinsic goal (e.g., 20% expected return). Common interactions are to observe the latest and historical financial data or submit new orders to the exchange, etc. Here I construct a Double Deep Q Network (combination of a Double Q-learning system with a deep neural network) aka DDQN that operates on multiple stocks and perform trading tasks by taking one of three possible actions (buy, sell and hold) at the close of each trading day.

In Q-learning, the action-value function, Q, is calculated by adding the immediate expected reward to the best possible outcome of the onward states. In any state, the algorithm exploits the action that has the best expected Q-value (see [2] for details). The Q-learning agent interacts with an environment through a sequence of observations, actions and rewards. The goal of the agent is to select actions in a fashion that maximizes cumulative future reward.

In stock trading, an observation (state) is a combination of portfolio and market inputs and represented as an array of [# of stocks owned, current stock prices, cash in hand]. In any state, based on the current and historical market data, the agent chooses an action which changes the portfolio structure as well as some market inputs. As a result, a new state is obtained .

The goal of the agent is to learn, by trial-and-error, which action maximizes its long-run rewards. In the beginning of the training process, the expected Q-value of all possible actions are equal. After that, for every action, the algorithm receives the reward signal from the environment and update the Q-values. However, since the environment evolves stochastically and may be influenced by the actions chosen, the agent balances its desire to obtain a large immediate reward by acting greedily and the opportunities that are available in the future (see also [3, 4]).

One problem with standard Q-learning is that the *max* operator (see Sec. 2.2) uses the same values both to select and to evaluate an action. This makes it more likely to select overestimated values, resulting in overoptimistic value estimates. To prevent this, the concept of Double Q-learning has been introduced [5, 6]. In Double Q-learning, the selection is decoupled from the evaluation. Since the input space can be massively large, I use a Deep Neural Network [7] to approximate the Q(s, a) function through backward propagation where ‘s’ is the state and ‘a’ is the optimal action associated with that state to maximize its returns over the lifetime of the episode. Over multiple iterations, the Q(s, a) function converges to find the optimal action in every possible state it has explored.

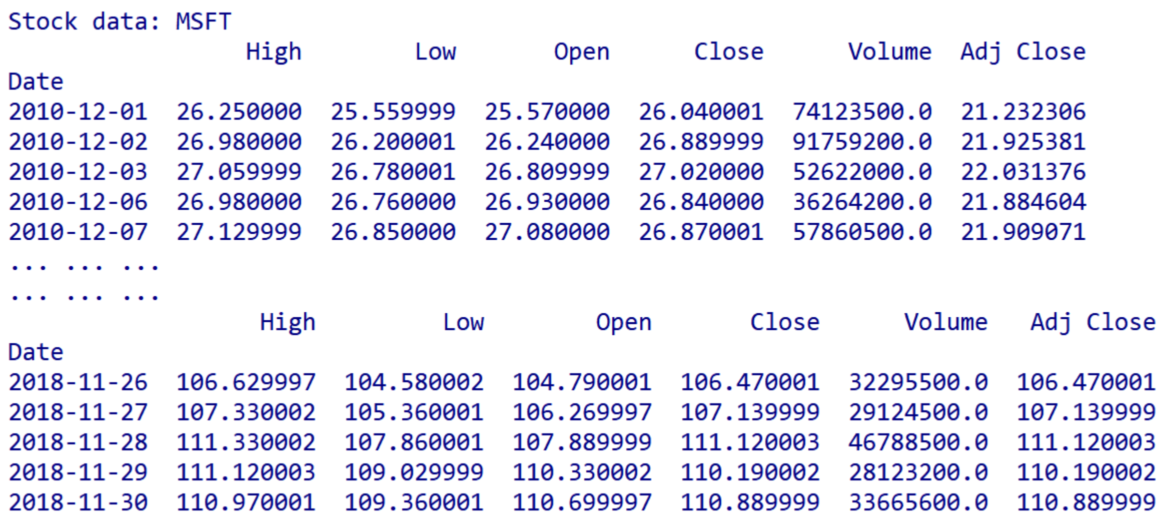
1.3 Metrics

A reward is a scalar feedback signal that indicates how well an agent is doing at a specific step. The agent’s job is to maximize the cumulative reward. There are several possible reward functions we can pick from. An obvious one is the instantaneous reward of daily profit. The net profit from that trade can be positive or negative. That’s the reward signal. As the agent maximizes the total cumulative reward, it learns to trade profitably. The reward function is calculated as

$$R = v_t - v_{t-1},$$

where v_t = portfolio value (cash_in_hand + stock_owned * stock_price) at the t-th day.

I consider an initial investment of $10,000$ for trading on 3 stocks (MSFT, AMZN and IBM). The goal is to maximize profit with expected return of at least 20%.



```
Stock data: MSFT
                 High        Low       Open      Close      Volume  Adj Close
Date
2010-12-01  26.250000  25.559999  25.570000  26.040001  74123500.0  21.232306
2010-12-02  26.980000  26.200001  26.240000  26.889999  91759200.0  21.925381
2010-12-03  27.059999  26.780001  26.809999  27.020000  52622000.0  22.031376
2010-12-06  26.980000  26.760000  26.930000  26.840000  36264200.0  21.884604
2010-12-07  27.129999  26.850000  27.080000  26.870001  57860500.0  21.909071
... ... ...
... ... ...
                  High         Low        Open       Close      Volume   Adj Close
Date
2018-11-26  106.629997  104.580002  104.790001  106.470001  32295500.0  106.470001
2018-11-27  107.330002  105.360001  106.269997  107.139999  29124500.0  107.139999
2018-11-28  111.330002  107.860001  107.889999  111.120003  46788500.0  111.120003
2018-11-29  111.120003  109.029999  110.330002  110.190002  28123200.0  110.190002
2018-11-30  110.970001  109.360001  110.699997  110.889999  33665600.0  110.889999
```

Figure 1: A sample of MSFT stock data

2 ANALYSIS

2.1 Data Exploration and Visualization

The dataset used in this project include 8 years of daily data (ranging from 12/01/2010 to 11/30/2018) of different stocks downloaded from Yahoo finance. This dataset is divided into a training set (12/01/2010 - 11/30/2014) and a test set (12/01/2014 - 11/30/2018). Only the daily 'Adj Closing' price is used in this study, though other features can be easily incorporated into the model. An example of stock pricing information is displayed in Fig. 1.

Description of the data:

- **Open** - The price of the share when the stock market opens in the morning for trading.
- **High** - The highest value of the share for that day over the course of the trading day.
- **Low** - The lowest value of the share for that day over the course of the trading day.
- **Close** - The final price of the share for that day at the close of the trading day.
- **Volume** - The number of share traded for that day.
- **Adj Close** - An adjusted closing price is a stock's closing price on any given day of trading that has been amended to include any distributions and corporate actions (such as stock splits, dividends/distributions and rights offerings) that occurred at any time before the next day's open. The adjusted closing price is often used when examining historical returns or performing a detailed analysis of historical returns.

Before choosing the stocks for this project, I analyzed the stock data for different companies I was interested in investing. The stock data along with their 60 days moving average (60ma) has been plotted in Fig. 2. When building a portfolio, it is also important to take into account the correlation among different stocks (Fig. 3). For two negatively correlated stocks, if the price of one performs worse than usual, the other will likely do better than usual. The gain in one stock is therefore likely to offset the loss in the other. If the stocks are positively correlated, on the other hand, they tend to rise and fall together. Investing on negatively correlated stocks reduces the risk of catastrophic losses in the portfolio. However, it does have drawbacks as well. First, such a portfolio's profit potential is also limited. Since the rise in some stocks will likely mean a drop in others, such a portfolio exhibits mild gains as well as small losses.

For this project, I have chosen Microsoft Corporation (MSFT), IBM (IBM) and Amazon.com, Inc. (AMZN) stocks after analyzing the moving average and correlation data to investigate the portfolio behaviour. In Fig. 4, I have plotted the trading volume along with 60 days moving average of 'Adj Close' for the past 4 years (test data).

2.2 Algorithm and Techniques

The DDQN algorithm has been employed in this project to perform the trading tasks and the goal is to maximize the long-run rewards. Two most import elements of RL algorithm are the agent and environment. The agent interacts with the environment to perform certain tasks and collect rewards as shown in Fig. 5.

- The environment receives action A_t from the agent, and produces state S_{t+1} and a scalar reward R_{t+1}.
- The agent receives state S_{t+1} and a scalar reward R_{t+1} from the environment and moves to the next time step (which is now the current world as the algorithm sees it) where state $S_t \equiv S_{t+1}$ and $R_t \equiv R_{t+1}$. $Q(s, a)$, a function that predicts one of the possible actions A_t than can be taken at current state S_t . In our case Q is a neural network.

This loop continues until certain terminal condition is met.

One problem in the Q-learning/DQN algorithm is that the agent tends to overestimate the Q function value, due to the *max* in the formula used to set targets:

$$Q(s, a) \rightarrow r + \gamma max_a Q(s', a)$$

A solution to this problem was proposed by *Hado van Hasselt (2010) [5]* and termed 'Double Learning'. The idea of Double Q-learning is to reduce overestimations by decomposing the

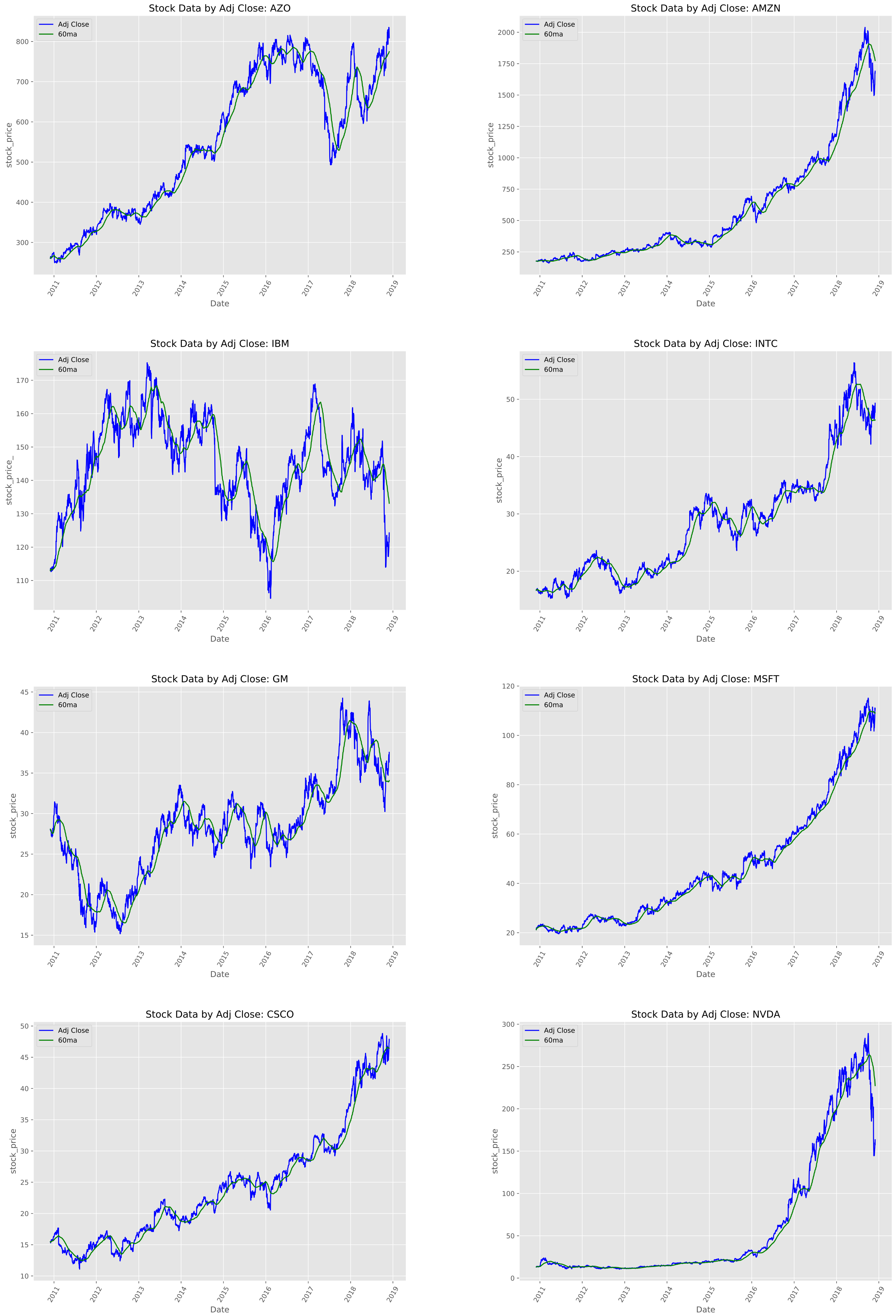


Figure 2: Stock data and 60 days moving average for various stocks

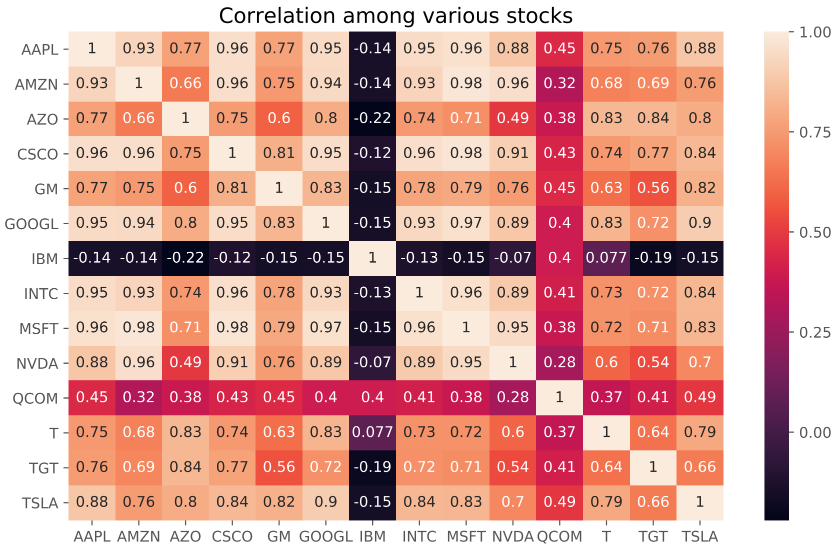


Figure 3: Heatmap of correlation among various stocks

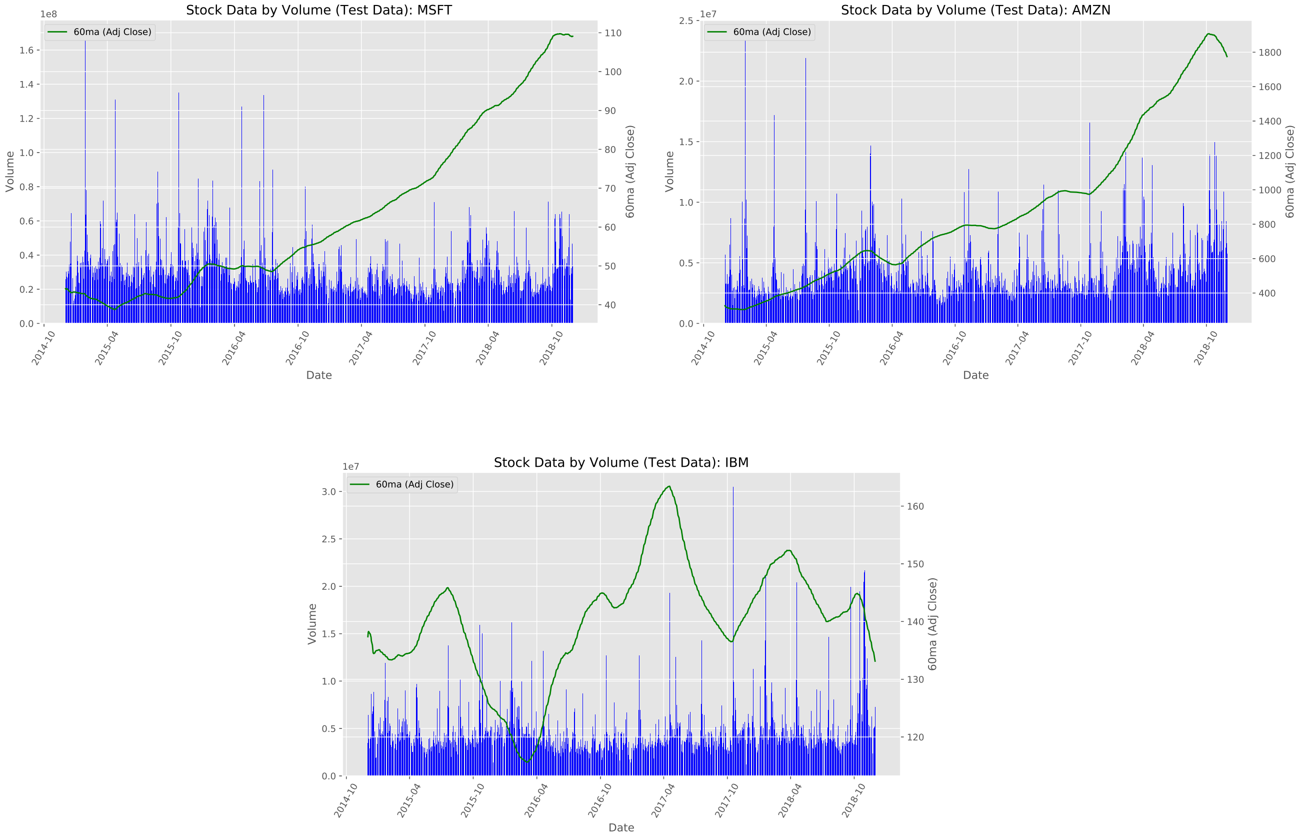


Figure 4: Plots of stock volume data for MSFT, AMZN and IBM

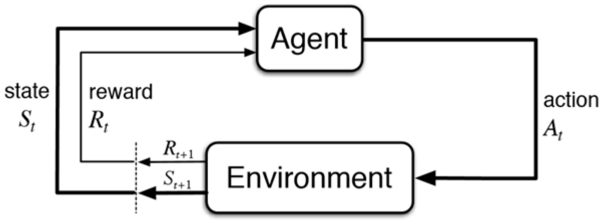


Figure 5: Interaction between the agent and the environment in RL

max operation in the target into action selection and action evaluation. In this new algorithm, two Q functions – Q_1 and Q_2 – are independently learned. One function is then used to determine the maximizing action and second to estimate its value. Either Q_1 or Q_2 is updated randomly with a formula:

$$Q_1(s,a) \rightarrow r + \gamma Q_2(s', argmax_a Q_1(s',a))$$

or,

$$Q_2(s,a) \rightarrow r + \gamma Q_1(s', argmax_a Q_2(s',a))$$

It was proven that by decoupling the maximizing action from its value in this way, one can indeed eliminate the maximization bias.

Environment

The environment class implements the following attributes/methods using the OpenAI/gym interface:

- Init: For initialization of the environment at the beginning of each episode.
- State: At any given point, the state is represented as an array of # of shares owned, current stock prices, cash-in-hand.
- Step: The change in environment after one time step. With each call to this method, the environment returns four attributes:
    1. next_state: the new state as a result of the action performed by the agent at each time step
    2. reward: produces reward associated with the action performed by the Agent in the current state. In our case, the reward is simply the daily profit and loss.
    3. done: whether we have reached the end of the episode.
    4. info: Contains diagnostic information. In our case it contains current portfolio value and profit information.

- Reset: To reset the environment at the beginning of each episode. In our case, it initializes the # of shares owned to zero, resets the stock price to the initial price at the first time step $(2010 - 12 - 1)$ and also resets cash in hand to the initial investment $(10,000)$.

- Trade: At each time step of an episode, this method does the calculations for the trading operation based on the actions (buy, sell or hold) and updates the values for cash-in-hand and # of shares owned for each stock.

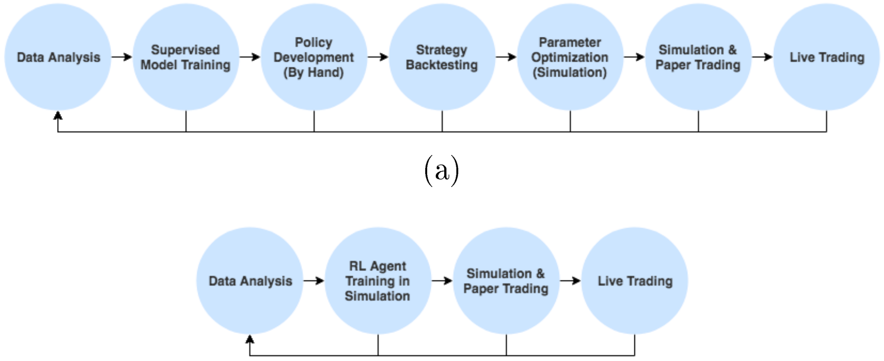
Agent

The agent is an MLP (Multi Layer Perceptron) multi-class classifier neural network taking in $(2n + 1)$ inputs from the environment resulting in 3^n actions subject to maximizing the overall reward in every step. If we possess 3 stocks (a_1, a_2 and a_3) and the price of the stocks are b_1, b_2 and b_3 per share, respectively, and if we have c amount of cash in hand, then the input to the neural net will be array of all these variables, i.e., [b_1, b_2, b_3, a_1, a_2, a_3, c]. After every action, it receives the next observation (state) and the reward associated with its previous action. Since the environment is stochastic in nature, the agent operates through an MDP (Markov Decision Process), i.e., the next action is entirely based on the current state and not on the history of prices/states/actions and it discounts the future reward(s) with a certain measure (gamma). The score is calculated in every step and the cumulative reward per episode is calculated by summing all the individual scores in the lifetime of an episode. The agent's performance over its training is evaluated by the cumulative rewards. Since the input space can be massively large, we use a Deep Neural Network to approximate the $Q(s, a)$ function through backward propagation. Over multiple iterations, $Q(s, a)$ function converges to find the optimal action in every possible state it has explored.

The agent has two main components:

- **Memory:** It contains a list of events in the tuple form of <state, action, reward, next_state, message>. The Agent stores the information through iterations of exploration and exploitation.

- **Brain:** This is the Fully Connected, Feed-Forward Neural Net which is trained on the past experiences (memory). Given the current state as input, it predicts the next optimal action. As the neural net is being trained, the experience is stored in the memory at the end of each episode and hence the network collects more data as it continues to be trained. As a result of that, the Q function converges with more iterations and the agent's performance increases over time until it reaches a saturation point.


Data Analysis
Supervised Model Training
Policy Development (By Hand)
Strategy Backtesting
Parameter Optimization (Simulation)
Simulation & Paper Trading
Live Trading
Data Analysis
RL Agent Training in Simulation
Simulation & Paper Trading
Live Trading

(a)

(b)

Figure 6: Workflow for (a) traditional trading strategy development, and (b) trading strategies using RL

2.3 Benchmark

A typical workflow for traditional trading strategy development looks something like Fig. 6(a):

1. **Data Analysis:** First, one (human trader) needs to perform exploratory data analysis to find trading opportunities. The output of this step is an "idea" for a trading strategy that should be validated.

2. **Supervised Model Training:** Next step is to train one or more supervised learning models (such as SVM, linear regression) to predict quantities of interest (e.g., price prediction, quantity prediction, etc.) that are necessary for the strategy to work.

3. **Policy Development:** Based on the current state of the market and the outputs of the supervised models, the agent (human) comes up with a rule-based policy that determines what actions to take. Note that this policy may also have parameters, such as decision thresholds, that need to be optimized. This optimization is done later.

4. **Strategy Backtesting:** Once the policy is established, the agent (human) uses a simulator to test an initial version of the strategy against a set of historical data. The simulator can take things such as order book liquidity, network latencies, fees, etc into account. If the strategy performs reasonably well in backtesting, one can move on and do parameter optimization.

5. **Parameter Optimization:** One can now perform a search, for example a grid search, over possible values of strategy parameters like thresholds or coefficient, again using the simulator and a set of historical data. Here, overfitting to historical data is a big risk, and one must be careful about using proper validation and test sets.

6. **Simulation & Paper Trading:** Before the strategy goes live, simulation is done on new market data, in real-time. That's called paper trading and helps prevent overfitting. Only if the strategy is successful in paper trading, it is deployed in a live environment.

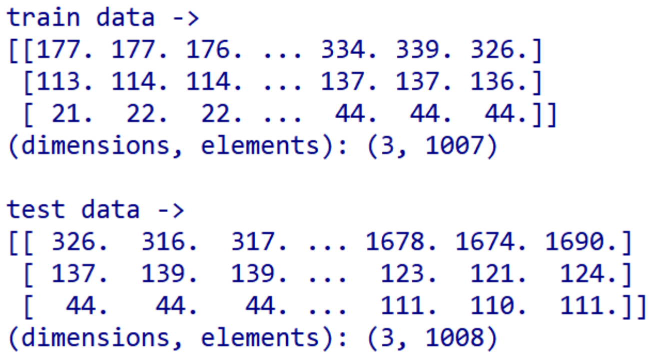
7. **Live Trading:** The strategy is now running live on an exchange.

In the traditional strategy development approach, we must go through several steps, a pipeline, before we get to the metric we actually care about. Developing trading strategies using RL is much simpler and more principled than the traditional approach (see Fig. 6(b)).

In the traditional approach, steps 1-3 are largely based on intuition, and we don't know if our strategy works until the optimization in steps 4-5 is done, possibly forcing us to start from scratch. In fact, every step comes with the risk of failing and forcing us to start from scratch. We do not explicitly take into account environmental factors such as latencies, fees, and liquidity until step 4. Policies are developed independently from supervised models even though they interact closely. Supervised predictions are an input to the policy. Policies are limited to what humans can come up with. Moreover, parameter optimization is inefficient.

The RL process is the formation and adjustment of the policies (Q-table) by inspecting an action and assessing its reward. In the RL approach, instead of needing to hand-code a rule-based policy, RL agent directly learns a policy. The policy can be parameterized by a complex model, such as a Deep Neural Network, we can learn policies that are more complex and powerful than any rules a human trader could possibly come up with. RL agents are trained in a simulation, and that simulation can be as complex as we want, taking into account latencies, liquidity and fees. RL agents learns powerful policies parameterized by Neural Networks, they can also learn to adapt to various market conditions by seeing them in historical data, given that they are trained over a long time horizon and have sufficient memory. This allows them to be much more robust to changing markets. By building an increasingly complex simulation environment that models the real world, we can train very sophisticated agents that learn to take environment constraints into account.

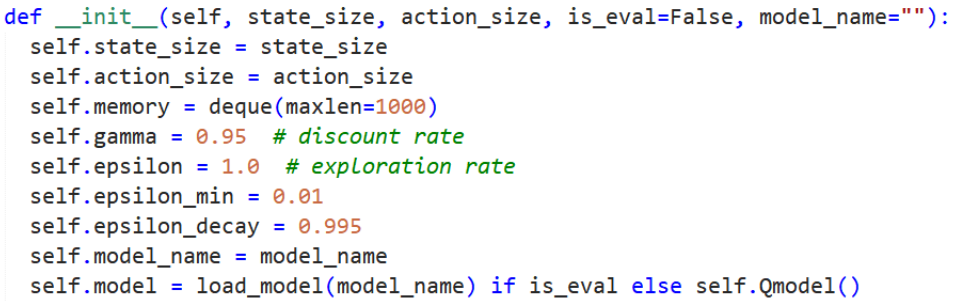
Several interesting reports/articles can be found on the web on automated trading based on RL ([4, 8, 9, 10]), out of which I was motivated by this work [11] in particular. However, the author only considered a single stock and did not take into account any initial investment. In this project, I will consider multiple stocks and an initial investment of $10,000$. In another example ([12]), a different author considered the DQN approach to tackle a similar problem. In this project, my goal is to develop/implement an improved algorithm that will enhance the performance and reliability of the model as well as be able to tackle more complex situations. Finally the results obtained will be comapared with the exisitng ones (e.g., the model reported in [12]).



```
train data ->
[[177. 177. 176. ... 334. 339. 326.]
 [113. 114. 114. ... 137. 137. 136.]
 [ 21.  22.  22. ...  44.  44.  44.]]
(dimensions, elements): (3, 1007)

test data ->
[[ 326.  316.  317. ... 1678. 1674. 1690.]
 [ 137.  139.  139. ...  123.  121.  124.]
 [  44.   44.   44. ...  111.  110.  111.]]
(dimensions, elements): (3, 1008)
```

Figure 7: Training and testing sets after data preprocessing



```
def __init__(self, state_size, action_size, is_eval=False, model_name=""):
  self.state_size = state_size
  self.action_size = action_size
  self.memory = deque(maxlen=1000)
  self.gamma = 0.95  # discount rate
  self.epsilon = 1.0  # exploration rate
  self.epsilon_min = 0.01
  self.epsilon_decay = 0.995
  self.model_name = model_name
  self.model = load_model(model_name) if is_eval else self.Qmodel()
```

Figure 8: Trading agent parameter initialization

2.4 Methodology

2.4.1 Data Preprocessing

Rather than reading data from Yahoo's finance API to a DataFrame every time, I first save the DataFrame to a CSV file and then read data from the CSV file into a DataFrame from next time onwards. When the CSV data is read, the date is parsed as a datetime object and the 'Date' column is set as the index.

I use the 'Adj Close' column data from the original DataFrame for each stock to create an n-dimensional array of data. This data is split into training and testing sets (Fig. 7) which are then used to create the trading environment for training and testing separately.

2.4.2 Implementation

The project consists of the following four files: run.py, envs.py, agent.py and utils.py. The implementation process of the algorithm for the training and testing phases are discussed below.

Training Phase:

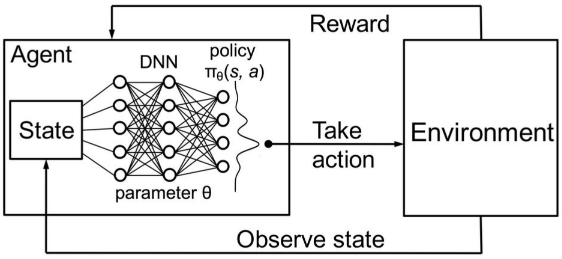


Figure 9: Internal details of agent-environment interaction

- Once the data is ready, create an instance of the trading environment class (env) for training using the Gym interface and initialize the variables (such as, stock_owned, stock_price and cash_in_hand) defined within the class.

- Create an instance of the trading agent class (trade_agent) and initialize the variables such as state_size and action_size. Also define the parameter values within the class (see Fig. 8).

- Define the neural network architecture and training parameters, and loss function.

- Reset the variables in 'env' at the beginning of each episode.

- At each time step of each episode, the current 'state' is fed to the agent as it produces one of 3^n possible actions (n = number of stocks) predicted by the neural net (see Fig. 9).

- The action predicted by the agent is then passed to the 'env'. At each time step, the 'step' method within 'env' calculates the instantaneous profit and returns the next state, reward and current portfolio value.

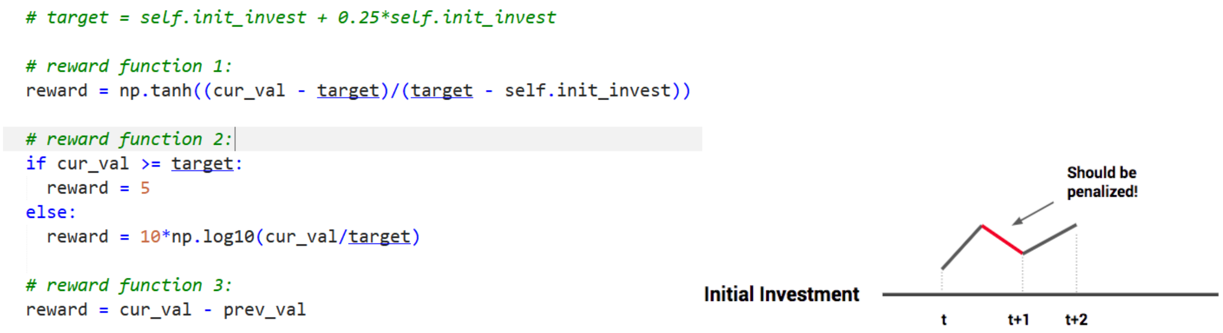
- At each time step, the $<$$state(s)$, $action(a)$, $reward(r)$, $next\_state(s')$$>$ tuples are stored in the memory and total cumulative score/profit is calculated for that step.

- At the end of each episode, the total reward ($\sum score$ for all time steps) for the episode is stored in a list.

- As the agent is trained, the memory/buffer (defined as a deque of maxlen=1000) continues to grow. If $memory > batch\_size$ (= 25), the agent takes random batches of $<s$, a, r, $s'>$ from the buffer to calculate the TD-Target, $Q(s, a)$. This step is referred to as 'experience replay'.

- Finally, the weights of the model are saved in a separate file.

- After 1000 training episodes, based on the total cumulative rewards throughout all episodes, the performance of the agent is assessed and the results are plotted.



```
# target = self.init_invest + 0.25*self.init_invest

# reward function 1:
reward = np.tanh((cur_val - target)/(target - self.init_invest))

# reward function 2:
if cur_val >= target:
  reward = 5
else:
  reward = 10*np.log10(cur_val/target)

# reward function 3:
reward = cur_val - prev_val
```

(a) (b)

Figure 10: (a) Reward functions, (b) The third reward function penalizing negative profit at each time step

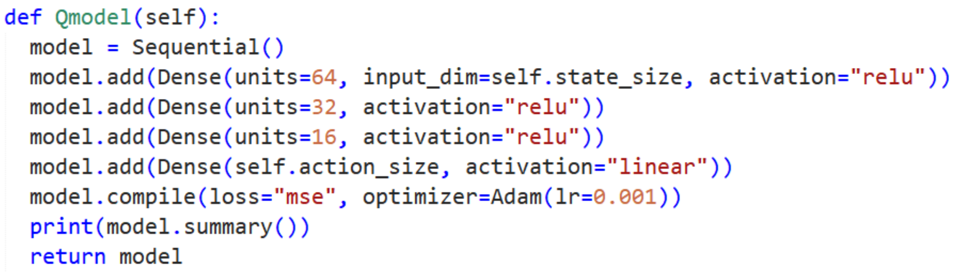
Testing Phase:

- Recreate the environment with the test datasets and load the weights of the previously trained model.
- Follow all the processes of the training phase except the steps of storing the tuples in the memory and experience replay.
- The agent's performance in the testing phase is assessed based on the total cumulative rewards throughout all episodes and the results are plotted.

2.4.3 Refinement

The most important factor for RL is the reward function. I experimented with different reward functions such as shown in Fig. 10(a). The first two won't help the agent to learn from the delta value between each $(t, t+1)$ step, as illustrated by the graph in Fig. 10(b). Therefore, the best reward function is the third one which was implemented in the final model.

I tuned some parameters such as the discount rate (gamma) and chose gamma to be 0.95 for the final model. I also examined the performance of the model using different memory/buffer and batch sizes. I did not observe any noticeable effect on the performance of the model for different configurations, however, the performance is obviously affected if the memory/batch size is too large or too small [13]. The memory and batch size for the final model were chosen to be 1000 and 32, respectively, although different batch sizes can be input as a command-line argument. I also examined the model by applying different sell and buy conditions. For example, when buying, buy as many as *cash in hand* allows, and when selling, sell all the shares. In the final model, I applied some thresholds for buying and selling decisions: when buying, buy as many as $cash\ in\ hand > 10\%\ of\ initial\_investment$ allows, and when selling, instead of selling all the shares in a single trade, always keep 20% of the shares in hand. This



```
def Qmodel(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=16, activation="relu"))
  model.add(Dense(self.action_size, activation="linear"))
  model.compile(loss="mse", optimizer=Adam(lr=0.001))
  print(model.summary())
  return model
```

Figure 11: Neural Net Architecture for the final model

way, a more robust model can be obtained. I analyzed the system with a varying length of training window (e.g., 2.5, 5, 10 and 15 years). The problem with longer training window is that training time increases as more data is fed. Using a training window of more than 10 years could be very time consuming as the system would then take too long to generate the results. I chose a training window of 4 years (splicing 8 years data into training and testing set by 0.5) that yielded reasonable outcomes. However, model performance would improve (converged to a more stable/less volatile solution) if the amount of training data would have been increased. It should also be noted that for financial data, using too much data (that is, using a training window that extends far into the past) could be detrimental to model performance [14].

3 Results

3.1 Model Evaluation and Validation

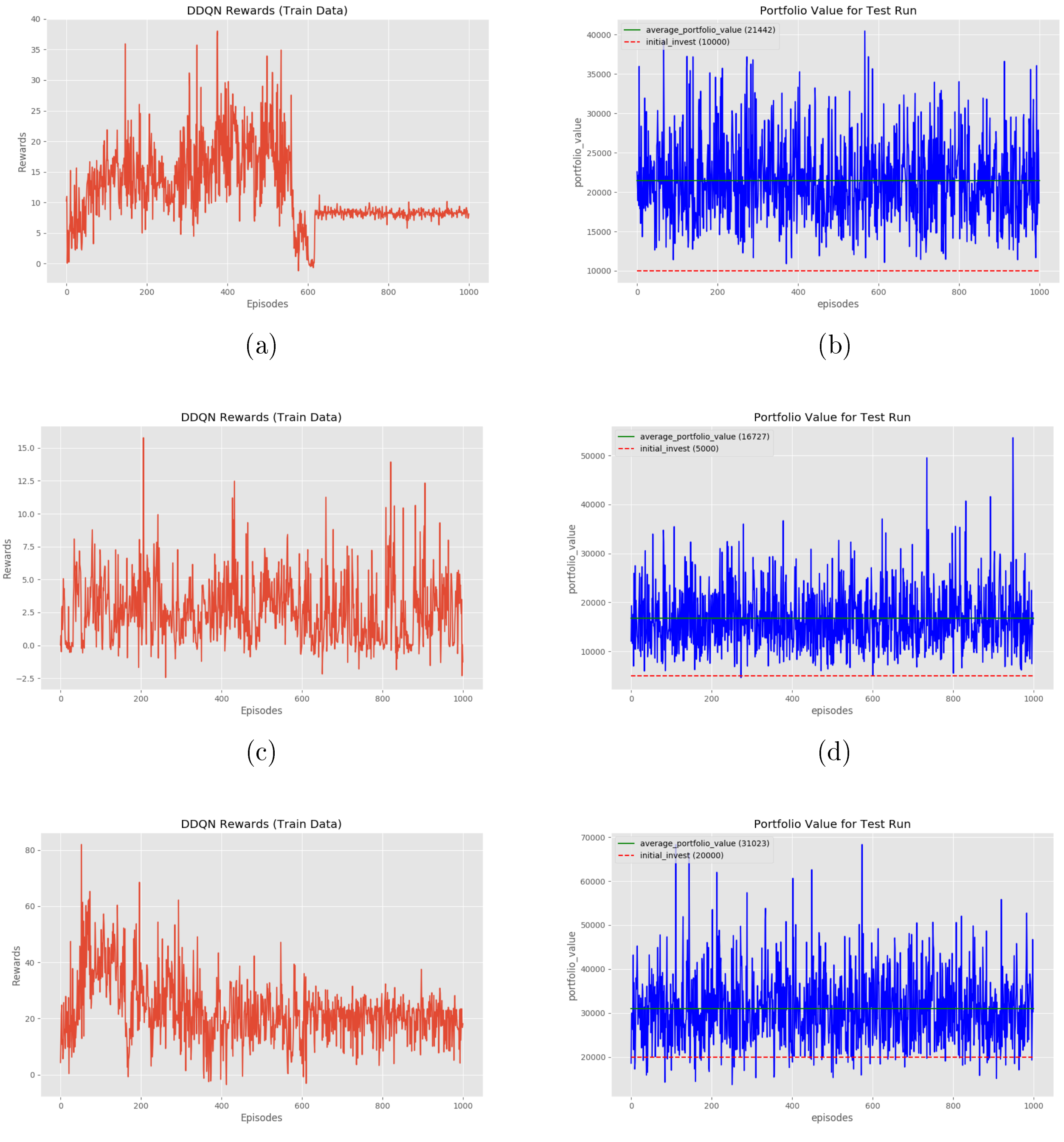
I experimented with different neural net architectures, i.e., different numbers of layers and nodes. For the final model, I chose the minimum possible configuration that produces the optimal performance. The final architecture consists of 2 hidden layers (see Fig. 11). The other configurations does not produce much noticeable difference/improvement in performance than the one chosen.

The total cumulative rewards throughout training episodes are plotted in Fig. 12(a). In Fig. 12(b), the portfolio values for each episode during the testing phase have been plotted. The model performs reasonably well both in the training and testing phases.

In order to verify the robustness of the model, I investigated the performance of the model with different inputs:

1. 4 stocks (CSCO, AZO, GM, INTC) and initial investment of 20,000, and

2. 1 stock (NVDA) and initial investment of 5,000.

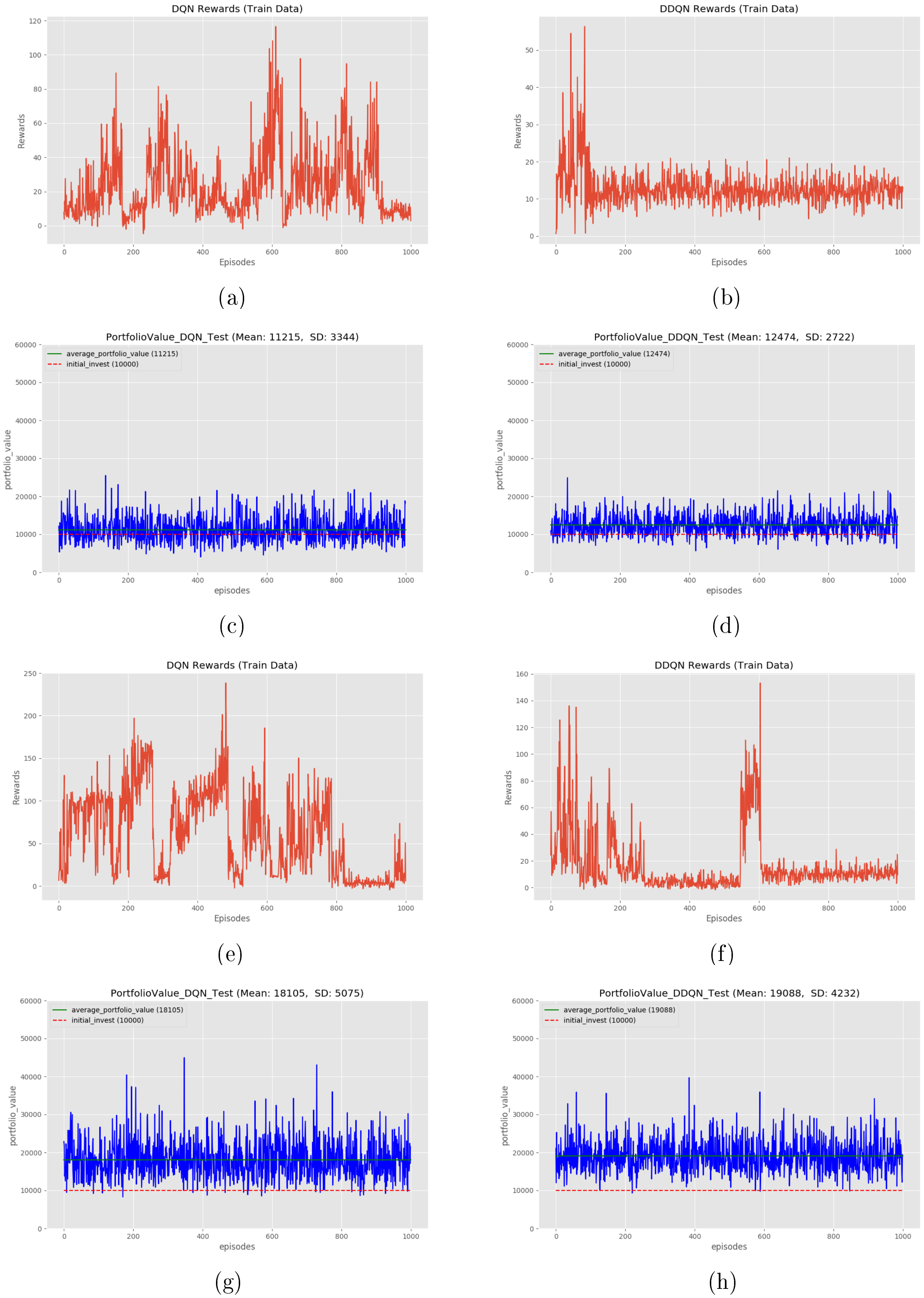


(a) (b)

(c) (d)

(e) (f)

Figure 12: Model performance in the case of 3 stocks (MSFT, AMZN and IBM) and initial investment of 10,000: (a) rewards vs episodes for the training set, and (b) portfolio_value vs episodes for the test set. Model performance in the case of 1 stock (NVDA) and initial investment of 5,000: (c) rewards vs episodes for the training set, and (d) portfolio_value vs episodes for the test set. Model performance in the case of 4 stocks (AZO, CSCO, GM, INTC) and initial investment of 20,000: (e) rewards vs episodes for the training set, and (f) portfolio_value vs episodes for the test set.



(a) (b) (c) (d) (e) (f) (g) (h)

Figure 13: Comparison between the benchmark model (a, c, e, g) and the current model (b, d, f, h). The dataset for these results includes 12 years of daily data ranging from 12/01/2006 to 11/30/2018. The data was split into training and test set (7:3). Stocks used in (a - d) are MSFT, IBM and QCOM. Stocks used in (e - h) are MSFT, IBM and AMZN.

In all cases, the performance of the model was consistent and the agent was able to carry out the trading operations in the desired way and generate reasonable profits. The results are plotted in Fig. 12(c-f). It is interesting to note that with the same system settings, after 1000 episodes, the performance of the model with 3 stocks looks more stable and less volatile.

3.2 Justification

Note that in all cases in Fig. 12, the agent was able to generate a reasonable profit. For example, with MSFT, AMZN and IBM stocks and an initial investment of $10,000$, the agent was able to make an average profit of $11,442$ in 4 years which is 20.95% annualized ROI. But as we can see the portfolio values are highly volatile, indicating the instability of our agent and overfitting of the training data. Although we can see a decent average profit of $11,442$, the variance (or risk) is too high to be ignored. I did not consider some real-world factors, such as transaction cost. Another issue is that in this project, I have experimented the system only with a maximum of four stocks. However, as we increase the number of stocks, the action and state space will grow exponentially, which would make the system even harder to converge. Therefore, although the results seem promising, I would not consider this model to be fully realistic yet and some more improvements need to be done.

In order to compare the results for the current model with the benchmark one [12], I have produced the results for the benchmark model using the same code provided in [15] but for a different time window (12/1/2006 - 11/30/2018). The data was divided into training and test set (7 : 3). I have also produced the results for the current model using the same training and test dataset. The performance of the two models are then compared in Fig. 13. It is obvious in Fig. 13(a, e, b, f) that in the case of the current model, the algorithm converges to a solution much faster than the benchmark model. Also the solution is relatively more stable for the current model. Moreover, the porfolio values in the case of the current model exhibit less volatility relative to its benchmark counterpart (*cf.* Fig. 13(c, d, g, h)). It is interesting to note that in comparison with the benchmark model, average profit generated by the current model is higher, whereas standard deviation (SD) is less. Therefore, based on the results shown in Fig. 13, it can be inferred that the current model performs better than the benchmark model.

4 Conclusion

4.1 Free-Form Visualization

Fig. 14 shows how the trading agent executes the trading operations (sell/hold/buy) based on the stock data for three stocks (MSFT, AMZN and IBM) to generate an average profit of

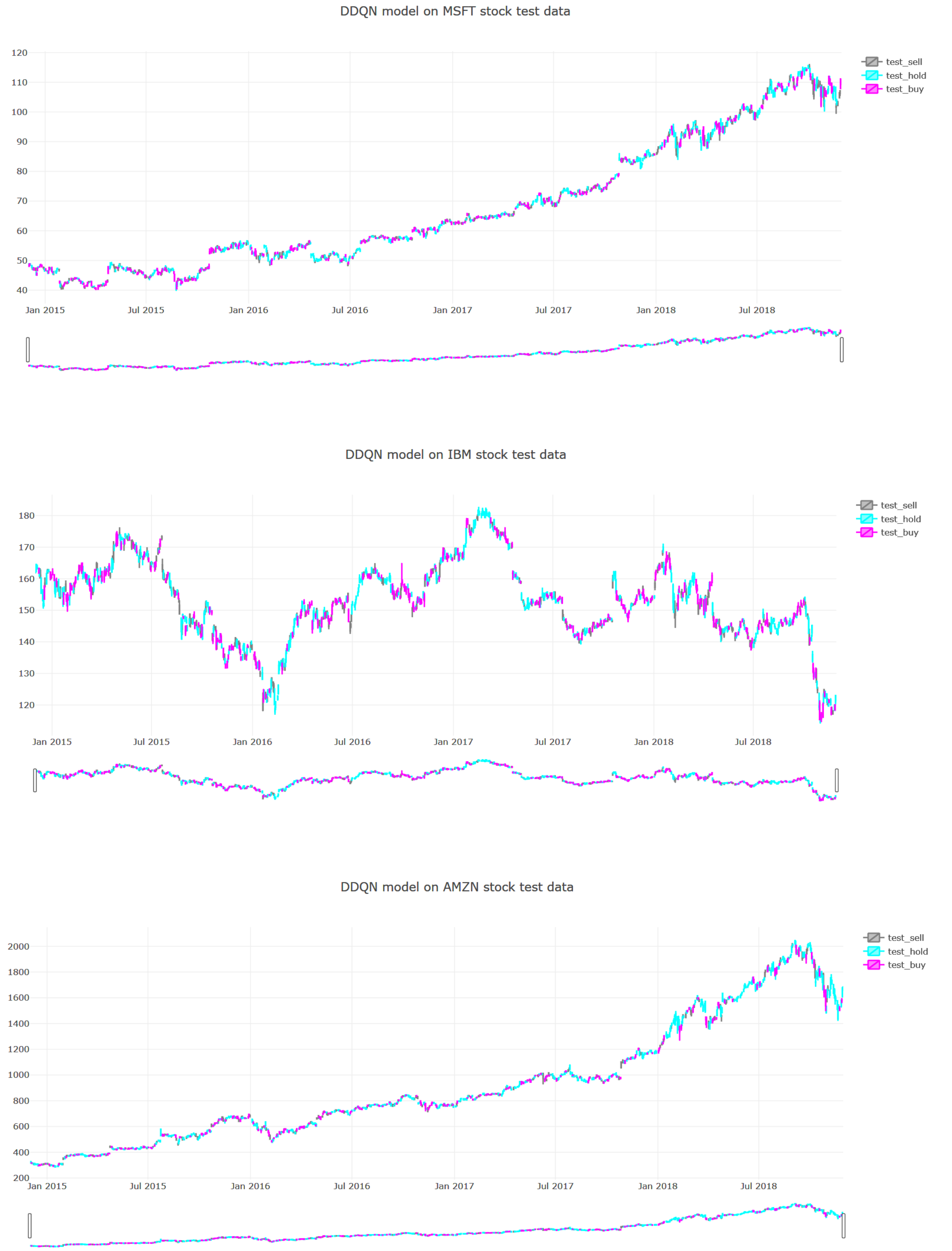


Figure 14: Stock trading using DDQN algorithm. Only the results for the test data are shown. The original HTML files are saved in the Github repo.

$11,442$. It seems that (Double) Q-trading system has learned how to take different actions in different market situations. For example, in a downside market, it learns selling is more profitable than holding, and thus tends to hold a short position. This sensitivity against market status can be largely attributed to the power of the deep Q-network in discovering the status of the market from the vast and noisy historical price signals. However, although the agent does reasonably well in making profit in all cases considered in the previous section, its performance exhibits instability as seen in Fig. 12.

4.2 Reflection

The process for this project can be summarized as follows.

1. The problem to be dealt with in this project has been defined through background study. The availability of sufficient data for training, testing and verification for this project was checked. The initial idea and concept were obtained from Udacity's "Machine Learning for Trading" course [1].

2. The stock data was downloaded (from Yahoo Finance API) and preprocessed.

3. Proper understanding of the datasets (stock data) and its features were acquired, and initial data analysis and statistical comparison among various stocks were carried out. For steps 2-3, these tutorials [16] were quite helpful in this regard.

4. The traditional approach and the existing results were discussed, and the benchmark for this project was set.

5. Before developing/implementing the improved algorithm, the results for other existing models were reproduced and analyzed. The algorithm/technique to be employed in this project was studied through various reference materials.

6. Inspection of the code at every possible level was conducted in order to verify if the results produced by the model is reliable. For example, initially when I developed the code, I did not notice that the order of the train and test data was reversed with respect to the date (index) after data preprocessing for which the model was performing very poorly. I spent quite a lot of time to make various adjustments to improve the model performance without realizing that the actual problem was with the data itself. Once that problem was identified and fixed, the performance of the model improved as desired.

7. A reasonable length of the data window was selected. Initially, I was using a window of 15 years with 75% data ($\sim$ 11 years) for training set, which made the system run too slow as the time to execute each episode was too long. Therefore running the system

for even 500 episodes took several hours. I then reduced the size of the data window to 8 years using 50% of the data (4 years) as training set. Now the execution time per episode significantly reduced without much affecting the performance of the model.

8. The weights of the model during the training phase were saved to be used in the testing phase.

9. Various system parameters were tuned to improve the performance of the model and the configuration producing the most consistent/stable results was selected for the final model.

10. The final model was tested with various different inputs to verify the robustness of the model.

11. Finally, the performance of the model was compared with the benchmark model.

Since I did not have any finance background, initially I had to make myself familiarize with the real-world trading system (at least gather some basic idea) before designing/improving the RL trading environment and the trading agent. Another challenge was to choose the reward function that will be able to capture the actual profit and loss at each time step. I read some reference materials ([3, 17]) and experiemented with several reward functions before selecting the one for the final model.

4.3 Improvement

In this project, I have implemented DDQN algorithm for stock trading which exhibits better stability, less volatility and faster convergance in the model performance compared to its benchmark/DQN couterpart (see also [11, 12]). Another approach that has been suggested in [18] termed Duelling DQN which shows some promising results by outperforming its DQN and DDQN counterparts (see also [19]). Therefore it would be interesting to apply duelling DQN technique to the model implemeted in this project. Due to limitations of the computing power, I could only produce very limited results with longer training window. With an increase in training data, better performance could be acheived. In addition, if we add more features to the model, such as, binary company news (e.g., positive/negative tweets), company performance information (market share, growth rate, market capitalization, earning, revenue, profit margin etc.) and competitors' news & performance, more reliable performance can be achieved. Moreover, we need to consider some real-world factors, such as transaction costs, in the algorithm.

References

[1] https://www.udacity.com/course/machine-learning-for-trading–ud501

[2] ”Reinforcement Learning: An Introduction”, 2nd Edition, Richard S. Sutton and Andrew G. Barto

[3] http://cslt.riit.tsinghua.edu.cn/mediawiki/images/5/5f/Dtq.pdf

[4] https://medium.com/@gaurav1086/machine-learning-for-algorithmic-trading-f79201c8bac6

[5] H. Hasselt, A. Guez, and D. Silver, “Deep Reinforcement Learning with Double Q-Learning”, Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence (AAAI-16)

[6] https://towardsdatascience.com/self-learning-ai-agents-part-ii-deep-q-learning-b5ac60c3f47

[7] https://www.nature.com/articles/nature14236

[8] https://lucenaresearch.com/deep-reinforcement-learning/

[9] http://www.wildml.com/2018/02/introduction-to-learning-to-trade-with-reinforcement-learning/

[10] https://www.youtube.com/watch?v=rRssY6FrTvU&amp;t=469s

[11] https://www.kaggle.com/itoeiji/deep-reinforcement-learning-on-stock-data

[12] https://shuaiw.github.io/2018/02/11/teach-machine-to-trade.html

[13] https://www.padl.ws/papers/Paper%2018.pdf

[14] https://robotwealth.com/optimal-data-windows-for-training-a-machine-learning-model-for-financial-prediction/

[15] https://github.com/ShuaiW/teach-machine-to-trade

[16] https://pythonprogramming.net/handling-stock-data-graphing-python-programming-for-finance/

[17] https://arxiv.org/abs/1807.02787

[18] https://arxiv.org/abs/1511.06581

[19] http://torch.ch/blog/2016/04/30/dueling_dqn.html