**Stock Trading Using**

**Reinforcement Learning**

A project Report

San Jose State University  
**CMPE 297-99**

By:

**Vidhi Shah**

**Reetika Goel**

**Pragya Gautam**

**Sithara Krishnamurthy**

**ABSTRACT**

Trading is an activity of buying and selling financial instruments with different goals, like making a profit, gaining protection from future price movement or just getting what you need. In the financial market, people have been trying to predict future price movements, as this promises lots of benefits. This problem is known to be complex and there are lots of financial consultants, investment funds, banks, and individual traders who are trying to predict the market and find the best moments to buy and sell to maximize profit. Using Reinforcement Learning we can address this problem, where we have some observation of the market and we want to make a decision if we want to buy a stock, sell a stock, or wait. If we buy before the price goes up, our profit will be positive, otherwise, we’ll get a negative reward. For the implementation, we used Yandex Company Stocks dataset for the year 2015 and 2016, and RL model architecture based on DQN and Double DQN using an FFDQN model. We train the agent to learn when it would be the best time to buy a single share and then close the position to maximize the profit. In this project, we aim to get as much profit as possible.

**Table of Contents**

[1. Introduction 4](#_Toc26538516)

[2. Goals & Deliverables 5](#_Toc26538517)

[3. Scope, Future Work & Risks 5](#_Toc26538518)

[4. Project Planning (Scrum/Agile) 6](#_Toc26538519)

[5. Implementation 7](#_Toc26538520)

[6. Result 11](#_Toc26538521)

[7. Conclusion 11](#_Toc26538522)

## **Introduction**

In the financial market, there are many financial instruments for which prices change over time. Trading involves frequent transactions, such as the buying and selling of stocks, commodities, currency pairs, or other instruments. The goal is to generate returns that outperform buy-and-hold investing. A trader would most likely look at some charts of a stock’s price action. From there, they would combine this visual information with their prior knowledge of similar price action to make an informed decision of which direction the stock is likely to move.

There are lots of financial consultants, investment funds, banks, and individual traders who have been trying to predict future price fluctuations, as this guarantees a lot of benefits. However, this is a very complex problem. To decide whether to buy, sell, or wait for any financial instrument, one needs careful observations of the market and then predict to find the best time to buy and sell and get maximum profit.

The above problem can be addressed using a Reinforcement Learning perspective where we have some observation of the market and we want to make a decision: buy, sell, or wait. The profit is positive when buying is done before the prices increase otherwise it is negative. In this project, we will aim to get as much profit as possible.

To address the above problem, Yandex data dataset is used. This dataset has 2 CSV files of Yandex Company stocks for year 2015 and 2016. In this CSV file, the first two columns are the date and time for a minute, the next four columns are open, high, low, and close prices and the last value represents the amount of buy and sell orders performed during the bar (every minute interval).

## **Goals & Deliverables**

The goal of this project is to start with replicating an existing model for trading

functionality using Gym’s Env class API. Create an agent and environment and then investigate whether it will be possible for our agent to learn when the best time is to buy one single share and then close the position to maximize the profit. The purpose here is to show an increase in the existing feed-forward deep Q-network’s (FFDQN) performance.

The environment will have an agent that will be trained to do nothing, buy a share or

close a position. If the agent has already got the share, nothing will be bought, otherwise a small percentage of the current price will be paid. The step-by-step reward will be equal to the last movement or the agent will receive a full reward at once only after the close action. We will train the agent using different learning algorithms and tune their hyperparameters to identify the best learning for our agent in the environment.

## **Scope, Future Work & Risks**

Currently, the scope of the system consists of open, high, low, and close prices given to

an agent. The environment is implemented in the StocksEnv class. To improve the overall

system, we will be tuning the RL model with:

● Different hyperparameter values

● Different architectures

For future work, we will try to tune the model including different datasets to see how the

model performs in different conditions.

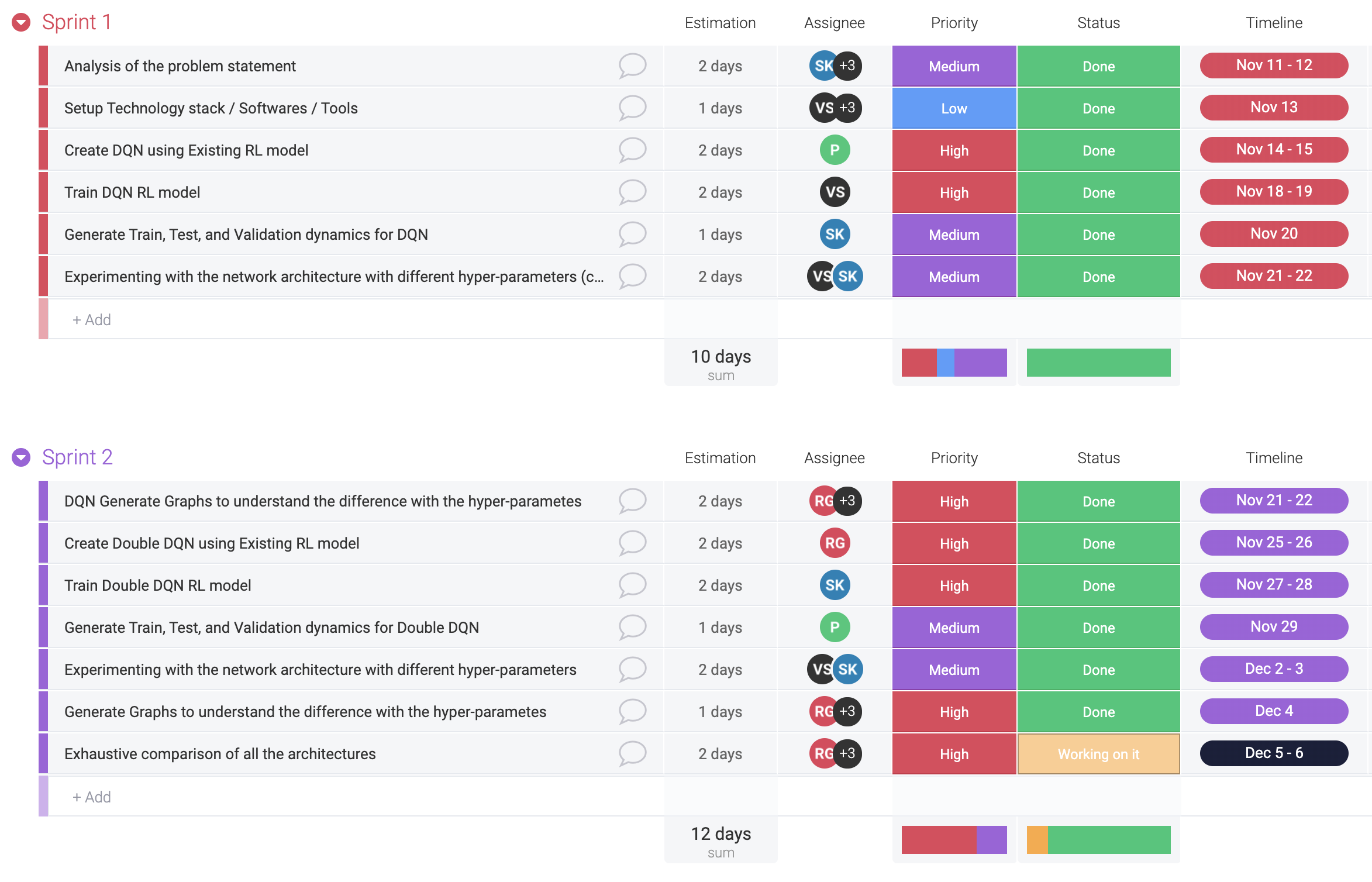
The existing model is built with an assumption to convert every bar “open, high, low, and

close” prices to three numbers as high, low, and close prices. However, this is a risk since we

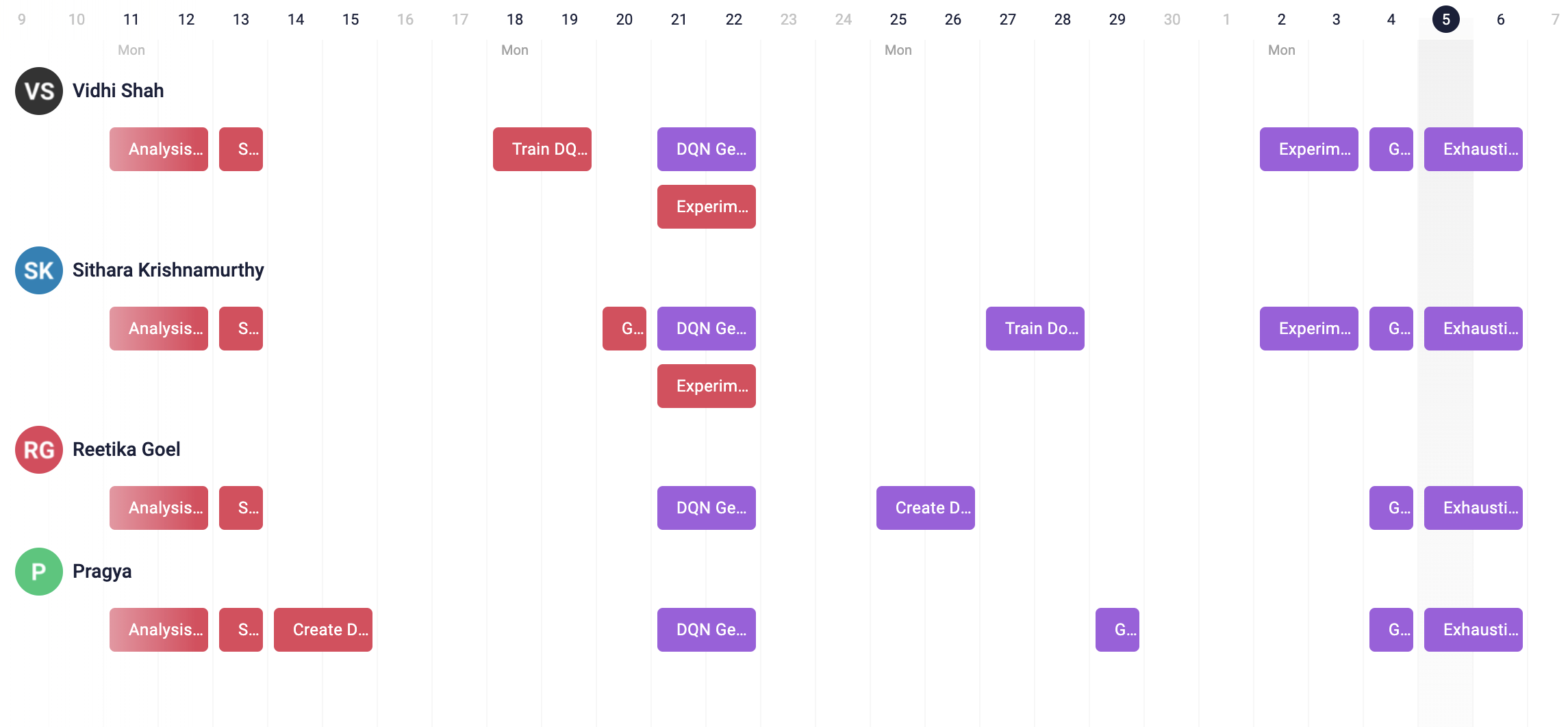
will have a loss of price information which may lead to a suboptimal solution.

## **Project Planning (Scrum/Agile)**

As part of the deep reinforcement learning project, we “The Seekers” team, started with a project planning meeting. We planned 2 sprints for the course of the project, where each sprint lasted 2 weeks. To list and track our tasks, we used ‘Monday’, an Agile project management software. In the sprint planning meetings, we listed all our backlog items, defined the user stories for each sprint, gave story weight/points for the tasks and assigned them between the team members.



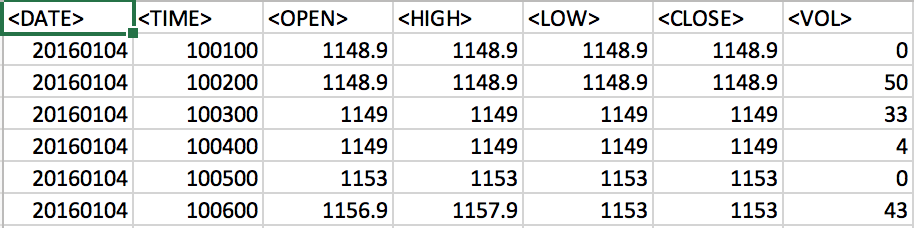
As part of this project, we started with an analysis of the problem statement i.e. how to implement stock trading using reinforcement learning. We investigated whether it will be possible for an agent to learn when it would be the best time to buy a single share and then close the position to maximize the profit. During our analysis, we observed an existing RL model architecture based on DQN and Double DQN using an FFDQN model on the Yandex data dataset. We also discovered that the existing models can further be trained with different architectures such as Duel Double DQN, Noisy Network, N-Step DQN, etc. for future enhancements.



## **Implementation**

**Dataset**

For our analysis, we are using the Russian stock market prices for the period of 2015-2016. It is a time-series data present in a CSV. The CSV files have M1 bars, which means that every row in the CSV corresponds to a single minute in time and price movement during this minute is captured with four prices: open, high, low, and close. Here, an open price is the price at the beginning of the minute, high is the maximum price during the interval, low is the minimum price, and the close price is the last price of the minute time interval. Every minute interval is called bar and allows us to have an idea of price movement within the interval. We are using the data of Yandex company stocks for the year 2015-2016 to train and validate the agent. Below is a snapshot of the dataset.



**Environment**

The finance domain is large and complex, but for our problem, we are using a simple environment, which uses price as an observation. We will investigate whether it will be possible for our agent to learn when the best time is to buy one single share and then close the position to maximize the profit.

The trading functionality is built using the Gym’s environment-class API. It uses several internal classes to keep its state and encode observations. Our environment-class provides two ways to create its instance. The first way is to call the class method with data directory as the argument and another way is to construct the class instance directly. The constructor of the environment accepts lots of arguments to tweak the environment’s behavior and observation representation like prices, bars count, commission, reward on close and others.

We are implementing the basic trading agent in its simplest form.

**Observations**

The observation will include the following information:

* N past bars, where each have open, high, low, and close prices.
* An indication that the share was bought some time ago (it will be possible to have only one share at a time)
* Profit or loss we currently have from our current position (the share bought)

**Actions**

At every step, which will be after every minute’s bar, the agent can take one of the following actions:

* Do nothing: Skip the bar without taking actions.
* Buy a share: If the agent has already got the share, nothing will be bought, otherwise we’ll pay the commission, which is usually some small percentage of the current price.
* Close the position: If we’ve got no share previously bought, nothing will happen, otherwise we’ll pay the commission for the trade.

**Reward**

The reward for the agent is received by splitting it into multiple steps during the ownership of the share.

The prices in our environment observation are represented using a relative movement, like “stock has grown 1% during the last bar” or “stock has lost 5%".To achieve this, we have converted every bar “open, high, low, and close” prices to three numbers showing high, low, and close prices represented as a percentage to the open price. This will help the system to find repeating patterns in the price level.

**Model**

We are using a simple feed-forward network with three layers as the architecture of DQN, to output Q values.

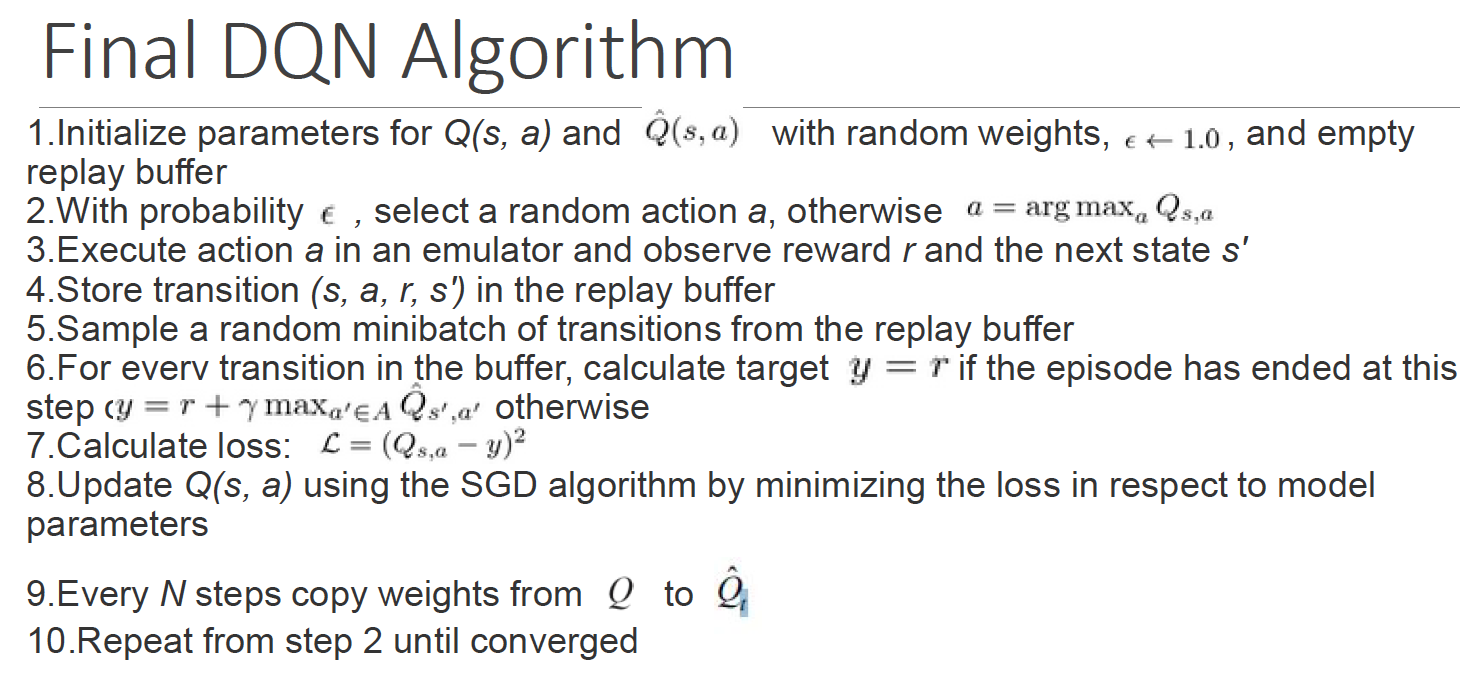
**Training**

* They’re using epsilon-greedy action selection to perform exploration. The epsilon linearly decays over the first 1M steps from 1.0 to 0.1.
* A simple experience replay buffer of size 100k is being used, which is initially populated with 10k transitions.
* For every 1000 steps, we calculate the mean value for the fixed set of states to check the dynamics of the Q-values during the training.
* For every 100k steps, we perform validation: 100 episodes are played on the training data and on previously unseen quotes. Characteristics of orders are recorded in Tensor Board, such as the mean profit, the mean count of bars, and share held. This step allows us to check for overfitting conditions.

We are training our agent using DQN and Double DQN algorithms.

**DQN**

When deep neural networks are used as general nonlinear function approximators in the context of Q-learning, the system is referred to as Deep Q Network or DQN for short.



**Double DQN**

DQN has a tendency to overestimate values for Q, which may be harmful to training performance and sometimes can lead to suboptimal policies. A simple change in the DQN algorithm fixes overestimation completely and this architecture is known as double DQN. The core implementation is very simple. We need to do is to slightly modify the loss function in Double DQN. Here, we calculate the best action to take in the next state using our main trained network, but values corresponding to this action come from the target network.

* train\_DQNDouble.py - This is the main class, which is used to load the data, create an environment, define the actions, observations, and agent, gather rewards. The training of the agent is defined here, and the model is saved here.
* data.py - This class has functions to read the CSV, load the data and generate the prices dictionary.
* models.py - Class with the DQN neural net layer.
* common.py - This class has a function that takes the batch of transitions and packs it into the set of NumPy arrays. It also has functions to calculate the loss.
* environ.py - The environ class has class to create the environment. It has functions to reset env, step function for environment and action, and state class.
* run\_model.py - This class loads the model, trades on prices provided to it and draws the plots with the profit change over time.

## **Result**

Please view APPENDIX A



## **Conclusion**

## In this project, we generated a practical example of reinforcement learning and implemented the trading agent for the stock trading environment. We tried two different training algorithms: A Deep Q-Network and a Double Deep Q-Network, both having a simple feed-forward neural network with a price history on input. The validation was also performed on two datasets: our training data and the previously unseen data from the same stock, but for a different time period. Below are some of the conclusions that we would like to make based on the results achieved:

**INTRA Algorithm Comparison (For distinct hyperparameter values)**

* Comparing DQN for distinct gamma values yielded that, lesser the gamma value, better the length of an episode is for an agent. Over the training time, the length of the episode increased from 6 bars to 16 bars in case of GAMMA = 0.95, while for GAMMA = 0.99, it increased from 6 bars to up till 12 bars. This means that the agent is holding the share longer and longer to increase the final profit in case of lesser GAMMA value.
* Similarly, comparing reward values for different Alpha values for DQN algorithm showed that, lower learning rates generates higher rewards.
* Looking at rewards for DQN algorithm with 0% commission versus 0.1% commission and different Alpha and Gamma values, it is seen that reward values are better without commission.

**INTER Algorithm Comparison (For different training algorithms)**

* Looking at last 100 training steps, DQN outperformed double DQN to generate better values for a length of an episode.
* Comparing the last 100 rewards, Double DQN looks marginally better than DQN. Saying that, one cannot really conclude which one of it wins over the other. This is because for Alpha = 0.01, Gamma = 0.99, DQN got better rewards than Double DQN while for Alpha = 0.01, Gamma = 0.95 it is reverse.