Imperial College Business School

MSc Risk Management & Financial Engineering 2018-2019
Applied Project

OPTIMAL TRADE EXECUTION: A DEEP REINFORCMENT LEARNING APPROACH

August 2019

Client Specification

Our client is Bank of America Merrill Lynch, Electronic Trading Equities team. The client has been aware of the increasing attempts in exploring and consequently applying Machine Learning (ML) techniques in different problems of the financial industry. After the increasing evidence of Reinforcement Learning (RL) success Stochastic Optimal Control problems, and including the Optimal Trade Execution one the bank is increasingly keen to explore its potential for its Equities Electronic Trading - Execution business. These successes include both in the academia, for example (Ning et al, 2018) (Dabérius et al, 2019) and the industry (J.P. Morgan, 2016).

Table of Contents

1 Introduction

Our problem falls under the umbrella of the Market Microstructure literature in finance and more specifically is concerned around the Limit Order Book and Price Impact fields. On the industry side, it falls under the electronic execution business which is mainly concerned with 3 problems: Scheduling (also known as Optimal Trade Execution or simply Optimal Execution), Order Type, Routing. Traditionally, the first breakthroughs to the Optimal Execution problem came around the 2000s, with most famously the (Almgren & Chris, 2000). These methods formulate the problem as a Stochastic Optimal Control problem, make assumptions on the Price Impact process, and solve analytically by Dynamic Programming. Another approach to solving Stochastic Optimal Control problems which is instead model free would be Reinforcement Learning. Since the major breakthrough in Reinforcement Learning around 2013 and more specifically with the formulation of Deep Reinforcement Learning approach, as famously publicized by (Deepmind Technologies, 2013), Stochastic Optimal Control problems have been gaining more and more attention through a Deep Reinforcement Learning approach.

At the same time, on the industry side, the field of optimal execution has increasingly been gaining attention for different reasons. As trading is becoming more and more electronified, with estimated percentage of volume of the electronic trading in US stocks was 85% in 2012 (Glantz and Kissell, 2013), the field started playing a more central role in the trading business of different financial institutions. As a consequence, we aim to explore the potential of improvements in such areas, through a Reinforcement Learning approach.

The rest of this paper is structured as follows. In the Section 2 we formulate our problem, and outline our Reinforcement Learning approach to solving it. In Section 3 we outline the specifications to our method, and the data used, and finally in Section 4 present the results and discuss them.

This paper aims to use the (Ning et al, 2018) paper as the main resource for formulating the financial side of the problem in a Reinforcement Learning setting, and the (DeepMind Technologies, 2013) for the formulation of general (Deep) Reinforcement Learning. We aim to further examine the insights given by (Ning et al, 2018) and provide some of our own ones, through examining some different specifications on the method applied.

2 Methodology Overview

2.1 *Problem Introduction*

To explain the problem, let us first introduce the Limit Order Book (LOB). As shown in Figure 1, when a trader enters the market, the current price available for him to buy[1] or "ask" by the market (Level 1), is limited to only a specific volume, roughly 2000 shares in this case. As soon as a trader's order exceeds the Level 1 volume, then the remaining of the amount will be executed at the next price levels (Level 2 onwards), accordingly. This effect of a trader's order on the LOB and consequently the market price of a stock, is what we call Price Impact. The aim of Optimal Execution is to make sure that orders sent to the market, and especially large ones, have a minimal price impact, and therefore incur minimum transaction costs.

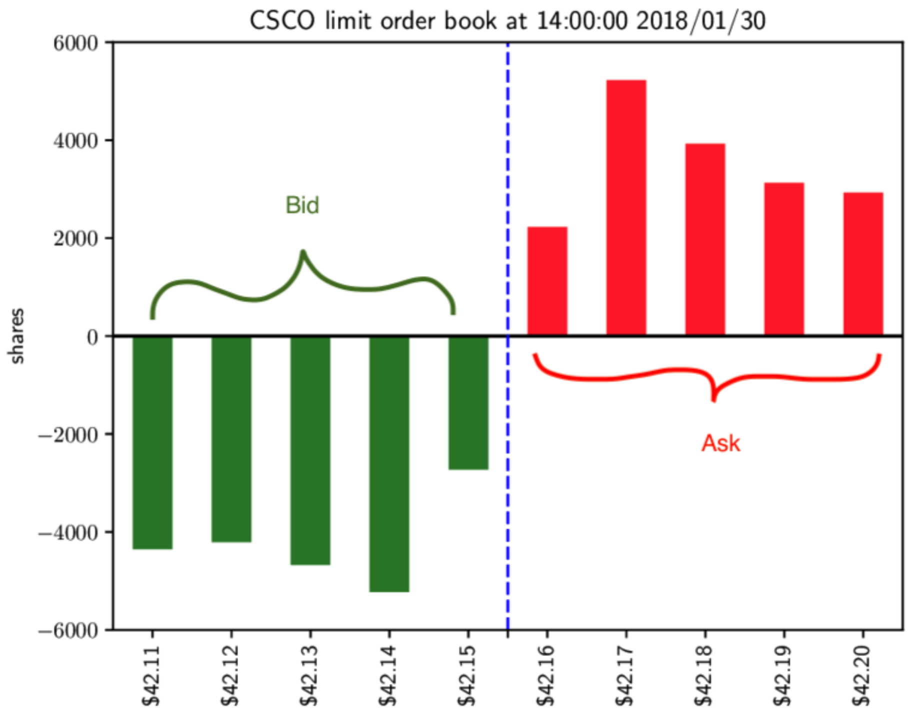


Figure 1 *Limit Order Book of Cisco (CSCO) on 2018/01/30 14:00:00. Green side of the book represents the "Bid" orders in the placed in the book, or the "bid" offers of the market and red side represents the "Ask" orders placed in the book. On the x-axis we have the price and on the y-axis the volume of shares at the specific price, as indicated by each bar. The Level 1 of the book on the ask side is at $42.16, Level 2 at $42.17 and so on, and on the Bid side consequently, starting from $42.15 for Level 1.*

[1] Please note that here we assume that a trader can only execute a Market Order for simplicity. A market order(MO) is when an order (of a fixed number of shares to buy or sell) is matched by the best available price remaining in the LOB. (Ning et al, 2018)

2.2 *Problem formulation*

We formulate our problem mathematically, according to (Almgren & Chriss, 2000). Suppose a trader has q units of a security and wants to completely liquidate before time T. We divide T into N intervals of length $\tau = T/N$ and specifically, discretised times $t_k = k\tau$ for k = 0,1,…,N. Moreover, let a trading trajectory be the sequence q_0, …, q_N where q_k is the number of units we hold at time t_k or the *inventory*, and consequently $x_k = q_k - q_{k-1}$ is the units of shares to be executed at time t_k, or *schedule*. Our initial holding is $q_0 = q$ and at T we requite $x_N = 0$. Finally denote the price at t = k by p_k , then the *total revenue of trading, R*, can be defined by

$$R = \sum_{k=0}^{N} p_k x_k$$

Assuming a risk-neutral trader (practically does not care about variance of revenues), then we aim to maximize the expectation of our revenues by finding the "optimal" schedule, that in other words minimizes our price impact. Mathematically we aim to find,

$$arg_{x_0,..,x_N} maxE(R)$$

The problem is traditionally solved by assuming a price process and a functional form for the Temporary and Permanent Price Impact incurred from executing a stock. The solution can be then analytically derived through the use of dynamic programming methods among others. An obvious drawback to this is that the validity of the solution highly depends on the assumptions made.

The optimal trajectory under the above formulation (risk-neutral trader), is to sell the stock at a constant rate over the whole liquidation period. This strategy is known as Time Weighted Average Price (TWAP). It also assumes, through the price process assumption, that no information helps determine the direction of future price movement.

2.3 *Reinforcement Learning*

For outlining the Reinforcement Learning setting we mainly refer to (Stanford, 2017) and (DeepMind Technologies, 2013). As indicated in Figure 2, the Reinforcement Learning (RL) setting involves an agent interacting with an environment, by giving it an action, a_t, and then an environment, given an action from the agent, provides him back with a reward, r_t, and the next state s_{t+1}. Mathematically RL can be formulated through a Markov Decision process (MDP). The MDP satisfies the Markov property which says that the current state completely characterizes the state of the world, and can be formulated by the following set of variables (S,A,R,P,γ):

- S: set of possible states
- A: set of possible actions
- R (a,s): distribution of reward given state-action pair

- P(a,s): transition probability for next state given state-action pair
- γ: discount factor

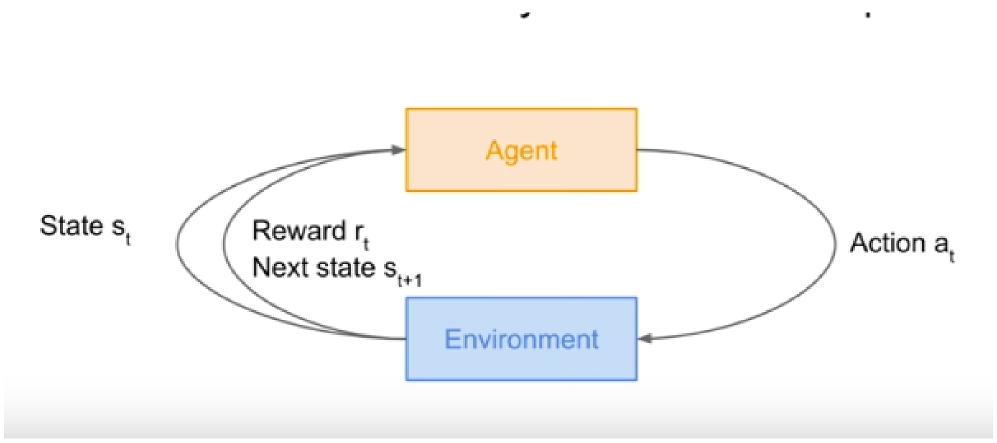


Figure 2 *Reinforcement Learning Setting. At each time = 0 the environment samples initial state s_0 ~ $p(s_0)$. Then for t= 0 until done, agent selects action a_t, environment samples reward r_t ~ R(. | s_t , a_t), and the next state s_{t+1} ~ P(.|s_t,a_t). The Agent receives reward r_t and next state s_{t+1}.*

Moreover, let a policy, π(s), be a function of S to A specifying what action to take at each state. The objective of a MDP is to find the policy π* that maximizes the cumulative discounted reward, $\sum_{t\geq 0} \gamma_t r_t$. Since rewards may involve randomness (e.g transition probabilities, rewards), formally we aim to find the following

$$\pi^* = \arg max_x E\left[\sum_{t\geq 0} \gamma_t r_t \mid \pi\right]$$

Note that one of the pros of using Reinforcement Learning is that the algorithm is model free.

2.4 *Deep Q-Learning (DQN)*

In Q-Learning, a specific RL approach, given an optimal policy π and a state-action pair, the Q-function is defined as follows,

$$Q^\pi(s,a) = E\left[\sum_{t\geq 0} \gamma_t r_t \mid s_0 = s, a_0 = a, \pi\right]$$

The optimal Q-value function Q*, is the maximum expected cumulative reward from a given state-action pair,

$$Q^*(s,a) = max_\pi E\left[\sum_{t\geq 0} \gamma_t r_t \mid s_0 = s, a_0 = a, \pi\right]$$

Q* satisfies the following Bellman equation:

$$Q^*(s,a) = max_\pi E[r + \gamma max_{a'} Q^*(s',a') \mid s,a]$$

This can be solved through the Value iteration algorithm which updates the above Bellman equation iteratively. The problem with this approach is that it's impractical, since it has to compute Q(s,a) for every state-action pair, making it computationally inefficient and thus affecting its scalability. Instead, a solution to this is using a functional approximator $\hat{Q}(s,a;\theta)$, to estimate Q(s,a) (DeepMind Technologies, 2013). In Deep Q-Learning this is a Neural Network (NN).

Specifically, when training the Neural Network, we aim to minimize the following Loss Function,

$$L_\iota(\theta_\iota) = E[(y_i - \hat{Q}(s,a;\theta_\iota)^2]$$

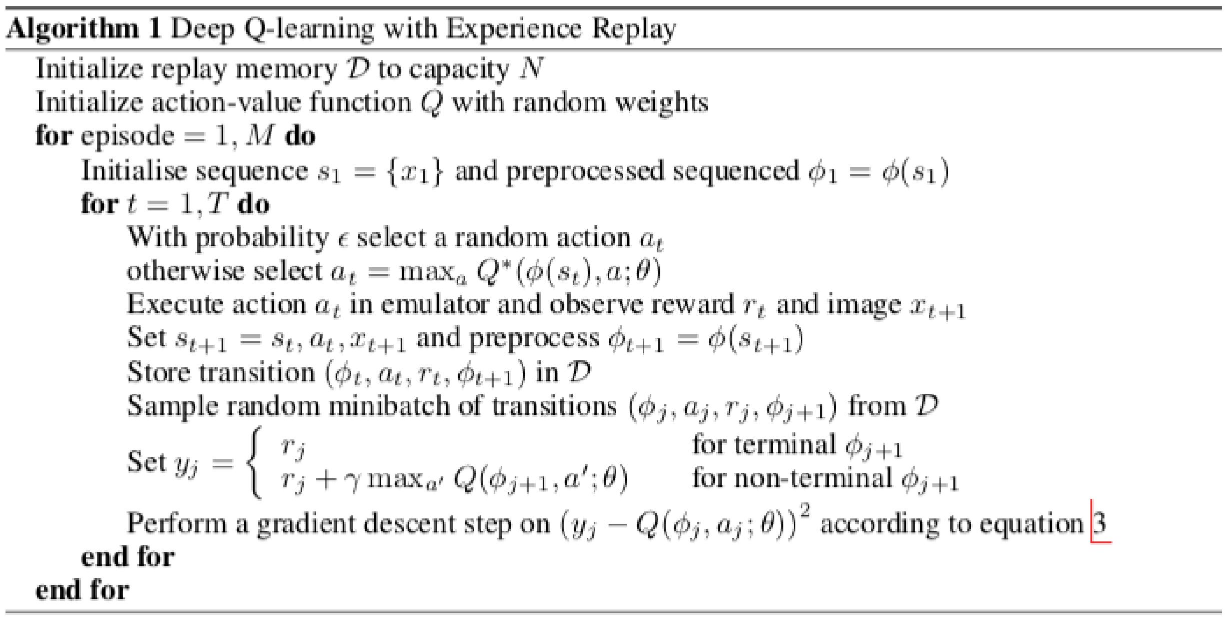
Where $y_i = E[r + \gamma max_{a'} Q^*(s',a';\theta_{\iota-1}) \mid s,a]$, is the "target" for iteration i, defined accordingly to the above Bellman equation. Note that it's parameters θ_{i-1} are held fixed from the previous iteration, when optimising L_i.

To optimize the above loss function stochastic gradient updates can be performed (Backpropagation), and the parameters of our network are updated at every time step.

Experience replay: The Neural Network, or Q-Network in this case, is given historical data for training through the Experience Replay method. In this method the agent's experiences are stored at each time step, $e_t = (s_t, a_t, r_t, s_{t+1})$ in the replay memory, D, gathered over many episodes played. Then the Neural Network is trained on a random mini batch from of the memory, instead of consecutive samples. Sampling randomly improves converge of the Q-Network by tackling problems such as consecutive samples are correlated, and biased feedback loops that might arise from the fact that the current Q-network parameters determine next training samples, and thus may be potentially dominated by similar actions.

E-greedy policy: The DQN learns about the optimal strategy $a = max_a\hat{Q}(s,a;\theta)$, while following an e-greedy policy. This means that according to a pre-assumed distribution (e.g. Uniform), dependent on a parameter ε, the agent performs random action, i.e. explores, or performs the optimal/greedy action, i.e. exploits. This ε parameter decays with a specified range making sure that as the agent learns more about the state-action space, it starts exploiting at a more frequent rate than exploring.

The pseudo-code presented in the Figure 3 below, summarizes all of the Deep Q-Learning procedure.



Algorithm 1 Deep Q-learning with Experience Replay

```
Initialize replay memory $\mathcal{D}$ to capacity $N$
Initialize action-value function $Q$ with random weights
for episode $= 1, M$ do
    Initialise sequence $s_1 = \{x_1\}$ and preprocessed sequenced $\phi_1 = \phi(s_1)$
    for $t = 1, T$ do
        With probability $\epsilon$ select a random action $a_t$
        otherwise select $a_t = \max_a Q^*(\phi(s_t), a; \theta)$
        Execute action $a_t$ in emulator and observe reward $r_t$ and image $x_{t+1}$
        Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$
        Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in $\mathcal{D}$
        Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from $\mathcal{D}$
        Set $y_j = \begin{cases} r_j & \text{for terminal } \phi_{j+1} \\ r_j + \gamma \max_{a'} Q(\phi_{j+1}, a'; \theta) & \text{for non-terminal } \phi_{j+1} \end{cases}$
        Perform a gradient descent step on $(y_j - Q(\phi_j, a_j; \theta))^2$ according to equation 3
    end for
end for
```

Figure 3 *Deep Q-Learning algorithm pseudo-code*

3 Methodology Specifications

We implement the methodology of the DQN on the Optimal Execution problem as outlined in the Part 3 of the problem. For outlining the specifications of our method followed below, we use the notation and terminology as defined in Part 3.

3.1 *Data and Feature Engineering*

We test our approach on the Apple (AAPL) stock, from 22-January-2019 to 02-August-2019, using the Close Price of the stock on every 10th minute of the trading day. In total this makes up to 5500 observations.

We split our dataset to training and validation. Training makes up the first 70% of the observations and Validation the last 30%. In training we update the weights of the Neural Network estimating the Q-function. In Validation we use fixed weights, and specifically the ones estimated at the last step of training, to test the performance of the DQN Out-Of-Sample, using the weights we expect to have helped the NN converge to the actual Q-function. Fixing the trained weights for validation is not a usual approach in RL, but since this is a special case where our state space is deterministic given the RL policy, we use this for evaluation purposes.

Lastly, we normalise our inputs (states of RL setting defined below) of the NN. We do this through a transformation of all the state variables, plus the price, to the [0,1] space using the transformation $\frac{x-\min(x)}{\max(x)-\min(x)}$, on each variable x. Normalisation is an important part in the ML pipeline since features with different scaling can affect negatively the performance of the algorithms, due to the level of sensitivity that the attributes exhibit while trying to determine their impact to the model. For example, features with larger absolute values might get prioritised by the ML algorithm without having greater prediction power.

3.2 *Reinforcement Learning Setting*

The values reported below represent the non-normalised version of them.

States: Inventory, Time

- Inventory: Set to $q_0 = 100$ shares. Only take discrete values $q_k = \{0,1,\dots,100\}$
- Time: Set to discrete values $t_k = \{0, 10, \dots, 500\}$, representing minutes, and with T = 500.

Action: Execute x_k amount of shares, where x is an integer. For simplicity we assume that only market orders to sell are possible.

- Actions are constraint to: $0 \leq x_k \leq q_k$, and $x_{T-1} = q_{T-1}$

Rewards: Represent the rewards from t=0 to the time constraint, T, of executing the order

(i) $R = \sum_{k=0}^{T} q_k(p_{k+1} - p_k) - a(x_k)^2$

where,

a = 0.01 and it intuitively represents the liquidity of the stock. The term $a(x_k)^2$ represents the penalty arising from the Price Impact of executing shares x_k. As denoted, it is assumed to be of a quadratic form. (Ning et al, 2018)

We also introduce and test an alternative approach to the reward function that intuitively seems to make sense. We represent rewards it in terms of the income received at each time point minus the Price Impact penalty,

(ii) $R = \sum_{k=0}^{T} x_k p_k - a(x_k)^2$

We then "calibrate" parameter a. First, we assume average price, p_k is 0.5 (remember variables have been scale to [0,1]) and plug it into the term in the summation. Then we find a such that the term in the summation is maximum for the value of x_k equal to TWAP strategy which is theoretically the optimal one under the states of only time and inventory. Specifically x_k=0.02 for T=50 and a normalised inventory, q_t, and this gives a value of $a = 12.5$.

Gamma, γ: We define γ to be 0.99.

3.3 *Neural Network Architecture*

We use an architecture which has a separate output unit for each possible action. The inputs to the network are only the state representations. The outputs correspond to the predicted Q-values of the individual action-state pair. The main advantage of this type of architecture is the ability to compute Q-values for all possible actions in a given state with only a single forward pass through the network. A graphical representation of a Neural Network is shown in Figure 4 below. Our specifications are as follows and we denote with "OR" the cases we test more than an specification,

- Structure: 6 Layers of 20 nodes each (Ning et al, 2018) OR 3 Layers with 20 nodes (Kelly et al, 2018). Output nodes = 100 (equal to the number of possible actions)
- Activation function: ReLU (output activation = linear)

We use the popular choice of ReLU as an activation function accepting the potential cost of a less flexible/non-linear NN output (compared to Sigmoid or Tanh), in order to gain in computation speed and avoid problems arising from vanishing gradients with such activation functions.

- Optimizer: Adam

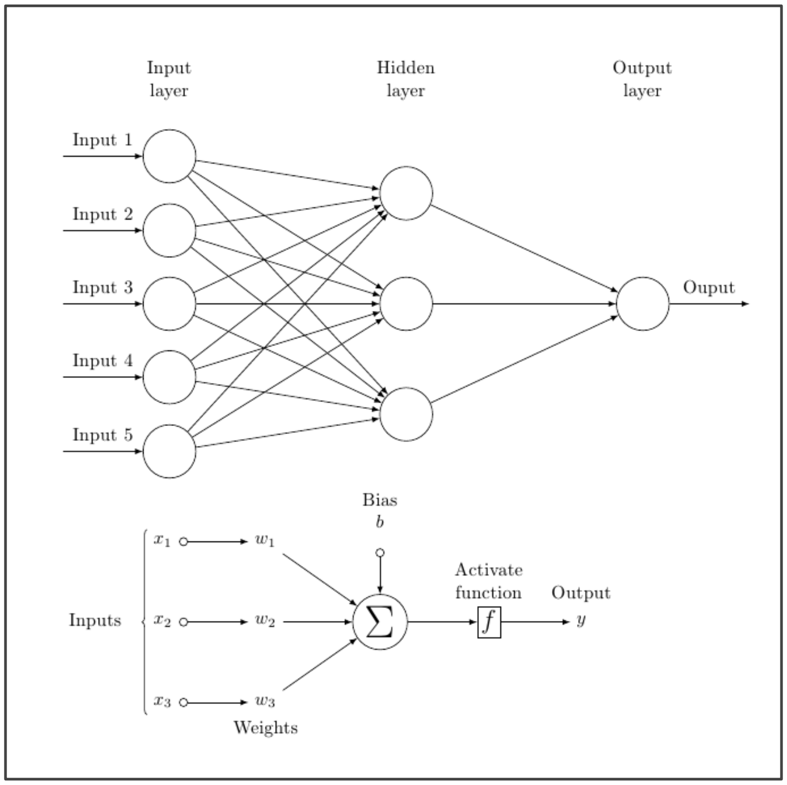


Figure 4 Graph of a Neural Network

3.4 *Training Neural Network*

For training we use the following specifications, with some being the ones required through the Keras library, Sequential() class:

- *Episodes* = 100
- *E-greedy, exploration action distribution:* $\varepsilon_t \sim Binomial(q_t,\ \frac{1}{T-t_k})$ OR $\varepsilon_t \sim Uniform \sim (0, q_t)$

The reasons for testing a Binomial distribution for the ε-Greedy exploration action, over the popular uniform, is that it selects on average the action that a TWAP strategy would select theoretically helping convergence of the network. (Ning et al, 2018)

- *Replay Memory* = 10,000
- *Epsilon = 1*
- *Epsilon decay* = 0.995 (Linear and begins decaying after we enter experience replay, i.e as soon as we have more memories than the batch size)
- *Learning rate* = 0.001
- *Batch size*: 32
- Pre-training of NN weights on Boundary cases chosen randomly: (i) hold the full inventory until

T-1 and then sell all shares at T, (ii) sell all shares at t=0

This is done in order to increase the stability of the network (Ning et al, 2018)

The rest of the above parameters are chosen according to common practice, including (DeepMind Technologies, 2013). No particular tuning of the parameters is made due to the training procedure of a RL setting.

3.5 *Evaluation Metric*

To evaluate the performance of our solution, we define *Profit and Loss with Transaction Cost (P&L)* computed over all episodes for each order as follows,

$$P\&L = \sum_{e=0}^{1000} \sum_{k=0}^{T} [x_k p_k - a(x_k)^2$$

As a measure of relative performance, we use the percentage point improvement of *P&L relative to TWAP*, defined as

$$\Delta P\&L = \frac{P\&L^{agent} - P\&L^{TWAP}}{P\&L^{TWAP}} \times 100$$

4 Results & Analysis

We first start by analysing the convergence of our Q-function approximator according to a few variations attempted. Our main tool for determining the quality of convergence is the average rewards over each episode. Also, we benchmark our optimal policy to the TWAP strategy which as explained earlier is what we would expect it to be. The below tests are performed on 100 episodes.

4.1 *Reward Functional Form: Rewards (i) vs Rewards (ii)*

From the Figure 5 and 6 below we can observe that Rewards (i) are more stable in terms of convergence of the Q-function compared to Rewards (ii). However, we continue with the Rewards (ii) since we find them more relevant to the problem and they still show some potential for convergence.

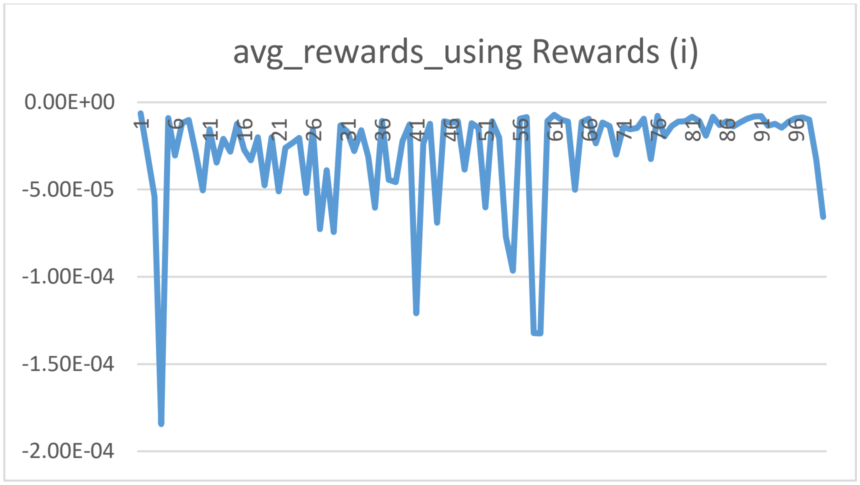


Figure 5 Average reward graph for the DQN with Reward function (i). Here the 3 Layer NN is used with a Binomial e-greedy. Y-axis represents rewards and x-axis number of episodes.

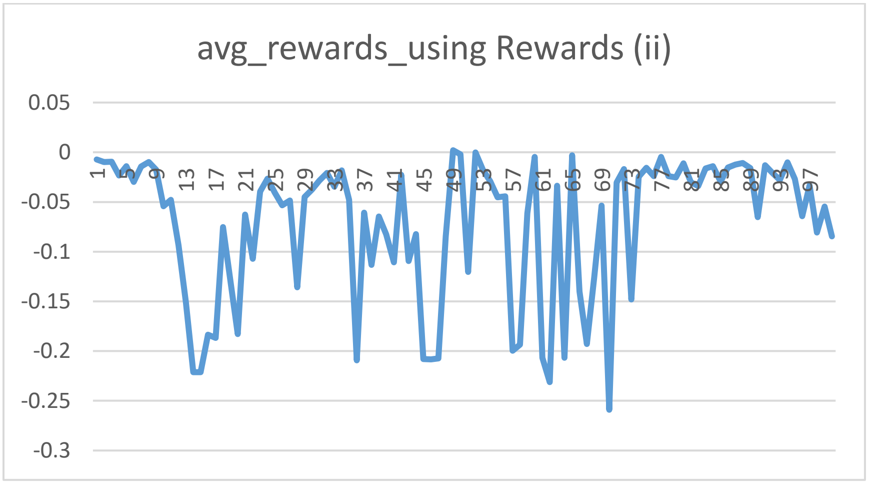


Figure 6 Average reward graph for the DQN with Reward function (ii). Same specification as Figure 5.

4.2 *Uniform vs Binomial e-greedy distribution*

As shown in the average reward graphs below, we can see that a uniform distribution for the e-greedy action, helps the agent converge quicker to higher levels of reward. We attribute this to the higher exploration rates that it gives in comparison to the Binomial. Since our state space is significantly larger than the (Ning et al, 2018), we find it reasonable that the Uniform seems to help the DQN perform better by exploring a greater range of states. We therefore choose the Uniform distribution for our specification.

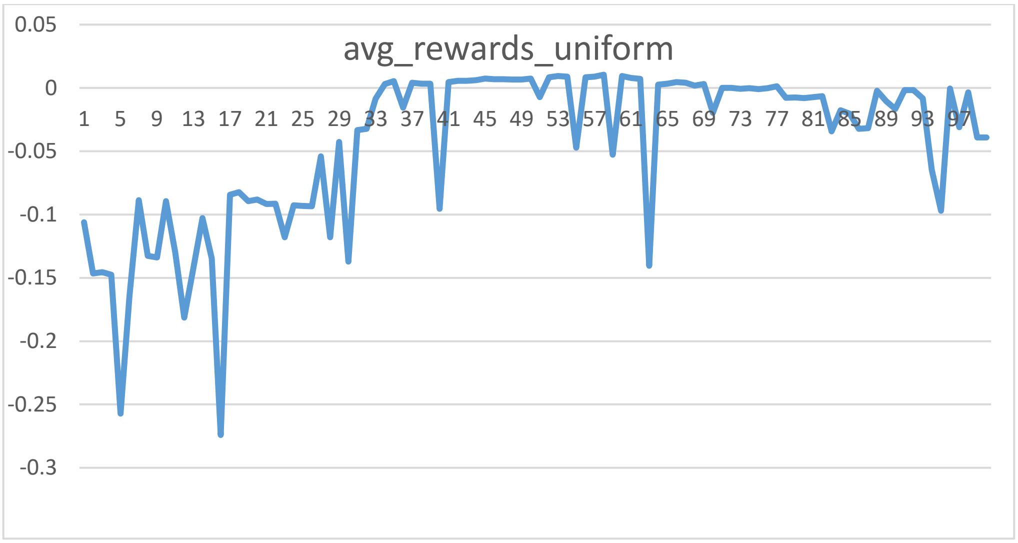


Figure 7 Average rewards using a Uniform distribution for the e-greedy action.

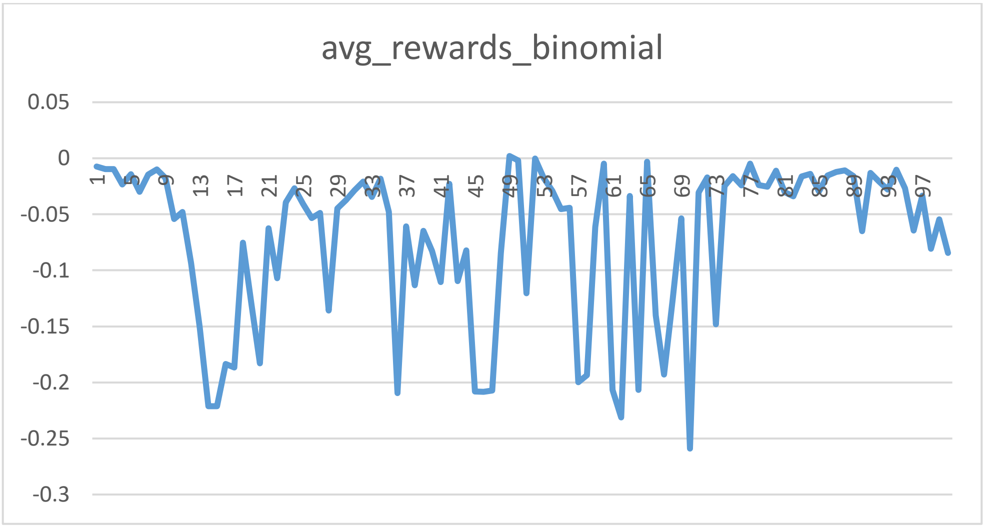


Figure 8 Average rewards using a Uniform distribution for the e-greedy action

4.3 *3 Layer Neural Network vs 6 Layer Neural Network*

The 3 Layer Neural Network seems to be doing better than the 6 Layer one. We expect the 6 Layer at least to more closely catch up with the 3 Layer one as we increase the number of episodes, since it has more weights to update, however the results here do not seem much more promising for the 6 Layer one over the 3, therefore we choose to continue with the 3 Layer one.

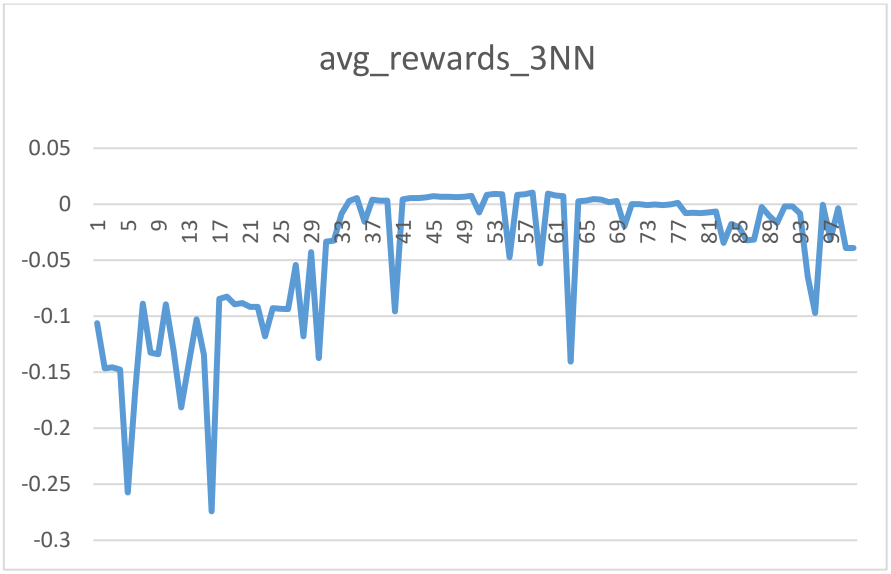


Figure 9 3 Layer Neural Network with 20 nodes. We use Rewards (ii) and Uniform e-greedy.

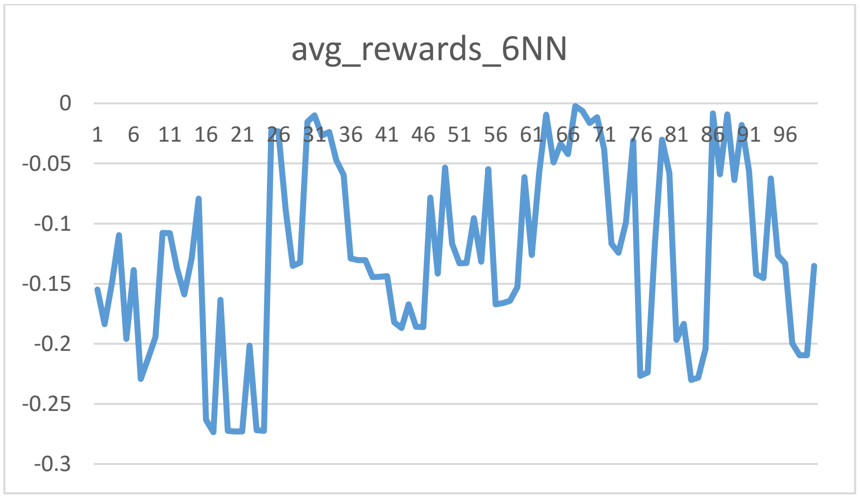


Figure 10 6 Layer Neural Network with 20 nodes. We use Rewards (i) and Uniform e-gredy.

4.4 *Evaluation*

Finally, we conclude according to the above that our best result is the 3 Layer Neural Network with Rewards (ii) and Uniform e-greedy. Further illustration of the convergence of this DQN follows in the Appendix. Below we present its execution path according to the Q-function approximator as estimated in the last step of the Training set. Figure 11, illustrates that our DQN agent has learnt a policy very close to the optimal TWAP. The two strategies only differ slightly in the first few steps of the execution and the last few ones.

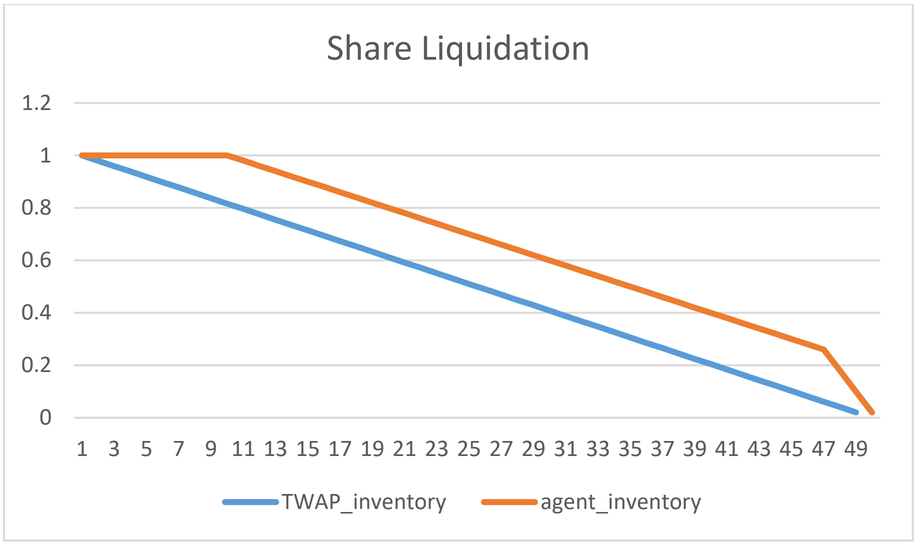


Figure 11 x-axis represents time (T=50) and y-axis amount of shares (normalised) left in the inventory.

Numerically, our agent performed 0.0005% more P&L relative to TWAP. We attribute the reason of slightly over performing the TWAP to the randomness of the stock price. The result is calculated over the Validation set which includes 34 episodes.

5 Conclusions & Next Steps

Through this paper we have examined the use of Reinforcement Learning in the Optimal Execution problem.

The results of this paper seem to reinforce the positive conclusions derived from the other relevant papers introduced, most notably the (Ning et al, 2018). Moreover, the paper has examined the problem using different data, different time periods and different specifications to the formulation of the problem. Some of the main differences in specifications include the use of a DQN instead of a Double DQN used by the paper mentioned (See Appendix for Double DQN performance snapshot), the use of a 3 Layer Neural Network over a 6 Layer one, incorporating the Uniform distribution for the e-greedy action in the problem, and finally formulating a different method for capturing the rewards.

Our Reinforcement Learning agent has managed to learn close enough what is mathematically derived to be the optimal solution to the problem under this setting, by simply interacting in the environment defined, and with only the minimum inputs of the setting. One of the most important implications of this is that it shows the potential of a RL agent to learn “optimal” strategies in more complex settings where there might not exist analytical solutions, or most importantly the setting is not fully understood by humans.

Some obvious extensions to our work could involve including more variables for the states, like returns, in order to test the RLs potential for returns prediction, and using actual LOB data that determine the Price Impact rather than assuming a penalty function for it.

6 References

Almgren, R. and Chriss, N. (2001). Optimal execution of portfolio transactions. *The Journal of Risk*, 3(2), pp.5–39.

Dabérius, K., Granat, E. and Karlsson, P. (2019). Deep Execution - Value and Policy Based Reinforcement Learning for Trading and Beating Market Benchmarks. *SSRN Electronic Journal*.

DeepMind Technologies, Mnih, V., Kavukcuoglu, K., Silver, D., Graves, A., Antonoglou, I., Wierstra, D. and Riedmiller, M. (2013). Playing Atari with Deep Reinforcement Learning.

Glantz, M., Kissell, R., Mun, J. and Paul, K. (2014). *Multi-asset risk modeling : techniques for a global economy in an electronic and algorithmic trading era*. Amsterdam: Academic Press, San Diego.

Gu, S., Kelly, B.T. and Xiu, D. (2018). Empirical Asset Pricing Via Machine Learning. *SSRN Electronic Journal*.

J.P. Morgan (2016). *Active Learning in Trading Algorithms*.

Kearns, M. and Nevmyvaka, Y. (2013). Machine Learning for Market Microstructure and High Frequency Trading.

Nevmyvaka, Y., Feng, Y. and Kearns, M. (n.d.). *Reinforcement Learning for Optimized Trade Execution*.

Ning, B., Ho, F., Ling, T. and Jaimungal, S. (2018). Double Deep Q-Learning for Optimal Execution

7 Appendix

7.1 *Convergence of DQN*

The below graph indicates the average rewards over 100 episodes our best variation of the DQN, as outlined in Part 4, and namely the 3 Layer NN, with Uniform e-greedy distribution, and Rewards (ii). The graph illustrates that the point reported in Part 4, as the optimal strategy of the DQN after training, seems to be pretty much consistent over longer episodes, namely 1000. This is an indication that it should be close to the real Q value, and in other words that are function approximator Q(s,a;θ), is near convergence.

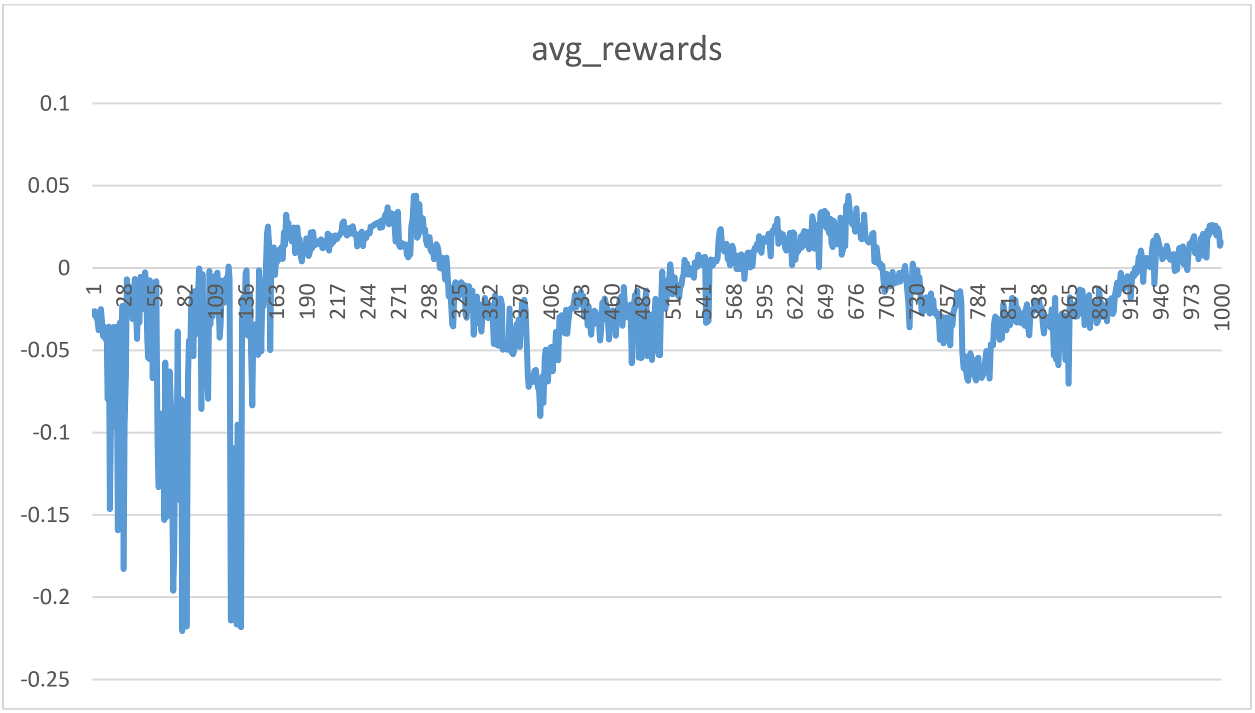


Figure 12 Average Rewards over 1000 episodes of training for our best variation of the DQN

7.2 *Double DQN performance snapshot*

Here we run a Double DQN method to compare the performance of our studied DQN. In Figure 12, we see the training performance of the Double DQN over 500 episodes and under the same specifications of our DQN: the 3 Layer NN, with Uniform e-greedy distribution, and Rewards (ii) and the rest as defined in Part 3. This is the version of the agent used by (Ning et al, 2018) with slightly different configurations. We see that the Double DQN does not seem to promise any better results than the simple DQN.

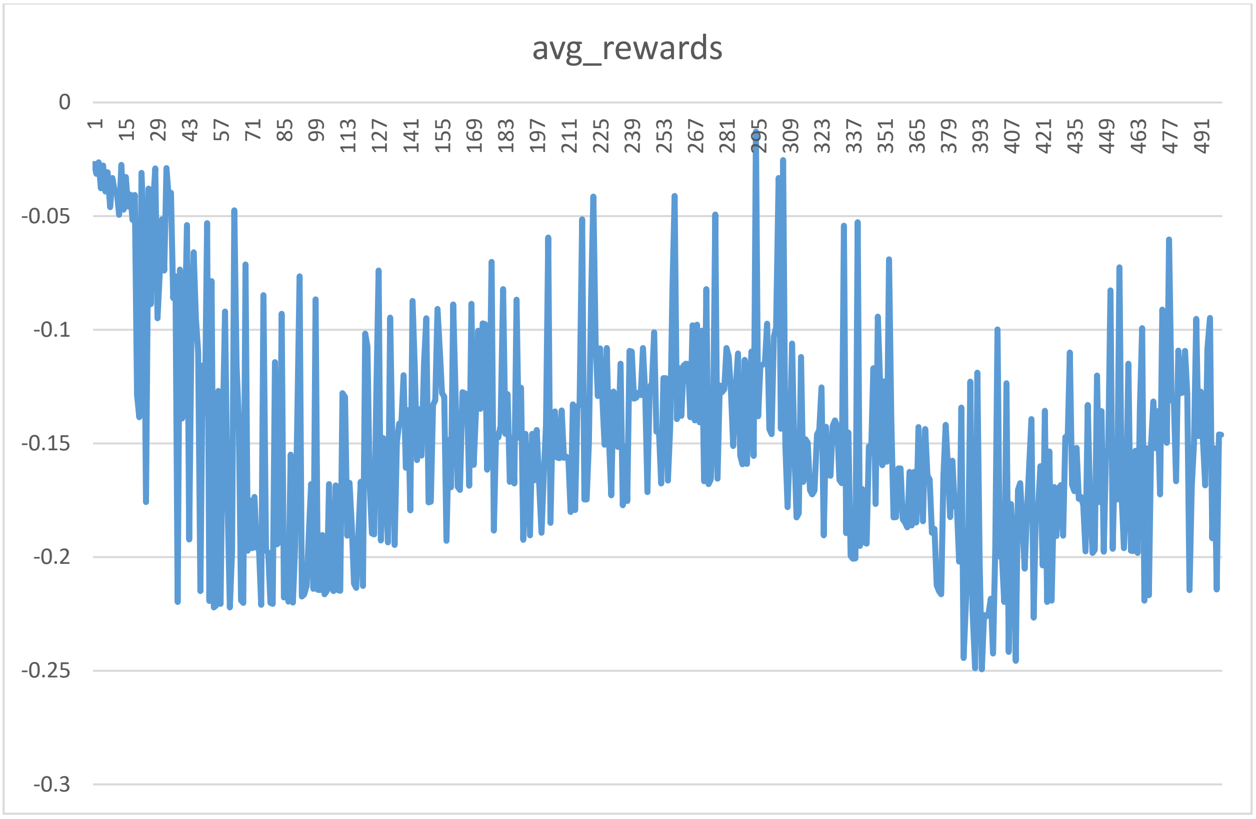


Figure 13 Double DQN average rewards per episode over 500 episodes of training

8 Code

```
import random
import numpy as np
from collections import deque
from keras.models import Sequential
from keras.layers import Dense
from keras.optimizers import Adam
import pandas as pd
import collections
from keras import backend as K
K.clear_session()


def normalise(x):
    return (x-min(x))/(max(x)-min(x))

#Code Controls
EPISODES = 1000
MEMORY = 10000
INITIAL_INVENTORY = 21
INITIAL_INVENTORY_SCALED = 1
TIME_CONSTRAINT_FOR_EXECUTION = 11
A = 0.01

TRAINING = True
TRAIN_BOUNDARIES= False
LOAD_PRETRAINED_WEIGHTS = True

PANDL_REWARD = False
EGREEDY = "Binomial"
NN = "NN=3_NN"
OPTIMIZER = 'RMSprop'
FILENAME = "{}ep_{}_{}A={}Actions={}TimeConstr={}Opt={}REW=".format(EPISODES,NN, EGREEDY,A,
INITIAL_INVENTORY, TIME_CONSTRAINT_FOR_EXECUTION, OPTIMIZER,str(PANDL_REWARD))
STATE_SIZE = 2

# DATA PREP
DF_RAW = pd.read_csv('./Data_Sets/APPL10minTickData.csv', header=0)
EXECUTION_TIMES_SCALED = normalise(np.array(range(TIME_CONSTRAINT_FOR_EXECUTION)))
TIME_CONSTRAINT_FOR_EXECUTION_SCALED = max(EXECUTION_TIMES_SCALED)
TIME_POINTS_FOR_EXECUTION = len(EXECUTION_TIMES_SCALED)
TIME_UNIT = 1/(TIME_POINTS_FOR_EXECUTION-1)
TRAIN_OBSERVATIONS = int(0.7 * len(DF_RAW))
if TRAINING == True:
    DF = DF_RAW.iloc[:TRAIN_OBSERVATIONS, ]
    REAL_TIME = 0
    END_TIME = len(DF)
else:
    DF = DF_RAW.iloc[TRAIN_OBSERVATIONS:, ]
    REAL_TIME = TRAIN_OBSERVATIONS
    END_TIME = TRAIN_OBSERVATIONS + len(DF)
    EPISODES = 34

PRICES = np.array(DF['close'].to_numpy())
PRICES = normalise(PRICES)

def run():
    """
    Main Function
    Output: tables of the results and the weights of the trained agent
    """
    initial_state = State(inventory=INITIAL_INVENTORY_SCALED, time=0)
    env = Env(initial_state, PRICES, TIME_CONSTRAINT_FOR_EXECUTION_SCALED,
TIME_POINTS_FOR_EXECUTION, REAL_TIME, END_TIME)

    state_size = STATE_SIZE
    action_size = INITIAL_INVENTORY
    agent = DQNAgent(state_size, action_size, TRAINING)
    if TRAINING != True:
        agent.load("{}_weights.h5".format(FILENAME))
    (TRAINING == True and LOAD_PRETRAINED_WEIGHTS == True)


        agent.load("100ep_NN=3_NN_UniformA=0.01Actions=21TimeConstr=11Opt=RMSpropREW=_weights.h5")
        twap = TWAP(INITIAL_INVENTORY_SCALED, TIME_POINTS_FOR_EXECUTION)

        done = False
        batch_size = 32

        PandL_agent_array = np.array([])
        PandL_TWAP_array = np.array([])
        PandL_vs_TWAP_array = np.array([])
        rewards_array = np.array([])
        avg_rewards = np.array([])

        for e in range(EPISODES):
            state = env.reset_game()
            state = np.reshape(state.state_as_list(), [1, state_size])
            print("REAL TIME is: " + str(env.real_time))
            print("start episode:" + str(e))
            for time in EXECUTION_TIMES_SCALED:
                print("inventory is: " + str(env.state.inventory))
                print("time is: " + str(env.state.time))
                if TRAIN_BOUNDARIES == True:
                    if time == 0:
                        index = np.random.binomial(1, 1 / 2)
                    action = agent.act_boundary_conditions(state, time, env.time_constraint_for_execution, index)
                else:
                    action = agent.act(state, time, env.time_constraint_for_execution)
                next_state, reward, done = env.step(action)
                next_state = next_state.state_as_list()
                next_state = np.reshape(next_state, [1, state_size])
                agent.remember(state, action, reward, next_state, done)
                state = next_state
                PandL_agent_array = np.append(PandL_agent_array, env.PandL(action))
                PandL_TWAP_array = np.append(PandL_TWAP_array, env.PandL(twap.act()))
                PandL_vs_TWAP_array = np.append(PandL_vs_TWAP_array, ((PandL_agent_array[env.real_time-1-
        REAL_TIME] - PandL_TWAP_array[env.real_time-1- REAL_TIME]) / PandL_TWAP_array[env.real_time-1-
        REAL_TIME]) * 100)
                rewards_array = np.append(rewards_array, reward)

                if done:
                    print("DONE")
                    print("episode: {}/{}, P&L_vs_TWAP: {}%, time: {}, e: {:.2}".format(e, EPISODES,
        PandL_vs_TWAP_array[env.real_time-1-REAL_TIME], time, agent.epsilon))
                    avg_rewards = np.append(avg_rewards, np.mean(rewards_array))
                    rewards_array = np.array([])
                    break
                if (len(agent.memory) > batch_size and TRAINING == True):
                    agent.replay(batch_size)

        total_PandL_agent = sum(PandL_agent_array)
        total_PandL_TWAP = sum(PandL_TWAP_array)
        PandL_vs_TWAP = ((total_PandL_agent-total_PandL_TWAP)/total_PandL_TWAP)*100
        print("PandL_vs_TWAP is {}%".format(PandL_vs_TWAP) )

        avg_rewards_df = pd.DataFrame({'avg_rewards': avg_rewards})
        print("Avg_rewards are {}".format(avg_rewards))
        avg_rewards_df.to_csv('avg_rewards_{}Train={}NN={}REW=.csv'.format(EPISODES, TRAINING,NN,
        str(PANDL_REWARD)))

        agent.save("{}_weights.h5".format(FILENAME), PandL_vs_TWAP_array,PandL_agent_array,
        PandL_TWAP_array)



        class State(object):


          def __init__(self, time, inventory):
        self.time = time # time period in an episode (integer)
        self.inventory = inventory # number of shares yet to be executed

          def state_as_list(self):
        # return the state in the correct format
        return [self.time, self.inventory]

        class TWAP(object):
        """
        Class defining the TWAP strategy
        """

        def __init__(self, initial_inventory, time_points_for_execution):
            self.initial_inventory = initial_inventory
            self.time_points_for_execution = time_points_for_execution
        def act(self):
            action = self.initial_inventory / self.time_points_for_execution
            return action

        class Env(object):
          """
          Class of the environment
          """
          def __init__(self, state, prices, time_constraint_for_execution, time_points_for_execution, real_time, end_time):
        self.state = state
        self.prices = prices
        self.time_constraint_for_execution = time_constraint_for_execution
        self.time_points_for_execution = time_points_for_execution #number of discrete time points where execution
        can happen within an episode
        self.real_time = real_time #time period along the whole of our data set (does not reset at each episode)
        self.end_time = end_time #final time of our dataset

          def reset_game(self):
        self.state.inventory = INITIAL_INVENTORY_SCALED
        self.state.time = 0
        return self.state

          def step(self, action):
        '''
        - step function should take an action and return the next state, the reward and done (done would mean that
        the episode has finished)
        - action is an integer number of shares bought in one period
        '''
        reward = self.reward(self.state.inventory, action)
        self.state.time = round(self.state.time + TIME_UNIT, 2)
        self.real_time = (self.real_time + 1) if (self.real_time +1) % self.end_time != 0 else REAL_TIME
        self.state.inventory = round(self.state.inventory - action,2)
        return (self.state, reward, self.is_done())

          def is_done(self):
        '''
         Determines when an episode ends. Returns boolean
        '''
        return self.state.time == self.time_constraint_for_execution

          def get_price(self):
        return self.prices[self.real_time - REAL_TIME]

          def reward(self, remaining_inventory, action):
              if PANDL_REWARD == True:
                  return action*self.get_price() - 2.5*(action**2)
              return remaining_inventory*(self.get_price() -  self.get_price()) - A*(action**2)

          def PandL(self, action):
              return action*self.get_price() - A*(action**2)

        class DQNAgent:
        """
```

```
DQN Agent
Takes as inputs state_size, action_size and TRAINING.
state_size: Number of different state variables
action_size: number of different possible actions
TRAINING: (Bool) Whether the DQN Agent is being Trained (True) or Validated (False)
"""
def __init__(self, state_size, action_size, TRAINING):
    self.state_size = state_size
    self.action_size = action_size
    self.memory = deque(maxlen=MEMORY)
    self.gamma = 0.99 # discount rate
    self.epsilon = 1.0  if TRAINING else 0.0
    self.epsilon_min = 0.01 if TRAINING else 0.0
    self.epsilon_decay = 0.995
    self.learning_rate = 0.001
    self.model = self._build_model()

def _build_model(self):
    # Neural Net for Deep-Q learning Model
    model = Sequential()
    model.add(Dense(20, input_dim=self.state_size, activation='relu'))
    model.add(Dense(20, activation='relu'))
    model.add(Dense(20, activation='relu'))
    model.add(Dense(self.action_size, activation='linear'))
    model.compile(loss='mse',
                  optimizer=OPTIMIZER)
    return model

def remember(self, state, action, reward, next_state, done):
    self.memory.append((state, action, reward, next_state, done))

def act(self, state, time, time_constraint_for_execution):
  '''
  Determines the actions of the agent
  '''
  inventory = state[0][1]

  if time == (time_constraint_for_execution - TIME_UNIT):
    action = inventory
    print("last action is " + str(action))
  elif inventory == 0:
    action = 0 #to make it's time consistent with TWAP for comparison purposes
  elif np.random.rand() <= self.epsilon:
      if EGREEDY == 'Binomial':
          n = inventory*INITIAL_INVENTORY
          p = TIME_UNIT/(time_constraint_for_execution - time)
          action = np.random.binomial(n, p)
          action =  np.linspace(0,1,INITIAL_INVENTORY)[action] #scale back action to a normalised action
          print("E-greedy action " + str(action))
      elif EGREEDY == 'Uniform':
          action = random.randrange(self.action_size)
          action = np.linspace(0, 1, INITIAL_INVENTORY)[action]
          print("E-greedy action " + str(action))
  else:
      act_values = self.model.predict(state)
      action = np.argmax(act_values[0])/(INITIAL_INVENTORY-1)
      print("Optimal action " + str(action))

  if action > inventory:
      action = inventory
      print("action>intentory action is " + str(action))
  return round(action,2)

def act_boundary_conditions(self, state, time, time_constraint_for_execution, index):

    if index == 0:
        if time == (time_constraint_for_execution - TIME_UNIT):
            action = INITIAL_INVENTORY_SCALED
        else:
            action = 0
    elif index == 1:
        if time == 0:
            action = INITIAL_INVENTORY_SCALED
        else:
            action = 0

    return round(action,2)

def replay(self, batch_size):
  '''
    replay uses the memory values to train the network
  '''
  minibatch = random.sample(self.memory, batch_size)
  for state, action, reward, next_state, done in minibatch:
      target = reward
      if not done:
          target = (reward + self.gamma *
                    np.amax(self.model.predict(next_state)[0]))
      target_f = self.model.predict(state)
      action_range = np.linspace(0, 1, INITIAL_INVENTORY)
      action_index = [i for i in range(len(action_range)) if round(action_range[i].item(),2) == action][0]
      target_f[0][action_index] = target
      self.model.fit(state, target_f, epochs=1, verbose=0)
  if self.epsilon > self.epsilon_min:
      self.epsilon *= self.epsilon_decay

def load(self, name):
    '''
    Loads the weights of a trained agent
    '''
    self.model.load_weights(name)

def save(self, name, PandL_vs_TWAP, PandL_agent, PandL_TWAP):
    '''
    Saves the weights of the trained agent and our results to csvs
    '''
```

```
        self.model.save_weights(name)
        state_time = np.array([item[0][0][0] for item in list(self.memory)])
        state_inventory = np.array([item[0][0][1] for item in list(self.memory)])
        actions = np.array([item[1] for item in list(self.memory)])
        rewards = np.array([item[2] for item in list(self.memory)])
        next_state_time = np.array([item[3][0][0] for item in list(self.memory)])
        next_state_inventory = np.array([item[3][0][1] for item in list(self.memory)])
        done = np.array([item[4] for item in list(self.memory)])
        memory_df = pd.DataFrame({'state_time': state_time, 'state_inventory': state_inventory, 'action': actions,
'reward': rewards,'next_state - Inventory': next_state_time, 'next_state - Time': next_state_inventory,
'done':done})
        PandL_df = pd.DataFrame({'PandL_vs_TWAP': PandL_vs_TWAP, 'PandL_agent': PandL_agent,
'PandL_TWAP': PandL_TWAP})
        print(memory_df)
        memory_df.to_csv('Memory_{}Train={}.csv'.format(EPISODES,TRAINING))
        PandL_df.to_csv('P&L_{}Train={}.csv'.format(EPISODES, TRAINING))


if __name__ == "__main__":
    run()
```