Capstone Project

Machine Learning Engineer Nanodegree

By Kai Xi

Reinforcement Learning in Security Trading

I. Project Definition:

Project Overview

The financial market has always been a field of interest when it comes to computer technologies. Computerization of the order flow in the financial markets started as early as 1970s. Firms and individuals have long developed algorithms to trade financial instruments. With the recent advances in AI field, I was surprised to find only very limited publications on trading in relation to machine learning. The biggest reason probably lies in the nature of the business. The end goal of trading activity is about profits. If a trading strategy is proven valid, the creator has strong incentives not to make it public knowledge. A wide application of a strategy would necessarily make it less profitable.

I have been actively trading stocks for the past three years. The reason I started learning data analysis was to conduct more quantitative research. It eventually led to machine learning which I have found fascinating. I am very interested in finding out if how applicable machine learning is in this domain. It is a enormous task. I hope this can be a good start.

Trading, in its most basic form, is just buying and selling financial instruments, generally with the hope of making profits. Many professional traders have akin trading to doing sports - making the perfect trade is much like making the perfect swing while playing tennis. It works to the computer’s benefit that most inputs related to trading are quantifiable, if not explicitly just numbers. I believe with the right inputs, computer can learn to outperform humans. This project is an attempt to implement reinforcement learning in the setting of security trading under simplified environment.

I have researched extensively on the topic. Nothing came up that had the architecture to allow the flexibility that I want for tweaking. Below are a few links that I have looked into:
https://github.com/samre12/deep-trading-agent
https://launchpad.ai/blog/trading-bitcoin
https://github.com/lefnire/tforce_btc_trader

Problem Statement

The question posed here can potentially be huge. There are countless events driving the continuous and instantaneous ebbs and flows in hundreds of thousands of financial instruments. This project does not aim to produce a profitable trading system using reinforcement learning, but merely **construct a basic framework** (or recreate an existing framework) for trading environment.

This framework design, on the algorithm level, would include the definition of state space, specification of actions and rewards. All these elements need to go seamless together in the script so that the learning agent can be designed to learn to make trading decisions (through gathering and mapping values to its actions with relevant inputs).

Once the algorithm is designed, data needs to be gathered and processed so that it can be fed to model. The model will use the processed data to simulate trading activities over many, many iterations, get feedback and improve on itself. This entire framework, once set up, can be used to facilitate to future research and testing.

Metrics:

1. A functioning reinforcement model with complete end-to-end environment, reward and agent design is constructed - it runs with no scripting errors
2. Data are properly processed and fed into the model for training.
3. The learning agent will be trained with data through trials. An optimal policy is attained.
4. The returns (profits and losses from trading activities) generated using this learned optimal policy will then be recorded and evaluated.
5. The model performance (from the learned policy) will be benchmarked against zero return - rationale for this is explained in detail in the following section.

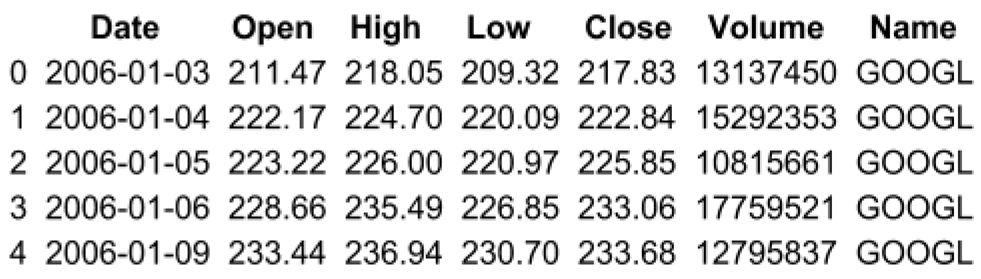
II. Data Analysis:

For the purpose of this project, daily stock data for Google is used (range from 2006-01-01 to 2018-01-01). Data files are downloaded from Kaggle (https://www.kaggle.com/szrlee/stock-time-series-20050101-to-20171231)

The original files have the following columns:

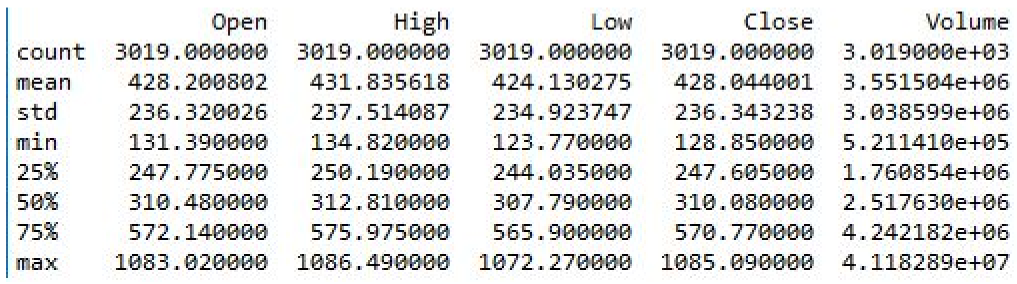
- **Open** - Price of the stock at market open (this is NYSE data so all in USD)
- **High** - Highest price reached in the day
- **Low** - Lowest price reached in the day
- **Close** - Price of the stock at market close
- **Volume** - Number of shares traded
- **Name** - The stock's ticker name

Below is a sample of the dataset:



	Date	Open	High	Low	Close	Volume	Name
0	2006-01-03	211.47	218.05	209.32	217.83	13137450	GOOGL
1	2006-01-04	222.17	224.70	220.09	222.84	15292353	GOOGL
2	2006-01-05	223.22	226.00	220.97	225.85	10815661	GOOGL
3	2006-01-06	228.66	235.49	226.85	233.06	17759521	GOOGL
4	2006-01-09	233.44	236.94	230.70	233.68	12795837	GOOGL

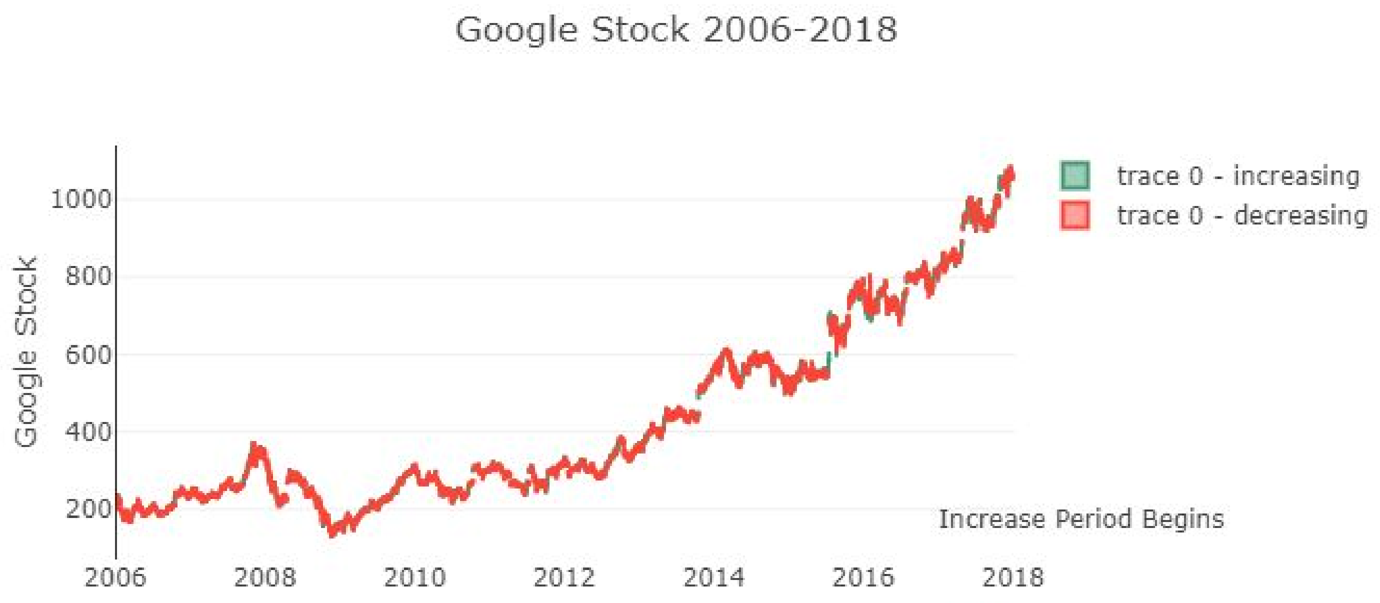
The descriptive stats are as follow:



```
              Open         High          Low        Close        Volume
count  3019.000000  3019.000000  3019.000000  3019.000000  3.019000e+03
mean    428.200802   431.835618   424.130275   428.044001  3.551504e+06
std     236.320026   237.514087   234.923747   236.343238  3.038599e+06
min     131.390000   134.820000   123.770000   128.850000  5.211410e+05
25%     247.775000   250.190000   244.035000   247.605000  1.760854e+06
50%     310.480000   312.810000   307.790000   310.080000  2.517630e+06
75%     572.140000   575.975000   565.900000   570.770000  4.242182e+06
max    1083.020000  1086.490000  1072.270000  1085.090000  4.118289e+07
```

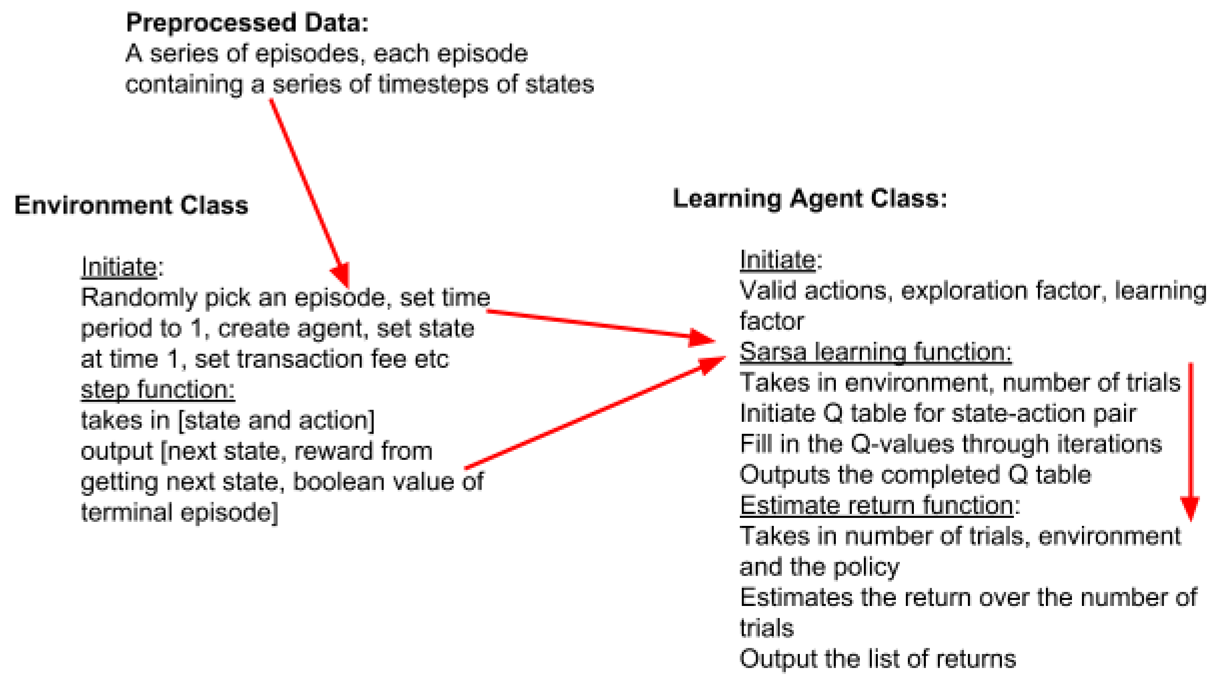
The dataset is very clean. No abnormalities are identified. All the prices have been adjusted for dividends and stock splits so that there is no sudden shift in price.

The plot below is created with the data file using plotly:



Algorithms and techniques:

To summarize, a series of states consisting of trading signals, daily closing prices will be constructed. Episodes will be formed as certain number of states in chronological order.



Preprocessed Data:
A series of episodes, each episode containing a series of timesteps of states

Environment Class

Initiate:
Randomly pick an episode, set time period to 1, create agent, set state at time 1, set transaction fee etc
step function:
takes in [state and action]
output [next state, reward from getting next state, boolean value of terminal episode]

Learning Agent Class:

Initiate:
Valid actions, exploration factor, learning factor
Sarsa learning function:
Takes in environment, number of trials
Initiate Q table for state-action pair
Fill in the Q-values through iterations
Outputs the completed Q table
Estimate return function:
Takes in number of trials, environment and the policy
Estimates the return over the number of trials
Output the list of returns

From episodes, the agent will learn to take actions from its observable input values (states). The way the agent does that is first initiating a table for each state-action pair. The reward that the agent learns at the end of each episode will be used to update the values in the state-action pair accordingly, at the learning rate we set.

$$Q_{t+1}(s_t, a_t) \leftarrow \underbrace{Q_t(s_t, a_t)}_{\text{old value}} + \underbrace{\alpha_t(s_t, a_t)}_{\text{learning rate}} \cdot \left(\overbrace{\underbrace{R_{t+1}}_{\text{reward}} + \underbrace{\gamma}_{\text{discount factor}} \underbrace{\max_a Q_t(s_{t+1}, a)}_{\text{estimate of optimal future value}}}^{\text{learned value}} - \underbrace{Q_t(s_t, a_t)}_{\text{old value}} \right)$$

The details of the algorithm design will be in the following “**Methodology**” section.

Benchmark:

The structure will be used as a baseline for future idea/parameter testing. The product of this project is more or less a blank page in terms of testing real inputs - the performance results would be benchmarked against zero return (the return of do-nothing policy). That is to say, because the agent has the choice of doing nothing, in which case it receives zero reward - it should learn to do at least that. Anything significantly (statistically speaking) below zero would indicate that the agent cannot learn properly.

III. Methodology:

Data Prepossessing:

As previously discussed, we want to create a list of states and a list episodes, each containing 10 sequential state from the data file. Each state should include the closing price (for reward calculation), observable state (indicators of closing price's relative position to long/short term low and high price.

Starting at the 21st (long period + 1) row of the data, for each closing price, the long term high and low, short term long and low are computed as following:

```
for i in range(trading_period_long-1, len(df)):

    list_high_short = []
    list_low_short = []

    for k in range(i-trading_period_short+1, i):
        list_high_short.append(df['High'][k])
        list_low_short.append(df['Low'][k])

    list_high_long = []
    list_low_long = []

    for k in range(i-trading_period_long+1, i):
        list_high_long.append(df['High'][k])
        list_low_long.append(df['Low'][k])

    closing_price = df['Close'][i]
    short_period_high = max(list_high_short)
    short_period_low = min(list_low_short)

    long_period_high = max(list_high_long)
    long_period_low = min(list_low_long)
```

Then the long term and short term indicators are computed as described in 'State' section above. These along with the closing price are put into a dictionary and append to the list of dictionary <states>.

Expressed in syntax:

```
states.append({'Close': closing_price,
               'ST Relative Indicator': position_relative_short,
               'LT Relative Indicator': position_relative_long})
```

The states are then grouped sequentially, 10 (max_time) at a time

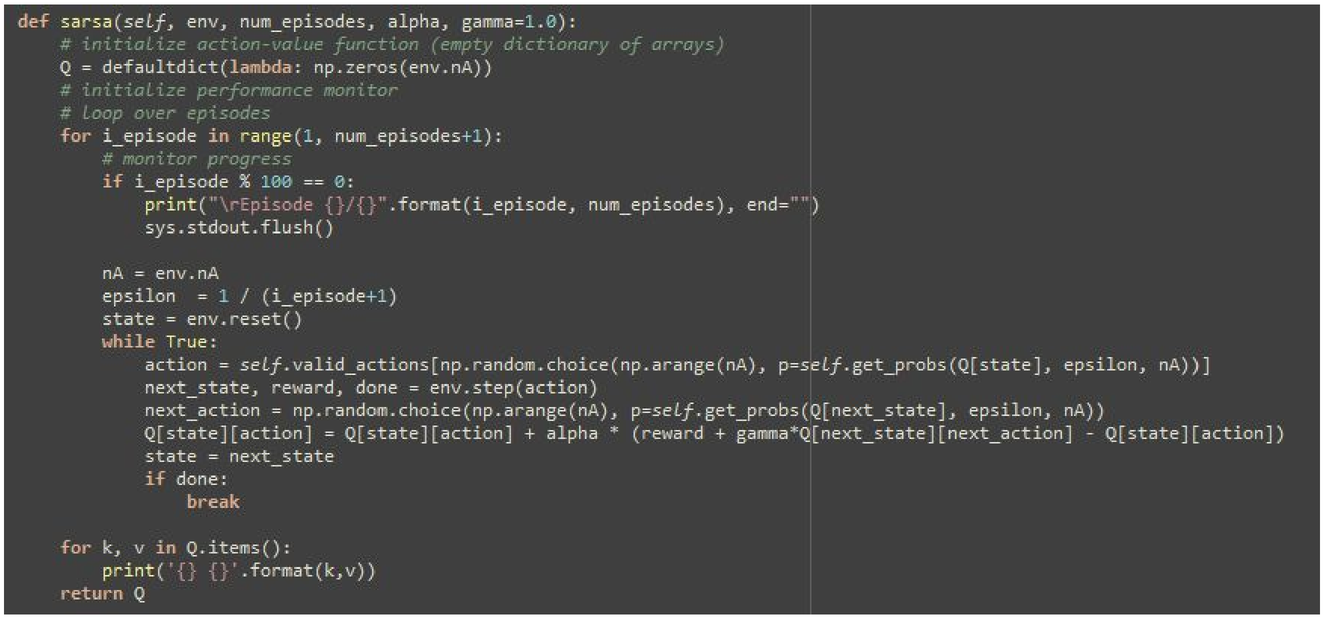
```
Episodes = []
for i in range (0, len(states)-max_time):
    episodes.append(states[i:i+max_time])
```

Environment Design & Implementations:

Now we need to take look at how we can set up the environment in order to apply reinforcement learning to this dataset. I have attempted various structures to set up the environment, the learning agent and how they interact. In the following part, the final version is discussed.

We will in turn discuss how the states, the actions, the rewards and the episodes can be constructed. For each section, the discussion of environment design element is follow by its implementation in codes.

Below is the core part of the Sarsa learning function:



```
def sarsa(self, env, num_episodes, alpha, gamma=1.0):
    # initialize action-value function (empty dictionary of arrays)
    Q = defaultdict(lambda: np.zeros(env.nA))
    # initialize performance monitor
    # loop over episodes
    for i_episode in range(1, num_episodes+1):
        # monitor progress
        if i_episode % 100 == 0:
            print("\rEpisode {}/{}".format(i_episode, num_episodes), end="")
            sys.stdout.flush()

        nA = env.nA
        epsilon  = 1 / (i_episode+1)
        state = env.reset()
        while True:
            action = self.valid_actions[np.random.choice(np.arange(nA), p=self.get_probs(Q[state], epsilon, nA))]
            next_state, reward, done = env.step(action)
            next_action = np.random.choice(np.arange(nA), p=self.get_probs(Q[next_state], epsilon, nA))
            Q[state][action] = Q[state][action] + alpha * (reward + gamma*Q[next_state][next_action] - Q[state][action])
            state = next_state
            if done:
                break

    for k, v in Q.items():
        print('{} {}'.format(k,v))
    return Q
```

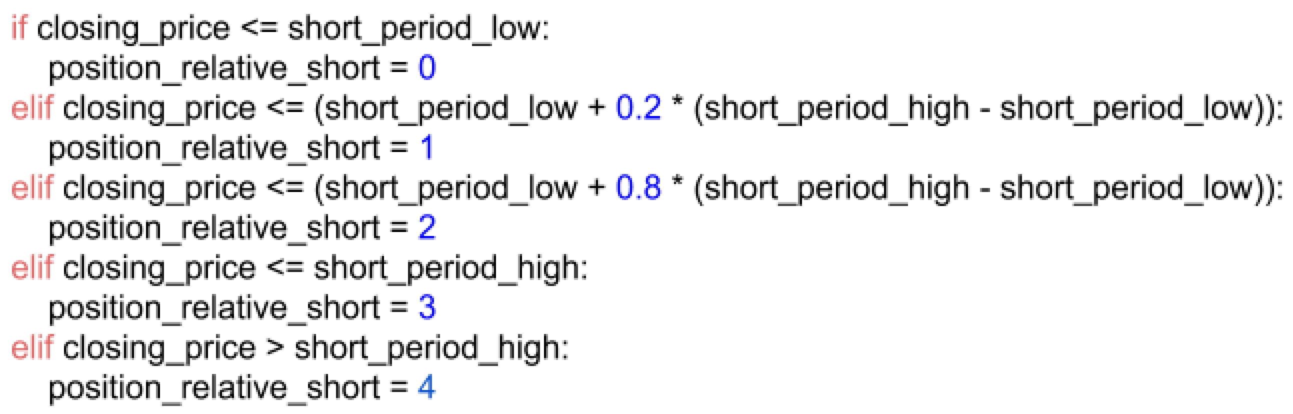
State:
The first step of the process is to come up a state structure that can:
1. reasonably represents important trading inputs
2. Is simple enough such that the number of state spaces is limited - for consideration of computing time and capacity

This is not an easy process. This step requires significant domain knowledge. First of all, prices and time are continuous. In real life situations, there is neither single time step nor sudden shift in price levels (or rarely so). Limited by my knowledge, discretization seems to be the natural way to go. Secondly, the inputs together need to be exhaustive of all possibility. Thirdly, it should not be based on absolute value of time and price, for that we want the model to be generally applicable.

With these consideration in mind, below are the state inputs and their rationale accordingly:

- Stock's relative price level to the short term trading range
  [0, 1, 2, 3, 4]
- Stock's relative price level to the long term trading range
  [0, 1, 2, 3, 4]
- Boolean indicator of whether there is a stock position
  [0, 1]
- Deadline: time until the max period is reached, assuming 10 time periods per episode
  [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

For example, below is the short term indicator expressed in syntax:

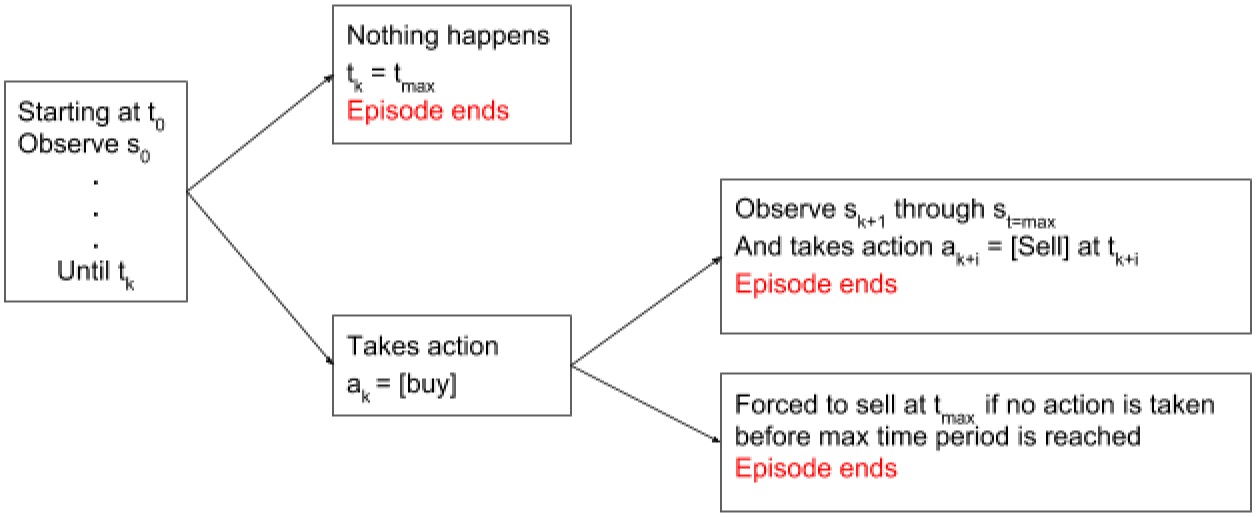


```
if closing_price <= short_period_low:
    position_relative_short = 0
elif closing_price <= (short_period_low + 0.2 * (short_period_high - short_period_low)):
    position_relative_short = 1
elif closing_price <= (short_period_low + 0.8 * (short_period_high - short_period_low)):
    position_relative_short = 2
elif closing_price <= short_period_high:
    position_relative_short = 3
elif closing_price > short_period_high:
    position_relative_short = 4
```

Action:
Actions are the easier to conceptualize. At each time period, the agent simply decides to buy, sell or hold. For the basic model, we assume all-or-nothing for stock and cash positions.

- Buying: converts all cash to stock at the closing price of the time period
- Selling: converts all stock to cash at the closing price of the time period
- Holding: cash or stock position remains the same

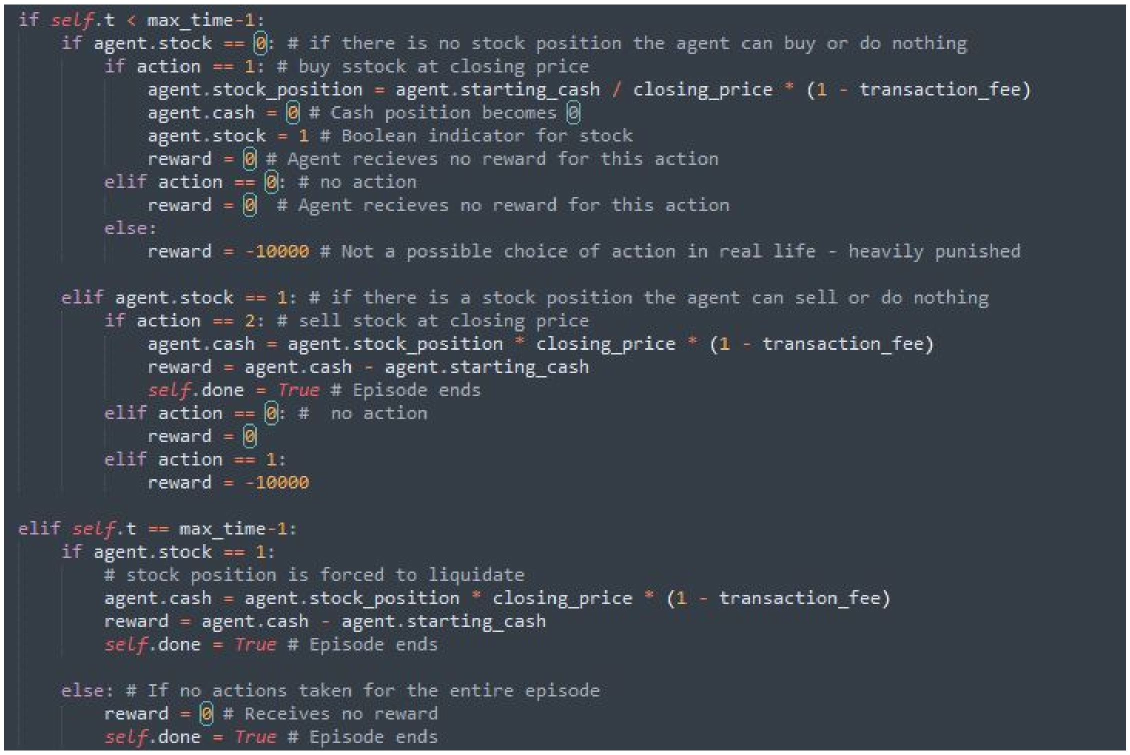
Episode:
For simplification, we will define an episode to be the deviation from and return to full cash position, or if the maximum number of time periods has been reached - that is the agent either takes no action throughout the maximum number of time periods, or completes a sequence of actions [buy and then sell]. For example, the agent would observe the states starting from t_0, make the decision to buy at t_a and then sell it at t_b, t_b <= t_{max} . At


Starting at t_0
Observe s_0
Until t_k
Nothing happens
$t_k = t_{max}$
Episode ends
Takes action
a_k = [buy]
Observe s_{k+1} through $s_{t=max}$
And takes action a_{k+i} = [Sell] at t_{k+i}
Episode ends
Forced to sell at t_{max} if no action is taken before max time period is reached
Episode ends

Reward:
When we try to develop trading strategies, what should we optimize for? Profit seem to be the obvious answer. However that is not the whole story. For example, how much the performance fluctuates matters too.

For simplicity, the basic model will assume zero reward until a complete buy-sell sequence occurs. The profits and losses will be the rewards.

In syntax, action [‘buy’] while having a stock position and action[‘sell’] while having no stock position are heavily punished so that the agent would not opt for these in the optimal policy



```
if self.t < max_time-1:
    if agent.stock == 0: # if there is no stock position the agent can buy or do nothing
        if action == 1: # buy sstock at closing price
            agent.stock_position = agent.starting_cash / closing_price * (1 - transaction_fee)
            agent.cash = 0 # Cash position becomes 0
            agent.stock = 1 # Boolean indicator for stock
            reward = 0 # Agent recieves no reward for this action
        elif action == 0: # no action
            reward = 0  # Agent recieves no reward for this action
        else:
            reward = -10000 # Not a possible choice of action in real life - heavily punished

    elif agent.stock == 1: # if there is a stock position the agent can sell or do nothing
        if action == 2: # sell stock at closing price
            agent.cash = agent.stock_position * closing_price * (1 - transaction_fee)
            reward = agent.cash - agent.starting_cash
            self.done = True # Episode ends
        elif action == 0: #  no action
            reward = 0
        elif action == 1:
            reward = -10000

elif self.t == max_time-1:
    if agent.stock == 1:
        # stock position is forced to liquidate
        agent.cash = agent.stock_position * closing_price * (1 - transaction_fee)
        reward = agent.cash - agent.starting_cash
        self.done = True # Episode ends

    else: # If no actions taken for the entire episode
        reward = 0 # Receives no reward
        self.done = True # Episode ends
```

Refinement:

During the process of coming up with the final structure for the model, I have had various attempts of different functionalities that I did not end up using. Here are a few examples:

1. I tried separating functions such as creating the Q table and choosing actions like the old self-driving-taxi project - it works but using the codes for Sarsa turned to be much simpler.
2. I tried using a dictionary of valid actions for having a stock position and not having a stock position. I find that this would make coding implementation much more difficult (in terms of constructing the Q table). Instead I just punished the invalid actions a lot in reward design - the agent would quickly learn not to take these actions (selling when there is no stock position or buying when there is no cash).
3. The discount factor and learning factor does not much significant impact on model performance - this will be discussed in more details in the **Result** section.

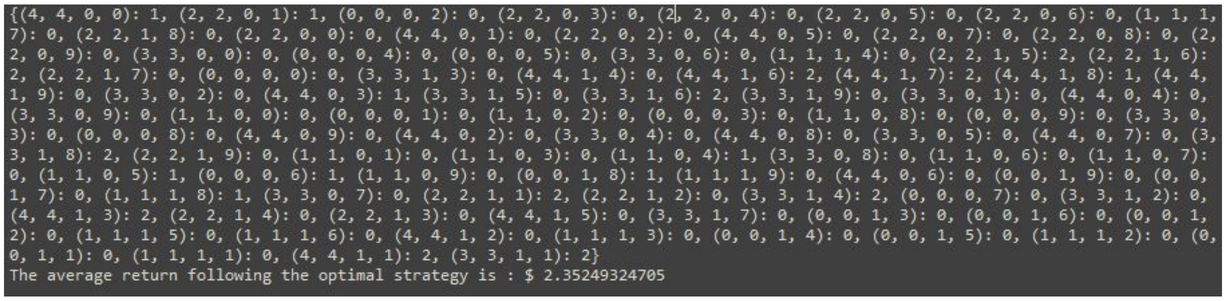
IV. Results:

Model Evaluation and Validation

The basic model designed with a Environment class and an Agent class. Because only the reward accumulated at the end of an episode matters, the agent utilizes Sarsa learning model, where each state-action (each unique set of inputs and action) pair is assigned a Q value. The Q values are continuously updated throughout training. The model is trained with 100000 randomly selected episodes.

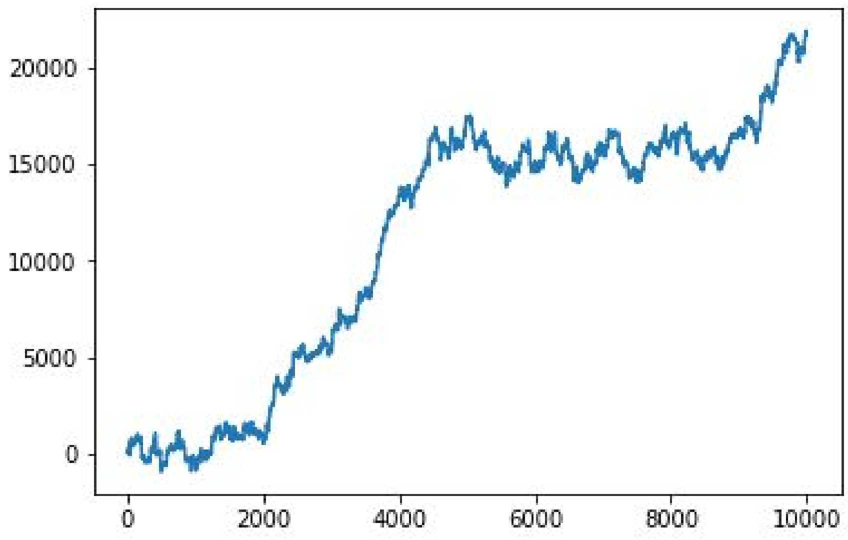
The optimal policy under this learning model is obtained after 50000 episodes of training. The optimal policy is then used to compute returns for 5000 trial episode. The average of the returns is computed. In the end, The result roughly breaks even under the model's assumptions (0.1% transaction cost).

Below is the resulting optimal policy after training and its estimated returns at 0.7 learning rate and discount rate of 1. [(short term indicator, long term indicator, stock position, time period): action]



```
{(4, 4, 0, 0): 1, (2, 2, 0, 1): 1, (0, 0, 0, 2): 0, (2, 2, 0, 3): 0, (2, 2, 0, 4): 0, (2, 2, 0, 5): 0, (2, 2, 0, 6): 0, (1, 1, 1,
7): 0, (2, 2, 1, 8): 0, (2, 2, 0, 0): 0, (4, 4, 0, 1): 0, (2, 2, 0, 2): 0, (4, 4, 0, 5): 0, (2, 2, 0, 7): 0, (2, 2, 0, 8): 0, (2,
2, 0, 9): 0, (3, 3, 0, 0): 0, (0, 0, 0, 4): 0, (0, 0, 0, 5): 0, (3, 3, 0, 6): 0, (1, 1, 1, 4): 0, (2, 2, 1, 5): 2, (2, 2, 1, 6):
2, (2, 2, 1, 7): 0, (0, 0, 0, 0): 0, (3, 3, 1, 3): 0, (4, 4, 1, 4): 0, (4, 4, 1, 6): 2, (4, 4, 1, 7): 2, (4, 4, 1, 8): 1, (4, 4,
1, 9): 0, (3, 3, 0, 2): 0, (4, 4, 0, 3): 1, (3, 3, 1, 5): 0, (3, 3, 1, 6): 2, (3, 3, 1, 9): 0, (3, 3, 0, 1): 0, (4, 4, 0, 4): 0,
(3, 3, 0, 9): 0, (1, 1, 0, 0): 0, (0, 0, 0, 1): 0, (1, 1, 0, 2): 0, (0, 0, 0, 3): 0, (1, 1, 0, 8): 0, (0, 0, 0, 9): 0, (3, 3, 0,
3): 0, (0, 0, 0, 8): 0, (4, 4, 0, 9): 0, (4, 4, 0, 2): 0, (3, 3, 0, 4): 0, (4, 4, 0, 8): 0, (3, 3, 0, 5): 0, (4, 4, 0, 7): 0, (3,
3, 1, 8): 2, (2, 2, 1, 9): 0, (1, 1, 0, 1): 0, (1, 1, 0, 3): 0, (1, 1, 0, 4): 1, (3, 3, 0, 8): 0, (1, 1, 0, 6): 0, (1, 1, 0, 7):
0, (1, 1, 0, 5): 1, (0, 0, 0, 6): 1, (1, 1, 0, 9): 0, (0, 0, 1, 8): 1, (1, 1, 1, 9): 0, (4, 4, 0, 6): 0, (0, 0, 1, 9): 0, (0, 0,
1, 7): 0, (1, 1, 1, 8): 1, (3, 3, 0, 7): 0, (2, 2, 1, 1): 2, (2, 2, 1, 2): 0, (3, 3, 1, 4): 2, (0, 0, 0, 7): 0, (3, 3, 1, 2): 0,
(4, 4, 1, 3): 2, (2, 2, 1, 4): 0, (2, 2, 1, 3): 0, (4, 4, 1, 5): 0, (3, 3, 1, 7): 0, (0, 0, 1, 3): 0, (0, 0, 1, 6): 0, (0, 0, 1,
2): 0, (1, 1, 1, 5): 0, (1, 1, 1, 6): 0, (4, 4, 1, 2): 0, (1, 1, 1, 3): 0, (0, 0, 1, 4): 0, (0, 0, 1, 5): 0, (1, 1, 1, 2): 0, (0,
0, 1, 1): 0, (1, 1, 1, 1): 0, (4, 4, 1, 1): 2, (3, 3, 1, 1): 2}
The average return following the optimal strategy is : $ 2.35249324705
```

Below is the cumulated returns over the policy episodes:



Justification:

The results are barely satisfactory in terms of profit (considering the $5000 capital and 10000 trades). But this is not a surprise given that we are basically only feeding the model two indicators for trading signal and that the agent is allowed to trade only under very strict conditions (trade at closing price, one trade per period, all or none full position size).

I have experimented with multiple learning factor and discount factor but the results have not differed significantly. I believe that is because the primary driver for the model robustness is the state inputs. For the number distinct space, 100000 episodes of training is adequate to capture the action values under the model’s assumption. In others words, the model has learned the best it can, from what it is given. The discount factor should not matter because only reward at the terminal state matters.

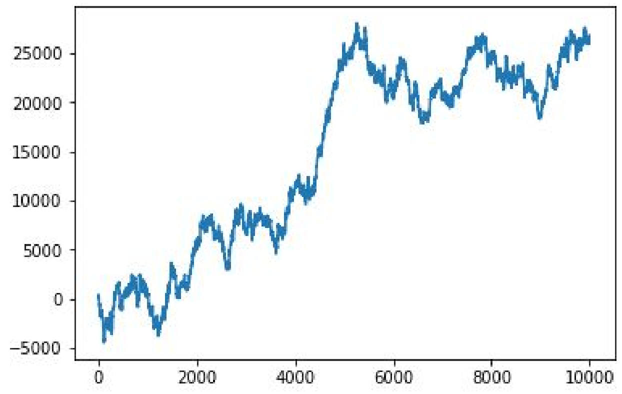
V. Conclusion:

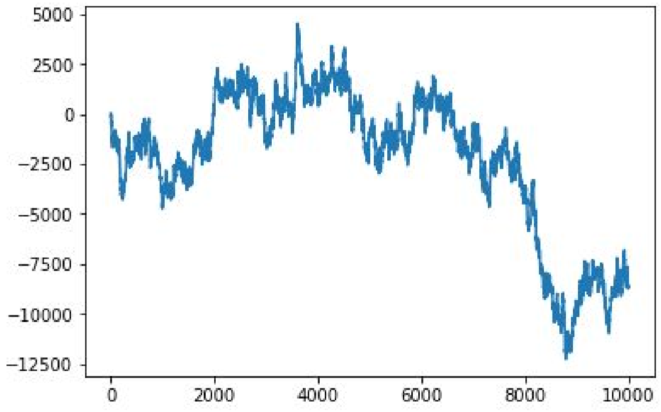
Summary:

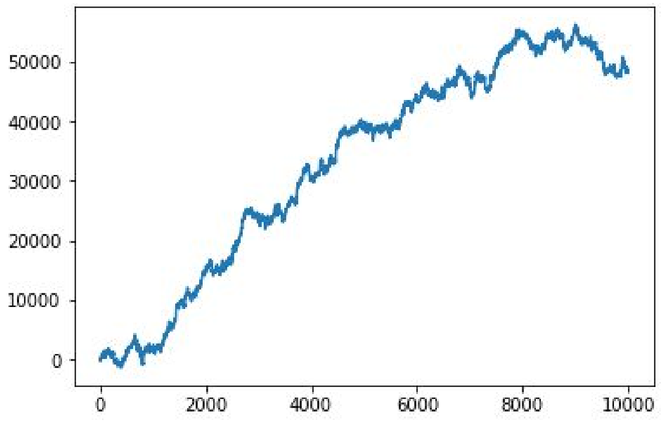
This project has implemented reinforcement learning in the setting of security trading. Starting from daily price data of a single stock, we have come up with a structure of episodes states, in which an agent has learnt to trade stock under very restricted conditions. While restricted, the conditions are, nonetheless, realistic. That is,anything the agent does, someone can do in real life.

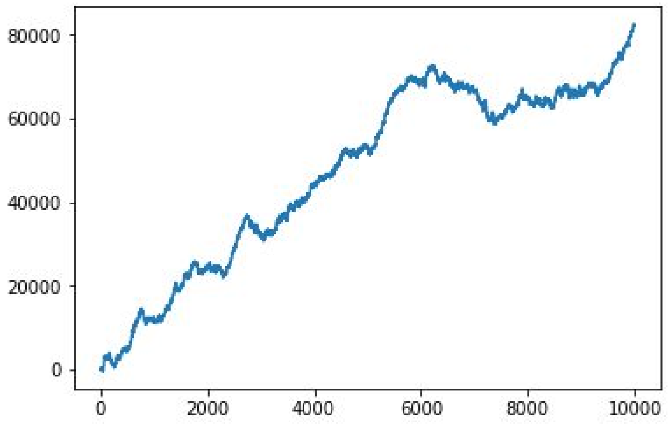
Free-form Visualization:

Below are more results from policy testing under the exact same setting. While performance varies, the average return does not fall below 0 significantly.









Reflection:

With limited coding experience and ML knowledge, figuring out how to design the environment and agent was the most difficult part - particularly how the variables are passed on and how the functions are interrelated with each other to make the entire model run. I barely knew what a ‘Class’ is when I started the project. It is fascinating that one could design an algorithm that learns to trade with under 500 lines of code (God knows how much money and how long it took me to do that). I have managed to develop it from scratch. And now that the framework is in place and I understand the process, testing ideas and making tweaks would be much easier.

Improvement:

State inputs: the inputs (essentially trade signals) are probably the most important elements in developing a successful trading agent with reinforcement learning framework. I think the state design is where the ‘art’ comes in If the inputs selected are actually the ones that matter, the rest should be easy. All that needs to be altered is what gets passed on as states in the <step()> function in the Environment Class. Without good selection of inputs, the model would be garbage-in-garbage-out. The model has provided a good framework for further experimentation.

Continuous state representations: time and prices are continuously moving in real life. While episodic and discrete representation is a start, it is still quite limiting. If I had more knowledge on subject, I would consider implementing function approximation for the price movements.

Action: trades can be made at any point in time. The size of the positions should be continuous too.