Reinforcement Learning Stock Trading Project Report

Zhenning Li (zhenning.li@student-cs.fr)

March 28, 2022

1 Overview

I choose stock trading project only because I am fond of making investments on financial market, and want to see how Reinforcement Learning can help better making profits. Since in trading, there are thousands of factors to be considered and many different ways to trade (buy, sell, any portion from 1% to 200% all possible), if I use basic Q-Learning, then the Q-table will be super large and the training process would take infinity to learn. Therefore, I choose DQL which is audacious enough to explore the market data and let neural network to study the states (market data in time series type) and make a wise choice.

2 Algorithms Implementation

In Deep Q-learning, I used a neural network to approximate the Q-value function. The state is given as the input and the Q-value of all possible actions is generated as the output. The comparison between Q-learning & deep Q-learning is wonderfully illustrated below:

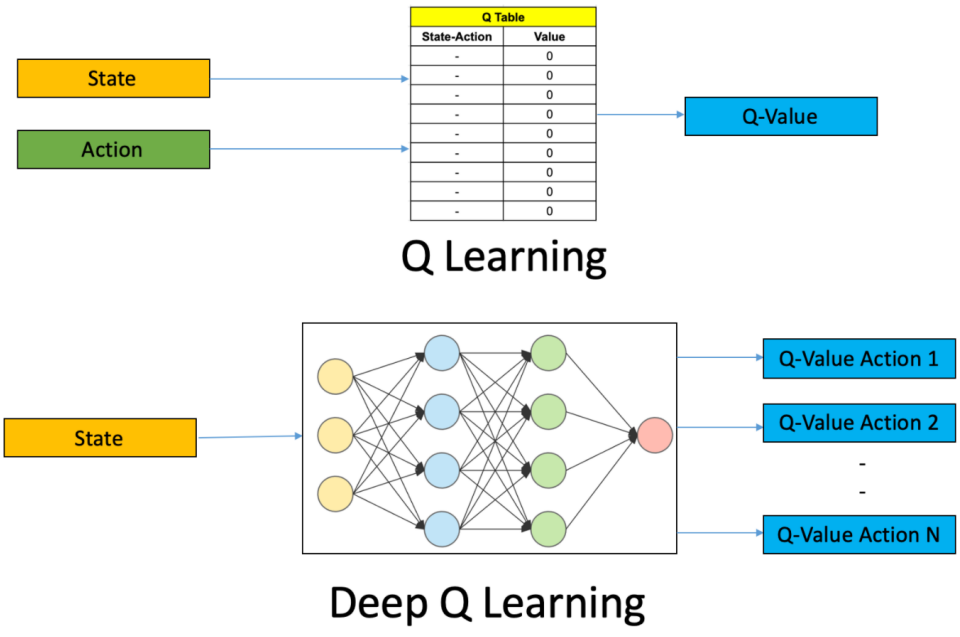

Figure 1: Deep Q-Learning Steps

3 Overall Logic

The fundamental logic of all the project is defined as below:

Get data -> Build Agent -> Train the Agent -> Evaluate Performance -> Optimize the Agent -> Train & Evaluate and Compare -> Optimize Input Data -> Train & Evaluate and Compare -> Conclusion Reached

More specifically, the steps are as follows:

Step 1) Get US stock data (DJI and Nasdaq tickers) from yfinance package which includes simple information including companies' Close Price, Open Price, Volume etc., and general market data including Nasdaq Index etc. Process the data and calculate some useful technical indicators (Moving Average, RSI, MACD, Pearson Correlation Coefficient with Nasdaq Index of same period etc.) to make each company's DataFrame has many columns as features (including Close, Volume, High-Low, Nasdaq Index of same period, Moving Average of 5 Days, RSI Index etc). I will use these information as training data to feed the Agent and let it learn the hidden pattern.

Step 2) Build Agent. Briefly, the Agent Class has the following functions:

__init__: it initializes a neural network and includes all deep learning and reinforcement Deep Q Learning hyperparameters including epsilon initial value, epsilon decay coefficient, gamma etc.

get_action: input a state and it will return one available action.

get_state: input the training dataset and it will transform the dataset format into a state value, which can be identified by the algorithm.

replay: train the data using mini batch method and do Q value update here. **Our Q-table is a dictionary and it get updates like this: target[action] += self.gamma * np.amax(Q_new[i])**. It returns the cost that needs to be minimized in neural network.

train: train the agent for a number of iterations, store inventory, current money, initial money, rate of return etc. and do states update and print the performance of this episode.

operate: after training I have a RL model, and I can use this function to use the RL model on other companies and see how it performs in the next period. Mainly I use this function to fight overfitting and determine most hyperparameters.

Step 3) Determine train and test data start and end date, download the data corresponding to this choice, and choose a set of hyperparameters and begin training.

Step 4) After training, I use the trained model on test dataset and see how our model performs. I choose some tickers as test companies and calculate each one's total investment return and the overall average return, and repeat this process for 100 times and calculate the average performance and standard deviation, which will be used in further evaluation.

Step 5) Visualize and evaluate the performances and try to use different set of hyperparameters to optimize the results. If the result did not increase much, try to optimize the input data (for example, use some clustering method to better differentiate companies in training dataset), change the factors taken into consideration and retrain the model, and repeat the above processes until a good and stable result is reached.

4 Implemented Agents Details

I put both environment and agents in the same class because I think the state spaces, action spaces and rewards are not complex and it's easier to comprehend and debug in one simple class.

Here is the detailed explanation of our Environment and Agent. I explain the functions and important things here.

4.1 State Value Example: Training and Testing dataset format

In loading training data (**state**) step, I have already received companies' DataFrame that has many features (close price, open price, technical indicators etc.) as columns, in processing data step, I choose some more useful columns data and flatten them into one dimension NumPy array format, since I are using Tensorflow's Dense layer and it automatically relates numbers of different positions in a list so there is no need to build a multi dimensional array as state. Besides, all the numbers in input datasets are normalized.

The DataFrame for a single company in one year period is like this:

	Close	Open	High	Low	Adj Close	Volume	difference	nasdaq_index	nasdaq_dividend_market_value	dji_adj_close	dji_volume	ma_5	ma_10	ma_30	rsi
0	75.087502	0.210292	0.220719	0.254288	0.231023	0.233767	0.056219	0.369533	0.352973	0.869622	0.137912	0.209817	0.201190	0.174544	0.828469
1	74.357498	0.213100	0.220658	0.258322	0.222082	0.262313	0.028830	0.357706	0.004615	0.849828	0.122037	0.199966	0.190788	0.162239	0.828469
2	74.949997	0.202734	0.218760	0.246774	0.229339	0.188763	0.093287	0.366101	0.010215	0.855624	0.139132	0.207962	0.199231	0.172226	0.828469
3	74.597504	0.221399	0.221637	0.261340	0.225022	0.163712	0.015239	0.365624	0.196154	0.845495	0.147102	0.203205	0.194208	0.166285	0.828469
4	75.797501	0.213131	0.232474	0.260354	0.239719	0.224812	0.094728	0.375669	0.003705	0.859154	0.189741	0.219399	0.211306	0.186512	0.828469
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
247	130.960007	0.927311	0.922121	0.956151	0.929212	0.109348	0.080725	0.978752	0.012723	0.976329	0.166766	0.941217	0.940240	0.966856	0.717055
248	131.970001	0.916944	0.934734	0.960092	0.941690	0.021691	0.139209	0.984320	0.021821	0.982256	0.000000	0.950042	0.952680	0.973868	0.743551
249	136.690002	0.949895	0.982245	0.989777	1.000000	0.204821	0.260297	1.000000	0.001033	0.999527	0.203681	0.977113	0.973027	0.983689	0.848387
250	134.869995	1.000000	1.000000	1.000000	0.977516	0.195767	0.311367	0.991853	0.007430	0.993747	0.275226	0.995034	0.991679	0.992460	0.755554
251	133.720001	0.969517	0.965714	0.988422	0.963309	0.131012	0.158156	0.995128	0.216435	1.000000	0.189922	1.000000	1.000000	1.000000	0.699377

252 rows × 15 columns

Figure 2: Dataset Example

But this is not the kind of data (state value) that is fed into the model. The state value used for training is like this (data for 3 days period): (I select 3 factor model and make it flattened)

array([75.08750153, 74.35749817, 74.94999695, 0.23376706, 0.26231331, 0.18876348, 0.32309863, 0.21335679, 0.11140289, 0.86729234, 0.50515125, 0.5069423]

It is a one dimensional array list, and as we can see, the first 3 are close prices, the rest 9 (in 3 groups) corresponds to 3 other factors' value during that 3 days. This is the state value for each time point.

There is no need to worry about if the closing price is normalized or not, since when I fed the model, I will do another differencing to make sure every column data is normalized and reasonable.

1 factor model only involves training data of the closing price.

3 factor model involves training data of the closing price, daily volume and RSI index.

6 factor model involves training data of the closing price, daily volume, RSI index, High-Low Difference, Nasdaq-Index of the same day, and Moving Average on 5 days.

In this project, I only choose the 30 biggest companies that are in the DJI list (20 as training companies and 10 as testing companies), because if I do companies listed in Nasdaq, the result would be highly unstable since most of the Nasdaq companies are extremely small (numerous 10 million USD market capacity companies) and their daily volume is too small to cause liquidity risk, and of course, their price fluctuations are also extremely unreasonable due to the fact that seldom people would trade on them.

4.2 Agent Function Explanation: __init__

4.2.1 Parameter Explanation

The init function has the following parameters:

```
def __init__(self, state_size,
             window_size,
             trend_data,
             skip,
             batch_size,
             drop_out_rate = [0.3, 0.3, 0.3, 0.3, 0.3],
             GD_learning_rate = 1e-5,
             gamma = 0.95,
             epsilon = 0.5,
             epsilon_min = 0.01,
             epsilon_decay = 0.9997,
             no_of_factors = 3)
```

Listing 1: init function

window_size: it is the length of the window of the stock data. For example, if I decide to train the model using a window size of 3 and only has the close price as the feature, then, when t=0, the input training data is [0, 0, price_t=0], when t=4, the input training data is [price_t=2, price_t=3, price_t=4]. In theory, we should use a larger window size when our training data has longer periods (Here the default value is 64 or 128 * data periods (number of years)).

state_size: it is the size of the training state (since our input state is 1-dimensional NumPy array, we have to determine the this length of the window (the period)). Its default value is the same as window size.

trend_data: it is the input training data, in dict type. The key and value of this dict is companies' ticker names, and the 1-dimensional array data. This dict will be directly used for training.

skip: it is an advanced training parameter. It is for simulating daily data or weekly data of data of any days. For example, skip=1 means daily data, skip=5 means weekly data. In model optimizing steps I found usually weekly data is better in terms of return stability and absolute performance.

batch_size: I use mini batch method to train the data. A small batch size may result in the agent not learning any good pattern, and a large batch size may result in memory insufficiency.

drop_out_rate: it is a list that contains the drop out rates corresponding to each neural network layer in our model. I used a 5-fully connected layer network to build the decision making step, and to counter overfitting, the drop out rate has the default value of 0.3.

GD_learning_rate: I used Gradient Descent Optimizer to optimize the neural network, and this parameter is its learning rate.

gamma: it is a parameter in Deep Q Learning. It means the level of consideration that is taken for future rewards. Usually it is set around 0.95, so that the agent can fully consider what it can learn from future. If it is set a low value, theoretically the agent may find it difficult to reach to the target and thus cannot learn a good strategy.

epsilon, epsilon_min, epsilon_decay: epsilon is the initial value of epsilon in Deep Q Learning, it determines the balance between exploration and exploitation. A small epsilon means more exploitation and large epsilon means more exploration. Therefore, a small epsilon can make the score line plot more smooth, while usually a little larger epsilon can make the average performance better but with larger fluctuations. However, for a very large epsilon, usually the model is always trying exploring new things and if the state is complex, it may never reach a good performance. Therefore, a good way to train the model is to use a volatile epsilon: a larger epsilon at first, then decreases each time a bit by multiplying an epsilon_decay until a final epsilon (epsilon_min) is reached.

no_of_factors: it is purely for determining x-axis of visualization and operate() function's data length.

4.2.2 Other important initiations

Deep Learning X and Y variable initiation:

```
        self.X = tf.placeholder(tf.float32, [None, self.state_size])
        self.Y = tf.placeholder(tf.float32, [None, self.action_size])
```

Listing 2: X and Y variable initiation

Here X is the data for training, and Y is the prediction that is to maximize the reward.

Neural network initiation:

```
        layer1 = tf.layers.dense(self.X, 64, activation = tf.nn.relu)
        layer2 = tf.layers.dense(layer1, 256, activation = tf.nn.relu)

        layer_dropout = tf.keras.layers.Dropout(self.drop_out_rate[0])
        outputs = layer_dropout(layer2, training=True)

        layer3 = tf.layers.dense(outputs, 1024, activation = tf.nn.relu)

        layer_dropout_2 = tf.keras.layers.Dropout(self.drop_out_rate[1])
        outputs_2 = layer_dropout_2(layer3, training=True)

        layer4 = tf.layers.dense(outputs_2, 256, activation = tf.nn.relu)

        layer_dropout_3 = tf.keras.layers.Dropout(self.drop_out_rate[2])
        outputs_3 = layer_dropout_3(layer4, training=True)

        layer5 = tf.layers.dense(outputs_3, 64, activation = tf.nn.relu)

        layer_dropout_4 = tf.keras.layers.Dropout(self.drop_out_rate[3])
        outputs_4 = layer_dropout_4(layer5, training=True)
        layer6 = tf.layers.dense(outputs_4, 16, activation = tf.nn.relu)

        layer_dropout_5 = tf.keras.layers.Dropout(self.drop_out_rate[4])
        outputs_5 = layer_dropout_5(layer6, training=True)

        # logits is the prediction
        self.logits = tf.layers.dense(outputs_5, self.action_size)
```

Listing 3: Neural network initiation

Here I have a 5 fully connected network with dropouts, and self.logits returns the predicted action (buy or sell or do nothing).

Neural Network optimizer:

```
        self.cost = tf.reduce_mean(tf.square(self.Y - self.logits))
        self.optimizer =tf.train.GradientDescentOptimizer(GD_learning_rate).minimize(self.
    cost)
```

```
        self.sess.run(tf.global_variables_initializer())
```

Listing 4: Neural Network optimization

Here the cost is the loss, which is calculated by action->reward minus self.logits value. I want self.logits to be close to self.Y, because I want the agent to take the best action in each day, and self.logits has to move towards the maximal reward after iterations.

4.3 Agent Function Explanation: get_state()

The function is as follows:

```
def get_state(self, t, trend):
    window_size = self.window_size + 1
    d = t - window_size + 1
    block = trend[d : t + 1] if d >= 0 else -d * [trend[0]] + trend[0 : t + 1]
    res = []
    for i in range(window_size - 1):
        res.append(block[i + 1] - block[i])
    return np.array([res])
```

Listing 5: get_state function

This function is mainly for processing and unionizing the input data. The state value of each time point, is the 1-dimensional stock data as described before, but when I start training, the data I have is very limited and the list is very short, for example, if the window_size=4, but currently the state is at t=1, therefore the state value I have now is like [price_t=0, price_t=1], and here in this function, I added some 0s to make it like [0, 0, price_t=0, price_t=1], making all states of any time point have the same length.

4.4 Agent Function Explanation: get_action()

The function is as follows:

```
def get_action(self, state, available_actions, train=True):
    # epsilon-greedy method
    if not train or random.random() <= self.epsilon or not train:
        return np.random.choice(available_actions)

    # choose largest Q value after update
    action_values = self.sess.run(self.logits, feed_dict = {self.X: state})[0]
    return available_actions[self.argmax([action_values[action] for action in
    available_actions])]
```

Listing 6: get_action function

This function enables us to have one action after feeding it a state. I used epsilon-greedy method, that when a random number is less then epsilon, it does exploration, in other time, It chooses the best action it can take according to Q-table and neural network outcome.

4.5 Agent Function Explanation: replay()

The function is as follows:

```
    def replay(self, batch_size):
        # Train the data using mini_batch method
        mini_batch = []
        l = len(self.memory) # memory is deque, first in first out

        for i in range(l - batch_size, l):
            mini_batch.append(self.memory[i])
        replay_size = len(mini_batch)
```

```
        X = np.empty((replay_size, self.state_size))
        Y = np.empty((replay_size, self.action_size))

        states = np.array([a[0][0] for a in mini_batch])
        new_states = np.array([a[3][0] for a in mini_batch])

        # Get Q value based on actions got from states
        Q = self.sess.run(self.logits, feed_dict = {self.X: states})
        Q_new = self.sess.run(self.logits, feed_dict = {self.X: new_states})

        # Q value update & Network
        for i in range(len(mini_batch)):
            state, action, reward, next_state, done = mini_batch[i]
            target = Q[i]
            target[action] = reward

            # Update Q table when having positive return
            if not done:
                target[action] += self.gamma * np.amax(Q_new[i])
            X[i] = state
            # Here Y is the reward got from simulation
            Y[i] = target

        cost, _ = self.sess.run(
            [self.cost, self.optimizer], feed_dict = {self.X: X, self.Y: Y}
        )
        if self.epsilon > self.epsilon_min:
            self.epsilon *= self.epsilon_decay
        return cost
```

Listing 7: replay function

Here, replay function trains the data using mini batch method and do **Q value update** ($target[action] += self.gamma * np.amax(Q\_new[i])$). Replay returns the cost that needs to be minimized in neural network. The mini_batch records format like this: (state, action, reward(invest), next_state, if successful judgement (starting_money $<$ initial_money)).

4.6 Agent Function Explanation: train()

The function is as follows:

```
    def train(self, iterations, checkpoint, initial_money):
        # Begin to run episodes, every state update only depends on the next one
        for i in range(iterations):
            total_profit = 0
            inventory = []

            ticker = list(self.trend_data.keys())[i%(len(self.trend_data))]
            trend = self.trend_data[ticker][: close_price_length]
            lag1 = np.array(trend)[1:] - np.array(trend)[:-1]
            lag1 = [0] + list(lag1)

            state = self.get_state(0, lag1)
            starting_money = initial_money or 2*trend[0]
            current_money = starting_money

            for t in range(0, len(trend) - 1, self.skip):
                # make decision on current state
                available_actions = [0]

                if current_money >= trend[t] and t < (len(trend) - self.half_window):
                    available_actions.append(1)
                if len(inventory) > 0:
                    available_actions.append(2)

```

```
                action = self.get_action(state, available_actions)
                next_state = self.get_state(t + 1, lag1)

                if action == 1: # buy
                    inventory.append(trend[t])
                    current_money -= trend[t]

                elif action == 2: # sell
                    # buying price is the price when bought, first in first out
                    bought_price = inventory.pop(0)
                    total_profit += (trend[t] - bought_price)
                    current_money += trend[t]

                invest = ((current_money - starting_money) / starting_money)
                self.memory.append((state, action, invest,
                                    next_state, current_money < starting_money))

                # states update
                state = next_state
                batch_size = min(self.batch_size, len(self.memory))
                cost = self.replay(batch_size)
```

Listing 8: train function

Here, train function trains the agent for a number of iterations, stores inventory, current money, initial money, rate of return etc. and do states update and print the performance of this episode. In states update, what I only aim to do, is to calculate the cost of that episode and try to minimize it according to the neural network optimizer rule.

4.7 Agent Function Explanation: operate()

The function is as follows:

```
    def operate(self, initial_money, trend, print_out=True):
        starting_money = initial_money or 2*trend[0]
        current_money = starting_money
        states_sell = []
        states_buy = []
        inventory = []
        lag1 = np.array(trend)[1:] - np.array(trend)[:-1]
        lag1 = [0] + list(lag1)

        # state at time 0
        state = self.get_state(0, lag1)
        for t in range(0, int(len(trend)/self.no_of_factors) - 1, self.skip):
            # judge what actions I can have in action pools
            available_actions = [0]

            if current_money >= trend[t] and t < (len(trend)/self.no_of_factors - self.
    half_window):
                available_actions.append(1)
            if len(inventory) > 0:
                available_actions.append(2)

            action = self.get_action(state, available_actions, train=False)
            next_state = self.get_state(t + 1, lag1)

            if action == 1 and initial_money >= trend[t] and t < (len(trend)/self.
    no_of_factors - self.half_window):
                inventory.append(trend[t])
                current_money -= trend[t]
                states_buy.append(t)

            elif action == 2 and len(inventory):
                bought_price = inventory.pop(0)
                current_money += trend[t]
```

```
                states_sell.append(t)
                try:
                    invest = ((close[t] - bought_price) / bought_price) * 100
                except:
                    invest = 0
            state = next_state

        invest = ((current_money - starting_money) / starting_money) * 100
        total_gains = current_money - starting_money
        return states_buy, states_sell, total_gains, invest
```

Listing 9: get_state function

Here operate uses the trained RL model on other test companies and see how it performs in the future chosen periods. Mainly I use this function to fight overfitting and determine best hyperparameters.

5 Environment Description

5.1 State Space

As shown in the last section the state value format (1D array list like array([75.08750153, 74.35749817, 0.26231331, 0.18876348, 0.11140289, 0.86729234, 0.50515125, 0.5069423]), the state value function is a dictionary that stores this kind of states and their corresponding values. Since I are handling time series data, it's not possible to get all possible states into this dictionary (although states are theoretically discrete), and that's also the reason why I strictly refuse to use Q-Learning to do stock trading agent.

5.2 Action Space

To make the agent simple and focus on finding stock price patterns, I set that only 3 actions are available: **0 represents do nothing, 1 represents buy, and 2 represents sell**. All actions, no matter buy or sell, as long as it is chosen, the agent will buy or sell the maximal amount possible. Our initial money is 2 times the first stock closing price, and this is to reduce the risk of not having enough money to buy the stock after making wrong decision in the first round.

5.3 Rewards

In our agent, to make the overall logic simple, the reward is set to be the total return rate (in percentage), and it is only related to the amount of money gained divided by total initial money, not related to any other things like the number of operations (buying or selling), the company kind (good or bad, large or small) it chose etc.

6 Results and Visualizations

6.1 Results using 1 factor model

Firstly, check if the close price is problematic or not:

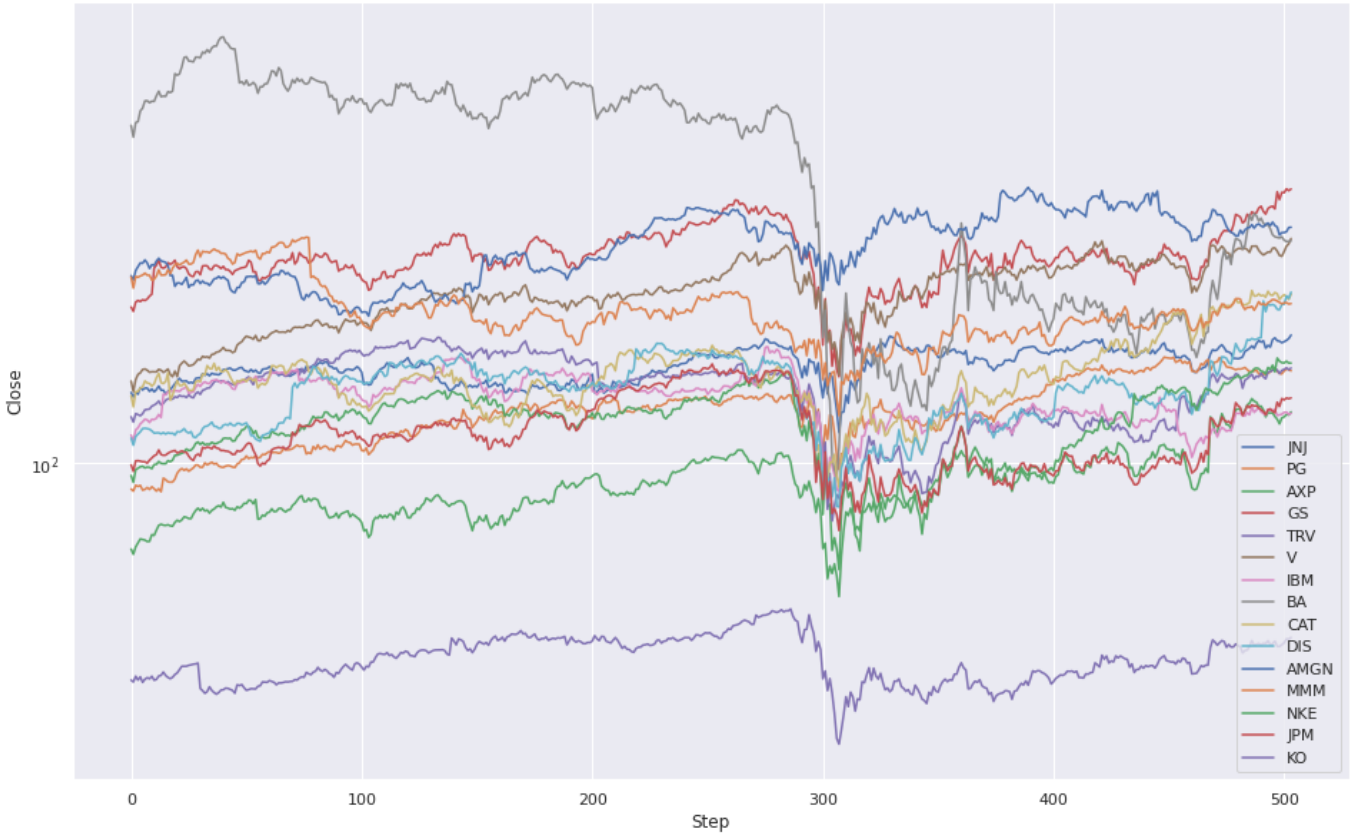

Figure 3: Training Dataset Closing Price Example

Parameters	Value	Explanation
train start date	2019-01-01	Use 2 years' data to train and 1 year's data to evaluate
train end date	2020-12-31	
test start date	2021-01-01	
test end date	2021-12-31	
window size	256	Since I am using 2 years' data, a larger window size is reasonable
skip	1	Use daily data to train
batch size	1024	
drop out rate	[0.3, 0.3, 0.3, 0.3, 0.3]	
GD learning rate	1e-5	
gamma	0.95	
epsilon	0.5	
epsilon min	0.01	
epsilon decay	0.9997	
no of factors	1	
iterations	50	50 iterations as a starting point

The testing result for one trial is:

```
1  print_final_return(all_test_data_1, agent=agent_1_factor_50_epochs, print_outcome=True, print_final_total_money=False)

Ticker INTC: total gains 62.4300, total investment 66.30%
Ticker TRV: total gains 34.6900, total investment 14.78%
Ticker AXP: total gains 15.7299, total investment 8.22%
Ticker CAT: total gains 15.5100, total investment 6.14%
Ticker JPM: total gains -7.5500, total investment -3.80%
Ticker CVX: total gains 66.7100, total investment 30.13%
Ticker V: total gains 41.5899, total investment 15.64%
Ticker MSFT: total gains 46.3099, total investment 22.90%
Ticker MMM: total gains -47.2600, total investment -12.37%
Ticker CSCO: total gains 2.9300, total investment 3.41%
15.134807235219743
```

Figure 4: 1 factor 50 iterations result of one trial

This result shows the tickers I am going to buy in test companies pool, and the return rate on each of them. Altogether, we will have an annual rate of return of 15.13%, which is rather great.

Let's choose some companies and see the buying and selling points on the graphs more clearly.

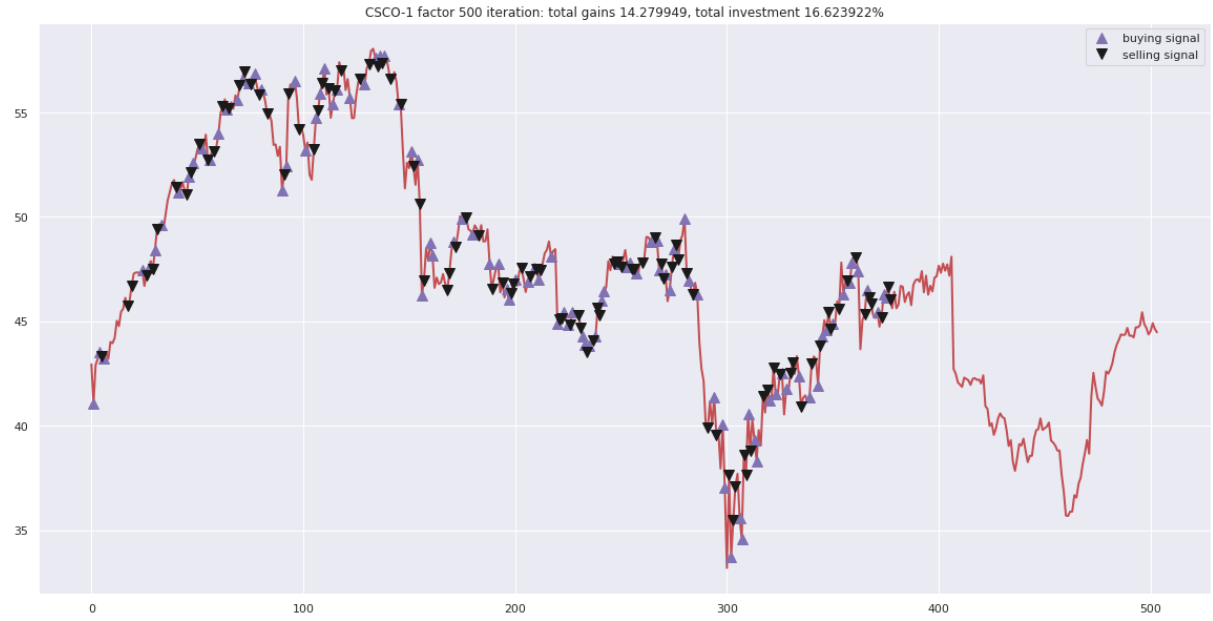

Figure 5: CSCO - 1 factor 500 iterations Agent Buying and Selling Points

Since each operate trial is not always the same (random seed is not set in our project), let's run 100 times and see the average return rate distribution.

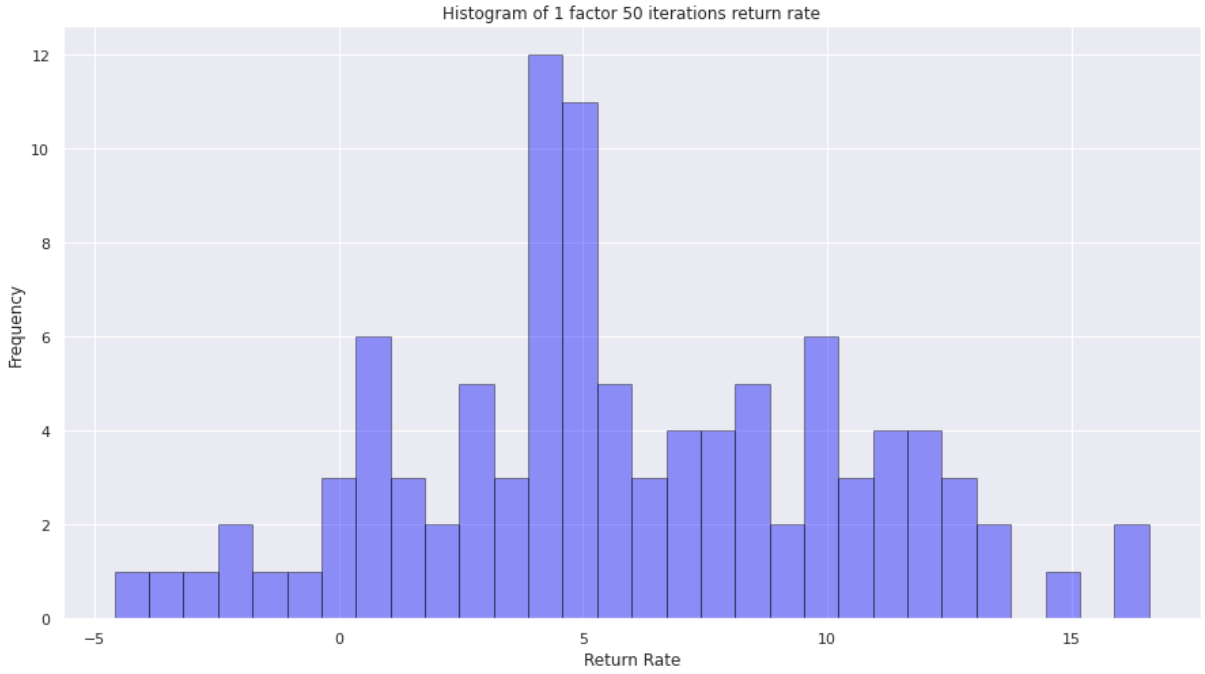

Figure 6: 1 factor 50 iterations result distribution

Here we can clearly see, the average annual rate of return is about 5%, and with this model and parameters, it seems very easy to get a positive return.

Now I try more iterations: iterations=250, 500 and 100 and see their results.

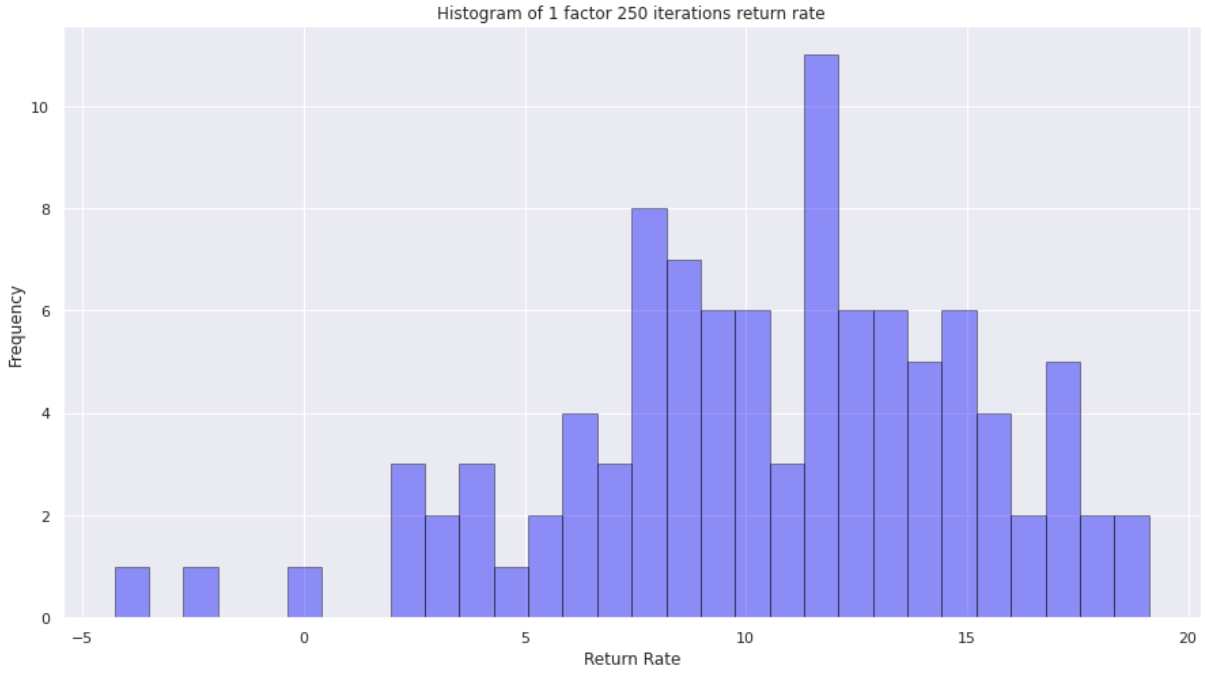

Figure 7: 1 factor 250 iterations result distribution

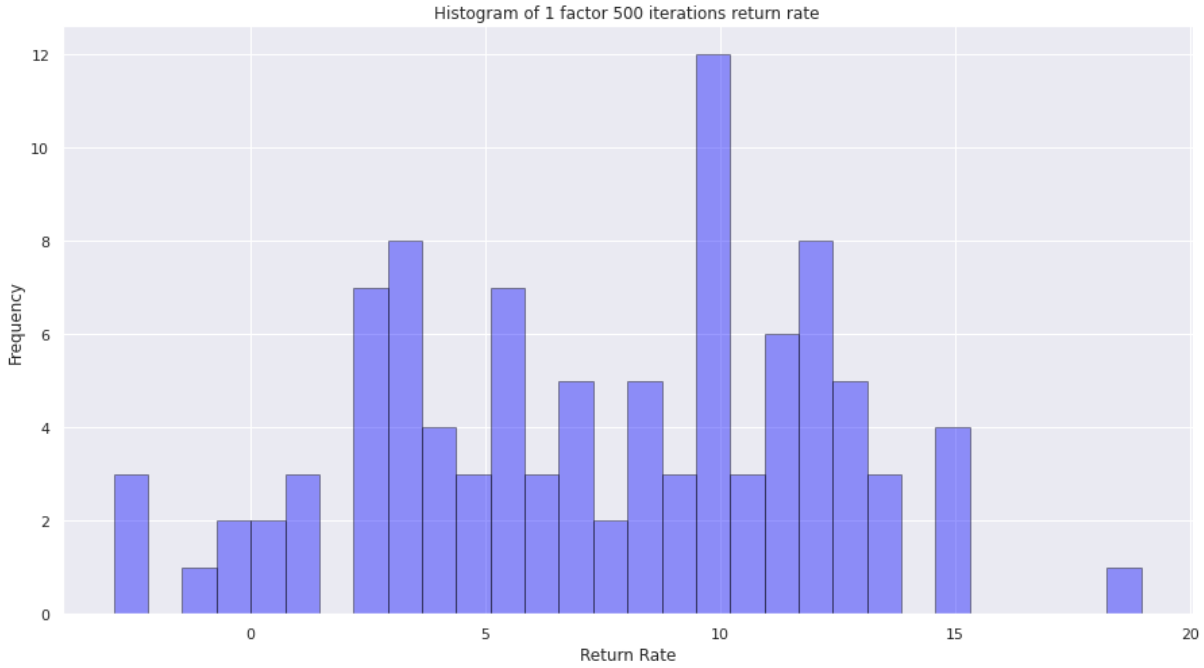

Figure 8: 1 factor 500 iterations result distribution

Figure 9: 1 factor 1000 iterations result distribution

Upon the above 4 graphs, we can clearly see the average is shifting to right, which is good because it means the agent is performing better and better with the number of episodes increased.

Here is the average and standard deviation of these 100 times' return results:

Iterations	Average Return Rate	Standard Deviation
50	5.89	4.51
250	10.51	4.6
500	7.42	4.59
1000	6.90	5.14

Therefore, the conclusion here is: the model performs the best when iteration=250, and it is comparatively more stable with less standard deviation. A return of 10.51% is very satisfactory because the bond return is less than 2% and the average earning return rate in US market is negative for individual investors.

Because it's rather difficult in this scenario to use training and validation dataset to determine the best epochs, so I did several trials and just choose the best performing one.

Now, I want to add more factors and see if the model can learn about the interactions of useful technical indicators.

6.2 Results using 3 factor model

Parameters	Value	Explanation
train start date	2019-01-01	Use 2 years' data to train and 1 year's data to evaluate
train end date	2020-12-31	
test start date	2021-01-01	
test end date	2021-12-31	
window size	256	Since I am using 2 years' data, a larger window size is reasonable
skip	1	Use daily data to train
batch size	1024	
drop out rate	[0.3, 0.3, 0.3, 0.3, 0.3]	
GD learning rate	1e-5	
gamma	0.95	
epsilon	0.5	
epsilon min	0.01	
epsilon decay	0.9997	
no of factors	3	
iterations	20	20, 50, 250, 500

After training, here I put the result of one trial and one example of the agent's buying and selling points for the company CAT in a clearer way.

```
Ticker INTC: total gains -18.8100, total investment -15.46%
Ticker CSCO: total gains 11.4300, total investment 11.80%
Ticker DIS: total gains 39.9801, total investment 13.49%
Ticker AAPL: total gains 26.4125, total investment 17.59%
Ticker KO: total gains 2.1000, total investment 1.91%
Ticker V: total gains -7.4600, total investment -1.95%
Ticker MMM: total gains -14.7400, total investment -4.09%
Ticker IBM: total gains -36.9599, total investment -14.27%
Ticker TRV: total gains 21.8500, total investment 7.94%
Ticker CAT: total gains 87.0400, total investment 28.91%
Finally, with a principle of 100.00, we earned 4.59
```

Figure 10: 3 factor 1 trial result

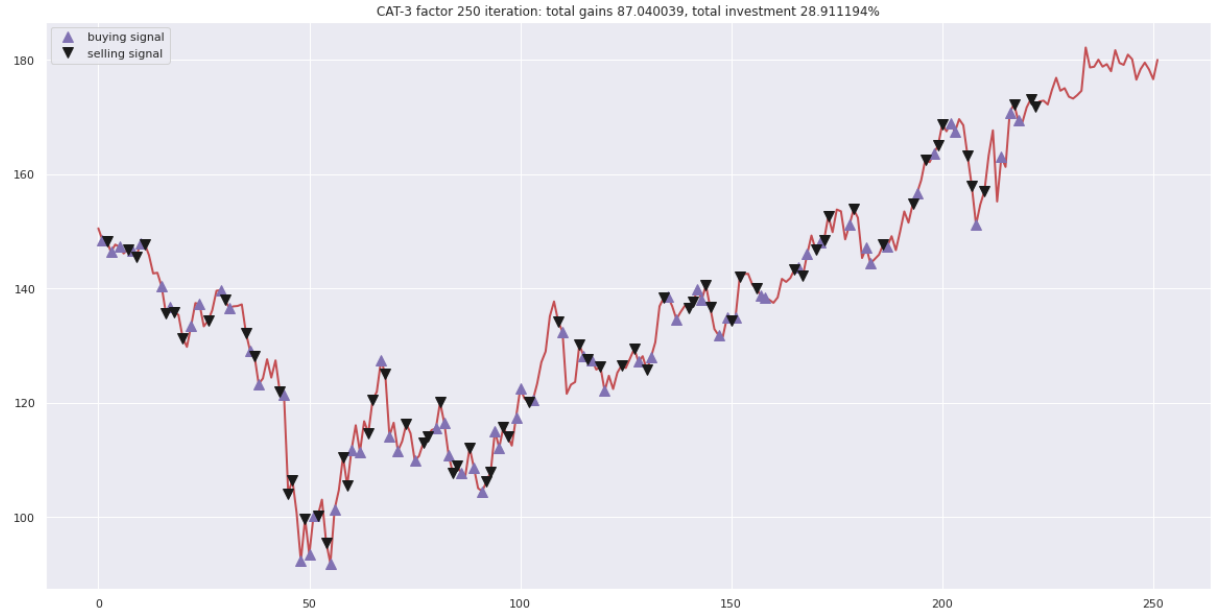

Figure 11: CAT - 3 factor 250 iterations Agent Buying and Selling Points

Here are the return results distributions:

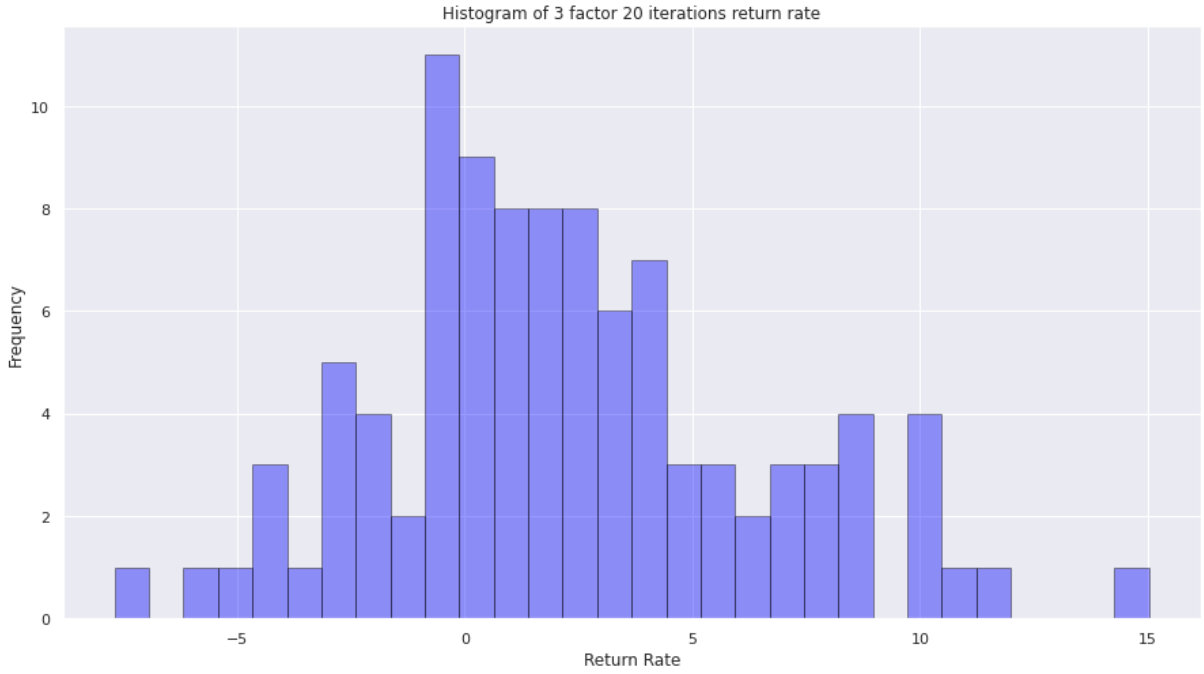

Figure 12: 3 factors 20 iterations result distribution

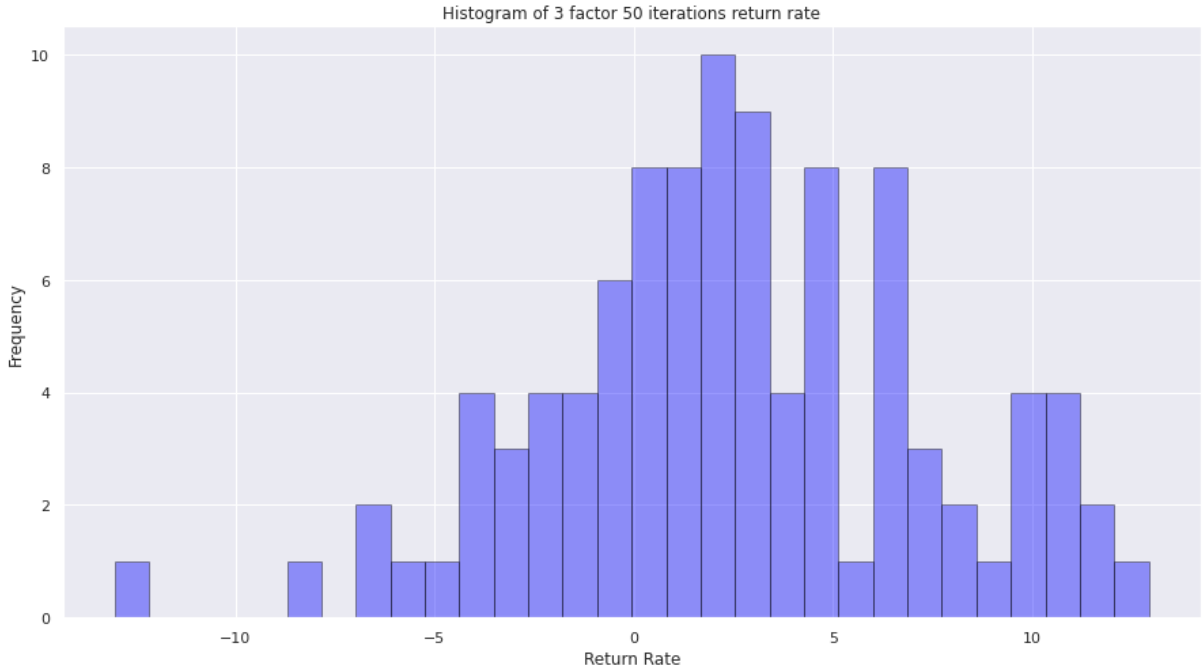

Figure 13: 3 factors 50 iterations result distribution

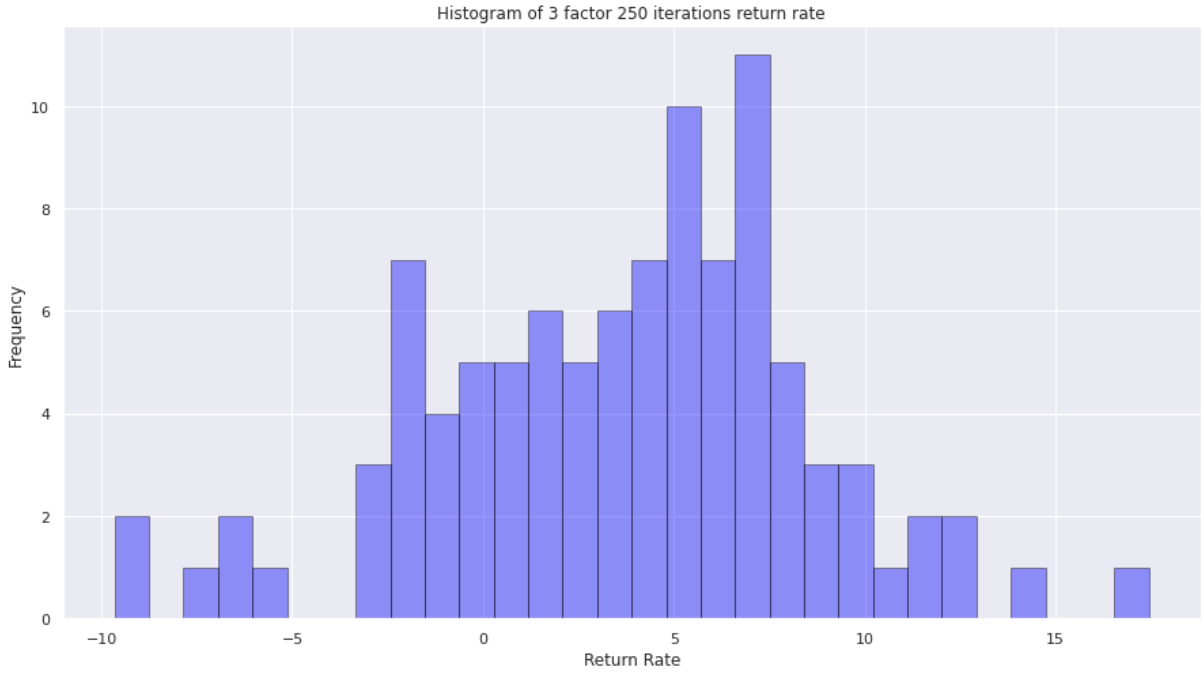

Figure 14: 3 factors 250 iterations result distribution

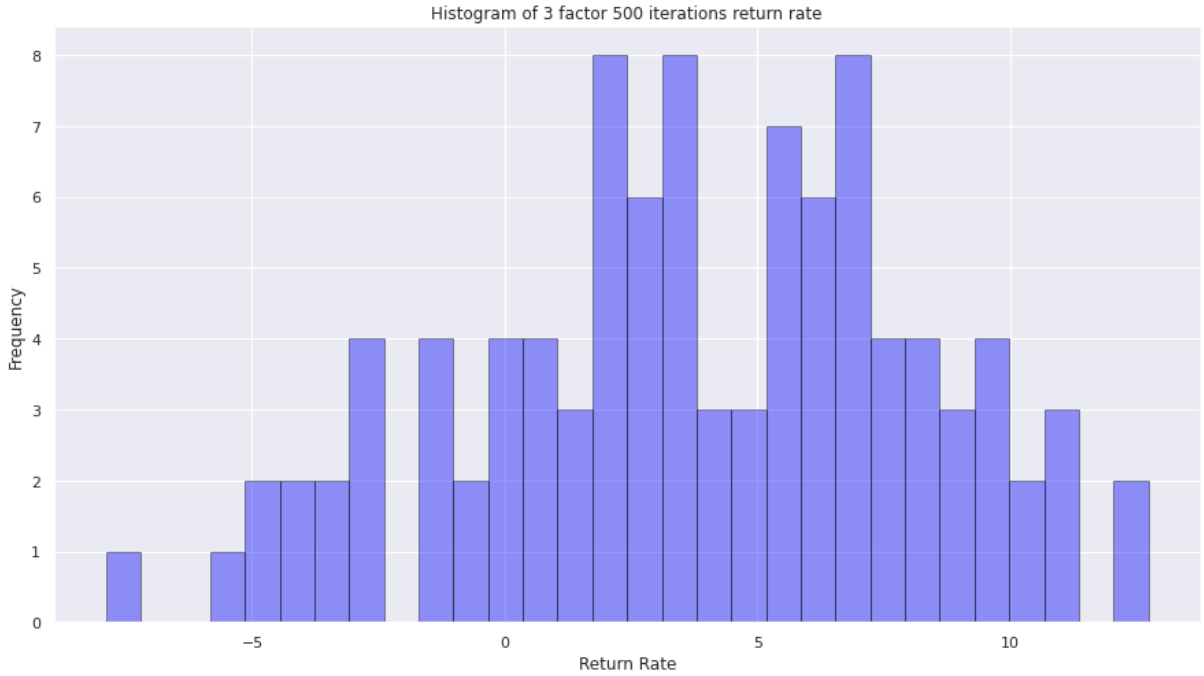

Figure 15: 3 factors 500 iterations result distribution

Upon the above 3 graphs, we can more clearly see the average is shifting to right, which is good because it means the agent is taking more information from the 3 factors and performed better and better with the number of episodes increased.

Here is the average and standard deviation of these 100 times' return results:

Iterations	Average Return Rate	Standard Deviation
20	1.30	3.86
50	2.52	3.56
250	3.64	3.41
500	3.77	3.10

Therefore, the conclusion here is: the model performs the best when iteration=500, and it is comparatively more stable with less standard deviation. A return of 3.77% is also very satisfactory because of the same reason as before, and in this 3-factor model, we see the standard deviation of each iteration scenario is all less than 1-factor model, indicating the model and return stability is better.

Now, I want to see if 6-factor model can have an even better stability and return rate.

6.3 Results using 6 factor model

Parameters	Value	Explanation
train start date	2019-01-01	Use 2 years' data to train and 1 year's data to evaluate
train end date	2020-12-31	
test start date	2021-01-01	
test end date	2021-12-31	
window size	256	Since I am using 2 years' data, a larger window size is reasonable
skip	1	Use daily data to train
batch size	1024	
drop out rate	[0.3, 0.3, 0.3, 0.3, 0.3]	
GD learning rate	1e-5	
gamma	0.95	
epsilon	0.5	
epsilon min	0.01	
epsilon decay	0.9997	
no of factors	6	
iterations	20	20, 50, 250, 500

After training, here I put the result of one trial and one example of the agent's buying and selling points for the company JPM in a clearer way.

```
Ticker MCD: total gains -12.4400, total investment -3.53%
Ticker TRV: total gains -33.2800, total investment -14.18%
Ticker AAPL: total gains 33.3000, total investment 42.17%
Ticker KO: total gains 20.5200, total investment 21.86%
Ticker CVX: total gains 27.9800, total investment 12.64%
Ticker V: total gains 10.2000, total investment 3.84%
Ticker CSCO: total gains 12.0800, total investment 14.06%
Ticker CAT: total gains -42.3100, total investment -16.74%
Ticker AXP: total gains 60.6800, total investment 31.71%
Ticker JPM: total gains -4.5200, total investment -2.28%
Finally, with a principle of 100.00, we earned 8.96
```

Figure 16: 6 factor 1 trial result

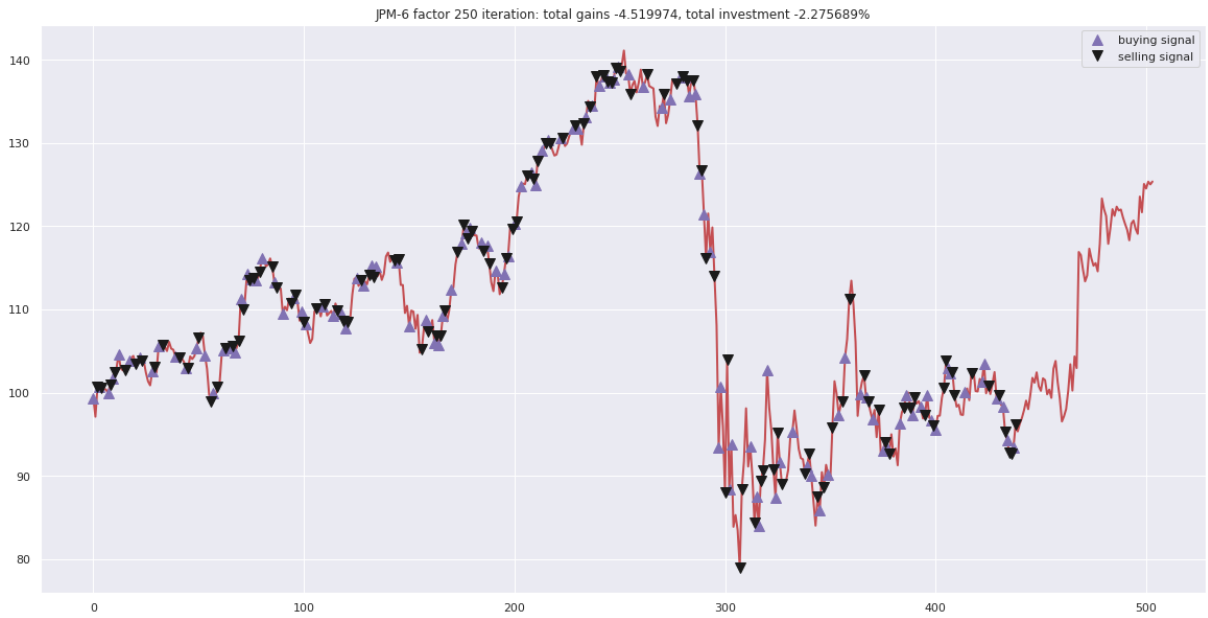

Figure 17: JPM - 6 factor 250 iterations Agent Buying and Selling Points

Figure 18: 6 factor 20 iterations result

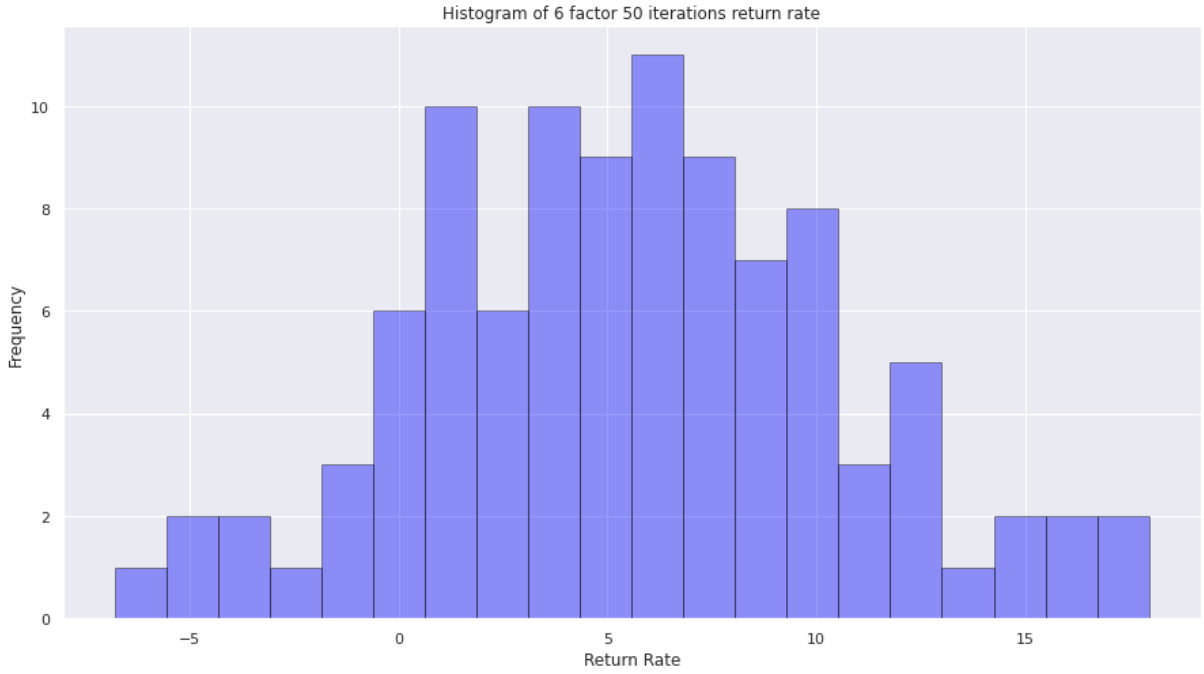

Figure 19: 6 factor 50 iterations result

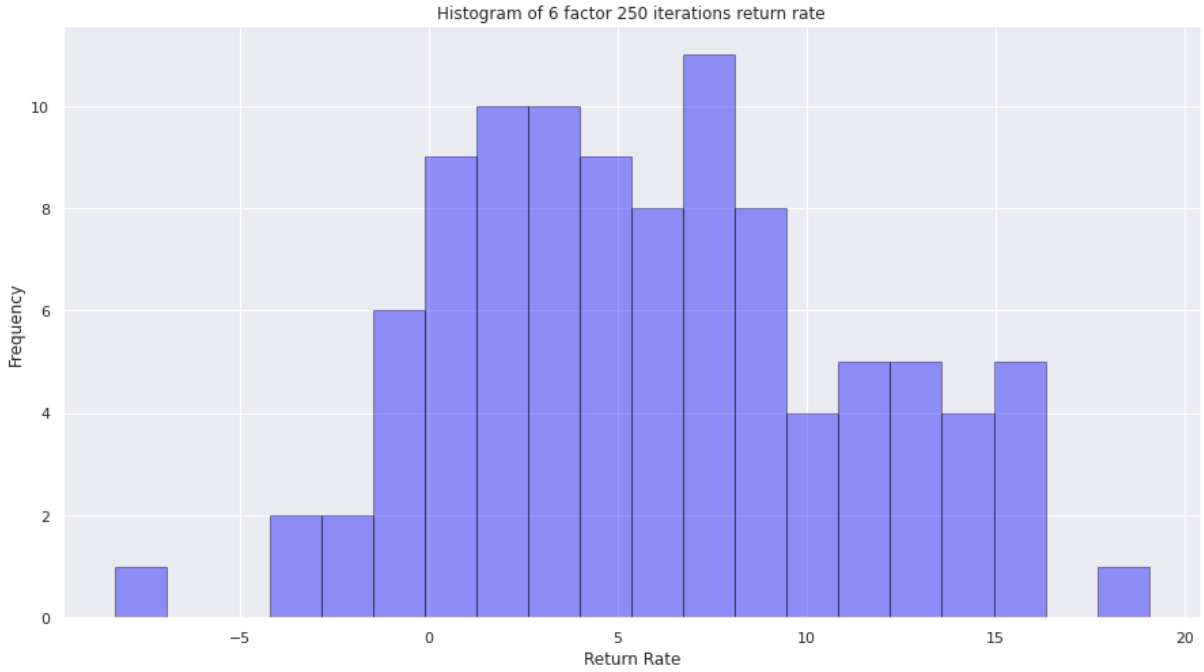

Figure 20: 6 factor 250 iterations result

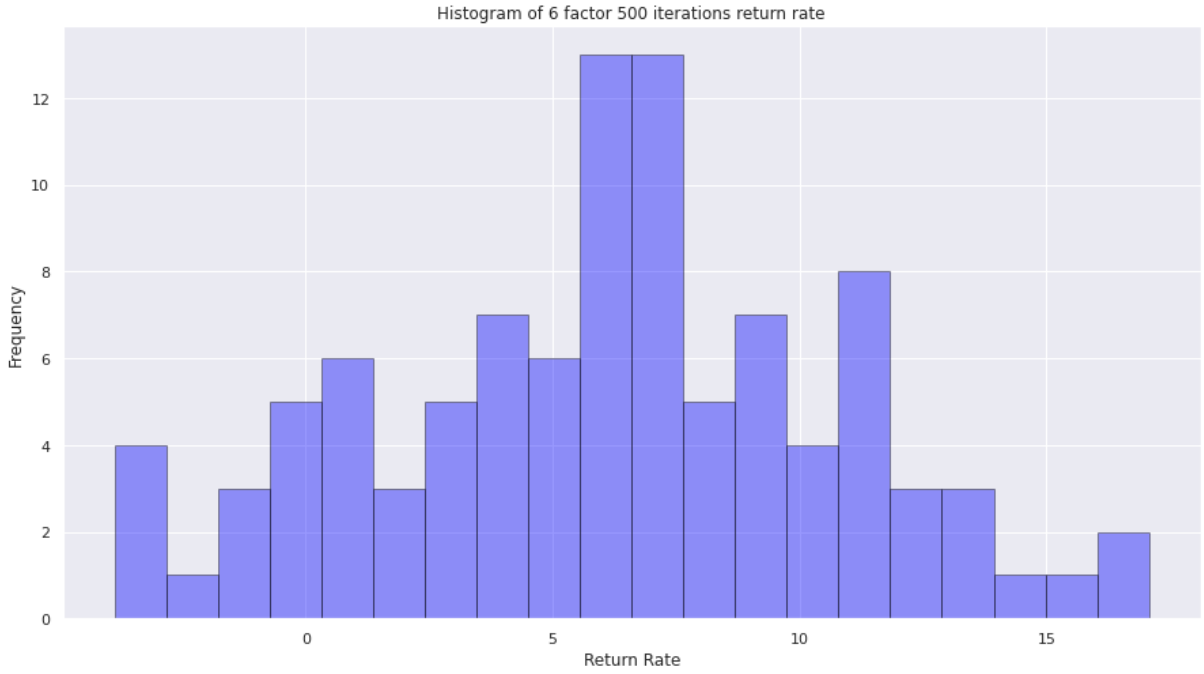

Figure 21: 6 factor 500 iterations result

Upon the above 4 graphs, we can clearly see, more iterations will result in a clearer and precise distribution plot, indicating the stability is better with the number of iteration increasing.

Here is the average and standard deviation of these 100 times' return results:

Iterations	Average Return Rate	Standard Deviation
20	8.63	4.52
50	6.57	4.40
250	6.97	3.41
500	7.57	3.39

Therefore, the conclusion here is: the best performing iteration here is 20, but since it is more volatile, I think iteration=500 has a better overall performance, since it reduces the draw down risk and tail risk, and there isn't a large difference between their absolute return rates.

7 State value optimization

I found that the return rate was quite reasonable or "ordinary" and I want something more "exciting". Since many companies are different in nature (for example, crude oil company stock price is almost completely negatively correlated to tech companies), the agent may learn a contradictory state and updates the wrong Q value. Here I plan to firstly cluster the companies based on their price actions and them train the agent on the largest cluster and evaluate the result.

7.1 Clustering Method

I choose to use KMeans to cluster the companies.

Firstly, extract the stock data and calculate daily price movements (close price of the second day minus

that of the first day), since this difference can basically show their different natures (positive movements and negative movements indicate the correlation between 2 companies is negative)

Secondly, use KMeans to cluster the DJI companies into 3 clusters.

Thirdly, I found that there are 2 largest clusters, and I will train separately on them. Here is the visualization of companies clustering (after PCA dimension reduction)

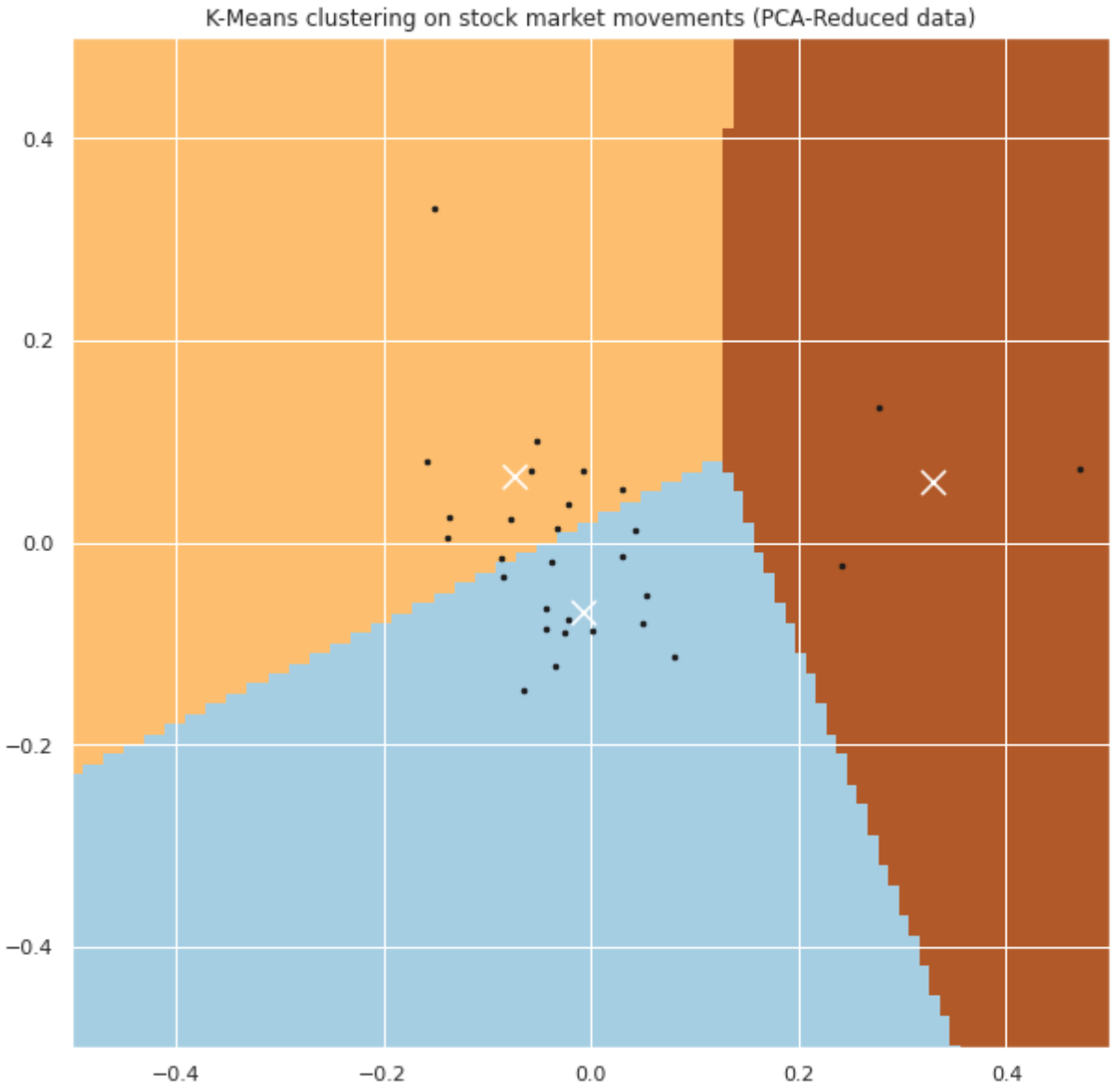

Figure 22: KMeans Companies Clustering After PCA

7.2 Train and test Cluster 1

Parameters are same as before:

Parameters	Value	Explanation
train start date	2019-01-01	Use 2 years' data to train and 1 year's data to evaluate
train end date	2020-12-31	
test start date	2021-01-01	
test end date	2021-12-31	
window size	128	Since I use less training companies
skip	1	Use daily data to train
batch size	1024	
drop out rate	[0.3, 0.3, 0.3, 0.3, 0.3]	
GD learning rate	1e-5	
gamma	0.95	
epsilon	0.5	
epsilon min	0.01	
epsilon decay	0.9997	
no of factors	3	
iterations	100	

Here is the result for one single trial and the corresponding buying and selling points:

```
Ticker V: total gains 150.4299, total investment 56.59%
Ticker JNJ: total gains -15.8800, total investment -6.22%
Ticker MCD: total gains 68.7700, total investment 19.53%
Ticker JPM: total gains 31.2101, total investment 15.71%
Finally, with a principle of 100.00, we earned 21.40
```

Figure 23: 3 factor 1 trial example on cluster 1

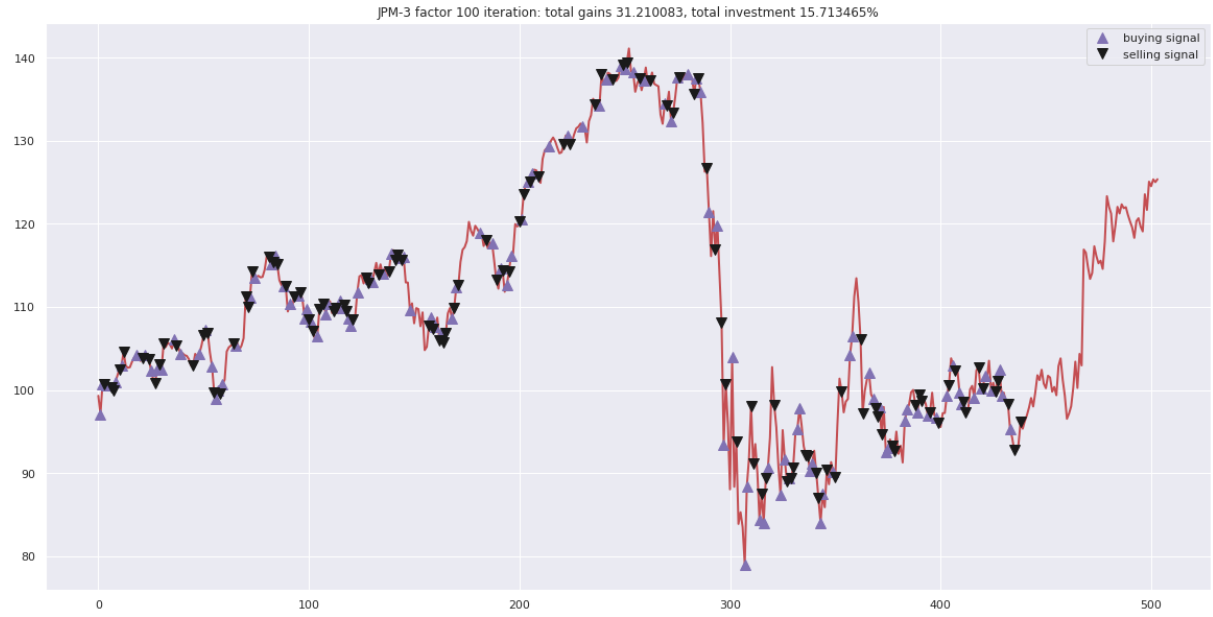

Figure 24: JPM 3 factor 100 iterations on cluster 1 buying and selling points

Compared with the previous JPM trading result, this time after clustering, the result becomes positive and

far better than before.

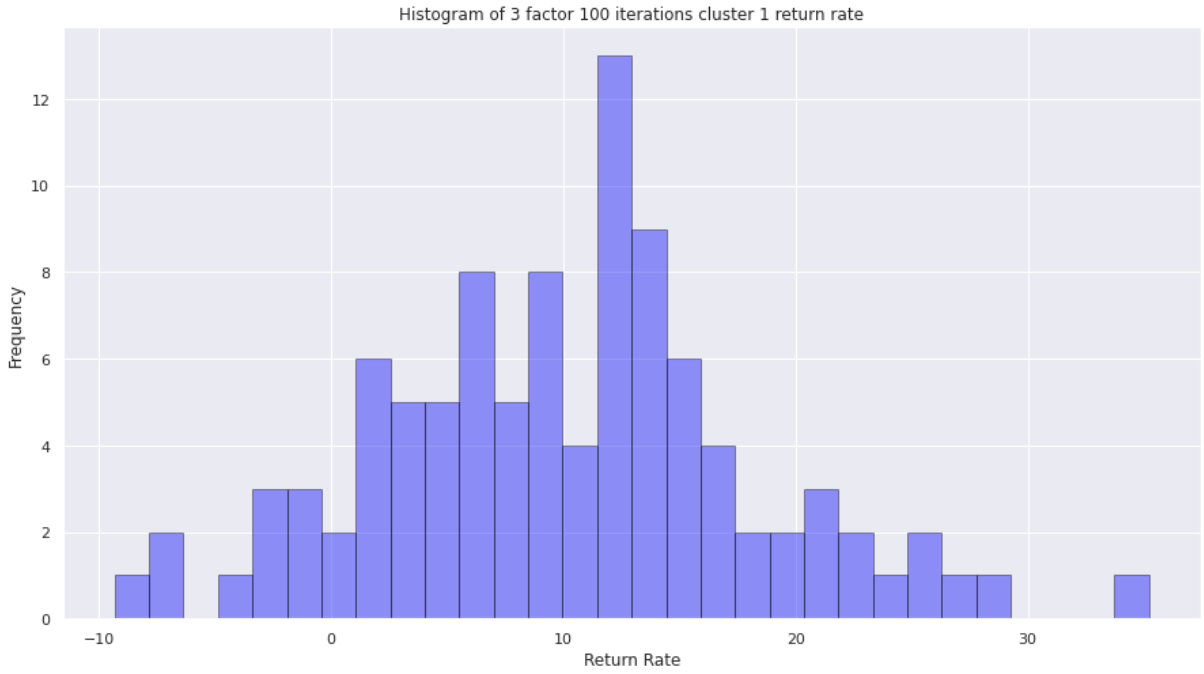

Figure 25: 3 factor 100 iterations on cluster 1 return rate distribution

7.3 Train and test Cluster 2

Here is the result for one single trial and the corresponding buying and selling points:

```
Ticker MMM: total gains -26.3201, total investment -6.89%
Ticker IBM: total gains -2.3997, total investment -1.09%
Ticker BA: total gains 308.8102, total investment 47.68%
Finally, with a principle of 100.00, we earned 13.23
```

Figure 26: 3 factor 1 trial example on cluster 2

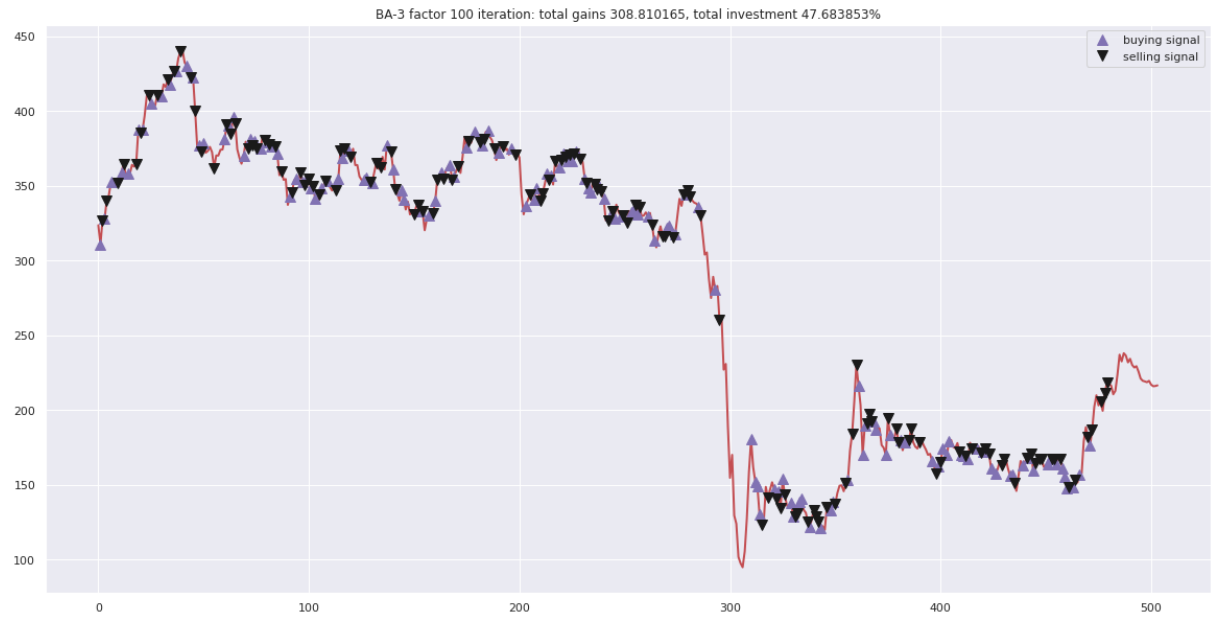

Figure 27: BA 3 factor 100 iterations on cluster 2 buying and selling points

7.4 Summary and Evaluation

Here is the average and standard deviation of these 100 times' return results:

100 Iterations	Average Return Rate	Standard Deviation
Cluster 1	10.08	8.09
Cluster 2	11.21	9.6

According to the above table, we can see that, after clustering the companies into 3 groups and choose 2 largest groups and split them into train and set dataset, we gained the best results so far: one that beats all the previous performances given same number of epochs and hyperparameters.

8 How to use this strategy in real life trading

Since when simulating on testing dataset, the return rates belong to a certain distribution, we can use this attribute to do real trading:

According to law of large numbers, the more trials we simulate, the distribution is more stable and more like a normal distribution. Therefore, in reality, we can simulate 10000 times, and store all the buying and selling points and amounts of each company in each time. Then we split our money into several parts, for example, if we have 5 test companies then we should split our money into 5 parts, and do real trading based on simulation results.

In this way, we can best utilize the model result distribution, and get a more stable return rate.

In theory, if we use the cluster 2 model (which has an average return of 10.08 and standard deviation of 8.09), then in each trial we have 90% possibility to have a positive return rate.

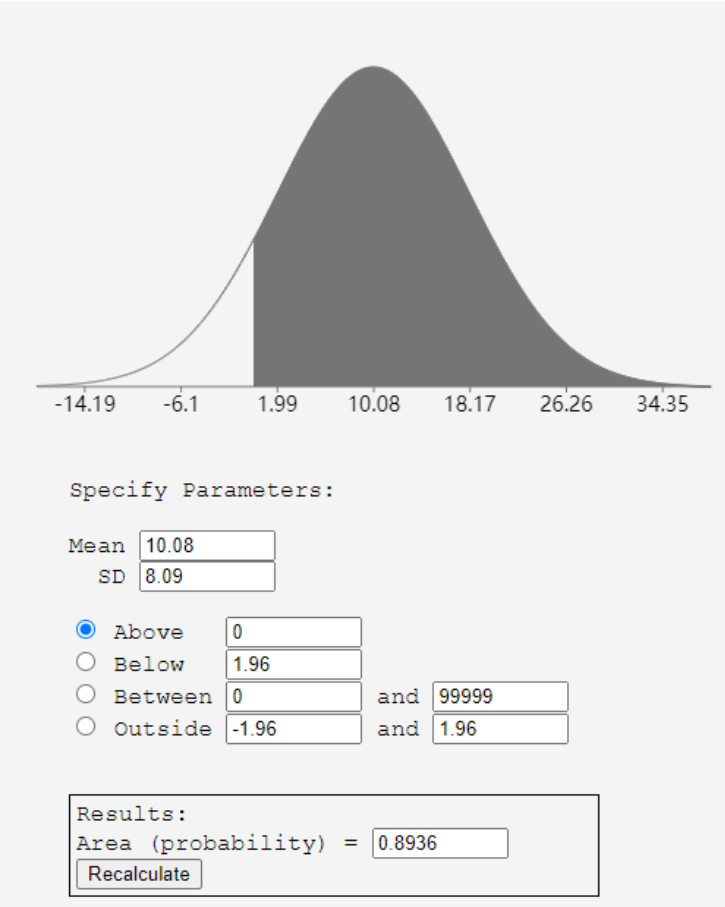

Figure 28: The expected return rate distribution in one trial

If we simulate for 100 times, then we almost can secure an average return (99.4% possibility), which is very stable and great.

9 Further Improvements

The model I used is not mature and too simple to handle the real financial market. I think improvements can be made in the following areas:

1) Make rewards calculation more complex. Now I only calculated rewards based on the final return, but since real market has some level of liquidity risk, I should also limit the trading frequency and each time buy a portion of our capital, not all. Therefore, in future, I can add punishments on the number of trades made and when I trade, let the agent buys a random portion of our capital and determine the best level of inventory in each state.

2) In real trading, I should not only focus on DJI stocks because it only has 30 companies, although they are the largest ones. Profitability comes from volatility and average return gap, so in theory smaller companies with less trading volume would easily result in arbitrage trading opportunities and I should filter the comparatively better ones from the 3000 Nasdaq companies and try our models again. The results should be better since they have larger volatility.

3) In real market people can both long and short, but in our agent I could only long. This is very important since short can offer you a profitable strategy no matter how bad the market is. This is easily down, by just add 2 actions (3 for shorting a stock, and 4 for closing that short position).

4) In our model prediction part, because of the nature of randomness, each time I have different return results and I need to repeat many times to reach the average yield. In future I should reduce this instability by building better deep learning model or adding more important factors (like correlation between the ticker and the mainboard index or ETF index etc.)

5) The factors I chose seems working well, but clearly there must be better factors that can yield better results. For example, macro data, news events and comments (NLP sentiment analysis on forum comments), correlation coefficients, analysts' opinions etc.

6) I did not spend much time on hyperparameter tuning on skip, drop out rate, initial epsilon these three, and I seriously think in different scenarios, for example, in training longer periods' data (5 years etc.), and if I use weekly data (skip=5), the predicted result would be far more smooth since daily fluctuations are so hard to predict but a longer period is easier (less frictions and irrationality).

7) For the Deep Learning network part, I used Dense (fully connected network with dropouts), which is a method that can automatically compute the combinations of the elements in the array. For time series data, I should also try RNN (use LSTM layers and treat each factor as an individual time series array).

8) Upon this model, I should not only train on 2019-2020 and test data in 2021, since I want to see if the model is stable in different periods, I should train multiple times from 2010-2011, 2011-2012, 2012-2013...until 2019-2020, and test correspondingly the data in 2012, 2013, 2014 until 2021, and check if the prediction returns list is similar and stable (similar average returns and similar standard variations).

9) When optimizing the input data, I can also use a rolling correlation coefficient or other time series similarity computation methods to find out historical trend pieces and their current corresponding pieces, and solely train on the historical similar pieces and test on the last similar piece. In theory, the result should be the best.

10) In real trading, there are many friction costs that are also should be considered, including: tax paid when selling a profited stock, commission fee when trades, price fluctuation caused by our agent buying and selling behavior (especially in Nasdaq small companies' stocks) etc.

11) In Clustering, here I used price Movements as the criteria to cluster. More advanced clustering methods can be used and more factors can be included to make this clustering more accurate. For example, use SVM or HDBSCAN if the factors are density based, which is very likely since the listed companies are usually either very similar (mostly) or very different.