Capstone Project

Machine Learning Engineer Nanodegree

In the final capstone project, I chose to implement the Q learning algorithms to design a stock trading program. The idea of the project came from papers of Google Deep mind which uses a Deep Convolutional Network to learn Q. I combined several knowledge on how to use Q learning for stock trading and the above mentioned deep Q learning network and finally designed my AI.
To train the program I chose some random range of time during the year 2016 to 2020 by simulating trades on the 506 stocks that constitute the S&P500 index. At the end I tested its performance using daily data since 2021 and contrasted the return of the traded portfolio with the SPY index.
The innovations of this project is that I will not teach the program what to do in each occasion because the number of possible combinations of stock prices is infinite. Instead I am going to give the program a brain, or a network of computer simulated neurons, and I am going to train the brain so it can infer what to do in each different possible combination of prices.

1. Model introduction

I referenced the algorithm in famous paper “Playing Atari with Deep Reinforcement Learning” by Google´s Deep Mind.

Q-Learning is a value-based algorithm in the reinforcement learning algorithm. Q represents the expectation of all future rewards for an action taken in a given state. So the main idea of the algorithm is to construct a Q table of state and action to store the Q value, and then select the action that can get the most profit based on the Q value.

While the states or possible cases in combinations of prices of stocks is almost infinite , the normal Q learning may never find an optimal policy. Thus the situation may be very similar to the case of playing Atari games. Instead of mapping a state-action pair to a q-value, it maps input states to (action, Q-value) pairs using deep neural networks(CNN) and a technique called experience replay.

Considering the characteristics of input data, I used a simpler version of Neural Network with hidden layers and a lineal output layer. Assume that Q(sj , aj) is our predicted value of Q, yj is our expected value of Q that can be calculated by the following formula:

$$yj = rj + \gamma \max_{a} Q(sj + 1, a)$$

where γ is the future reward discount.
Our target is to minimize the loss function of $(yj - Q(sj , aj))2$.

In order to do this, Deep Mind introduces “experience replay”. All the different transitions (st, at, rt, st+1) are stored in a set D. In each iteration, we can pass a random mini-batch subsampled from D to train the model, breaking similarities between each transition of states. I mimicked the above process by creating M different trade runs in random periods of times with random length. In addition, I followed the e-greedy exploration where e decays over time.
In summary, the algorithm can be concluded as follows:

Algorithm Deep Q Trader

Initialize preprocessed states $\varphi_t = \varphi(s_t,\varnothing)\ \forall\ s_t$
Initialize empty portfolio P to max capacity N
Initialize action-value function Q with random weights
for trade_run = 1, M **do**
  Select random range (T_i, T_f)
  Initialize set D to store samples
  for t = T_i, T_f **do**
    -With probability ε select a random action a_t, otherwise select a_t = maxarg Q* ($\varphi(s_t,P)$, a)
    -Execute action a_t buying, selling or holding designated stock and updating portfolio P.
    -Calculate portfolio return for period t and store as reward r_t
    -Update preprocess $\varphi_{t+1} = \varphi(s_{t+1}, P)$
    -Store transition (φ_t, a_t, r_t, φ_{t+1}) in D
    -Update decaying ε
  Set $y_j = r_j + \gamma$ maxarg $Q(\varphi_{j+1}, a)$ for each sample in D
  Perform a gradient descent step on $(y_j - Q(\varphi_j , a_j))^2$

2. Data preprocessing

The raw data we used are daily prices of 506 stocks that compose the S&P 500, downloaded from Yahoo finance. Each stock has the following information: Date Open, High, Low, Close, Volume and Adjusted Close.
I processed the data following the instructions on Udacity´s “Machine Learning for Trading” course. Since not all the stocks have coordinated trading dates, I defined the dates associated with SPY as all trading days where the market was open. Also, I chose the start of the training window at the year 2016, as more aged data may not represent recent patterns of market. To fill the missing data, we use the last known price of the stock.
The different scales of the prices of each stock makes them unsuitable to use them as inputs for a DNN. In order to scale them, I divided all the prices with the windowed mean and subtracted them by one to center the values to 0. The size of the window thus became one of the hyperparameters of the model. The other two features I employed are Bollinger’s bands and a feature to indicate if the stock is present in the portfolio. The lower Bollinger band is the windowed mean minus two times the windowed variance while the upper one is the windowed mean plus two times the variance. In order to reduce the dimensionality, I simply set the value to 1 if the stock price is over the upper Bollinger band, the values are set to -1 if the stock prices is under the lower Bollinger band and 0 if it is within both bands. Finally, I add a boolean variable for each stock to indicate whether it is already held by the current

portfolio.
After the above process, all the features will have approximately values between -1 and 1 and will be centered in 0.

3. Implementation

The implementation of the model requires more details design. The input of the DNN, i.e the states are composed by a list of centered prices and Bollinger band. I also want to include in the states the information for the last n days, thus n will also be a hyper-parameter in the model. As we can conclude that the number of features is generally three times of the stock number, but will vary with the number of n.

I tried two DNN structures to find the best model:

```
    """
    DNN 1
         Input Size   : STATE_SIZE
         Hidden 1 Size: (STATE_SIZE+NUM_ACTIONS)/2   relu
         Output Size  : NUM_ACTIONS                 linear

    DNN 2
         Input Size   : STATE_SIZE
         Hidden 1 Size: STATE_SIZE-(STATE_SIZENUM_ACTIONS)/3      relu
         Hidden 2 Size: STATE_SIZE-2*(STATE_SIZENUM_ACTIONS)/3    relu
         Output Size  : NUM_ACTIONS                              linear
    """
```

The output of the DNN is our predicted Q(sj , aj), thus the output number equals all possible actions. We use boolean variables corresponding to buy, hold and sell for each stock, so there is a total of 506*3 = 1518 different outputs in our DNN.
The training of the DNN requires M iterations. In every iteration, we selected a random point in time to start our trade, the automatic trader will then simulate a trade that lasts a random number of days between 30 and 120. Initially the portfolio starts empty and the value of e starts in 1.
After starting a trade, the model will choose actions depending on the value of e. Although we decay the value of e from 1, I set a minimum value of e which is 0.1. So the model may either choose a random valid action or look for the valid action that has the largest predicted value of Q. The valid action ensures us to not waste time learning Q values for actions like selling or holding a stock that we don´t have in the portfolio, buying a stock we already have or buying a stock when the portfolio is full. I realized the action selection process by functions.

```
#Check if an action is valid
def _check_valid_action(self,state,action_index):
    holds=state[-self.STOCK_COUNT:]
    index = int(action_index / 3)
    a = action_index % 3
    if a==0 and len(self.portfolio)>=self.MAX_PORTFOLIO_SIZE:
        return False
    if holds[index] == 0 and a > 0:
        return False
    if holds[index] == 1 and a == 0:
        return False
    return  True
```

```
#Choose next action
def _choose_next_action(self,state):
    new_action = np.zeros([self.ACTIONS_COUNT])
    self.action_index=0
    if random.random() <= self._probability_of_random_action:
        # choose an action randomly
            self.action_index = self._get_random_valid_action(state)
    else:
        # choose an action given our last state
        readout_t = self._session.run(self._output_layer, feed_dict={self._input_layer: [state]})[0]
        ordered_index= np.argsort(readout_t)
        for i in range(len(ordered_index)):
            self.action_index=ordered_index[len(ordered_index)-i-1]
            if self._check_valid_action(state, self.action_index):
                break

    new_action[self.action_index] = 1

    return new_action
```

After choosing the action, the portfolio is updated accordingly. With the updated portfolio we can calculate the next state by getting the new prices and calculating the new holds with the updated portfolio. Note that each stock in the portfolio is equally weighted and the maximum number of stocks a portfolio can hold is 20.
When one iteration of trade run ends, all the transitions (st, at, rt, st+1) stored in D will be used to train the DNN with stochastic gradient descend.
The complexity of the model implementation mainly came from the difficulty of calculations: the states are very big so it may take ages to calculate all the next states given an action. To solve this, I precalculated all the possible states and gained a significant boost in speed.

4. Tuning of the hyperparameters

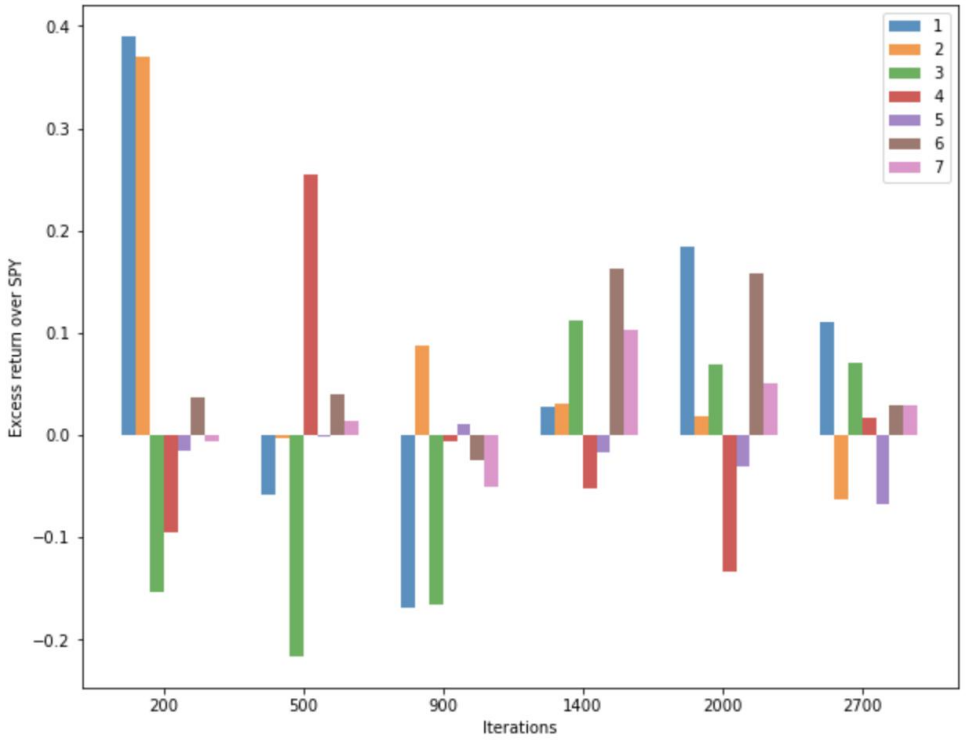
I started with iterating among the best combination of the number of hidden layers, number of days in state and the number of trade runs. From my perspective, these are the most important hyper-parameters of the model. The test run is set through 01/01/2021 to present.
Later, I went through a grid search over the combinations of learning rate and the window size for both the Bollinger´s Bands and the moving average. To prevent direct overfitting, I set the test window to another period: from 01/01/2019 to present.
Finally, a Roll Forward Cross Validation was implemented for the potentially best models selected from the first two steps. The total data set was divided into 18 folds and the final model will be determined by the average portfolio return of the last ten folds.

Through observing the trading logs of the agent, I recognized two different strategies patterns: Either it chooses to buy and hold or it trade certain stocks and do timing trades. These are also the two most popular trading strategies.

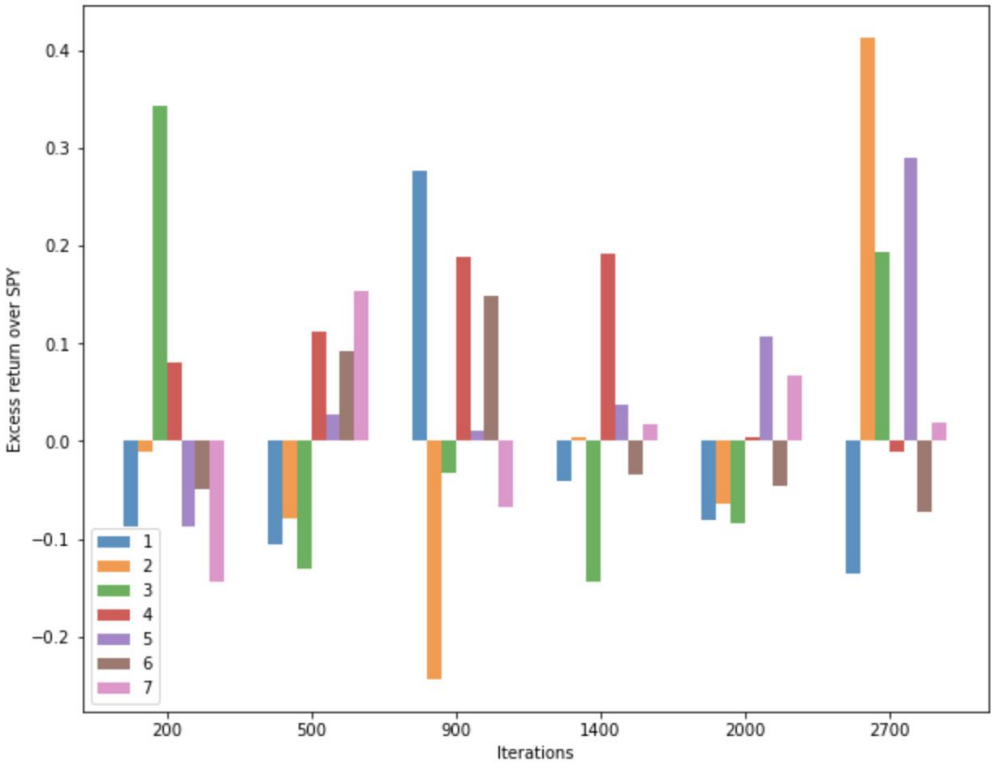
The buy and hold strategy is more lasting. The agent will pick a couple of stocks in the beginning of a trade run and keeps them in the portfolio till the end. The buy and hols strategy usually appears in the cases with more iterations and are commonly with great results.
The timing strategy is based on consecutive trading, and the agent would tend to continuously altering the composition of its portfolio. It is based on a s simple notion to buy and sell in the right moment. The average return of the timing strategy is less surprising than the long-term one, as sometimes the agent perform really poorly. Although the winning rates of the agent are significantly higher than 50% in most cases, it is rather rare for AI to outperform the SPY index.
Among the first two rounds of iterations, the best performers are often mixing the two strategies.

The following two figures show the test returns of different combinations of hyper-parameters: The first figure is a DNN with one hidden layer, while the lower one is a DNN with two hidden layers. After limiting our options in the first grid search, I continued to iterate through learning rate and moving average window size.

	200	500	900	1400	2000	2700
1	0.390373	-0.058439	-0.169196	0.027436	0.183892	0.110990
2	0.369727	-0.003421	0.086899	0.030997	0.018022	-0.062396
3	-0.152914	-0.216407	-0.165971	0.112808	0.068469	0.071178
4	-0.094615	0.254453	-0.005674	-0.052358	-0.133147	0.017261
5	-0.015077	-0.001801	0.010109	-0.017374	-0.031200	-0.067200
6	0.037413	0.039315	-0.024396	0.162081	0.157456	0.029433
7	-0.006665	0.014428	-0.050263	0.103395	0.050597	0.029502



	200	500	900	1400	2000	2700
1	-0.087135	-0.106489	0.276004	-0.041486	-0.080989	-0.135638
2	-0.010844	-0.079447	-0.243861	0.004190	-0.064632	0.413494
3	0.343632	-0.131534	-0.033066	-0.144047	-0.084532	0.193017
4	0.080289	0.112531	0.188707	0.191068	0.004401	-0.011432
5	-0.087467	0.027959	0.010231	0.037089	0.107285	0.289840
6	-0.048865	0.091543	0.147981	-0.034748	-0.046117	-0.072508
7	-0.144047	0.153713	-0.067504	0.016543	0.067596	0.018614



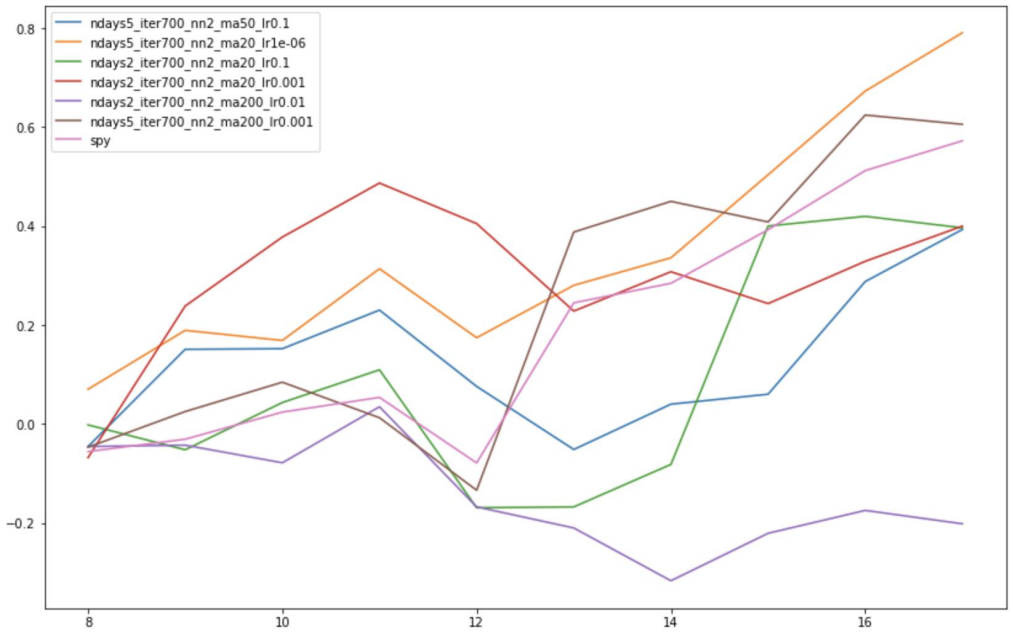
Some patterns are apparent. The agents with more simple structures tend to prefer a buy a timing strategy and a better performance was positively associated with a higher learning rate. Despite that they can bring unexpectedly great results, the performance is unstable and volatile.

Meanwhile a two-hidden layer model would have better performance with lower learning rates. A deeper network more more parameters seems to give the agents a better capacity of recognizing high-value stocks.

After my observation, the hyperparameters of the last five best candidates are as follows: [[5, 700, 2, 50, 0.1], [5, 700, 2, 20, 0.000001], [2, 700, 2, 20, 0.1], [2, 700, 2, 200, 0.001],[5, 700, 2, 200, 0.001]], representing accordingly the number of days in state, trade runs, number of hidden layers, window size and finally, learning rate. Later I did a final k-folds validation and choose the [5, 700, 2, 20, 0.000001] and [5, 700, 2, 200, 0.001].

5. Result analysis

To put all in a nutshell, in this capstone project I synthesized two popular models in Machine Learning, Q Learning and Deep Neural Networks to produce an agent that is expected to automatically trade stocks. The agent is trained by several random trade runs between 2016 and 2020 and was tested in a test set of recent data. Through a thorough grid search, I selected the best performing model and got significantly better results compared to the SPY. Moreover, this result was consistent over time.



One of the interesting discoveries I found in the project is the different types of trading strategies the agent matriculate depending on the status. Some models tend to prefer buy and hold or others prefer timing strategy. I am very impressed by the performance of the agent, especially after 2700 iterations. Many of them had attained a healthy mix of both proving to be very flexible to different situations.

Improvement

There are some improvements that we can expect in the future. For example, the training sessions are time-consuming due to the large dimension of the state and action. Thus I did not manage to look further into the model, for example I only saw how the model behave with two hidden layers, maybe with more layers I could have gotten other results. What is more, one drawback of the deep learning model is that we can only view its inputs and outputs without any knowledge of its internal workings. Its implementation is "opaque" (black).

I have some ideas about future improvements of the model. The first idea is to include the current return as input features and feed them into the model. Usually

when trading we has stop loss rules or cash in rules that tells us when to stop, probably if the agent develops this sort of strategies it can hugely outperform the current agent.

I also had some ideas about dimensionality reduction. I am thinking of using just small proportion of the stock in S&P index by implementing certain filters. Either we can choose the stocks that are most traded or have better recent performance. What is more, if the stocks that we selected have higher correlation between one another, the model will converge much faster.

Finally, in addition to the three features in the model, macroeconomic factors and traditional financial factors should also be considered in account. If possible, incorporating common alpha factors in quantitative finance or even natural language processing will be a boost to the model.