SCIE 3500 Final Report:
Bitcoin Trading Agent with Deep Q-Learning Algorithms

Student: Wang, Xinyi
Supervisor: Prof. Yao, Yuan

November 12, 2019

Abstract

This is the first half of my capstone project which is a continuation of the RIPS-HK 2018 group project. In this project, I focus on trying different ways to implement the deep Q-learning algorithm of the Bitcoin trading agent. Two kinds of neural networks are used: ResNet and recurrent neural network (RNN). I tried two approaches to implement the Q-network: the first one is to use the Q-network to map the state to the Q-values of all actions, the other one is to use the Q-network to map both the state and a specific action to a single Q-value. It turns out that the latter scheme has a better performance in this project. And instead of using the past experience to do memory reply, I tried to use the pre-known optimal policy which turned out to have a much better performance.

1 Introduction

In the year 2017, terms like "Bitcoin", "cryptocurrency" and "blockchain" became very popular. One of the reasons behind this is because that there was a very high peak in the price of Bitcoin. Investors can get good profits by simply buying some Bitcoins. But since the beginning of year 2018, the price of Bitcoin has kept dropping and inverstors are losing money in Bitcoin market. It is said that the winter of cryptocurrency has come. But theoretically, it is still possible to make profit in a bear market as long as we can buy when the price is at a relatively low place and sell when the price is at a relatively high place.

The idea is to tackle this problem using reinforcement learning. As shown in figure 1, there are two core elements in reinforcement learning: agent and environment. And the agent learns to act in a way to maximize the reward by interacting with the environment.

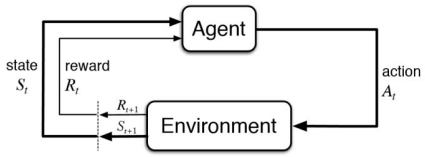


Figure 1: Reinforcement Learning (RL) [1]

To formulate this problem into a reinforcement learning setup, I define the state by a window of the continuous price series including the current Bitcoin price and the previous 180 prices.

Figure 2: S_t and A_t

The Bitcoin price data is obtained from Kaggle [5] and it is reduced from per minute prices to per 30 minute prices.

The whole data set has about 9k continuous Bitcoin prices. And to train the model, we used the first 7k data as the training set and the last 2k data as the testing set. Below is a plot of the Bitcoin prices in the training set (green) and testing set (red), from which we can see that the Bitcoin price time series is non-stationary. And this is a major challenge to the (deep) Q-learning algorithms in this report as the i.i.d. sampling assumption is violated.

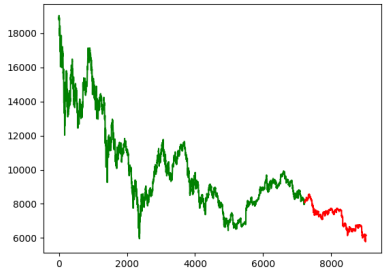


Figure 3: Bitcoin price time series

I simply defined two actions, BUY and SELL. And the reward function is defined as follow:

$$R_t = \begin{cases} p_{t+1}/p_t - 1, & A_t = BUY. \\ -p_{t+1}/p_t + 1, & A_t = SELL. \end{cases}$$

Where p_t is the Bitcoin price at the t-th timestep.

Note that in this setting, I assume that the the result of each action at any state is determinant. That is

$$P(S_{t+1}, R_t | S_t, A_t) = 1$$

2 Q-learning

In Q learning, we define the accumulated reward as $\sum_t \gamma^t R_t$ and use a Q function $Q(S_t, A_t)$ to approximate the maximum accumulated reward, which can be updated by the Bellmen equation:

$$Q(S_t, A_t) = R_t + \gamma \max_h Q(S_{t+1}, A_h)$$

The Q learning algorithm (i.e. value iteration) is therefor as follow:

Initialize $Q(s,a)$ arbitrarily
Repeat (for each episode):
  Initialize s
  Repeat (for each step of episode):
    Choose a from s using policy derived from Q
    Take action a, observe r, s'
    Update
      $Q(s,a) \leftarrow Q(s,a) + \alpha[r + \gamma \max_{a'} Q(s',a') - Q(s,a)]$
    $s \leftarrow s'$;
  Until s is terminal

Figure 4: Q learning algorithm [2]

In the RIPS-HK 2018 research program, we first tried to implement a Q learning trading agent by dynamic programming. This means that we discretize the state defined previously.

To do this, we used the rolling means and standard deviations in the scales of 24, 72, 168 and 720 to fit the current price into three slots. An illustration of using the scale of 24 is shown below:

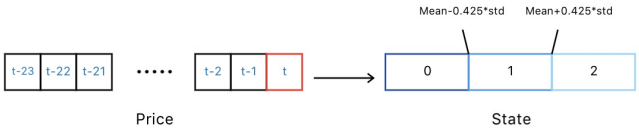


Figure 5: Discretize a continuous price series of 24 into 0, 1, 2

Then in this way, we have 3^4 states in total and each state has 2 actions, so we only need to keep a Q-table of size 162.

The output actions on the testing data is stabilized after a few epochs of training and the change of the values of the Q-table is within 10^{-10}.

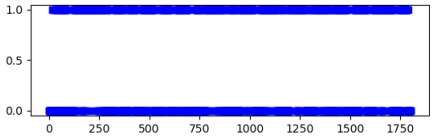


Figure 6: Output actions on the testing data

In figure 7, 0 represents the action BUY and 1 represents the action SELL. We can see that the agent changes its action quite frequently. To evaluate its performance, I designed a simple task: With a certain fixed amount of initial asset, let the agent interact with the test environment. If it takes action BUY, then at least 90% of the total asset is transferred into Bitcoin market. If it takes action SELL, then at most 10% of the total asset is transferred into Bitcoin market. The figure below illustrate the result of this simple task:

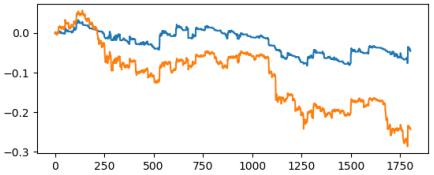


Figure 7: Price changes and asset value changes on the testing data

The orange curve is the percentage price change comparing to the initial price. And the blue curve is the percentage change of the asset value comparing to the initial asset.

As we can see, the Q learning trading agent can avoid large drops in the price changes but cannot take much advantages on the rising edges of the price. And this can serve as a baseline of this project.

3 Deep Q learning

In traditional Q learning, in order to keep a Q table, we can only have finitely many states. But if we approximate the Q function by a neural network instead of a table, we can have infinitely many states.

At t-th training step, the forward propagation can be formulated as follow:

$$y_t = R_t + \gamma \max_{A_h} Q(S_{t+1}, A_h, \theta)$$
$$\mathcal{L}(\theta)_t = (y_t - Q(S_t, A_t, \theta))^2$$

Where θ is the parameters of the neural network that we want to learn. y_t is the target value of $Q(S_t, A_t, \theta)$. $\mathcal{L}(\theta)_t$ is the loss function. Then we can do back propagation by gradient descent accordingly:

$$\frac{\partial \mathcal{L}(\theta)_t}{\partial \theta} = -2(R_t + \gamma \max_{A_h} Q(S_{t+1}, A_h, \theta) - Q(S_t, A_t, \theta)) \frac{\partial Q(S_t, A_t, \theta)}{\partial \theta}$$

$$\theta \leftarrow \theta - \alpha \frac{\partial \mathcal{L}(\theta)_t}{\partial \theta}$$

There are two ways to implement the Q-network. The first way is to use the Q-network to map both the state and a specific action to a single Q-value, i.e. $(S_t, A_t) \rightarrow Q(S_t, A_t)$ at each timestep t. The other way is to use the Q-network to map the state to the Q-values of all actions, i.e. $S_t \rightarrow (Q(S_t, A_1), Q(S_t, A_2), \ldots, Q(S_t, A_n))$ at each timestep t, where n is the total number of actions.

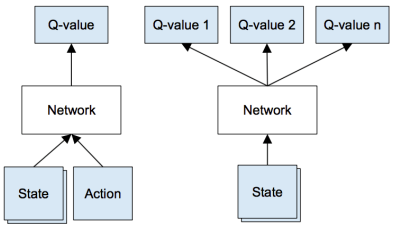


Figure 8: Two ways to implement the Q-network [3]

The second way is the most widely used one, which I think is because that it separates the actions from the inputs such that it only need to go through the whole network once to learn the Q values of all the actions at S_t. But in this way the Q network actually maps S_t to A_t since we will need to choose $\max_{A_h} Q(S_{t+1}, A_h, \theta)$ to train the network. And the other way preserves the original structure of the Q function which may give us a better result.

In details, I used the following algorithm to implement the deep Q learning trading agent:

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 ϵ 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 $(y_j - Q(\phi_j, a_j; \theta))^2$
  end for
end for

Figure 9: Deep Q learning algorithm [4]

Since the variance will be high if the network is trained with only one datum at each timestep, I used another technique called memory replay to train the Q network by batch. According to our back propagation equation, we only need $\{R_t, S_t, A_t, S_{t+1}\}$ to perform a back propagation. So we store a queue of $\{R_t, S_t, A_t, S_{t+1}\}$ generated through the training process and then we can sample a batch of data from to train the network each time.

To avoid the Q network to quickly converge to a local maximum at the beginning of the learning process by repeating its own experience while doing the memory replay, I used a technique called exploit and exploration, which means instead of choosing the action corresponding

to the maximum Q value at each state, I make random decisions with a probability ϵ. And this ϵ is decaying linearly from 1.0 to 0.001 such that the algorithm will stabilize in the end.

4 Implementation

I use the Huber loss as the loss function since it is more robust. And I use stochastic gradient descent(SGD) with a learning rate of 0.001 and a batch size of 32 to do the back propagation.

The discount factor γ is set to 0.95.

Since I am doing random sampling, I assume there are `total_num_data/batch_size` = 225 steps in one epoch.

4.1 Additional Information

To feed the Q network with more information, I use the same set up of state as in the section 1, but with two more channels: sentiment and MACD.

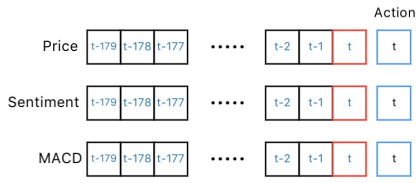


Figure 10: State with three channels

One additional channel is the sentiment information from Reddit, which was collected by my RIPS-HK teammate Katherine Thai. Reddit is a social media site where users can submit many different forms of content to specific communities known as subreddits. Using the Reddit Pushshift API [7], we collected the number of new posts in the Bitcoin subreddit [6], containing the word "hack" in the past 30 minutes for the same time span as the price data. Since the Bitcoin trading is based on the blockchain technology, if it is hacked then it is very likely that the price of Bitcoin will drop.

The other additional channel is the Moving Average Convergence Divergence (MACD) of the price data. The MACD is computed by subtracting a longer period exponential moving average (EMA) from a shorter period EMA. In our case, we take the longer period of 48 and shorter period of 12 as shown below.

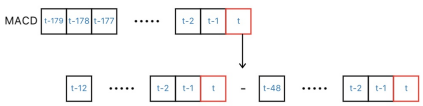


Figure 11: The Moving Average Convergence Divergence (MACD) channel

4.2 Q network

In this project, I used two different kind of neural networks: the ResNet and the recurrent neural network (RNN).

4.2.1 ResNet

ResNet is a kind of convolution neural network but uses residual blocks as its basic units which directly pass the input to the output layer.

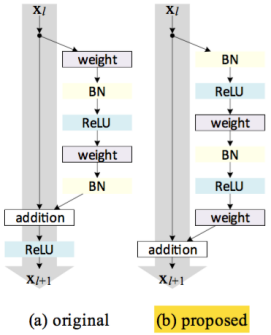


Figure 12: Residual block structure [9]

Since we only perform addition for every shortcut at the end of the residual block, so when we connect all the residual blocks together, there is actually a shortcut from the input layer to the output layer.

The intuition behind this is that when the neural network becomes deeper and deeper, gradients of the layers near the output layer vanishes so they do not contribute to the the final layer and even harm the final result. If we skip those layers, we may can get a better result. And this is equivalent to adding an identity shortcut to the output. So the function we want to learn becomes:

$$H(\boldsymbol{x}) = \boldsymbol{x} + F(\boldsymbol{x})$$

F is the function we actual learned from the neural network. This can also be understood as that the true underlying function H is usually more close to an identity function, rather than a zero function.

The implementation of the ResNet is adapted from [10], which is based on Keras. I changed all the 2-D convolutions into 1-D, and build a ResNet with 18 layers, which has 8 residual blocks.

In my implementation, each block has two 1-D convolution layer, all of their filters are of length 3. Each convolution layer in the first 2 blocks all has 64 filters. In the following blocks, the number of filters doubles per 2 blocks. I also added two dense layers before the output layer.

4.2.2 Recurrent neural network (RNN)

A RNN layer accept the input as in a sequence of timesteps. It not only output to the next layer, but also output to itself at the next timestep, which is called the hidden state. Below is an illustration of an unrolled RNN.

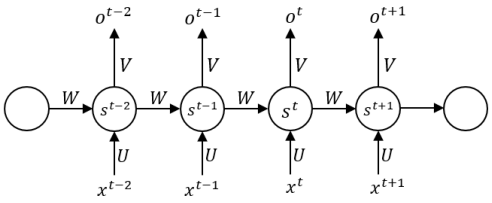


Figure 13: An unrolled RNN

Note that the weights connecting back to itself are shared between each timestep so it can capture the relation of the input data between different timesteps.

In my implementation, I used the gated recurrent units (GRU) as the basic RNN unit, which is illustrated as below:

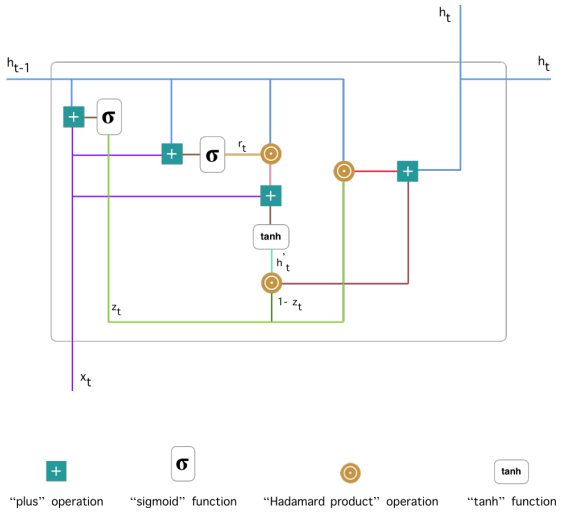


Figure 14: A GRU cell [11]

And the structure of my whole network is as follow:

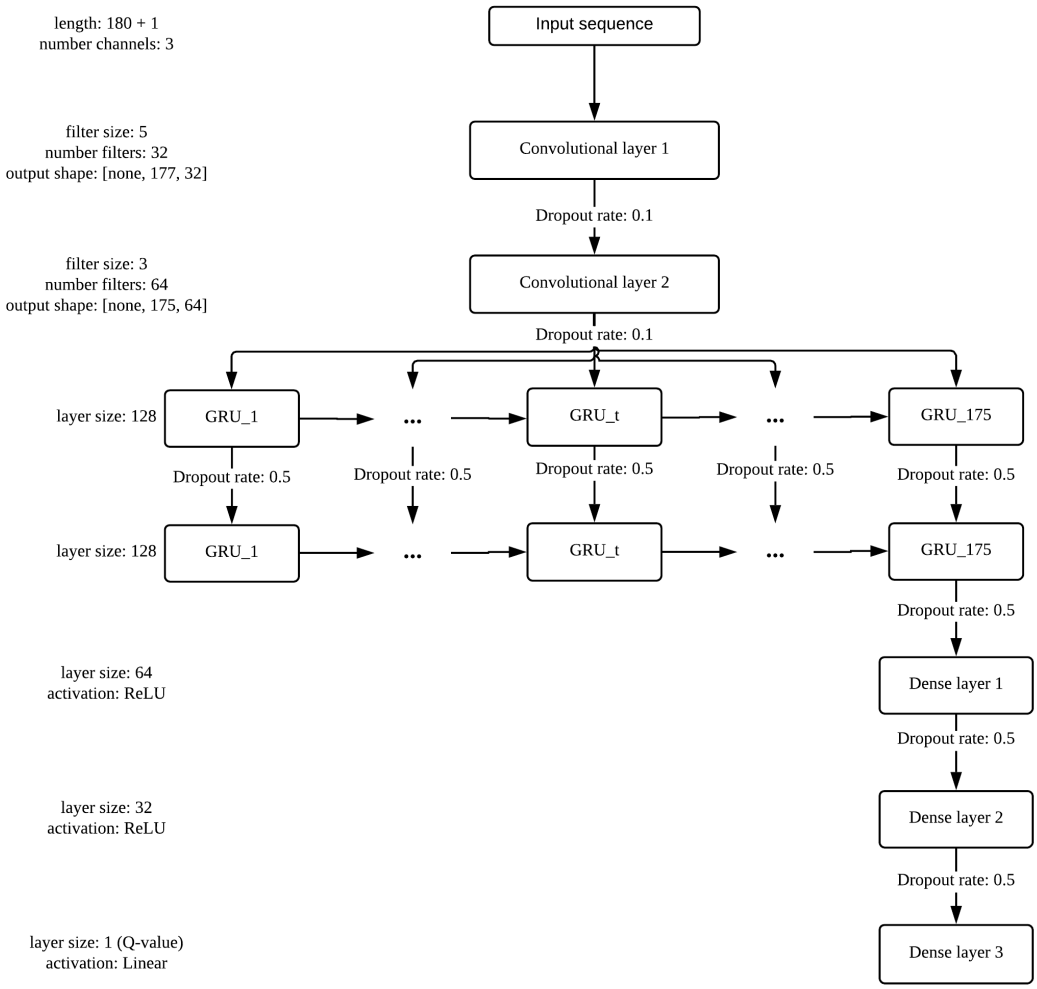


Figure 15: Network structure

5 Experiments

5.1 Scheme 1

I used the ResNet to approximate the Q function $Q: S_t \to (Q(S_t, A_1), Q(S_t, A_2), \ldots, Q(S_t, A_n))$. And the replay memory is gained along the training process, i.e. the decisions made in the past. Below is the output actions on the testing data and the result of the simple evaluation task in section 2:

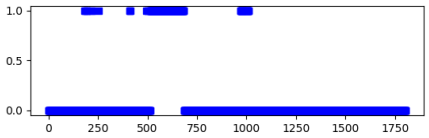


Figure 16: Output action on testing data

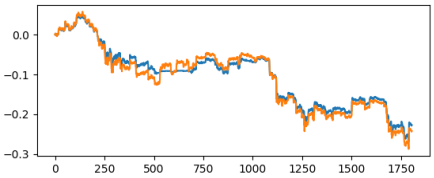


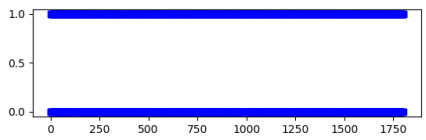
Figure 17: Price changes and asset value changes on the testing data

The orange curve is the percentage price change comparing to the initial price. And the blue curve is the percentage change of the asset value comparing to the initial asset.

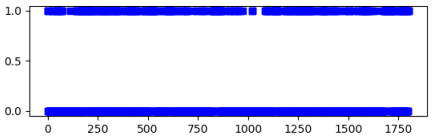
We can see that the agent mostly took action BUY(0) but that is not a good choice for a bear market. It performs worse than the Q-learning base line.

5.2 Scheme 2

I used both the ResNet and RNN to approximate the Q function $Q: (S_t, A_t) \to Q(S_t, A_t)$. And the replay memory is gained along the training process, i.e. the decisions made in the past. Below is the output actions on the testing data and the result of the simple evaluation task:



(a) ResNet



(b) RNN

Figure 18: Output action on testing data

(a) ResNet (b) RNN

Figure 19: Price changes and asset value changes on the testing data

The orange curve is the percentage price change comparing to the initial price. And the blue curve is the percentage change of the asset value comparing to the initial asset.

We can see that both ResNet agent and RNN agent took frequently changing actions. Although the ResNet agent avoided some decreasing edges, it also misses many rising edges. And the RNN agent did a relatively better job than the ResNet agent since it clearly detect some rising edges as well as decreasing edges. However, it still performs worse than the Q-learning base line.

5.3 Scheme 3

I used both the ResNet and RNN to approximate the Q function $Q: \ (S_t, A_t) \to Q(S_t, A_t)$. And the replay memory is gained from the optimal policy.

In our case, a continuous series of states are given and we know the reward in between each state, so the optimal policy $\pi : S \to A$ of the training data can be obtained in advance by:

$$\pi(S_t) = \begin{cases} BUY, & p_{t+1} > p_t. \\ SELL, & p_{t+1} \leq p_t. \end{cases}$$

Where n is the number of actions. Below is a simple illustration:

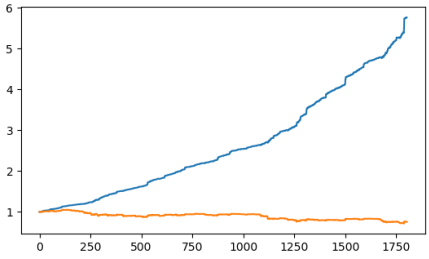


Figure 20: Optimal policy on testing data

The orange curve is the percentage price change comparing to the initial price. And the blue curve is the percentage change of the asset value comparing to the initial asset if the agent follows the optimal policy. This can be regarded as a upper bound of the best result that we can get on the testing set.

If we act in this way, the Bellman equation must still be valid since this yields the maximum accumulated reward by taking the maximum immediate reward at each timestep. And note that since we are already using the optimal policy as our replay memory, we do not need to do exploration anymore.

Instead of the formulation in section 2, at t-th training step, the forward propagation is formulated as follow:

$$y_t = R_t + \gamma Q(S_{t+1}, \pi^*(S_t), \theta)$$
$$\mathcal{L}(\theta)_t = (y_t - Q(S_t, A_t, \theta))^2$$

Where θ is the parameters of the neural network that we want to learn. y_t is the target value of $Q(S_t, A_t, \theta)$. $\mathcal{L}(\theta)_t$ is the loss function. Then we can do back propagation by gradient descent accordingly:

$$\frac{\partial \mathcal{L}(\theta)_t}{\partial \theta} = -2(R_t + \gamma Q(S_{t+1}, \pi^*(S_t, A_t), \theta) - Q(S_t, A_t, \theta)) \frac{\partial Q(S_t, A_t, \theta)}{\partial \theta}$$

$$\theta \leftarrow \theta - \alpha \frac{\partial \mathcal{L}(\theta)_t}{\partial \theta}$$

To do testing, we still choose the action yields the maximum Q value at the current timestep. Below is the output actions on the testing data and the result of the simple evaluation task:

(a) ResNet (b) RNN

Figure 21: Output action on testing data

(a) ResNet (b) RNN

Figure 22: Price changes and asset value changes on the testing data

The orange curve is the percentage price change comparing to the initial price. And the blue curve is the percentage change of the asset value comparing to the initial asset.

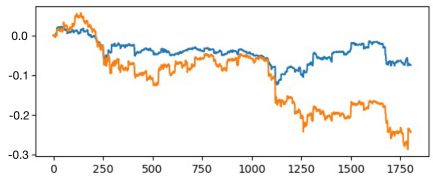
We can see that the ResNet agent mostly took action SELL(1) which is reasonable since the Bitcoin price is dropping generally. But it sold almost indifferently, which means it did not effectively detect the rising adges.

The RNN agent took frequently changing actions. Although the RNN agent failed to avoid in some decreasing edges at the first half, it did a relatively good job in detect the rising edge in the last half, since while the Bitcoin price is dropping, the asset value is actually increasing. And we can say that it perform no worse than the Q learning baseline.

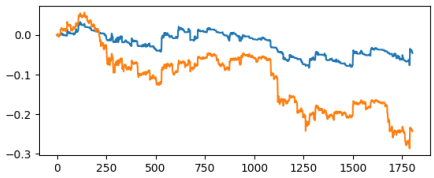
The code of this scheme is available on GitHub: https://github.com/WANGXinyiLinda/Deep-Q-Learning-Bitcoin-Trading-Agent

6 Conclusion

From the experiments above we can see that scheme 3 with RNN has the best performance. And it turns out that the simple Q-learning is a strong baseline and the best deep Q learning model I have tried does not perform much better than it.



(a) Scheme 3, RNN



(b) Q-learning baseline

Figure 23: Price changes and asset value changes on the testing data

In general, RNN performs better than ResNet, which I think is because that the input data is highly related between each timestep so RNN can better capture this feature.

A much better result will be yielded if we do memory reply with the optimal policy instead of past experience. I think that is because the optimal policy is a very strong additional information for Q learning.

7 Future Work

In this project, I assume the actions are deterministic, which means that the outcome of each action at a specific state is certain. But in reality, at any timestep, the price has a certain probability of both going up and going down. Then I think maybe we can learn this probability from the training data first, and then do reinforcement learning and then learn a stochastic policy.

I think one possible way to do it is to use a deep learning model to predict the possibility of the next state, which is widely used in the Language models. We can first discretize the price into some slots like what we did in section 2, and then predict the probability of the price at the next time step falling in each slot given a sequence of prices before it.

On top of what I have done right now, I want to try to use the policy gradients instead of Q learning, which allows me to have a continuous action space so I can actually take a position in the Bitcoin market.

And for the RNN part, I also want to try some other type of recurrent units like Long-Short-Term Memory (LSTM) units.

Acknowledgement

I would like to thank Chun Ho Chris Park, Matthew Thomas Sturm and Katherine Thai, my teammates in RIPS-HK 2018 research program who worked with me in the early stage of this project. I also want to thank Jonathan Yan, our sponsoring mentor and Queenie Lee, our academic mentor, who gave us a lot of help in our project.

I also wish to express my sincere gratitude to my SCIE3500 supervisor, professor Yao Yuan, who gave me a lot of insightful guidance and suggestions in the past three months.

References

[1] Richard S. Sutton, Andrew G. Barto. *Reinforcement Learning: An Introduction*. The MIT Press, 2017.

[2] Kao-Shing Hwang, Chia-Ju Lin, Chun-Ju Wu, Chia-Yue Lo. *Cooperation Between Multiple Agents Based on Partially Sharing Policy*, 2007

[3] Tambet Matiisen, Guest Post (Part I): Demystifying Deep Reinforcement Learning `https://ai.intel.com/demystifying-deep-reinforcement-learning/`

[4] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, et al. *Playing Atari with Deep Reinforcement Learning*, 2014

[5] Kaggle: Bitcoin Historical Data
`https://www.kaggle.com/mczielinski/bitcoin-historical-data`

[6] Bitcoin sub-reddit
`https://www.reddit.com/r/Bitcoin`

[7] Pushshift API
`https://github.com/pushshift/api`

[8] Kaiming He, Xiangyu Zhang, Shaoqing Ren, Jian Sun. *Deep Residual Learning for Image Recognition*. 2015.

[9] Kaiming He, Xiangyu Zhang, Shaoqing Ren, Jian Sun. *Identity Mappings in Deep Residual Networks*. 2016.

[10] raghakot, keras-resnet
`https://github.com/raghakot/keras-resnet`

[11] Simeon Kostadinov, *Understanding GRU networks*
`https://towardsdatascience.com/understanding-gru-networks-2ef37df6c9be`