Reinforcement Project Report

Group 19
518030910367 田亚博
518030910375 张世康
518030910360 郎伟临

2020 年 11 月 9 日

1 Easy21

The goal of this assignment is to apply reinforcement learning to solve a simple game, called Easy 21. This exercise is similar to the traditional Blackjack example. However, please note that the rules of the card game are different and nonstandard.

1.1 Q-learning

1.1.1 Introduction

Q-learning is a model-free reinforcement learning algorithm to learn quality of actions telling an agent what action to take under what circumstances. It does not require a model (hence the connotation "model-free") of the environment, and it can handle problems with stochastic transitions and rewards, without requiring adaptations.

For any finite Markov decision process (FMDP), Q-learning finds an optimal policy in the sense of maximizing the expected value of the total reward over any and all successive steps, starting from the current state. Q-learning can identify an optimal action-selection policy for any given FMDP, given infinite exploration time and a partly-random policy. "Q" names the function that the algorithm computes with the maximum expected rewards for an action taken in a given state.

1.1.2 Explanation

Q(s, a)is the expectation that the action a (a ∈ A) can obtain the revenue under the s state (s ∈ S) at a certain time. The environment will feedback the corresponding reward according to the action of the agent. Therefore, 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 obtain the maximum benefit according to the Q value.

Optimal value action function $Q^*(s,a) = max_\pi Q^*(s,a)$. And we can get the updating process of Q-table, where α is the learning rate and γ is the reward decay coefficient, which is updated by time difference method.

$$Q(s,a) \leftarrow Q(s,a) + \alpha[\gamma * max_{a'} Q(s',a') - Q(s,a)]$$

The above formula is the updated formula of Q-learning. According to the maximum Q(s',a') value in the next state s' multiplied by the decay *gamma* and the real return value, the Q(s, a) in the past Q-table is used as the Q estimation.

1.1.3 Result

We changed different hyperparameters to verify the impact of different hyperparameters on the winning rate. The results are as follows: (All results are the average of three experiments.)

- **Gamma** (All epsilon is 0.9, all learning rate is 0.01 and train 100,000 eposides)

gamma	0.1	0.3	0.5	0.7	0.9	1.0
winning rate	0.47059	0.46975	0.47664	0.47582	0.47643	0.47424

  We find that With the improvement of gamma, the winning rate will also increase slightly, but only about 0.5%, the overall impact is not big.

- **epsilon** (All gamma is 0.9, all learning rate is 0.01 and train 100,000 eposides)

epsilon	0.5	0.6	0.7	0.8	0.9	1.0
winning rate	0.47369	0.47235	0.47366	0.47501	0.47488	0.47448

  We find that With the improvement of gamma, the winning rate will also increase slightly, but only about 0.3%, the overall impact is not big.

- **learning rate** (All gamma is 0.9, and train 100,000 eposides and test 100,000 eposides each time)

learning rate	0.001	0.005	0.01	0.05	0.1
winning rate	0.46904	0.47235	0.47266	0.46392	0.45549

  We find that we have the best winning rate when learning rate is 0.01.

- **eposide** (All gamma is 0.9, and all learning rate is 0.01)

eposide	10	100	1000	5000	10000	50000	100000	1000000
winning rate	0.05791	0.28506	0.43211	0.45694	0.46255	0.47433	0.47384	0.47757

  We find that when eposide is less than 50000, winning rate can steadily improve, but after that, the winning rate will not increase much.

The learning curve against episode number is as following:

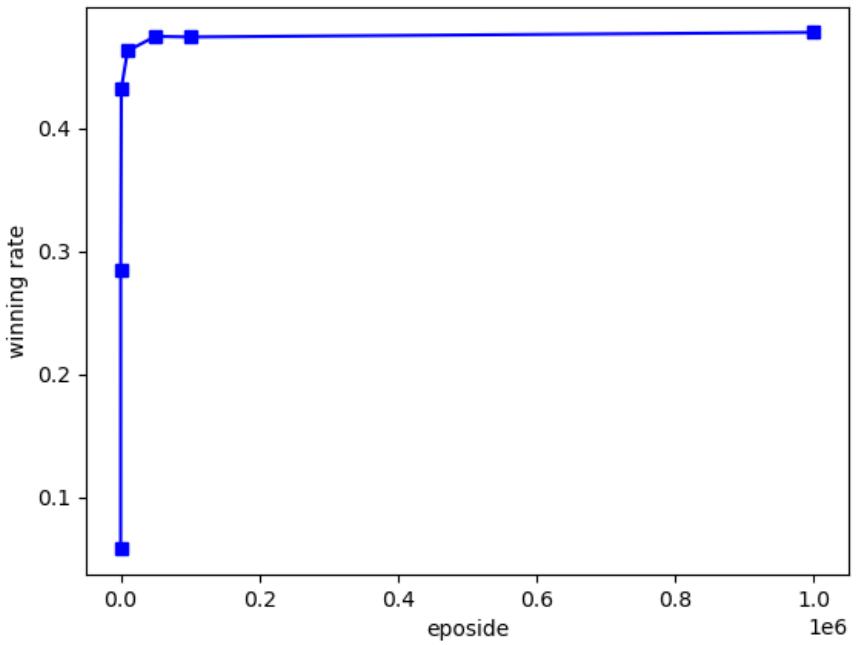

Figure 1: learning curve

Finally, the optimal state-value function is showing in the following figure:

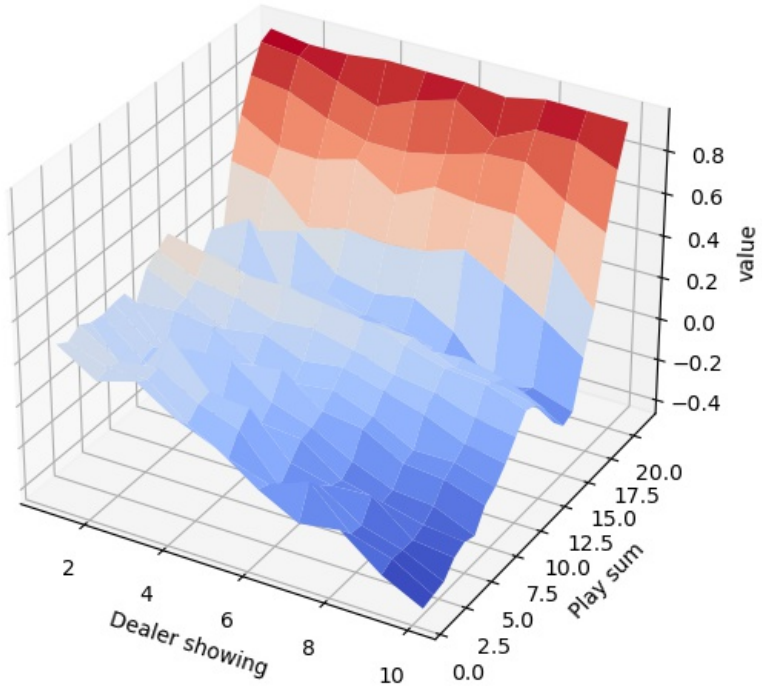

Figure 2: the optimal state-value function

1.2 Policy iteration

1.2.1 Introduction

Policy iteration is a broad-based algorithm in reinforcement learning. The main purpose is to obtain the optimal strategy. The strategy inheritance algorithm includes two simulations and interactive processes. One is strategy evaluation, which evaluates the strategy through a calculation function, so that the value function can be consistent with the current strategy; the other is strategy improvement, which improves the strategy based on the new value function , So that the strategy alternates with the current value function. In the process of strategy transformation, the two processes are learned alternately. And the core formula for updating the value is:

$$V(s) = r(s) + \gamma * \max(\sum_{s',r} p(s', r|s, \pi(s) * V(s')))$$

1.2.2 Explanation

For our code, I will introduce the following more important aspects:

- **Update the value of V-table:**
  For updating one value of v-table, there are two different situations:
  - For drawing cards, since we cannot determine the specific next state, we calculate the weighted average of $reward + \gamma \times value$ of all possible next states, where $\gamma = 1.0$. If it is a termination situation (only possible player out of bounds), reward is -1, otherwise reward is 0.
  - For not drawing cards, it is similar to not drawing cards, except that the person who draws the card is converted to the dealer. But the difference is that you can't switch to drawing cards after you don't draw the cards.Therefore, the reward is changed to the result that the player waits until the dealer draws to the end instead of the result of one step. For this, we use test.py to calculate that the player ends in this state and the probability of player winning, drawing and losing. And use the probability of winning minus the probability of losing as the reward.

  We use the above steps repeatedly until the v-table converges, and then update the value of the v-table. For convergence judgment, our judgment method is max error.

- **Strategy update** (use the above method to update the value at the same time):
  We created a 10*21 table to store the strategy (whether the player chooses to draw cards in this state,and initial strategy authority is all to draw cards). After we extract the strategy, we use the inverse strategy of this strategy to apply the value update formula. If the new value is greater than the original value, the strategy and value are updated, otherwise, the original strategy is maintained.

- **Create V-table**:
  According to our idea of updating the value, we created a $41 * 41$ table (V-table) to store the value of the player in different states.The number of rows in the table represents the sum of points of the dealer's cards, and the number of columns represents the sum of points of the players' cards.

- **Initialize the V-table**:
  For the end states(the result of the known outcome), if the player is out of bounds (the number

of columns is -9 - 0 and 22 - 31),it is initialized to -1, otherwise, if the dealer is out of bounds, it is initialized to 1. Both The out-of-bounds situation does not affect the update formula, so the assignment has no effect.If none is out of bounds, the value 1, 0 or -1 is assigned according to the comparison of the number of rows and the number of columns.
It is worth mentioning that for the situation where the number of dealer cards is 11 - 16, the player will not encounter these situations when making decisions, but these affect the player's decision in the current situation. So we use the test function to calculate after the player stops drawing the cards and the dealer has entered this situation, the probability of the player's winning, drawing, and losing, and use the probability of winning minus the probability of losing as the value of the situation. Other intermediate situations can be just initialized to 0.

1.2.3 Result

We changed different gamma values to see how it affects the winning rate. It is shown below: (All results are the average of three experiments.)

gamma	0.1	0.3	0.5	0.7	0.9	1.0
winning rate	0.46988	0.47225	0.47185	0.47549	0.47804	0.47522

We observe that with the increase of gamma, the winning rate has also increased slightly, but the increase is less than 1%, so the overall impact is not too big.

The code we implemented can get a winning rate of about 47.8%. (gamma is 0.9)

1.3 Comparison of Q-learning and Policy iteration

- From the results, the winning rate of policy iteration, which is 47.8%, is slightly higher than q-learning, which is 47.4%, but the difference is only about 0.4%. So it can be said that the winning rate is about the same.
- From an algorithmic point of view, Q-learning is model-free, and policy iteration is model-based.
- From a performance point of view, policy iteration converges slightly faster than Q-learning.

2 Quanser Robot platform

2.1 Introduction of Environment

2.1.1 CartPoleSwing

The model of CartPoleSwing is shown in Figure 3. The pendulum pivot is on the IP02 cart, and is measured using the Pendulum encoder. The centre of mass of the pendulum is at length, l_p , and the moment of inertia about the centre of mass is J_p . The pendulum angle, α, is zero when it is suspended perfectly vertically and increases positively when rotated counter-clockwise (CCW). The positive direction of linear displacement of the cart, x_c , is to the right when facing the cart. The position of the pendulum centre of gravity is denoted as the (x_p, y_p) coordinate.

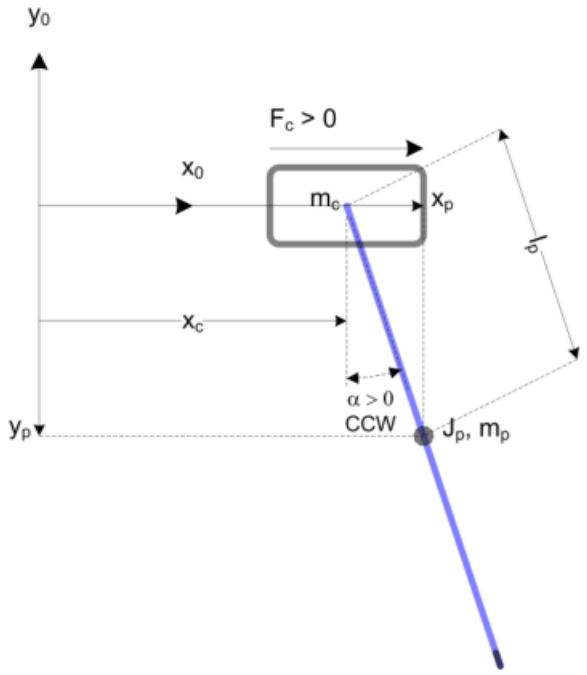

Figure 3: CartPoleSwing-model

action space: The action space of this environment is [-24, 24].

observation space: The observation space contains

- Cart Position x: [-0.407, 0.407],
- $sin\theta$: [-1, 1],
- $cos\theta$: [-1, 1],
- speed of cart x': $[-\infty, \infty]$,
- Rod angular velocity θ': $[-\infty, \infty]$.

episode reward: The episode reward is 4.642608791589737.

2.1.2 BallBalancer

Since the 2 DOF Ball Balancer uses two Rotary Servo Base Unit (SRV02) devices and the table is symmetrical, it is assumed that the dynamics of each axis is the same. The 2 DOF Ball Balancer is therefore modeled as two de-coupled "ball and beam" systems where we assume the angle of the x-axis servo only affects the ball movement in the x direction. Similarly for the y ball motion. The free body diagram of the Ball and Beam is illustrated in Figure 4.

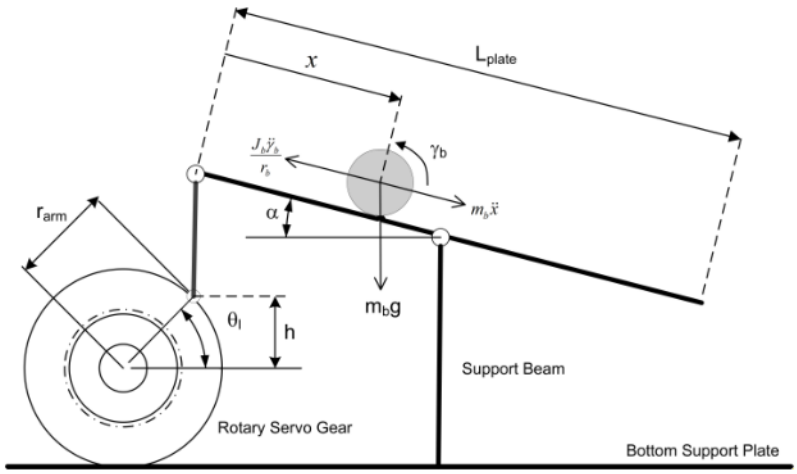

Figure 4: BallBalancer-model

action space: Applying a positive voltage causes the servo load gear to move in the positive, counter-clockwise (CCW) direction. This moves the beam upwards and causes the ball to roll in the positive direction (i.e., away from the servo towards the left). And We have two servos to control the rotation of the x-axis and y-axis. So the action space is $[-5, -5] \rightarrow [5, 5]$, which refers to voltage.
observation space:
state space lower bound: [-0.785 -0.785 -0.15 -0.15 -12.566 -12.566 -0.5 -0.5]
state space upper bound: [0.785 0.785 0.15 0.15 12.566 12.566 0.5 0.5]
These parameters represent

$$\theta_x,\ \theta_y,\ pos_x,\ pos_y,\ \theta'_x,\ \theta'_y,\ pos'_x,\ pos'_y$$

.
episode reward: The episode reward is 133.47330919806288.

2.1.3 Qube

The rotary pendulum model is shown in Figure 5. The rotary arm pivot is attached to the QUBE-Servo 2 system and is actuated. The arm has a length of L_r , a moment of inertia of J_r , and its angle θ increases positively when it rotates counter-clockwise (CCW). The servo (and thus the arm) should turn in the CCW direction when the control voltage is positive ($V_m > 0$).
The pendulum link is connected to the end of the rotary arm. It has a total length of L_p and it center of mass is at $L_p/2$. The moment of inertia about its center of mass is J_p . The inverted pendulum angle α is zero when it is hanging downward and increases positively when rotated CCW.

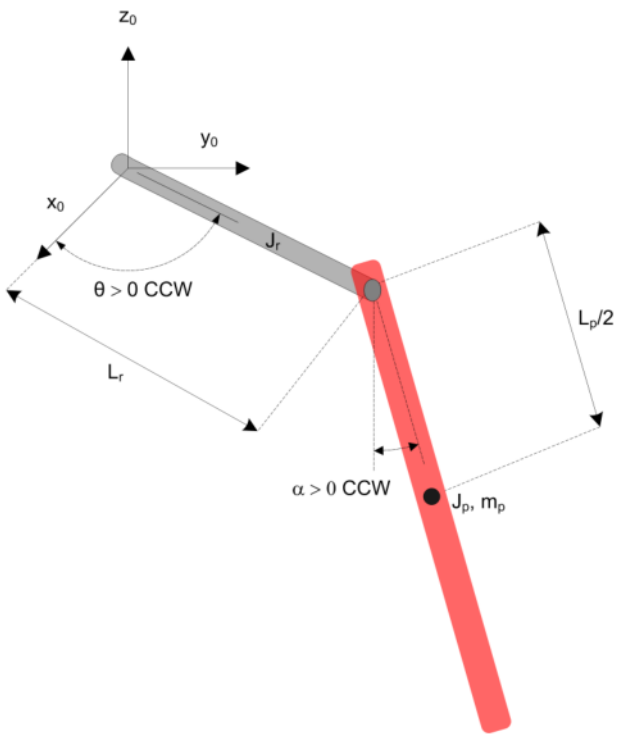

Figure 5: Qube-model

action space: The action space is $-5 \rightarrow 5$, which refers to the voltage input.
observation space:
state space lower bound: [-1. -1. -1. -1. -30. -40.]
state space upper bound: [1. 1. 1. 1. 30. 40.]
These parameters represent

$$cos\theta,\ sin\theta,\ cos\alpha,\ sin\alpha,\ \theta',\ \alpha'$$

.
episode reward: The episode reward is 0.0044975201431917385.

2.2 TRPO

2.2.1 Introduction

TRPO is a classical reinforcement learning algorithm, which is improved from the strategy gradient algorithm. When the step size is not suitable, the strategy corresponding to the updated parameters is a worse strategy. When the worse strategy is used for sampling and learning, the parameters updated again will be worse, so it is easy to lead to worse learning and finally collapse. Therefore, appropriate step size is very important for reinforcement learning. The core of TRPO is to solve this problem.

2.2.2 Explanation

TRPO decomposes the return function corresponding to the new strategy into the return function of the original strategy plus one other term. As long as the other items of the new strategy are greater than zero, we can get a scheme to improve the strategy all the time. Guided by this idea, we can get the following equation:

$$\eta(\tilde{\pi}) = \eta(\pi) + E_{\tau \in \tilde{\pi}} \left[\sum_{t=0}^{\infty} \gamma^t A_{\pi} \left(s_t, a_t \right) \right]$$

$$where\ A_{\pi}(s,a) = Q_{\pi}(s,a) - V_{\pi}(s) = E_{s' \sim P(s'|s,a)} \left[r(s) + \gamma V^{\pi} \left(s' \right) - V^{\pi}(s) \right]$$

$$\eta(\tilde{\pi}) = \eta(\pi) + \sum_{s} \rho_{\tilde{\pi}}(s) \sum_{a} \tilde{\pi}(a \mid s) A^{\pi}(s,a)$$

$$where\ \rho_{\pi}(s) = P\left(s_0 = s\right) + \gamma P\left(s_1 = s\right) + \gamma^2 P\left(s_2 = s\right) + \ldots$$

However, s is produced by the new distribution, which has a strong dependence on the new distribution. In fact, this formula is totally impossible to achieve in application, because we are trying to get a new strategy, so the other items here are completely unknown. Therefore, TRPO has adopted some techniques to solve this problem.

- Use the old strategy to replace the new strategy, because the difference between the two is not very big.
- Use importance sampling to process action distribution.
- The average KL divergence is used to replace the maximum KL divergence.
- Quadratic approximation for constrained problems and primary approximation for unconstrained problems.

Finally, the problem is reduced to this:

$$\begin{aligned} &\text{maximize}_{\theta}\ E_{s \pi_{\theta ld}, a \pi_{\theta_{old}}} \left[\frac{\pi_{\theta}(a \mid s)}{\pi_{\theta_{old}}(a \mid s)} A_{\theta_{old}}(s,a) \right] \\ &\text{subject to}\ E_{s} \pi_{\theta_{\text{old}}} \left[D_{KL} \left(\pi_{\theta_{\text{old}}}(\cdot \mid s) \| \pi_{\theta}(\cdot \mid s) \right) \right] \leq \delta \end{aligned}$$

2.2.3 Result

Limited by the experimental time, this experiment only uses one set of parameters to implement the TRPO methods on the three environments, and saves the visualization results.

Implement the TRPO methods on Qube, Ball Balancer, CartPoleSwing platforms and solve the problem of each environment.

The Qube training condition is as follows. The Ball Balancer and CartPoleSwing are the same.

```
a/e/r 0.015200038426921749 0.014437326487637328 1.0528291675011665
fval after -0.015200038426921686
Episode 3999    Last reward: 1.4619962860952809 Average reward 1.91
('lagrange multiplier:', tensor(0.3163), 'grad_norm:', tensor(0.5442))
fval before -1.3775637950133384e-17
a/e/r 0.006860276480039112 0.006326106580330403 1.0844389661991451
fval after -0.006860276480039126
Episode 4000    Last reward: 1.1838121100475987 Average reward 2.38

Process finished with exit code 0
```

Figure 6: TRPO-Qube

The experimental results obtained are saved in the same path of the project.

```
torch.save(policy_net, "policynet.pth")
torch.save(value_net, "valuenet.pth")
```

Figure 7: TRPO-Qube-saved

Run the Qube test program, and record the visualization result. The Ball Balancer and CartPoleSwing are the same.

```
Episode 12  Last reward: 0.292857264801392  Average reward 0.24
Episode 13  Last reward: 0.10763618786054008    Average reward 0.25
Episode 14  Last reward: 0.1567040049680792 Average reward 0.23
Episode 15  Last reward: 0.3634284966708222 Average reward 0.23
Episode 16  Last reward: 0.30570127725622254    Average reward 0.25
Episode 17  Last reward: 0.14703045250907407    Average reward 0.23
Episode 18  Last reward: 0.18457594983095127    Average reward 0.23
Episode 19  Last reward: 0.1230316056601086 Average reward 0.23
**********
Consume 0.9434778690338135 s in this iteration
```

Figure 8: TRPO-Qube-test

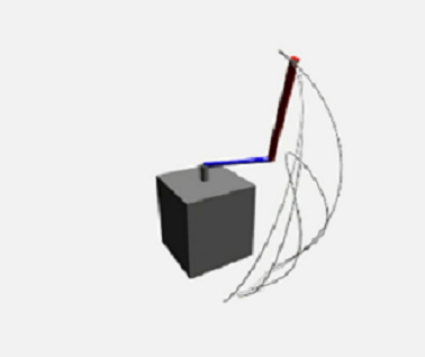

Figure 9: TRPO-Qube

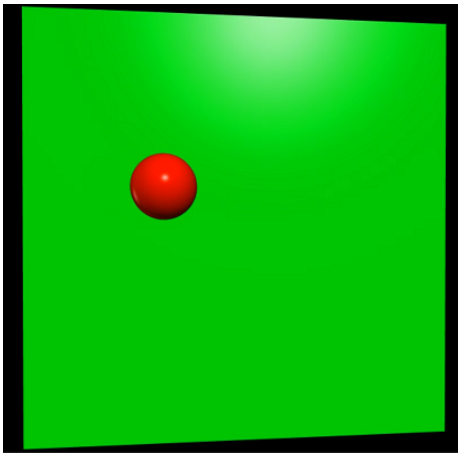

Figure 10: TRPO-Qube-test

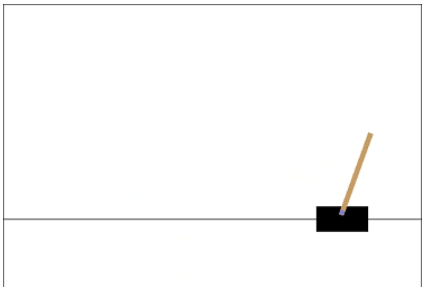

Figure 11: TRPO-Qube-test

The visualization test result are packed in the zip file named "TRPO-CartPoleSwing.mp4","TRPO-Balancer.mp4" and "TRPO-Qube.mp4"

Open the video file and we can observe the test result intuitively.

2.3 MPC

2.3.1 Introduction

The mechanism of MPC is to solve a finite time open-loop optimization problem online according to the current measurement information at each time of adoption, and the first element of the control sequence is applied to the controlled object. At the next sampling time, the above process is repeated: the new measured value is used as the initial condition to predict the future dynamics of the system, and the optimization problem is refreshed and solved again.

That is, the MPC algorithm includes three steps:

1) Forecast the future dynamics of the system;

2) (Numerical) Solve open-loop optimization problems;

3) Apply the first element (or the first part) of the optimized solution to the system

These three steps are repeated at each sampling time, and no matter what model is used, the measured value obtained at each sampling time is used as the initial condition for predicting the future dynamics of the system at the current time.

2.3.2 Explanation

For our problem, this algorithm can be implemented through the following steps.

- Obtain a large amount of data $D = ((s_t, a_t), s_{t+1})$ through a certain strategy $\pi(a_t \mid s_t)$ such as a random strategy.
- Learn a model $f(s_t, a_t)$ to minimize the error $\sum \|f(s_t, a_t) - s_{t+1}\|^2$.
- Use model $f(s_t, a_t)$ to predict.
- Perform the first step of predict to obtain a new piece of data.
- Repeat steps 3-5 several times, add the new data obtained to D and go back to step 2.

The essence of model predictive control is to keep re-planning so that errors will not accumulate too much. The more times we re-plan, the accuracy requirements of each of our plans can be reduced. Therefore, we can use some simple methods to plan, thereby reducing our computational complexity.

Then we analyze the code in detail.

- The DatasetFactory class in the code is used to generate experimental data, and the collect_random_dataset() function is just like the first step to use random strategies to generate a large amount of initial data.
- And then train the obtained data to get an initial model, just like the step 2.
- Then use the collect_mpc_dataset() function to continuously execute step 3 and step 4 to collect mpc data, and then use the obtained data to continue training the model.

The act() function in controller.py is used in collect_mpc_dataset() function. We use two optimization methods. In addition to the initial Artificial Bee Colony (ABC) optimization algorithm, we also use the random shooting method for testing , and got two sets of experimental results. In addition, we also tried different hyperparameters, including learning rate: 0.01, 0.001, 0.0001; and batch size: 256, 512, 1024. We also got several sets of experimental results, and then we will analyze the results.

2.3.3 Result

1. **CartPoleSwing**

   - **Artificial Bee Colony (ABC) algorithm**
     learning rate: First of all, we change learning rate to observe the effect of different learning rates on the results, the obtained curve is as follows: (All batch size is 512)

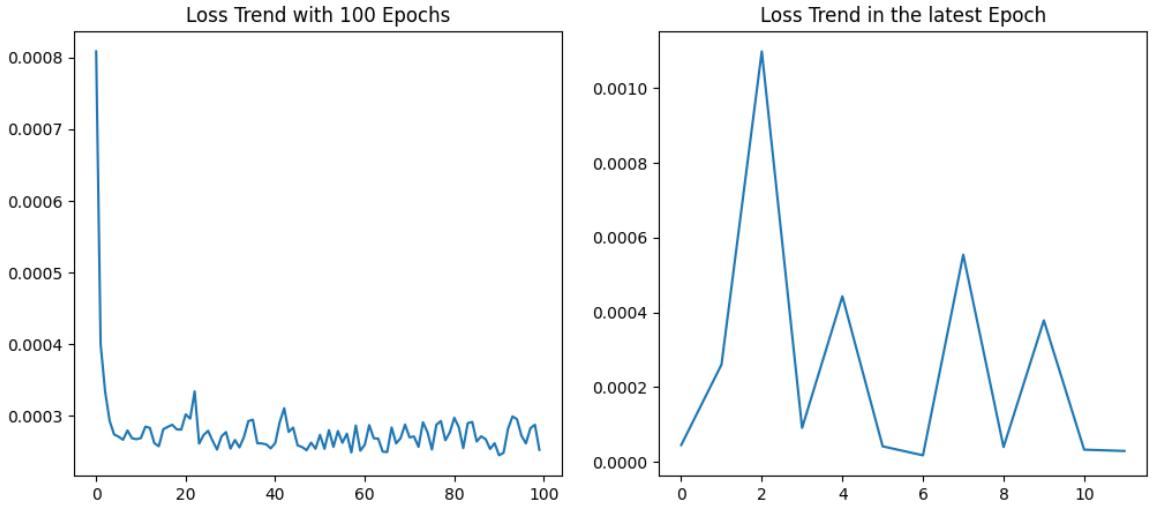

Figure 12: learning rate is 0.0001

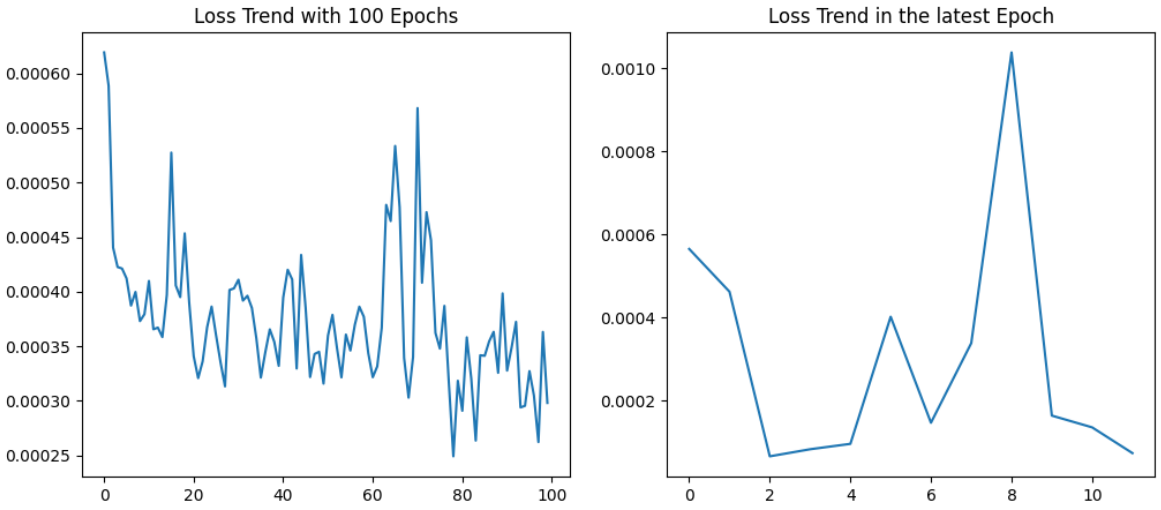

Figure 13: learning rate is 0.001

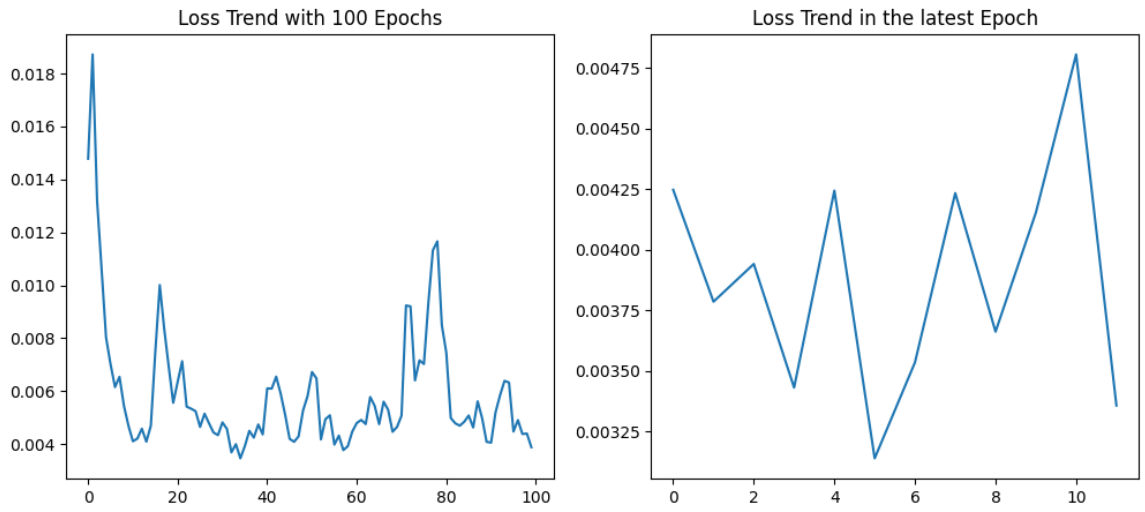

Figure 14: learning rate is 0.01

It can be seen from the above image comparison that as the learning rate increases, the loss shows an increasing trend. When the learning rate is 0.0001, the final loss stabilizes at about 0.0003 with very small fluctuations, but when the learning rate reaches 0.01, the final loss fluctuates around 0.0006 and fluctuates greatly.
So appropriately reducing the learning rate can reduce the loss without basically affecting the convergence rate.

batch size: we change different batch size to observe the effect of different batch size on the results, the obtained curve is as follows: (All learning rate is 0.001)

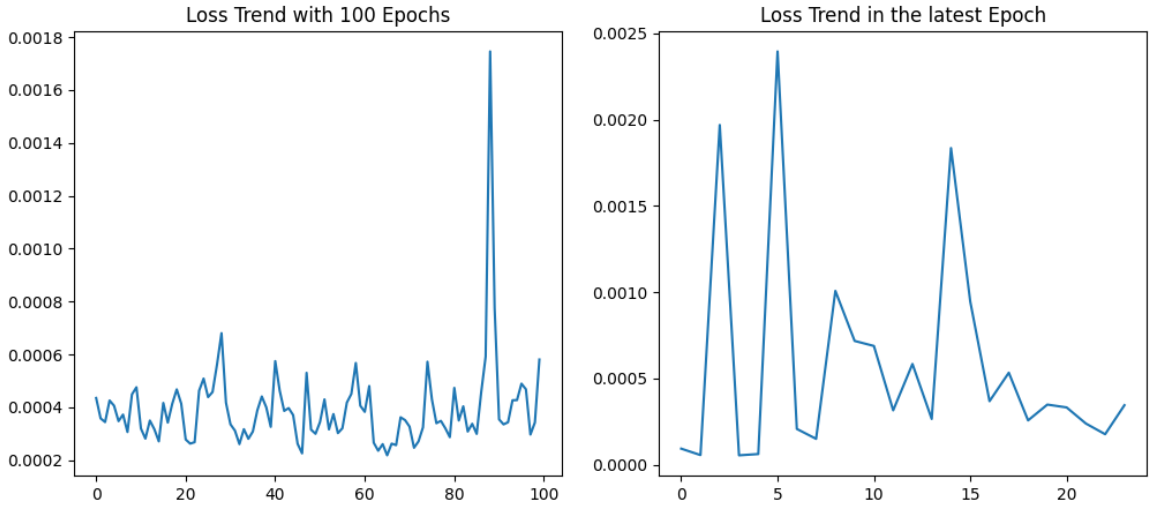

Figure 15: batch size is 256

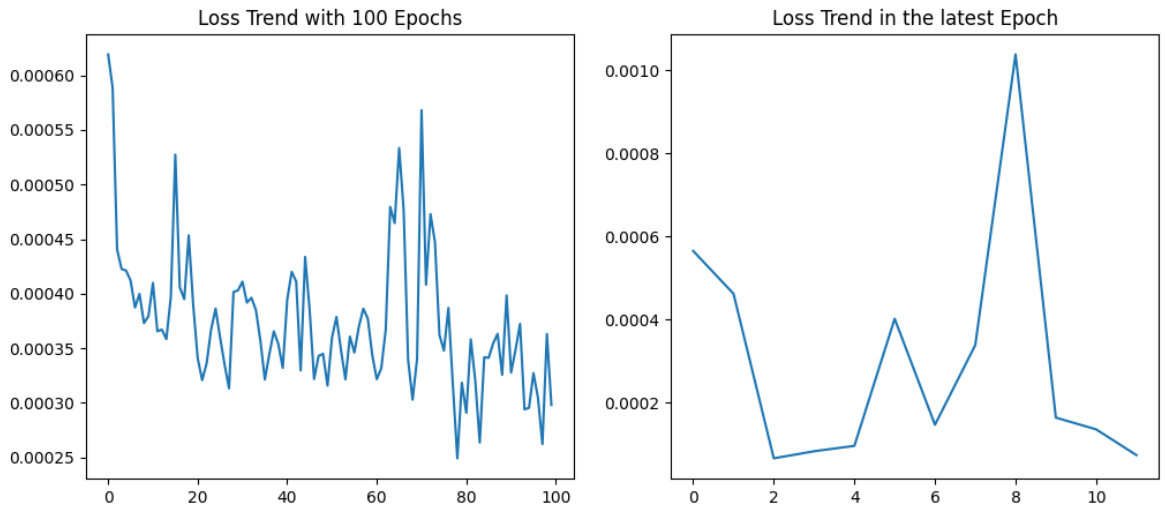

Figure 16: batch size is 512

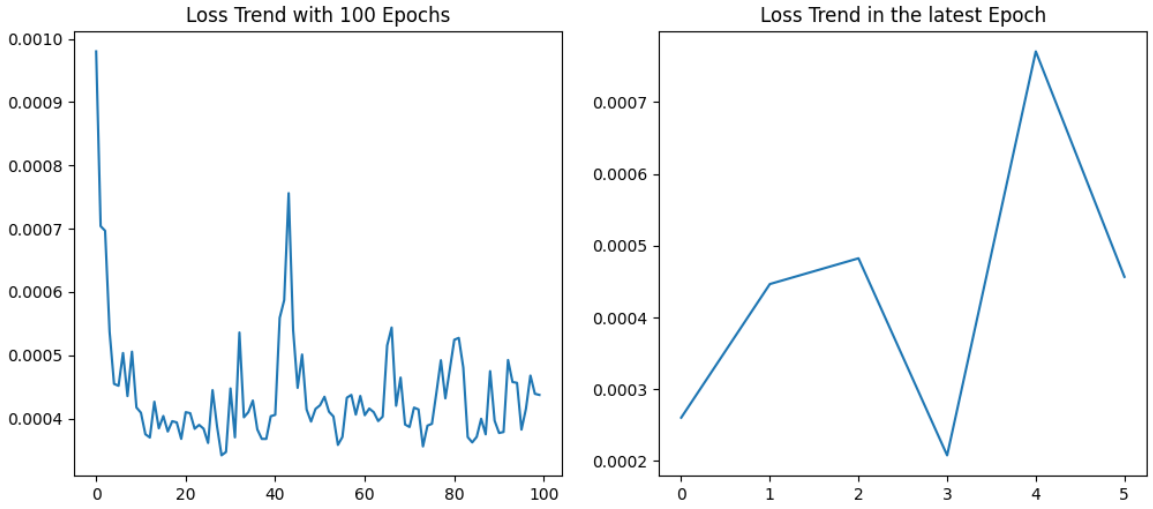

Figure 17: batch size is 1024

It can be seen from the comparison of the above images that the batch size has little effect on the loss, and the loss of the three different batch sizes is maintained at about 0.0004.

- **random shooting**
  learning rate: First of all, we change learning rate to observe the effect of different learning rates on the results, the obtained curve is as follows: (All batch size is 512)

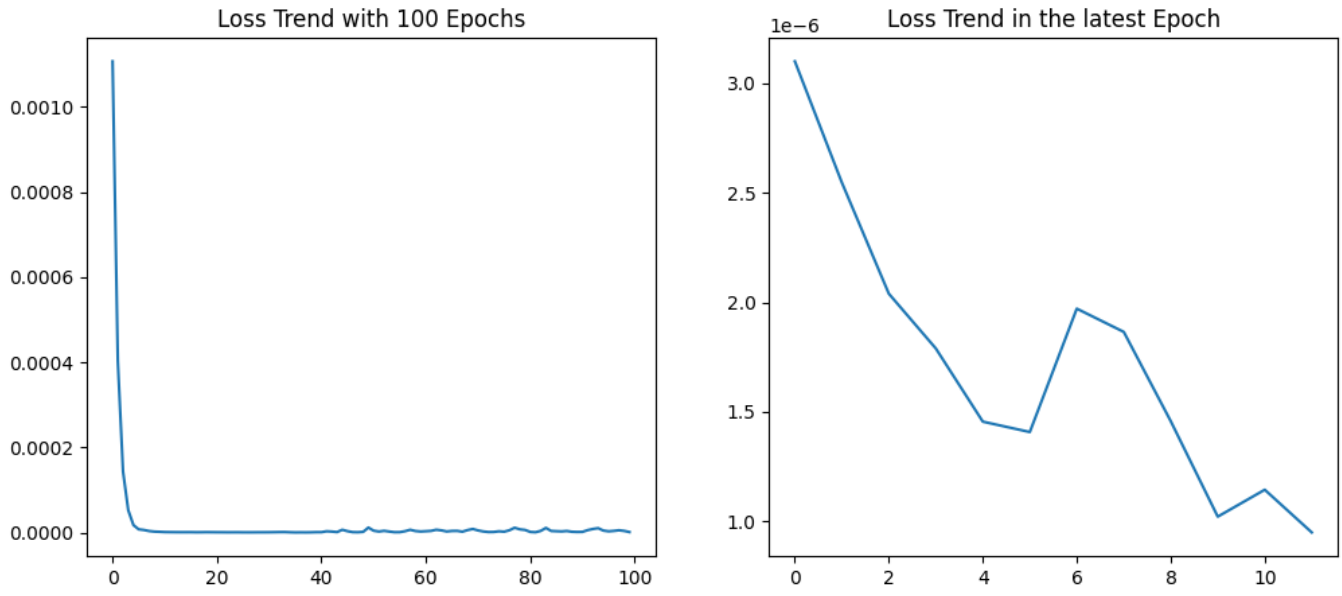

Figure 18: learning rate is 0.0001

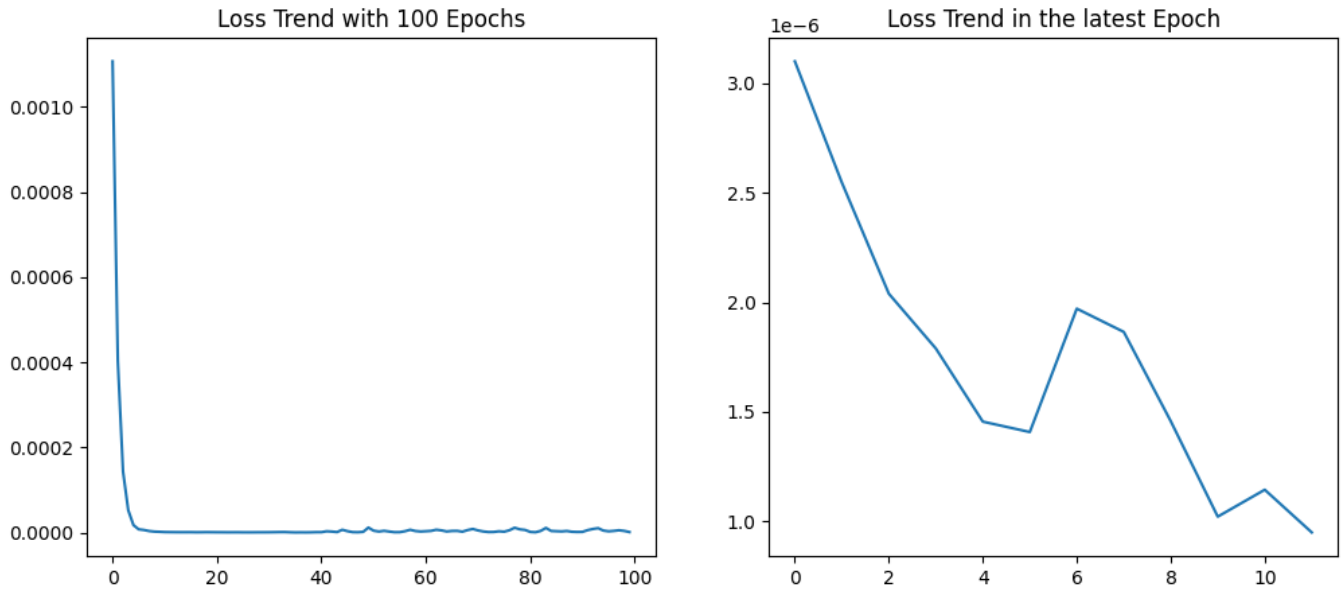

Figure 19: learning rate is 0.001

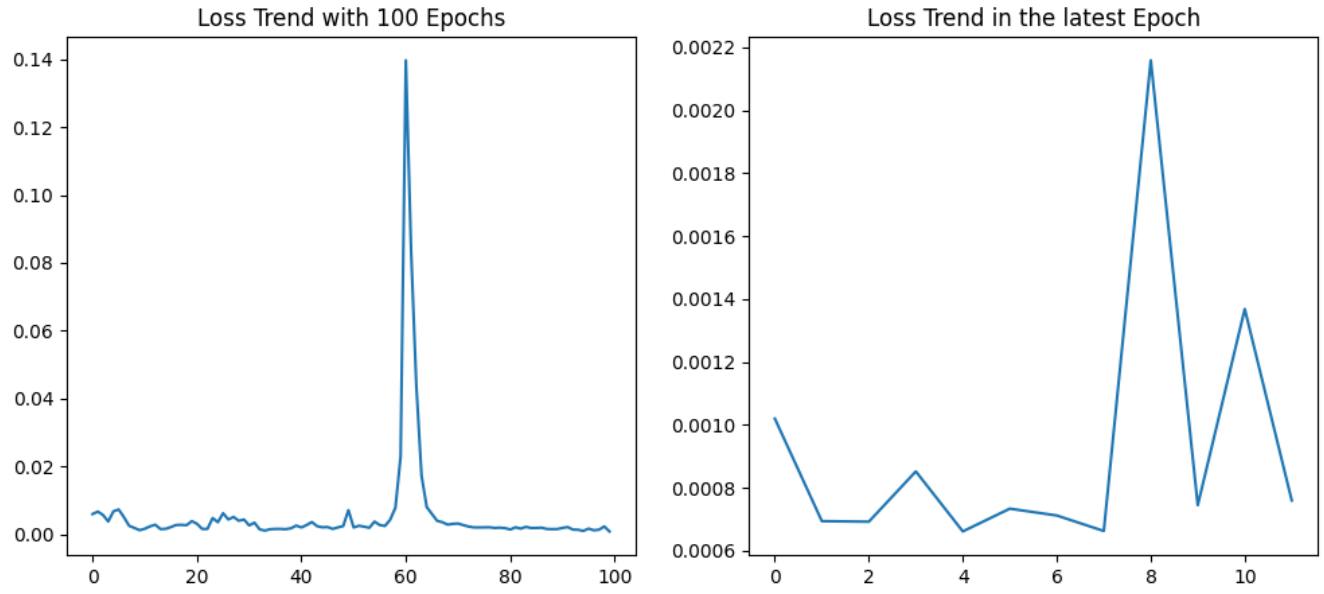

Figure 20: learning rate is 0.005

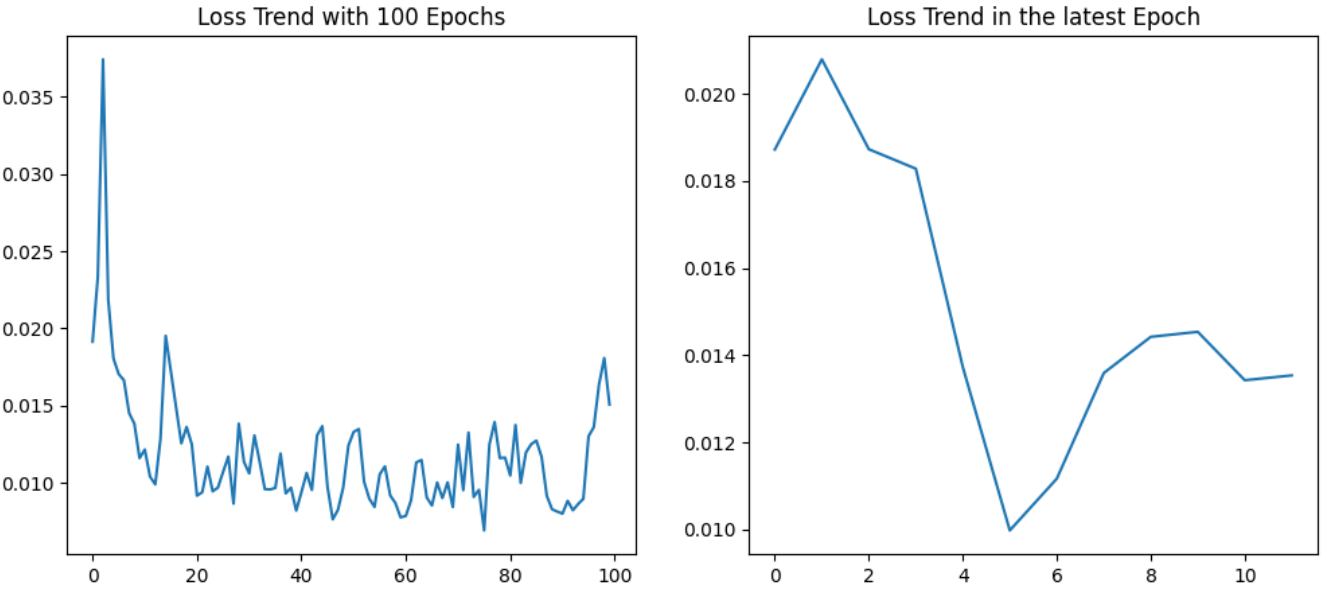

Figure 21: learning rate is 0.01

As can be seen from the comparison of the above images, the learning rate of 0.0001 is not much different from 0.001, and the loss value and the degree of fluctuation are not much different. As for the reason for the significant increase in loss at about the 62nd epoch when the learning rate is 0.005, it is temporarily unknown, I tend to think it is accidental. Similarly, as the learning rate increases, the overall loss also shows an increasing trend, and the fluctuations become more intense.
So when the learning rate is around 0.001, there are better results.

batch size: we change different batch size to observe the effect of different batch size on the results, the obtained curve is as follows: (All learning rate is 0.001)

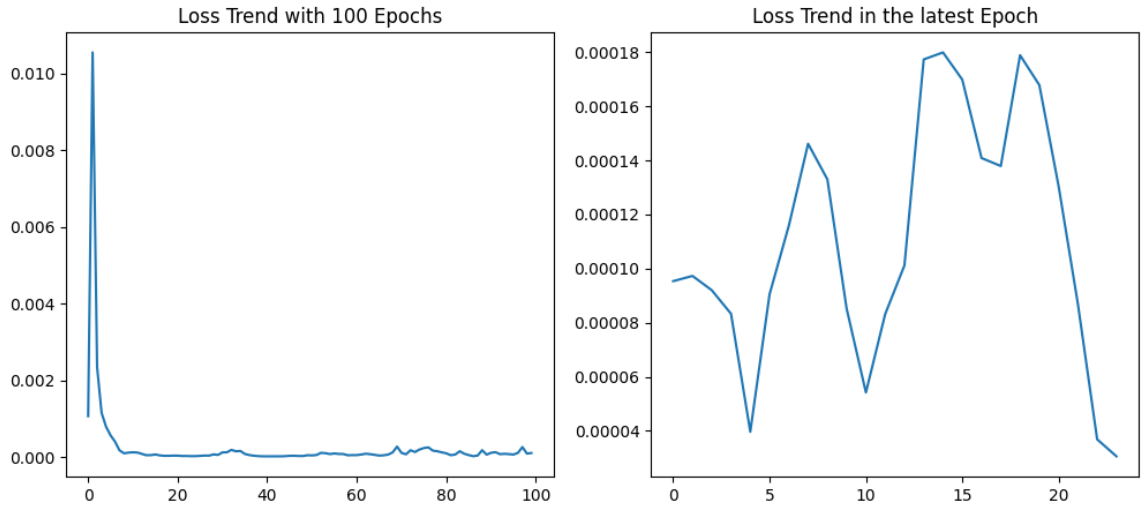

Figure 22: batch size is 256

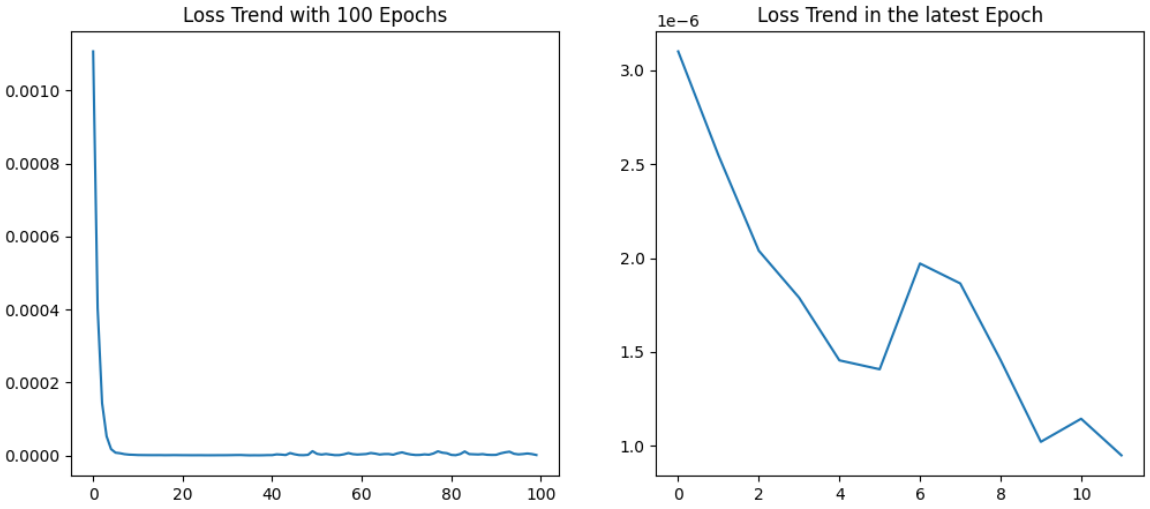

Figure 23: batch size is 512

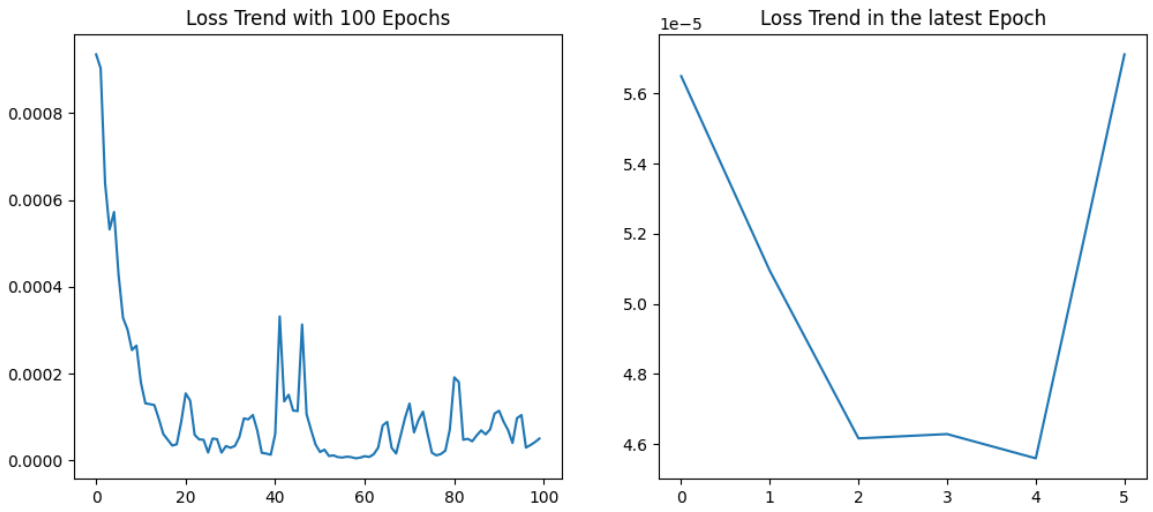

Figure 24: batch size is 1024

For random shooting methods, the batch size has a certain impact on the results. When the batch size is 256, the overall loss is larger than that of 512. Due to the influence of the coordinates, there may also be more severe oscillations, but it is not easy to find. When the batch size is 1024, the loss increases significantly and fluctuates sharply compared with 512.
So the best batch size for random shooting is 512.

Then we compare the reward of the ABC algorithm and the random shooting algorithm, as shown below: (learning rate is 0.001, batch size is 512)

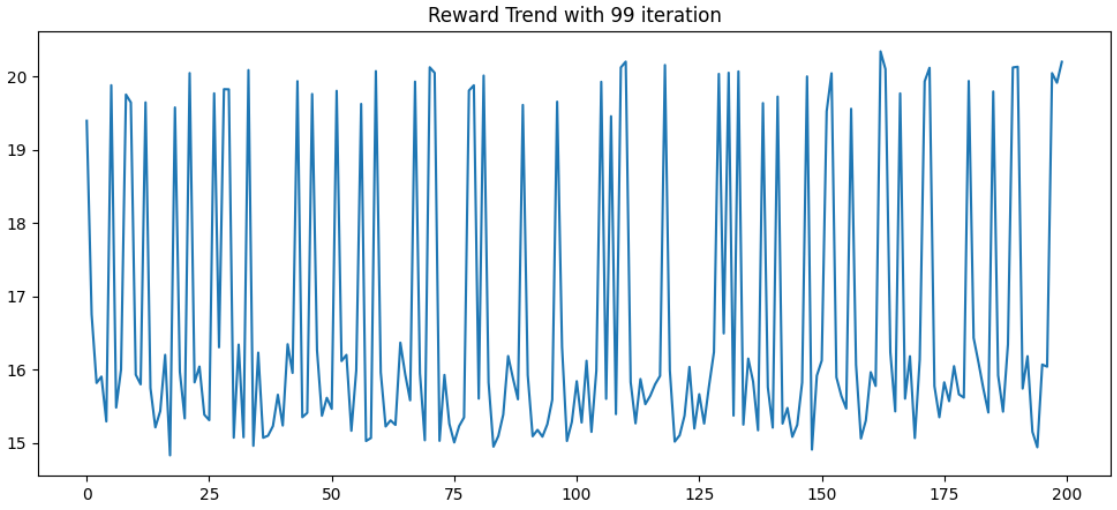

Figure 25: ABC reward

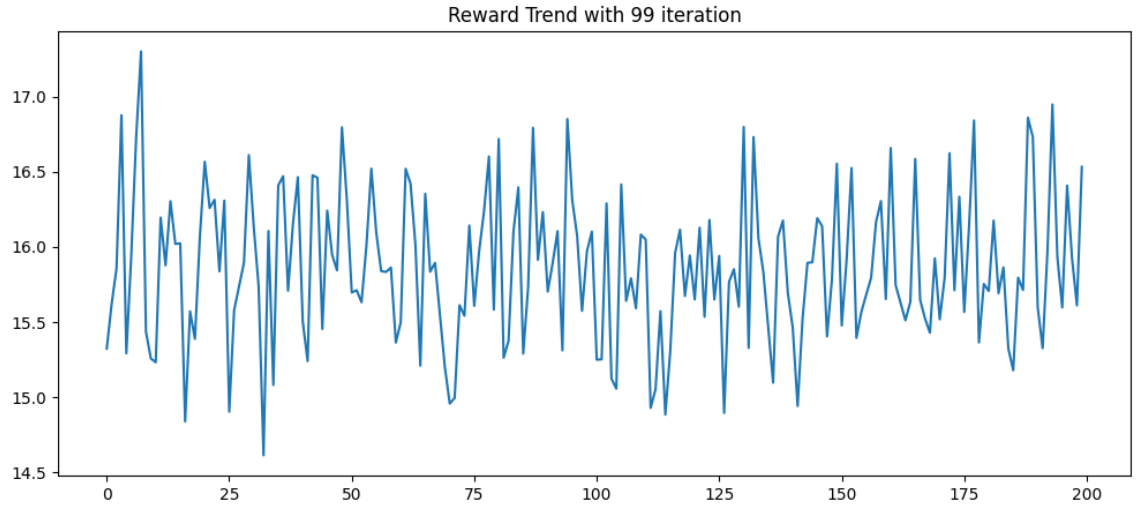

Figure 26: random shooting reward

From the comparison of the above figure, it can be found that the average reward of the ABC algorithm is about 18, while the average of the random shooting algorithm is less than 16. The loss of the two is almost the same, so the ABC algorithm is slightly better than the random shooting algorithm for this problem.

2. **Qube** Because Qube is extremely slow to train, it takes 2-3 days to train at a time, so there are not many valid data obtained, and it is impossible to form an effective comparison, so I only list it here.

   - ABC algorithm and random shooting algorithm comparison. (learning rate 0.001, batch size 512, the other parameters is the same)

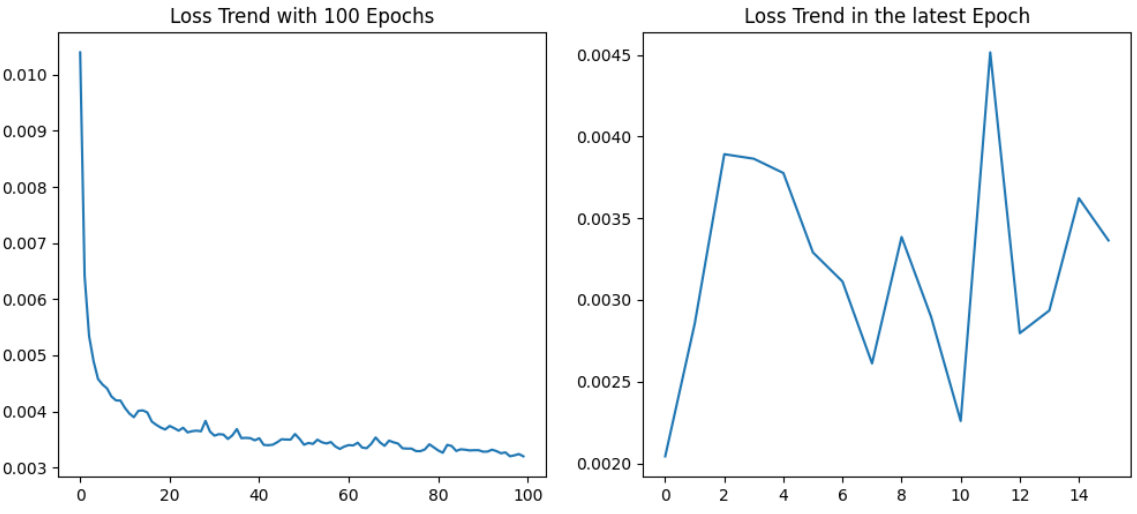

Figure 27: ABC loss

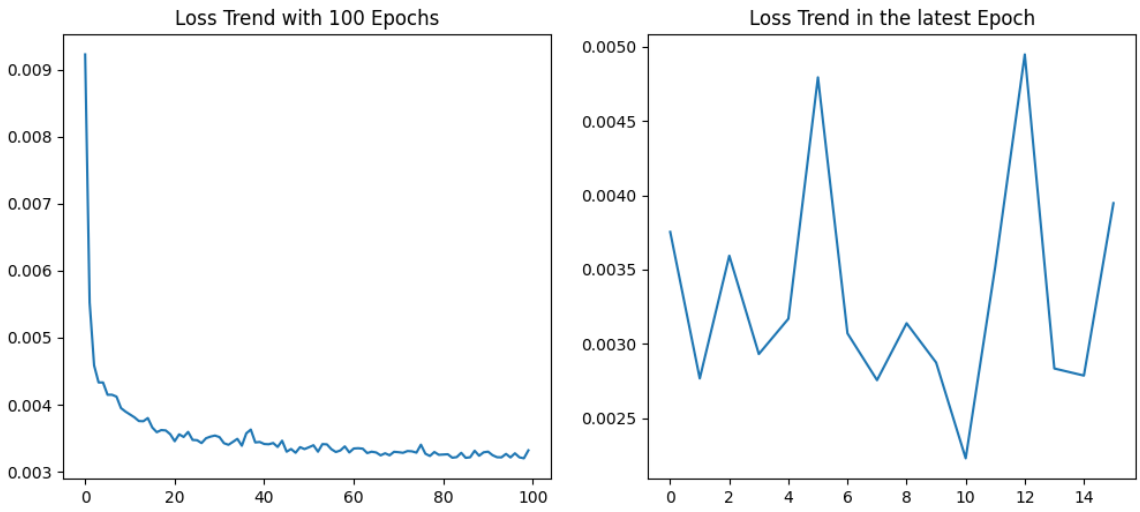

Figure 28: random shooting loss

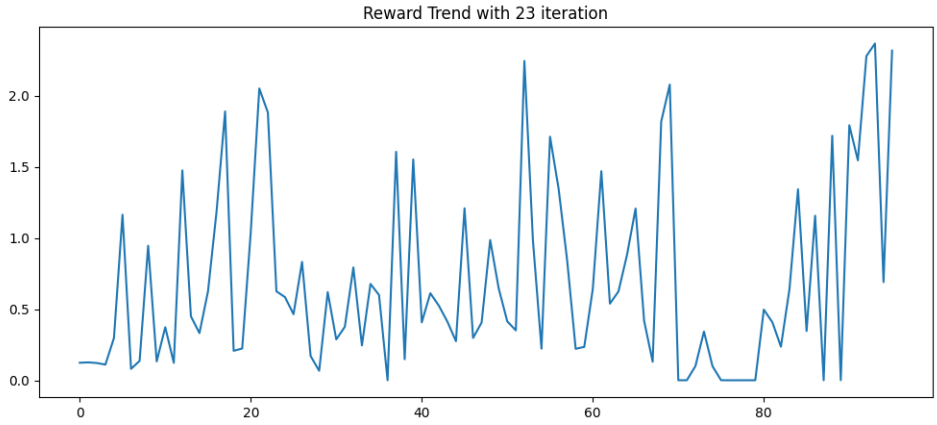

Figure 29: ABC reward

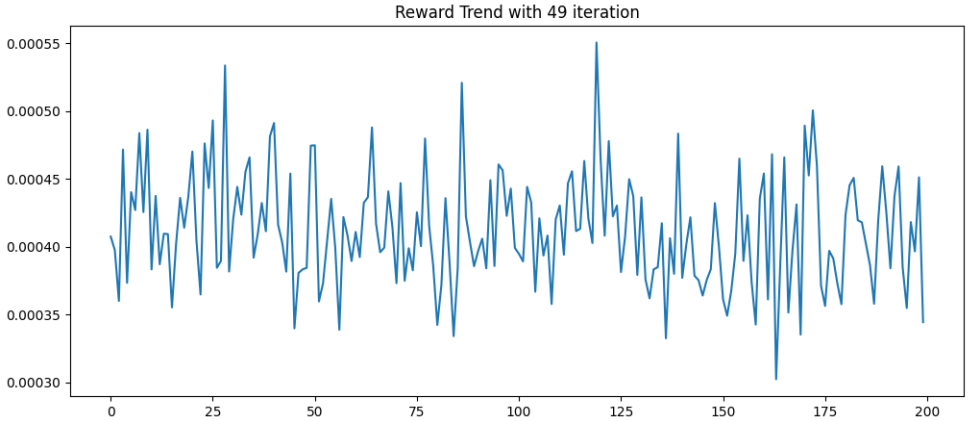

Figure 30: random shooting reward

Due to the time relationship, the ABC algorithm has only been trained for 24 rounds,

but it does not affect the overall comparison.
It can be seen from the comparison of the above images that the final convergence loss of the ABC algorithm and the random shooting algorithm is relatively close, but the reward of the ABC algorithm is much greater than the reward of the random shooting algorithm. So for this problem, the ABC algorithm should perform better.

- The origin parameters, which means ABC algorithm with learning rate is 0.001, batch size is 512, horizon is 5, numb_bees is 4, max_itrs is 10 and gamma is 0.999, we obtained the following results:

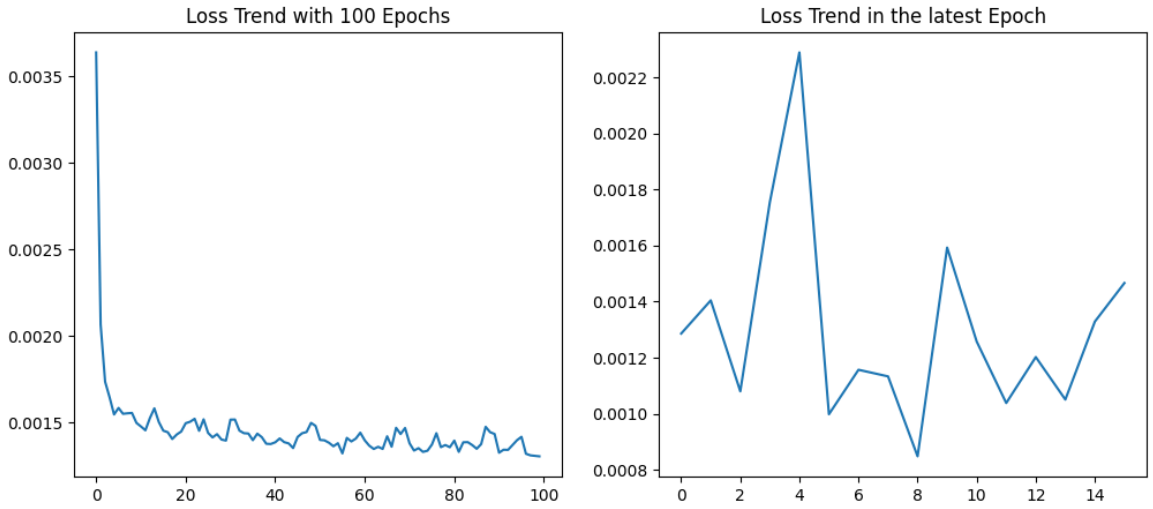

Figure 31: qube loss with origin parameters

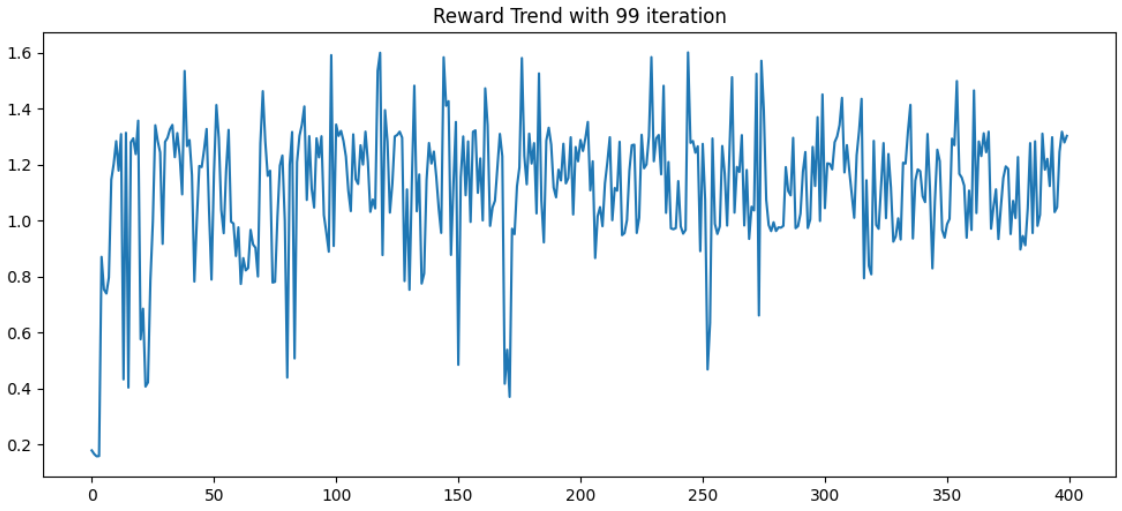

Figure 32: qube reward with origin parameters

The result we got looks acceptable.

2.4 Comparison of TRPO and MPC

- From the results, TRPO has a better reduction results than MPC.
- From an algorithmic point of view, TRPO is model-free, and MPC is model-based.
- From a performance point of view, TRPO has better training speed and good result.

3 Conclusion

Other results that we did not show can be found in the corresponding videos. Since the MPC method is extremely slow to train, we have accelerated the video of the MPC method to a certain extent.

Due to time constraints, some of our experiments were not completely completed.

This experiment is difficult and troublesome as a whole. We encountered various difficulties in the process of the experiment, but we tried to solve most of them. Due to the lack of basic knowledge, at the beginning of the experiment, there was confusion about where to start to solve the problem, and more thinking was needed to complete the experiment. In fact, in the process of completing the experiment, the design and simulation were successfully completed after learning and modifying the previous results.