EI339 Teamwork1

Yifan Lu, Siru Ouyang, Changqi Zhao

November 9, 2020

Contents

1 Introduction

This is the report for the Reinforcement Learning project of EI339 at SJTU. This project is mainly divided into two parts, Q-learning methods for Easy 21 games and deep reinforcement learning for Quanser Robot. In the following sections of this report, we will narrate the basic knowledge of these methods, how we implement these methods in python, what the results look like and how the parameters will affect the results. We also proposed some improvement for the required methods to make the result better.

2 Easy 21 games

2.1 Environment Implementation

The rule of easy21 is:

- cards' values are included between 1 to 10 (no heads in the game)
- player and dealer both start with a random card
- there are 2 possible actions, player/dealer can either hit (pick a new card) or stick (stop picking cards)
- black card's(probability $\frac{2}{3}$) values are added whereas red card's(probability $\frac{1}{3}$) values are subtracted to the score
- when the dealer plays, he follows a fixed strategy, he sticks if his score is between 17 and 21, otherwise he hits
- if cards' sum is below 1 or over 21 the player/dealer looses
- for each terminal state in Easy21, winning state is (player score $>$ dealer score), loosing state is (player score $<$ dealer score)

To implement the environment of easy21, we design a class to simulate the game procedure.

```
class Easy21():
  def draw_card(self):
    ...
    return (value,color)

  def draw_one_card(self,score):
    ...
    return score

  def dealer_turn(self,score):
    ...
    return score

  def go_bust(self,score):
    ...
    return true/false

  def step(self,state,action):
        ...
    return next_state,reward,terminal
```

The basic framework of easy21 is above. The function `draw_card` return the card number and the card color. The function `draw_one_card` receives player/dealer's score and return the score after he draw one card. The function `deal_turn` receives dealer's first card value, and follows the game rule, returning the final score. `go_bust` is used to judge whether the input score is out of valid range. The last function `step` simulate the game step after the player chooses to stick or hit.

```
  def step(self,state,action):
    # ---Input---
    # state: tuple, (dealer's first card, player's current sum)
```

```
    # action: int, 1 -> "stick", 0 -> "hit"
    #
    # ---Return---
    # next_state: tuple,
    # reward: int,
    # terminal: bool,

    dealer_score, player_score = state
    player_action = action

    if action == 1: # stick
      dealer_score = self.dealer_turn(dealer_score)
      terminal = True
      next_state = (dealer_score,player_score) # dealer's score may out of range
      if(self.go_bust(dealer_score)): # dealer go bust
        reward = 1
      elif(dealer_score==player_score): # two sums are the same.
        reward = 0
      else:
        reward = int((player_score-dealer_score)/np.abs(player_score-dealer_score)) # if
                                                player's sum is larger, then reward = 1

    else: # hit
      player_score = self.draw_one_card(player_score)
      next_state = (dealer_score,player_score)
      if(self.go_bust(player_score)):
        reward = -1
        terminal = True
      else:
        reward = 0
        terminal = False

    return next_state,reward,terminal
```

2.2 Temporal Difference (TD) in Q-learning

2.2.1 Introduction to Q-Learning

One of the early breakthroughs in reinforcement learning was the development of an off-policy TD control algorithm known as Q-learning [1], defined by

$$Q\left(S_t, a_t\right) \leftarrow Q\left(S_t, a_t\right) + \alpha\left[R_{t+1} + \gamma \max_a Q\left(S_{t+1}, a\right) - Q\left(S_t, a_t\right)\right]$$

The algorithm is :Before learning begins, Q is initialized to a possibly arbitrary fixed value (chosen by the programmer). Then, at each time t the agent selects an action a_t, observes a reward r_t, enters a new state s_{t+1} (that may depend on both the previous state s_t and the selected action), and Q is updated. The core of the algorithm is a Bellman equation as a simple value iteration update, using the weighted average of the old value and the new information.

Algorithm 1: Q-learning TD

Algorithm parameters: step size $\alpha \in (0, 1]$, small $\varepsilon > 0$;
Initialize $Q(s, a)$, for all $s \in \mathcal{S}^+, a \in \mathcal{A}(s)$, arbitrarily except that $Q(\text{terminal}, \cdot) = 0$;
Loop for each episode:;
  Initialize S ;
  Loop for each step of episode:;
    Choose a_t from S using policy derived from Q (e.g., ϵ -greedy) ;
    Take action a_t, observe R, S';
    $Q(S, a_t) \leftarrow Q(S, a_t) + \alpha\left[R + \gamma \max_a Q\left(S', a\right) - Q(S, a_t)\right]$;
    $S \leftarrow S'$;
  until S is terminal

we also introduce the ϵ-greedy method, which is applied to generated new action a_t.

$$a_t = \begin{cases} \arg\max_{a \in A} Q_t(s_t, a); & \text{exploitation with probability } 1-\epsilon \\ \text{rand}(A); & \text{exploration with probability } \epsilon \end{cases}$$

2.2.2 Implementation

The data structure we use to store Q table is python dictionary. The key is the tuple of (state,action). Here is how we initialize the Q table.

```
def init_Q_table(self):
  dealer = np.arange(1,11)
  player = np.arange(1,22)
  states = [(d,p) for d in dealer for p in player]
  Q_table = {}
  for state in states:
    Q_table[state] = np.zeros(2)
  return Q_table
```

For Q-learning, we simply follow the given formula. In one episode, we choose action by ϵ-greedy approach. And update the Q table by every time before game terminal.

```
while not terminal:
  tmp = np.random.rand()  # apply ε-greedy approach
  if tmp < self.epsilon:
    action = np.random.randint(2)
  else:
    action = np.argmax(self.Q[state])

  state_new,reward,terminal = game.step(state,action)

  self.Q[state][action] = (1-self.alpha)*self.Q[state][action] + \
              self.alpha*(reward+self.gamma*max(self.Q.get(state_new,(0,))))
  print(f"state: {state} \t action: {action} \t reward: {reward} \t value: {self.Q[state][
                                          action]}")
  state = state_new
```

2.2.3 Result Display

By setting episode number to 100000, $\alpha = 0.01$ and $\epsilon = 0.008$, we come with the following result shown in Fig.1.

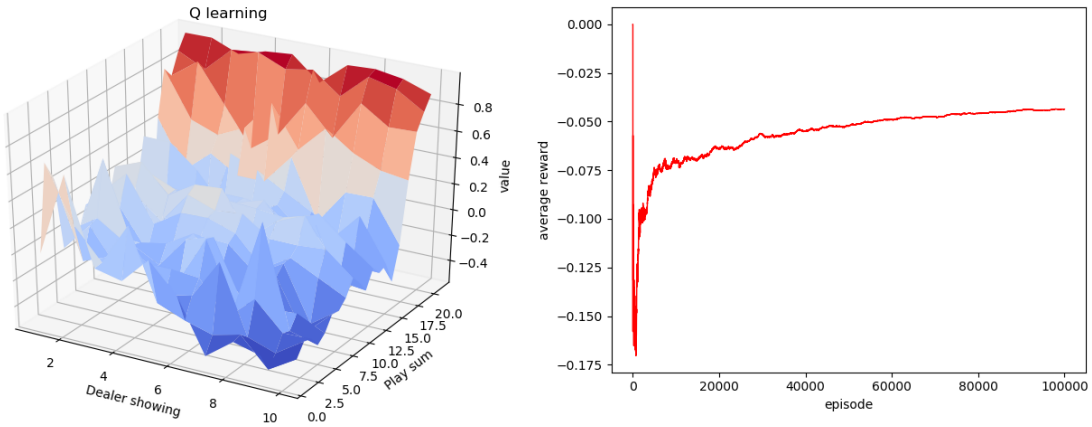


Figure 1: The best result of TD learning.

In order to explore the parameters, namely, γ and ϵ, we use ten candidate parameters each, run in

total 100 combinations. We plot the winning rate under each circumstance in Fig.2. From the figure, it is obvious to see that the highest wining rate is 48.19%.

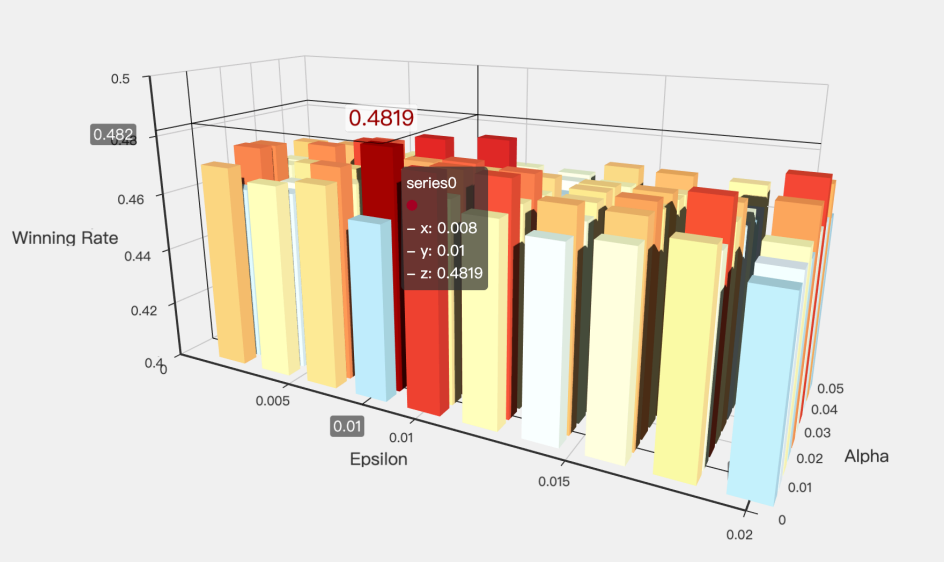


Figure 2: Result of TD against different parameters. **peak=0.4819**

2.3 Monte-Carlo (MC) in Q-learning

Another method to approximate the value function is to use Monte-Carlo approach.

2.3.1 Introduction to MC

Different from Temporal Difference, Monte Carlo updates the value function after a complete episode with incremental mean. Particularly, we use the **every-visit MC** method, which means that the every state seen in the episode will be updated. The process can be rewritten as following:

Algorithm 2: Every-Time MC

Input: original state, original action
Output: updated value

To evaluate state s;
Every time-step t that state s is visited in an episode;
Increment counter $N(s) \leftarrow N(s) + 1$;
Increment total return $S(s) \leftarrow S(s) + G_t$;
Value is estimated by mean return $V(s) = \frac{S(s)}{N(s)}$;
$V(s) \rightarrow v_\pi(s)$ as $N(s) \rightarrow \infty$;

2.3.2 Implementation

Firstly, we have to randomly initialize the states and actions, and create a dict to store all the elements that we've gone by, including states and actions. variable "totalreward" is for the increment total return.

```
game = Easy21()
dealer_score = np.random.randint(1,11)
player_score = np.random.randint(1,11)
state = (dealer_score,player_score)
```

```
action = np.random.randint(2)
totalreward = 0
terminal = False
trace = []
```

Since we have to use the increment mean, it is necessary to count for visits. Here, we use the matrix "N" to store the visiting information. All the visiting details are packed into "trace", which will be used later in updating. Also, the actions are chosen based on ϵ-greedy approach.

```
while not terminal:
  tmp = np.random.rand()
  if tmp < self.epsilon:
    action = np.random.randint(2)
  else:
    action = np.argmax(self.Q[state])
  self.N[state][action] += 1
  trace.append((state, action))
  state, reward, terminal = game.step(state, action)
  totalreward += reward
```

Finally, we use the formula introduced before to update the value in the following code:

```
for o_s, o_a in trace:
  alpha = 1/self.N[o_s][o_a]
  self.Q[o_s][o_a] += alpha*(totalreward-self.Q[o_s][o_a])
```

2.3.3 Result Display

After trial and error, we adjust the parameters to $\alpha = 0.1$, $\epsilon = 0.06$ and $episode = 100000$, and we get the following result in Fig.3.
It can be seen from the figure that when the reward drastically increases when the play sum reaches 17, this is intuitively because the dealer sticks when his cards sum up to 17.

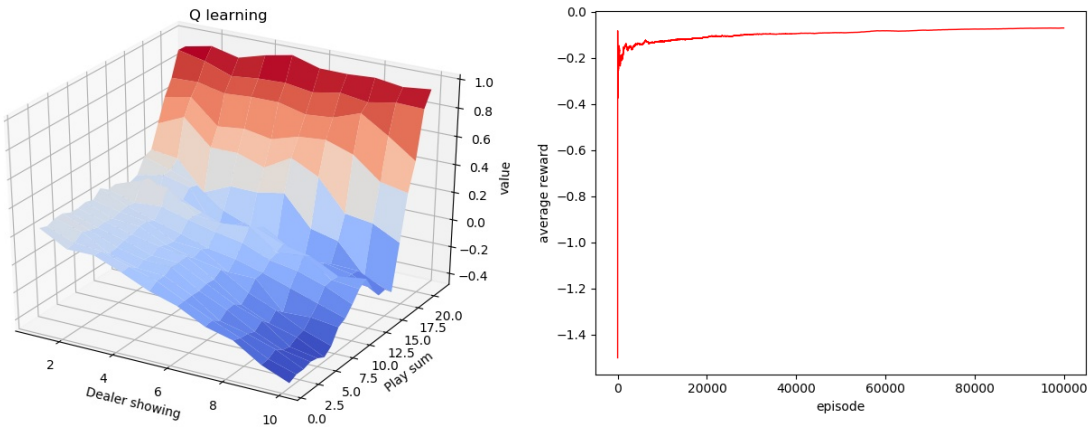


Figure 3: Reward and learning curve of the best result.

Similar as TD-learning, we plot the curve of winning rate against different parameters shown in Fig.4 with the highest winning rate 47.93%, slightly lower than TD-learning.

2.3.4 Analysis

Firstly, we compare the winning rate plot between Monte-Carlo and Temporal Difference. With the best number in this range, we will see that TD-learning is generally better than MC in that the

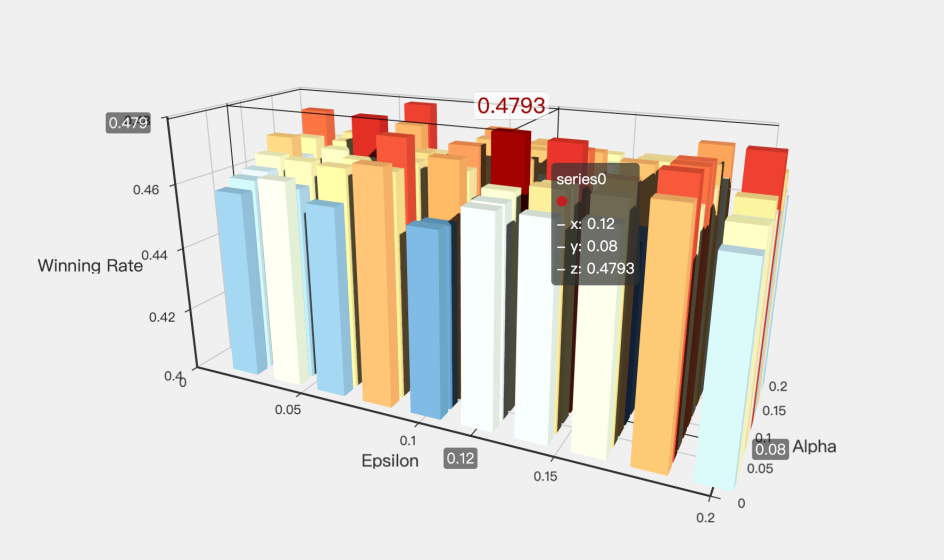


Figure 4: Result of MC against different parameters. **peak = 0.4793**

winning rate is higher in a whole.
To make the learning process better, we change epsilon as it goes in an episode as following:

```
epsilon = 100/(100+np.sum(self.Q[state][action]))
```

With episode number reaching 100000 and $\alpha = 0.1$, we come to the following result as shown in Fig.5, which seems better than constant ϵ.

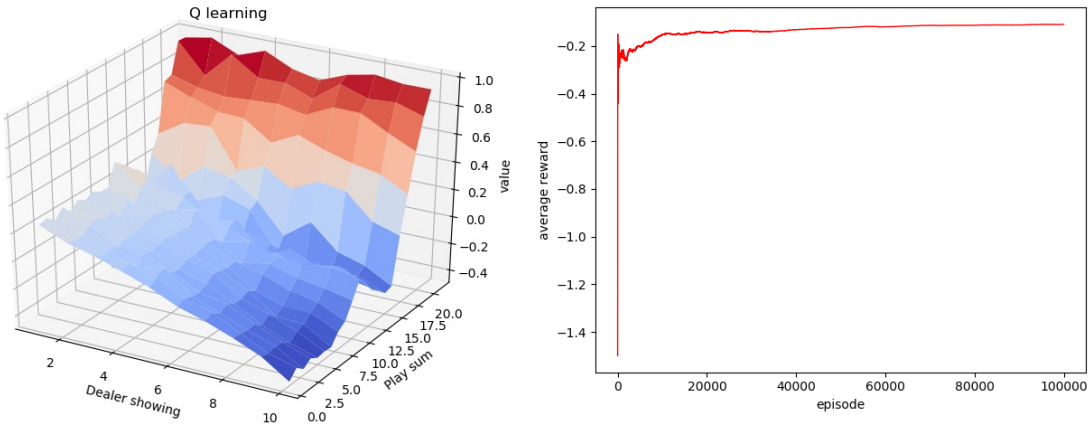


Figure 5: Result with changing ϵ

Universally, MC has great convergence properties while TD is more efficient than MC. The reason lies in that TD fully exploits Markov property, and is more effective in Markov-environment, while MC does not.
From the figure displayed in Fig.3 and Fig.5, we can see that learning process with a changing epsilon converges more quickly than that with a constant epsilon.

2.4 Policy Iteration

2.4.1 Introduction to policy iteration

The policy iteration includes two main steps:

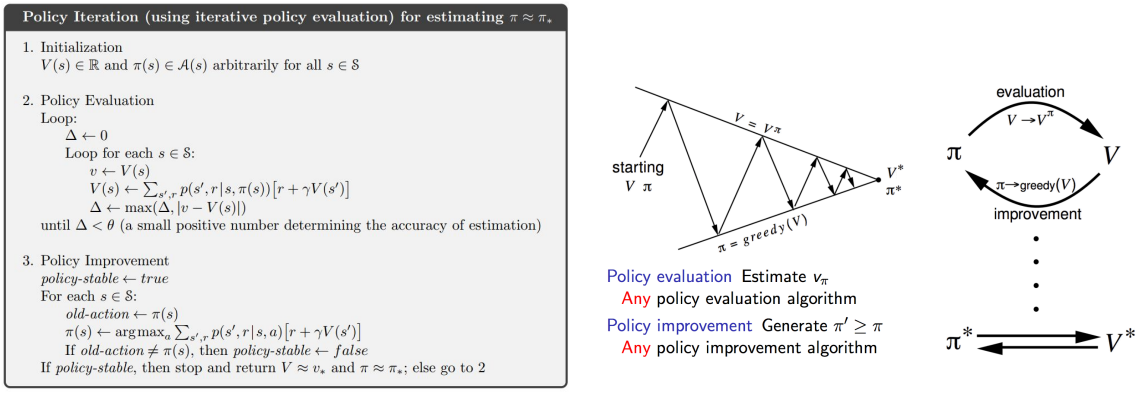


Figure 6: pseudo code and two basic steps of policy iteration

1. Policy Evaluation Estimate v_π Iterative policy evaluation
2. Policy Improvement Generate $\pi_0 \geq \pi$ Greedy policy improvement

Estimate v Iterative policy evaluation

$$\begin{aligned} v_{k+1}(s) &\doteq \mathbb{E}_\pi[R_{t+1} + \gamma v_k(S_{t+1}) \mid S_t = s] \\ &= \sum_a \pi(a|s) \sum_{s',r} p(s',r|s,a)\Big[r + \gamma v_k(s')\Big], \end{aligned}$$

Figure 7: mathematics expression of Bellman equation

2.4.2 Implementation

First give a look at the total structure of policy iteration.

```
class Policy_Iteration():

    def __init__(self,n_iter = 100000):
    ...
    def __init_P_table(self):
    ...
    def __init_V_table(self):
    ...
    def episode(self):
    ...
    def train(self):
    ...
```

Different from Q-learning, we need to initialize two tables here :
Policy and value.
Notice that I set the initialed value not 0:
(In fact,through experiment: setting as all 0 or other expected value for initialized value has little effect to the result.)
Policy are both 0.5 and value depends on the expected reward. (In fact,through experiment: setting as [1,0] , [0,1] or [2/3,1/3] is also OK.And the initialized policy has little to do with the result.)

```
    def __init_P_table(self):
        ...
        for state in states:
            P_table[state] = [0.5,0.5]
        # all hit
```

```
        return P_table
```

```
    def __init_V_table(self):
        # In fact,the value table is depending on the policy
        ...
        for state in states:
            if state[0] > state[1]:
                V_table[state] = -1
            elif state[0] < state[1]:
                V_table[state] = 1
            elif state[0] == state[1] :
                V_table[state] = 0
        return V_table
```

For the main function, we can discuss it in two parts:

1. Policy Evaluation
   The main purpose of this part is to solve for V(s).

```
 all_states = set()
for action in self.possible_actions:
    tmp = self.game.step(state,action)
    print(tmp)
    next_state = tmp[0]
    all_states.add(next_state)
    reward = tmp[1]
    totalreward += reward
    terminal = tmp[2]

    next_value = self.V[state]
    # Belleman equation
    print (self.P[state][action])
    self.V[state] += (self.P[state][action] *(reward + self.gamma * next_value))
```

   The core part is the Belleman equation:

```
self.V[state] += (self.P[state][action] *
                          (reward + self.gamma * next_value))
```

2. Policy Improvement
   Choose the best action and renew the policy

```
for state in all_states:
    value = -9999
    max_index = []
    result = [0,0]  # initialize the policy
```

   For every actions, calculate [reward + (discount factor) * (next state value function)]

```
for index, action in enumerate(self.possible_actions):
    next_state = tmp[0]
    reward = tmp[1]
    terminal = tmp[2]

    if not terminal:
        temp = reward + self.gamma * self.V[next_state]

        if temp == value:
        ...

        elif temp > value:
        ...

        # probability of action
try:
```

```
    prob = 1 / len(max_index)

    for index in max_index:
        self.P[state][index] = prob
except:
    continue
```

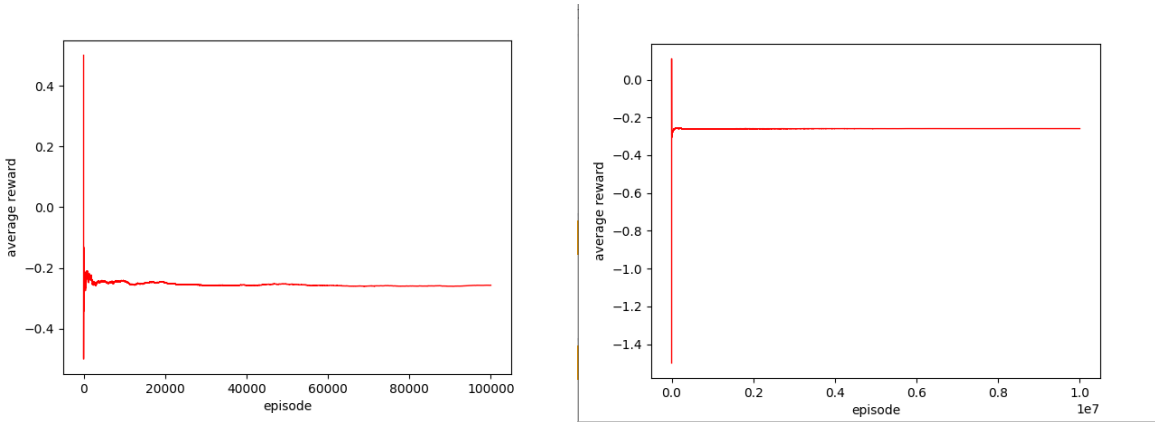


Figure 8: Left-hand side is the total reward of PI according to the increment of episode, right-hand side is the reward in more iterations.

Since the convergence happens very early,try to take less iteration times.
We can see that the convergence speed of Q learning is much more better than policy iteration.

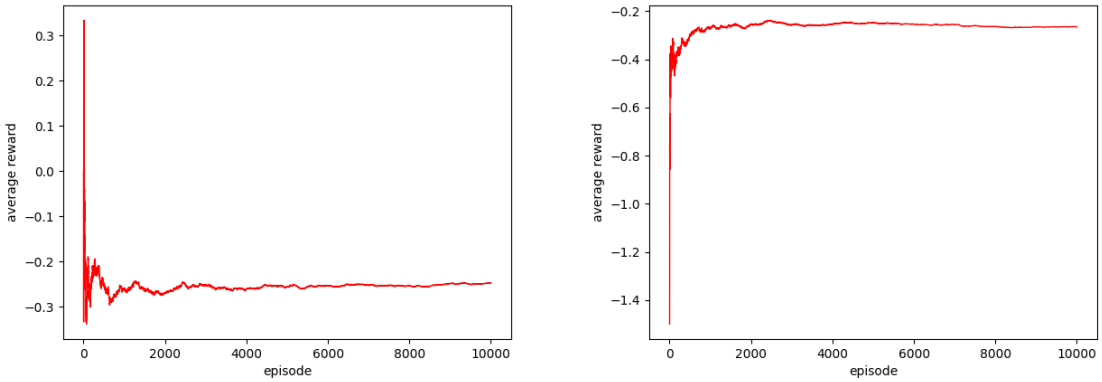


Figure 9: Less iteration times:10000

2.5 Conclusion and Comparison

2.5.1 Characteristic

This part elaborates the difference between policy iteration and value iteration.

Similarity
Q learning and policy iteration are both important methods in reinforcement learning.And the main idea of these two methods are the same:

- Decide the action which is going to take a_t according to current state s_t.

- Decide the next state s_{t+1}(after transmission) according to the current states$_t$ and current actiona_t.
- Give the responding reward r_{t+1} according to take certain action a_t at current state s_t.

So there are three main part here:

1. Environment S
2. Action of Agents A
3. Reward of environment R

By iterating the above many times, we can get a state-action chain: $s_0, a_0, s_1, a_1, ...s_{t-1}, a_{t-1}, s_t$.
Then the question comes: how to decide action by state and state by state in detail ? Here goes the answer.

1. From state to decide action
   We use π to denote the policy of agent which decide the choice of action.
   At t, choice of action has something to do with all the states and actions before t.
   ${a_t}^* = argmax_{a_{t,i}} p(a_{t,i}|s_0, a_0, ..., s_t)$
   Simplified as: ${a_t}^* = argmax_{a_{t,i}} p(a_{t,i}|s_t)$
   Then we have: $\pi(a|s) = P(A_t = a|S_t = s)$
2. From state to state
   The transmission: $P(S_{t+1}|S_t, a_t)$

Difference
Q learning is a kind of value iteration method while policy iteration is another.
Following the discussion in part "Same", we need to evaluate whether each action is a "good move" or a "bad move".
Here the value function and R play their parts.

$$v_\pi(s) = \sum_{a \in A} \pi(a|s)(R_s^a + \gamma \sum_{s' \in S} P_{ss'}^a v_\pi(s'))$$

$$q_\pi(s, a) = R_s^a + \gamma \sum_{s' \in S} P_{ss'}^a \sum_{a' \in A} \pi(a'|s') q_\pi(s', a')$$

By comparing the codes of two methods,we can see that in value iteration ,policy is changed as soon

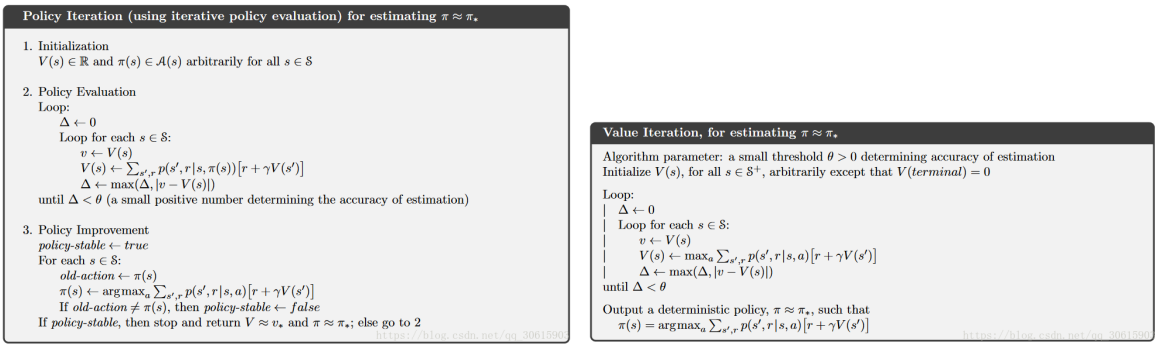


Figure 10: vi and pi algorithm

as value-state iteration is done. Instead of renew the policy after $q(s, a)$ is convergent.
So policy iteration has better convergent speed than value iteration.

2.5.2 Performance

Winning Rate

Both methods are not able to reach 50% winning rate.
While Q learning is able to achieve 47% and policy iteration is about 48%.
To get the above result, we need to deal with parameters such as α and ϵ, considering this process , policy iteration can reach its highest winning rate more easily.
In other words, at most of time, the winning rate of policy iteration is much more closer to 48% than Q learning to 47%.
In all, we can say that policy iteration has better performance than Q learning though the advantage is not very obvious.

Convergence Speed

policy iteration method has much better convergence speed than Q learning.
The above totalrewardepisode graph (Figure 5 & 8) can verify this apparently.

Amount of Calculation

Policy iteration does larger amount of calculation than Q learning.The main cost is in the Policy Evaluation part.
Considering time complexity,When the data size is large, Q learning may be a better choice.Otherwise, policy iteration may be the proper method.

3 Quanser Robot platform

3.1 Environment Introduction

Here we will give a detail introduction of each environment, including the structure of the model, the action space, observation space parameters, episode reward.

3.1.1 Qube

With OpenAI gym as the platform, we render the environment with the following code:

```
import gym
import quanser_robtos
env = gym.make('Qube-100-v0')
env.reset()
env.render()
```

Model Structure

After rendering the environment and read the guiding file, We can see that `Qube` is basically a rotary inverted pendulum model, with one arm and a servo in Fig.11.
The parameters of the system are fully illustrated in the figure. To speak in detail, θ and α denote the position of the arm and the pendulum link respectively, with angles greater than zero when rotating counter clockwise. The rest are the length and moment of inertia of the two sticks. The goal of our model is to make the servo stick up right.

Observation Space

By analyzing the source code of quanser_robots, the observation space is defined as follows:

```
self.observation_space = LabeledBox(
            labels=('cos_th', 'sin_th', 'cos_al', 'sin_al', 'th_d', 'al_d'),
            low=-obs_max, high=obs_max, dtype=np.float32)
```

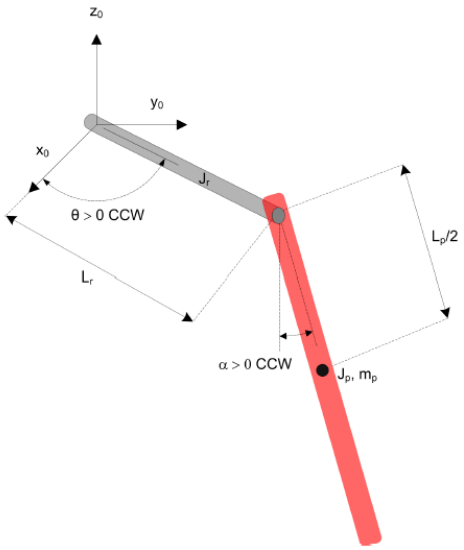


Figure 11: Structure of Qube

`cos_th`: the cosine value of θ.

`sin_th`: the sine value of α.

`cos_al`: the cosine value of θ.

`sin_al`: the sine value of α.

`th_d`: the derivative of θ, or more generally, angular velocity of the arm.

`al_d`: the derivative of α, or more generally, angular volocity of the pendulum link.

Action Space

The action space are difined in the following code:

```
self.action_space = LabeledBox(
            labels=('volts',),
            low=-act_max, high=act_max, dtype=np.float32)
```

`volts` indicates the control votage of the servo.

Episode Reward

```
def _rwd(self, x, a):
    th, al, thd, ald = x
    al_mod = al % (2 * np.pi) - np.pi
    cost = al_mod**2 + 5e-3*ald**2 + 1e-1*th**2 + 2e-2*thd**2 + 3e-3*a[0]**2
    done = not self.state_space.contains(x)
    rwd = np.exp(-cost) * self.timing.dt_ctrl
    return np.float32(rwd), done
```

3.1.2 BallBalancerSim

Similarly to `Qube`, we can call the `BallBalancerSim` environment with simple codes. And then we will see a ball on the plane.

```
import gym
import quanser_robtos
env = gym.make('BallBalancerSim-v0')
```

```
env.reset()
env.render()
```

Model Structure

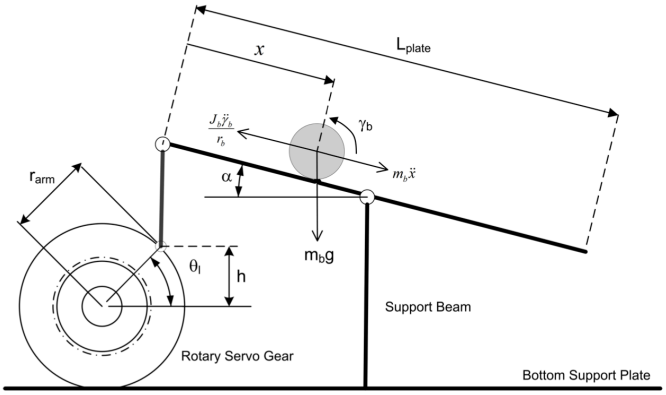


Figure 12: Structure of Ball Balancer

The free body diagram of the Ball and Beam is illustrated in Fig.16. Using this diagram, the equation of motion, or *EOM* for short, relating the motion of the ball, x, to the angle of the beam, α, can be found. Based on Newton's First Law of Motion, the sum of forces acting on the ball along the beam equals

$$m_b\ddot{x}(t) = \sum F = F_{x,t} - F_{x,r}$$

where m_b is the mass of the ball, x is the ball displacement, $F_{x,r}$ is the force from the ball's inertia, and $F_{x,t}$ is the translational force generated by gravity. Friction and viscous damping are neglected.

Observation Space

Dive into the source code of `BallBalancerSim`, we get that the observation space is composed of 8 elements.

```
self.observation_space = LabeledBox(
    labels=('theta_x', 'theta_y', 'pos_x', 'pos_x',
            'theta_x_dot_filt', 'theta_y_dot_filt', 'pos_x_dot_filt','pos_x_dot_filt'),
            low=-state_max, high=state_max, dtype=np.float32)
```

`theta_x`: x axis servo shaft angle

`theta_y`: y axis servo shaft angle

`pos_x`: ball position in meters along the x axis estimated by the "PGR Find Object" block

`pos_y`: ball position in meters along the x axis estimated by the "PGR Find Object" block

The remaining 4 parameters are about `velocity filter`. Filter data along one-dimension with an IIR or FIR filter

Action Space

We can also find the action space in source code.

```
self.action_space = LabeledBox(
    labels=('V_x', 'V_y'),
    low=-act_max, high=act_max, dtype=np.float32)
```

`V_x` and `V_y` representing voltage command for the X Axis Servo and for the Y Axis Servo

Episode Reward

```
def _rew_fcn(self, obs, action):
    err_s = (self._state_des - obs).reshape(-1,)  # or self._state
    err_a = action.reshape(-1,)
    quadr_cost = err_s.dot(self.Q.dot(err_s)) + err_a.dot(self.R.dot(err_a))

    obs_max = self.state_space.high.reshape(-1, )
    act_max = self.action_space.high.reshape(-1, )

    max_cost = obs_max.dot(self.Q.dot(obs_max)) + act_max.dot(self.R.dot(act_max))
    # Compute a scaling factor that sets the current state and action in relation to the
                                                    worst case
    self.c_max = -1.0 * np.log(self.min_rew) / max_cost

    # Calculate the scaled exponential
    rew = np.exp(-self.c_max * quadr_cost)  # c_max > 0, quard_cost >= 0
    return float(rew)
```

3.1.3 CarPoleSwing

The carpole is a classic environment used in control and in reinforcement learning.
It consists in a pole attached to a cart moving on a horizontal track.
The only actuated part of the system is the cart, which can be controlled usually by a horizontal force, or in our case by the voltage in input of the cart's engine.
We propose two classic tasks:

1. Pole balancing
2. swing-up

Pole balancing requires just to keep the pole balanced in vertical position without exceeding a certain threshold. Pole swing-up requires to swing-up the pole from the resting position and then balance it. Similarly to `Qube` and `BallBalancerSim` environment, we can call the `CarPoleSwing` environment with simple codes. The running result will show on the browser.

```
import gym
import quanser_robots
env = gym.make('CartpoleSwingShort-v0')
env.reset()
env.render()
```

Model Structure

The linear Single Pendulum Gantry (SPG) model is shown in Figure 8. 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.

Action Space

```
# Spaces
self.sensor_space = LabeledBox(
    labels=('x', 'theta'),
    low=-sens_max, high=sens_max, dtype=np.float32)
```

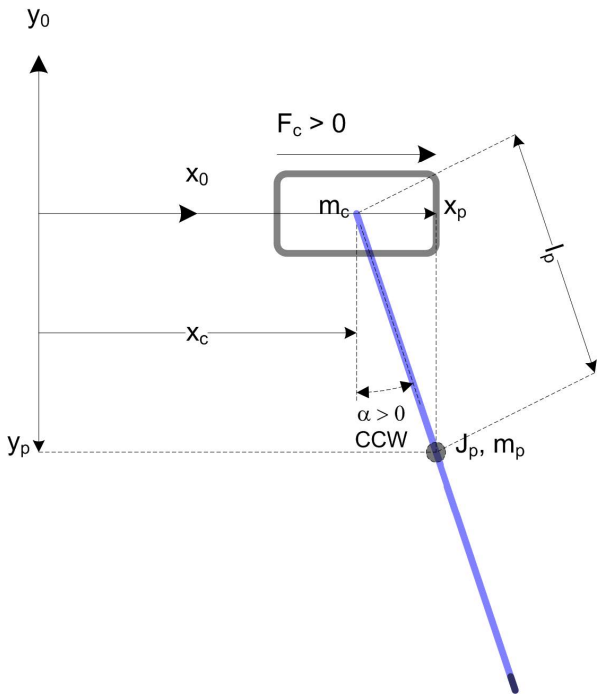


Figure 13: Structure of Ball CartPoleSwing

```
self.state_space = LabeledBox(
    labels=('x', 'theta', 'x_dot', 'theta_dot'),
    low=-state_max, high=state_max, dtype=np.float32)

self.observation_space = LabeledBox(
    labels=('x',  'sin_th', 'cos_th', 'x_dot', 'th_dot'),
    low=-obs_max, high=obs_max, dtype=np.float32)

self.action_space = LabeledBox(
    labels=('volts',),
    low=-act_max, high=act_max, dtype=np.float32)
```

The overall system can be described with two entities: the position x of the cart on the track and the angle of the pole theta.In order to control the system, we also need the derivates of this quantities, x_dot and theta_dot. In order to make the controller easier, and robust to angles which are out of the range (-pi, pi), we provide the cosine and the sine of theta.

Episode Reward

```
def _rwd(self, x, a):

    x_c, th, _, _ = x
    rwd = -np.cos(th)

    # Normalize theta to [-pi, +pi]
    th = np.mod(th + np.pi, 2. * np.pi) - np.pi

    done = False
    if self.stabilization:
        done = np.abs(th - np.sign(th) * np.pi) > self.stabilization_th
    done = done or np.abs(x_c) > self._x_lim - self.safe_range

    return np.float32(rwd) + 1., done
```

3.2 Model Prediction Control (MPC)

3.2.1 Introduction to MPC

Model Predictive Control or MPC is an advanced method of process control that has been in use in the process industries such as chemical plants and oil refineries since the 1980s and has proved itself. Model Predictive Controllers rely on the dynamic models of the process, most often linear empirical models obtained by system identification.

In the paper *Neural Network Dynamics for Model-Based Deep Reinforcement Learning with Model-Free Fine-Tuning* [2], the author shows that neural network dynamics models can in fact be combined with MPC to achieve excellent sample complexity in a model-based reinforcement learning. It produces stable and plausible gaits that accomplish various complex locomotion tasks. The paper also propose using deep neural network dynamics models to initialize a model-free learner, in order to combine the sample efficiency of model-based approaches with the high task specific performance of model-free methods.

The basic idea of MPC algorithm is 14:

1. Collecting training data
2. Training the model
3. Using model to estimate optimal action sequence
4. Choosing the first action in the sequence
5. Collecting current state to dataset and repeat 3. - 5.

Algorithm 1 Model-based Reinforcement Learning

gather dataset $\mathcal{D}_{\text{RAND}}$ of random trajectories
initialize empty dataset $\mathcal{D}_{\text{RL}}$, and randomly initialize $\hat{f}_\theta$
for iter=1 **to** max_iter **do**
  train $\hat{f}_\theta(\mathbf{s}, \mathbf{a})$ by performing gradient descent on Eqn. 1, using $\mathcal{D}_{\text{RAND}}$ and $\mathcal{D}_{\text{RL}}$
  for $t = 1$ **to** T **do**
    get agent's current state $\mathbf{s}_t$
    use $\hat{f}_\theta$ to estimate optimal action sequence $\mathbf{A}_t^{(H)}$ (Eqn. 3)
    execute first action $\mathbf{a}_t$ from selected action sequence $\mathbf{A}_t^{(H)}$
    add $(\mathbf{s}_t, \mathbf{a}_t)$ to $\mathcal{D}_{\text{RL}}$
  end for
end for

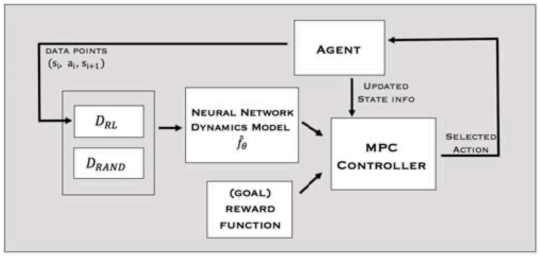


Figure 14: Algorithm of MPC

In our inference code, the structure of nerual network model is a MLP(multilayer perceptron), which is the simplest non-linear model structure. MLP consists of input layer, hidden layers and output layer. The activate function such as ReLU provide the non-linearity.

We can use various methods to estimate optimal action sequences, and here we will introduce 3 approaches(Random shooting; Simulated annealing; Artificial bee colony algorithm) to this problem.

3.2.2 Random Shooting Algorithm (RS)

This is the algorithm that proposed in the original paper. We just randomly generate a set of action sequences, evaluating their reward value, and choose the best one with the highest value.

Note that the action space can be more than one dimension, so the design of generating action sequences can be a little tricky with the help of `numpy` package.

```
class RandomSeeds():
    def __init__(self,lower, upper, fun):
        self.H = len(lower)
        self.lower = np.array(lower)
        self.upper = np.array(upper)
        self.evaluate = fun
        # randomly initialize actions on H steps
        self.actions = self.lower+(self.upper-self.lower)*np.random.rand(*self.lower.shape)
        self.value = fun(self.actions)

class RandomShooting():
    def __init__(self, lower, upper, fun, N):
        ...

    def run(self):
        candidates = [RandomSeeds(self.lower,self.upper,self.fun) for i in range(self.N)]
        _candidates = sorted(candidates, key=lambda x: x.value)
        self.solution = _candidates[0].actions
        return _candidates[0].value
```

3.2.3 Simulated Annealing Algorithm (SA)

Simulated annealing (SA) [3] is a probabilistic technique for approximating the global optimum of a given function. Specifically, it is a metaheuristic to approximate global optimization in a large search space for an optimization problem. It is often used when the search space is discrete. For problems where finding an approximate global optimum is more important than finding a precise local optimum in a fixed amount of time, simulated annealing may be preferable.

Actually, simulated annealing is a greedy algorithm, but its search process introduces stochastic factor simulated annealing algorithm to accept a solution worse than the current solution with a certain probability, so it is possible to jump out of the local optimal solution and reach the global optimal solution.

In our experiment, it can search a better solution with higher reward after SA, but it takes much more time to execute, and the hyperparameter is difficult to measure. As both RS and SA running N samples to find the best one, its time efficiency is apparently lower than RS.

```
import numpy as np

class Annealer_Core():
    def __init__(self,lower, upper, fun, scale):
        ...

    def move(self):
        # randomly search for every dimension
        # Here state is actions actually.
        actions = self.actions + (self.upper-self.lower)*(np.random.rand(*self.lower.shape)-
                                                          0.5)*self.scale
        actions = np.minimum(self.upper,actions)
        actions = np.maximum(self.lower,actions)
        return actions,self.evaluate(actions)

    def anneal(self):
        ...
        while T > T_min:
            new_actions, new_value = self.move()
            if new_value < self.old_value:
                self.old_value = new_value
                self.actions = new_actions
            else:
                p = np.exp((self.old_value - new_value) / T)
                r = np.random.uniform(0, 1)
                if p > r:
                    self.old_value = new_value
```

```
                    self.actions = new_actions
            T = T * a
        return self.actions, self.old_value

class SimAnneal():
    def __init__(self, lower, upper, fun, scale=1e-5):
        ...
    def run(self):
        for i in range(self.MAX_ITER):
            sa = Annealer_Core(self.lower,self.upper,self.fun,self.scale) # simulate
                                                                    annealing
            actions, value = sa.anneal()
            if not self.best_value:
                self.solution = actions
                self.best_value = value
            else:
                if value<self.best_value:
                    self.solution = actions
                    self.best_value = value
        self.solution = list(self.solution)
        return self.best_value
```

3.2.4 Artificial Bee Colony Algorithm (ABC)

In paper *An idea based on honey bee swarm for numerical optimization* [4], the author proposed a algorithm based on the intelligent foraging behaviour of honey bee swarm.

In the ABC model, the colony consists of three groups of bees: employed bees, onlookers and scouts. It is assumed that there is only one artificial employed bee for each food source. In other words, the number of employed bees in the colony is equal to the number of food sources around the hive. Employed bees go to their food source and come back to hive and dance on this area. The employed bee whose food source has been abandoned becomes a scout and starts to search for finding a new food source. Onlookers watch the dances of employed bees and choose food sources depending on dances.

The main step ABC algorithm is

1. Initialize food source, setting relevant parameters
2. Assign an employed bee to the honey source, and search among sources and generate new honey source
3. Evaluate the fitness, and choose the reserved honey source
4. Calculate the possibility of honey source to be followed
5. The onlooker was searched in the same way as the employed bees, and the remaining honey source was determined by greedy selection
6. Judge whether the honey source meets the condition to be abandoned. If yes, the corresponding employed bee will become scouts, otherwise, go directly to 8.
7. Scouts generate a random honey source
8. Judge whether the algorithm meets the termination conditions. If so, it will terminate and output the optimal solution. Otherwise, it goes to 2.

3.2.5 Result Display

We tried the **default configure** for every method (except the SA on BallBalancer, which is very very slow). The default parameter is not very perfect, we continue the on the method ABC to find the best parameter.

1. **Qube-100-v0**

   We first rerun the given default configure file. We think it is a basic step of researching, to get a overall perspective of previous experiment.

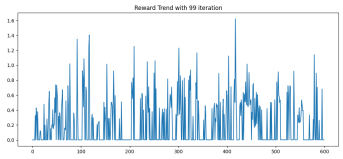
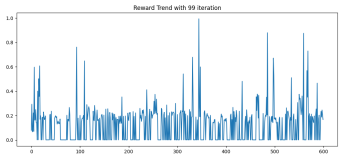
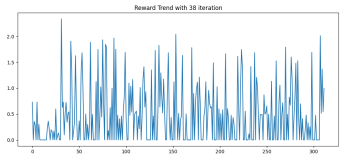
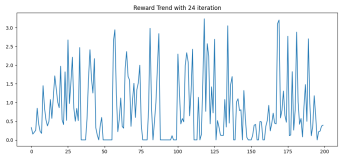
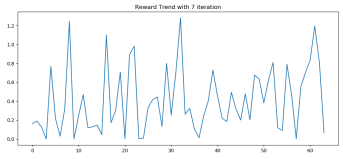
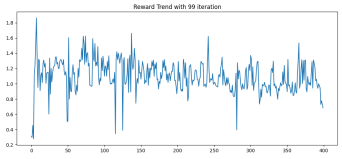
config	Reward	**config**	Reward
config-1		config-2	
config-4		config-5	
config-6		config-7	

Table 1: rerun the given default configure of Qube

Then we can find it's the **config-5** achieves the best reward – getting a reward greater than 3. So based on **config-5**, we start to tune these hyperparameters. Here we changes the batchsize and the horizon.

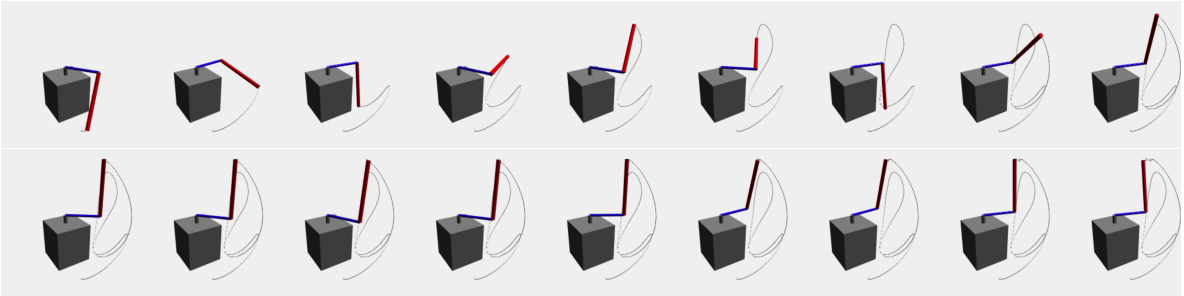


Figure 15: Movement of our Qube (successful, reward = 2.03)

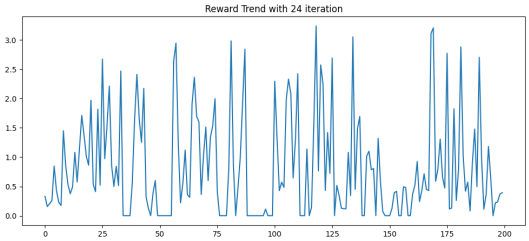
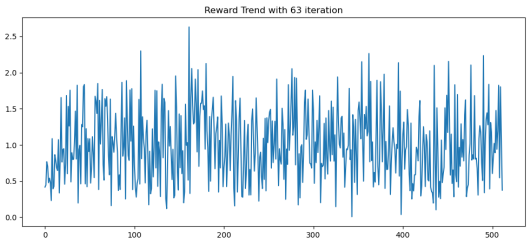
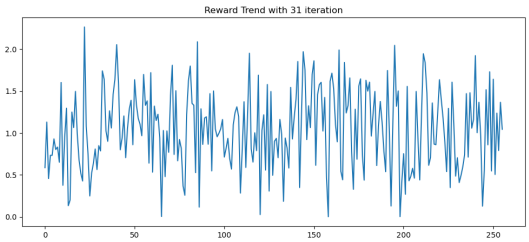
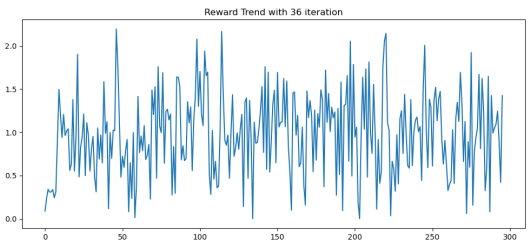
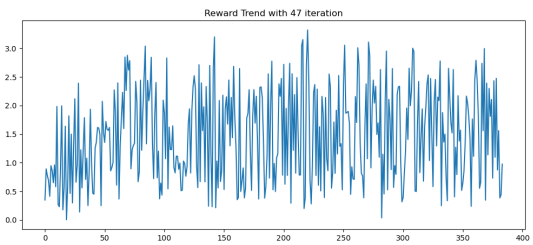
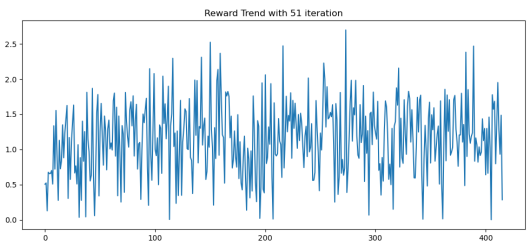
config	Batchsize	Horizon	Reward
config-5	64	30	
config-16	32	30	
config-17	64	50	
config-18	64	60	
config-19	128	30	
config-20	128	40	

Table 3: Tuning Hyper parameters of Qube

Analysis:

Comparing **config-5**, **config-16**, **config-19**, we find that batchsize actually affect the performance. Between batchsize 32, 64 and 128, we find that batchsize 128 can get reward 3 more quickily, while the batchsize 32 only stop at reward 2.5 .

Then we fix the batchsize= 64, and test the horizon's effect. Comparing **config-5**, **config-17**, **config-18**, we are surprised to find larger horizon here makes no difference, or even reduce the performance. Both **config-17** and **config-18** failed to get reward of 3.

Then we continue to tune the Horizon on fixed batchsize= 128. We also find **config-19**(horizon=30) is greater than **config-20**(horizon=40). And it further confirmed our conjecture.

In conclusion, **Batchsize** is quite critical. The larger, the better. And we should not require too high **Horizon**, the optimal value should be less than 50.

2. **CartPoleSwing**

We also first rerun the given configure file.

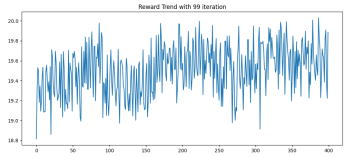
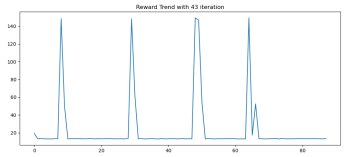
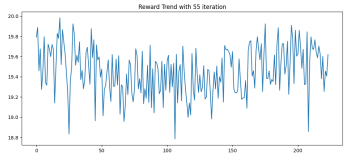
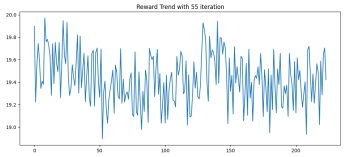
config	Reward	**config**	Reward
config-1		config-2	
config-3		config-4	
config-5		/	/

Table 4: rerun the given default configure of CartPoleSwing

Here the best configure file should be **config-2**, which can get out of the dilemma (moving ahead to the bound). Here this agent can learn to move forward and backward to swing the pole. However, the agent seems hesitant, failed to supply a speed fast enough to finally swing up the pole.

Then we focus on the **config-2**, and tune some hyper parameters of it.

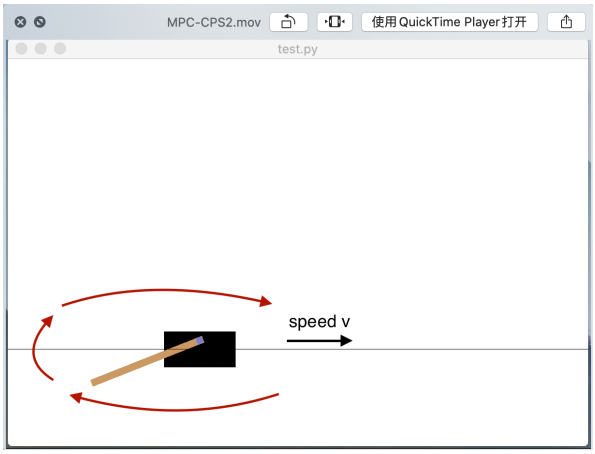


Figure 16: Movement of our agent

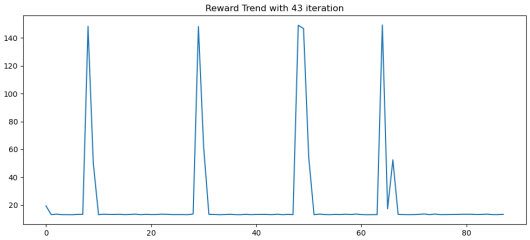
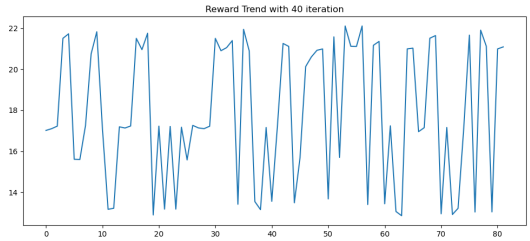
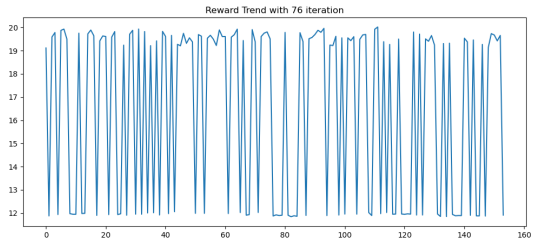
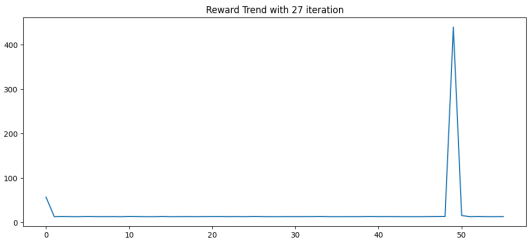
config	Max N Step	Horizon	Reward
config-2	500	30	
config-8	500	40	
config-9	500	20	
config-10	1000	30	

Table 6: Tuning Hyper parameters of CartPoleSwing

Analysis:

Here we try to increase and decrease the horizon on fixed Max N Step=500. But both these two version of agent just learned to move to the bound ahead, leaving a best reward of 20.

Here we believe that increasing Max N step, **config-10** can get better reward. But it is so cost expensive that we have to spend more than 2 hours to run one iteration (on CPU). Finally we get a **reward greater than 400** at episode near 50. If we train more times, it's possible to get much greater reward. But we find that the testing phase is not very pleasant, the pole can not swing to the top. It's reasonable because peak reward only appear once, and most of the time the reward near 13.

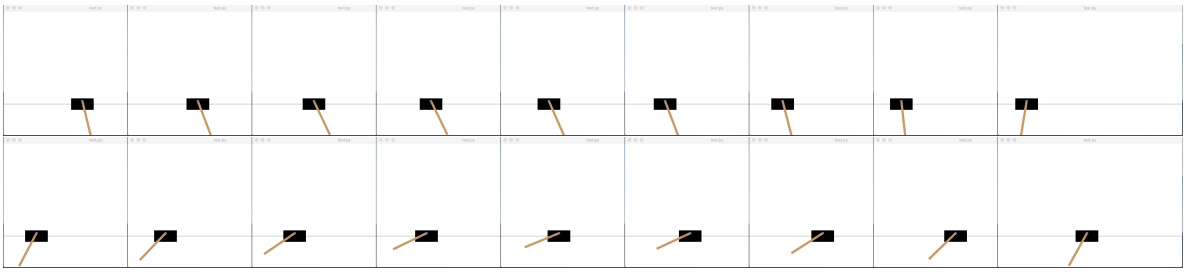
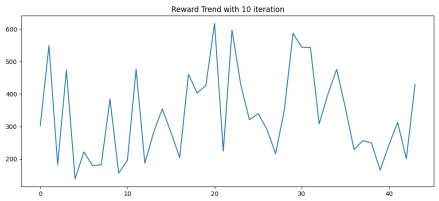
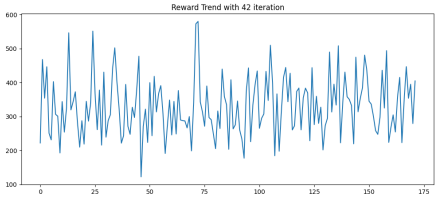
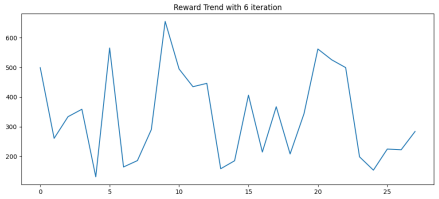
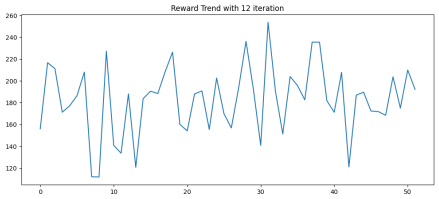
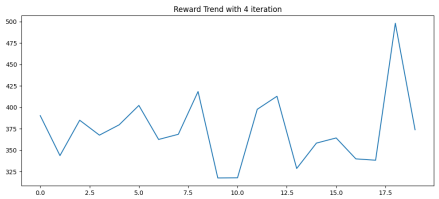
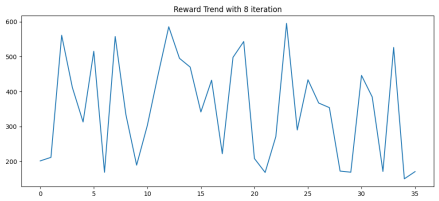
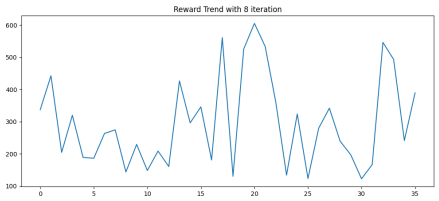
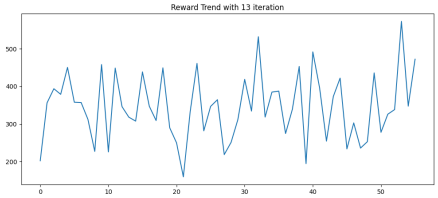
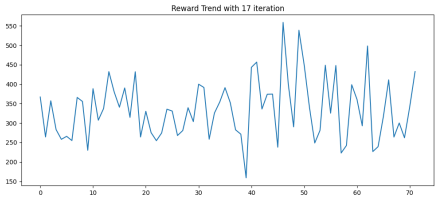


Figure 17: Movement of our CartPoleSwing (reward = 107.3)

3. **BallBalancerSim**

Here we have no default configure file, we start to tune the hyperparameter ourselves. And we want to distinguish the disparity of ABC and RS algorithm.

config	max n step	horizon	gamma in ABC	random N in RS	Reward
config-21	1000	5	0.999	/	
config-22	1000	5	/	100	
config-24	1500	5	0.999	/	
config-25	500	5	0.999	/	
config-26	1000	8	0.999	/	
config-27	1000	5	0.99	/	
config-28	1000	5	0.9	/	

config-29	1000	5	/	200	
config-30	1000	5	/	300	

Analysis:
In this section, we compared ABC algorithm and RS algorithm in detail. Comparing **config-21**(ABC) and **config-22**(RS), we find both these two optimizer can achieve reward of **600**.

We first increase the *Max N Step* in ABC algorithm. We use **config-21**(*Max N Step*=1000), **config-24**(*Max N Step*=1500) and **config-25**(*Max N Step*=500). Then we find that the **config-25**(*Max N Step*=500) failed to get the same performance of the other config. It demonstate that parameter – *Max N Step* – has a great influence. It is reasonable that if we limit the max N step, it is difficult for the agent to search the best action.

Then we change the gamma in ABC optimizer. We use **config-21**(gamma=0.999), **config-27**(gamma=0.99) and **config-28**(gamma=0.9). It can be seen that curve is very similar, ranging from 100 to 600. If we have more computational resource, we will try more gamma configure.

Finallly we change the N in RS optimizer. We use **config-22**(N=100), **config-29**(N=200) and **config-30**(N=300). Higher N requires longer computation time. For the result, **config-30**(N=300) have a higher lower bound, which shows increasing N is useful to some degrees. However, it seems not to be the bottleneck of The MPC algorithm. Because they all fail to exceed reward of 600.

3.3 Trust Region Policy Optimization (TRPO)

3.3.1 Introduction to TRPO

TRPO is a scalable algorithm for optimizing policies in reinforcement learning by gradient descent. Model-free algorithms such as policy gradient methods do not require access to a model of the environment and often enjoy better practical stability. Consequently, while straightforward to apply to new problems, they have trouble scaling to large, nonlinear policies. TRPO couples insights from reinforcement learning and optimization theory to develop an algorithm which, under certain assumptions, provides guarantees for monotonic improvement.
In TRPO, we have to maximize the accumulated reward $\eta(\pi_\theta) = E_{s_0,a_0,\ldots} \sum_{t=0}^{\gamma} {}^t r(s_t)$, however, the right-hand side is sampled using $\pi_\theta(a_t|s_t)$, which cannot be easily used to calculate $\eta(\pi_\theta)$. Hence, a surrogate function is introduced to approximate $\eta(\pi_\theta)$, or to be accurately speaking, it should be the lower bound of $\eta(\pi_\theta)$ and when the surrogate function increases, $\eta(\pi_\theta)$ is at least non-decreasing.
For the sake of finding the surrogate function, we need to prove the following expression:

$$\eta(\tilde{\pi}) = \eta(\pi) + \sum_s \rho_{\tilde{\pi}}(s) \sum_a \tilde{\pi}(a|s)$$

Take π as the old policy and $\tilde{\pi}$ as the new, as long as $\sum_s \rho_{\tilde{\pi}}(s) \sum_a \tilde{\pi}(a|s) > 0$, the accumulated reward will be improved. In fact, we are maximizing $\eta(\tilde{\pi})$.

(a) Line search (like gradient descent) (b) Trust region

Figure 18: Demonstration of trust region notion

In the expression, $\rho_{\tilde{\pi}}$ is the frequency of every state occurring in the new policy $\tilde{\pi}$. In order to minimize the number of sampling, we define such surrogate function as

$$L_\pi(\tilde{\pi}) = \eta(\pi) + \sum_s \rho_\pi(s) \sum_a \tilde{\pi}(a|s) A_\pi(s,a)$$

which substitute the new policy into the old one.

Proof 3.1

$$\begin{aligned}
E_{\tau|\tilde{\pi}} \sum_{t=0}^{\infty} \gamma^t A_\pi(s_t|a_t) &= E_{\tau|\tilde{\pi}} \sum_{t=0}^{\infty} \gamma^t (r(s_t) + \gamma V_\pi(s_{t+1}) - V_\pi(s_t)) \\
&= -E_{s_0} V_\pi(s_0) + E_{\tau|\tilde{\pi}} \sum_{t=0}^{\infty} \gamma^t r(s_t) \\
&= -\eta(\pi) + \eta(\tilde{\pi}) \\
&= \sum_{t=0}^{\infty} \sum_s P(s_t = s|\tilde{\pi}) \sum_a \tilde{\pi}(a|s) \gamma^t A_\pi(s,a) \\
&= \sum_s \sum_{t=0}^{\infty} \gamma^t P(s_t = s|\tilde{\pi}) \sum_a \tilde{\pi}(a|s) A_\pi(s,a) \\
&= \sum_s \rho_{\tilde{\pi}}(s) \sum_a \tilde{\pi}(a|s)
\end{aligned}$$

To stat the frequency of the old policy π is relatively easy as we have already done the sampling of π. Also, to optimize $L_\pi(\tilde{\pi})$ is simple as we can directly compute the gradient for it. Notice:

$$L_{\pi_{\theta_0}} = \eta(\pi_{\theta_0})$$

$$\nabla_\theta L_{\pi_{\theta_0}}(\pi_\theta)|_{\theta=\theta_0} = \nabla_\theta \eta(\pi_\theta)|_{\theta=\theta_0}$$

The above inference are based on the condition that $L_\pi(\tilde{\pi})$ is approximate to $\eta\tilde{\pi}$. To restrict θ in the trust region is to limit its updating rate as shown in Fig.18. The so-called trust region is KL-divergence restriction.

In order to guarantee the condition, the original paper [5] propose the notion of **Trust Region**, which means that when θ adjust in a certain range, the condition is always satisfied. In the whole, the optimization of TRPO becomes:

$$\max_\theta L_{\theta_{old}}(\theta)$$

$$\text{subject to } D_{KL}^{max}(\theta_{old}, \theta) \leq \delta$$

where $\eta(\theta) := \eta(\pi_\theta)$, $L_\theta(\tilde{\theta}) := L_{\pi_\theta}(\pi_{\tilde{\theta}})$, and $D_{KL}(\theta|\tilde{\theta}) := D_{KL}(\pi_\theta||\pi\pi_{\tilde{\theta}})$ This restriction is not easy to deal with, since it requires us to restrict all the stated, which costs a lot. Therefore, we can further make an approximation which utilizes the optimal with average, i.e.

$$\max_\theta L_{\theta_{old}}(\theta)$$

$$\text{subject to } \bar{D}_{KL}^{\rho\theta_{old}}(\theta_{old}, \theta) \leq \delta$$

where $\bar{D}_{KL}^{\rho\theta_{old}}(\theta_1, \theta_2)$ stands for the frequency of stated sampled using policy π, i.e. $E_{s\sim\rho_\theta} D_{KL}(\pi_{\theta_1}(\cdot|s)||(\pi_{\theta_2})(\cdot|s))$

Directly get the expection in the expression is impossible, we use Monte-Carlo to approximate. According to **Importance Sampling**, $\pi_\theta(a|s) = q(a|s)\frac{\pi_\theta(a|s)}{q(a|s)}$. Then we get

$$\sum_a \pi_\theta(a|s_n) A_{\theta_{old}(s_n,a)} = E_{a\sim q} \frac{\pi_\theta(a|s_n)}{q(a|s_n)} A_{\theta_{old}(s_n,a)}$$

In conclusion, we get the expression used for training:

$$\max E_{s\sim\rho_{\theta_{old}}, a\sim q} \frac{\pi_\theta(a|s)}{q(a|s)} Q_{\theta_{old}}(s,a)$$

$$\text{subject to } E_{s\sim\rho_{\theta_{old}}} D_{KL}(\pi_{\theta_{old}}(\cdot|s)||\pi_\theta(\cdot|s)) \leq \delta$$

The overall process of the TRPO algorithm is shown in Fig.19.

Algorithm 1 Trust Region Policy Optimization

Input: initial policy parameters θ_0, initial value function parameters ϕ_0

Hyperparameters: KL-divergence limit δ, backtracking coefficient α, maximum number of backtracking steps K

for $k = 0, 1, 2, ...$ **do**

Collect set of trajectories $\mathcal{D}_k = \{\tau_i\}$ by running policy $\pi_k = \pi(\theta_k)$ in the environment.

Compute rewards-to-go $\hat{R}_t$.

Compute advantage estimates, $\hat{A}_t$ (using any method of advantage estimation) based on the current value function V_{ϕ_k}.

Estimate policy gradient as

$$\hat{g}_k = \frac{1}{|\mathcal{D}_k|} \sum_{\tau \in \mathcal{D}_k} \sum_{t=0}^{T} \nabla_\theta \log \pi_\theta(a_t|s_t)|_{\theta_k} \hat{A}_t.$$

Use the conjugate gradient algorithm to compute

$$\hat{x}_k \approx \hat{H}_k^{-1} \hat{g}_k,$$

where $\hat{H}_k$ is the Hessian of the sample average KL-divergence.

Update the policy by backtracking line search with

$$\theta_{k+1} = \theta_k + \alpha^j \sqrt{\frac{2\delta}{\hat{x}_k^T \hat{H}_k \hat{x}_k}} \hat{x}_k,$$

where $j \in \{0, 1, 2, ...K\}$ is the smallest value which improves the sample loss and satisfies the sample KL-divergence constraint.

Fit value function by regression on mean-squared error:

$$\phi_{k+1} = \arg\min_\phi \frac{1}{|\mathcal{D}_k|T} \sum_{\tau \in \mathcal{D}_k} \sum_{t=0}^{T} \left(V_\phi(s_t) - \hat{R}_t \right)^2,$$

typically via some gradient descent algorithm.

end for

Figure 19: Overall process or TRPO algorithm

3.3.2 Implementation of TRPO

The core idea of TRPO method is to let the new or updated policy reward function to be monotonous.The a very natural idea comes out:
Write out the reward function corresponding to the new policy in the form of sum of the reward function corresponding to the old policy and another terms. In mathematical words:

$$\eta(\hat{\pi}) = \eta(\pi) + E_{s_0, a_0, \cdots \hat{\pi}} \left[\sum_{t=0}^{\infty} \gamma^t A_\pi \left(s_t, a_t \right) \right]$$

Figure 20: core idea of TRPO

From this "very important equation": we need to discuss η and A_π.

1. η is a discounted reward function

$$\eta(\pi_\theta) = \mathop{\mathrm{E}}_{\tau \sim \pi_\theta}\left[\sum_{t=0}^{\infty} \gamma^t r_t\right]$$

Figure 21: Discounted reward function

While we all know that the final goal of RL is to maximize the the expected discounted rewards.

2. A_π is an advantage function

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

Figure 22: A_π

And in TRPO, the iteration method is called MM(Minorize-Maximization).In detailed explaination, that is we use a surrogate function M to decide the current policy.
M = L(θ) - KL
We can see that M has several characters:

- M is the lower bounded function of η
- M is useful to evaluate the discounted reward η of the current policy
- Easy to optimize(we can use a quadratic equation to approximately evaluate the surrogate function)

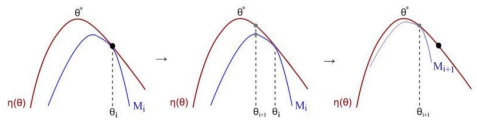


Figure 23: In the iteration we find better M to be the current policy and redo the process of evaluating the new lower bound of policy

Building the module of policy network.

```
class Policy_Network(nn.Module):
    def __init__(self, obs_space, act_space):
        super(Policy, self).__init__()
    ...

    def forward(self, x):
        ...
        return action_mean, action_log_std, action_std
```

Building the module of value network.

```
class Value_Network(nn.Module):
    def __init__(self, obs_space):
        super(Value_Network, self).__init__()
        ...

    def forward(self, x):
        ...
        return state_values
```

The conjugate gradients method of optimizing the model.

```
def conjugate_gradients(Avp, b, nsteps, residual_tol=1e-10):

    for i in range(nsteps):
        _Avp = Avp(p)
        alpha = rdotr / torch.dot(p, _Avp)
        x += alpha * p
        r -= alpha * _Avp
        new_rdotr = torch.dot(r, r)
        betta = new_rdotr / rdotr
        p = r + betta * p
        rdotr = new_rdotr
        if rdotr < residual_tol:
            break
    return x
```

Do the linear search to ensure the restrictive condition is satisfied.

```
def linesearch(model, f, x, fullstep , expected_improve_rate,
               max_backtracks=10, accept_ratio=.1):
    return False, x
```

The conjugate gradients and linear search method are the step after the linear approximation and the quadratic restriction to solve Updating the parameters.(Including many important parts of fucntions).

```
def update_params(batch):
    ...
    def get_value_loss():
    def get_loss():
    def get_kl():
    ...
        trpo_step(policy_net, get_loss, get_kl, args.max_kl, args.damping)
    loss.append(get_loss())
```

```
for i in reversed(range(rewards.size(0))):
        returns[i] = rewards[i] + args.gamma * prev_return * masks[i]
        deltas[i] = rewards[i] + args.gamma * prev_value * masks[i] - values.data[i]
        advantages[i] = deltas[i] + args.gamma * args.tau * prev_advantage * masks[i]

        prev_return = returns[i, 0]
        prev_value = values.data[i, 0]
        prev_advantage = advantages[i, 0]

    targets = Variable(returns)
```

Use the LBFGS to optimize the value loss.
Build the function $L_\pi(\hat{\pi})$

```
    def get_value_loss(flat_params):
        ...

        value_loss = (values_ - targets).pow(2).mean()
```

Using the LBFGS algothrim to optimize the unconstrained problem.

```
        flat_params, _, opt_info = optimize.fmin_l_bfgs_b(func=get_value_loss, x0=
                                                  get_flat_params_from(self.value).
                                                  double().numpy(), maxiter=25)
        set_flat_params_to(self.value, torch.Tensor(flat_params))
```

normalize the advantage function.

```
        advantages = (advantages - advantages.mean()) / advantages.std()
```

Calculate the loss of the policy network.

```
def get_loss(volatile=False):
        if volatile:
            with torch.no_grad():
                action_means, action_log_stds, action_stds = policy_net(Variable(states))
        else:
            action_means, action_log_stds, action_stds = policy_net(Variable(states))

        ...
        return action_loss.mean()
```

Another important trick used in TRPO is to use **KL-divergence** to ensure the approximation, which is implemented as following:

```
def get_kl():
    mean1, log_std1, std1 = policy_net(Variable(states))

    mean0 = Variable(mean1.data)
    log_std0 = Variable(log_std1.data)
    std0 = Variable(std1.data)
    kl = log_std1 - log_std0 + (std0.pow(2) + (mean0 - mean1).pow(2)) / (2.0 * std1.pow(2))
                                                        - 0.5
    return kl.sum(1, keepdim=True)
```

Each step of TRPO.

```
def trpo_step(model, get_loss, get_kl, max_kl, damping):
    ...
    return loss
```

Calculate the loss of policy network.
Calculate the gradients.

```
def trpo_step(model, get_loss, get_kl, max_kl, damping):
    loss = get_loss()
    grads = torch.autograd.grad(loss, model.parameters())
    loss_grad = torch.cat([grad.view(-1) for grad in grads]).data
```

Calculate the loss of policy network.

```
    def Fvp(v):
        kl = get_kl()
        kl = kl.mean()
```

Mean value of kl divergence.
Overall, the training process of TRPO is written to be the following:

```
running_state = ZFilter((num_inputs,), clip=5)
running_reward = ZFilter((1,), demean=False, clip=10)

reward_list = []
for i_episode in trange(args.max_iter_num):
    memory = Memory()

    num_steps = 0
    reward_batch = 0
    num_episodes = 0
    while num_steps < args.batch_size:
        state = env.reset()
        state = running_state(state)

        reward_sum = 0
        for t in range(10000):            # Don't infinite loop while learning
            action = select_action(state)
            action = action.data[0].numpy()
```

```
            next_state, reward, done, _ = env.step(action)
            reward_sum += reward

            next_state = running_state(next_state)

            mask = 1
            if done:
                mask = 0

            memory.push(state, np.array([action]), mask, next_state, reward)

            if args.render:
                env.render()
            if done:
                break

            state = next_state
        num_steps += (t-1)
        num_episodes += 1
        reward_batch += reward_sum

    reward_batch /= num_episodes
    batch = memory.sample()
    update_params(batch)
    reward_list.append(reward_batch)

    if i_episode % args.log_interval == 0:
        ...(plot figure of loss and reward)

    if args.save_model_interval > 0 and (i_episode + 1) % args.save_model_interval == 0:
        ..(save model parameters)
```

3.3.3 Possibe Improvement of TRPO

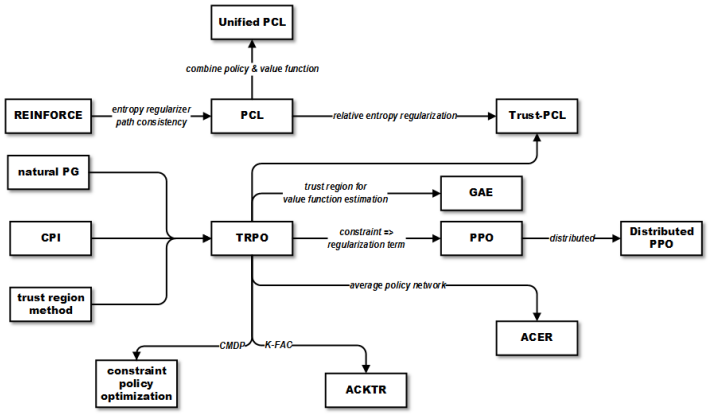


Figure 24: The related methods around TRPO

Proximal Policy Optimization Algorithms : PPO [6]
PPO method is the improved Algorithm of TRPO, it is more easy to implement and has better data efficiency.

Method	How to ensure the stability of policy updating	How to optimize the function
TRPO	constraint method	(a). Do linear approximation to the goal function (b). Do quadratic constraints to restrcit (c). Do conjugate gradient and line search
PPO	punishment term(regular term)	linear optimziation: goal function based on clipped probability ratio

Table 8: Comparison between TRPO and PPO

$$\begin{aligned} \max_{\theta} \quad & L_{old}(\theta) \\ s.t. \quad & \bar{D}_{KL}(\theta_{old}, \theta) \leq \delta \end{aligned}$$

Figure 25: TRPO method optimizing

Algorithm 1 Proximal Policy Optimization (adapted from [8])

for $i \in \{1, \cdots, N\}$ **do**
  Run policy π_θ for T timesteps, collecting $\{s_t, a_t, r_t\}$
  Estimate advantages $\hat{A}_t = \sum_{t'>t} \gamma^{t'-t} r_{t'} - V_\phi(s_t)$
  $\pi_{\text{old}} \leftarrow \pi_\theta$
  for $j \in \{1, \cdots, M\}$ **do**
    $J_{PPO}(\theta) = \sum_{t=1}^{T} \frac{\pi_\theta(a_t|s_t)}{\pi_{old}(a_t|s_t)} \hat{A}_t - \lambda \text{KL}[\pi_{old}|\pi_\theta]$
    Update θ by a gradient method w.r.t. $J_{PPO}(\theta)$
  end for
  for $j \in \{1, \cdots, B\}$ **do**
    $L_{BL}(\phi) = -\sum_{t=1}^{T} (\sum_{t'>t} \gamma^{t'-t} r_{t'} - V_\phi(s_t))^2$
    Update ϕ by a gradient method w.r.t. $L_{BL}(\phi)$
  end for
  if $\text{KL}[\pi_{old}|\pi_\theta] > \beta_{high} \text{KL}_{target}$ **then**
    $\lambda \leftarrow \alpha\lambda$
  else if $\text{KL}[\pi_{old}|\pi_\theta] < \beta_{low} \text{KL}_{target}$ **then**
    $\lambda \leftarrow \lambda/\alpha$
  end if
end for

Figure 26: psedo code of PPO

Another kind of goal function is using self-adapated KL penality term while the performance may not be as good as the clipping one.

$$L^{CLIP}(\theta) = \hat{\mathbb{E}}_t[\min(r_t(\theta)\hat{A}_t, \text{clip}(r_t(\theta), 1-\epsilon, 1+\epsilon)\hat{A}_t)]$$

$$L^{KLPEN}(\theta) = \hat{\mathbb{E}}_t[\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{old}}(a_t|s_t)}\hat{A}_t - \beta \, \text{KL}[\pi_{\theta_{old}(\cdot|s_t)}, \pi_\theta(\cdot|s_t)]]$$

Figure 27: The two kind of goal function of PPO

1. Clipped Surrogate Objective
2. Adaptive KL Penalty Coefficient

DRPO
Distributed PPO [7].
DPRO method is very useful to deal with situations that reaward function is not so "well-defined" as in many assumptions of RL algothrims.

The shortage of PG method is that variance is too high and too sensitive to hype-parameters.

1. In TRPO solution is using trust region to constraint.
2. In PPO, trust region is implemented as regular term
3. In DRPO, the clloecting of data and calculation of gradients are distributed in several works, like the idea of A3c algothrim.

PCL & TPCL
Path Consistency Learning [8, 9] is a kind of off-policy AC method.
PCL adds the discounted entropy regularizer in the traditional expect regression function,which could help to avoid converging too fast to reach sub-optimal solution.

$$O_{\mathrm{ENT}}(s,\pi) = O_{\mathrm{ER}}(s,\pi) + \tau\mathbb{H}(s,\pi)$$

Figure 28: Expected Return after the regulation

$$O_{\mathrm{PCL}}(\theta,\phi) = \sum_{s_{i:i+d}\in E} \frac{1}{2} C(s_{i:i+d},\theta,\phi)^2$$

$$V_\rho(s) = \tau \log \sum_a \exp Q_\rho(s,a)/\tau$$
$$\pi_\rho(a|s) = \exp\left(Q_\rho(s,a) - V_\rho(s)\right)/\tau$$

Figure 29: Object ,Value and Policy Function of PCL

We can also add the idea of trust region, build the TPCL method.
In TPCL:
Combining the optimizing similar to TRPO and the conclusion in PCL

$$V^*(s_0) = \mathbb{E}_{r_i,a_i,s_i}[\gamma^d V^*(s_d) + \sum_{i=0}^{d-1} \gamma^i (r_i - (\tau+\lambda)\log\pi^*(a_i|s_i) + \lambda \log\tilde{\pi}(s_i|s_i))]$$

Figure 30: softmax temporal consistency constraint

TPCL is easier to implement than TRPO,it only needs simple gradient descent. **GAE**
Generalized advantage estimator [10]
According to the disadvantages of PG: too high variance. GAE is mostly discussing reduce variance under the premise o f controlling bias.

$$g = \mathbb{E}[\sum_{t=0}^{\infty} \Psi_t \nabla_\theta \pi_\theta(a_t|s_t)]$$

Figure 31: The gradient in PG

In this model,γ in discounted MDP is the parameter to reduce variance.Another parameter is similar to λ in TD(λ) ,used in weight of different k-step A function. Considering these two parameters,evaluating the advantage function is the weighed average index of different k step evaluations.

$$\hat{A}_t^{GAE(\gamma,\lambda)} = \sum_{l=0}^{\infty} (\gamma\lambda)^l \delta_{t+l}^V$$

Figure 32: evaluating the advantage function

ACER
Actor critic with experience replay [11]

$$\min_z \frac{1}{2}||\hat{g}_t^{\text{acer}} - z||_2^2$$
$$s.t. \quad \nabla_{\phi_\theta} D_{KL}[f(\cdot|\phi_{\theta_a}(x_t))||f(\cdot|\phi_\theta(x_t))]^T z \leq \delta$$

Figure 33: After including the trust region,the constraint problem

$$z^* = \hat{g}_t^{\text{acer}} - \max\{0, \frac{k^T \hat{g}_t^{\text{acer}} - \delta}{||k||_2^2}\}k$$

Figure 34: Notice that it is a linear constraint can be solved as quadratic programming problem

Comparison between ACER and TRPO: In TRPO, we use conjugate gradient method which includes a lot of Fisher-vector products calculation. To get better performance in large scales of data, ACER maintains an average policy network to represent the slip averaging of former policy instead of constraints of policy updating in TRPO.

3.3.4 Results Display

This part demonstrates the best results of the three different environments in quanser robots, namely, Qube, CartpoleSwing and BallBalancer. For each part, we will report the hyper-parameters we use in the experiment along with the loss and reward curve with respect to episodes.

Qube-100-v0
For this platform, the best hyper-parameters is set as following, and the detailed meaning corresponding to each parameter is written in the form of "help":

```
parser.add_argument('--gamma', type=float, default=0.995, metavar='G',
                    help='discount factor (default: 0.995)')
parser.add_argument('--env-name', default="Qube-100-v0", metavar='G',
                    help='name of the environment to run')
parser.add_argument('--tau', type=float, default=0.97, metavar='G',
                    help='gae (default: 0.97)')
parser.add_argument('--l2-reg', type=float, default=1e-3, metavar='G',
                    help='l2 regularization regression (default: 1e-3)')
parser.add_argument('--max-kl', type=float, default=1e-2, metavar='G',
                    help='max kl value (default: 1e-2)')
parser.add_argument('--damping', type=float, default=1e-1, metavar='G',
                    help='damping (default: 1e-1)')
```

```
parser.add_argument('--seed', type=int, default=543, metavar='N',
                    help='random seed (default: 1)')
parser.add_argument('--batch-size', type=int, default=18000, metavar='N',
                    help='random seed (default: 1)')
```

Due to the time limit and the lack of computing resources (such as GPU), we only ran a few set of parameters. We ran 800 episodes for this environment, and the best result of loss and reward is given in Fig.35 respectively. We can see that the reward converges after around 300 episodes and the loss is negative, which means that the result perfectly fits.

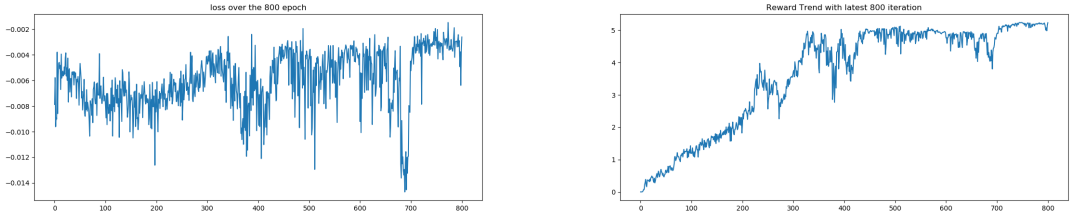


Figure 35: Loss and reward of Qube over 800 episodes.

The final result of the environment after rendering is shown in Fig.36 From the figure we can see the trajectory of the stick and its final state, i.e. holding straight up.

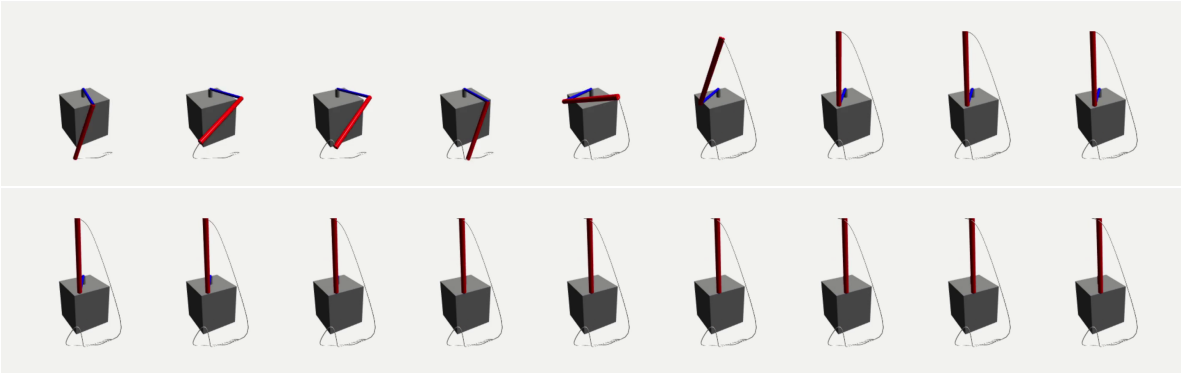


Figure 36: Movement of our Qube (reward = 5.09)

BallBalancerSim-v0

The hyper-parameters set for BallBalancer is $gamma = 0.995$ and batch size is to be 15000. From Fig.37, we can see that the reward converges after around 200 episodes.

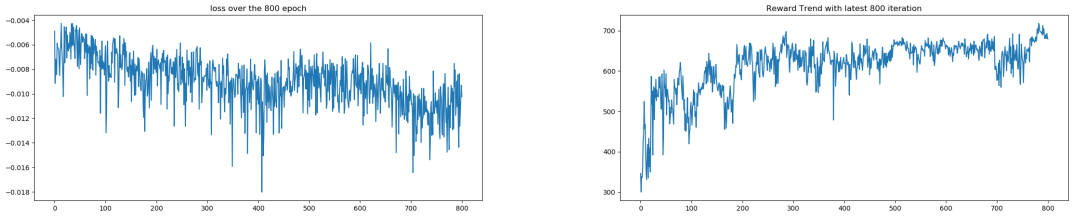


Figure 37: Loss and reward of BallBalancer over 800 episodes.

According to the video attached in the file, we will see that the in the testing phase, the ball moves straight from below, and eventually stops at the center of the box.

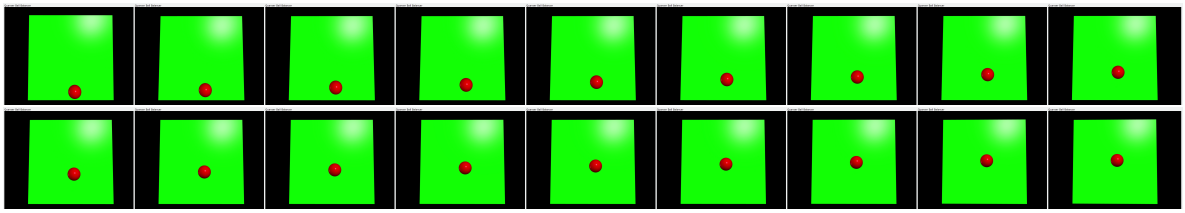


Figure 38: Movement of our BallBalancer (reward = 703.6)

CartpoleSwingShort-v0

The training loss and episode reward is shown in Fig.39. After seeing the reward trend, we infer that the rendered result may not be so good, since the reward varies drastically after going over 9000 at around 100 episodes. The reward seems not be truly converged.

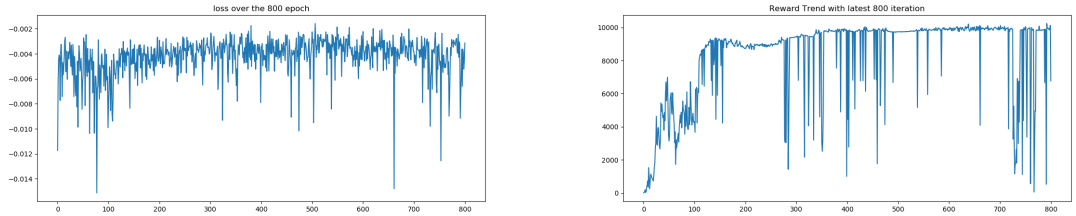


Figure 39: Loss and reward of CartpoleSwing over 800 episodes.

During the testing state, we rendered the environment to see what happens and our guess turned out to be correct. The stick cannot be hold quite steadily. Actually, the reward during the test state is far less than the training state. There are two reasons that may account for this phenomenon. One is that the samples are over-fitting during the train state, and the parameters trained is not suitable for test stage. The other is that the model structure is completely a failure.

Figure 40: Movement of our CartPoleSwing

3.3.5 Analysis and Conclusion

From the above experimental results, we can see that TRPO methods works for **Qube** and **BallBalancer**, but not for **CartpoleSwing**. Therefore we analyze the influences hyper-parameters brought to the performance using **Qube** and **BallBalancer** as an example. We mainly consider two factors, learning rate and batch size.

To be more specific, we set batch size to 12000, 15000, 18000 separately, and learning rate 0.99, 0.995, 0.998.

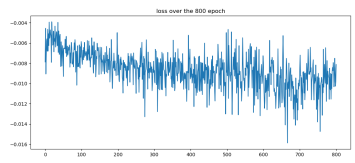
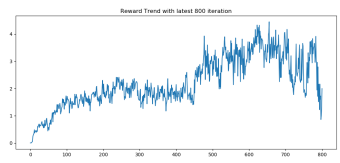
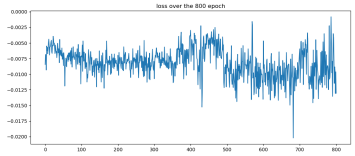
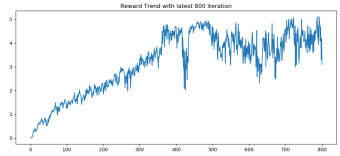
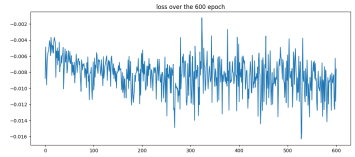
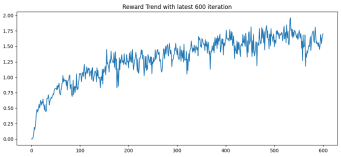
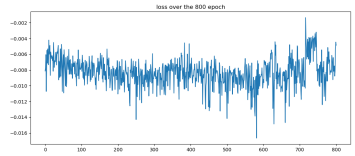
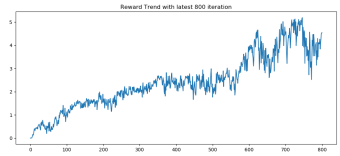
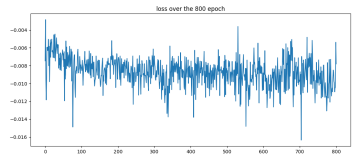
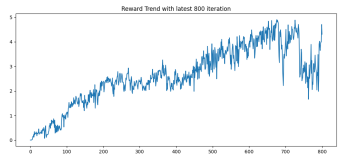
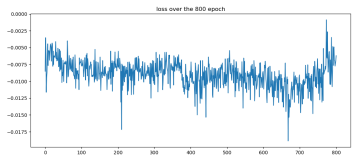
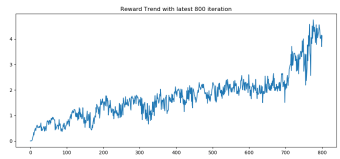
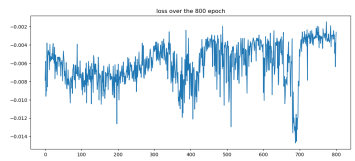
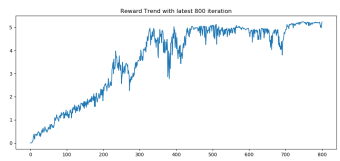
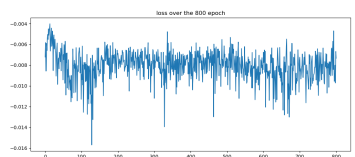
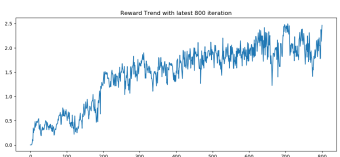
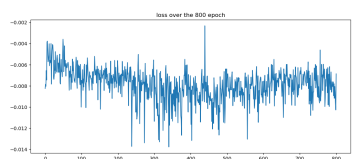
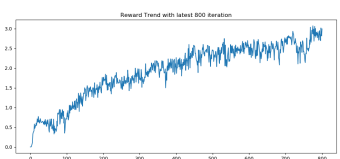
Batch Size	Learning Rate	Loss	Reward
15000	0.99		
15000	0.995		
15000	0.998		
12000	0.995		
12000	0.99		
12000	0.998		
18000	0.995		
18000	0.99		
18000	0.998		

Table 9: Different hyper-parameters and the result of Qube.

From the above table, we can see that under the same batch size, as the learning rate decreases, the convergence speed certainly accelerates. And with the same learning rate, the larger the batch size is, the faster the convergence speed is.
However, we notice that in the middle part of this table, the reward plunges sooner or later during training. The reason certainly has to do with the over small learning rate, and by adjusting batch size, we can make up for this issue.
As we decrease the learning rate, though by a little, the influence is far-reaching. Evidently, from the figure above we can see that the reward seems not convergent under the same condition, i.e. batch size, episode etc, compared with previous several experiments. We conduce this phenomenon to a much lower speed in gradient decent.

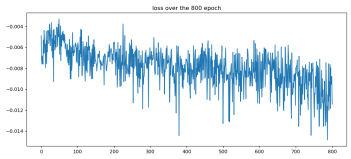
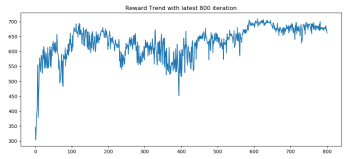
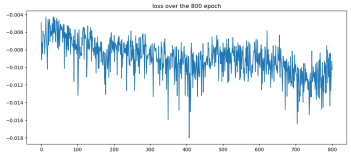
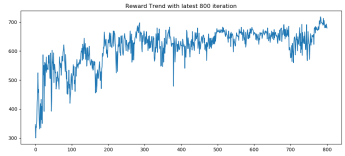
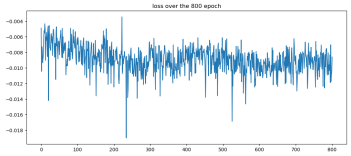
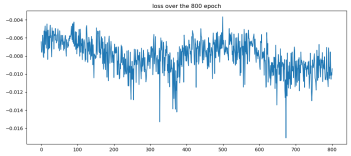
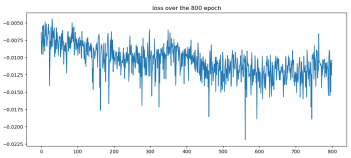
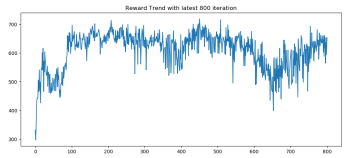
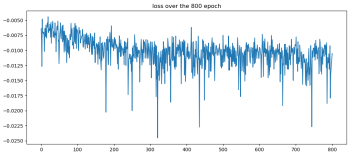
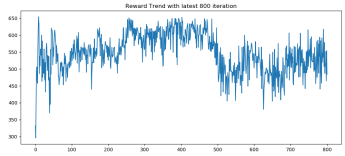
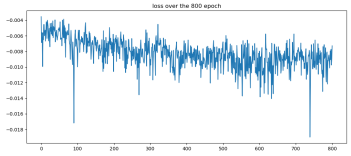
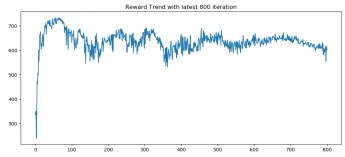
Batch Size	Learning Rate	Loss	Reward
15000	0.99		
15000	0.995		
15000	0.998		
12000	0.99		
12000	0.995		
12000	0.998		
18000	0.995		

Table 10: Different hyper-parameters and the result of BallBalancer.

From the table of BallBalancer, the loss of all the parameters generally forms a decreasing trend. When batch size is equal to 12000, the reward is likely to drop a lot at around 500 or 600 episodes. The reason may lies in that the batch size is so small that the result is likely to wave for a long time. In the meantime, the reward of batch size 18000 keeps decreasing after reaching some summit point. We guess that it is because of the over-fitting phenomenon due to the large batch size.

The adjustment of hyper-parameters is very tricky. If the batch size is set to be small, the converging speed may be very slow and it is difficult to converge. Once set too large, the network might converge to some local optimizations. Learning rate is more or less similar. If set too large, the model may skip the optimizations and can never converge. On the other hand, if it is too small, the convergence speed will be too slow to tolerate.

3.4 Comparison between MPC and TRPO

We start our analysis and comparison of the above two methods with respect to different methods. The following are based on the best hyper-parameters of each method.

1. **Qube-100-v0**

   Both MPC and TRPO performs well in this platform. But TRPO takes **less time** than MPC (especially when MPC use many bees in ABC optimizer), and g**et a better reward** (greater than 5). The TRPO reward curve has a monotonous increasing trend, while the MPC reward curve has a big fluctuation. Sometimes, the MPC reward is less than 1, which is a great difference between MPC and TRPO.

2. **BallBalancerSim-v0**

   In this platform, TRPO succeeded to move the ball to the center. Though MPC algorithm has a trend of moving ball to the center, an iteration often ends before the ball moves to the middle. The best reward of MPC is 600+, while TRPO can get reward 700+. But due to the TRPO's speed advantage, its rendering process is more pleasant.

3. **CartpoleSwintShort-v0**
   In our settlements, both TRPO and MPC are not able to handle this platform very well. As demonstrated in the video, model trained by TRPO cannot hold the stick steadily as it swings up. One comfort is that it learns to swing up. It has a similar case with MPC in that the stick hangs from left to right, but cannot hold up right either. The reward of TRPO (over 9000) is drastically higher than MPC, which is around 100.

From the analysis above, we infer that TRPO is obviously better than MPC under our limited computing sources.
However, in TRPO, it uses KL-divergence as an restriction for the new distribution and the old. However, there are two main problems with KL-divergence. The first one is that KL-divergence is very hard to calculate. The second one is that KL-divergence relies heavily on symmetry, which can cause very large variance sometimes.

References

[1] Christopher Watkins. Learning from delayed rewards. 01 1989.

[2] Anusha Nagabandi, Gregory Kahn, Ronald S Fearing, and Sergey Levine. Neural network dynamics for model-based deep reinforcement learning with model-free fine-tuning. In *2018 IEEE International Conference on Robotics and Automation (ICRA)*, pages 7559–7566. IEEE, 2018.

[3] Michael Steinbrunn, Guido Moerkotte, and Alfons Kemper. Heuristic and randomized optimization for the join ordering problem. *Vldb Journal*, 6(3):191–208, 1997.

[4] D. Karaboga. An idea based on honey bee swarm for numerical optimization. 2005.

[5] John Schulman, Sergey Levine, Pieter Abbeel, Michael I. Jordan, and Philipp Moritz. Trust region policy optimization. In Francis R. Bach and David M. Blei, editors, *Proceedings of the 32nd International Conference on Machine Learning, ICML 2015, Lille, France, 6-11 July 2015*, volume 37 of *JMLR Workshop and Conference Proceedings*, pages 1889–1897. JMLR.org, 2015.

[6] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *CoRR*, abs/1707.06347, 2017.

[7] Nicolas Heess, Dhruva TB, Srinivasan Sriram, Jay Lemmon, Josh Merel, Greg Wayne, Yuval Tassa, Tom Erez, Ziyu Wang, S. M. Ali Eslami, Martin A. Riedmiller, and David Silver. Emergence of locomotion behaviours in rich environments. *CoRR*, abs/1707.02286, 2017.

[8] Ofir Nachum, Mohammad Norouzi, Kelvin Xu, and Dale Schuurmans. Bridging the gap between value and policy based reinforcement learning. In Isabelle Guyon, Ulrike von Luxburg, Samy Bengio, Hanna M. Wallach, Rob Fergus, S. V. N. Vishwanathan, and Roman Garnett, editors, *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, 4-9 December 2017, Long Beach, CA, USA*, pages 2775–2785, 2017.

[9] Ofir Nachum, Mohammad Norouzi, Kelvin Xu, and Dale Schuurmans. Trust-pcl: An off-policy trust region method for continuous control. In *6th International Conference on Learning Representations, ICLR 2018, Vancouver, BC, Canada, April 30 - May 3, 2018, Conference Track Proceedings*. OpenReview.net, 2018.

[10] John Schulman, Philipp Moritz, Sergey Levine, Michael I. Jordan, and Pieter Abbeel. High-dimensional continuous control using generalized advantage estimation. In Yoshua Bengio and Yann LeCun, editors, *4th International Conference on Learning Representations, ICLR 2016, San Juan, Puerto Rico, May 2-4, 2016, Conference Track Proceedings*, 2016.

[11] Ziyu Wang, Victor Bapst, Nicolas Heess, Volodymyr Mnih, Rémi Munos, Koray Kavukcuoglu, and Nando de Freitas. Sample efficient actor-critic with experience replay. In *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings*. OpenReview.net, 2017.