BE PROJECT REPORT

REPORT OF PROJECT TWO FOR EI339

Reinforcement Learning in Quanser Robot platform

Gong Chen
Yang Zhaojing
Su Haodong
Department of Electronics and Telecommunications
SHANGHAI JIAO TONG UNIVERSITY

Contents

1 Introduction to the Reinforcement Learning Environment of Quanser Robots

In this part, we will firstly give a detail introduction to the three reinforcement environment (Qube, BallBalancer and CartpoleSwingShort), including their **physical models**, **equations**, **state space**, **observation space**, **action space** and **episode reward**. Then, we will briefly show the reinforcement learning methods we adopt, including **TRPO**(Trust Region Policy Optimization) and **MPC**(Model Predictive Control).

1.1 Qube-100-v0

- **Physical Model**

  The physical model of Qube environment is shown in figure 1. The rotary arm pivot is attached to the QUBE-Servo 2 system and is actuated. The arm has a length of $\boldsymbol{L_r}$, a moment of inertia of J_r, and its angle $\boldsymbol{\theta}$ increases positively when it rotates counter-clockwise. The pole should turn in the counter-clockwise direction when the control voltage is positive ($\boldsymbol{V_m} > 0$). The pendulum link is connected to the end of the rotary arm. It has a total length of $\boldsymbol{L_p}$ and its center of mass is at $\frac{\boldsymbol{L_p}}{\boldsymbol{2}}$. The moment of inertia about its center of mass is $\boldsymbol{J_p}$. The inverted pendulum angle $\boldsymbol{\alpha}$ is zero when it is hanging downward and increases positively when rotated counter-clockwise.

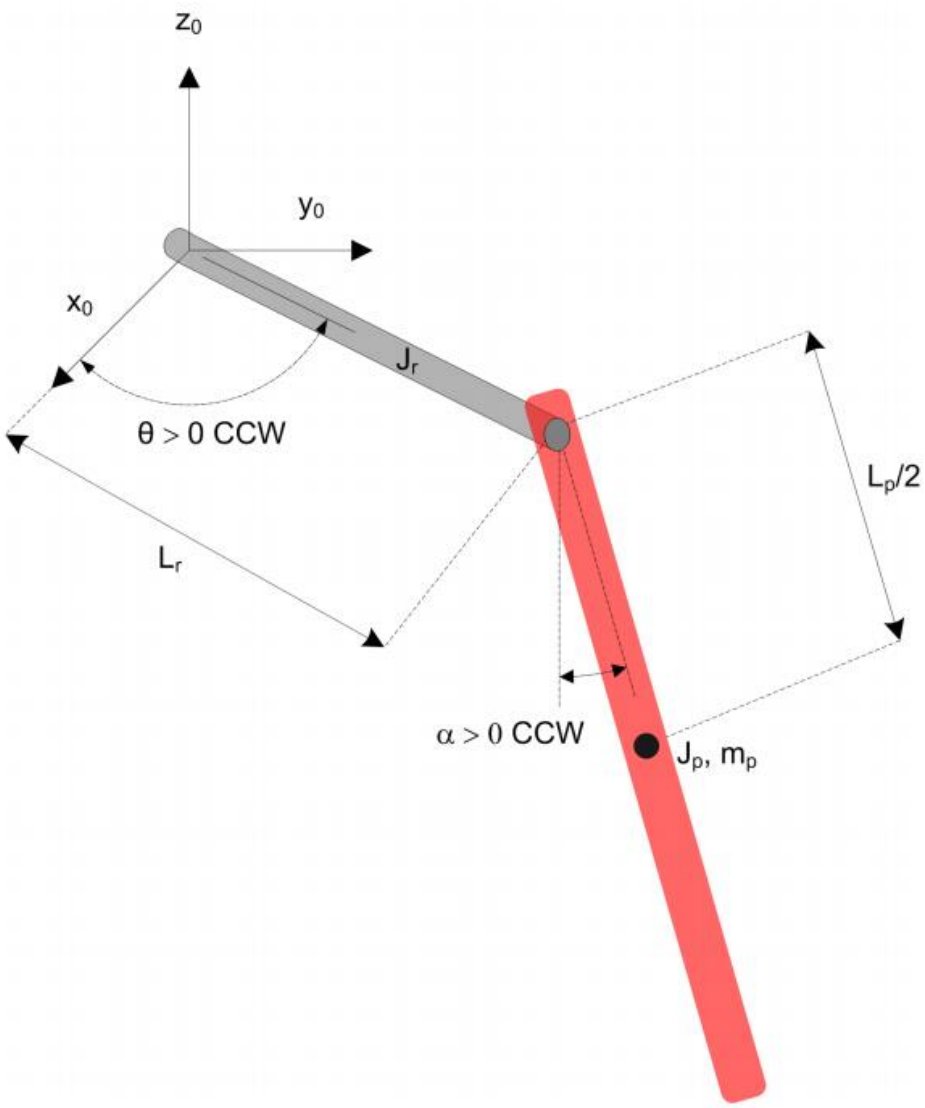

Figure 1.1: Rotary inverted pendulum model

Figure 1: model: Qube

- **Equations of Motion**

$$\begin{cases} [m_p L_r^2 + \frac{1}{4} m_p L_p^2 - \frac{1}{4} m_p L_p^2 \cos(\alpha)^2 + J_r]\ddot{\theta} + [\frac{1}{2} m_p L_p L_r \cos(\alpha)]\ddot{\alpha} \\ +[\frac{1}{2} m_p L_p^2 \sin(\alpha)\cos(\alpha)]\dot{\theta}\dot{\alpha} - [\frac{1}{2} m_l L_p L_r \sin(\alpha)]\dot{\alpha}^2 = \tau - D_r\dot{\theta} \\ \frac{1}{2} m_p L_p L_r \cos(\alpha)\ddot{\theta} + (J_p + \frac{1}{4} m_p L_p^2)\ddot{\alpha} - \frac{1}{4} m_p L_p^2 \cos(\alpha \sin(\alpha))\ddot{\theta}^2 + \frac{1}{2} m_p L_p g \sin(\alpha) = -D_p\dot{\alpha} \\ \tau = \frac{k_m(V_m - k_m\dot{\theta})}{R_m} \end{cases} \tag{1}$$

- **Goal**

  The goal of the Qube environment is to swing the pendulum and keep it vertical for as long time as it could.

- **State Space**

  To describe a state of the model briefly, at least four parameters(can be seen more vividly through figure 1) are needed: θ and α to describe the position of the objective; $\dot{\theta}$ and $\dot{\alpha}$ to describe the trend of the objective. The range of the parameters can be seen in table 1

θ	α	$\dot{\theta}$	$\dot{\alpha}$
$-2 \sim 2$	$-4\pi \sim 4\pi$	$-30.0 \sim 30.0$	$40.0 \sim 40.0$

Table 1: Qube: State Space

- **Observation Space**

  To simplify the calculating process, when observing the objective, it returns $\sin(\alpha)$, $\cos(\alpha)$, $\sin(\theta)$ and $\cos(\theta)$ instead of returning θ and α directly. As a result, we need 6 parameters. The range of these 6 parameters can be seen in table 2.

$\sin(\theta)$	$\cos(\theta)$	$\sin(\alpha)$	$\cos(\alpha)$	$\dot{\theta}$	$\dot{\alpha}$
$-1 \sim 1$	$-1 \sim 1$	$-1 \sim 1$	$-1 \sim 1$	$-30.0 \sim 30.0$	$40.0 \sim 40.0$

Table 2: Qube: Observation Space

- **Action Space**

  The only action we can do is to control the voltage V_m which is related to the acceleration of θ and α directly. When $V_m > 0$, the servo and arm should turn in the counter-clockwise direction but otherwise when $V_m < 0$. The range of action space V_m is $-5 \sim 5$.

- **Episode Reward**

  In this environment, we firstly define the cost and then use the negative cost to denote rewards we get in one step.

$$cost = [(\alpha \mid 2\pi) - \pi]^2 + 0.005\dot{\alpha}^2 + 0.1\theta^2 + 0.02\dot{\theta}^2 + 0.003 \tag{2}$$

1.2 BallBalancerSim-v0

- **Physical Model**

  The free body diagram of the Ball and Beam is illustrated directly in figure 2. We need to apply a positive voltage causing the servo load gear to move in the positive, counter-clockwise (CCW) direction. This moves the beam upwards and causes the ball to roll in the positive direction which means $V_M > 0 \rightarrow \dot{\theta}_l > 0 \rightarrow \dot{x} > 0$. If the ball wants to be stationary at a certain point , the force from the ball's momentum must be equal to the force produced by the gravity. The mathematical expression of the motion of the ball can be seen in part Equations of Motion.

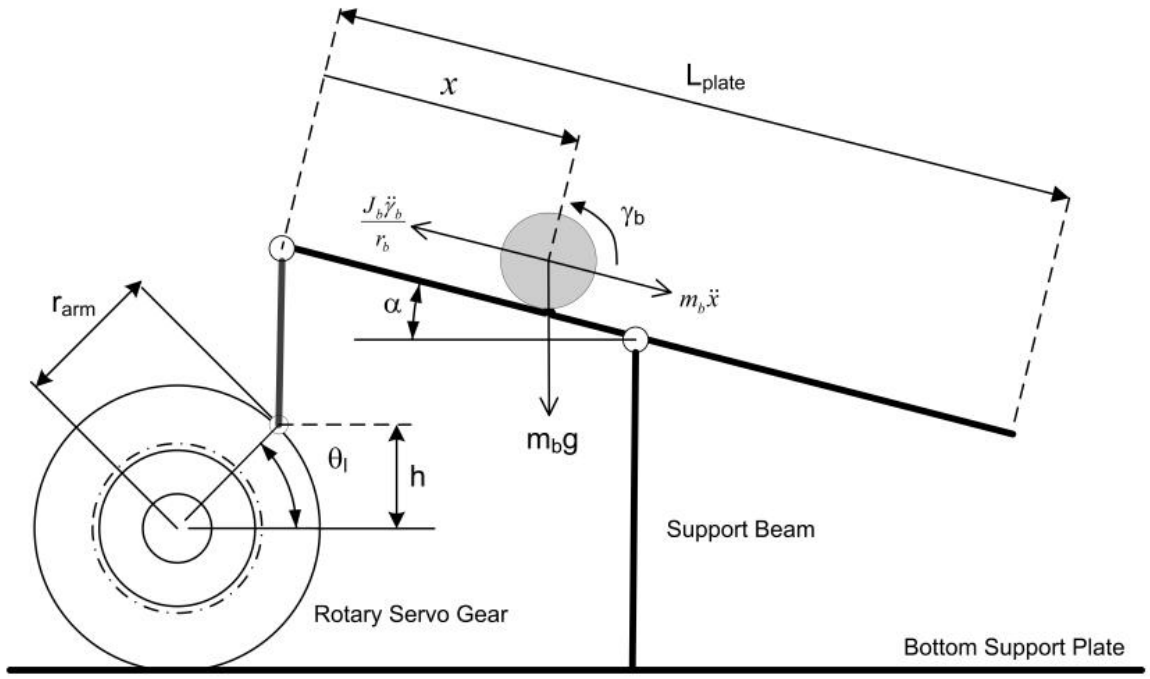

Figure 2.1: Modeling ball on plate in one dimension.

Figure 2: model: BallBalancer

The complete open-loop system is represented by the block diagram shown in figure 3. The transfer function $\boldsymbol{P_s(s)}$ represents the dynamics between the servo input motor voltage and the resulting load angle. The transfer function $\boldsymbol{P_{bb}(s)}$ describes the dynamics between the angle of the servo load gear and the position of the ball. And the model can be decoupled to x-axis and y-axis.

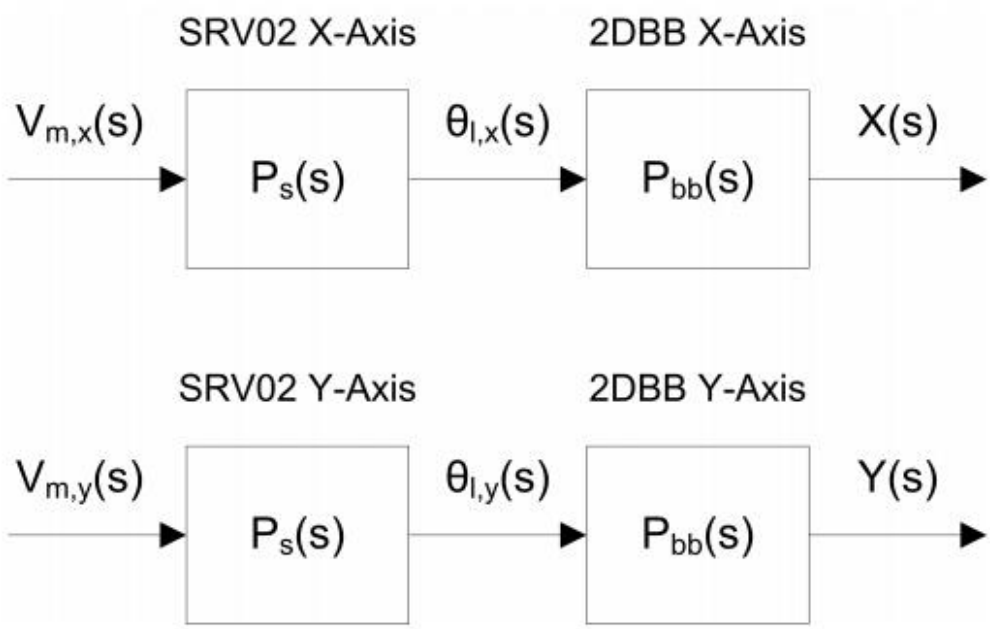

Figure 3: model: BallBalancer

The corresponding equations can be seen as follows:

$$\begin{cases} P_{bb}(s) = \frac{X(s)}{\Theta_l(s)} = \frac{K_{bb}}{s^2} \\ P_s(s) = \frac{\Theta_l(s)}{V_m(s)} = \frac{K}{s(\tau s+1)} \end{cases} \Rightarrow P(s) = P_{bb}(s)P_s(s) = \frac{K_{bb}K}{s^3(\tau s+1)} \tag{3}$$

where $P_{bb}(s)$ is the servo angle to ball position transfer function and $P_s(s)$ is the voltage to servo angle transfer function. The nominal model parameters of SRV02 are $K = 1.53rad/(V-s)$ and $\tau = 0.0248s$.

- **Equations of Motion**
  Based on Newton's First Law of Motion, the sum of forces acting on the ball along the beam equals:

$$\begin{cases} m_b\ddot{x}(t) & = \sum F = F_{x,t} - F_{x,r} \\ F_{x,t} & = m_b g \sin\alpha(t) \\ F_{x,r} & = \frac{\tau_b}{r_b} = \frac{J_b\ddot{\gamma}_b(t)}{r_b} = \frac{J_b\ddot{x}(t)}{r_b^2} \end{cases} \Rightarrow \ddot{x}(t) = \frac{m_b g \sin\alpha(t) r_b^2}{m_b r_b^2 + J_b} \tag{4}$$

where $\boldsymbol{m_b}$ is the mass of the ball, $\boldsymbol{x}$ is the ball displacement, $\boldsymbol{F_{x,r}}$ is the force from the ball's inertia, $\boldsymbol{F_{x,t}}$ is the translation force generated by gravity, $\boldsymbol{r_b}$ is the radius of the

ball and γ_b is the ball angle. Friction and viscous damping are neglected.
Using the schematic given in Figure **??**, we can find the equation of motion that represents the ball's motion with respect to the servo angle θ_l:

$$\begin{cases} \sin\alpha(t) = \frac{2h}{L_{plate}} \\ \sin\theta_l(t) = \frac{h}{r_{arm}} \end{cases} \Rightarrow \sin\alpha(t) = \frac{2r_{arm}\sin\theta_l(t)}{L_{plate}}$$
$$\Rightarrow \ddot{x}(t) = \frac{2m_b g r_{arm} r_b^2}{L_{plate}(m_b r_b^2 + J_b)} \sin\theta_l(t) \approx \frac{2m_b g r_{arm} r_b^2}{L_{plate}(m_b r_b^2 + J_b)} \theta_l(t) \tag{5}$$

- **Goal**
  The training goal of this reinforcement environment is to keep the ball stable in the center and not exceed the range which is just the range of the stating space.

- **State Space**
  To describe a state of the model specifically, at least eight parameters(can be seen more clearly and vividly through figure 2) are needed: θ_x and θ_y to describe the angle θ_l between beam and ground, pos_x and pos_y to describe the position of the ball in relation to the beam, $\dot{\theta}_x$, $\dot{\theta}_y$, $\dot{pos}_x$, $\dot{pos}_y$ to describe the motion trends of the ball and beam. The range of the parameters can be seen in table 3.

θ_x	θ_y	pos_x	pos_y	$\dot{\theta}_x$	$\dot{\theta}_y$	$\dot{pos}_x$	$\dot{pos}_y$
$-\frac{\pi}{4} \sim \frac{\pi}{4}$	$-\frac{\pi}{4} \sim \frac{\pi}{4}$	$-0.15 \sim 0.15$	$-0.15 \sim 0.15$	$-4\pi \sim 4\pi$	$-4\pi \sim 4\pi$	$-0.5 \sim 0.5$	$-0.5 \sim 0.5$

Table 3: BallBalancer: State Space

- **Observation Space**
  Observation space is used to calculate the reward and action directly and quickly. In this environment, the observation space is different from the state space. We need to obtain the measurement of the state which is related to $\theta_x, \theta_y, pos_x, pos_y$ and calculate the derivative of the measurement instead of directly using $\dot{\theta}_x, \dot{\theta}_y, \dot{pos}_x, \dot{pos}_y$. To reduce the calculating cost, the method we adopt is to change the form of measurement from continuous variable to discrete variable and then use a filter to obtain the derivative. We still use $\theta_x, \theta_y, pos_x, pos_y$ to represent the measurement and $\dot{\theta}_x, \dot{\theta}_y, \dot{pos}_x, \dot{pos}_y$ to represent its derivative. The range of these parameters can be seen in table 4.

θ_x	θ_y	pos_x	pos_y	$\dot{\theta}_x$	$\dot{\theta}_y$	$\dot{pos}_x$	$\dot{pos}_y$
$-\frac{\pi}{4} \sim \frac{\pi}{4}$	$-\frac{\pi}{4} \sim \frac{\pi}{4}$	$-0.15 \sim 0.15$	$-0.15 \sim 0.15$	$-4\pi \sim 4\pi$	$-4\pi \sim 4\pi$	$-0.5 \sim 0.5$	$-0.5 \sim 0.5$

Table 4: BallBalancer: Observation Space

- **Action Space**

  The action we can take is controlling the servo input motor voltage $\boldsymbol{V_m}$ which is related to the angle of the servo load gear and thus the position of the ball. The voltage V_m can be decomposed to $\boldsymbol{V_{m,x}}$ along the x-axis and $\boldsymbol{V_{m,y}}$ along the y-axis. The range the action space can be seen if table 5.

$V_{m,x}$	$V_{m,y}$
$-50. \sim 5.0$	$-5.0 \sim 5.0$

Table 5: BallBalancer: Action Space

- **Episode Reward**

  Firstly, we calculate the cost at this time which is related to observation (θ_x, θ_y, pos_x, pos_y, $\dot{\theta}_x$, $\dot{\theta}_y$, $\dot{pos}_x$, $\dot{pos}_y$) and goal state (θ_x^*, θ_y^*, pos_x^*, pos_y^*, $\dot{\theta}_x^*$, $\dot{\theta}_y^*$, $\dot{pos}_x^*$, $\dot{pos}_y^*$) as well as our action ($V_{m,x}, V_{m,y}$):

$$\begin{aligned} Cost = {} & 0.01(\theta_x - \theta_x^*) + 0.01(\theta_y - \theta_y^*) + (pos_x - pos_x^*) + (pos_y - pos_y^*) \\ & + 0.0001(\dot{\theta}_x - \dot{\theta}_x^*) + 0.0001(\dot{\theta}_y - \dot{\theta}_y^*) + 0.01(\dot{pos}_x - \dot{pos}_x^*) + 0.01(\dot{pos}_y - \dot{pos}_y^*) \end{aligned} \tag{6}$$

  Then we normalize the cost by dividing the maximum value and take its negative as the reward of one step in the episode.

1.3 CartpoleSwingShort-v0

- **Physical Model**

  The linear Single Pendulum Gantry (SPG) model is shown in Figure 4. The centre of mass of the pendulum is at length $\boldsymbol{l_p}$ and the moment of inertia about the centre is $\boldsymbol{J_p}$. The pendulum angle $\boldsymbol{\alpha}$ 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 $\boldsymbol{x_c}$ is to the right when facing the cart. The position of the pendulum centre of gravity is denoted as the $\boldsymbol{(x_p, y_p)}$ coordinate.

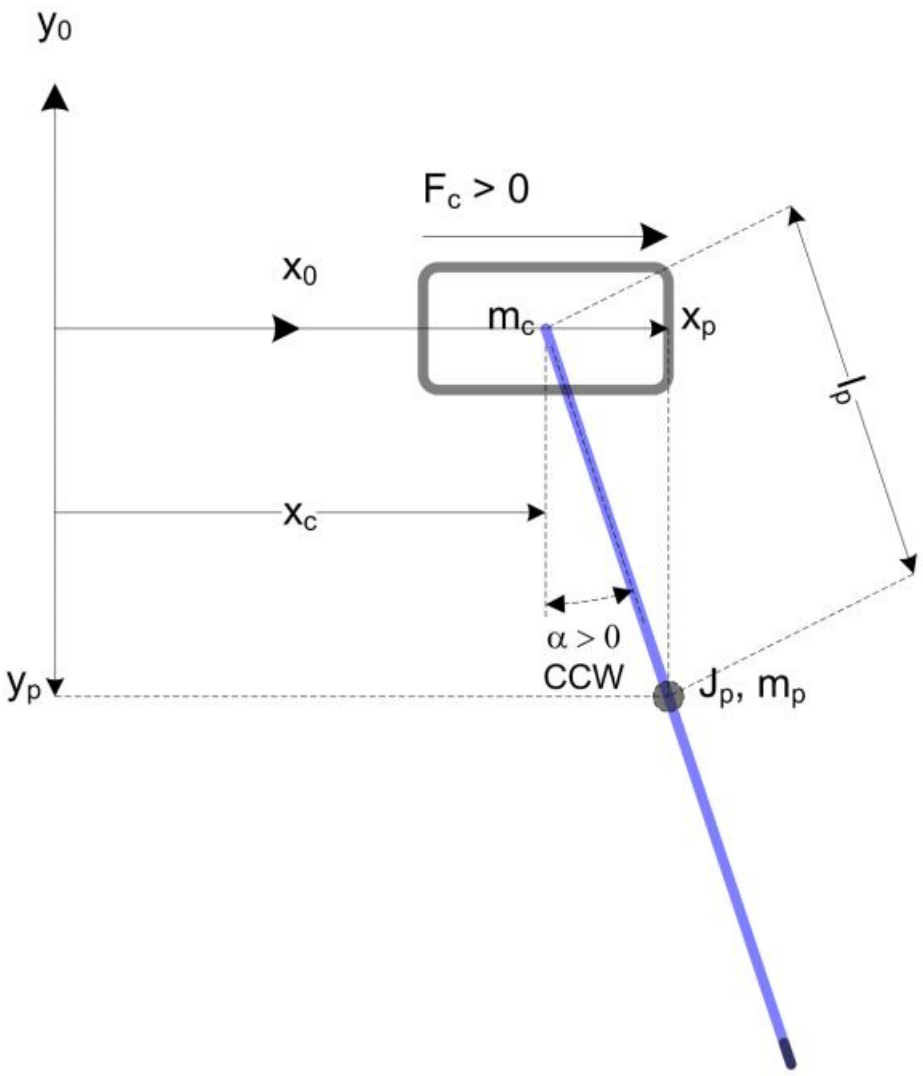

Figure 4: model: CartpoleSwingShort

- **Equations of Motion**
  The Lagrangian of the system L is described as

$$L = T - V \tag{7}$$

  where $\boldsymbol{T}$ is the total kinetic energy of the system and $\boldsymbol{V}$ is the total potential energy of

the system. Then we separately induce the expressions of T and V:

$$
\begin{aligned}
&V = V_{pendulum} = -M_p g l_p \cos(\alpha) \\
&T = T_{cart} + T_{pendulum} \\
&\begin{cases}
T_{cart} & = T_{kinetic\ energy} + T_{rotational\ energy} \\
& = \frac{1}{2} M \dot{x}_c^2 + \frac{\eta_g J_m K_g^2 \dot{x}_c^2}{2 r_{mp}^2} = \frac{1}{2} J_{eq} {\dot{x}_c}^2 \\
T_{pendulum} & = T_{kinetic\ energy} + T_{rotational\ energy} \\
& = \frac{1}{2} M_p \sqrt{{\dot{x}_p}^2 + {\dot{y}_p}^2} + \frac{1}{2} J_p \dot{\alpha}^2 \\
\dot{x_p} & = \dot{x}_c + l_p \cos(\alpha) \dot{\alpha} \\
\dot{y_p} & = l_p \sin(\alpha) \dot{\alpha}
\end{cases} \\
&\Rightarrow T = \frac{1}{2}(J_{eq} + M_p) {\dot{x}_c}^2 + M_p l_p \cos(\alpha) \dot{\alpha} \dot{x}_c + \frac{1}{2}(J_p + M_p l_p^2) \dot{\alpha}^2
\end{aligned}
\tag{8}
$$

The generalized forces $\boldsymbol{Q_{x_c}}$ and $\boldsymbol{Q_{\alpha}}$ are used to describe the non-conservative forces acting on the liner cart and acting on the pendulum which can be expanded in details:

$$
\begin{cases}
Q_{x_c} & = F_c - B_{eq} \dot{x}_c \\
Q_{\alpha} & = -B_p \dot{\alpha}
\end{cases}
\tag{9}
$$

According to the energy transformation and equation (8) and (9), we obtain the equations of motion:

$$
\begin{aligned}
&(J_{eq} + M_p) \ddot{x}_c + M_p l_p \cos(\alpha) \ddot{\alpha} - M_p l_p \sin(\alpha) \dot{\alpha}^2 = F_c - B_{eq} \ddot{x}_c \\
&M_p l_p \cos(\alpha) \ddot{x}_c + (J_p + M_p l_p^2) \ddot{\alpha} + M_p l_p g \sin(\alpha) = -B_p \dot{\alpha}
\end{aligned}
\tag{10}
$$

where F_c is generated by the servo motor and can be described as:

$$
F_c = (\frac{\eta_g K_g K_t}{R_m r_{mp}})(-\frac{K_g K_m \dot{x}_c}{r_{mp}} + \eta_m V_m)
\tag{11}
$$

- **Goal**

  The goal of this reinforcement learning environment is to train the model to learn how to swing the pole and keep it vertical as long time as it could.

- **State Space**

  To describe a state of the system, at least four parameters are needed(can be seen more clearly and vividly through figure 4): $\boldsymbol{x}$ to describe cart position, $\boldsymbol{\alpha}$ to describe

pole angle, $\dot{x}$ to describe cart velocity and $\dot{\alpha}$ to describe pole velocity at tip. The range of the parameters are provided through table 6.

x	α	$\dot{x}$	$\dot{\alpha}$
$-2.4 \sim 2.4$	$-41.8° \sin 41.8°$	$-\infty \sim \infty$	$-\infty \sim \infty$

Table 6: CartpoleSwingshort: State Space

- **Observation Space**
  The observation space is used to calculate reward and action quickly and briefly. Thus, it returns $\cos(\alpha)$ and $\sin(\alpha)$ instead of returning α directly. The range of the five parameters is provided through table 7.

x	$\cos(\alpha)$	$\sin(\theta)$	$\dot{x}$	$\dot{\alpha}$
$-2.4 \sim 2.4$	$-1 \sim 1$	$-1 \sim 1$	$-\infty \sim \infty$	$-\infty \sim \infty$

Table 7: CartpoleSwingshort: Observation Space

- **Action Space**
  The only action we can do is to control the voltage V_c imposed on the cart whose range is $-24.0 \sim 24.0$.

- **Episode Reward**
  The reward given in the code is related to α which is the angle of the pendulum. The reward of one step is:

$$reward = -\cos(\alpha) \tag{12}$$

2 TRPO: Trust Region Policy Optimization

2.1 Introduction

- **Summary**
  TRPO method is based on the prove that minimizing a certain surrogate objective function guarantees policy improvement with non-trivial steps. This means that the model will keep becoming better and will not degenerate anyway. However, when implementing the practical algorithm, we use sampling method to estimate a distribution which has errors and may impose negative influence on the model.

- **Several Important Definitions**

  Let π denote a policy, then its expected reward $\eta(\pi)$ is :

$$\begin{aligned} \eta(\pi) &= \mathbb{E}_{s_o,a_0,\ldots}[\sum_{t=0}^{\infty}\gamma^t r(s_t)] \\ s_0 \sim \rho_0(s_0), a_t \sim \pi(a_t|s_t), s_{t+1} &\sim P(s_{t+1}|s_t,a_t) \end{aligned} \tag{13}$$

  The action value function $Q_\pi(s_t,a_t)$(expected rewards of choosing action a_t at state s_t), value function $V_\pi(s_t)$(expected reward of state s_t) and advantage A_π(expected reward difference between state s_t and choosing action a_t at state s_t) are defined as:

$$\begin{aligned} Q_\pi(s_t,a_t) &= \mathbb{E}_{s_{t+1},a_{t+1},\ldots}[\sum_{l=0}^{\infty}\gamma^l r(s_t+1)] \\ V_\pi(s_t) &= \mathbb{E}_{a_t,s_{t+1},\ldots}[\sum_{l=0}^{\infty}\gamma^l r(s_t+1)] \\ A_\pi(s,a) &= Q_\pi(s,a) - V_\pi(s) \end{aligned} \tag{14}$$

- **How to Improve the Current Policy π to $\tilde{\pi}$**

  Firstly, we have the following equation which has been proved:

$$\begin{aligned} \eta(\tilde{\pi}) &= \eta(\pi) + \mathbb{E}_{s_0,a_0,\cdots\sim\tilde{\pi}}[\sum_{l=0}^{\infty}A_\pi(s_t,a_t)] \\ &= \eta(\pi) + \sum_s \rho_{\tilde{\pi}}(s)\sum_a \tilde{\pi}(a|s)A_\pi(s,a) \\ where\ \rho_\pi(s) &= P(s_0=s) + \gamma P(s_1=s) + \gamma^2 P(s_2=s) + \ldots \end{aligned} \tag{15}$$

  But the complexity dependency of $\rho_{\tilde{\pi}}(s)$ on $\tilde{\pi}$ makes it difficult to optimize directly, so we use the distribution of old policy π instead of the distribution of new policy $\tilde{\pi}$:

$$L_\pi(\tilde{\pi}) = \eta + \sum_s \rho_\pi(s)\sum_a \tilde{\pi}(a|s)A_\pi(s,a) \tag{16}$$

  Secondly, we use the importance sampling method to estimate distribution of new policy:

$$\begin{aligned} L_\pi(\tilde{\pi}) &= \eta(\pi) + \sum_s \rho_\pi(s)\mathbb{E}_{a,q}[\frac{\tilde{\pi}(a|s_n)}{q(a|s_n)}A_\pi(s_n,a)] \\ &= \eta(\pi) + \mathbb{E}_{s,\rho_\pi,a,\pi}[\frac{\tilde{\pi}(a|s)}{\pi(a|s)}A_\pi(s,a)] \end{aligned} \tag{17}$$

  Thirdly, we find that near $\pi(\theta_{old})$, we can improve both the surrogate function and the

original function. Thus, we need to decide the learning step.

- **How to Choose Uitable Steps**
  Because $\eta(\tilde{\pi}) \geq L_{\pi}(\tilde{\pi}) - CD_{KL}^{\max}(\pi, \tilde{\pi})$ proved by prior works, we obtain the lower bound of $\eta(\tilde{\pi})$:

$$\begin{aligned} M_i(\pi) &= L_{\pi_i}(\pi) - CD_{KL}^{\max}(\pi_i, \tilde{\pi}) \\ \Rightarrow \eta(\pi_{i+1}) - \eta(\pi_i) &\geq M_i(\pi_{i+1}) - M(\pi_i) \end{aligned} \tag{18}$$

  Thus, we need only to guarantee that we find a policy π_{i+1} that maximizes M_i and then $\eta(\pi_{i+1}) - \eta(\pi_i) \geq M_i(\pi_{i+1}) - M(\pi_i) \geq 0$. Finally, we can change the problem to an optimizing problem.

- **Form Solvable Optimizing Problem**
  If the step we take each time is small, the problem can be changed to (we use θ_{old} to denote the old policy π):

$$\begin{aligned} &maximize\, \mathbb{E}_{s,\rho_{\theta_{old}},a,\pi_{\theta_{old}}}[\frac{\pi_{\theta(a|s)}}{\pi_{\theta_{old}}} A_{\theta_{old}}(s,a)] \\ &subject\ to\ D_{KL}^{\max}(\theta_{old}, \theta) \leq \delta \end{aligned} \tag{19}$$

  All that remains is to replace the expectations by sample average and replace the $A_{\theta_{old}}$ by an empirical estimate.

- **Connection with Prior Works**
  We can use natural policy gradient to update equation 19 by using a linear approximation to L and a quadratic approximation to the D_{KL} constraint resulting in the following problem:

$$\begin{aligned} &maximize_{\theta}\ [\Delta L_{\theta_{old}}(\theta)|_{\theta=\theta_{old}} \cdot (\theta - \theta_{old})] \\ &subject\ to\ \frac{1}{2}(\theta_{old} - \theta)^T A(\theta_{old})(\theta_{old} - \theta) \leq \delta \end{aligned} \tag{20}$$

  This differs from the original approach. But when implementing the algorithm, we still use the natural gradient policy to update.

2.2 Implementation

We divide our implementation into four parts: constructing **neural network** to estimate state value and state policy, **collecting raw data** through interacting with environment, **calculating the advantage** A of each state and **updating the model**.

2.2.1 Neural Network to Estimate State Value and Obtain Policy

This class $network$ includes another two classes named $Value\_net$ and $Policy\_net$. $Value\_net$ is a 3-layer neural network which takes a tensor $state$ as input and returns its estimated value. $Policy\_net$ is also a **3-layer neural network** which takes a tensor $state$ as input and returns its policy distribution in the form of $(mean, variance)$. The codes can be seen as follows:

```
class network(nn.Module):
    def __init__(self, num_states, num_actions):
        super(network, self).__init__()

        self.Value_net = Value_net(num_states)
        self.Policy_net = Policy_net(num_states, num_actions)

    def forward(self, x):
        state_value = self.Value_net(x)
        pi = self.Policy_net(x)
        return state_value, pi

class Value_net(nn.Module):
    def __init__(self, num_states):
        super(Value_net, self).__init__()
        self.fc1 = nn.Linear(num_states, 64)
        self.fc2 = nn.Linear(64, 64)
        self.value = nn.Linear(64, 1)

    def forward(self, x):
        x = F.tanh(self.fc1(x))
        x = F.tanh(self.fc2(x))
        value = self.value(x)
        return value

class Policy_net(nn.Module):
    def __init__(self, num_states, num_actions):
        super(Policy_net, self).__init__()
        self.fc1 = nn.Linear(num_states, 64)
        self.fc2 = nn.Linear(64, 64)
        self.action_mean = nn.Linear(64, num_actions)
        self.sigma_log = nn.Parameter(torch.zeros(1, num_actions))

    def forward(self, x):
```

```
        x = F.tanh(self.fc1(x))
        x = F.tanh(self.fc2(x))
        mean = self.action_mean(x)
        sigma_log = self.sigma_log.expand_as(mean)
        sigma = torch.exp(sigma_log)
        pi = (mean, sigma)

        return pi
```

2.2.2 Collect Raw Training Data

We train the model for about 1000 episodes and there are 10000 steps in each episode. During each episode, we collect new 10000 data which are used to train the model. We collect the training data through interacting with the environment and record each step's observation, reward, action, done or not, estimated value through $Value\_net$. The codes can be seen as follows:

```
for update in range(num_updates):
    obs = self.running_state(self.env.reset())
    mb_obs, mb_rewards, mb_actions, mb_dones, mb_values = [], [], [], [], []
    for step in range(self.args.nsteps):
        with torch.no_grad():
            obs_tensor = self._get_tensors(obs)
            value, pi = self.net(obs_tensor) # state_value, (action_mean, action_std)
                # choose action according to sampling the normalization (action_mean,
                    action_std)
        actions = select_actions(pi)
        mb_obs.append(np.copy(obs))
        mb_actions.append(actions)
        mb_dones.append(self.dones)
        mb_values.append(value.detach().numpy().squeeze())

        # execute action and get new obs, reward, done
        obs_, reward, done, _ = self.env.step(actions)
        self.dones = done
        mb_rewards.append(reward)
        if done: # after executing the actions the state is done
            obs_ = self.env.reset()
        obs = self.running_state(obs_)
        episode_reward += reward #
```

```
        mask = 0.0 if done else 1.0
        final_reward *= mask # if done then final_reward = episode_reward else
            final_rewards unchanged
        final_reward += (1 - mask) * episode_reward
        episode_reward *= mask # if done then episode_reward=0 else episode_reward
            unchanged
```

2.2.3 Calculate Advantage of Each Data

We use the equation $A(s) = [R(s \rightarrow s') + \gamma V(s') - V(s)]$ to calculate a state's advantage without considering the future where R means reward from s to s', γ means discounting rate and $V(s)$ means the estimated value of the state s. We accumulate the advantages in the future by adding two discounting factors γ and τ to obtain the real advantage. The code can be seen as follows:

```
mb_advs = np.zeros_like(mb_rewards) # [0,0,...,0]
lastgaelam = 0
for t in reversed(range(self.args.nsteps)):
    if t == self.args.nsteps - 1:
        nextnonterminal = 1.0 - self.dones
        nextvalues = last_value
    else:
        nextnonterminal = 1.0 - mb_dones[t + 1]
        nextvalues = mb_values[t + 1]
    delta = mb_rewards[t] + self.args.gamma * nextvalues * nextnonterminal -
        mb_values[t] # A(s)=[R(s->s') + r*V(s') - V(s)]
    mb_advs[t] = lastgaelam = delta + self.args.gamma * self.args.tau *
        nextnonterminal * lastgaelam # advs[t] = A(t) + r*tau*advs[t+1]
mb_returns = mb_advs + mb_values # A(t)+V(t) = new_V(t)
mb_advs = (mb_advs - mb_advs.mean()) / (mb_advs.std() + 1e-5) # normalize advantages
```

2.2.4 Use Natural Gradient to Update Neural Network

In our TRPO algorithm,there are two kinds of loss: values loss and policy loss. The value loss is defined as the mean variance between values we estimated through neutral network and the values calculated by accumulating discounted rewards in the future. The policy loss is

calculated by the following equation:

$$policy\ loss = -\exp[\log(Policy_{new}) - \log(Policy_{old})] \times advantage \tag{21}$$

where $Policy_{new}$ and $Policy_{old}$ means the new policy distribution and the old policy distribution. **The natural gradient is used to calculate KL divergence and the expected improvement of our policy**. The **value loss** is used to optimize $Value\_net$ through **Adam optimizer**. And the **policy loss** is used to update $Policy\_net$ through natural gradient way based on the method taught in the introduction part of TRPO. The codes are so long that putting them here not only occupies space but also has less meaning. Thus, we attached the codes with our notes in our file.

2.3 Figures and Results Analyzing

In the first part, we will show the reward and policy loss curve against episode numbers in three Quanser Robot environments through figures and then we will analyze the curve theoretically combined with real situation.

In the second part, we will adjust some hyper-parameters like learning rate α, batch size, discounting factor γ, max KL divergence and τ. The figures and objective analyzing in details will also be shown clearly.

Learning rate α is used to optimize the Value Net model which takes state s as input and return the state's estimated value v. We use the Adam Optimizer to optimize the neural net of Value model. **Batch Size** controls the amount of data used to train the Value model at a time. The **parameter** τ is is used to calculate the advantage of a certain state s with policy π. **Discounting Factor** γ is used to calculate the experimented value of a state s and the corresponding advantage value.

The y-axis of our figures is **reward sum** or **policy loss** of one episode. The reward sum means the rewards the agent obtained in total in one episode and the policy loss means the difference between new policy and old policy which is evaluated in the following way:

$$loss = -\exp(log\ of\ Policy_{new} - log\ of\ Policy_{old}) \times advantage \tag{22}$$

where $Policy_{new}$ and $Policy_{old}$ mean the distribution of new and old policy, and the definition of $advantage$ is showed in the introduction of TRPO.

2.3.1 Figures of TRPO in Three environments

- **Qube**

  The rewards and policy loss in each episode against episode numbers is shown in figure 5.

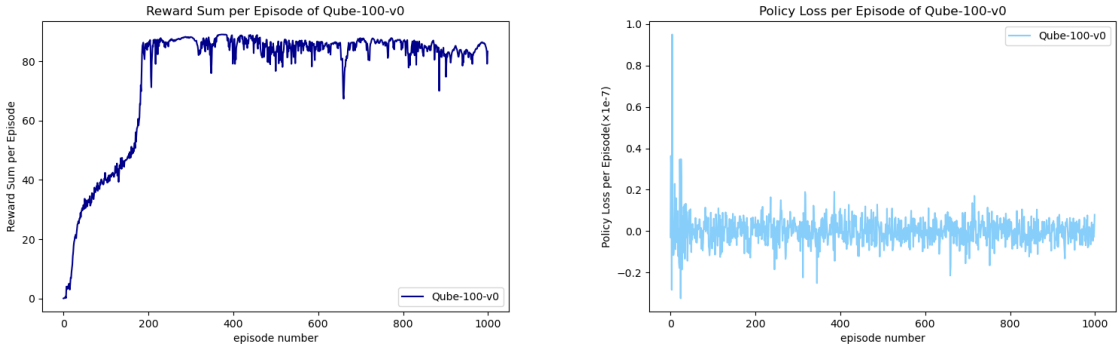

Figure 5: Rewards and Policy Loss against episode numbers in Qube

We can see clearly that the reward per episode will reach the peak 90 and then converge after 220 episodes. The curve sometimes fluctuates largely. This is because the agent will inevitably choose some bad actions according to the policy distribution, resulting in really low rewards sum.

At the same time, the policy loss fluctuate really largely in the first 100 episodes and then fluctuate slightly around 0 from -0.1 to 0.1 which to some extent converges.

- **CartpoleSwing**

  The rewards and policy loss in each episode against episode numbers is shown in figure 6.

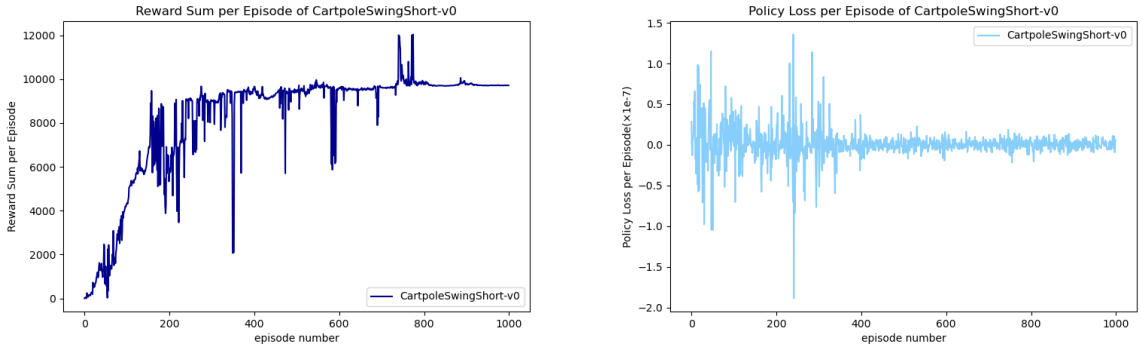

Figure 6: Rewards and Policy Loss against episode numbers in CartpoleSwing

We can see clearly that the reward per episode will reach the peak 10000 and then converge after 900 episodes. The curve sometimes fluctuates largely. This is because the agent will inevitably choose some bad actions according to the policy distribution, resulting in really low rewards sum.

At the same time, the policy loss fluctuate really largely in the first 400 episodes and then fluctuate slightly around 0 from -0.1×10^{-7} to 0.1×10^{-7} which to some extent converges.

- **BallBallancer**

  The rewards and policy loss in each episode against episode numbers is shown in figure 7.

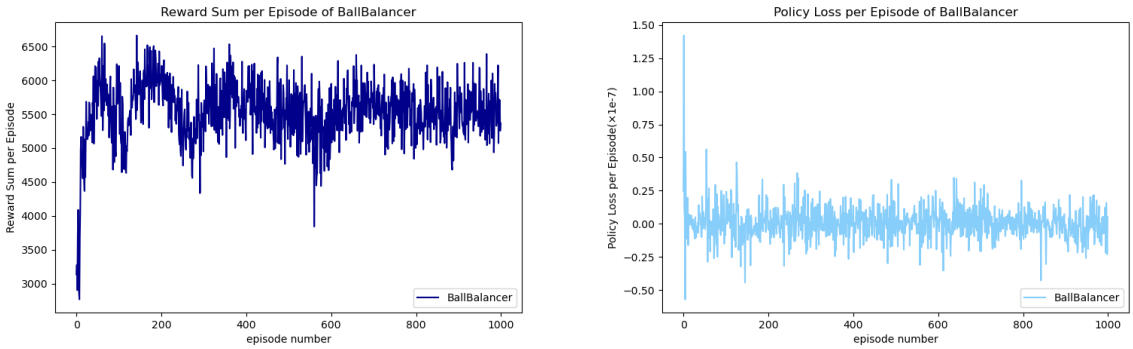

Figure 7: Rewards and Policy Loss against episode numbers in BallBalancer

We can see clearly that the reward per episode can not converge to some stable point even after 1000 episodes. The curve always fluctuates largely around 5500. As for the reasons, **we think that TRPO method may not be really suitable for BallBalancer environment**, so it will take a really long time for the model to converge and we need more training episodes.

The policy loss fluctuates really largely in the first 100 episodes and then fluctuate slighter around 0 from -0.25×10^{-7} to 0.25×10^{-7} but still can not converge to 0.

2.3.2 Theoretical Analyzing of Different Hyper-Parameters

- **Learning Rate** α

  The learning rate is used in the Adam optimizer to optimize the $Value\_net$.
  The larger learning rate is, the less time it will take to converge. Too large learning rate

will cause more vibration in parameters as well as training effects and the model can not converge to a stable point.

The smaller learning rate is, the more likely it is to reach global optimum. Too small learning rate will cause a lot time spent on training.

- **Batch Size**
  The batch size is the size of data used to train the $Value\_net$ in a batch.
  when using gradient descent like Adam optimzer, larger batch size will make the direction of descent more accurate and result in less vibration. However, if the batch size is too large, the model will stuck in local optimum. Small batch size will add more randomness to the training process and worse training effect.

- **Discounting Factor γ**
  The discounting factor is used when calculating the accumulated discounted value of a certain state considering the future.
  The nearer discounting factor is to 1, the farther future our model will consider, because it regards current reward and future rewards as of the same significance.

- **Default Value of Hyper-Parameter**
  We use the idea of controlling variables in experiments. We set a default configuration for hyper-parameters. Then, in each experiment ,we adjust one specific hyper-parameter and evaluate its influence. The default configurations are: learning rate $\alpha = 0.0003$, batch size = 128, discounting factor $\gamma = 0.99$, $\tau = 0.95$, max KL divergence is 0.01. When experimenting and training, we set that in each episode the agent takes 10000 steps, so we record the reward sum of each episode which has the same meaning as mean reward.

2.3.3 Different Hyper-Parameters in Qube Environment

- **Learning Rate α**
  In the Qube environment, we have tried three learning rate: 0.0003, 0.001 and 0.01. The reward and policy loss per episode can be seen in figure 8.

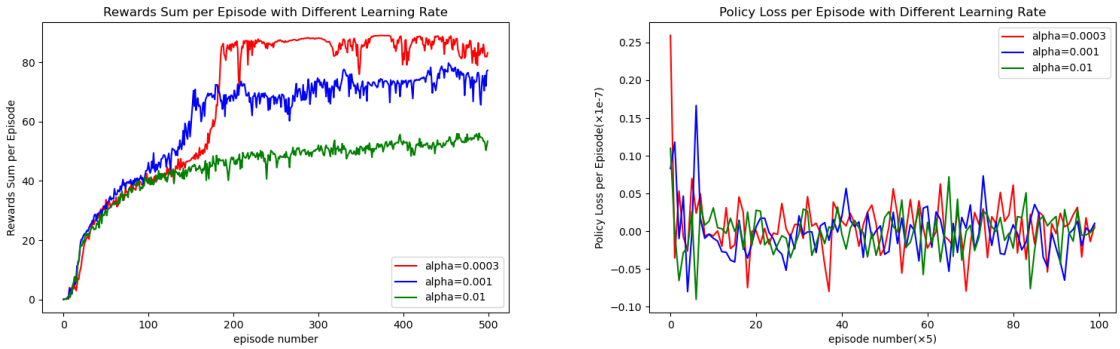

Figure 8: Rewards and Policy Loss with Different Learning Rate in Qube

Reward Sum: In the first 50 episodes, three lines have nearly the same episode reward as well as increasing trend. Then, the increasing rate order becomes: $\alpha = 0.001 > \alpha = 0.0003 > \alpha = 0.01$. This is reasonable because the larger learning rate is, the more quickly for the model to update and the green line with largest learning rate converges to a low point earliest. The time three lines spent on converging and the final episode reward are in accord with our theoretically analyzing that lower learning rate has better effects and results but longer training time.

Policy Loss: The graph shows that learning rate has little influence on the policy loss in Qube environment.

- **Batch Size**

  In the Qube environment, we have tried three Batch Size: 32, 128 and 512. The reward and policy loss per episode can be seen in figure 9.

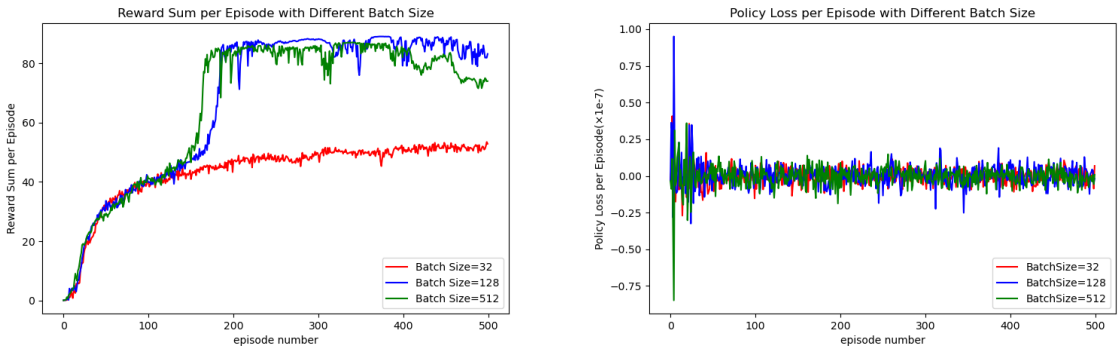

Figure 9: Rewards and Policy Loss with Different Batch Size in Qube

Reward: The figure shows that when batch size is small like 32, the final reward and

effect is largely lower than situations with higher batch size. This is accord with our former analyzing that when using gradient descent, larger batch size will make the direction of descent more accurate and results in less vibration. The line with batch size 128 has better training effect than batch size 512 because too large batch size may cause the model being stuck in local optimum.

Policy Loss: The line with batch size 32 has better performance as it fluctuates slighter than the other two cases. This is because it uses less training data at a time which may not be a good thing.

- **Discounting Factor γ**
  We tried three values of discounting factor: 0.90, 0.95 and 0.99. The rewards and policy loss with γ is shown in figure 10.

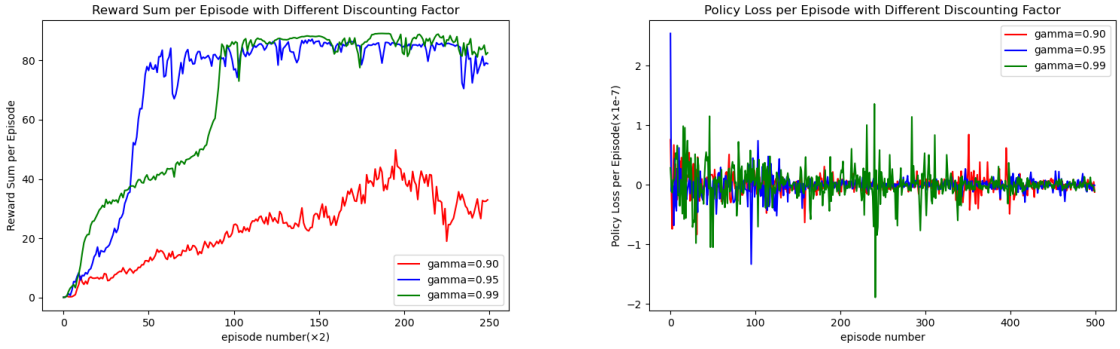

Figure 10: Rewards and Policy Loss with Different Discounting Factor in Qube

Reward: The line with $\gamma = 0.90$ performs largely worse than the other two lines(low reward and can not converge). This is in line with our former analyzing that it considers less and nearer about the future. The line with $\gamma = 0.95$ converges earlier than line with $\gamma = 0.99$, because it gives less weight on future reward and thus less change when updating.

Policy Loss: The line with $\gamma = 0.95$ performs the best with less vibration, showing that $\gamma = 0.95$ is more suitable for Qube environment.

- **Parameter τ to Calculate advantage**
  This parameter is used to calculate the advantage of a state s and we tried three values: 0.85, 0.90 and 0.95. The rewards and policy loss with different τ is shown in figure 11.

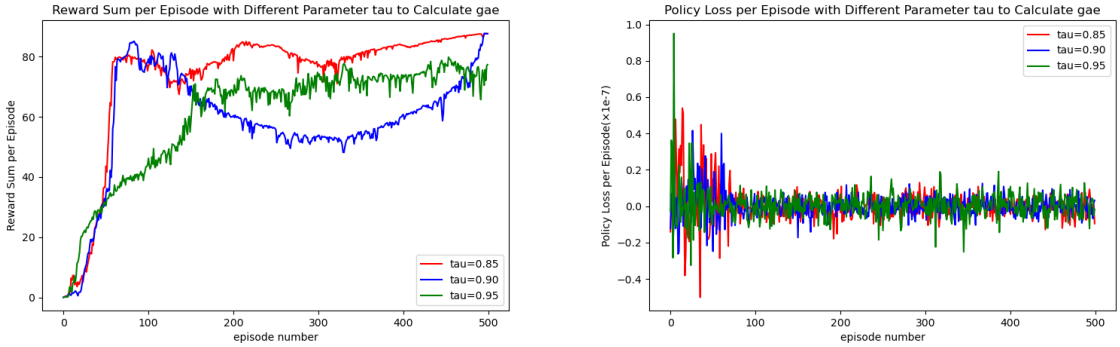

Figure 11: Rewards and Policy Loss with Different Parameter τ in Qube

Reward: The line with $\tau = 0.85$ performs the best as it increases the most quickly and also has the highest converged value. When calculating the advantage of current state, less tau means considering less about the future advantage which may not be a good thing. Thus, we have not decided yet how to explain this confusing phenomenon.

Policy Loss: According to the figure, τ seems to have really slight influence on the policy loss.

- **Max KL Divergence**
  Max KL divergence is used to calculate the scaled kl divergence in the natural gradient process. The rewards and policy loss with different max kl divergence is shown in figure 12.

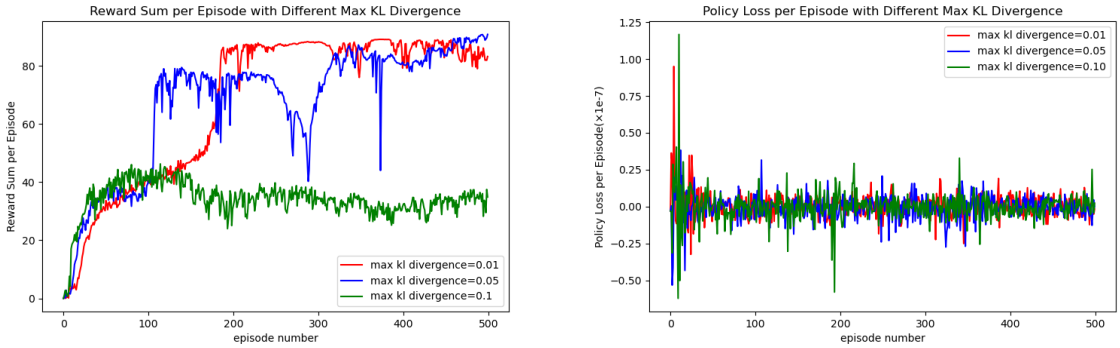

Figure 12: Rewards and Policy Loss with Different Max KL Divergence in Qube

Reward: The figure shows that the less max KL divergence is, the better training effect the model will have. This is also reasonable, because 0.1 and 0.05 are so large that they are not accord with the real situation

Policy Loss: We can see clearly from the graph that line with max Kl divergence=0.01 fluctuates much slighter than the other two lines.

2.3.4 Different Hyper-Parameters in CartpoleSwing Environment

- **Learning Rate α**

  We have tried three learning rate: 0.0003, 0.001 and 0.01. The rewards and policy loss with different learning rate can be seen in figure 13.

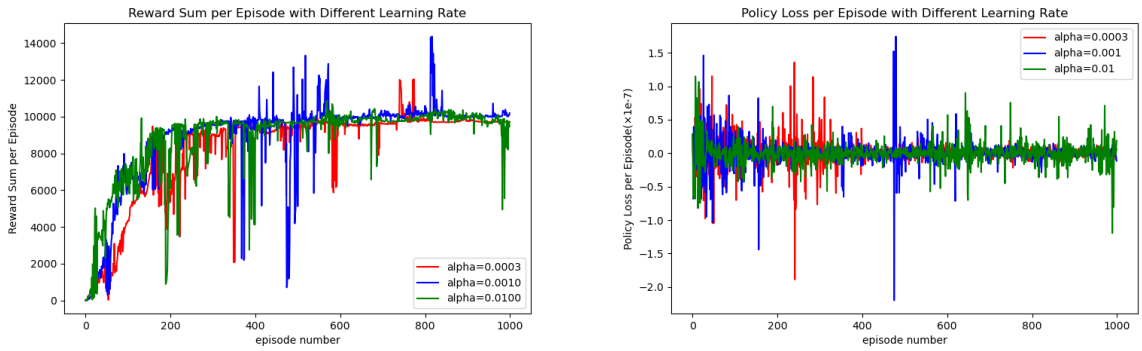

Figure 13: Rewards and Policy Loss with Different Learning Rate in CartpoleSwing

Reward: After 600 episodes, all the curves converge at the same value at about 10000. But it is still easy to see that line with learning rate $\alpha = 0.0003$ performs the best as it fluctuates mush slighter than the other two lines, which is in line with our analyzing in section 2.3.2.

Policy Loss: It is easy to see that after 400 episodes, the line with $\alpha = 0.0003$ converges better and fluctuate much slighter than the other two lines as it is totally covered by other lines. This is also in line with our theoretical analyzing that lower learning rate causes the paramter and training effects to vibrate less.

- **Batch Size**

  We have tried three batch size: 32, 128, 512. The rewards and policy loss per episode with different batch is shown in figure 14.

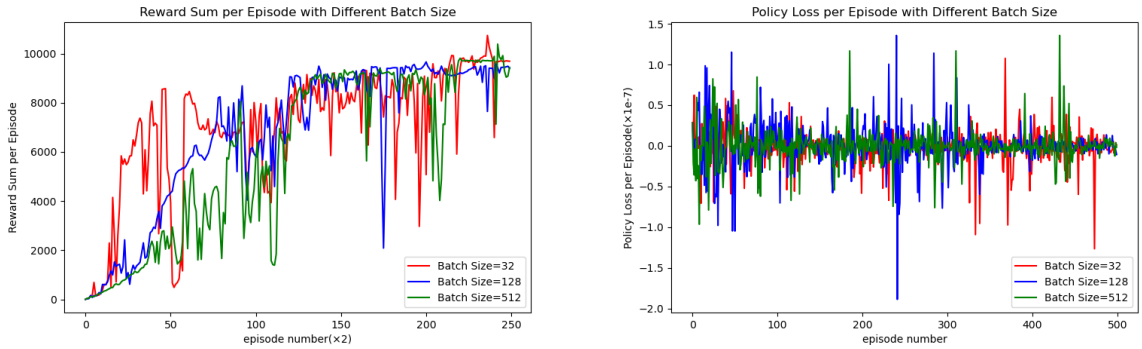

Figure 14: Rewards and Policy Loss with Different Batch Size in CartpoleSwing

Reward: All the three lines fluctuate a lot and eventually reach the same peak value at about 10000. This contradicts our theoretical analyzing in the former section that larger batch size has better training effect as the model parameters vibrate less. The reason we have not yet decided.

Policy Loss: All the lines seem to have not really good performance. Thus, we can obtain the conclusion that: Batch Size has little effect on the model training in CartpoleSwing environment.

- **Discounting Factor** γ
  We experimented three discounting factor: 0.90, 0.95 and 0.99 and their reward as well as policy loss against episode numbers are shown in figure 15.

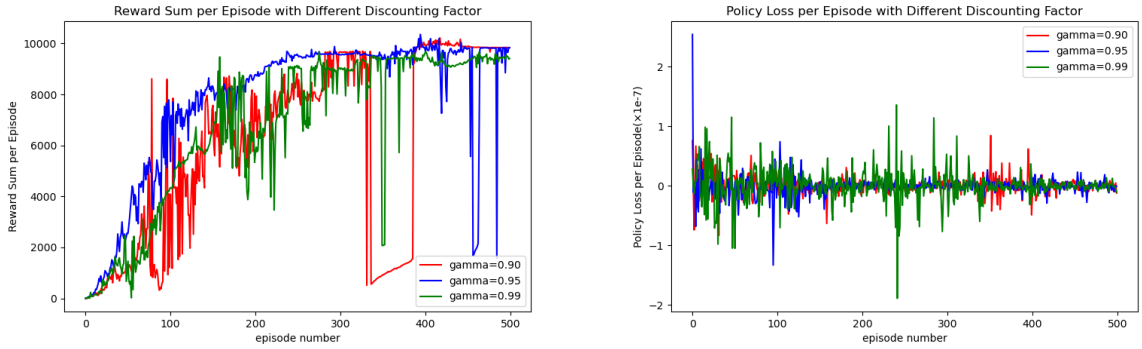

Figure 15: Rewards and Policy Loss with Different Discounting Factor in CartpoleSwing

Reward: All the three lines are not stable in the figure and finally seem to have the same training reward, because of the properties of CartpoleSwing environment like rewards calculation. Among them, the line with the least $\gamma = 0.90$ fluctuates much

more violently than the other two lines. This is in line with our former analyzing in section 2.3.2 that less γ means considering less and shorter about the future value and thus more easy to get a worse policy.

Policy Loss: From the graph, we can clearly see that after 150 episodes, line with $\gamma = 0.95$ fluctuates much slighter around 0 than the other two lines and have better results.

2.3.5 Different Hyper-Parameters in BallBalancer Environment

- **Learning Rate** α

  We have experimented three learning rates: 0.0001, 0.001 and 0.01. The rewards and policy loss are show in figure 16.

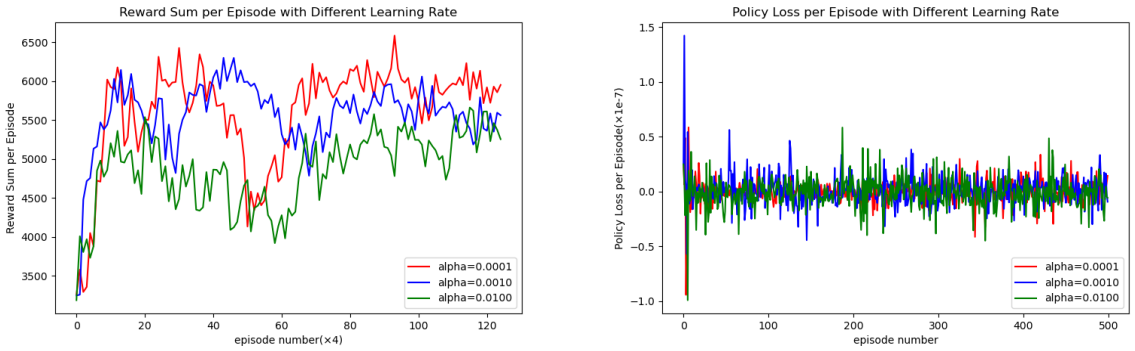

Figure 16: Rewards and Policy Loss with Different Learning Rate in BallBalancer

Reward: Different from the former two environments, it is hard for the model of BallBalancer to converge. However, we can still conclude from the figure that the lower learning rate α is, the better performance and higher reward it will be, which is accord with our analyzing in section 2.3.2. For the limit of time, we run only 500 episodes for each learning rate, so the curve have not converged yet for some properties of BallBalancer environment.

Policy Loss: From the graph we can see that learning rate has little influence on the policy loss per episode.

- **Batch Size**

  We have tried three bitch sizes: 32, 128 and 512. The rewards and policy loss with different batch size are shown in figure 17.

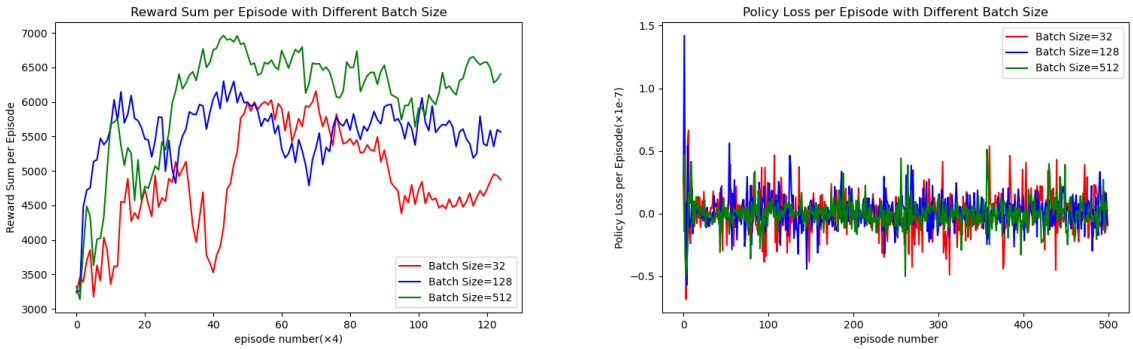

Figure 17: Rewards and Policy Loss with Different Batch Size in BallBalancer

Reward: In the first 20×100 episodes, the increasing rate of reward is in the order: batch size 128 > 512 > 32. Finally, the model with batch size 512 performs the best, model with batch 32 performs the worst. This is reasonable because (a.) the larger the batch size is, the more stable the model will be and thus the more slowly the model will be updated but also the more accurately the model will be trained (b.) when the batch size is too small, it will add more randomness to the training process and thus the model will be unstable.

Policy Loss: The figure 17 directly shows that after 250 episodes, the model with batch size 512 fluctuates more slightly around 0 than the other two lines.

- **Discounting Factor** γ In the BallBalancer environment, we experimented three discounting rate: 0.90, 0.95 and 0.99. The reward and policy loss with different discounting rate are shown in figure 18.

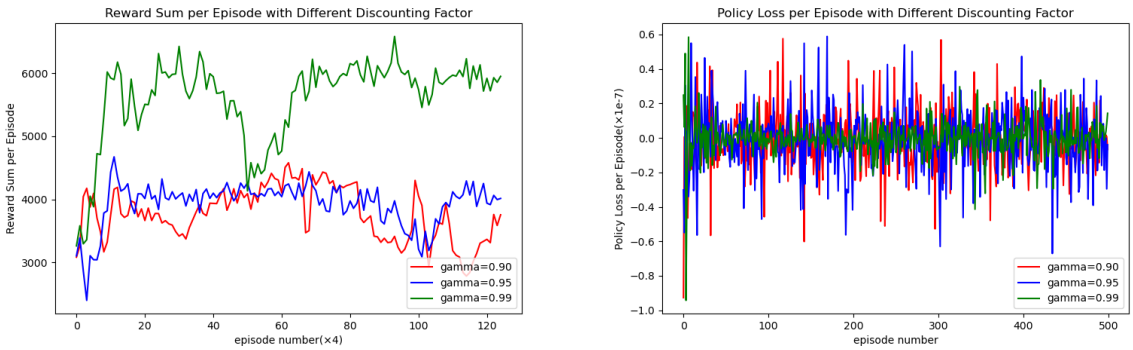

Figure 18: Rewards and Policy Loss with Different Discounting Factor in BallBalancer

Reward: The graph clearly shows that the line with $\gamma = 0.99$ performs the best. This

is in line with our theoretical analyzing that higher value of discounting rate means considering future more and farther, leading to better training effect.

Policy Loss: The graph directly tells us that the line with highest discounting factor performs the best as it fluctuates the most slightly among all the situations, which is accord with our analyzing in secton 2.3.2.

2.4 Training Effect of Three Environments

We have rendered the environment and take three testing videos to see the effects. They are attached in our file.

2.5 Conclusions

After experimenting TRPO in three Quanser Robot environments and rendering the environment to see our training effects, we can draw some conclusions:

- TRPO is super suitable for Qube environment. The model not only is trained quickly to converge, but also has great training effect. It can achieve the task in a really short time. The video is attached in the file.
- TRPO is suitable for BallBalancer environment. Although the reward and policy loss graph may seem not really good because they do not converge, their training effects are good when we test and render the environment.
- TRPO is not really suitable for CartpoleSwing. Although the reward curve and policy loss of this environment can converge greatly after 800 training episodes, it does not perform well when rendering the environment. When testing our model and rendering the environment, we find that instead of trying to make the pole vertical, the cart swings the pole like a fan, which also obtains really a lot of rewards. After discussing, we think that the reward calculation defined in CartpoleSwing is not suitable for TRPO.

After comparing results as well as figures with different hyper-parameters in three environments and combining our theoretical analyzing in section 2.3.2, we can draw some conclusions corresponding to choice of hyper-parameters:

- The learning rate should be properly small so that the model can finally reach global optimum.

- The batch size should be properly large to make the direction of model's gradient descent accurate, resulting in less vibration.
- The discounting rate should be properly near enough to 1 to let the model consider more and farther about the future's value.
- The parameter τ which is used to calculate current state's and policy's advantage should be properly small to pay more attention to current state's advantage instead of considering advantages of the future too much.
- The parameter max KL divergence should be properly small enough to simulate the maximum KL divergence in the real situation.

3 MPC: Model Predictive Control

3.1 Introduction

Model Predictive Control (MPC) is one of the most successful modern control techniques, both regarding its popularity in academics and its use in industrial applications. An important advantage of this type of control is its ability to cope with hard constraints on controls and states.In MPC, the control input is synthesized via the repeated solution of nite horizon optimal control problems on overlapping horizons.
MPC is a model-based reinforcement learning method. Therefore, the algorithm first learn an approximate model based on explorations in the environment. Then, value iteration or other methods is used to get the optimal policy on the pre-trained model. Finally, test the derived-policy in the initial environment and get the reward.

3.2 Learn the Dynamic Model

We denote the dynamic model as $\hat{f}_\theta(s_t, a_t)$. It is actually a neural network model, the parameter θ represents the weights of different attributes in the network. This model is usually very difficult to learn when s_t and its next state s_{t+1} is quite similar. Therefore, we learn a dynamic model to predict the change in s_t instead. The network would take current state s_t and action a_t as input, and output the predicted next state $\hat{s}_{t+1}$: $\hat{s}_{t+1} = s_t + \hat{f}_\theta(s_t, a_t)$.

3.2.1 Collect and Process Training Data

The first step to learn dynamic model is to collect training data. In the algorithm, we collect training data by sampling starting configuration $s_0 \sim p(s_0)$, executing random actions at each time step and recording the trajectories $\tau = (s_0, a_0, ..., s_{T-2}, a_{T-1}, s_{T-1})$ of length T. Next is the data processing stage. We slice the trajectories τ into training data inputs (a_t, s_t) and the corresponding output labels $s_{t+1} - s_t$. Then, we subtract the mean value and divide it by the standard deviation of the data to ensure the loss function weights different attributes of the stage equally.

The implementation of this part is shown below:

```
def collect_random_dataset(self):
    # self is an object of class DataFactory that we use to collect and process data
    datasets = []
    labels = []
    for i in range(self.n_random_episodes):
        data_tmp = []
        label_tmp = []
        state_old = self.env.reset()
        for j in range(self.n_max_steps):
            action = self.env.action_space.sample()
            data_tmp.append(np.concatenate((state_old, action)))
            state_new, reward, done, info = self.env.step(action)
            label_tmp.append(state_new - state_old)
            if done:
                break
            state_old = state_new
        data_tmp = np.array(data_tmp)
        label_tmp = np.array(label_tmp)
        if datasets == []:
            datasets = data_tmp
        else:
            datasets = np.concatenate((datasets, data_tmp))
        if labels == []:
            labels = label_tmp
        else:
            labels = np.concatenate((labels, label_tmp))
    data_and_label = np.concatenate((datasets, labels), axis=1)
    # Merge the data and label into one array and then shuffle
    np.random.shuffle(data_and_label)
    testset_len = int(datasets.shape[0] * self.testset_split)
    data_len = datasets.shape[1]
```

```
32
33 def preprocess(self, x):
34     x = (x - self.mean_data) / self.std_data
35     return x
```

3.2.2 Train the Dynamic Model

In the experiment, we set the dynamic model to be a MLP with four layers: input layer, output layer and two hidden layers.
We train the dynamics model $\hat{f}_\theta(s_t, a_t)$ by minimizing the error

$$\epsilon(\theta) = \frac{1}{|D|} \sum_{(s_t, a_t, s_{t+1}) \in D} \frac{1}{2} ||(s_{t+1} - s_t) + \hat{f}_\theta(s_t, a_t)||^2 \tag{23}$$

While training on the training dataset D, we also calculate the mean squared error in Equation 23 on a validation set D_{valid}, which is composed of trajectories not stored in the training dataset.
The implementation of this part is shown below:

```
1  class DynamicModel(self):
2      def __init__(self):
3          model_config = config["model_config"]
4          self.n_states = model_config["n_states"]
5          self.n_actions = model_config["n_actions"]
6          self.model = MLP(self.n_states + self.n_actions, self.n_states, model_config[
               "n_hidden"],
7                           model_config["size_hidden"])
8          # MLP with two hidden layers
9
10     def train(self, trainset, testset=0):
11         # Normalize the dataset and record data distribution (mean and std)
12         datasets, labels = self.norm_train_data(trainset["data"],trainset["label"])
13         if testset != 0:
14             test_datasets, test_labels = self.norm_test_data(testset["data"],testset[
                   "label"])
15         train_dataset = MyDataset(datasets, labels)
16         train_loader = torch.utils.data.DataLoader(train_dataset, batch_size=self.
               batch_size, shuffle=True)
17         total_step = len(train_loader)
18         print(f"Total training step per epoch [{total_step}]")
```

```
        loss_epochs = []
        for epoch in range(1, self.n_epochs + 1):
            loss_this_epoch = []
            for i, (datas, labels) in enumerate(train_loader):
                datas = self.Variable(torch.FloatTensor(np.float32(datas)))
                labels = self.Variable(torch.FloatTensor(np.float32(labels)))
                self.optimizer.zero_grad()
                outputs = self.model(datas)
                loss = self.criterion(outputs, labels)
                loss.backward()
                self.optimizer.step()
                loss_this_epoch.append(loss.item())
            loss_epochs.append(np.mean(loss_this_epoch))
            if self.save_model_flag:
                torch.save(self.model, self.save_model_path)
            if self.save_loss_fig and epoch % self.save_loss_fig_frequency == 0:
                self.save_figure(epoch, loss_epochs, loss_this_epoch)
                if testset != 0:
                    loss_test = self.validate_model(test_datasets, test_labels)
        return loss_epochs

class MyDataset(data.Dataset):
    def __init__(self, datas, labels):
        self.datas = torch.tensor(datas)
        self.labels = torch.tensor(labels)

    def __getitem__(self, index):  # return tensor
        datas, target = self.datas[index], self.labels[index]
        return datas, target

    def __len__(self):
        return len(self.datas)
```

3.2.3 Model Predictive Control

In order to use the learned mode $\hat{f}_\theta(s_t, a_t)$ together with reward function $r(s_t, a_t)$, we formulate a model-based controller that is both computationally tractable and robust to inaccuracies in the learned dynamics model. First, we optimize the sequence of actions $A_t^H = (a_t, \cdots, a_{t+H-1})$ over a nite horizon H, using the learned dynamics model to predict future

states:

$$A_t^H = \arg\max_{A_t^H} \sum_{k=t}^{k=t+h-1} r(\hat{S}_k, a_k) \tag{24}$$

$$\text{where } \hat{s}_t = s_t, \hat{s}_{k'+1} = \hat{s}_{k'} = \hat{s}_t + \hat{f}_\theta(\hat{s}_{t'}, a_{t'})$$

Calculating the exact optimum of Equation 24 is difcult due to the dynamics and reward functions being nonlinear, but many techniques that exist for obtaining approximate solutions to nite horizon control problems are sufcient for this task. In this work, we use three different methods(Artificial Bee Colony, Cannon and Random). We will give a deeper introduction to these moethods in next section.
Then, we use model predictive control (MPC): the policy executes only the first action a_t, receives updated state information s_{t+1}, and recalculates the optimal action sequence at the next time step

3.2.4 Improve Model Predictive Control with Reinforcement Learning

To improve the performance of our model-based learning algorithm, we gather additional on-policy data by alternating between gathering data with our current model and retraining our model using the aggregated data.
First, random trajectories are collected and added to dataset D_{RAN} , which is used to train $\hat{f}_\theta$ by performing gradient descent on Equantion 23. Then, the model-based MPC controller gathers T new on-policy datapoints and adds these datapoints to a separate dataset D_{RL}. The dynamics function $\hat{f}_\theta$ is then retrained using data from both D_{RAN} and D_{RL} . Note that during retraining, the neural network dynamics functions weights are warm-started with the weights from the previous iteration. The algorithm continues alternating between training the model and gathering additional data until a predened maximum iteration is reached.

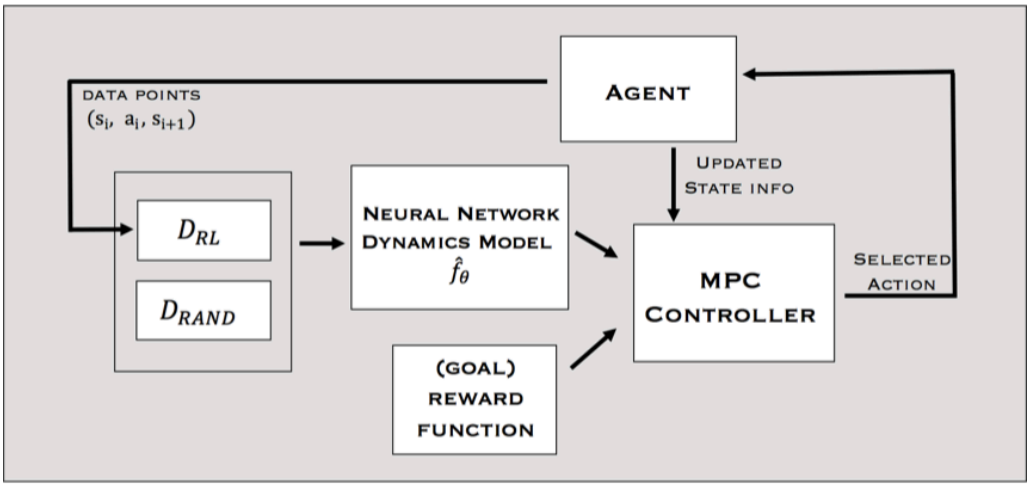

Figure 19: MPC with Reinforcement Learning

The implementation is shown below.

```
def collect_mpc_dataset(self, mpc, dynamic_model, render = False):
    datasets = []
    labels = []
    reward_episodes = []
    for i in range(self.n_mpc_episodes):
        data_tmp = []
        label_tmp = []
        reward_episode = 0
        state_old = self.env.reset()
        for j in range(self.n_max_steps):
            if render:
                self.env.render()
            action = mpc.act(state_old, dynamic_model)
            action = np.array([action])
            data_tmp.append(np.concatenate((state_old, action)))
            state_new, reward, done, info = self.env.step(action)
                reward_episode += reward
                label_tmp.append(state_new - state_old)
                if done:
                    break
            state_old = state_new
        data_tmp = np.array(data_tmp)
        label_tmp = np.array(label_tmp)
        if datasets == []:
            datasets = data_tmp
        else:
```

```
            datasets = np.concatenate((datasets, data_tmp))
        if labels == []:
            labels = label_tmp
        else:
            labels = np.concatenate((labels, label_tmp))
        reward_episodes.append(reward_episode)
    self.mpc_dataset = {"data": datasets, "label": labels}
    self.mpc_dataset_len = datasets.shape[0]
    all_datasets = np.concatenate((datasets, self.all_dataset["data"]))
    all_labels = np.concatenate((labels, self.all_dataset["label"]))
    self.all_dataset = {"data": all_datasets, "label": all_labels}
    if self.save_flag:
        self.save_datasets(self.all_dataset)
    return reward_episodes
```

The pseudocode of this reinforcement learning process is in Algorithm 1.

Algorithm 1 MPC with Reinforcement Learning

gather dataset D_{RAN} of random trajectories
initialize empty dataset D_{RL}, and randomly initialize dynamic model $\hat{f}_\theta$
for $i = 1 \rightarrow max_i trs$ **do**
    train $\hat{f}_\theta(s,a)$ by performing gradient descent on Equation 23 using D_{RAND} and D_{RL}
    for $i = 1 \rightarrow T$ **do**
        get agents current state s_t
        use $\hat{f}_\theta$ to estimate optimal action sequence A_t^H by Equation 24
        execute rst action a_t from selected action sequence A_t^H
        add (s_t, a_t) to D_{RL}
    end for
end for

3.3 MPC Controller Optimizer

MPC controller is the most important factor in the training of dynamic model and its performance depends on the optimization method. The performance of MPC controller determines whether the approximation of dynamic model to the real environment is good enough. In this project, we use three different optimization method for MPC controller: Artificial Bee Colony, Cannon and Random Sampling.

3.3.1 Artificial Bee Colony

This method is inspired by the intelligent foraging behavior of honey bees. The algorithm is specifically based on the model proposed by Tereshko and Loengarov (2005) for the foraging behaviour of honey bee colonies. In ABC, a colony of artificial forager bees (agents) search for rich artificial food sources (good solutions for a given problem).
The source code of ABC can be easily found on Internet and it's quite complex, so we don't show it here. We only show how to use ABC in the MPC controller.

```
def act(self, state, dynamic_model):
    optimizer = Hive.BeeHive( lower = [float(self.action_low)] * self.horizon, upper
        = [float(self.action_high)] * self.horizon, fun = self.evaluator.evaluate,
        numb_bees = self.numb_bees,
    max_itrs = self.max_itrs, verbose=False)
    return optimizer.solution[0]
```

3.3.2 Cannon

This method is inspired by the trajectory of cannonball. After the cannonball is fired from cannon, we can get its landing position. In the next fire, we will adjust the landing position by comparing last cannonball's landing position and the target's location – this is exactly how Cannon works
We first set starting state(landing position) randomly. Then, we update the action a_t we take in the corresponding state to a_{t+1}, according to the reward. When the difference between a_t and a_{t+1} is smaller than , we suppose that the model have converged.
The implementation of Cannon is shown below.

```
class Cannon(object):
    def __init__(self, lower, upper, fun):
        self.lower = lower
        self.upper = upper
        self.horizen = len(self.lower)
        self.evaluate = fun
        self.solution = None

    def run(self):
        epsilon = 0.1
        action_delta = 1 # should be set larger than $\epislon$
        reward_episode = 0
```

```
        render = False
        left_action = self.lower
        right_action = self.upper

        cur_action = []
        for tmp in range(self.horizen):
            cur_action.append(left_action[tmp] + random.random() * (right_action[tmp]
                - left_action[tmp]) )
        cur_action = np.array(cur_action)

        label_tmp = []

        while(action_delta > epsilon):
            action_delta = 0
            for step in range(self.horizen):
                left_reward = self.evaluate(left_action)
                right_reward = self.evaluate(right_action)
                if left_reward > right_reward:
                    action_delta += abs(cur_action[step] - right_action[step])
                    left_action[step] = cur_action[step]
                    cur_action[step] = (right_action[step] + cur_action[step]) / 2
                else:
                    action_delta += abs(cur_action[step] - left_action[step])
                    right_action[step] = cur_action[step]
                    cur_action[step] = (left_action[step] + cur_action[step]) / 2
        self.solution = cur_action
        cur_reward = self.evaluate(cur_action)
        return cur_action, cur_reward

def act(self, state, dynamic_model):
        optimizer = Cannon( lower = [float(self.action_low)] * self.horizon, upper =
            [float(self.action_high)] * self.horizon, fun = self.evaluator.evaluate)
        return optimizer.solution[0]
```

3.3.3 Random Sampling

This method is easy to understand. K candidate action sequences are randomly generated, and the corresponding state sequences are predicted using the learned dynamics model. The rewards for all sequences are calculated, and the candidate action sequence with the

highest expected cumulative reward is chosen.

The implementation of Random Sampling is shown below:

```
fun_e = self.evaluator.evaluate
K = 250
mid = [(float(self.action_low)+float(self.action_high))/2]*self.horizon
random_best = 100
best_action = 0
for k in range(K):
    action_set = []
    for j in range(self.horizon):
        random.seed(time.time()*(j+k+1)*(k+1))
        sub_action = random.random()*(float(self.action_high)-float(self.action_low))
            +float(self.action_low)
        action_set += [sub_action]
    for i in range(len(action_set)):
        action_set[i] += mid[i]
    res = fun_e(action_set)
    if (res<random_best):
    # the result of fun_e is the opposite number of reward
        random_best = res
        best_action = action_set[0]
return best_action
```

3.4 Result with Different Optimization Methods

We apply MPC to three different quanser robots environments: Qube, CartPoleSwing and Ballbalancer. For each environment, we apply three optimization methods to the optimizer, in order to see the effect optimizer to the performance of MPC.

- **Qube**

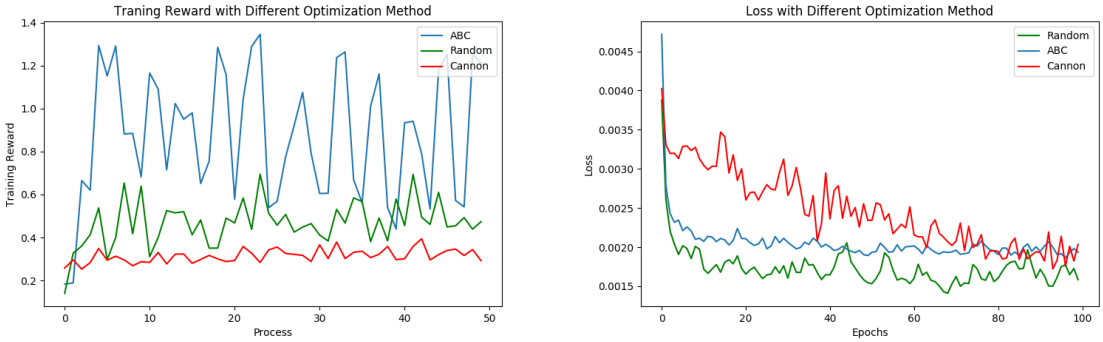

Figure 20: Different Optimization Methods in Qube

Reward: In the reward figure, ABC performs best among three methods, while cannon's perfomance is the weakest. We can also see that the reward fluctuates greatly. This is one of the characteristics of MPC. Since the traning data in each process are different, the agent gets different training reward.
Loss: In the loss figure, all three methods' losses are decreasing and they are all small, which suggests that these methods are reliable. However, the decreasing trend of Cannon is obviously slower than the other two methods, implying that its weaker adjustment to the environment in Qube.

- **CartPoleSwing**

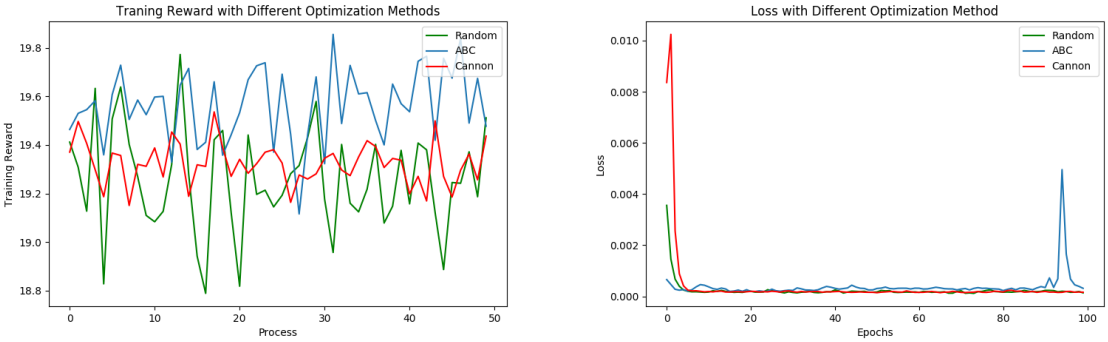

Figure 21: Different Optimization Methods in CartPoleSwing

Reward: In the reward curve, we can see that the performance of ABC is best, and Cannon is in the second place. This shows Cannon can get a result in CartPoleSwing
Loss: In the loss reward, the initial loss of three methods are different, which shows that different methods have different adjustment ability to the environment. After a

few epochs, all methods' loss decrease quickly, showing that they are reliable. It is remarkable that between epochs 80 and 100, the loss of ABC suddenly climbs to a peak. Some sudden change in the newly collected data may lead to this result.

- **BallBalancer**

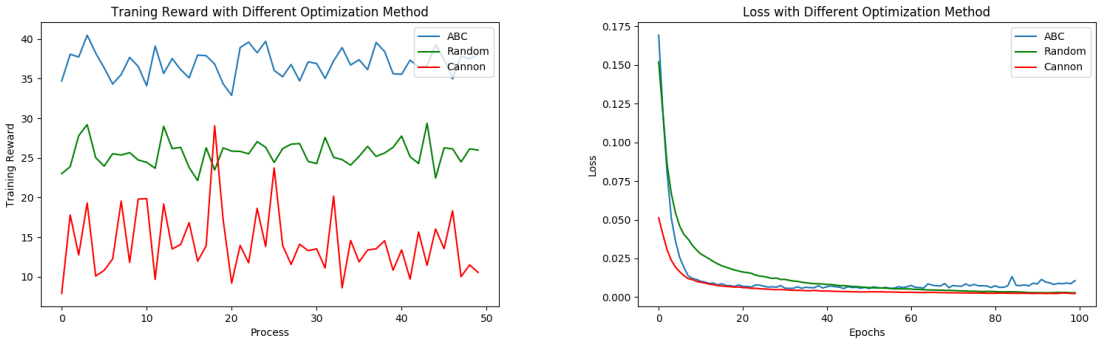

Figure 22: Different Optimization Method in BallBalancer

Reward: In this environment, the difference between performance of each method is quite obvious. The reward figure shows that ABC gets the best performance, while Cannon gets the worst. This result is easy to understand, since ABC and Random spend a large amount of time searching for the optimal action in the state, while Cannon just take one step forward in its horizon.
Loss: In the loss figure, all methods' loss trend in decreasing to a small value, which shows they work in the right way.

- **Conclusion**
  Here we can draw some conclusions to MPC and the three optimization methods based on the results above.
  Artificial Bee Colony(ABC) performs best in all environments, showing that it has the best performance among our methods. But it's also the method that takes the longest time to run.
  Random method's performance is worse than ABC, but better than Cannon, showing that its performance is in the middle position, and so is its time complexity.
  Cannon method has the relatively worst performance in three methods, but it also has advantage: its running time is greatly shorter than the other two methods. If the

training stage has a very strict running time limitation, then this method can come in handy. Besides, its performance is close to ABC in specific environment(such as CartPoleSwing), which suggests that it can be applied to some specific problems.

3.5 Hyper-parameters

There are some hyper-parameters that will influence the performance of MPC greatly. Here we just list all of them and explain their impact. The detailed discussion of each hyper-parameter is in the next section along with figures.

- **Learning Rate** α determines how large each step the agent take.
- **Epochs** determines how many epochs in a single reinforcing process to train the dynamic model.
- **Batch Size** determines how many data are provided for the model training process.
- **Horizon** determines how many steps the agent looks forward.
- **Number of Max Steps** determines how many steps the agent can take at most in a single training episode.
- **Discounting Factor** γ determines experimented reward starting from state s.

3.6 Result with Different Hyper-Parameters

We've also tried different combinations of hyper-parameters, and we'll show the performance of them by figures. Further more, we will discuss the impact that each hyper-parameter has on the performance of MPC.
We use the idea of controlling variables in experiments. First, we set a default configuration for hyper-parameters. Then, in each experiment, we adjust one specific hyper-parameter and see the changes to the result. Most experiments are carried out in Qube, except the experiment of horizon which is carried out in CartPoleSwing. The default configurations are shown in table 8, 9.

learning rate	epochs	batch size	optimizer	horizon	number of max steps
0.001	100	512	ABC	5	500

Table 8: Default Configuration for Qube

learning rate	epochs	batch size	optimizer	horizon	number of max steps
0.001	100	512	Cannon	8	1000

Table 9: Default Configuration for CartPoleSwing

- **Learning Rate**

  Learning rate is one of the most important hyper-parameters in the reinforcement learning algorithm. It determines how large each step the agent takes. On the one hand, if the step is too large, the policy will never perhaps converge. On the other hand, if the step is too small, the policy will take a long time to converge or will be stuck in a local minimun and unable to get out of it.

  In our experiment, we set the learning rate to be 0.01, 0.005 and 0.001.The results are shown in figure 23.

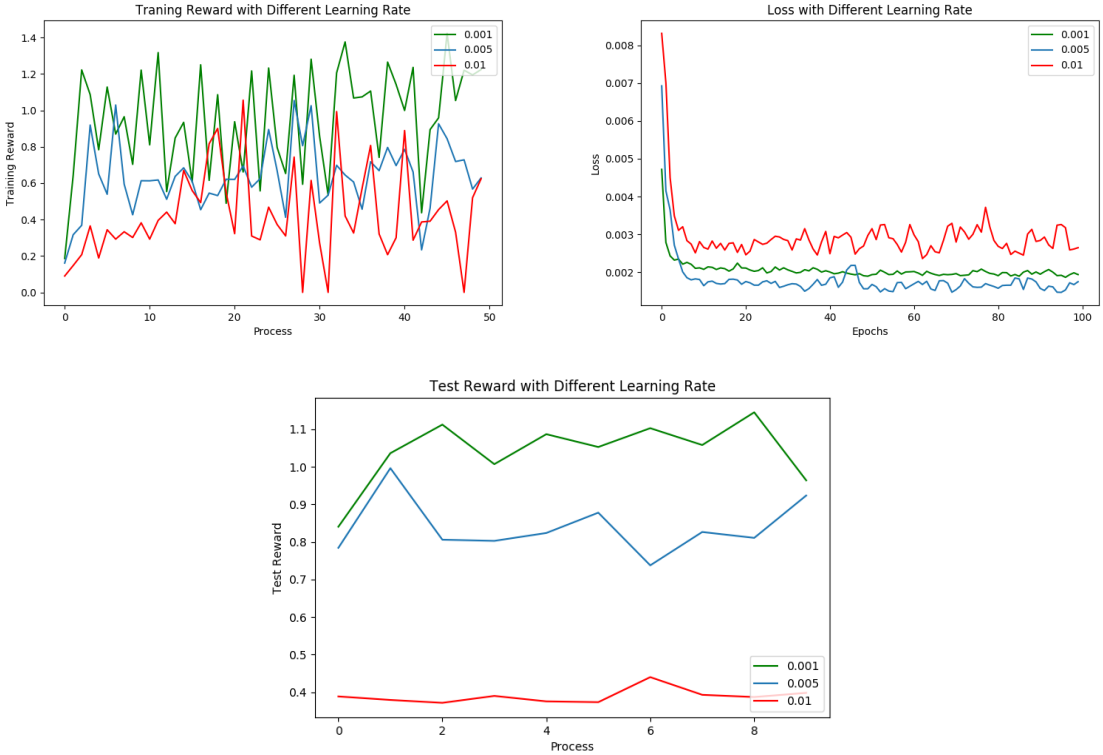

Figure 23: Results with Different Learning Rate

Reward:The result is just like what we've expected: the reward of smaller learning rate is larger. With smaller learning rate, the agent is more likely to reach the state with higher reward.

Loss: All experimental groups learning loss is decreasing, which suggests that their learning processes are on the right track.

- **Epochs**

  The number of epochs determines how many epochs in a single reinforcing process to train the dynamic model. Theoretically, before the policy converges, with larger number of epochs the training effect is better. But after the policy converges, more training epochs may lead to over-fitting. Therefore, the number of epochs should be set properly.

  In the expriement, we set the number of epochs to be 60, 80 and 100. The results are shown in figure 24.

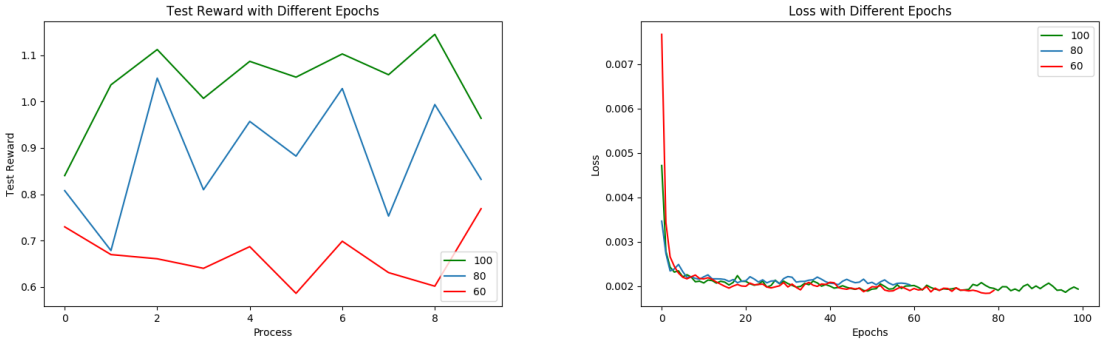

Figure 24: Results with Different Epochs

Test Reward: As we can see in the curve, the reward of larger number of epochs is higher. From this result we can imply two conclusions: First, with number of epochs smaller than 100, the policy will gradually converge, which is the same as we've analyzed; Second, with number of epochs larger than 100, the policy is possible to over-fit.

Loss: All groups training loss is decreasing quickly, suggesting the learning processes are on the right track.

- **Batch Size**

  Batch size determines how many data are provided for the model training process. If the batch size is too small, the model will not be good enough to represent the environment, which leads to bad performance in reinforcement learning process.

  In experiment, we set the batch size to be 32, 128 and 512. The results are shown in figure 25.

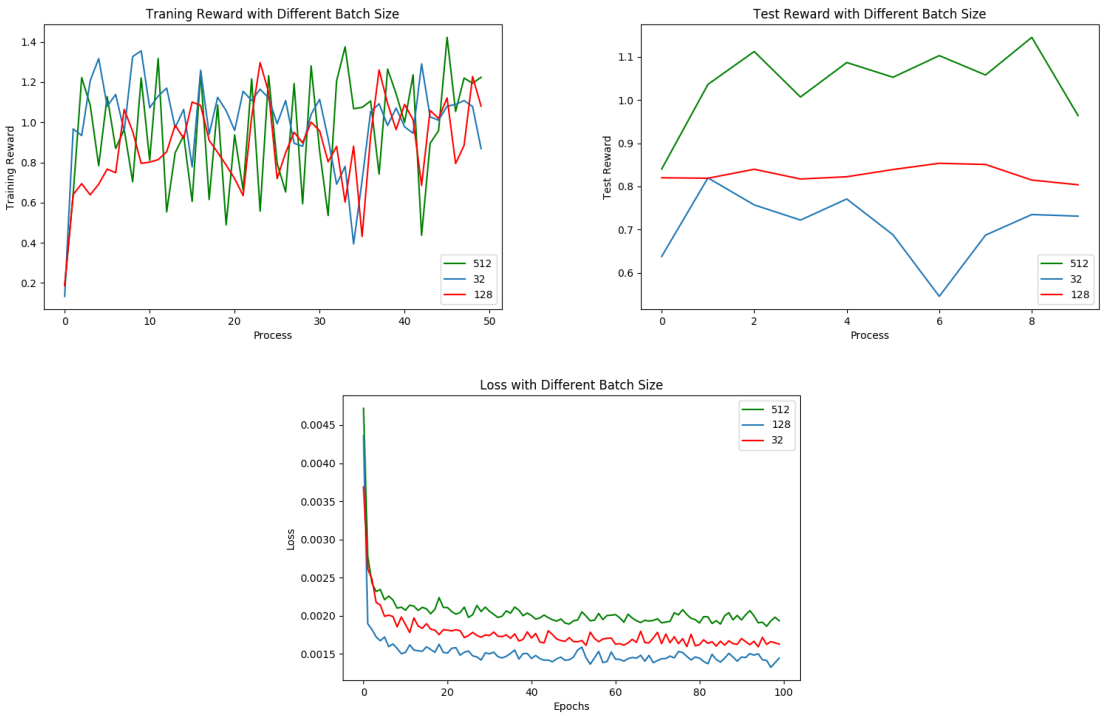

Figure 25: Results with Different Batch Size

Reward: In the training reward curve, three experimental groups' performance are similar to each other. However, if we focus on the fluctuation of the curve, we can find that the fluctuation of larger batch size's reward is less drastic, which suggests that it might face more states because of better dynamic model trained with larger batch size . The test reward curve is in accord with this analysis, since larger batch size group gets better performance in test.

Loss: The loss trend of all groups are identical, which suggests that the learning processes run in the right way.

- **Horizon**

  Horizon determines how many steps the agent looks forward. The agent with larger horizon is possible to see good(or bad) reward in the future so that it can choose the action that gets to the states with higher rewards and escape from states with bad rewards.

  In this experiment, horizon is set to be 2,5 and 8. The results are shown in figure 26.

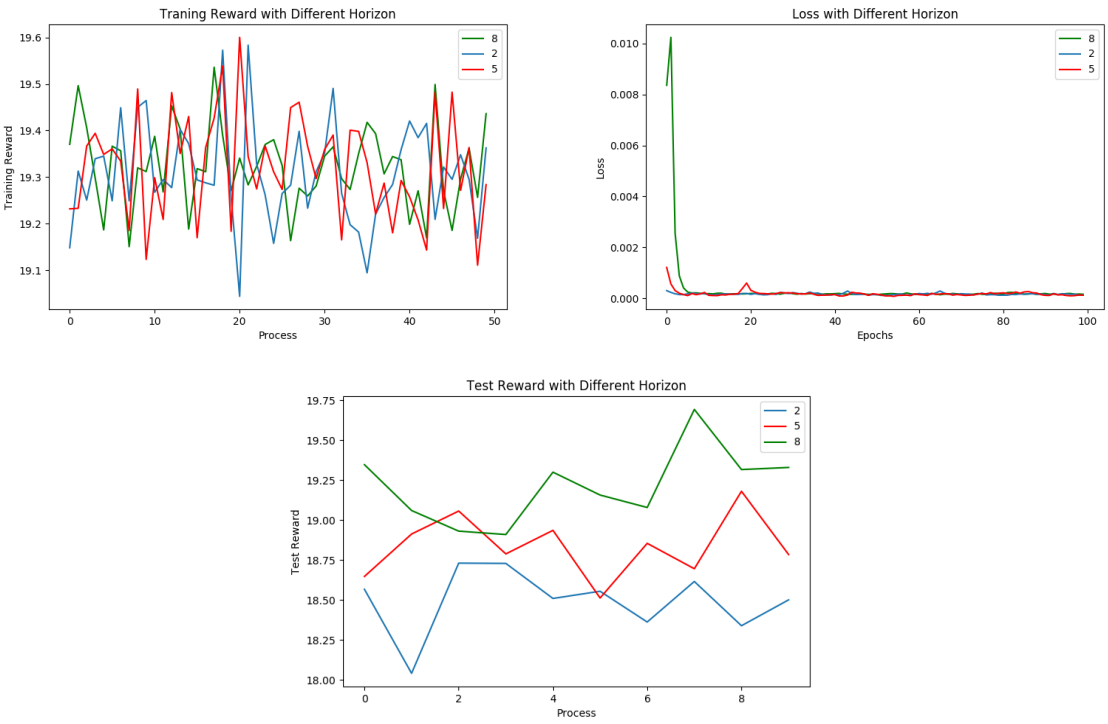

Figure 26: Results with Different Horizon

Reward: The training curve shows that each group's reward is close to each other. But note that the fluctuation of reward of group with smaller horizon is more drastic. The reason is that with larger horizon, the model can make a better approximation to the real environment. As a result, the learning curve will be smoother. The test curve shows that group with larger horizon has better performance.

Loss: The loss curve is a little strange and out of our expectation. In the beginning, loss of group with horizon 8 is largest, suggesting that the agent have some difficulty responding to the environment at first. But after a few epochs, the loss decreases quickly, showing the algorithm's strong adjustment ability.

- **Number of Max Steps**

  The number of max steps determines how many steps the agent can take at most in a training episode. If the number of max steps is too small, the training effect may not be ideal. if the nnumber of max steps is too big, the policy may overfit. In this experiment, we set the number of max steps to be 500 and 1000. The results are shown in figure 27.

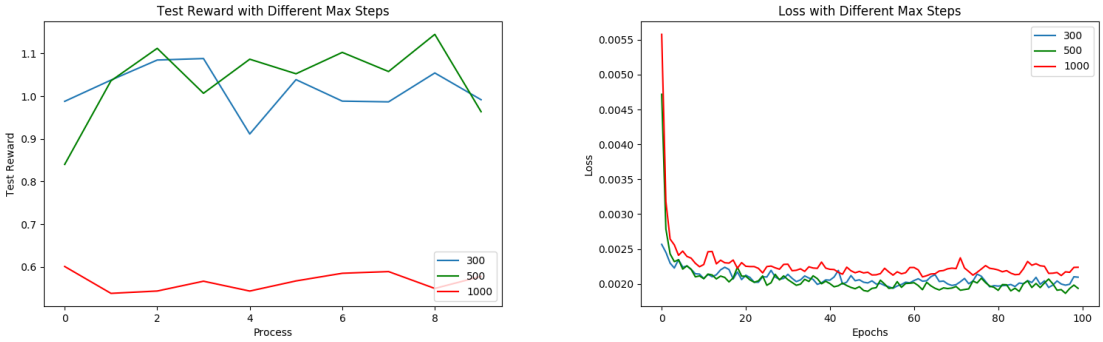

Figure 27: Results with Different Number of Max Steps

- **Conclusion**
  In this section, we discuss some hyper-parameters that will make some influences on the performance of MPC. Here are our conclusions to these hyper-parameters.
  - The learning rate should properly small so that the agent can finally reach global maximum.
  - The epochs should be properly large so that the policy can converge and avoid over-fitting.
  - The batch size should be large enough to produce abundant data for the learning process.
  - The horizon should be large enough so that the policy can get optimal reward.
  - The max steps in learning process should be properly large so that the policy can converge and avoid over-fitting.

4 Comparison between TRPO and MPC

4.1 Difference of Method

- TRPO is a model-free method which does not use the transition probability distribution and the reward function associated with the Markov decision process. This can be thought of as an explicit trial-and-error algorithm. Thus, TRPO is more practical in real situation because it does not need to learn about the environment.
- MPC is a typical model-based method. Its performance greatly depends on the MPC Controller Optimizer. If the Optimizer is good enough, the model trained will be close

to the real environment and the learning effect will be satisfying. However,in all environments, MPC's final performance lags behind TRPO. This is because the model trained in MPC is not equal to the real environment. But when it comes to some problem in which model-free method cannot work, model-based method like MPC may be still working.

4.2 Suitability for Each Environment

- **TRPO** is super suitable for Qube environment. The model not only is trained quickly to converge, but also has great training effect. It can achieve the task in a really short time. The video is attached in the file.
- **TRPO** is not really suitable for BallBalancer environment.The reward and policy loss graph may seem not really good because they do not converge, but their training effects are good when we test and render the environment.
- **TRPO** is not really suitable for CartpoleSwing. Although the reward curve and policy loss of this environment can converge greatly after 800 training episodes, it does perform very well when rendering the environment. When testing our model and rendering the environment, we find that the cart swings the pole like a fan, which also obtains high rewards but is not accord with our expectation. After thinking and discussing, we regard the definition of reward as the reason. The reward is $\cos(\alpha)$ which means if the pole swings quickly, it can also get high rewards.
- The dimension of all three environments' state space is small, so **MPC** works in all environments in the project.

5 Reference

1. A. Nagabandi, G. Kahn, Ronald S. Fearing and S. Levine, "Neural Network Dynamics for Model-Based Deep Reinforcement Learning with Model-Free Fine-Tuning", 2017
2. D. Q. Mayne, J. B. Rawlings, C. V. Rao and P. O. M. Scokaert, "Constrained model predictive control: Stability and optimality", 2000
3. J. Schulman, S. Levine, P. Moritz, M. Jordan, and P. Abbeel, "Trust region policy optimization", 2015