University of Chinese Academy of Sciences

# 倒立摆实验报告

**课程名称:** 强化学习

**报告题目:** 倒立摆实验报告

**学生姓名:** 李昊宸

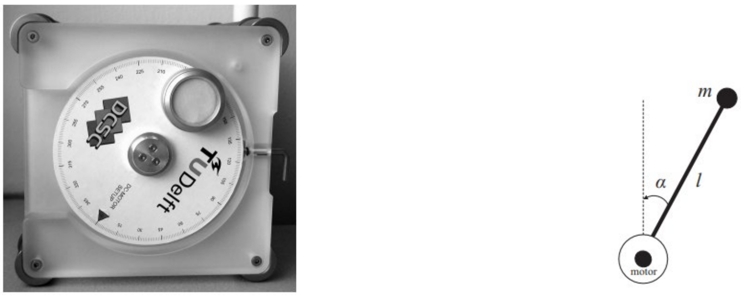
**报告时间:** **2022 年 4 月 11 日**

# 目　录

# 第1章 模型描述

## 1.1 问题介绍

倒立摆是将一个物体固定在一个圆盘的非中心点位置，由直流电机驱动将其在垂直平面内进行旋转控制的系统(图 1)。由于输入电压是受限的,电机并不能提供足够的动力直接将摆杆推完一圈。相反,需要来回摆动收集足够的能量，然后才能将摆杆推起并稳定在最高点。




(a) 真实系统 (b) 示意图

图 1 倒立摆问题

## 1.2 数学模型

表 1 给出了倒立摆的物理系统参数：

| 变量 | 取值 | 单位 | 含义 |
|---|---|---|---|
| $m$ | 0.055 | kg | 重量 |
| $g$ | 9.81 | $\mathrm{m/s^2}$ | 重力加速度 |
| $l$ | 0.042 | m | 重心到转子的距离 |
| $J$ | $1.91 \cdot 10^{-4}$ | $\mathrm{kg \cdot m^2}$ | 转动惯量 |
| $b$ | $3 \cdot 10^{-6}$ | $\mathrm{Nm \cdot s/rad}$ | 粘滞阻尼 |
| $K$ | 0.0536 | Nm/A | 转矩常数 |
| $R$ | 9.5 | $\Omega$ | 转子电阻 |

表 1 倒立摆问题系统参数

根据参数，可以建立倒立摆系统的连续时间动力学模型：

$$\ddot{\alpha}=\frac{1}{J}(mgl\sin(\alpha)-b\dot{\alpha}-\frac{K^2}{R}\dot{\alpha}+\frac{K}{R}u)$$

其中系统的状态为二维组合 $s=[\alpha,\dot{\alpha}]^T$。角 $\alpha\in[-\pi,\pi)rad$，角速度 $\dot{\alpha}\in[-15\pi,15\pi]rad/s$，电压 $u\in[-3,3]V$。

```
def get_diff_diff_alpha(alpha, diff_alpha, u):
    m = 0.055
    g = 9.81
    l = 0.042
    J = 1.91e-4
    b = 3e-6
    K = 0.0536
    R = 9.5
    return (m*g*l*np.sin(alpha) - b*diff_alpha-(K**2)*diff_alpha/R +
K*u/R)/J
```

# 第2章 研究思路

## 2.1 离散化奖励模型

为将连续时间离散化，我们设置倒立摆的采样时间 $Ts = 0.005s$ 。在此基础上，倒立摆系统的离散时间动力学模型由下式给出：

$$\begin{cases} \alpha_{k+1} = \alpha_k + Ts\,\dot{\alpha}_k \\ \dot{\alpha}_{k+1} = \dot{\alpha}_k + Ts\ddot{\alpha}(\alpha_k, \dot{\alpha}_k, u) \end{cases}$$

```
def get_new_state(old_alpha, old_diff_alpha, action):
    Ts = 0.005
    new_alpha = old_alpha + Ts * old_diff_alpha
    new_diff_alpha = old_diff_alpha + Ts * get_diff_diff_alpha(old_alpha,
old_diff_alpha, action)
    #速度限制：[-15pi，15pi]
    max_diff_alpha = 15 * np.pi
    if new_diff_alpha < -max_diff_alpha:
        new_diff_alpha = -max_diff_alpha
    elif new_diff_alpha > max_diff_alpha:
        new_diff_alpha = max_diff_alpha
        return new_alpha, new_diff_alpha
```

控制目标是将摆杆从最低点 $s = [-\pi, 0]^T$ 摆起并稳定在最高点 $s = [0, 0]^T$ 。奖励函数定义成如下二次型形式：

$$\Re(s,u) = -s^T Q_{reward} s - R_{reward} u^2$$

$$Q_{reward} = \begin{pmatrix} 5 & 0 \\ 0 & 0.1 \end{pmatrix}, R_{reward} = 1$$

展开后也即：

$$\Re(\alpha, \dot{\alpha}, u) = -5\alpha^2 - 0.1\dot{\alpha}^2 - u^2$$

最大化该奖励函数的过程，也就是最小化摆点到平衡位置的角度差$|\alpha|$，同时在步进过程中，将角加速度和电压约束到尽量小，以实现系统的稳定。

```
def get_reward(alpha, diff_alpha, action):
    Rrew = 1
        return -(5*alpha**2+0.1*diff_alpha**2)-Rrew*action**2
```

## 2.2 离散化状态空间

采用离散化法，划分连续的角度空间 $S_\alpha$ 和角速度空间 $S_{\dot{\alpha}}$。

将角度空间 $\alpha \in [-\pi, \pi) rad$ 划分为连续且独立的 200 份：

$$\alpha_i = [-\pi + \frac{\pi i}{100}, -\pi + \frac{\pi(i+1)}{100}) rad, i = 0, 1, ..., 199$$

将角速度空间 $\dot{\alpha} \in [-15\pi, 15\pi] rad/s$ 也划分为连续且独立的 200 份：

$$\dot{\alpha}_j = [-15\pi + \frac{3\pi j}{40}, -15\pi + \frac{3\pi(j+1)}{40}) rad/s, j = 0, 1, ..., 199$$

据此，构建离散状态空间 $S = \{S_{ij} = (\alpha_i, \dot{\alpha}_j), i = 1, 2, ..., 199; j = 1, 2, ..., 199\}$

```
def get_indice(alpha, diff_alpha, num_alpha, num_diff_alpha):
    max_diff_alpha = 15 * np.pi
    #正则化 alpha，范围[-pi，pi)
    norm_alpha = (alpha + np.pi) % (2 * np.pi) - np.pi
    #alpha 下标范围[0,num_alpha-1]
    indice_alpha = int((norm_alpha + np.pi)/(2*np.pi) * num_alpha)
    #diff_alpha 下标范围[0,num_diff_alpha-1]
    indice_diff_alpha = int((diff_alpha + 
max_diff_alpha)/(2*max_diff_alpha) * num_diff_alpha)
    if indice_diff_alpha == num_diff_alpha:
        indice_diff_alpha -= 1
        return indice_alpha, indice_diff_alpha
```

## 2.3 离散化动作空间

采用离散的有限动作集，选取 3 个代表性动作 $u \in \{A_0 = -3V, A_1 = 0V, A_2 = 3V\}$。

```
actions = np.array([-3, 0, 3])
```

# 第3章　方法实现

## 3.1　Q-Learning

首先，采用 Q-Learning 方法求解该模型。

### 3.1.1　目标更新

Q-Learning 是一种离策略学习：选取行动采用 ε -greedy(Q)策略，而更新目标的策略使用 greedy(Q)。

迭代中，在每一步长的探索中，智能体通过 ε -greedy(Q)策略选取当前状态 $s_t$ 对应的动作 $a_t$，并据此生成数据 $(s_t, a_t, r_{t+1}, s_{t+1})$。

更新 Q 函数时，以贪心策略 $\pi(s) = \arg\max_\alpha Q(s,a)$ 的 Q 值作为时间差分目标，即：

$$
\begin{aligned}
Q_\pi(s,a) &= \Re_s^a + \gamma \sum_{s'} P_{ss'}^a \sum_{a'} \pi(a'|s') Q_\pi(s',a') \\
&= \Re_s^a + \gamma \sum_{s'} P_{ss'}^a \max_{a'} Q_\pi(s',a') \\
&\approx r_{t+1} + \gamma \max_{a'} Q(s_{t+1},a')
\end{aligned}
$$

据此，得到 Q 函数的迭代公式：

$$
Q(s_t,a_t) = Q(s_t,a_t) + \alpha(r_{t+1} + \gamma \max_{a'} Q(s_{t+1},a') - Q(s_t,a_t))
$$

### 3.1.2　算法实现

```
def train(num_alpha, num_diff_alpha):

    #状态限制
    min_alpha, max_alpha = -np.pi, np.pi
    min_diff_alpha, max_diff_alpha = -15 * np.pi, 15*np.pi
    #Q 动作状态-价值函数
    Q = np.zeros([3, num_alpha, num_diff_alpha])
    #离散动作
    actions = np.array([-3, 0, 3])
    # 迭代次数
    N = 20000
    #ε-greedy 超参
    epsilon = 1.0
    epsilon_limit = 0.01
    # 初始学习率
    lr = 1.0
    #衰减率
```

```
    decay = 0.9995
    #折扣因子
    gamma = 0.98
    # 初始状态
    state_alpha, state_diff_alpha = -np.pi, 0
    total_step = 300
    optimal = -1e7
    finalerror = 2*np.pi
    total_angle = np.pi
    # 探索步长限制
    step_control = 300
    #收敛目标控制
    stable_target = 2
    converted_alpha, converted_diff_alpha = 0.05, 0.01
    #迭代
    for i in range(N):
        #每一轮迭代初始化
        alpha, diff_alpha, iteration, total_reward = state_alpha, state_diff_alpha, 0, 0
        #真实坐标到离散状态空间转换
        indice_alpha, indice_diff_alpha = env.get_indice(alpha, diff_alpha, num_alpha, num_diff_alpha)
        #收敛控制变量
        stable_state = 0
        #更新 ε 和学习率
        epsilon = max(decay * epsilon, epsilon_limit)
        lr = decay * lr
        angles, total_acts = [], []
        angles.append(alpha)
        error_a = np.abs((alpha-min_alpha) % (max_alpha - min_alpha) + min_alpha)
        #开始迭代
        while iteration < step_control:
            iteration += 1
            #Π= ε-greedy(Q)采样动作 at ~ Π(st)
            greedy_action = env.get_greedy_action(Q, indice_alpha, indice_diff_alpha, epsilon)
            total_acts.append(actions[greedy_action])
            #执行动作，获取观测量 rt+1，st+1
            new_alpha, new_diff_alpha = env.get_new_state(alpha, diff_alpha, actions[greedy_action])
            error = np.abs((new_alpha-min_alpha) % (max_alpha - min_alpha) + min_alpha)
            if error < error_a:
                min_angle = new_alpha
            error_a = error
            #收敛控制
```

```
            if error_a < converted_alpha and np.abs(diff_alpha) < con-
verted_diff_alpha:
                angles.append(new_alpha)
                stable_state += 1
                #如果误差小于收敛阈值，并且连续保持收敛状态，即稳定在最高点
                if stable_state == stable_target:
                    break
            #新状态到状态空间转换
            indice_new_alpha, indice_new_diff_alpha =
env.get_indice(new_alpha, new_diff_alpha, num_alpha, num_diff_alpha)
           #greedy(Q)贪心策略选取用于更新的动作
            max_newQa = np.max(Q[:, indice_new_alpha, indi-
ce_new_diff_alpha])
            #获取更新动作的奖励值
            reward = env.get_reward(alpha, diff_alpha, ac-
tions[greedy_action])
           #更新 Q
            deta = reward + gamma * max_newQa -
Q[greedy_action][indice_alpha][indice_diff_alpha]
            Q[greedy_action][indice_alpha][indice_diff_alpha] += lr * deta
            alpha, diff_alpha, indice_alpha, indice_diff_alpha = new_alpha,
new_diff_alpha, indice_new_alpha, indice_new_diff_alpha
            angles.append(alpha)
            total_reward += reward
        total_step = iteration if iteration < total_step else total_step
        min_angle = np.abs((min_angle + np.pi) % (2 * np.pi) - np.pi)
            total_angle = min_angle if min_angle < total_angle else to-
tal_angle
```

## 3.2 SARSA

Q-Learning 是离策略学习，在更新时永远选择奖励最高的动作，不考虑带来的其他后果，因此比较激进。为作比较，我们再采用 SARSA 方法求解该模型。

### 3.2.1 目标更新

SARSA 是一种在策略学习：选取行动和更新目标的策略都使用 ε -greedy(Q)。

迭代中，在每一步长的探索中，智能体执行当前状态 $s_t$ 对应的动作 $a_t$，据此生成数据 $(s_t, a_t, r_{t+1}, s_{t+1})$。

随后，根据策略 ε -greedy(Q)采样动作 $a_{t+1}$，并根据 $(s_t, a_t, r_{t+1}, s_{t+1}, a_{t+1})$ 更新 Q，即：

$$Q(s_t, a_t) = Q(s_t, a_t) + \alpha(r_{t+1} + \gamma Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t))$$

### 3.2.2 算法实现

```
def train(num_alpha, num_diff_alpha):

    #状态限制
    min_alpha, max_alpha = -np.pi, np.pi
    min_diff_alpha, max_diff_alpha = -15 * np.pi, 15*np.pi
    #Q 动作状态-价值函数
    Q = np.zeros([3, num_alpha, num_diff_alpha])
    #离散动作
    actions = np.array([-3, 0, 3])
    # 迭代次数
    N = 20000
    #ε-greedy 超参
    epsilon = 1.0
    epsilon_limit = 0.01
    # 初始学习率
    lr = 1.0
    #衰减率
    decay = 0.9995
    #折扣因子
    gamma = 0.98
    # 初始状态
    state_alpha, state_diff_alpha = -np.pi, 0
    total_step = 300
    optimal = -1e7
    finalerror = 2*np.pi
    total_angle = np.pi
    # 探索步长限制
    step_control = 300
    #收敛目标控制
    stable_target = 2
    converted_alpha, converted_diff_alpha = 0.05, 0.01
    #迭代
    for i in range(N):
        #每一轮迭代初始化
        alpha, diff_alpha, iteration, total_reward = state_alpha,
state_diff_alpha, 0, 0
        #真实状态到离散状态空间转换
        indice_alpha, indice_diff_alpha = env.get_indice(alpha, diff_alpha,
num_alpha, num_diff_alpha)
        #Π= ε-greedy(Q), 根据该策略采样动作 at~Π(st)
        greedy_action = env.get_greedy_action(Q, indice_alpha, indi-
ce_diff_alpha, epsilon)
        # ε 和学习率更新
        epsilon = max(decay * epsilon, epsilon_limit)
        lr = decay * lr
        angles, total_acts = [], []
        angles.append(alpha)
```

```
        error_a = np.abs((alpha-min_alpha) % (max_alpha - min_alpha) +
min_alpha)
        while iteration < step_control:
            iteration += 1
            total_acts.append(actions[greedy_action])
            #根据 st，at 获取新状态 st+1
            new_alpha, new_diff_alpha = env.get_new_state(alpha,
diff_alpha, actions[greedy_action])
            error = np.abs((new_alpha-min_alpha) % (max_alpha - min_alpha)
+ min_alpha)
            if error < error_a:
                min_angle = new_alpha
            error_a = error
            #收敛控制
            if error_a < converted_alpha and np.abs(diff_alpha) < con-
verted_diff_alpha:
                angles.append(new_alpha)
                stable_state += 1
                print(stable_state)
                if stable_state == stable_target:
                    break
            #新状态映射到离散状态空间
            indice_new_alpha, indice_new_diff_alpha =
env.get_indice(new_alpha, new_diff_alpha, num_alpha, num_diff_alpha)
            #在策略更新：根据 st+1 选择下一步动作 at+1
            greedy_new_action = env.get_greedy_action(Q, indice_new_alpha,
indice_new_diff_alpha, epsilon)
            #计算执行 at 到达 st+1 的奖励
            reward = env.get_reward(alpha, diff_alpha, ac-
tions[greedy_action])
            #根据 st+1 和 at+1 和 reward 更新 Q(st,at)
            update_Q = reward + gamma *
Q[greedy_new_action][indice_new_alpha][indice_new_diff_alpha] -
Q[greedy_action][indice_alpha][indice_diff_alpha]
            Q[greedy_action][indice_alpha][indice_diff_alpha] += lr *
update_Q
            #步进
            alpha, diff_alpha, indice_alpha, indice_diff_alpha,
greedy_action = new_alpha, new_diff_alpha, indice_new_alpha, indi-
ce_new_diff_alpha, greedy_new_action
            angles.append(alpha)
            total_reward += reward
        total_step = iteration if iteration < total_step else total_step
        min_angle = np.abs((min_angle + np.pi) % (2 * np.pi) - np.pi)
        total_angle = min_angle if min_angle < total_angle else total_angle
```

# 第4章　实验结果

## 4.1　Q-Learning

Q-Learning 方法在 3900 轮迭代时，出现第一次收敛，收敛步长为 262。经过 20000 轮迭代后，最小收敛步长为 167，收敛时最终距离稳定状态角度的误差为 0.018567 rad。

下面对 Q 函数进行了可视化，为了视觉效果，对 Q 函数进行了取反。

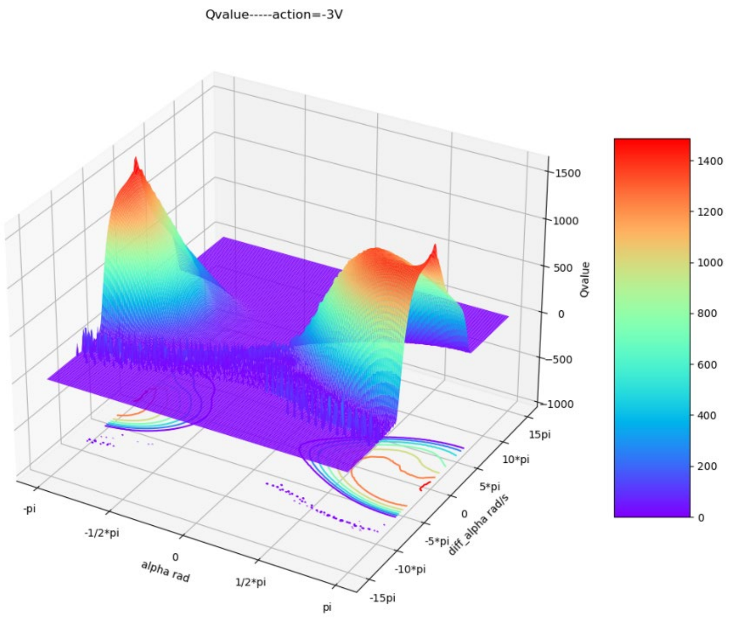



图 2 Q-Learning Q 动作-状态函数可视化（动作取-3V）

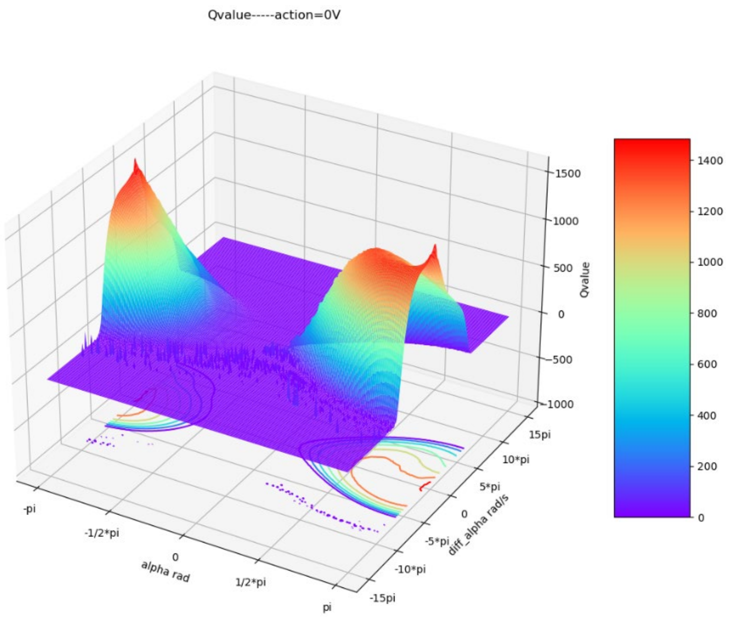



图 3 Q-Learning Q 动作-状态函数可视化（动作取 0V）

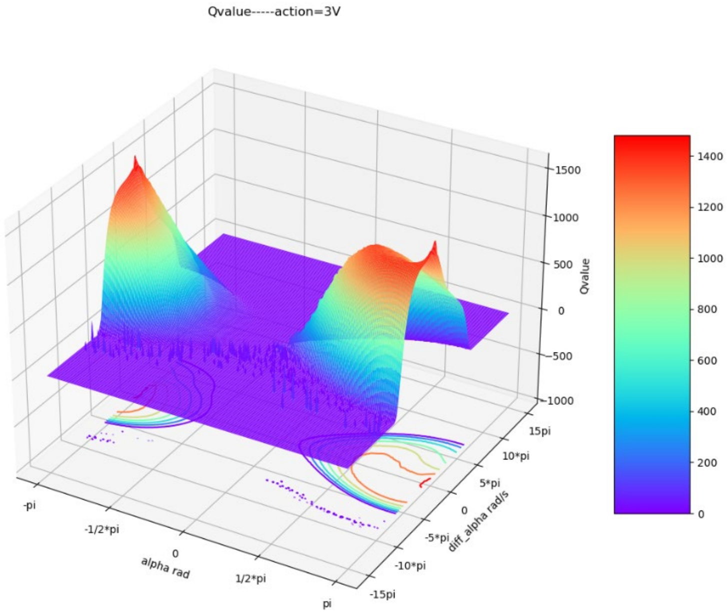



图 4 Q-Learning Q 动作-状态函数可视化（动作取 3V）

可以看出，无论选取哪个动作，在初始状态附近（如 $a = \pm\pi$）时，Q 值达到峰值，这与初始状态的奖励最低常识相吻合。并且，Q 函数以目标状态（$a$=0，$\dot{\alpha}$=0）为中心，形成了鞍点。

下面对于状态空间，在 Q 函数上采取 greedy 策略，将动作-状态选择进行了可视化：

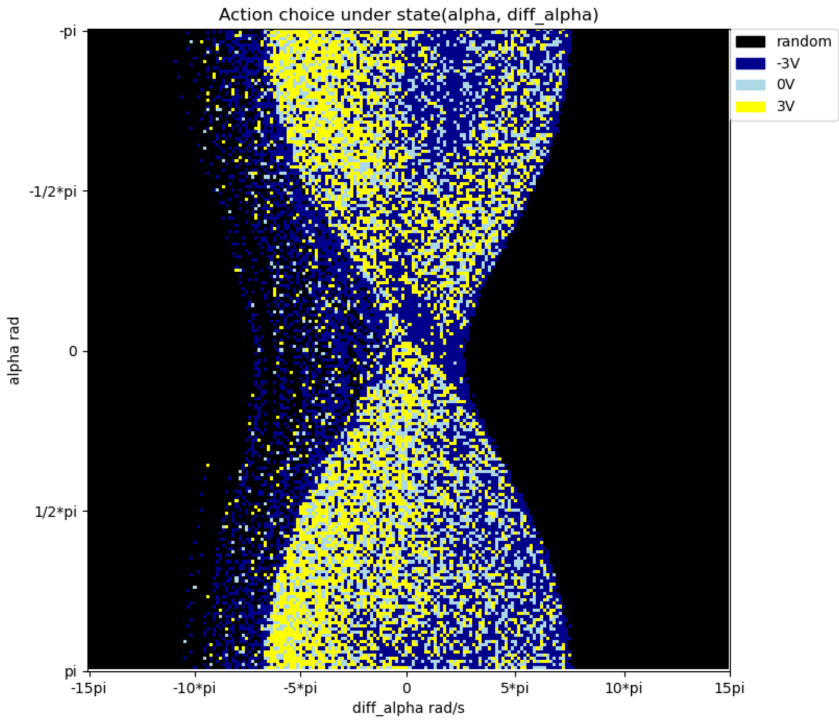



图 5 Q-Learning greedy 策略下的 Q 动作-状态函数可视化

可以看出，动作 $u_0 = -3V$ 与 $u_2 = 3V$ 存在明显的分界线。理论上，动作 $u_1 = 0V$

应分布在分界线附近，但是由于 Q-Learning 较为激进的学习策略，学习到的结果倾向于最快速度的到达平衡位置，尽量不采取对状态影响最小的 $u_1 = 0V$ 动作。

## 4.2 SARSA

SARSA 方法在 6700 轮迭代时，出现第一次收敛，收敛步长为 297。经过 20000 轮迭代后，最小收敛步长为 292，收敛时最终距离稳定状态角度的误差为 0.016288 rad。

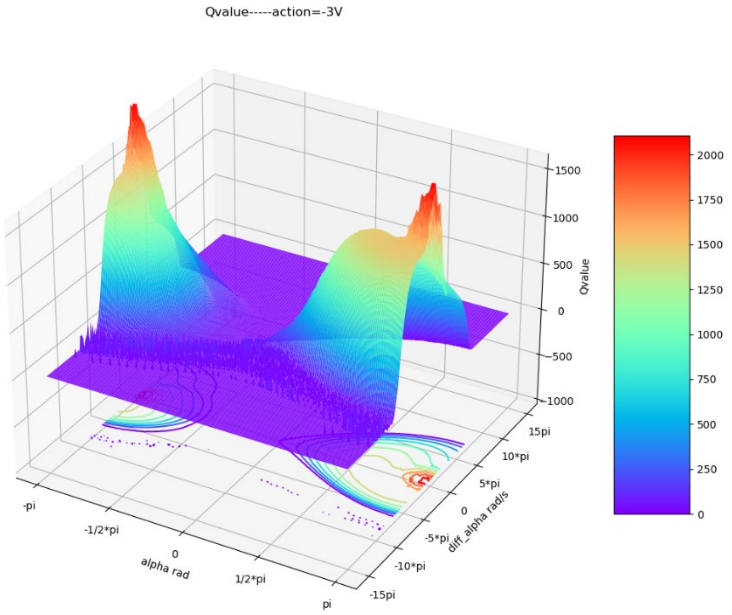



图 6 SARSA Q 动作-状态函数可视化（动作取-3V）

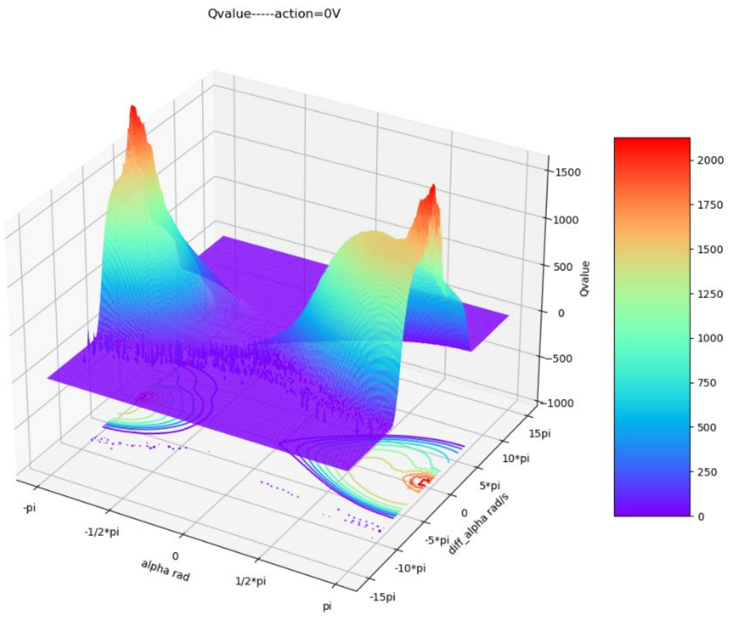



图 7 SARSA Q 动作-状态函数可视化（动作取 0V）

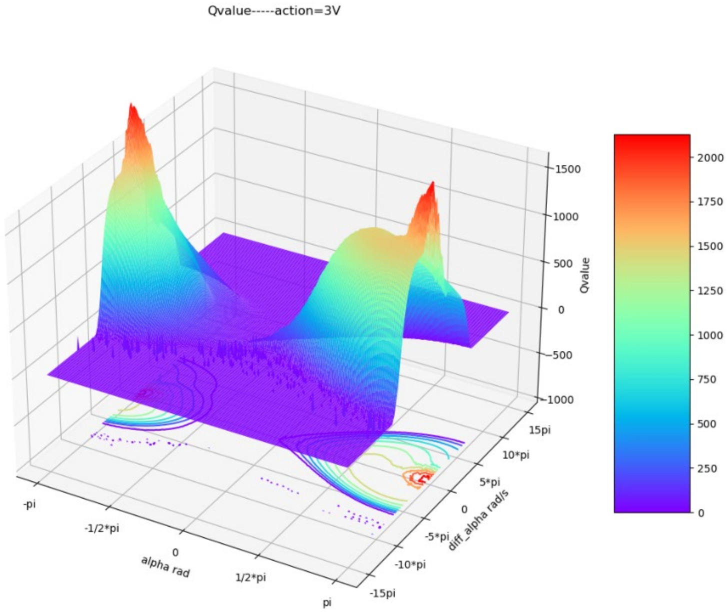



图 8 SARSA Q 动作-状态函数可视化（动作取 3V）

可以看出，无论选取哪个动作，在初始状态附近（如 $a=\pm\pi$）时，Q 值达到峰值，这与初始状态的奖励最低常识相吻合。相比于 Q-Learning 的 Q 函数，SARSA 的 Q 函数更为陡峭，这与 SARSA 的迭代步长相对更长相吻合。

下面对于状态空间，在 Q 函数上采取 greedy 策略，将动作-状态选择进行了可视化：

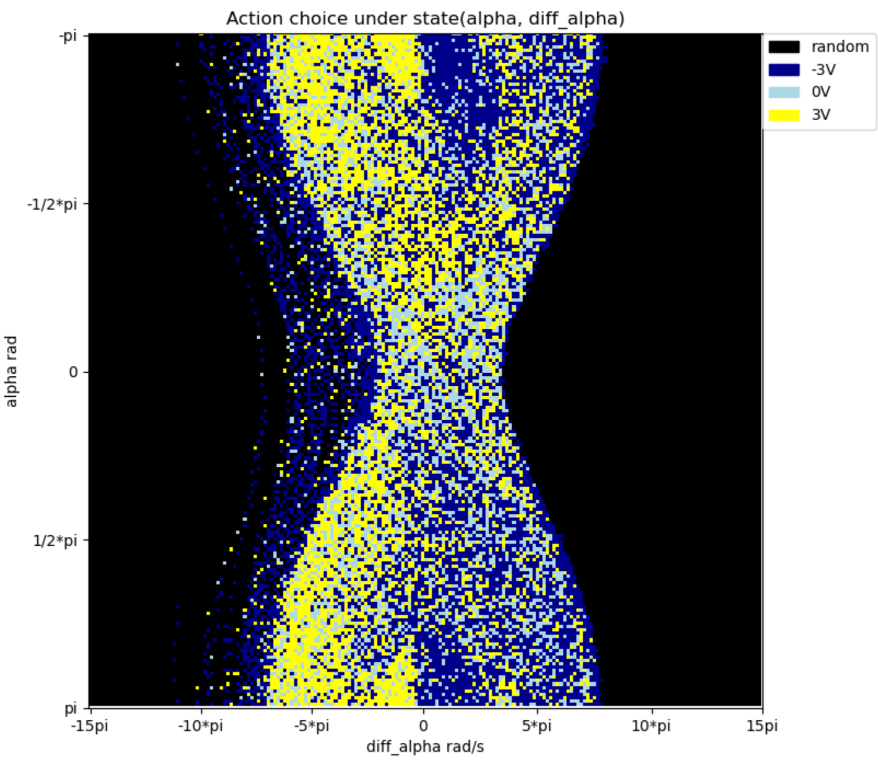



图 9 Q-Learning greedy 策略下的 Q 动作-状态函数可视化

可以看出，动作 $u_0 = -3V$ 与 $u_2 = 3V$ 存在明显的分界线。并且，$u_1 = 0V$ 动作的出现数量明显高于 Q-Learning 的决策。这也证实了 SARSA 更趋向于“安全”“稳定”的动作选择。

### 4.3 Q-Learning 与 SARSA 的学习过程对比

图 10 给出了前 7000 次迭代中，Q-Learning 与 SARSA 方法在停止本轮迭代时的最小 loss。

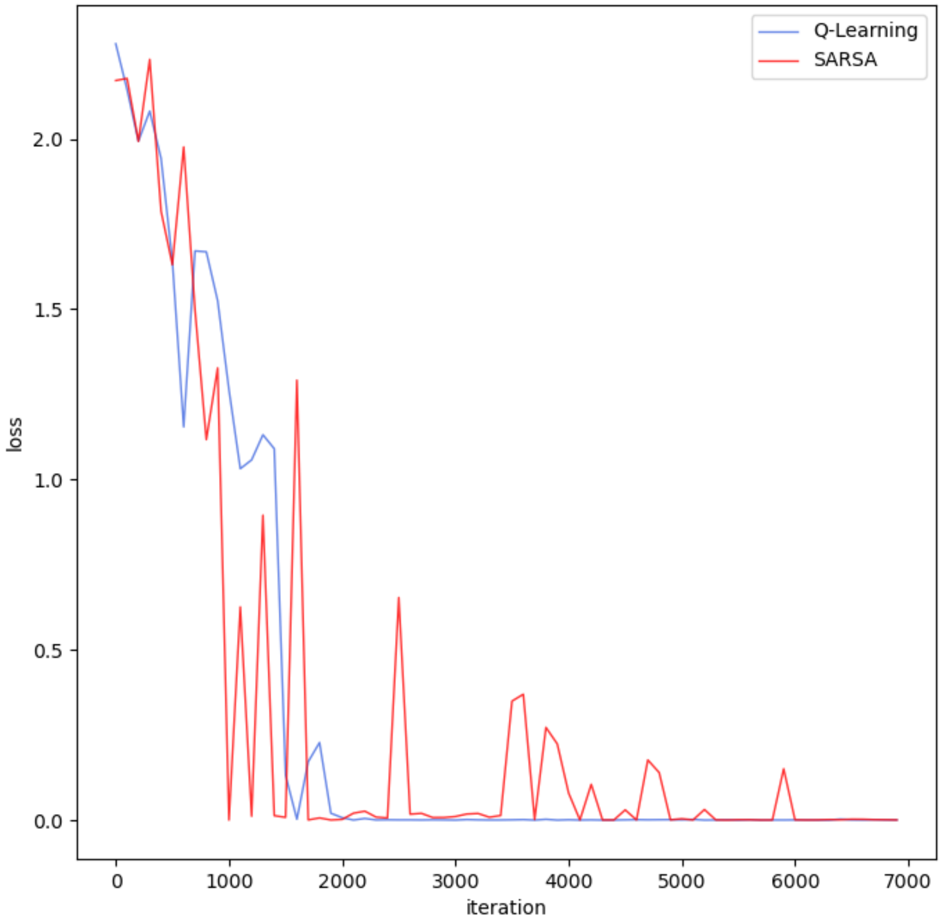



图 10 Q-Learning 与 SARSA 前 7000 轮迭代的 loss 变化

可以看出，Q-Learning 方法的 loss 下降更快，收敛也更快，这与 Q-Learning 每次都选取当前的贪心策略，不在乎之后是否会带来危险，以及 SARSA 每次都要多尝试一步的 ε -贪心策略，更加保守的做出决策相吻合。

另外，我们对最终 Q-Learning 和 SARSA 的决策控制进行了可视化。视频可访问以下链接：

Q-Learning：https://github.com/Therock90421/21-22Reinforcement_learning_experiements/blob/main/inverted_pendulum/Q_Learning.gif

SARSA：https://github.com/Therock90421/21-22Reinforcement_learning_experiements/blob/main/inverted_pendulum/SARSA.gif