

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](#_Toc100849732)

[1.1 问题介绍 1](#_Toc100849733)

[1.2 数学模型 1](#_Toc100849734)

[第2章 研究思路 3](#_Toc100849735)

[2.1 离散化奖励模型 3](#_Toc100849736)

[2.2 离散化状态空间 4](#_Toc100849737)

[2.3 离散化动作空间 4](#_Toc100849738)

[第3章 方法实现 5](#_Toc100849739)

[**3.1 Q-Learning** 5](#_Toc100849740)

[3.1.1 目标更新 5](#_Toc100849741)

[3.1.2 算法实现 5](#_Toc100849742)

[**3.2 SARSA** 7](#_Toc100849743)

[3.2.1 目标更新 7](#_Toc100849744)

[3.2.2 算法实现 8](#_Toc100849745)

[第4章 实验结果 10](#_Toc100849746)

[**4.1 Q-Learning** 10](#_Toc100849747)

[**4.2 SARSA** 12](#_Toc100849748)

[**4.3 Q-Learning与SARSA的学习过程对比** 14](#_Toc100849749)

# 模型描述

## 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。

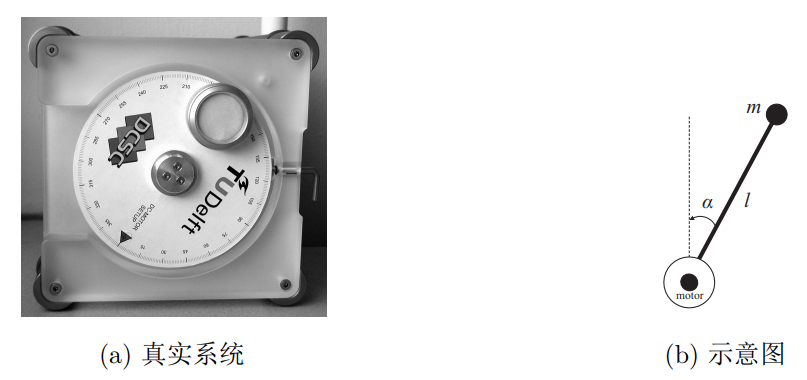


图1 倒立摆问题

## 数学模型

表1给出了倒立摆的物理系统参数：

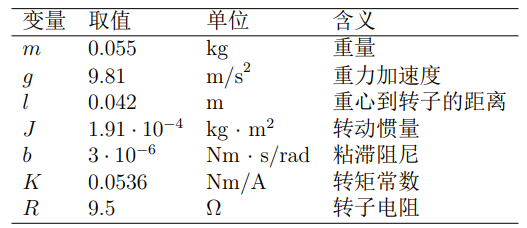


表1 倒立摆问题系统参数

根据参数，可以建立倒立摆系统的连续时间动力学模型：



其中系统的状态为二维组合。角，角速度，电压。

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

# 研究思路

## 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。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：



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摆起并稳定在最高点。奖励函数定义成如下二次型形式：





展开后也即：



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，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

## 离散化状态空间

采用离散化法，划分连续的角度空间和角速度空间。

将角度空间划分为连续且独立的200份：



将角速度空间也划分为连续且独立的200份：



据此，构建离散状态空间

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

## 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。

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

# 方法实现

## **Q-Learning**

首先，采用Q-Learning方法求解该模型。

### 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)策略选取当前状态对应的动作，并据此生成数据。

更新Q函数时，以贪心策略的Q值作为时间差分目标，即：



据此，得到Q函数的迭代公式：



### 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) < converted\_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, indice\_new\_diff\_alpha])

*#获取更新动作的奖励值*

            reward = **env**.**get\_reward**(alpha, diff\_alpha, actions[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 total\_angle

## **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。

### 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对应的动作，据此生成数据。

随后，根据策略ε-greedy(Q)采样动作，并根据更新Q，即：



### 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, indice\_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) < converted\_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, actions[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, indice\_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

# 实验结果

## **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。

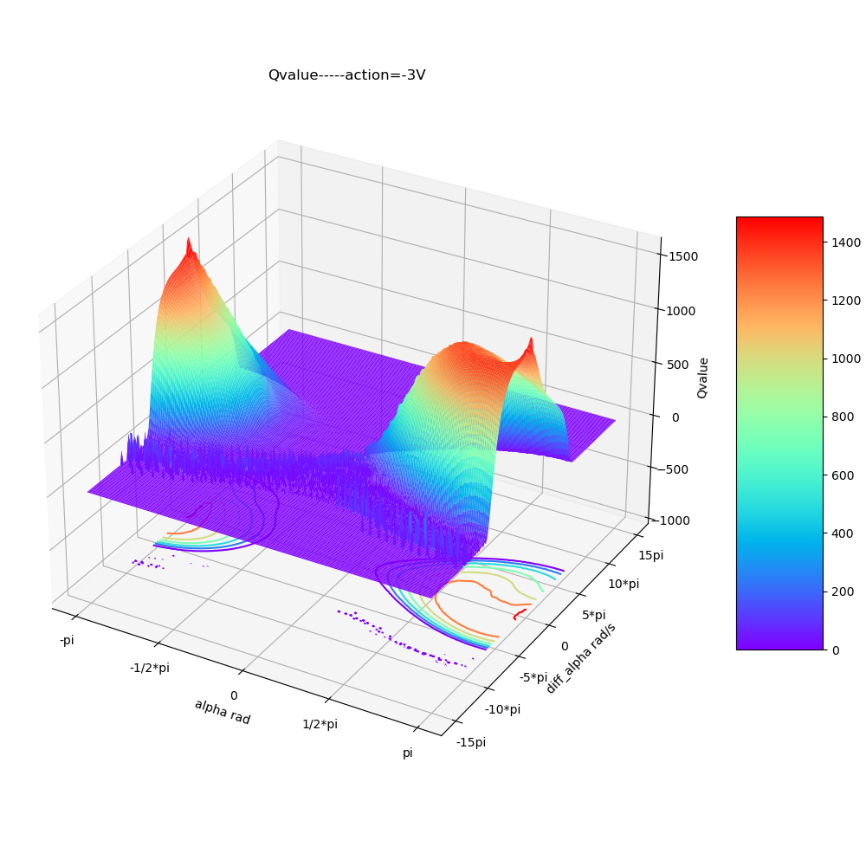


图2 Q-Learning Q动作-状态函数可视化（动作取-3V）

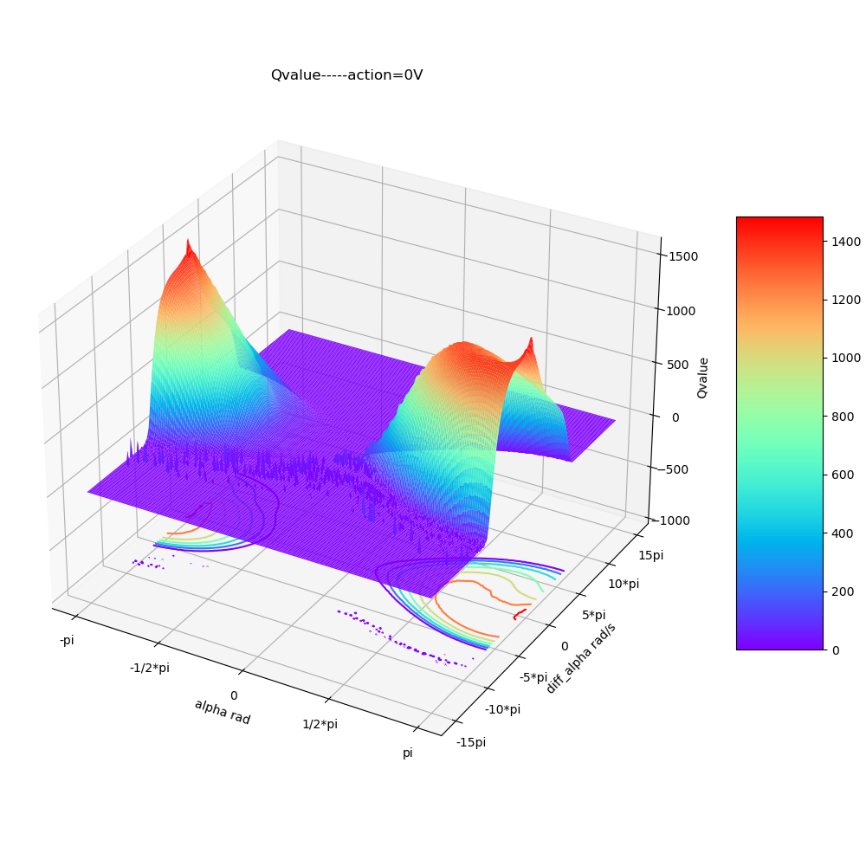


图3 Q-Learning Q动作-状态函数可视化（动作取0V）

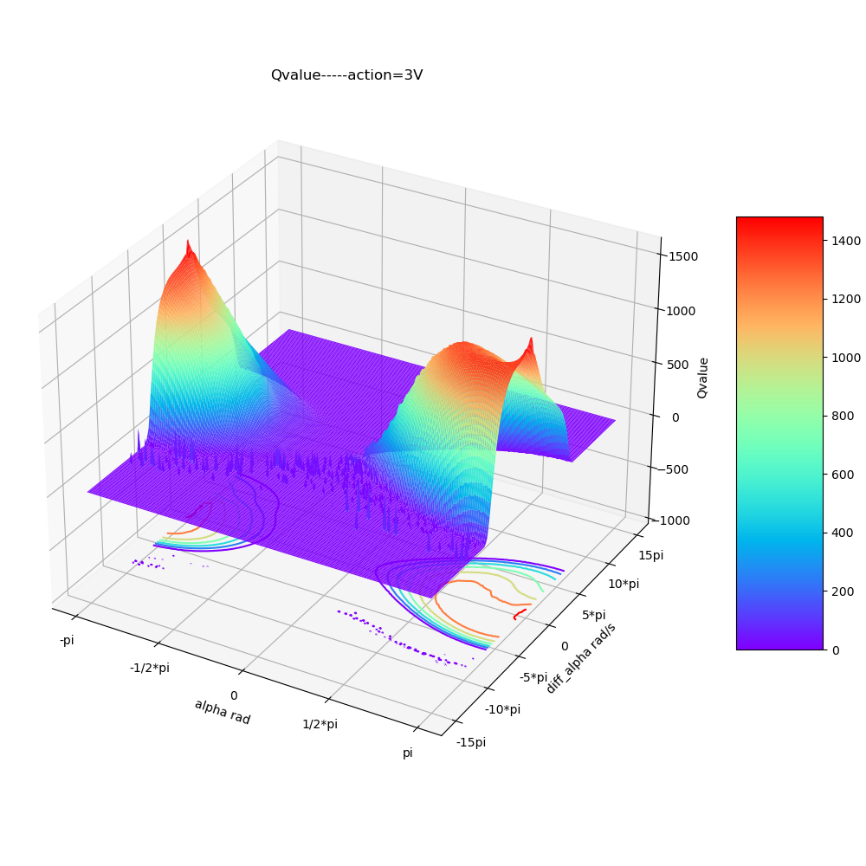


图4 Q-Learning Q动作-状态函数可视化（动作取3V）

可以看出，无论选取哪个动作，在初始状态附近（如）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（）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：

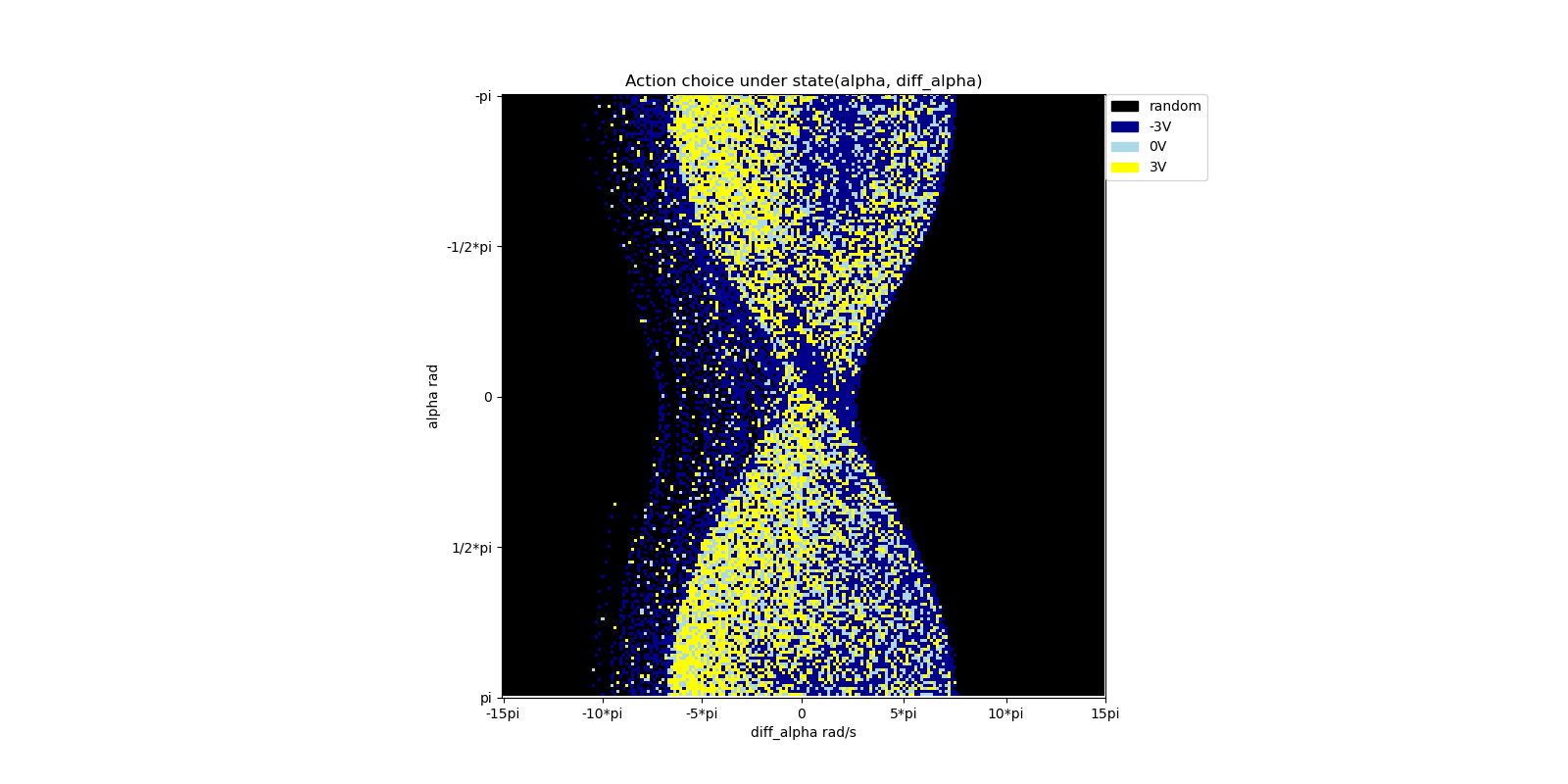


图5 Q-Learning greedy策略下的Q动作-状态函数可视化

可以看出，动作与存在明显的分界线。理论上，动作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动作。

## **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。

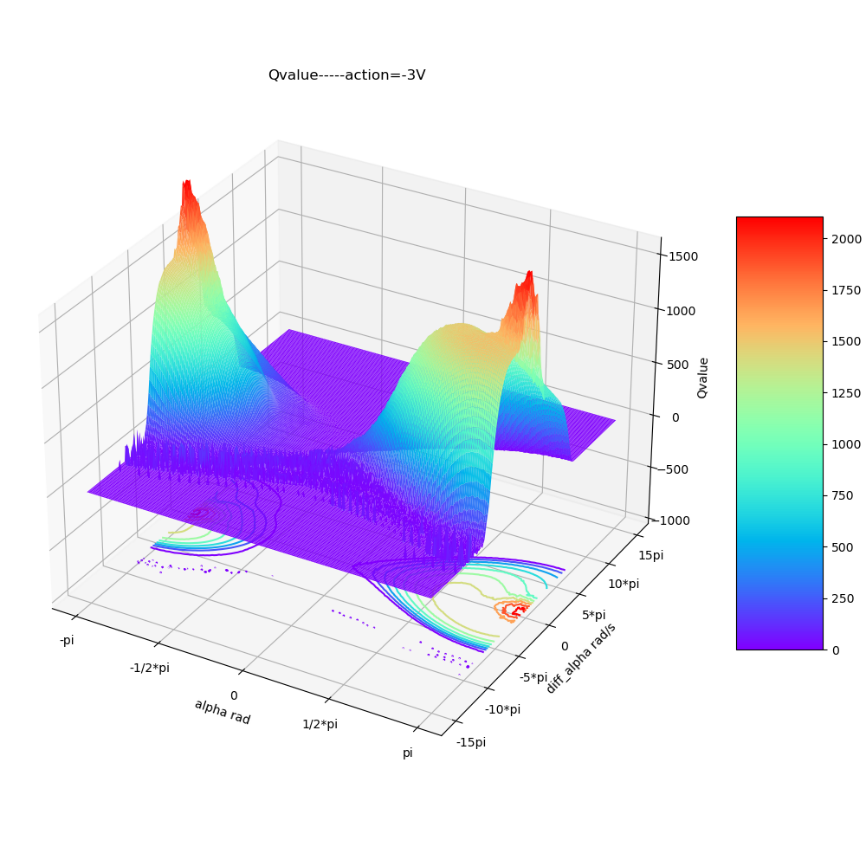


图6 SARSA Q动作-状态函数可视化（动作取-3V）

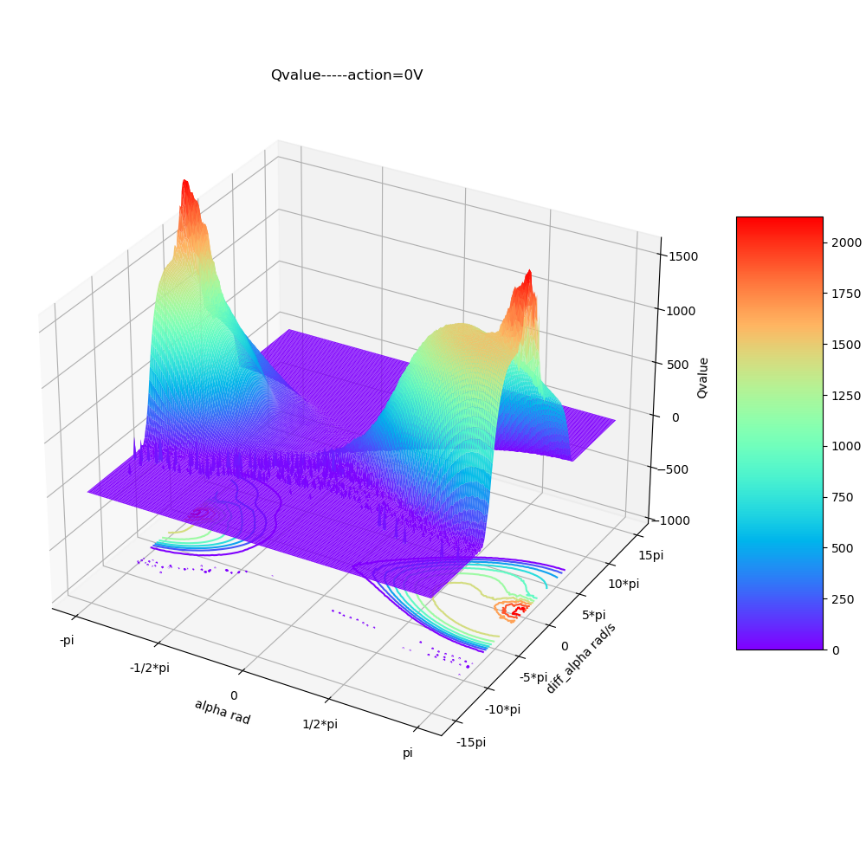


图7 SARSA Q动作-状态函数可视化（动作取0V）

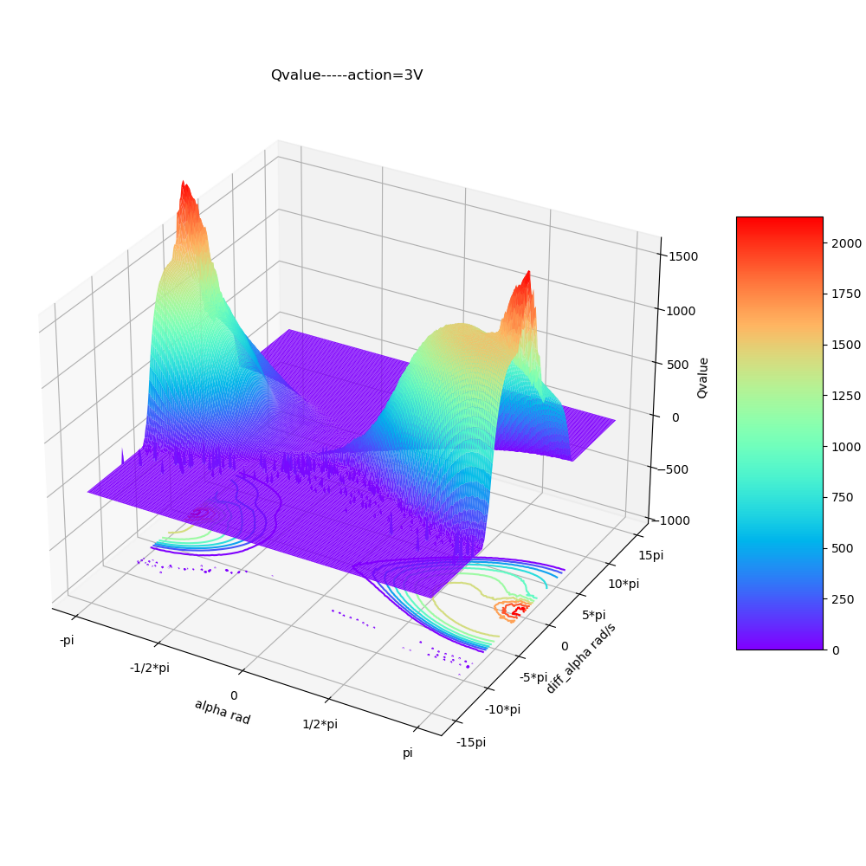


图8 SARSA Q动作-状态函数可视化（动作取3V）

可以看出，无论选取哪个动作，在初始状态附近（如）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：

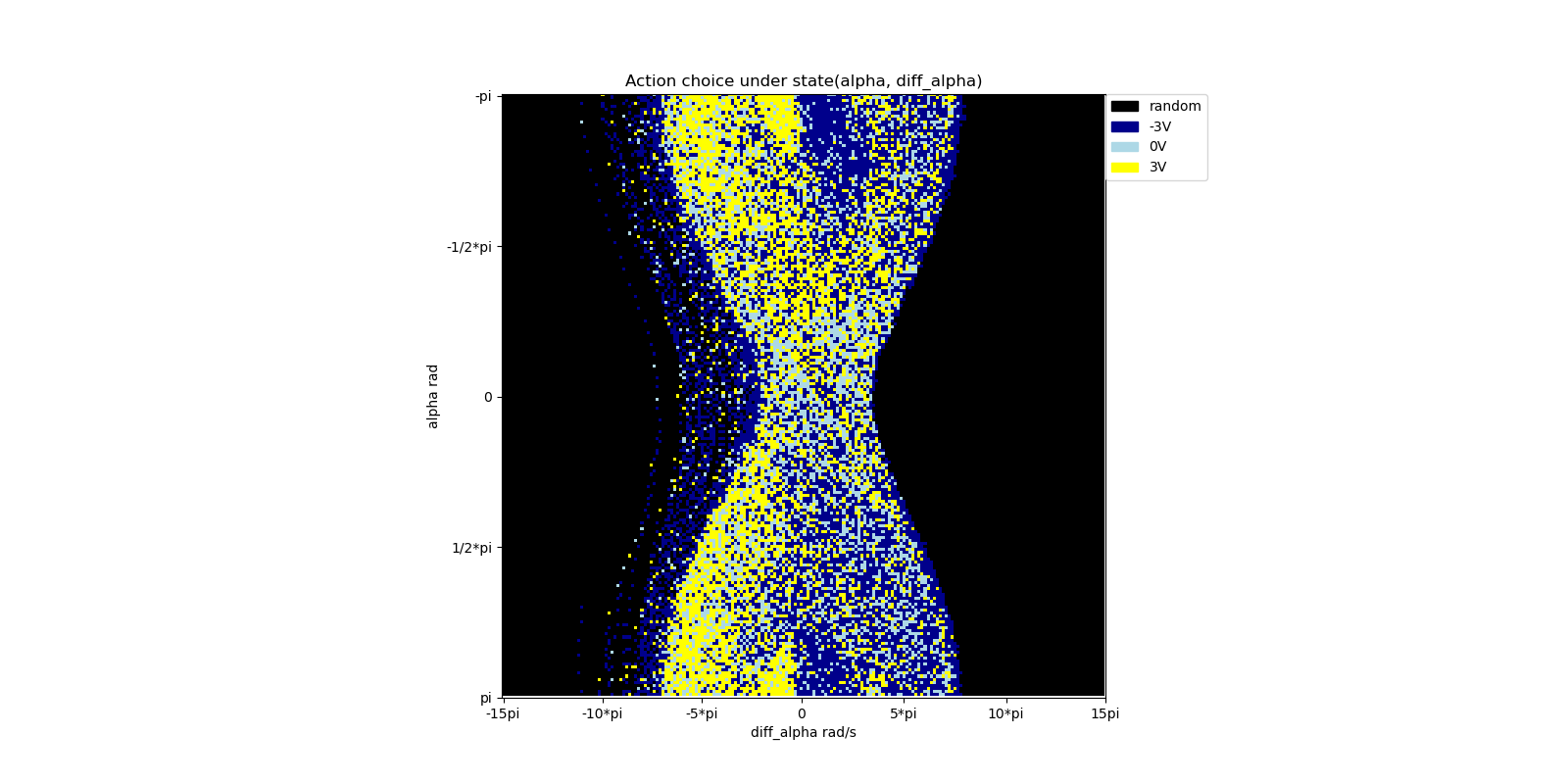


图9 Q-Learning greedy策略下的Q动作-状态函数可视化

可以看出，动作与存在明显的分界线。并且，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。

## **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。

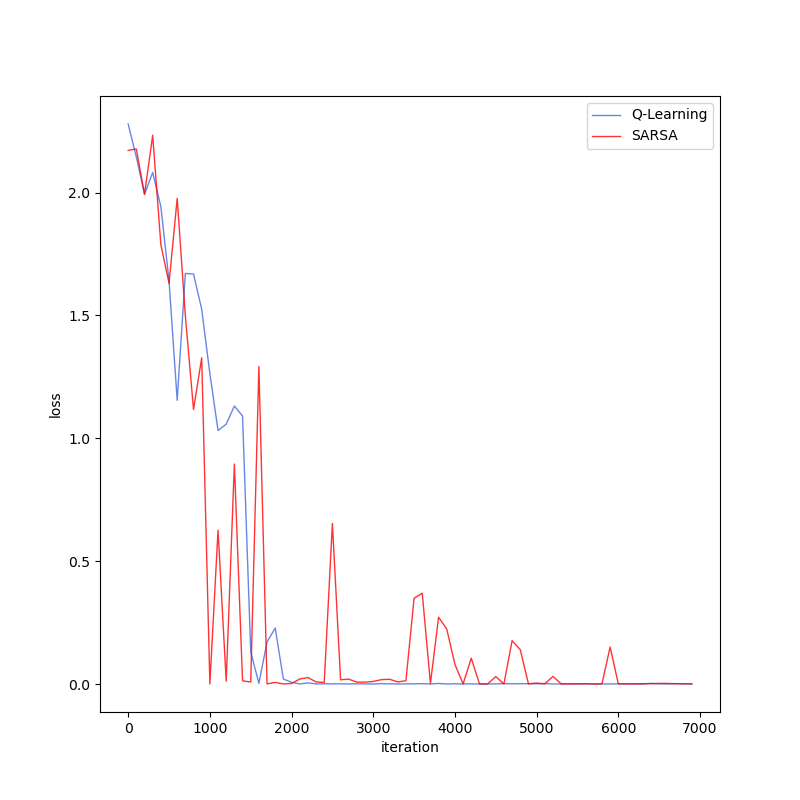


图10 Q-Learning与SARSA前7000轮迭代的loss变化

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>