**Cliff Walking实验报告**

1120210526 马煜

# 源代码架构说明

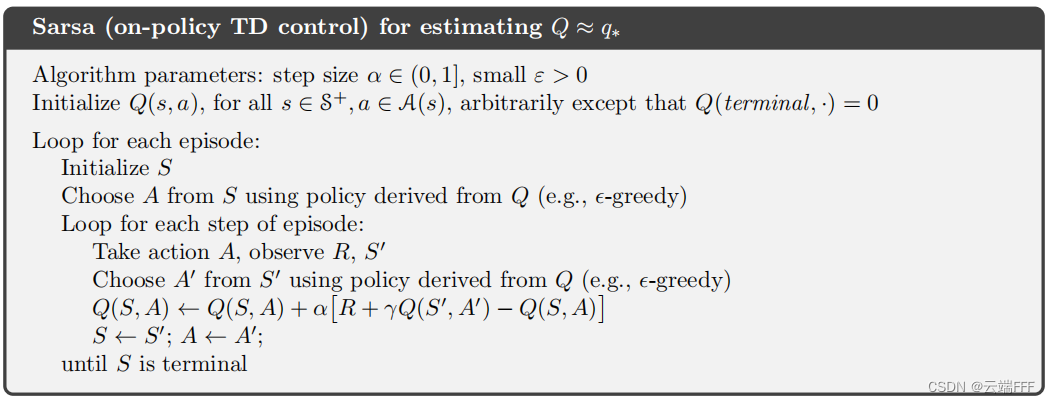
参考RLexample的代码示例，我对照重写了sarsa和Q-learning核心部分的代码，见q-learning.py和sarsa.py。

环境GridWorld和egreedy\_policy以及结果展示部分直接使用RLexample的代码示例，作为头文件引入。

# Sarsa

1. **实验过程**

sarsa是基于时序差分方法框架的一种算法，其更新Q函数的流程如下：



这段代码的核心部分是"SARSA"函数。其中Q函数的值被表示为一个表格，这个表格是一个二维数组，其维度大小分别对应于状态（state）和动作（action）的数量。在开始时，这个Q表格的所有值都被初始化为0。此外，一个名为"ep\_rewards"的列表被创建，用于存储每个episode的总奖励值。

在"SARSA"函数的主要循环中，首先每个episode开始前会调用"reset"函数，以重置环境状态。然后，根据ε-greedy策略（即以一定的概率进行探索或选择Q值最高的动作），从Q函数中选择当前状态对应的动作。这个选择过程是由名为"egreedy\_policy"的函数实现的。一旦动作被选定，它会被执行，同时根据"cliff walking"规则计算执行该动作的奖励值以及是否达到了终点（结束标志"done"）。接着，算出时间差分（TD）的目标值和误差，然后利用这些值来更新当前状态的Q函数值。最后，状态和动作被更新。

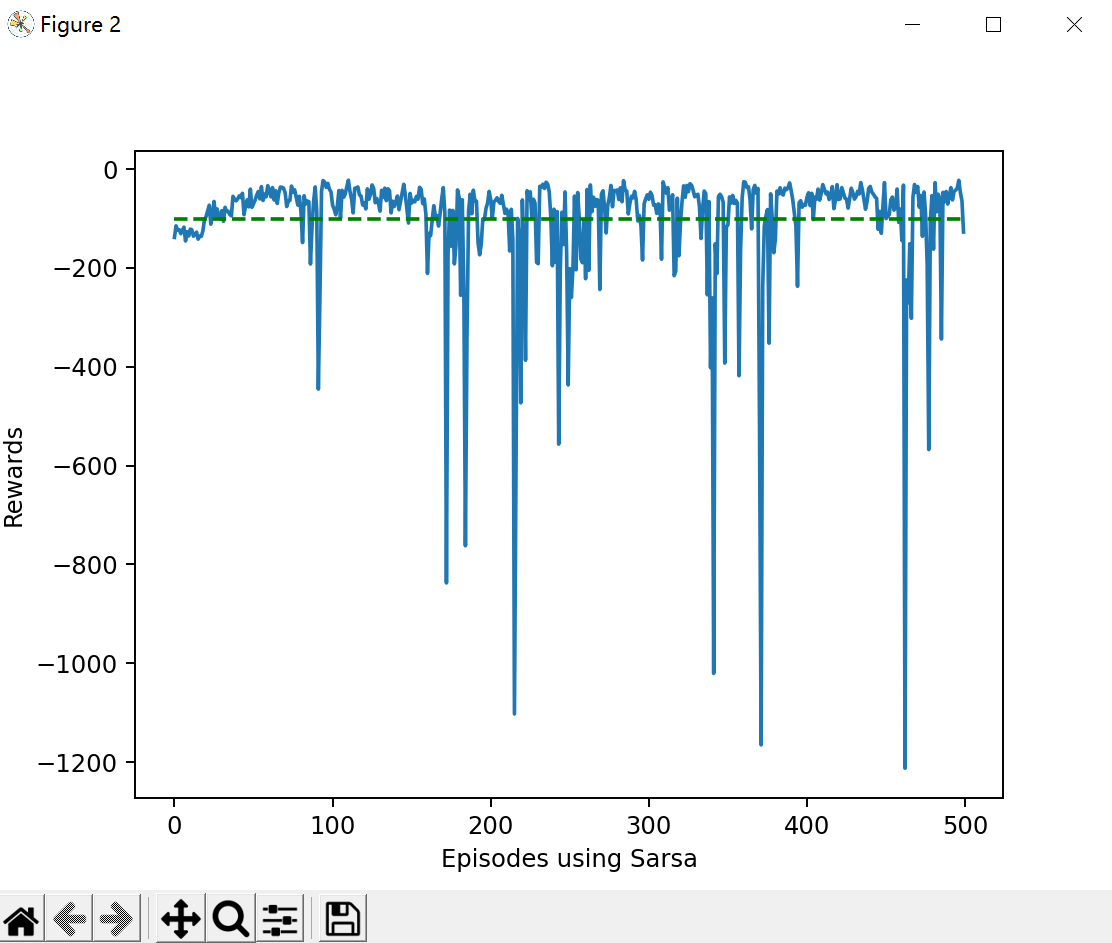
"SARSA"函数通过检查"done"来判断是否一个episode结束，如果是，它会将当前episode的总奖励值添加到"ep\_rewards"列表中。如果"render"参数设置为True，该函数还会在每次选择并执行动作后显示图形界面，以可视化整个训练过程。

最终，"SARSA"函数返回一个包含每个episode总奖励值的列表，以及最终经过训练的Q函数表格。这个Q表格包含了在训练过程中学到的值，可用于智能体在环境中做出更好的决策。

最后，调用10次sarsa函数训练以便对算法平均性能进行评估。

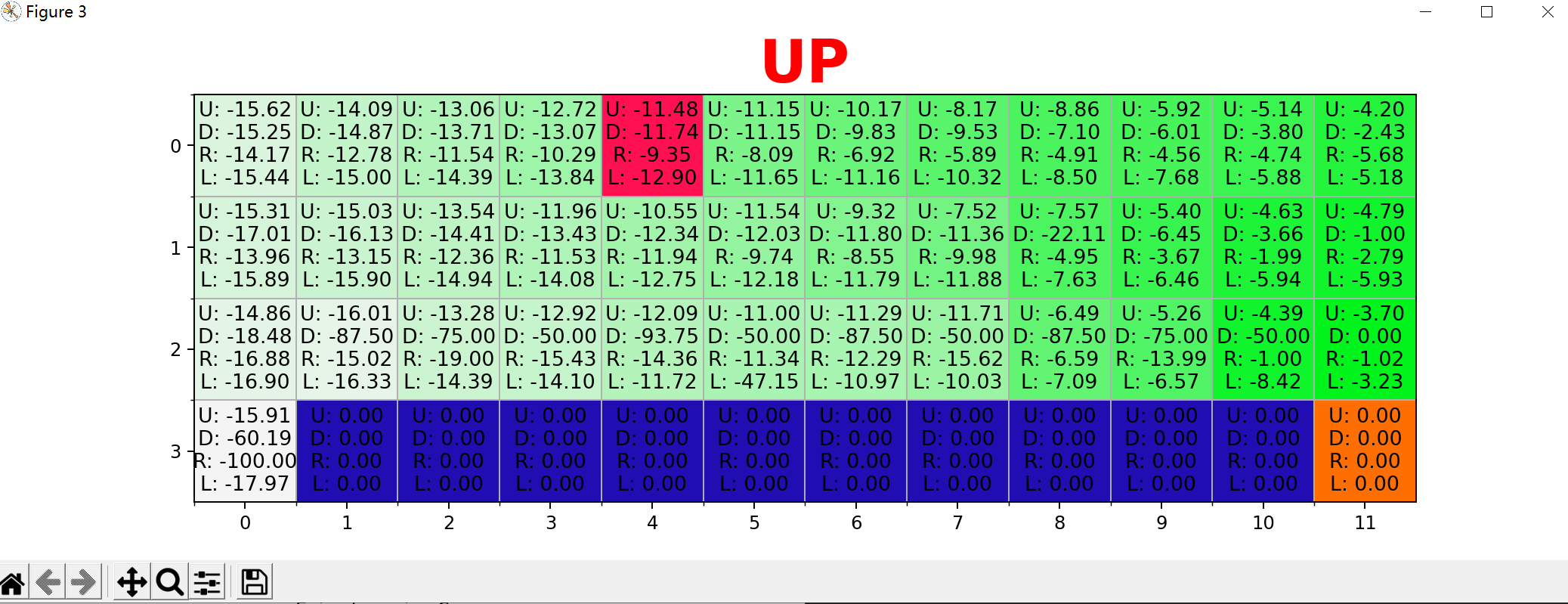
1. **实验结果及分析**

中间过程：可以看到各episode平均总奖励值如下图所示，发现奖励值收敛速度较快，但由于探索等原因经常会出现大幅波动。



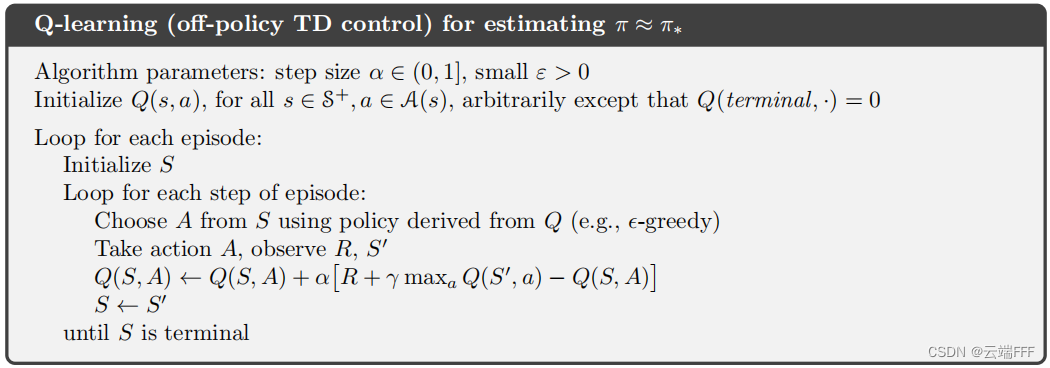
最终结果呈现：

多次运行程序发现sarsa的训练结果不太稳定,下图是一次并不是直接从起点出发直接远离悬崖的训练结果示意。



# Q-learning

1. **实验过程**Q-learning是一种基于ε-greedy策略的特殊sarsa算法，其流程见下图：



这段代码的整体结构与"sarsa.py"相似，包括GridWorld类和egreedy\_policy函数等都完全相同。它主要实现了Q-learning算法，与"SARSA"不同，Q-learning是一种off-policy算法，它在更新Q值时使用了最大估计的动作值，而不是根据实际执行的动作来更新。

具体算法请看代码注释：

def **q\_learning**(

*env*,                   *# 环境对象，表示智能体的操作环境*

*num\_episodes*=500,      *# 训练的总周期数（默认为500个周期）*

*render*=True,           *# 是否在每个周期中渲染环境（默认为True）*

*exploration\_rate*=0.1,  *# 探索率（ε-greedy策略中的ε值，默认为0.1）*

*learning\_rate*=0.5,     *# 学习率（默认为0.5）*

*gamma*=0.9,             *# 折扣因子（默认为0.9）*

):

*# 初始化Q值表，所有Q值都初始化为0*

    q\_values = np.**zeros**((num\_states, num\_actions))

*# 存储每个周期的总奖励*

    ep\_rewards = []

*# 开始训练循环，每个周期都是一个episode*

    for \_ in range(*num\_episodes*):

*# 重置环境，开始新的episode*

        state = *env*.reset()

        done = False

        reward\_sum = 0

*# 在一个episode中不断选择动作并与环境交互，直到episode结束*

        while not done:

*# 使用ε-greedy策略选择动作*

            action = **egreedy\_policy**(q\_values, state, *exploration\_rate*)

*# 执行选定的动作并观察环境的反馈*

            next\_state, reward, done = *env*.step(action)

*# 累积奖励*

            reward\_sum += reward

*# 更新Q值*

*# 根据Q-learning的更新规则计算目标Q值 (td\_target)*

            td\_target = reward + *gamma* \* np.**max**(q\_values[next\_state])

*# 计算TD误差*

            td\_error = td\_target - q\_values[state][action]

*# 使用学习率更新Q值*

            q\_values[state][action] += *learning\_rate* \* td\_error

*# 更新状态为下一个状态*

            state = next\_state

*# 如果需要，渲染当前环境状态*

            if *render*:

*env*.render(q\_values, *action*=actions[action], *colorize\_q*=True)

*# 存储本周期的总奖励*

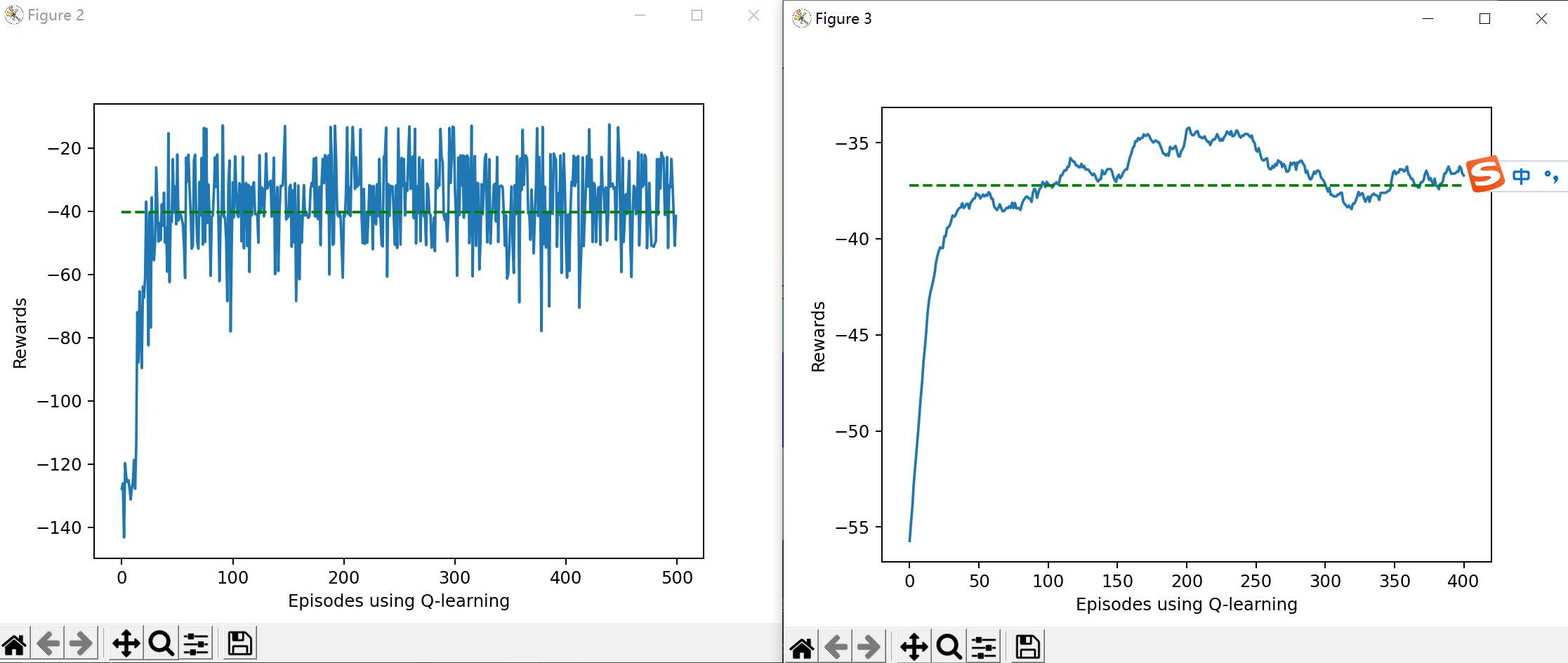
        ep\_rewards.**append**(reward\_sum)

*# 返回每个周期的总奖励列表和学习后的Q值表*

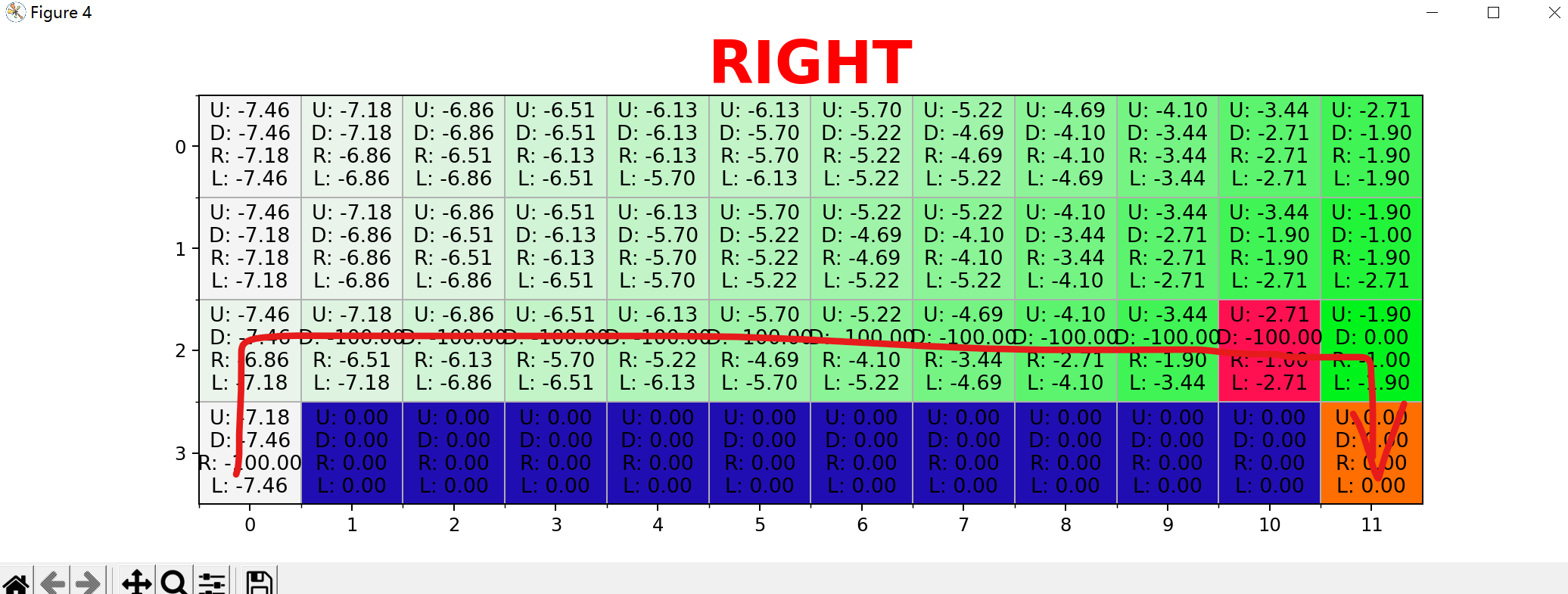
    return ep\_rewards, q\_values

1. **实验结果分析**

中间过程：可见Q-learning的奖励值在收敛后就基本稳定了，而不像sarsa那样易出现大幅改变。



最终结果：agent每次选择Q值最大的动作执行，最终结果一定是会按照最贴近cliff的一条路线执行。



# 综合对比分析

Q-learning 是一种 off-policy 算法，它优化的是一个不同于实际执行策略的策略，因此，它在更新 Q 表格时不考虑当前执行的动作。相反，Q-learning 使用已知的最大估计动作值来更新 Q 值，而不是根据实际执行的动作来更新。这意味着，Q-learning 可以选择在当前策略下不常用的动作，因为它的更新不受实际执行的策略影响。

在学习的过程中，Q-learning 会维护一个最大估计的动作值函数，以便选择最优动作。即使在某些情况下，下一步可能会采取不安全的随机动作，Q-learning 也不会受到影响，因为它不直接考虑执行的动作。这使得 Q-learning 更倾向于选择最优策略，即使在探索新动作时也会更加谨慎。

所以 Sarsa 知道它下一步的动作有可能会跑到悬崖那边去，它就会在优化自己的策略的时候，尽可能离悬崖远一点。这样子就会保证，它下一步哪怕是有随机动作，它也还是在安全区域内。