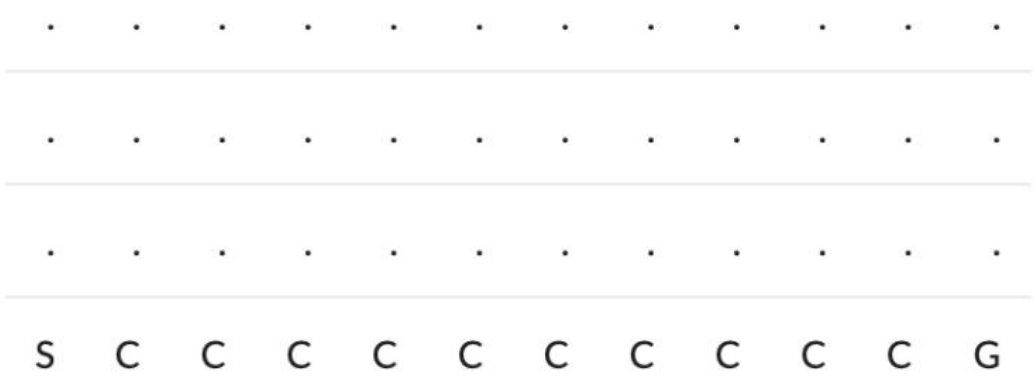
多智能体大作业

姓　　名	安佳豪
学　　号	19351002
院　　系	智能工程学院
专　　业	智能科学与技术

2021 年 12 月 10 日

1. Cliff Walking 任务

题目描述：实现策略迭代和值迭代， 完成 Cliff Walking 任务



智能体需要从左下角的起点 S 出发，避开悬崖 C，到达终点 G。动作 0 为上走， 动作 1 为右走，动作 2 为下走，动作 3 为左走。当智能体掉到悬崖时会得到-100 的奖励，当智能体走到终点时会得到 0 的奖励，否则智能体每一步都会得到-1 的奖励。代码文件 1_RL.py 中提供了 Cliff Walking 以及算法框架的实现，补充剩余内容，并描绘出最终训练到的策略函数。

① 策略迭代

思路分析

Policy Iteration (using iterative policy evaluation) for estimating $\pi \approx \pi_*$

1. Initialization
   $V(s) \in \mathbb{R}$ and $\pi(s) \in \mathcal{A}(s)$ arbitrarily for all $s \in \mathcal{S}$

2. Policy Evaluation
   Loop:
   $\Delta \leftarrow 0$
   Loop for each $s \in \mathcal{S}$:
   $v \leftarrow V(s)$
   $V(s) \leftarrow \sum_{s',r} p(s', r|s, \pi(s))\left[r + \gamma V(s')\right]$
   $\Delta \leftarrow \max(\Delta, |v - V(s)|)$
   until $\Delta < \theta$ (a small positive number determining the accuracy of estimation)

3. Policy Improvement
   policy-stable $\leftarrow$ *true*
   For each $s \in \mathcal{S}$:
   old-action $\leftarrow \pi(s)$
   $\pi(s) \leftarrow \arg\max_a \sum_{s',r} p(s', r|s, a)\left[r + \gamma V(s')\right]$
   If *old-action* $\neq \pi(s)$, then *policy-stable* $\leftarrow$ *false*
   If *policy-stable*, then stop and return $V \approx v_*$ and $\pi \approx \pi_*$; else go to 2

策略迭代的整体思路是，策略评估和策略提升这两个过程的不断循环交替，直至

最后得到最优策略，策略评估和策略提升两个过程需要做的事情已经在上图给出。所以我们需要做的任务是编写策略评估和策略提升两个函数，然后利用这两个函数实现策略迭代，从而解决任务。

代码实现

A. 策略评估函数

```
# 策略评估函数
def policy_valuation(self):
    # 一些参数的设置
    threshold = 1e-10
    gamma = 0.9
    while True:
        new_value_table = np.copy(self.V)   # 赋值生成一个新的值列表
        # 循环，遍历所有的状态
        for i in range(4):
            for j in range(12):
                action = self.PI[i][j]  # 返回当前策略当前状态下对应的动作
                next_i, next_j, reward = env.step(i, j, action)  # 返回当前状态下执行动作得到的下一状态和奖励
                self.V[i][j] = reward + gamma * new_value_table[next_i][next_j]  # 计算策略下的状态价值
        # 如果两个更新之间的差值小于阈值，则退出循环
        if np.sum((np.fabs(new_value_table - self.V))) <= threshold:
            break
    return self.V
```

可以将该代码和上面思路分析中得到的流程图进行对照看，可以看到策略评估的**整体流程**是遍历所有的状态，然后执行当前策略下对应状态的动作，将得到的下一状态和奖励带入到公式中进行计算得出值，然后更新值函数列表。关于代码中的一些**具体细节**可以看上面的注释内容，已经很详细了。

B. 策略提升函数

```
# 策略提升函数
def policy_improvement(self):
    # 参数的设定
    gamma = 0.9
    # 循环，遍历所有的状态
    for i in range(4):
        for j in range(12):
            # 创建列表存储当前状态下执行不同动作的价值
            action_table = np.zeros(4)
            # 循环，遍历所有的动作
            for action in range(4):
                next_i, next_j, reward = env.step(i, j, action) # 返回当前状态执行动作得到的下一状态及奖励
                action_table[action] = reward + gamma * self.V[next_i][next_j]  # 计算当前状态下执行该动作获得的奖励
            # 策略提升，选取获取奖励最大的动作更新策略
            self.PI[i][j] = np.argmax(action_table)
    return self.PI
```

这里是策略提升函数，策略提升函数的**整体流程**是遍历所有的状态，然后遍历循环每个状态下对应的四种动作，获取执行动作后的下一状态和奖励，并利用得到的值来计算当前状态下执行当前动作的状态值，选取其中最大的来作为策略更新。同样**具体细节**可以看上面的注释。

C. 策略迭代

```python
def learn(self):
    # Implement your code here
    # ...
    # 循环，交错调用策略评估和策略提升函数
    while True:
        last_policy = np.copy(self.PI)
        self.policy_valuation()
        self.policy_improvement()
        # 如果前后两次的策略没有更新，表示已经收敛，所以退出循环
        if(np.all(last_policy == self.PI)):
            print('策略迭代结束')
            break
    print(self.PI)
```

这里是策略迭代函数，**整体流程**就是不断的循环交错使用策略评估和策略提升函数，同时我们保存了上一次循环下的策略，当这一次循环后发现策略并没有获得更新，说明我们的策略已经到达了收敛的状态，故退出循环。**具体细节**同样参见代码注释。

到这里，我们策略迭代就已经完全实现了，在主函数中调用策略迭代函数就可以得到最后的结果了。

D. 主函数调用

```python
if __name__ == '__main__':
    np.random.seed(0)
    env = CliffWalking()

    PI = PolicyIteration(env)
    PI.learn()
```

主函数中调用策略迭代函数

实验结果

```
hon\debugpy\launcher' '53235' '--
策略迭代结束
[[1 1 1 1 1 1 1 1 1 1 1 2]
 [1 1 1 1 1 1 1 1 1 1 1 2]
 [1 1 1 1 1 1 1 1 1 1 1 2]
 [0 0 0 0 0 0 0 0 0 0 1 1]]
```

上面输出的东西是我们最后得到的最优策略，从左下角的出发点跟着上面的指示得到策略指令为[0 1 1 1 1 1 1 1 1 1 1 2]。
做了一个简单的可视化如下：

代码

```
go = {0:'↑', 1:'→', 2:'↓', 3:'←'}
start_pos_x = 3
start_pos_y = 0
list = []
while True:
    st = PI.PI[start_pos_x][start_pos_y]
    list.append(go[st])
    start_pos_x, start_pos_y, re = env.step(start_pos_x, start_pos_y, st)
    if(start_pos_x ==3 and start_pos_y==11):
        break
print(list)
```

可视化结果

```
策略迭代结束
[[1 1 1 1 1 1 1 1 1 1 1 2]
 [1 1 1 1 1 1 1 1 1 1 1 2]
 [1 1 1 1 1 1 1 1 1 1 1 2]
 [0 0 0 0 0 0 0 0 0 0 1 1]]
['↑', '→', '→', '→', '→', '→', '→', '→', '→', '→', '→', '→', '↓']
```

② **值迭代**

思路分析

初始化：$V(s)=0\ \forall s$

循环迭代：$i=0,1,2,\ldots\ldots,N$

$$V(s)=\max_a\left(R_s^a+\gamma\sum_{s'}P_a(s,s')V(s')\right)$$

若$\forall s$，$V(s)$收敛，停止循环

输出：最佳策略π_i

$$\pi_i(s)=\arg\max_a\left(R_s^a+\gamma\sum_{s'}P_a(s,s')V(s')\right)$$

值迭代的原理和策略迭代类似，不同的是值迭代主要思路为在迭代过程中只更新值函数，迭代完成后通过构造最佳策略，值迭代不需要想策略迭代一样分成两个函数，相反值迭代全部的都在一个函数中完成，对于值状态和策略状态的更新放在不同的循环中即可。我个人是把值迭代看成策略迭代的简易版。（因为一个函数解决问题）

代码实现

```
def learn(self):
    # Implement your code here
    # ...
    # 一些基本参数的设置
    threshold = 1e-20
    gamma = 0.9
    # 不断循环
    while True:
        # 创建每次迭代更新的状态价值表
        new_value_table = np.copy(self.V)
        # 循环，遍历所有状态
        for i in range(4):
            for j in range(12):
                # 创建空的动作价值列表
                action_value = np.zeros(4)
                # 循环，遍历所有动作
                for action in range(4):
                    # 返回当前状态-动作下一步的状态和奖励
                    next_x, next_y, reward = env.step(i, j, action)
                    # 计算动作的累积期望奖励
                    action_value[action] = reward + gamma * new_value_table[next_x][next_y]
                self.V[i][j] = max(action_value)  # 更新状态值表
                self.PI[i][j] = np.argmax(action_value)  # 记录当前最佳策略
        # 价值表前后两次更新之差小于阈值时停止循环
        if np.sum((np.fabs(new_value_table - self.V))) <= threshold:
            print('值迭代结束，值迭代最优策略如下')
            break
    print(self.PI)
```

上面是值迭代的代码实现，**整体流程**为首先初始化状态价值表，然后遍历所有状态，之后创建空数组保存最佳策略，遍历动作后求出最佳策略，对状态值和策略进行更新。每次迭代中判断价值表前后两次更新之差是否小于阈值，若小于则认为迭代收敛，停止迭代。**具体细节**参见代码注释。

主函数调用

```
VI = ValueIteration(env)
VI.learn()
```

主函数中调用值迭代，打印出最优策略。

实验结果

```
hon\debugpy\launcher" "58696" -- "d:\大三上\多智能体\大作业\1_RL.py"
策略迭代结束，策略迭代最优策略如下
[[1 1 1 1 1 1 1 1 1 1 1 2]
 [1 1 1 1 1 1 1 1 1 1 1 2]
 [1 1 1 1 1 1 1 1 1 1 1 2]
 [0 0 0 0 0 0 0 0 0 0 1 1]]
策略迭代路线图如下
['↑', '→', '→', '→', '→', '→', '→', '→', '→', '→', '→', '↓']
值迭代结束，值迭代最优策略如下
[[1 1 1 1 1 1 1 1 1 1 1 2]
 [1 1 1 1 1 1 1 1 1 1 1 2]
 [1 1 1 1 1 1 1 1 1 1 1 2]
 [0 0 0 0 0 0 0 0 0 0 1 1]]
值迭代路线图如下
['↑', '→', '→', '→', '→', '→', '→', '→', '→', '→', '→', '↓']
```

从左下角的出发点开始，到达右下角的终点。值迭代给出的策略指令也为[0 1 1 1 1 1 1 1 1 1 1 2]，可视化结果也和策略迭代一致。

至此，第一大题撒花完结。

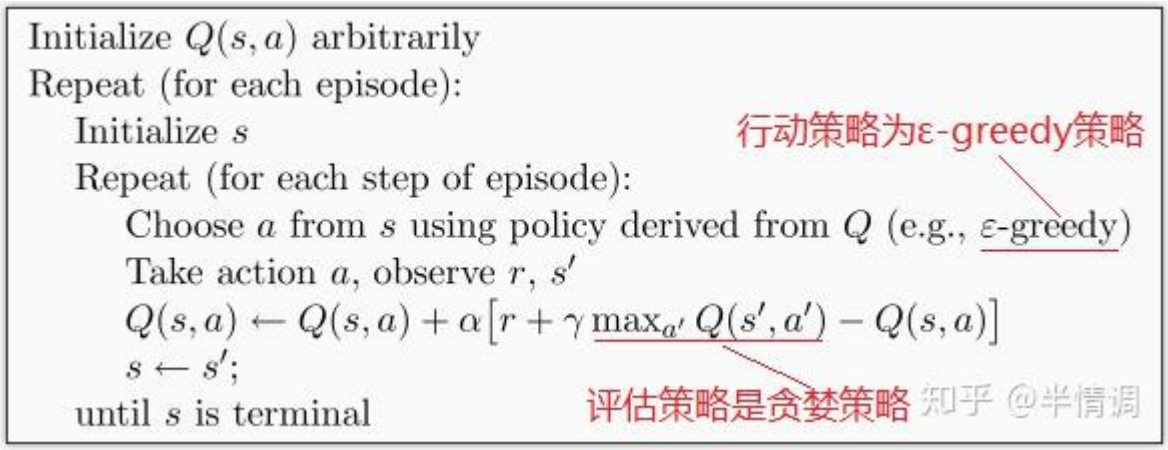
2. Cliff Walking 任务

（是的，第二大题的标题还是它）

题目描述：实现 Q-learning 以及 SARSA 算法，完成悬崖行走（Cliff Walking）任务。画出两种算法在训练过程中的奖励曲线，画出两种算法在任务中的行动轨迹，分析两者的不同并解释原因

① Q-learning 算法

思路分析



Q-Learning 的目的是学习特定 State 下、特定 Action 的价值。是建立一个 Q-Table，以 State 为行、Action 为列，通过每个动作带来的奖赏更新 Q-Table。Q-learning 的基本思路是先假设自己下一步选取最大奖赏的动作，然后更新值函数，再根据策略来选取动作。

代码实现

A. select_action 函数

```
def select_action(self, obs, if_train=True):
    # Implement your code here
    # ...
    if np.random.uniform(0, 1) < (1.0 - self.epsilon):  #根据table的Q值选动作
        Q_list = self.Q[obs, :]  # 从Q表中选取状态(或观察值)对应的那一行
        maxQ = np.max(Q_list)  # 获取这一行最大的Q值
        action_list = np.where(Q_list == maxQ)[0]  # np.where找出最大值所在的位置
        action = np.random.choice(action_list)  # 选取最大值对应的动作
    else:
        action = np.random.choice(self.act_n)  #有一定概率随机探索选取一个动作
    return action
```

该函数的**整体流程**是有一个判断，有两种情况，一种是更具 table 中的 Q 值来选取下一步动作，另一种情况是随机选取一个动作。这两种情况发生的概率取决于我们设定的 epsilon，**具体细节**参见注释。

B. Update 函数

```
def update(self, transition):
    obs, action, reward, next_obs, done = transition
    # Implement your code here
    # ...
    predict_Q = self.Q[obs, action]
    if done:
        target_Q = reward  # 如果到达终止状态， 没有下一个状态了，直接把奖励赋值给target_Q
    else:
        target_Q = reward + self.gamma * np.max(self.Q[next_obs, :])
    self.Q[obs, action] += self.lr * (target_Q - predict_Q)
```

这个函数实现的功能是对 Q-table 进行更新，更新的原则可以对照上面图中的更新公式。

$$Q(s,a) \leftarrow Q(s,a) + \alpha\left[r + \gamma \max_{a'} Q(s',a') - Q(s,a)\right]$$

C. Q_learning_train 函数

```
def q_learning_train(args):
    env = args.env
    agent = args.agent
    episodes = args.episodes
    max_step = args.max_step
    rewards = []
    mean_100ep_reward = []
    for episode in range(episodes):
        episode_reward = 0  # 记录一个episode获得的总奖励
        # Implement your code here
        # ...
        obs = env.reset()  # 重置环境，重新开始新的一轮
        for t in range(max_step):
            # Implement your code here
            # ...
            action = agent.select_action(obs)  # 选取一个动作
            next_obs, reward, done, _ = env.step(action)
            agent.update((obs, action, reward, next_obs, done))
            obs = next_obs
            episode_reward += reward
            if done: break
        print(f'Episode {episode}\t Step {t}\t Reward {episode_reward}')
        rewards.append(episode_reward)
        if len(rewards) < 100:
            mean_100ep_reward.append(np.mean(rewards))
        else:
            mean_100ep_reward.append(np.mean(rewards[-100:]))
    return mean_100ep_reward
```

这个函数实现的功能是对 Q-learning 算法进行训练，我们补充的内容为红框部分，一个是再循环前对环境进行重置，下面的红框部分则是选取一个动作，将该动作带入 step 中，更新 Q-label 并计算一个 episode 中的总奖励。

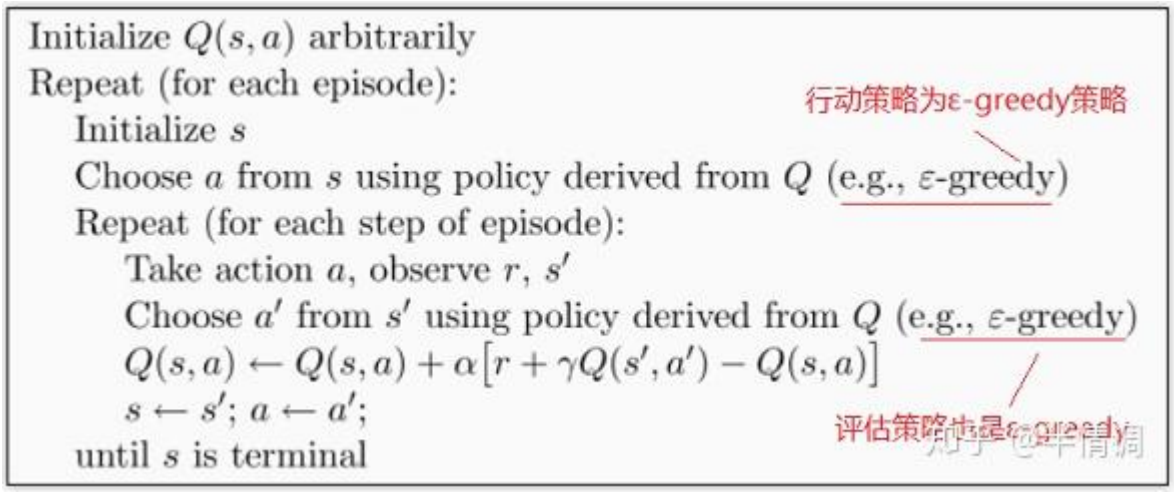
D. Q_learning_test 函数

```python
def q_learning_test(args):
    # Implement your code here
    # ...
    env = args.env
    agent = args.agent
    total_reward = 0
    obs = env.reset()
    while True:
        Q_list = agent.Q[obs, :]
        maxQ = np.max(Q_list)
        action_list = np.where(Q_list == maxQ)[0]
        action = np.random.choice(action_list)    # 获取下一个动作
        next_obs, reward, done, _ = env.step(action)  # 得到奖励reward和done
        total_reward += reward
        obs = next_obs
        env.render()  # 输出渲染
        if done:
            break
```

这个函数是测试函数，它的主体架构和训练函数是相同的，在这个函数中我们使用了 render 函数渲染观察物体的运动轨迹。最后在 done 时，退出循环。

② SARSA 算法

思路分析



SARSA 也是采用 Q-table 的方式存储动作值函数；而且决策部分和 Q-Learning 是一样的，也是采用ε-greedy 策略。不同的地方在于 Sarsa 的更新方式是不一样的。Sarsa 是 on-policy 的更新方式，它的行动策略和评估策略都是ε-greedy 策略。Sarsa 是先做出动作后更新。

代码实现

Select_action 函数，uodate 函数等与 Q-learning 算法相同就不再赘述，下面分析不同的部分。

A. Sarsa_train 函数

```python
def sarsa_train(args):
    env = args.env
    agent = args.agent
    episodes = args.episodes
    max_step = args.max_step
    rewards = []
    mean_100ep_reward = []
    for episode in range(episodes):
        episode_reward = 0  # 记录一个episode获得的总奖励
        # Implement your code here
        # ...
        obs = env.reset()  # 重置环境，重新开始新的一轮
        action = agent.select_action(obs)  # 根据算法选取一个动作
        for t in range(max_step):
            # Implement your code here
            # ...
            next_obs, reward, done, info = env.step(action)  # 将action作用于环境并得到反馈
            next_action = agent.select_action(next_obs)  # 根据下一状态，获取下一动作
            # 训练SARSA算法，更新Q表格
            agent.update((obs, action, reward, next_obs, next_action, done))
            action = next_action
            obs = next_obs  # 存储上一次观测值
            episode_reward += reward
            if done: break
        print(f'Episode {episode}\t Step {t}\t Reward {episode_reward}')
        rewards.append(episode_reward)
        if len(rewards) < 100:
            mean_100ep_reward.append(np.mean(rewards))
        else:
            mean_100ep_reward.append(np.mean(rewards[-100:]))
    return mean_100ep_reward
```

红框是我们补充的代码，黄框中特意标出来的是 SARSA 算法与 Q-learning 算法不同的地方，这里 action 的选取放在了前面，充分地体现出该算法是先做动作后更新。

运行结果

```
Episode 381      Step 16         Reward -116
Episode 382      Step 18         Reward -19
Episode 383      Step 16         Reward -17
Episode 384      Step 14         Reward -15
Episode 385      Step 16         Reward -17
Episode 386      Step 14         Reward -15
Episode 387      Step 14         Reward -15
Episode 388      Step 16         Reward -17
Episode 389      Step 18         Reward -19
Episode 390      Step 20         Reward -21
Episode 391      Step 16         Reward -17
Episode 392      Step 14         Reward -15
Episode 393      Step 21         Reward -22
Episode 394      Step 14         Reward -15
Episode 395      Step 14         Reward -15
Episode 396      Step 14         Reward -15
Episode 397      Step 18         Reward -19
Episode 398      Step 14         Reward -15
Episode 399      Step 14         Reward -15
```

这个输出结果是训练函数中的下面命令的输出结果

```python
print(f'Episode {episode}\t Step {t}\t Reward {episode_reward}')
```

我们设定的 episode 为 400，所以一共循环输出了两个 400。

```
o  o  o  o  o  o  o  o  o  o  o  o
o  o  o  o  o  o  o  o  o  o  o  o
x  o  o  o  o  o  o  o  o  o  o  o
o  C  C  C  C  C  C  C  C  C  C  T

o  o  o  o  o  o  o  o  o  o  o  o
o  o  o  o  o  o  o  o  o  o  o  o
o  x  o  o  o  o  o  o  o  o  o  o
o  C  C  C  C  C  C  C  C  C  C  T

o  o  o  o  o  o  o  o  o  o  o  o
o  o  o  o  o  o  o  o  o  o  o  o
o  o  x  o  o  o  o  o  o  o  o  o
o  C  C  C  C  C  C  C  C  C  C  T

o  o  o  o  o  o  o  o  o  o  o  o
o  o  o  o  o  o  o  o  o  o  o  o
o  o  o  x  o  o  o  o  o  o  o  o
o  C  C  C  C  C  C  C  C  C  C  T

o  o  o  o  o  o  o  o  o  o  o  o
o  o  o  o  o  o  o  o  o  o  o  o
o  o  o  o  x  o  o  o  o  o  o  o
o  C  C  C  C  C  C  C  C  C  C  T

o  o  o  o  o  o  o  o  o  o  o  o
o  o  o  o  o  o  o  o  o  o  o  o
o  o  o  o  o  x  o  o  o  o  o  o
o  C  C  C  C  C  C  C  C  C  C  T
```

这个输出结果是测试函数中下面这个命令带来的输出

```
env.render()  # 输出渲染
```

输出结果中的 x 表示我们轨迹运动变化，上面只截取了一部分，当我们观察完全部的轨迹运动图后发现。

Q-Learning 算法的运动轨迹如下

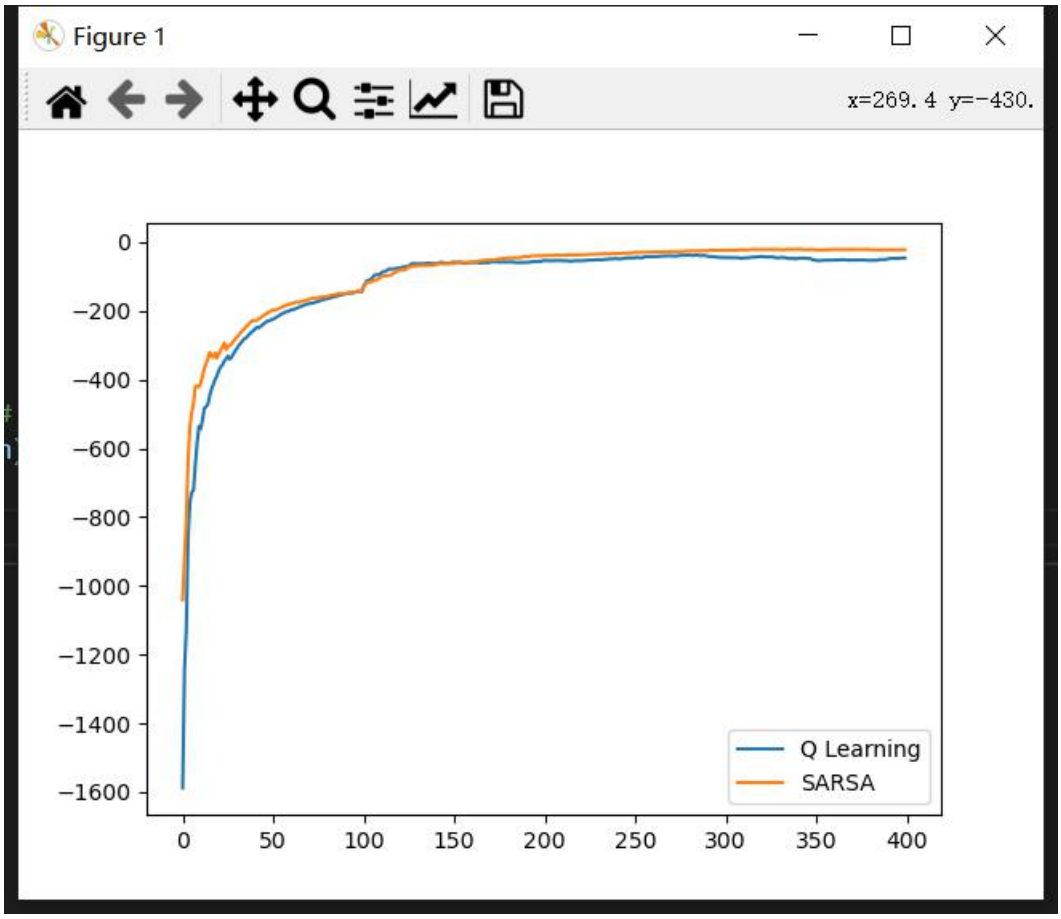
SARSA 算法的运动轨迹如下

两个算法得到的最优策略是不同的。

分析原因

这时因为 Sarsa 更新 Q 值的策略为，其产生数据的策略和更新 Q 值的策略相同，即属于 on-policy 算法；而 Q-learning 更新 Q 值的策略为贪婪策略，其产生数据的策略和更新 Q 值的策略不同，即属于 off-policy 算法；对于 Sarsa 算法而言，它的迭代速度较慢，它选择的路径较长但是相对比较安全，因此每次迭代的累积奖励也比较多，对于 Q-leaning 而言，它的迭代速度较快，由于它每次迭代选择的是贪婪策略因此它更有可能选择最短路径，不过这样更容易掉入悬崖，因此每次迭代的累积奖励也比较少。

两种算法的奖励曲线如下



从上面的奖励曲线也可以看出 SARSA 算法相比于 Q-learning 算法迭代速度较慢的，同时在大多数情况下 SARSA 算法的奖励值是略大于 Q-learning 的这也验证了**对于 Sarsa 算法而言，它的迭代速度较慢，它选择的路径较长但是相对比较安全，因此每次迭代的累积奖励也比较多，对于 Q-leaning 而言，它的迭代速度较快，由于它每次迭代选择的是贪婪策略因此它更有可能选择最短路径，不过这样更容易掉入悬崖，因此每次迭代的累积奖励也比较少。**