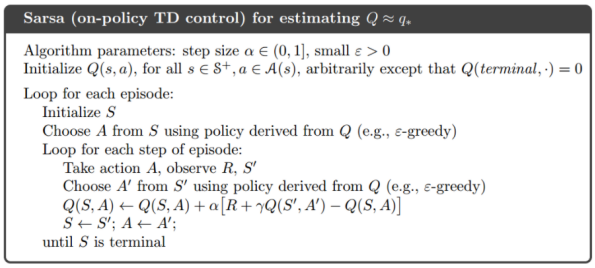
高级算法 作业3

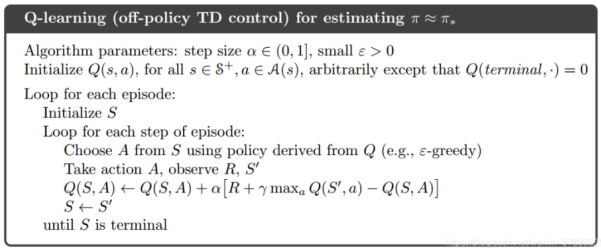
> 姓名：TRY
>
> 学号：
>
> 专业：计算机科学与技术

1. 实验原理

- $Sarsa$**算法**
  - **伪代码**：



  - **公式解释**：先基于当前状态`S`，使用`ϵ`-贪婪法按一定概率选择动作`A`，然后得到奖励`R`，并更新进入新状态`S'`，基于状态`S'`，使用`ϵ`-贪婪法选择`A'`（即online-policy，仍然使用同样的ϵ-贪婪）。
  - **算法分析**：建立一个`Q Table`来保存状态`S`和将会采取的所有动作`A`，`Q(S,A)`。在每个回合中，先随机初始化第一个状态，再对回合中的每一步都先从Q Table中使用`ϵ`-贪婪基于当前状态`S`选择动作`A`，执行`A`，然后得到新的状态`S'`和当前奖励`R`， 同时再使用`ϵ`-贪婪得到在`S'`时的`A'`，直接利用`A'`更新表中`Q(S,A)`的值，继续循环到终点。
- $Q\_learning$**算法**
  - **伪代码**：



  - **公式解释**： 先基于当前状态`S`，使用`ϵ`-贪婪法按一定概率选择动作`A`，然后得到奖励`R`，并更新进入新状态`S'`，基于状态`S'`，直接使用贪婪法从所有的动作中选择最优的`A'`（即offline-policy，不是用同样的`ϵ`-贪婪）。
  - **算法分析**：建立一个`Q Table`来保存状态`S`和将会采取的所有动作`A`，`Q(S,A)`。在每个回合中，先随机初始化第一个状态，再对回合中的每一步都先从Q Table中使用`ϵ`-贪婪基于当前状态`S`选择动作`A`，执行`A`，然后得到新的状态`S'`和当前奖励`R`，再在状态`S'`下选择最优的`A'`，同时更新表中`Q(S,A)`的值，继续循环到终点。整个算法就是一直不断更新`Q table`里的值，再根据更新值来判断要在某个 state 采取怎样的 action最好。

- $on-policy$和$off-policy$的区别：
  - 判断$on-policy$和$off-policy$在于估计时所用的策略与更新时所用的策略是否为同一个策略。$Sarsa$选了什么动作来估计Q值就一定会用什么动作来更新state，一定会执行该动作（会有贪婪率），属于on-policy；而$Q-Learning$则不然，估Q值时选max的，而执行动作时使用了ϵ-贪婪，即使用了两套策略，属于off-policy。

2. 实现思路

- 首先，设置`take_action`函数，用于返回在`(x,y)`处选取action `a`后的下一个位置和reward。其中，`x`的范围是`[0,11]`，`y`的范围是`[0,3]`。
  - 这里的`a`的取值范围是`[0,3]`，代表上、右、下、左；并在函数内部设置了`act_move`数组，用来更新`x`,`y`。

```
def take_action(x, y, a):   # 返回才取action a之后的位置，和reward
    destination = False
    if x == width - 1 and y == 0:
        destination = True
    act_move = [[0, 1], [1, 0], [0, -1], [-1, 0]]   # means up,right,down,
left
    temp_x = x + act_move[a][0]
    temp_y = y + act_move[a][1]
    if 0<=temp_x<width and 0<=temp_y<height:
        x = temp_x
        y = temp_y
    if destination is True:
        return x, y, 0      # q(terminal_state) = 0
    if 0 < x < width-1 and y==0:
        return 0, 0, -100
    else:
        return x, y, -1
```

- **max_q函数**：返回`(x,y)`状态下可采取获得的最大利润的action的下标；**epsilon_policy函数**：利用ϵ-policy选取下一个动作。

```
def max_q(x, y, q):     # 返回(x,y)状态下可采取获得的最大利润的action的下标
    max_q = q[x][y][0]
    max_action = 0
    for i in range(1, 4):  # [1,4)
        if q[x][y][i] >= max_q:
            max_q = q[x][y][i]
            max_action = i
    return max_action


def epsilon_policy(x, y, q, epsilon):   # 利用episilon-policy选取下一个动作
    random_action = random.randint(0,3)     # 包含两端
    if random.random() < epsilon:   # 随机出来的数是小于epsilon,表示随机选取
aciton
        action = random_action
    else:   # 随机出来的数是大于epsilon，表示选择best_reward_action
        action = max_q(x, y, q)
    return action
```

- $Sarsa$**函数**：设置runs=20轮，用于求均值；而每一轮迭代500次，用于画图时显示reward随着迭代的增加的变化。每一次迭代都是从起点(0,0)遍历到终点。而在每次迭代中，根据上面算法原理的公式$Q(S,A)=Q(S,A)+\alpha[R+\gamma Q(S',A')-Q(S,A)]$进行更新。

```
def sarsa(q):
    runs = 20
    rewards = np.zeros(500)
    for j in range(runs):
        for i in range(500):
            reward_sum = 0
            x = 0
            y = 0
            action = epsilon_policy(x, y, q, epsilon)   # 根据epsilon_policy
选择action
            while 1:
                x_next, y_next, reward = take_action(x, y, action)
                reward_sum += reward
                action_next = epsilon_policy(x_next, y_next, q, epsilon)
                q[x][y][action] += alpha * (reward + gamma * q[x_next]
[y_next][action_next] - q[x][y][action])
                if x == width - 1 and y == 0:
                    break
                x = x_next
                y = y_next
                action = action_next
            rewards[i] += reward_sum
    rewards /= runs
    avg_rewards = []
    for i in range(9):
        avg_rewards.append(np.mean(rewards[:i + 1]))
    for i in range(10, len(rewards) + 1):
        avg_rewards.append(np.mean(rewards[i - 10:i]))
    return avg_rewards
```

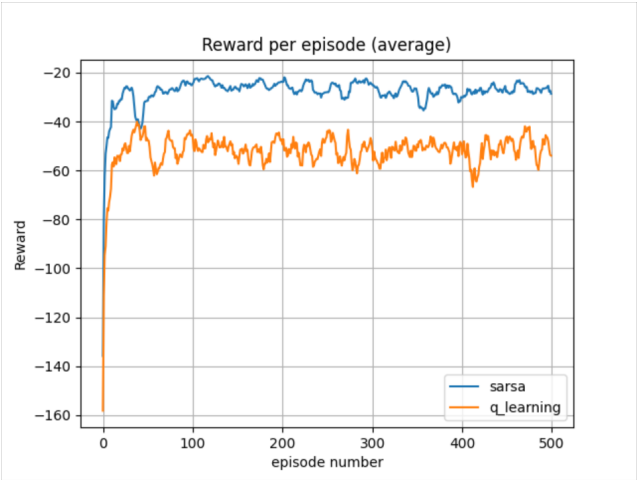
- $Q-learning$**函数**：整体方法和$Sarsa$相同，只是更新公式改为：$Q(S,A)=Q(S,A)+\alpha[R+\gamma max_a Q(S',a)-Q(S,A)]$，且不会直接取`action a`作为下一轮的action。

```
def q_learning(q):
    runs = 20
    rewards = np.zeros(500)
    for j in range(runs):
        for i in range(500):
            reward_sum = 0
            x = 0
            y = 0
            while 1:
                action = epsilon_policy(x, y, q, epsilon)  # 根据
epsilon_policy选择action
                x_next, y_next, reward = take_action(x, y, action)
                reward_sum += reward
                action_next = max_q(x_next, y_next, q)
                q[x][y][action] += alpha * (reward + gamma * q[x_next]
[y_next][action_next] - q[x][y][action])
                if x == width - 1 and y == 0:
```

```
                    break
                x = x_next
                y = y_next
            rewards[i] += reward_sum
    rewards /= runs
    avg_rewards = []
    for i in range(9):
        avg_rewards.append(np.mean(rewards[:i+1]))
    for i in range(10, len(rewards) + 1):
        avg_rewards.append(np.mean(rewards[i-10:i]))
    return avg_rewards
```

3. 实验结果

3.1 Reward per episode



3.2 Optimal path of two methods

```
"D:\Pycharm\PyCharm Community Editic
Sarsa Optimal Path:
→ → → → → → → → → → → ↓
↑ * * * * * * * * * * ↓
↑ * * * * * * * * * * ↓
↑ * * * * * * * * * * G

Q-learning Optimal Path:
* * * * * * * * * * * *
* * * * * * * * * * * *
→ → → → → → → → → → → ↓
↑ * * * * * * * * * * G

Process finished with exit code 0
```

```
"D:\Pycharm\PyCharm Community Editi
Sarsa Optimal Path:
* → → → → → → → → → ↓ *
→ ↑ * * * * * * * * → ↓
↑ * * * * * * * * * * ↓
↑ * * * * * * * * * * G

Q-learning Optimal Path:
* * * * * * * * * * * *
* * * * * * * * * * * *
→ → → → → → → → → → → ↓
↑ * * * * * * * * * * G

Process finished with exit code 0
```

The cliff是一个悬崖，上面的小方格表示可以走的道路。S为起点，G为终点。悬崖的reward为-100，小方格的reward为-1。$Q-learning$的结果为最优路径，$Sarsa$的结果为次优路径。这是由于在$Sarsa$更新的过程中，如果在悬崖边缘处，下一个状态由于是随机选取可能会掉下悬崖，因此当前状态值函数会降低，使得agent不愿意走靠近悬崖的路径。而Q-learning在悬崖边选取的下一步是最优路径，不会掉下悬崖，因此更偏向于走悬崖边的最优路径。 $Sarsa$是`online-policy`策略，会遭遇"探索-利用"的矛盾，利用目前已知的最优选择，收敛到**局部最优**，即每次探索出来的路径可能不同，甚至可能到达不了终点。 而$Q_learning$是`offline-policy`策略，计算下一状态的预期收益时使用了`max`操作，直接选择最优动作，最终会收敛到**全局最优**。