人工智能实验报告 第十三周

姓名:丁晓琪 学号:22336057

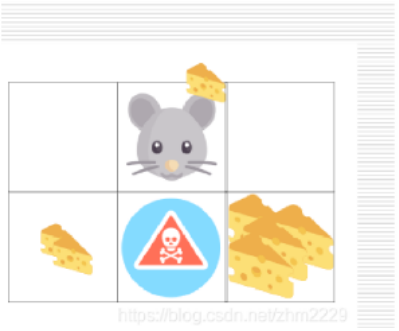
一.实验题目

You will implement Sarsa and Q-learning for the Maze environment from `OpenAI Gym`. We have provided custom versions of this environment. In the scenario, a red rectangle (agent) are initialized at the maze made up of 4×4 grids and can only observe its location. At each time step, agent can move to one of four neighboring grids. The reward is +1 if agent is located at the yellow grid, -1 if agent reaches the black grid, otherwise 0.

二. 实验内容

1.算法原理

- 背景:
  - `s`:当前状态， `a`:当前状态选择动作， `s'`:下一个状态， `a'`:下一个状态选择的动做
  - `Q(s,a)`：Q值函数，给定`s`状态选择`a`动作得到的预期回报。实验中`Q(s,a)`用二维表格数组存储，行代表状态，列代表采取动作



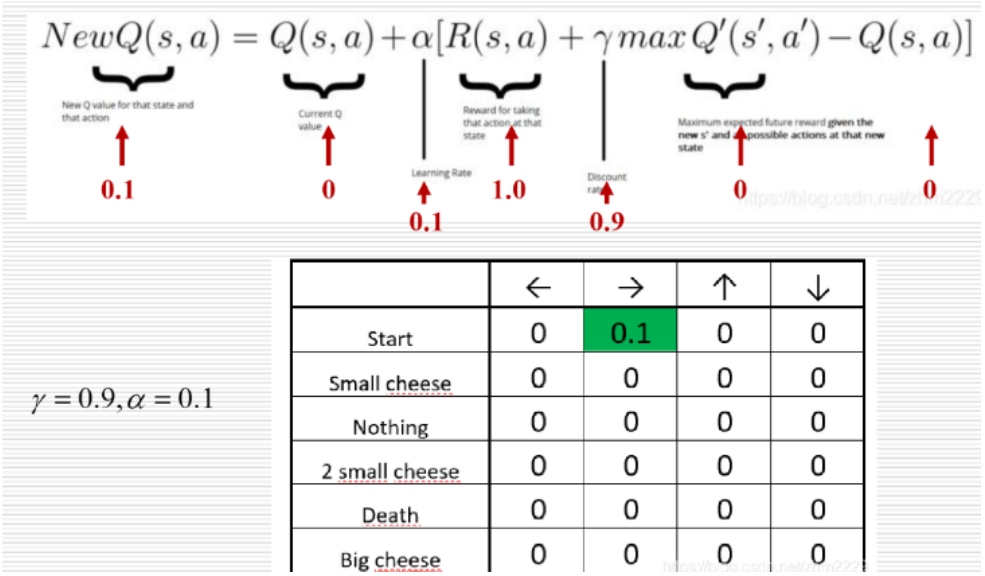
	←	→	↑	↓
Start	0	0	0	0
Small cheese	0	0	0	0
Nothing	0	0	0	0
2 small cheese	0	0	0	0
Death	0	0	0	0
Big cheese	0	0	0	0

  - `TD target`：时间差分目标，$TD = reward(s,a) + \gamma Q(s',a')$（当前动作$\alpha$在$s$下获得的真实奖励+衰减因子*下一个状态和选择动作下的期望值）。比$Q(s,a)$可信，用于迭代更新$Q(s,a)$
- Q_learning:
  - 已知：当前状态s.
  - 训练中动作选择：$\epsilon - greedy$策略，规定一个ϵ值（选择贪心策略的概率），当随机数概率大于ϵ时选择Q值最大的动作，否则随机选择动作

    （1）选择动作能得到最大的期望回报

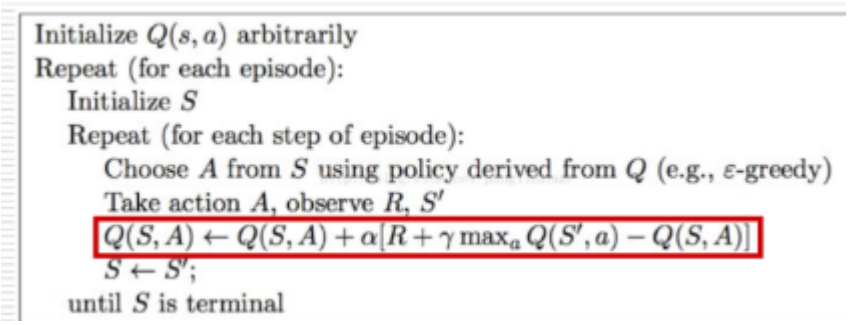
    （2）能够探索环境状态，从而学习整合不同动作选择下的期望回报
  - $Q(s,a)$的更新：根据学习率α和当前$Q(s,a)$与贪心策略下$TDtarget$的差距更新：

    $Q(s,a) = Q(s,a) + \alpha[reward(s,a) + \gamma max_a Q(s',a') - Q(s,a)]$



	←	→	↑	↓
Start	0	0.1	0	0
Small cheese	0	0	0	0
Nothing	0	0	0	0
2 small cheese	0	0	0	0
Death	0	0	0	0
Big cheese	0	0	0	0

- 伪代码：

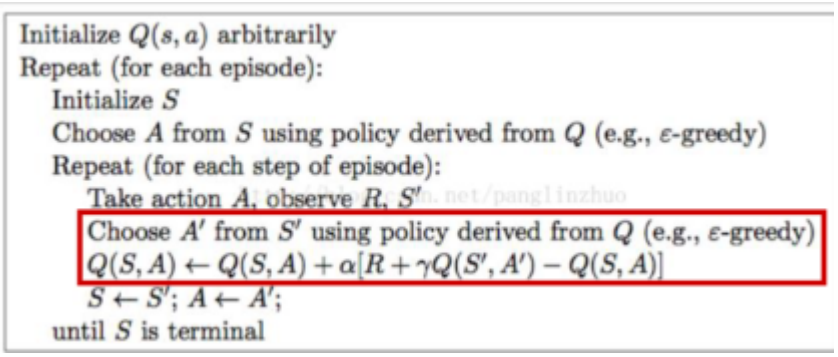


- Sarsa:
  - 已知：当前状态s.
  - 训练中动作选择：$\epsilon - greedy$策略，规定一个ϵ值（选择贪心策略的概率），当随机数概率大于ϵ时选择Q值最大的动作，否则随机选择动作

  （1）选择动作能得到最大的期望回报

  （2）能够探索环境状态，从而学习整合不同动作选择下的期望回报
  - Q(s,a)更新: 与Q_learning不同，$TDtarget$中动作a'的选择不是完全的贪心策略（选择让Q(s',a')最大的a'），而是和训练中动作选择一样的$\epsilon - greedy$策略

    $$Q(s,a) = Q(s,a) + \alpha[reward(s,a) + \gamma Q(s',a') - Q(s,a)]$$
  - 伪代码：



2.关键代码展示

- 游戏背景：

  `4x4` 的网格，黑色代表障碍 `reward=-1`,黄色代表终点 `reward=1`

```
            #     获得当前所在网格索引：
             s = env.canvas.coords(env.rect)
             state[0]=int((s[1]-5)//40)
             state[1]=int((s[0]-5)//40)
            #  选择动作和获得下一个状态和当前奖励：
             observation_, reward, done = env.step(action)
```

- Q_learning:
  - 相关参数设置和Q(s,a)表格建立（全部初始化为0）

```
        self.actions = actions  # a list
        self.lr = learning_rate
        self.gamma = reward_decay
        self.epsilon = e_greedy  #随机选择动作的概率
        #表格存储16个状态s在4个a的状态下的Q（s,a），用二维数组存储，行为0-15（i+j），列为0-3对应a
        #全部初始化为0
        self.q_table=np.zeros((16,len(self.actions)))

```

  - 动作选择：$\epsilon - greedy$策略

    注意：选择动作后要判断下一个状态是否越界

    在贪心策略时，如果最大Q值对应的动作有多个要随机选择，否则可能出现前期Q值都初始化为0，每次开始探索选择总是选择一个特定方向的动作，探索别的动作的概率低，学习缓慢。还可能出现在两个状态间来回运动切换，陷入循环。

```

    def choose_action(self, observation):
        i=observation[0]   #行：
        j=observation[1]   #列：
        s=i*4+j            #在q_table中的位置
        state_=[i,j]
        # 2.生成随机数概率与epsilon比较 epsilon-greedy策略
        action=0
        p=np.random.rand()
        # 【1】在0-epsilon范围内，随机选择  #注意选择动作检查下一个状态是否存在
        if(p<=self.epsilon):
            action = np.random.randint(0, len(self.actions))
            self.get_s_(state_,action)
            while(self.check_state_exist(state_)==0): #越界重新生成
                state_=[i,j]                          #复原状态
                action = np.random.randint(0, len(self.actions))
                self.get_s_(state_,action)
            return action
        # 【2】.选择Q值最大的动作
        else:
```

```
            #坚持越界行为，越界重新生成
            q_s_a=np.copy(self.q_table[s])  #深拷贝不要破坏q_table
            max_arr=[]                      #存放最大值对应的索引
            max=np.amax(q_s_a)
            for index in range(0,len(q_s_a)):
                if(q_s_a[index]==max):
                    max_arr.append(index)
            ran= np.random.randint(0, len(max_arr))  #在拥有最大值的动作中随机选一个
            action=max_arr[ran]

            self.get_s_(state_,action)
            while(self.check_state_exist(state_)==0): #越界重新生成
                state_=[i,j]
                q_s_a[action]=-10000 #失去竞选资格
                max_arr=[]
                max=np.amax(q_s_a)
                for index in range(0,len(q_s_a)):
                    if(q_s_a[index]==max):
                        max_arr.append(index)
                ran= np.random.randint(0, len(max_arr))
                action=max_arr[ran]
                self.get_s_(state_,action)
            return action

```

- Q(s,a)的更新学习：a'选择是贪心策略。

$$Q(s,a)=Q(s,a)+\alpha[reward(s,a)+\gamma max_aQ(s',a')-Q(s,a)]$$

```
    def learn(self, s, a, r, s_):
        ''' update q table ''' #s：当前状态坐标，a：动作，r：a动作后的实际奖励 s_:a动作后的坐标
        state=s[0]*4+s[1]
        Q_s_a=self.q_table[state][a]
        state_new=s_[0]*4+s_[1]
        Q_s_a_newmax=np.amax(self.q_table[state_new])
        #更新q_table
        self.q_table[state][a]=Q_s_a+self.lr*(r+self.gamma*Q_s_a_newmax-Q_s_a)
```

- 补充：如何规避学习时，总是只选择某方向的动作和 在某些状态间切换循环而不向外探索。

  ϵ下降：在训练初期将ϵ设置为较高值，让主体有较大的概率随机向外探索学习更新$Q(s,a)$。在训练过程中逐渐下降ϵ,使得在$Q(s,a)$收敛时，主体能够与较大的概率选择最优的动作，学习走到终点的路径

```
        if episode%10==0:
            RL.epsilon-=0.02
        if episode>90:
            RL.epsilon=0.05
```

- 总：让主体进行100次游戏，在迭代中更新Q值和学习`reward`最大的路径

```
    for episode in range(100):
        # initial observation
```

```
        observation = env.reset()
        if episode%10==0:
            RL.epsilon-=0.02
        if episode<10:
            RL.epsilon=0.05
        while True:
            env.render()  #更新上一次的动作
            #选择动作
            state=[0,0]
            s = env.canvas.coords(env.rect)
            state[0]=int((s[1]-5)//40)
            state[1]=int((s[0]-5)//40)
            action = RL.choose_action(state)
            #执行动作
            observation_, reward, done = env.step(action)
            #学习，更新表格
            ############
            state_=[0,0]
            s = env.canvas.coords(env.rect)
            state_[0]=int((s[1]-5)//40)
            state_[1]=int((s[0]-5)//40 )
            RL.learn(state, action, reward, state_)

            observation = observation_

            # break while loop when end of this episode
            if done:
                break
```

- Sara:
  - 相关参数设置：同Q_learning
  - 动作选择：同Q_learning
  - Q值更新：a'采用$\epsilon-greedy$策略：随机数大于ϵ时选择s'下最大的Q(s',a')值，否则随机选择s'下的Q(s',a')值

```
    def learn(self, s, a, r, s_):
        ''' update q table ''' #s：当前状态的像素坐标，a：动作，r：a动作后的实际奖励 s_:a动作后的像素坐标
        # YOUR IMPLEMENTATION HERE #
        state=s[0]*4+s[1]
        Q_s_a=self.q_table[state][a]
        state_new=s_[0]*4+s_[1]
        #也需要用eplision来选择S_时的q
        p=np.random.rand()
        if p>self.epsilon:
            Q_s_a_newmax=np.amax(self.q_table[state_new])
        else:
            action = np.random.randint(0, len(self.actions))
            Q_s_a_newmax=self.q_table[state_new][action]
        #更新q_table
        #改一下每个空格子的奖励，

        self.q_table[state][a]=Q_s_a+self.lr*(r+self.gamma*Q_s_a_newmax-Q_s_a)
```

```
        print("now:",s,"  q_s_a:",self.q_table[state])
```

- 补充：为了规避训练时在某个地方陷入死循环，在选择动作时为可能陷入循环的动作加以惩罚

```
    self.last_action=-3 #记录上一次选择动作
    def choose_action(self, observation):
        # 1.解锁状态（当前位置）
            #行：左上角x坐标/一个方格像素值 #修改放在主函数解
        i=observation[0]
            #列：  左上角y坐标/一个方格像素值
        j=observation[1]
            #在q_table中的位置
        s=i*4+j
        state_=[i,j]
        # YOUR IMPLEMENTATION HERE #
        # 2.生成随机数概率与epsilon比较
        action=0
        p=np.random.rand()
        # 3.在0-0.1范围内，随机选择   #这里选择动作要看动作后的状态是否存在
        if(p<=self.epsilon):
            action = np.random.randint(0, len(self.actions))
            self.get_s_(state_,action)
            while(self.check_state_exist(state_)==0): #越界重新生成
                state_=[i,j]#复原状态
                action = np.random.randint(0, len(self.actions))
                self.get_s_(state_,action)
            self.last_action=action
            return action
        # 4.选择Q值最大的动作
        else:
            #坚持越界行为，越界重新生成
            q_s_a=np.copy(self.q_table[s])  #深拷贝不要破坏q_table
                #重复惩罚：要是和上一次动作相反，也就是会使当前状态回到上一个状态时加个惩罚
                if(self.last_action!=-3):
                 if(self.last_action<=1 ):
                     q_s_a[1-self.last_action]-=0.005
                 else:
                     q_s_a[5-self.last_action]-=0.005

            max_arr=[]
            max=np.amax(q_s_a)
            for index in range(0,len(q_s_a)):
                if(q_s_a[index]==max):
                    max_arr.append(index)
            ran= np.random.randint(0, len(max_arr))
            action=max_arr[ran]

            self.get_s_(state_,action)
            while(self.check_state_exist(state_)==0): #越界重新生成
                state_=[i,j]
                q_s_a[action]=-10000 #失去竞选资格
                max_arr=[]
                max=np.amax(q_s_a)
                for index in range(0,len(q_s_a)):
                    if(q_s_a[index]==max):
```

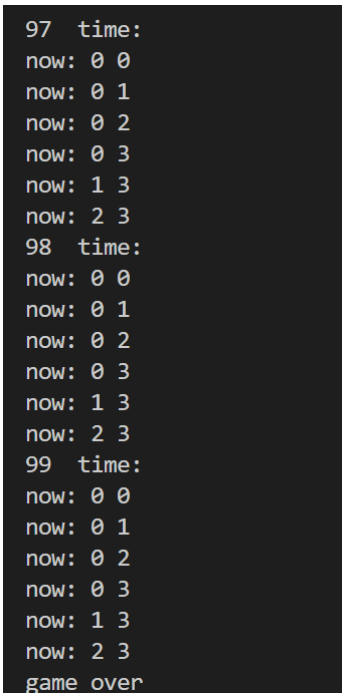
```
                        max_arr.append(index)
                ran= np.random.randint(0, len(max_arr))
                action=max_arr[ran]
                self.get_s_(state_,action)
            self.last_action=action
            return action
```

三.实验结果及分析

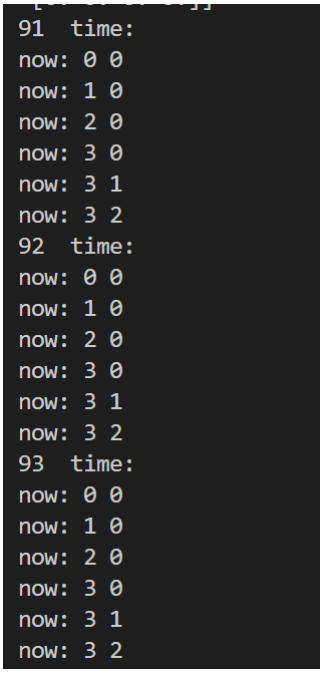
- Q_learning: 在最后10次迭代中，路线收敛到了一条，并且能够到达终点坐标（2，2）（由于最后一个状态的时候，一次游戏已经结束，所以没有打印出来）

```
91  time:
now: 0 0
now: 0 1
now: 0 2
now: 0 3
now: 1 3
now: 2 3
92  time:
now: 0 0
now: 0 1
now: 0 2
now: 0 3
now: 1 3
now: 2 3
93  time:
now: 0 0
now: 0 1
now: 0 2
now: 0 3
now: 1 3
now: 2 3
```

```
94  time:
now: 0 0
now: 0 1
now: 0 2
now: 0 3
now: 1 3
now: 2 3
95  time:
now: 0 0
now: 0 1
now: 0 2
now: 0 3
now: 1 3
now: 2 3
96  time:
now: 0 0
now: 0 1
now: 0 2
now: 0 3
now: 1 3
now: 2 3
```



- Sara:在最后10次迭代中，路线收敛到了一条，并且能够到达终点坐标（2，2）（由于最后一个状态的时候，一次游戏已经结束，所以没有打印出来）



```
94  time:
now: 0 0
now: 1 0
now: 2 0
now: 3 0
now: 3 1
now: 3 2
95  time:
now: 0 0
now: 1 0
now: 2 0
now: 3 0
now: 3 1
now: 3 2
96  time:
now: 0 0
now: 1 0
now: 2 0
now: 3 0
now: 3 1
now: 3 2
```

```
97  time:
now: 0 0
now: 1 0
now: 2 0
now: 3 0
now: 3 1
now: 3 2
98  time:
now: 0 0
now: 1 0
now: 2 0
now: 3 0
now: 3 1
now: 3 2
99  time:
now: 0 0
now: 1 0
now: 2 0
now: 3 0
now: 3 1
now: 3 2
game over
```

四.参考内容

课程ppt