Assignment 3

Peng Tang 517020910038

In experiment, we are going to find the best solution for Cliff Walking problem by using on-policy learning (Sarsa) and off-policy learning(Q-learning) and compare their performance with different parameters.

1 Analysis

1.1 Sarsa

Sarsa is an on-policy learning, which means we need to use same policy to take actions and update Q-values. We can implement Sarsa for this problem as following steps.

- We set the start state and take actions by using ϵ-greedy policy(shown as following equation) until reaching the terminal state.

$$\pi(a|s) = \begin{cases} 1-\epsilon+\frac{\epsilon}{4} & a = \arg\max_a Q(s,a) \\ \frac{\epsilon}{4} & a \neq \arg max_a Q(s,a) \end{cases}$$

- In this process, we use the ϵ-greedy policy to update the Q-values by using the following equation.

$$Q(s,a) \leftarrow Q(s,a) + \alpha(R + \lambda Q(s',a') - Q(s,a))$$

where a' is taken by using ϵ-greedy; R is -100 if stepping into the region marked "The Cliff", otherwise -1.

- We do above two steps for enough times until all Q-values converge.
- Then we can find the best path from the start state to the goal state by using greedy policy.

1.2 Q-learning

Q-learning is an off-policy learning, which means we need to use different policies to take actions and update Q-values. We can implement Q-learning for this problem as following steps.

- We set the start state and take actions by using ϵ-greedy policy(shown as following equation) until reaching the terminal state.

$$\pi(a|s) = \begin{cases} 1-\epsilon+\frac{\epsilon}{4} & a = \arg\max_a Q(s,a) \\ \frac{\epsilon}{4} & a \neq \arg max_a Q(s,a) \end{cases}$$

- In this process, we use the greedy policy to update the Q-values by using the following equation.

$$Q(s,a) \leftarrow Q(s,a) + \alpha(R + \lambda \max_{a'} Q(s',a') - Q(s,a))$$

where R is -100 if stepping into the region marked "The Cliff", otherwise -1.

- We do above two steps for enough times until all Q-values converge.
- Then we can find the best path from the start state to the goal state by using greedy policy.

2 Code

2.1 Sarsa Class

```
class Sarsa(object):
    def __init__(self, discount, maxIteration, epsilon, alpha) -> None:
        self.discount = discount
        self.maxIteration = maxIteration
        self.epsilon = epsilon
        self.alpha = alpha

        self.Q_Value = np.random.random((4,12,4))
        self.Q_Value[3][11][0] = self.Q_Value[3][11][1]=self.Q_Value[3][11][2]=self.Q_Value[3][11][3]=0
        self.terminal = [3,11]

        self.actions = [[-1,0],[1,0],[0,-1],[0,1]]
        self.cliffstate = [[3,1],[3,2],[3,3],[3,4],[3,5],[3,6],[3,7],[3,8],[3,9],[3,10]]

    def findOptimalPath(self):

        iteration = 0
        cost = []
        while iteration < self.maxIteration:
            self.alpha = 0.99*self.alpha
            state = [3,0]
            nextstate = [0,0]

            tempcost = 0
            action = self.getaction(state)
            while state !=self.terminal:
                nextstate[0] = max(0,min(3,state[0]+self.actions[action][0]))
                nextstate[1] = max(0,min(11,state[1]+self.actions[action][1]))

                reward = -1
                if nextstate in self.cliffstate:
                    reward = -100
                    nextstate[0] = 3
                    nextstate[1] = 0

                tempcost += reward
                nextaction = self.getaction(nextstate)

                self.Q_Value[state[0]][state[1]][action] = (1-self.alpha)*self.Q_Value[state[0]][state[1]][action] + self.alpha*(reward+self.discount*self.Q_Value[nextstate[0]][nextstate[1]][nextaction])

                state[0] = nextstate[0]
                state[1] = nextstate[1]
                action = nextaction
```

```
            iteration += 1
            cost.append(tempcost)

        state = [3,0]
        actionlist = []
        realaction = ['up','down','left','right']
        while state!=self.terminal:
            action = np.argmax(self.Q_Value[state[0]][state[1]])
            actionlist.append(realaction[action])
            state[0] = max(0,min(3,state[0]+self.actions[action][0]))
            state[1] = max(0,min(11,state[1]+self.actions[action][1]))

            if state in self.cliffstate:
                state[0] = 3
                state[1] = 0

        return actionlist, cost

    def getaction(self,state):
        greedyAction = np.argmax(self.Q_Value[state[0]][state[1]])
        p = np.random.random()
        if p < 1-self.epsilon:
            return greedyAction
        else:
            return np.random.randint(4)
```

- We use this class to implement Sarsa.
- In this class, **discount** is the discount factor(λ in the equation); **maxIteration** is the number of episodes we sample; **alpha** is the step size(α in the equation) and we decrease it every episode by multiplying 0.99; **epsilon** is the ϵ in the ϵ-greedy policy; **Q_value** is used to record the Q-values and we initialize it randomly; **terminal** is the goal; **actions** includes four actions, up, down, left and right; **cliffstate** includes all states marked "The Cliff".
- The **findOptimalPath** function is used to sample and update Q-values and find the optimal path from the start state to the goal state at the end. We implement this function as our analysis and we use the **cost** to record the total rewards of every episode. And the function will return the actions we need to take from the start state and the costs of all episodes.
- The **getaction** function is used to choose a action by using the ϵ-greedy policy.

2.2 Q_learning Class

```
class Q_learing(object):
    def __init__(self, discount, maxIteration, epsilon, alpha) -> None:
        self.discount = discount
        self.maxIteration = maxIteration
        self.epsilon = epsilon
        self.alpha = alpha

        self.Q_Value = np.random.random((4,12,4))
        self.Q_Value[3][11][0] = self.Q_Value[3][11][1]=self.Q_Value[3][11]
[2]=self.Q_Value[3][11][3]=0
        self.terminal = [3,11]

        self.actions = [[-1,0],[1,0],[0,-1],[0,1]]
```

```
        self.cliffstate = [[3,1],[3,2],[3,3],[3,4],[3,5],[3,6],[3,7],[3,8],
[3,9],[3,10]]

    def findOptimalPath(self):

        iteration = 0
        cost = []
        while iteration < self.maxIteration:
            self.alpha = 0.99*self.alpha
            state = [3,0]
            nextstate = [0,0]
            tempcost = 0
            while state !=self.terminal:
                action = self.getaction(state)
                nextstate[0] = max(0,min(3,state[0]+self.actions[action]
[0]))
                nextstate[1] = max(0,min(11,state[1]+self.actions[action]
[1]))

                reward = -1
                if nextstate in self.cliffstate:
                    reward = -100
                    nextstate[0] = 3
                    nextstate[1] = 0

                tempcost += reward
                self.Q_Value[state[0]][state[1]][action] = (1-
self.alpha)*self.Q_Value[state[0]][state[1]][action] + self.alpha*
(reward+self.discount*np.max(self.Q_Value[nextstate[0]][nextstate[1]]))

                state[0] = nextstate[0]
                state[1] = nextstate[1]

            iteration += 1
            cost.append(tempcost)

        state = [3,0]
        realaction = ['up','down','left','right']
        actionlist = []
        while state!=self.terminal:
            action = np.argmax(self.Q_Value[state[0]][state[1]])
            actionlist.append(realaction[action])
            state[0] = max(0,min(3,state[0]+self.actions[action][0]))
            state[1] = max(0,min(11,state[1]+self.actions[action][1]))

            if state in self.cliffstate:
                state[0] = 3
                state[1] = 0

        return actionlist, cost

    def getaction(self,state):
        greedyAction = np.argmax(self.Q_Value[state[0]][state[1]])
        p = np.random.random()
        if p < 1-self.epsilon:
            return greedyAction
        else:
            return np.random.randint(4)
```

This class is similar to the Sarsa and the difference is the way to update the Q-values in **findOptimalPath** function. We use greedy policy rather than ϵ-greedy in this class.

2.3 Main Function

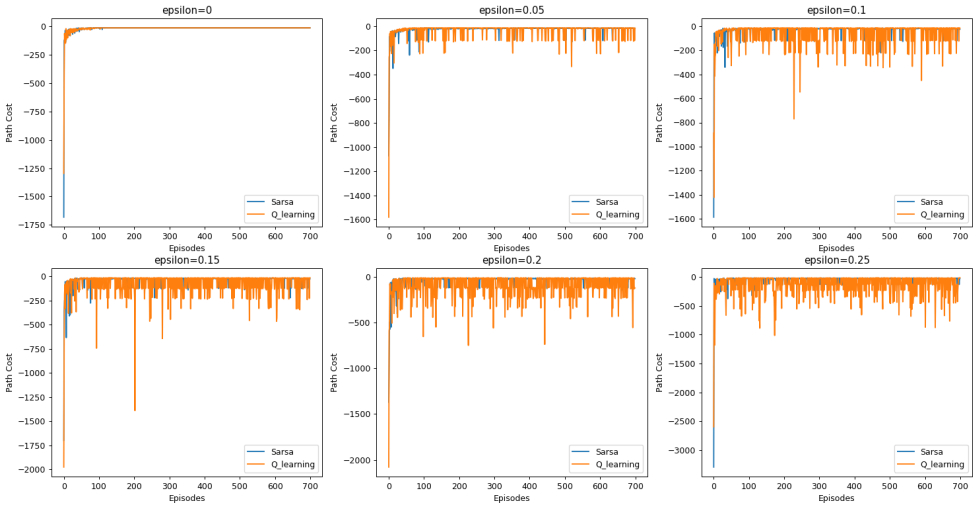
```
def main():

    np.random.seed(0)
    discount = 0.9
    maxIteration = 700
    epsilon = [0,0.05,0.1,0.15,0.2,0.25]
    alpha = 0.5
    SarsaRewardList = []
    Q_learningRewardList = []
    CostTime = []
    Sarsa_ActionList = []
    Q_learingActionList = []


    for i in range(len(epsilon)):
        t1 = time.time()
        SarsaModel = Sarsa(discount=discount,
maxIteration=maxIteration,epsilon=epsilon[i],alpha=alpha)
        Sarsa_Action, Sarsa_Cost = SarsaModel.findOptimalPath()
        Q_learingModel = Q_learing(discount=discount,
maxIteration=maxIteration,epsilon=epsilon[i],alpha=alpha)
        Q_learing_Action, Q_learing_Cost = Q_learingModel.findOptimalPath()
        t2 = time.time()

        CostTime.append(t2-t1)
        Sarsa_ActionList.append(Sarsa_Action)
        Q_learingActionList.append(Q_learing_Action)
        SarsaRewardList.append(Sarsa_Cost)
        Q_learningRewardList.append(Q_learing_Cost)




    subfig = [231,232,233,234,235,236]
    plt.figure(figsize=(20,10),dpi=90)
    for i in range(len(epsilon)):
        plt.subplot(subfig[i])
        plt.plot(list(range(maxIteration)),SarsaRewardList[i],label="Sarsa")

 plt.plot(list(range(maxIteration)),Q_learningRewardList[i],label="Q_learnin
g")
        plt.xlabel("Episodes")
        plt.ylabel("Path Cost")
        plt.legend()
        plt.title("epsilon="+str(epsilon[i]))
    plt.savefig('imgs/costChanges.png')
    plt.close()


    plt.plot(epsilon, CostTime)
    plt.xlabel("Different Epsilons")
```

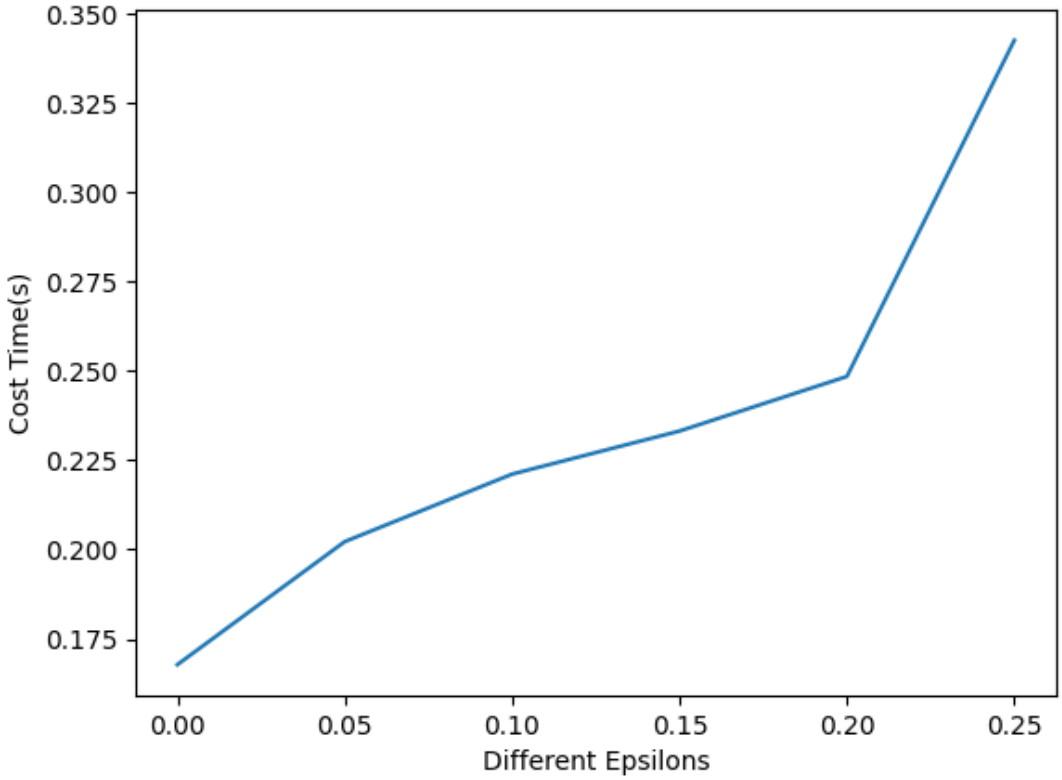
```
    plt.ylabel("Cost Time(s)")
    plt.savefig('imgs/CostTime.png')
    plt.close()


    print("Sarsa actions:")
    for i in range(len(epsilon)):
        print("epsilon="+str(epsilon[i])+": ",Sarsa_ActionList[i])
    print()
    print("Q_learning actions:")
    for i in range(len(epsilon)):
        print("epsilon="+str(epsilon[i])+": ",Q_learingActionList[i])
```

This function is used to call Sarsa and Q_learning and output the results. In order to analyze the influence of different ϵ values, we set six different values$(0, 0.05, 0.1, 0.15, 0.2, 0.25)$ and compare their performance.

3 Results

By calling the **main()** function, we get the following results.






```
PS E:\大三下\强化学习\作业\homework3\code> D:\python\python.exe "e:\大三下\强化学习\作业\homework3\code\solution.py"
Sarsa actions:
epsilon=0:  ['up', 'right', 'right', 'right', 'right', 'right', 'right', 'right', 'right', 'right', 'right', 'right', 'down']
epsilon=0.05:  ['up', 'up', 'right', 'right', 'right', 'right', 'right', 'right', 'right', 'right', 'right', 'right', 'down', 'down']
epsilon=0.1:  ['up', 'up', 'up', 'right', 'right', 'right', 'right', 'right', 'right', 'right', 'right', 'right', 'right', 'right', 'down', 'down', 'down']
epsilon=0.15:  ['up', 'up', 'up', 'right', 'right', 'right', 'right', 'right', 'right', 'right', 'right', 'right', 'right', 'right', 'down', 'down', 'down']
epsilon=0.2:  ['up', 'up', 'up', 'right', 'right', 'right', 'right', 'right', 'right', 'right', 'right', 'right', 'right', 'right', 'down', 'down', 'down']
epsilon=0.25:  ['up', 'up', 'up', 'right', 'right', 'right', 'right', 'right', 'right', 'right', 'right', 'right', 'right', 'right', 'down', 'down', 'down']

Q_learning actions:
epsilon=0:  ['up', 'right', 'right', 'right', 'right', 'right', 'right', 'right', 'right', 'right', 'right', 'right', 'down']
epsilon=0.05:  ['up', 'right', 'right', 'right', 'right', 'right', 'right', 'right', 'right', 'right', 'right', 'right', 'down']
epsilon=0.1:  ['up', 'right', 'right', 'right', 'right', 'right', 'right', 'right', 'right', 'right', 'right', 'right', 'down']
epsilon=0.15:  ['up', 'right', 'right', 'right', 'right', 'right', 'right', 'right', 'right', 'right', 'right', 'right', 'down']
epsilon=0.2:  ['up', 'right', 'right', 'right', 'right', 'right', 'right', 'right', 'right', 'right', 'right', 'right', 'down']
epsilon=0.25:  ['up', 'right', 'right', 'right', 'right', 'right', 'right', 'right', 'right', 'right', 'right', 'right', 'down']
PS E:\大三下\强化学习\作业\homework3\code>
```

- From the first figure, we can see that as the ϵ becomes bigger, the fluctuation of curves is bigger. This is because the ϵ is bigger, we explore more randomly. What more, if ϵ is 0, the Sarsa is same as Q-learning according our algorithm. And if ϵ is not 0, we can see that the fluctuation of Q-learning is bigger than the Sarsa. I think it is because that for the Q-learning, the Q-values of the states near the "the Cliff" states is bigger, then the agent will step into these states more often and then it will step into the "the Cliff" states easier by taking actions with ϵ-greedy.
- From the second figure, we can see that as the ϵ is bigger, we need more time to sample the fixed number of episodes. The is obvious because ϵ is bigger, we need explore more. When ϵ is too big(maybe bigger than 0.4), we need very long time to sample.
- From the third figure, we can see that with different ϵ values, Sarsa will give different actions while Q-learning will give same action.
    - For Sarsa, because it uses ϵ-greedy policy to update the Q-values, then as the ϵ is bigger, the Q-values of the states near "The Cliff" region will become smaller than usual. Thus, when we take actions by using greedy policy, we will stay away from "The Cilff" region.
    - For Q-learning, because it uses greedy policy to update the Q-values, then the Q-values of the states near the goal states will become bigger although these state is also near "The Cliff" states in this problem.

- From this problem, we can see that when ϵ is 0, we can get the best performance. However, this is because the state space and action space are smaller, we needn't to explore. When these two spaces become bigger, we need to explore more to find the best solution.