```
# -*- coding: utf-8 -*-
"""cliffwalking.ipynb

Automatically generated by Colaboratory.

Original file is located at
    https://colab.research.google.com/drive/1-c9qXwTcmGxNTNG62iOgn3cVqkTTRvO0
"""

#Required Libraries
import numpy as np
import matplotlib.pyplot as plt

# Values required by the agent
alpha = 0.1
gamma = 1.0
epsilon = 0.1 #referenced from figure 6.6 from Sutton and Barto
episodes = 500
runs = 10
action_types = 4 #total number of actions(up/down/left/right)

# defining a class for the gridworld environmenr
class cliffgrid(object):
  def __init__(self):
    self.rows = 4
    self.columns = 12
    self.actions = [[0,1],[0,-1],[-1,0],[1,0]]
    self.reset()

  def reset(self):#resets the agents position
    self.x = 0
    self.y = 0
    self.end_X = 11
    self.end_y = 0
    self.stop = False

  def agentstatus(self):#required for checking the status of the agent
    return tuple((self.x,self.y)), self.stop

  def agentposition(self,x,y):
    x = max(x, 0)
    y = max(y, 0)
    x = min(x, self.columns - 1)
    y = min(y, self.rows - 1)
    return x,y

  def agentmovement(self, action):#required to move the agents position and update
rewards
    self.stop = False
    self.x += self.actions[action][0]
    self.y += self.actions[action][1]
    self.x, self.y = self.agentposition(self.x, self.y)

    if self.x >= 1 and self.x <= 10 and self.y == 0: #resets environemt if agent
falls off
      reward = -100
      self.reset()
    elif self.x == self.columns -1 and self.y == 0:# sets reward to 0 and stops the
agent when goal is reached
```

```
      reward = 0
      self.stop = True
    else:
      reward = -1 #rewards for moving in the correct path
    return ((self.x, self.y)), reward, self.stop

def defaultdict(default_type):
    class DefaultDict(dict):
        def __getitem__(self, key):
            if key not in self:
                dict.__setitem__(self, key, default_type())
            return dict.__getitem__(self, key)
    return DefaultDict()

#calculates the egreedypolicy
def epsilongreedypolicy(Q, state):
    action = np.argmax(Q[state])
    act = np.ones(action_types) * epsilon / action_types
    act[action] += 1 - epsilon
    return act

#Qlearning
def Qlearning(grid):
    Q=defaultdict(lambda: np.zeros(action_types))
    rewards = [] #stores the rewards
    for i in range(episodes): #resets the environment for each episode
        grid.reset()
        state, stop = grid.agentstatus()
        sumofreward = 0.0

        while 1:
            prob = epsilongreedypolicy(Q, state)#probability for the next state
            action = np.random.choice(np.arange(action_types), p=prob) #randomizes
action
            nextstate, reward, stop = grid.agentmovement(action)
            nextaction = np.argmax(Q[nextstate])
            #Q learning update formula
            Q[state][action] = Q[state][action] + alpha * (reward + gamma *
Q[nextstate][nextaction] - Q[state][action])
            state = nextstate
            if stop:
                break
            sumofreward += reward #sums up the total rewards
        rewards.append(sumofreward)
    return Q, rewards

#SARSA
def sarsa(grid):
    Q=defaultdict(lambda: np.zeros(action_types))
    rewards = []
    for episode in range(episodes): #resets environment
        grid.reset()
        state, stop = grid.agentstatus()
        prob = epsilongreedypolicy(Q, state)
        action = np.random.choice(np.arange(action_types), p=prob)
        sumofreward = 0.0
        while 1:
            nextstate, reward, stop = grid.agentmovement(action)
            prob = epsilongreedypolicy(Q, nextstate)
```

```
            nextaction = np.random.choice(np.arange(action_types),
                                                p=prob)
            #SARSA update formula
            Q[state][action] = Q[state][action] + alpha * (
                    reward + gamma * Q[nextstate][nextaction] - Q[state][action])
            if stop:
                break
            state = nextstate
            action = nextaction
            sumofreward += reward
        rewards.append(sumofreward)
    return Q, rewards

def plot(episodelength,average,label):
    length = len(episodelength)
    episodelength = [episodelength[i] for i in range(length) if i % runs == 0]
    average = [average[i] for i in range(length) if i % runs == 0]
    plt.plot(episodelength, average,label=label)

qlearngrid=cliffgrid() # creates a new Q-learning environment
q1, rewards = Qlearning(qlearngrid)
print(sum(rewards)/episodes)
sarsagrid = cliffgrid()# creates a new SARSA environment
q2, rewards = sarsa(sarsagrid)
print(sum(rewards)/episodes)
qlearnaverage = []
sarsaaverage=[]
for i in range(runs):
  q1, qlearnreward = Qlearning(qlearngrid) #Stores the states and rewards for Q
learning
  q2, sarsareward = sarsa(sarsagrid) # Stores the states and rewards for Sarsa
  #Calculates the average
  qlearnaverage=np.array(qlearnreward) if len(qlearnaverage) == 0 else
qlearnaverage + np.array(qlearnreward)
  sarsaaverage= np.array(sarsareward) if len(sarsaaverage) == 0 else sarsaaverage +
np.array(sarsareward)


qlearnaverage /= runs
sarsaaverage/= runs

#plots the graph
plot(range(episodes), qlearnaverage,label='Q-learning='+str(epsilon))
plot(range(episodes), sarsaaverage,label='Sarsa='+str(epsilon))
plt.title("Cliff walking")
plt.ylabel('Sum of rewards during episode')
plt.xlabel('Episode')
plt.ylim(-500,0)
plt.legend()
plt.show()

qlearnaverage
```