Homework 3

1.

1\. The pseucode for the Monte Carlo similar to given in chapter 2.4 is

$$Q_t(S_t, A_t) = \frac{1}{t} \sum_{i=1}^{t} R_{oi}(S_t, A_t)$$

$$= \frac{1}{t} \left[\sum_{i=1}^{t-1} R_{oi}(S_t, A_t) + R_t(S_t, A_t) \right]$$

$$= \frac{1}{t} \left[(t-1) Q_{t-1}(S_t, A_t) + R_t(S_t, A_t) \right]$$

$$= Q_{t-1}(S_t, A_t) + \frac{1}{t} \left[R_t(S_t, A_t) - Q_{t-1}(S_t, A_t) \right]$$

→ Complete Monte Carlo ES [Exploring Starts] pseucode for finding optimal policy.

(i) Inilization Step

- $\pi(S) \in A(S)$, for $s \in S$ {Possible states}
- Set $R(S, A) \leftarrow [\,]$ for all $s \in S$, $a \in A(s)$
- $Q(s, a) \in 0$ or R [for all $s \in S, a \in A(s)$]
- $C(s, a) = 0$ [Count of occurance of (s, a)]

(ii) Algorithmic Steps

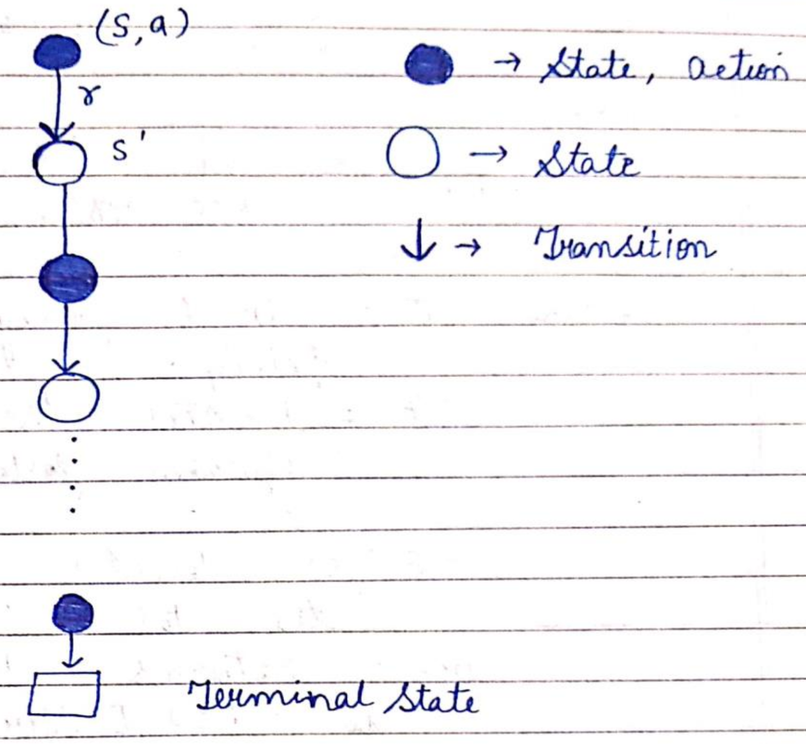
→ for each episcode:
  → Choose Randomly $S_0 \in S$
  → Set $A_0 = \pi(S_0)$
  → Generate the episcode from policy π as: $S_0, A_0, R_1, S_1, A_1, R_2 \ldots A_{T-1}, R_T$
  → $G \leftarrow 0$, $L \leftarrow [\,]$
  → for $t = T-1 : 0$:
    → $G \leftarrow \gamma G + R_{t+1}$
    → ~~L.append(St, At)~~
    → if S_t, A_t not in L:
      → L.append(S_t, A_t)
      → $C(S_t, A_t) = C(S_t, A_t) + 1$
      → $Q(S_t, A_t) = Q(S_t, A_t) + \frac{1}{C(S_t, A_t)} \left[G - Q(S_t, A_t) \right]$
      → $\pi(S_t) = \underset{a \in A(S_t)}{\arg\max}\, Q(S_t, a)$

→ Return π, $Q(S_t, A_t)$

2.

2. The Backup diagram for Monte Carlo Estimation for $q_\pi(s, a)$ is given below :-

→ In estimation of $q_\pi(s, a)$, it starts with initial s_0, a_0 and the estimation cycle ends when the episode reaches the terminal state.



3.

3. → The expression for the weighted importance sampling is given as

$$V(s) = \frac{\sum_{t \in \mathcal{J}(s)} \rho_{t:T(t)-1} G_t}{\sum_{t \in \mathcal{J}(s)} \rho_{t:T(t)-1}} \quad \text{if } \sum_{t \in \mathcal{J}(s)} \rho_{t:T(t)-1} \neq 0$$

$$, 0 \text{ otherwise}$$

where

$$\rho_{t:T-1} = \prod_{k=t}^{T-1} \frac{\pi(A_k | S_k)}{b(A_k | S_k)}$$

where π = target policy and b is behaviour policy.

$T(t)$ = denotes the termination of episode following the time stamp t

$t \in \mathcal{J}(s)$ denotes the time stamp when the state 's' is occured.

G_t : Returns from time stamp 't'+1 to $T(t)$ [Termination]

→ writing the equation for $Q(s,a)$ in weighted importance ~~sto~~ sampling is

$$Q(s,a) = \frac{\sum_{t \in \mathcal{J}(s,a)} \rho_{t:T(t)-1} G_t}{\sum_{t \in \mathcal{J}(s,a)} \rho_{t:T(t)-1}} \quad \text{if denominator} \neq 0$$

$$= 0 \quad , \text{otherwise}$$

where

$\mathcal{J}(s,a)$ denotes the time stamps when the (s,a) occured in the whole episode.

4.

5.1.

Code :

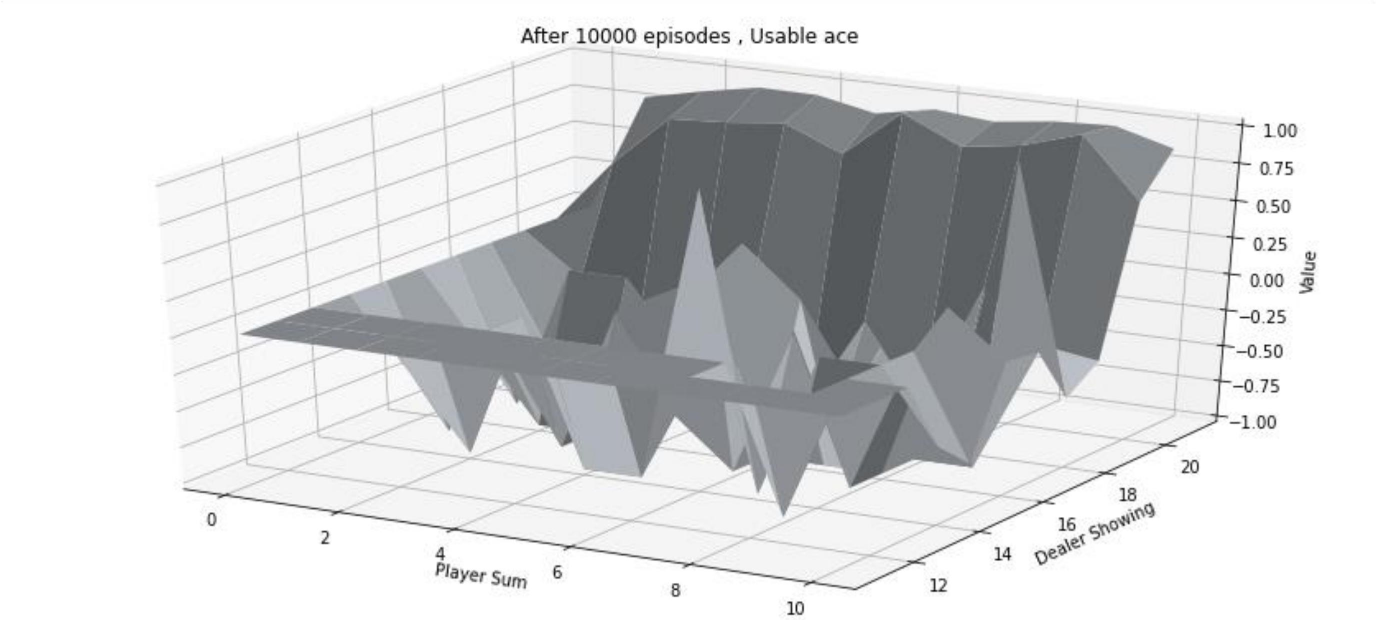
```
np.random.seed(42)

class BlackJack():
  def __init__(self):
    self.play = []
    self.deal = []
    self.decklist = list(np.arange(1,11)) + [10,10,10]
    self.start()

  def playcard(self):
    return np.random.choice(self.decklist)

  def playhand(self):
      return [self.playcard(),self.playcard()]

  def currentusability(self,draw):
    total_sum = sum(draw)
    if 1 in draw:
      if total_sum + 10<=21:
        return True
      else :
        return False
    else :
      return False

  def currentsum(self,draw):
    if self.currentusability(draw)==True:
      total_sum = sum(draw) + 10
    else :
      total_sum = sum(draw)
    return total_sum

  def currentbust(self,draw):
    total_sum = sum(draw)
    if total_sum>21:
      return True
    else :
      return False

  def score(self,draw):
    if self.currentbust(draw)==True:
      return 0
    else :
      return self.currentsum(draw)

  def step(self, action):

    end = True
    # Hit : Player turn
```

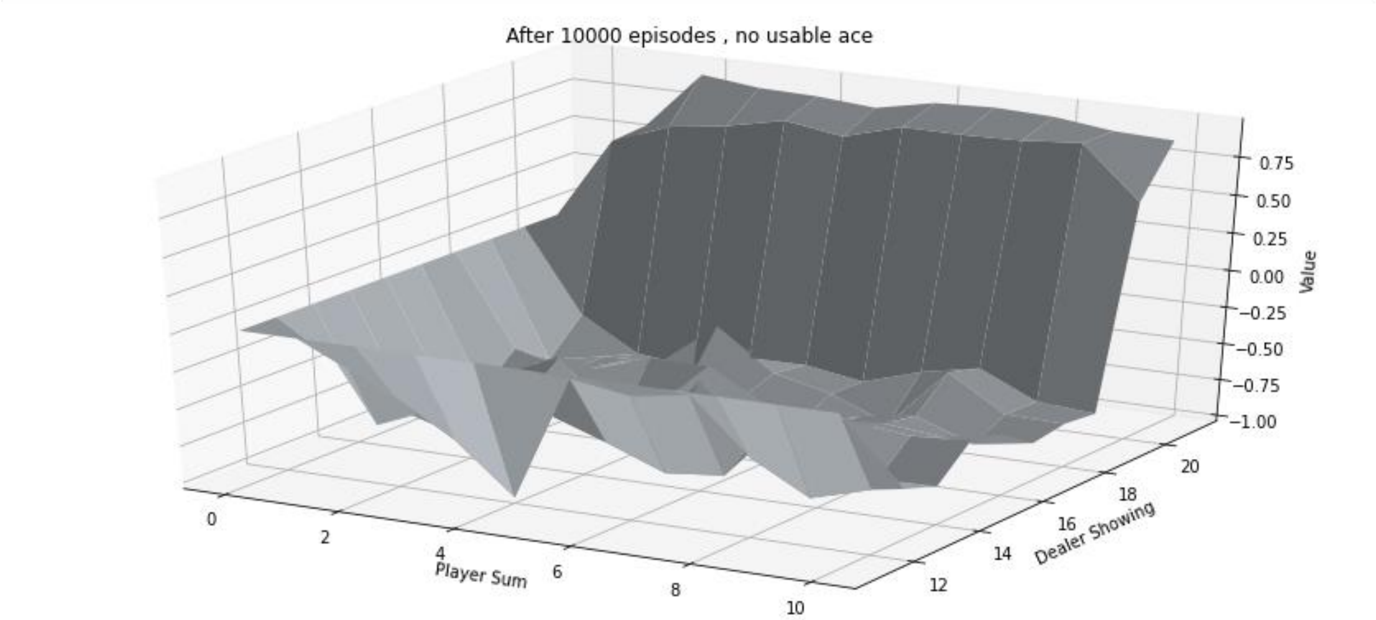
```python
    if action==1:
        self.play.append(self.playcard())
        if self.currentbust(self.play):
            reward = -1
        else:
            end = False
            reward = 0

    # Stick: Dealers turn
    else:

        while self.currentsum(self.deal) < 17:
            self.deal.append(self.playcard())
        playsc = self.score(self.play)
        dealsc = self.score(self.deal)
        if playsc == dealsc:
          reward = 0
        elif playsc > dealsc:
          reward = 1
        else :
          reward = 0
    observation = [self.currentsum(self.play), self.deal[0], self.currentusability(
self.play)]
    step_results =  [observation , reward , end , {}]
    return step_results

  def start(self):
      self.deal = self.playhand()
      self.play = self.playhand()
      while self.currentsum(self.play) < 12:
          self.play.append(self.playcard())

      observation = [self.currentsum(self.play), self.deal[0], self.currentusabilit
y(self.play)]
      return observation

def Monte_Carlo_Prediction(Episode_Lenght,Policy,BlackJact_Env,Alpha):
    Return_St = defaultdict(float)
    Count_St = defaultdict(float)
    Value_St = defaultdict(float)

    for i in range(Episode_Lenght):
      # Generating Episode (S,A,R)
      Episode = []
      States = []
      S0 = BlackJact_Env.start()
      for _ in range(1000):
          A0 = Policy(S0)
          observation = BlackJact_Env.step(A0) # (State,Reward,End)
          Episode.append([observation[0],observation[1],observation[2]])
          States.append(tuple(observation[0]))
          if observation[2]==True:
              break
          S0 = observation[0]
```

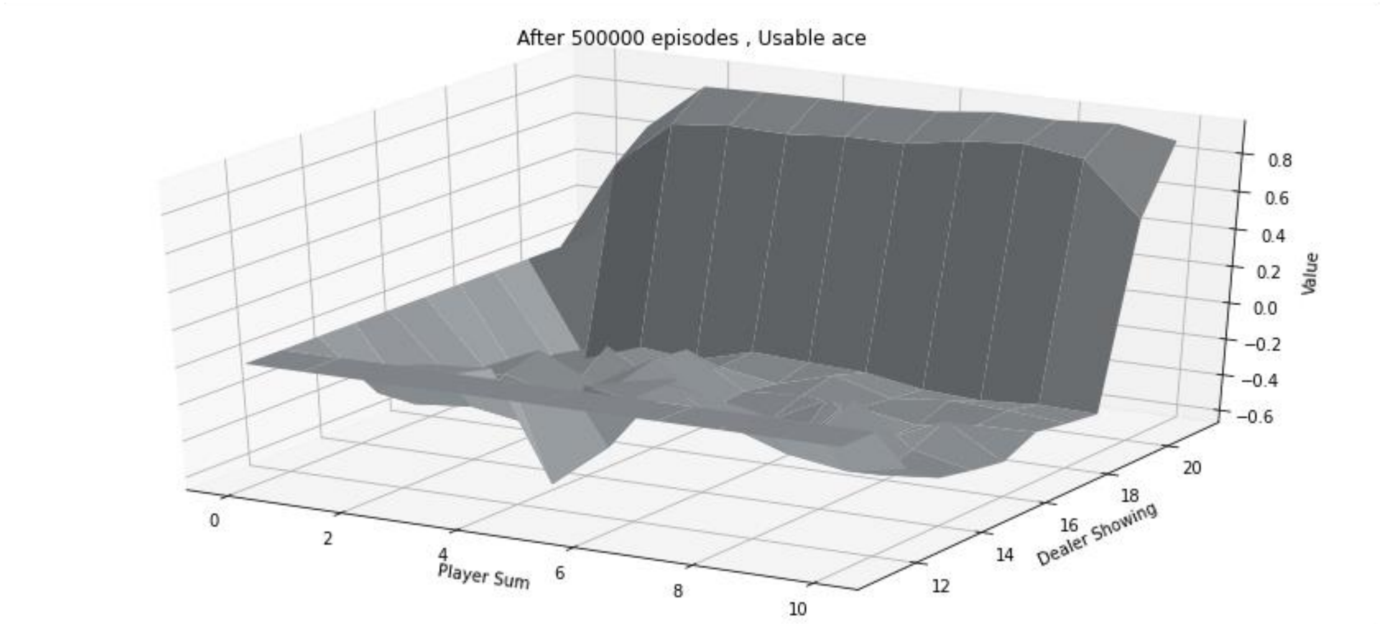
```
      G = 0
      States_Visited = []
      for i in range(len(Episode)-1,-1,-1):
        G = Alpha*G + Episode[i][1]
        if States[i] in States_Visited:
          continue
        else :
          Return_St[States[i]] = Return_St[States[i]] + G
          Count_St[States[i]] = Count_St[States[i]] + 1
          Value_St[States[i]] = Return_St[States[i]]/Count_St[States[i]]
          States_Visited.append(States[i])

    return Value_St

def Policy(State):
    Player_Score,_,_ = State
    if Player_Score>=20:
      return 0
    else :
      return 1

def Plotting_Value_Function(Value_St,count):
  Play_show = [i[0] for i in Value_St.keys()]
  Deal_show = [i[1] for i in Value_St.keys()]

  Play_range = np.arange(min(Play_show)-1,max(Play_show)+1)
  Deal_range = np.arange(min(Deal_show)-1,max(Deal_show)+1)
  P, D = np.meshgrid(Play_range,Deal_range)
  T = np.dstack([P, D])

  # Usable and usable ace
  usevalues = lambda _: Value_St[(_[0], _[1], True)]
  unusevalues = lambda _: Value_St[(_[0], _[1], False)]

  P1 = np.apply_along_axis(usevalues,2,T)
  P2 = np.apply_along_axis(unusevalues,2,T)

  fig = plt.figure(figsize=(15, 7))
  ax = fig.add_subplot(111, projection='3d')
  ax.plot_surface(D,P,P1,color = 'aliceblue')
  ax.set_xlabel('Player Sum')
  ax.set_ylabel('Dealer Showing')
  ax.set_zlabel('Value')
  plt.title("After "+str(count)+" episodes , Usable ace")
  plt.show()

  fig = plt.figure(figsize=(15, 7))
  ax = fig.add_subplot(111, projection='3d')
  ax.plot_surface(D,P,P2,color = 'aliceblue')
  ax.set_xlabel('Player Sum')
  ax.set_ylabel('Dealer Showing')
  ax.set_zlabel('Value')
  plt.title("After "+str(count)+" episodes , no usable ace")
```

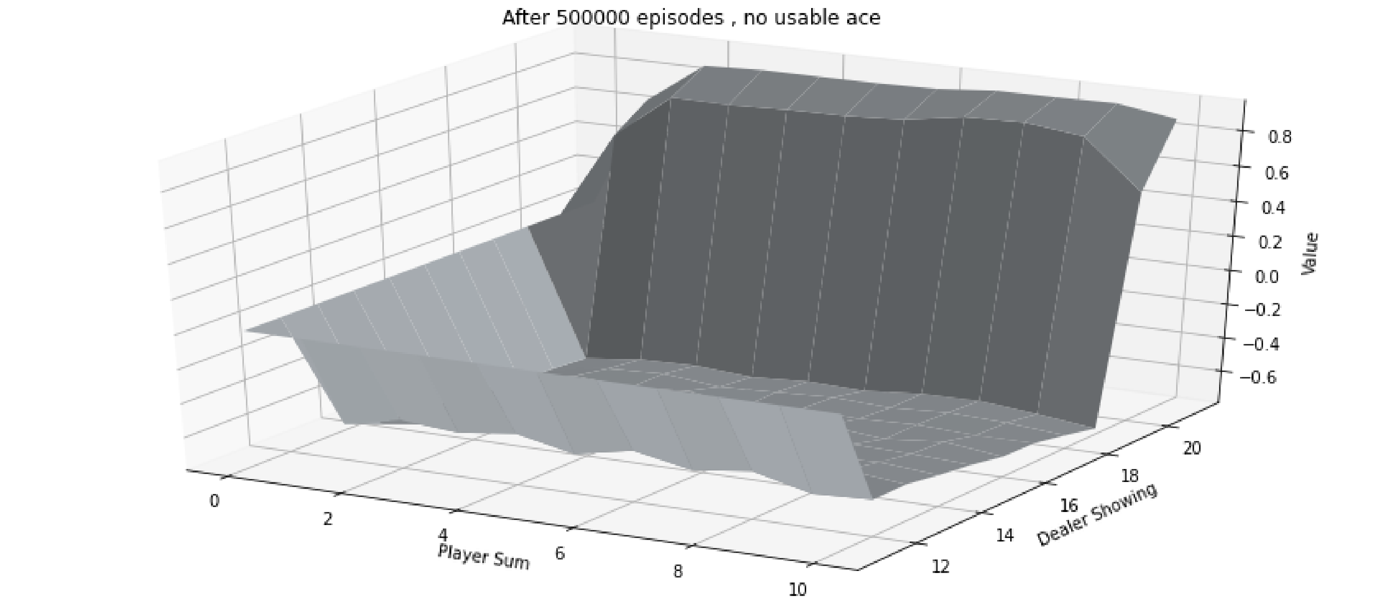
```
  plt.show()

def RemoveDict(V):
  for i in list(V.keys()):
    if i[0]>21:
      del V[i]

bj = BlackJack()

V1 = Monte_Carlo_Prediction(10000,Policy,bj,1)
RemoveDict(V1)
Plotting_Value_Function(V1,count=10000)

V2 = Monte_Carlo_Prediction(500000,Policy,bj,1)
RemoveDict(V2)
Plotting_Value_Function(V2,count=500000)
```

Output :






After 500000 episodes , Usable ace
Player Sum
Dealer Showing
Value


After 500000 episodes , no usable ace
Player Sum
Dealer Showing
Value

5.2

Code :

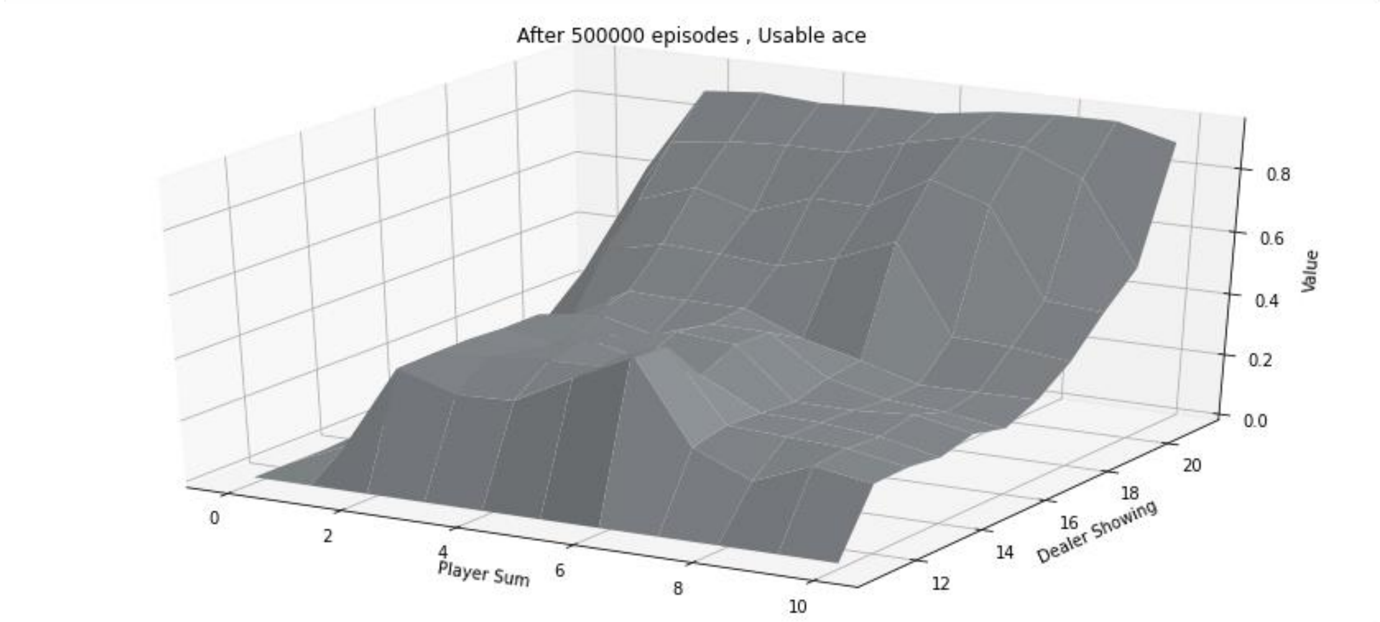
```
np.random.seed(42)

class BlackJack():
  def __init__(self):
    self.play = []
    self.deal = []
    self.decklist = list(np.arange(1,11)) + [10,10,10]
    self.start()

  def playcard(self):
    return np.random.choice(self.decklist)

  def playhand(self):
      return [self.playcard(),self.playcard()]

  def currentusability(self,draw):
    total_sum = sum(draw)
    if 1 in draw:
      if total_sum + 10<=21:
        return True
      else :
        return False
    else :
      return False

  def currentsum(self,draw):
    if self.currentusability(draw)==True:
      total_sum = sum(draw) + 10
    else :
      total_sum = sum(draw)
    return total_sum

  def currentbust(self,draw):
    total_sum = sum(draw)
    if total_sum>21:
      return True
    else :
      return False

  def score(self,draw):
    if self.currentbust(draw)==True:
      return 0
    else :
      return self.currentsum(draw)

  def step(self, action):

    end = True
    # Hit : Player turn
    if action==1:
```

```
            self.play.append(self.playcard())
            if self.currentbust(self.play):
                reward = -1
            else:
                end = False
                reward = 0

        # Stick: Dealers turn
        else:

            while self.currentsum(self.deal) < 17:
                self.deal.append(self.playcard())
            playsc = self.score(self.play)
            dealsc = self.score(self.deal)
            if playsc == dealsc:
              reward = 0
            elif playsc > dealsc:
              reward = 1
            else :
              reward = 0
        observation = [self.currentsum(self.play), self.deal[0], self.currentusability(
self.play)]
        step_results =  [observation , reward , end , {}]
        return step_results

    def start(self):
        self.deal = self.playhand()
        self.play = self.playhand()
        while self.currentsum(self.play) < 12:
            self.play.append(self.playcard())

        observation = [self.currentsum(self.play), self.deal[0], self.currentusabilit
y(self.play)]
        return observation

def Monte_Carlo_Control(Episode_Lenght,Policy,BlackJact_Env,Alpha,Epislon):
    Return_St = defaultdict(float)
    Count_St = defaultdict(float)
    Q_St = defaultdict(float)

    for i in range(Episode_Lenght):
      # Generating Episode (S,A,R)
      Episode = []
      State_Action = []
      S0 = BlackJact_Env.start()
      for _ in range(1000):
          ActionProb = Policy(S0,Q_St,Epislon)
          A0 = np.random.choice(2,p = ActionProb)
          observation = BlackJact_Env.step(A0) # (State,Reward,End)
          Episode.append([observation[0],observation[1],observation[2]])
          State_Action.append((tuple(observation[0]),A0))
          if observation[2]==True:
              break
          S0 = observation[0]
```

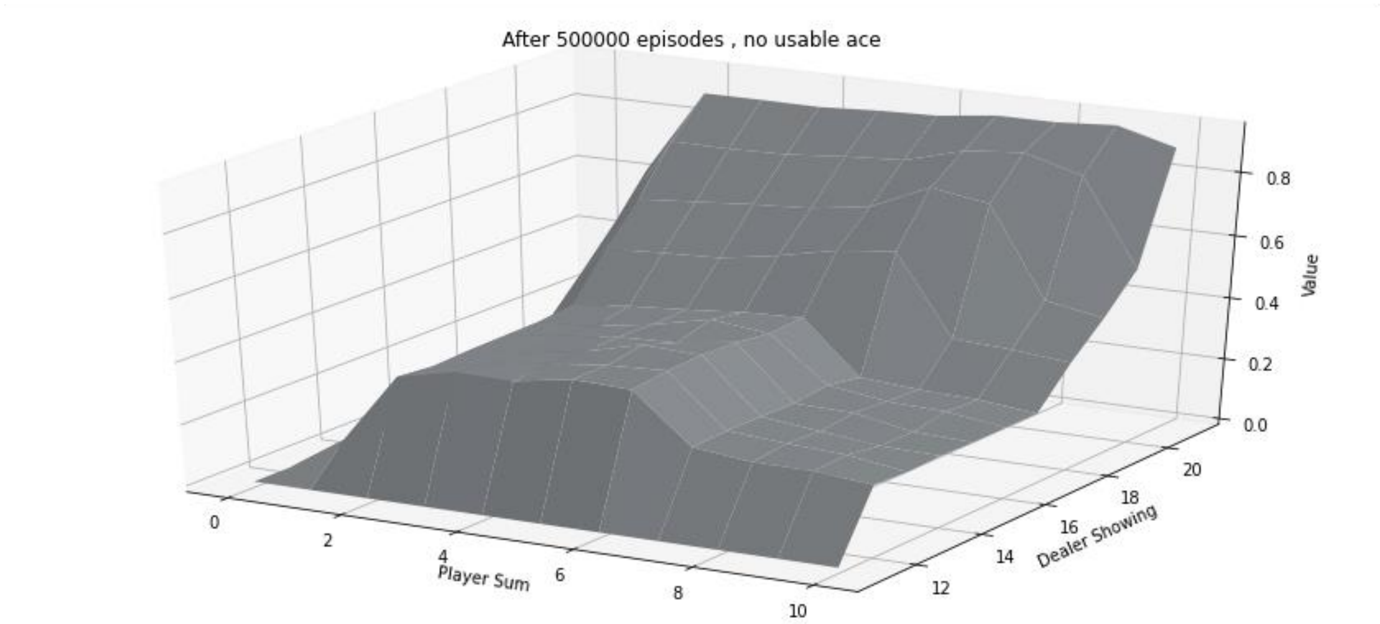
```
      G = 0
      States_Action_Visited = []
      for i in range(len(Episode)-1,-1,-1):
        s = State_Action[i][0]
        a = State_Action[i][1]
        G = Alpha*G + Episode[i][1]
        if State_Action[i] in States_Action_Visited:
          continue
        else :
          Return_St[(s,a)] = Return_St[(s,a)] + G
          Count_St[(s,a)] = Count_St[(s,a)] + 1
          Q_St[(s,a)] = Return_St[(s,a)]/Count_St[(s,a)]
          States_Action_Visited.append((s,a))

    return Q_St

def NewPolicy(State,Q_St,Epislon):
    keys = list(Q_St.keys())
    Q_value_0 = Q_value_1 = 0
    if len(Q_St):
      Pi = np.array([0.5,0.5])
    else:
      if (State,0) in keys:
        Q_value_0 = Q[(State,0)]
      if (State,1) in keys:
        Q_value_1 = Q[(State,1)]
      Q_value = np.array([Q_value_0,Q_value_1])
      Optimal_Act = np.argmax(Q_value)
      Pi = np.zeros(2)
      Pi[Optimal_Act] = (1-Epislon) + (Epislon/2)
      Pi[1-Optimal_Act] = (Epislon/2)
    return Pi

def Plotting_Value_Function(Value_St,count):
  Play_show = [i[0] for i in Value_St.keys()]
  Deal_show = [i[1] for i in Value_St.keys()]

  Play_range = np.arange(min(Play_show)-1,max(Play_show)+1)
  Deal_range = np.arange(min(Deal_show)-1,max(Deal_show)+1)
  P, D = np.meshgrid(Play_range,Deal_range)
  T = np.dstack([P, D])

  # Usable and usable ace
  usevalues = lambda _: Value_St[(_[0], _[1], True)]
  unusevalues = lambda _: Value_St[(_[0], _[1], False)]

  P1 = np.apply_along_axis(usevalues,2,T)
  P2 = np.apply_along_axis(unusevalues,2,T)

  fig = plt.figure(figsize=(15, 7))
  ax = fig.add_subplot(111, projection='3d')
  ax.plot_surface(D,P,P1,color = 'aliceblue')
  ax.set_xlabel('Player Sum')
```

```
  ax.set_ylabel('Dealer Showing')
  ax.set_zlabel('Value')
  plt.title("After "+str(count)+" episodes , Usable ace")
  plt.show()

  fig = plt.figure(figsize=(15, 7))
  ax = fig.add_subplot(111, projection='3d')
  ax.plot_surface(D,P,P2,color = 'aliceblue')
  ax.set_xlabel('Player Sum')
  ax.set_ylabel('Dealer Showing')
  ax.set_zlabel('Value')
  plt.title("After "+str(count)+" episodes , no usable ace")
  plt.show()

def GetValueFromStateValue(Q_St):
  Q_df = pd.DataFrame()
  state = []
  action = []
  value = []
  for i in list(Q1.keys()):
    key = i
    state.append(key[0])
    action.append(key[1])
    value.append(Q1[i])

  Q_df['State'] = state
  Q_df['Action'] = action
  Q_df['Value'] = value
  Q_df.sort_values(by=['State', 'Action'],inplace =True)

  i = 0
  V1 = defaultdict(float)
  while i<len(Q_df)-1:
    if Q_df.iloc[i]['State'] == Q_df.iloc[i+1]['State']:
      V1[Q_df.iloc[i]['State']] = max(Q_df.iloc[i]['Value'],Q_df.iloc[i]['Value'])
      i = i + 2
    else :
      V1[Q_df.iloc[i]['State']] = Q_df.iloc[i]['Value']
      i = i + 1

  return V1

def RemoveDict(V):
  for i in list(V.keys()):
    if i[0]>21:
      del V[i]
```

Output :



After 500000 episodes , Usable ace
0
2
4
6
8
10
Player Sum
12
14
16
18
20
Dealer Showing
0.0
0.2
0.4
0.6
0.8
Value



After 500000 episodes , no usable ace
0
2
4
6
8
10
Player Sum
12
14
16
18
20
Dealer Showing
0.0
0.2
0.4
0.6
0.8
Value

5.

5. In Monte Carlo method, the algorithm needs whole episode walk. The episode is based on the random ε soft policy which is completely different the true policy.

Temporal difference method uses the other estimates without waiting for final outcome to update the estimate.

In case of driving home example, the monte carlo method update the estimates only once when the car arrived at home. while in case of temporal difference learning, the algorithm updates the estimate after the car enters the highway. The methode TD must be chosen here it guides the update process from the initial phase of driving it self.

Hence, the TD updates are better than MC updates.

The same sort of thing will happen the original task as well provided

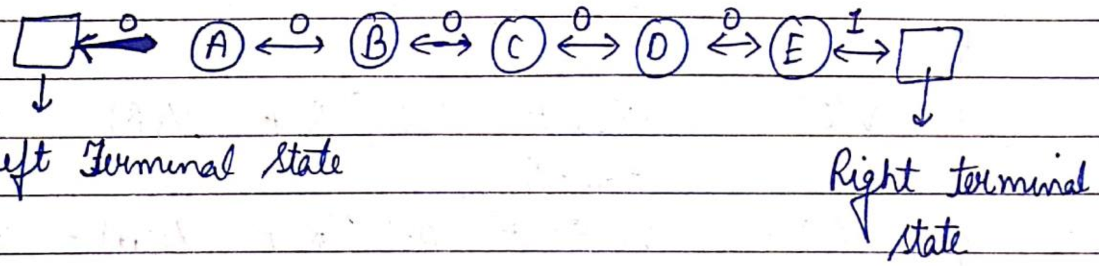
→ accurate state information

→ accurate initilization of state values.

6.

6. (6.3) → As given in the graph, during the first episode, only $V(A)$ is changed. The above observation ~~about~~ tells the following things that happened in the first episode:

→ The episode may start from any non terminal state but the terminal state is left terminal state.



→ So, reward is 0.

→ The value of $V(A)$ is low during the first episode.

→ Given values: $\alpha = 0.1$

$\gamma = 1$

$V(s) \leftarrow 0.5$ for all $s \in S$

Calculating $V(A) \Rightarrow$

$$V(s) = V(s) + \alpha [R + \gamma V(s') - V(s)]$$

$V(A) = 0.5 + 0.1[0 + 1 \times 0 - 0.5]$
$= 0.45$

Value change in $V(A) = 0.5 - 0.45$
$= 0.05.$

$V(A)$ is lowered by 0.05.

→ Consider the simple episode walk starting from 0 as
D, 0, C, 0, B, 0, A, 0, Left Terminal State]

$V(D) = V(D) + [R + 1 \times V(C) - V(D)] * 0.1$
$= 0.5 + 0.1[0 + 0.5 - 0.5]$
$= 0.5 + 0.1[0]$
$= 0.5$

$V(C) = V(C) + 0.1[R + V(B) - V(C)]$
$= 0.5$
$V(B) = V(B) + 0.1[R + V(A) - V(B)]$
$= 0.5$

From above calculations, it is clear that only $V(A)$ is change and lowered by 0.05 while other remains same during first episode.

In general
$V(S) = V(S) + \alpha[R + \gamma V(S') - V(S)]$
$= 0.5 + 0.1[0 + 1 \times 0.5 - 0.5]$
$= 0.5$ for all $s \in [B, C, D, E]$

(6.4) On the basis of given graph of random walk, the performance of algorithm change with change in value of step size.

Both the algorithms performs bad if the value of α is too high or very low. But the wider values of $\alpha \in [0,1]$ will not affects the plots much.

When the α value is high, then
- → It perform good in the beginning of the episode.
- → The RMSE error will increase after some episodes.

For the fixed value of α, the TD algorithm performs better than MC algorithm because
- → Updates the value function as one step dynamic.
- → Only need the one advance next state for computation
- → This gives importance to older values.

(6.5) In the given random walk, the value of states is taken to be 0.5 initially. ~~This is~~ This value is more close to the actual value of the state. When the episodes starts, the RMSE goes down because the error in actual and predicted values is less. As the episcodes passes, the TD ~~alg~~ algorithm updates the values of state. This will lead to the ~~At~~ change in RMSE value i.e. it will increase.

No, I don't think that this will always occur. It will change if the initially values are not close to the actual values of state. If the initialization is more random, then the RMSE curve will go from high values to low value and then algorithm will converge.

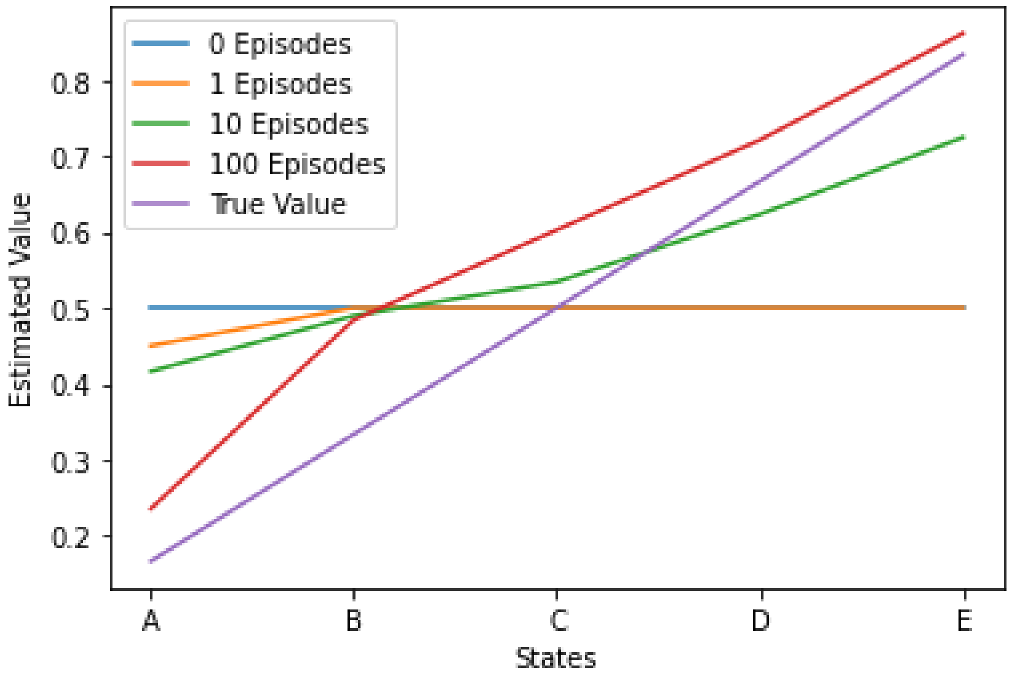
Code :

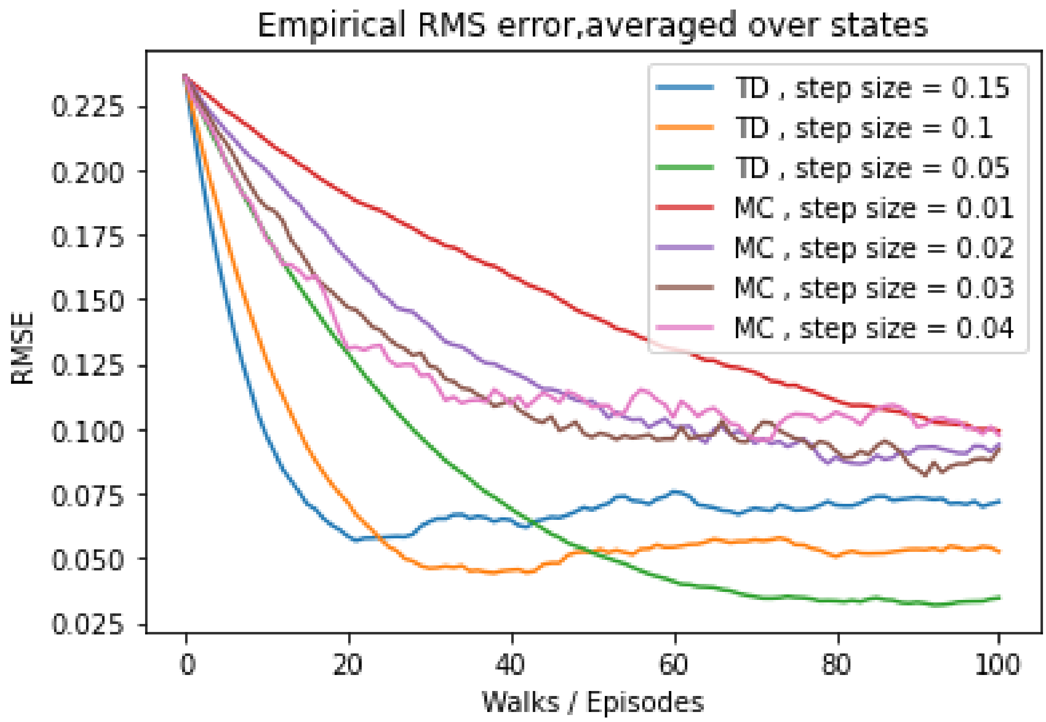
```
class Random_Walk:

  def __init__(self):
    self.V_Est = np.array([0.5]*7)
    self.V_True = np.array([0,1/6,2/6,3/6,4/6,5/6,1])
    self.Action = [0,1]

  def TD(self,initial_state,step_size,discounting):
      episode = [initial_state]
      received_rewards = []
      state = initial_state
      while True:
        a = np.random.binomial(1,0.5)
        if a == self.Action[1]:
          new_state = state + 1
          if state ==6:
            reward = 1
          else :
            reward = 0
        else:
          new_state = state - 1
          reward = 0
        episode.append(new_state)
        received_rewards.append(reward)

        self.V_Est[state] = self.V_Est[state] + step_size*(reward + discounting*sel
f.V_Est[new_state] - self.V_Est[state])
        state = new_state
        if state in [0,6]:
            break
      return [episode,received_rewards]

  def MC(self,initial_state,step_size , discounting):
    episode = [initial_state]
    received_rewards = []
    state = initial_state

    while True:
      a = np.random.binomial(1,0.5)
      if a == self.Action[1]:
        new_state = state + 1
        if state ==6:
          reward = 1
        else :
          reward = 0
      else:
        new_state = state - 1
        reward = 0

      episode.append(new_state)
      received_rewards.append(reward)
```

```
      state = new_state
      if state in [0,6]:
        break

    G = 0
    for i in range(len(received_rewards)-1,-1,-1):
      G = G + discounting*received_rewards[i]
      self.V_Est[episode[i]] = (1-step_size)*self.V_Est[episode[i]] + step_size*G

      return  [episode,received_rewards]

def TD_Epsiode(Number_Episodes,step_size,discouting):
  V_Est_Epi_mean_TD_Res = []
  for i in Number_Episodes:
    V_Est_Epi = []
    RMC = Random_Walk()
    for _ in range(i):
      RMC.TD(3,step_size=step_size, discounting=discouting)
      V_Est_Epi.append(RMC.V_Est)
    V_Est_Epi = np.array(V_Est_Epi)
    V_Est_Epi_mean_TD = np.mean(V_Est_Epi,axis=0)
    V_Est_Epi_mean_TD_Res.append(V_Est_Epi_mean_TD)

  return V_Est_Epi_mean_TD_Res

def MC_Epsiode(Number_Episodes,step_size,discouting):
  V_Est_Epi_mean_MC_Res = []
  for i in Number_Episodes:
    V_Est_Epi = []
    RMC = Random_Walk()
    for _ in range(i):
      RMC.MC(3,step_size=step_size, discounting=discouting)
      V_Est_Epi.append(RMC.V_Est)
    V_Est_Epi = np.array(V_Est_Epi)
    V_Est_Epi_mean_MC = np.mean(V_Est_Epi,axis=0)
    V_Est_Epi_mean_MC_Res.append(V_Est_Epi_mean_MC)

  return V_Est_Epi_mean_MC_Res

def TD_Plot(V_Est , V_True, V_Est_Epi_mean_TD_Res,Number_Episodes):

  for i in range(len(Number_Episodes)):
    if Number_Episodes[i]==0:
      plt.plot(V_Est[1:6],label=str(Number_Episodes[i])+' Episodes')
    else:
      plt.plot(V_Est_Epi_mean_TD_Res[i][1:6],label=str(Number_Episodes[i])+' Episod
es')
  plt.plot(V_True[1:6],label='True Value')
  plt.xlabel('States')
  plt.ylabel('Estimated Value')
  plt.xticks(np.arange(5), ['A', 'B', 'C','D','E'])
  plt.legend()
  plt.show()
```

```python
def RMSE_Calculation(Step_MC,Step_TD,Episodes,V_True):
  MC_RMSE = {}
  for i in Step_MC:
    error_per_step = []
    step_size = i
    for j in range(1,Episodes):
      V = MC_Epsiode([j],step_size,discouting=0.9)
      error = math.sqrt(mean_squared_error(V_True[1:6],V[0][1:6]))
      error_per_step.append(error)
    error_per_step.reverse()
    MC_RMSE[i] =  error_per_step

  TD_RMSE = {}
  for i in Step_TD:
    error_per_step = []
    step_size = i
    for j in range(1,Episodes):
      V = TD_Epsiode([j],step_size,discouting=0.9)
      error = math.sqrt(mean_squared_error(V_True[1:6],V[0][1:6]))
      error_per_step.append(error)
    error_per_step.reverse()
    TD_RMSE[i] =  error_per_step

  return (MC_RMSE , TD_RMSE)

def RMSE_Plot(MC_RMSE,TD_RMSE):

  for i in list(MC_RMSE.keys()):
    plt.plot(MC_RMSE[i],label='MC , step size = '+str(i))
  for i in list(TD_RMSE.keys()):
    plt.plot(TD_RMSE[i],label='TD , step size = '+str(i))

  plt.xlabel('Walks / Episodes')
  plt.ylabel('RMSE')
  plt.title('Empirical RMS error,averaged over states')
  plt.legend()
  plt.show()

Number_Episodes = [0,1,10,100]
step_size = 0.1
discouting = 0.9
V_Est_Epi_mean_TD_Res = TD_Epsiode(Number_Episodes,step_size,discouting)
V_True = np.array([0,1.6,2/6,3/6,4/6,5/6,1])
V_Est = np.array([0.5]*7)
TD_Plot(V_Est , V_True, V_Est_Epi_mean_TD_Res,Number_Episodes)

Step_MC = [0.01,0.02,0.03,0.04]
Step_TD = [0.1,0.05,0.15]
(MC_RMSE , TD_RMSE) = RMSE_Calculation(Step_MC,Step_TD,100,V_True)
RMSE_Plot(MC_RMSE,TD_RMSE)
```

Output :


0 Episodes
1 Episodes
10 Episodes
100 Episodes
True Value
Estimated Value
0.2
0.3
0.4
0.5
0.6
0.7
0.8
A
B
C
D
E
States


Empirical RMS error,averaged over states
TD , step size = 0.15
TD , step size = 0.1
TD , step size = 0.05
MC , step size = 0.01
MC , step size = 0.02
MC , step size = 0.03
MC , step size = 0.04
RMSE
0.025
0.050
0.075
0.100
0.125
0.150
0.175
0.200
0.225
0
20
40
60
80
100
Walks / Episodes

7.

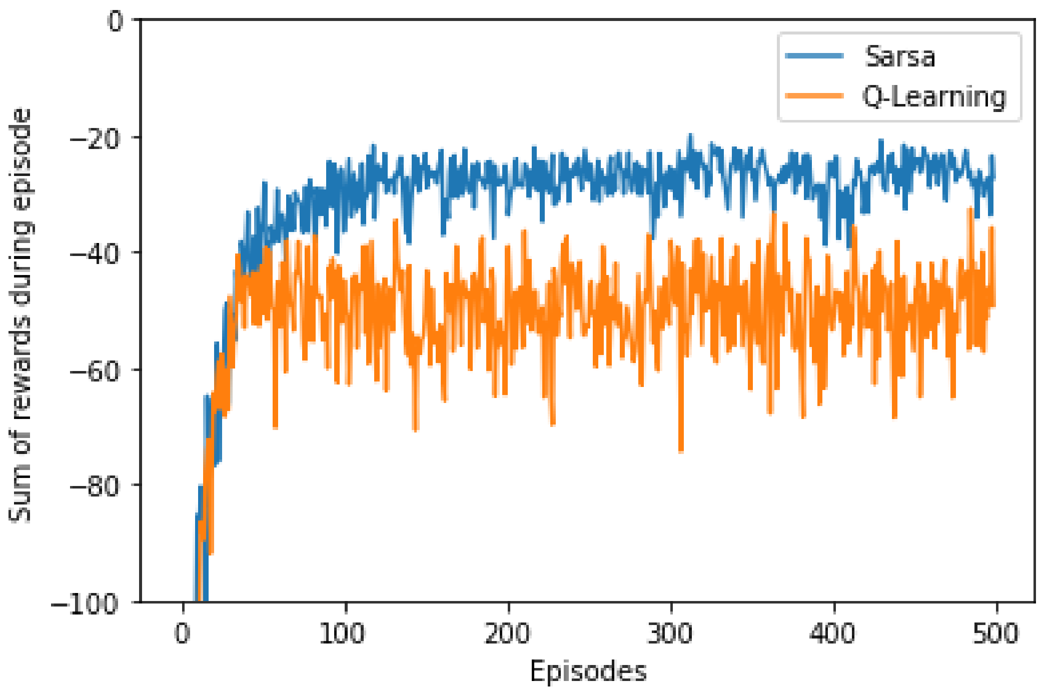
Code :

```
np.random.seed(40)
class CliffWalking:

  def __init__(self,gridsize,stepsize,epislon,gamma,action,startstate,goalstate):
    self.gridsize = gridsize
    self.stepsize = stepsize
    self.epislon = epislon
    self.gamma = gamma
    self.action = action
    self.startstate = startstate
    self.goalstate = goalstate
    self.Q_St = np.zeros((gridsize[0],gridsize[1],4))
    self.Q_St[startstate[0],startstate[1]] = 0
    self.Q_St[goalstate[0],goalstate[1]] = 0

  def NextStateReward(self,State,Action):
    h = self.gridsize[0]
    w = self.gridsize[1]
    if Action==0: # UP
      i = State[0]-1
      j = State[1]
      r = -1
    elif Action==1: # DOWN
      i = State[0]+1
      j = State[1]
      r = -1
    elif Action==2: # LEFT
      i = State[0]
      j = State[1]-1
      r = -1
    elif Action==3: # DOWN
      i = State[0]
      j = State[1]+1
      r = -1

    if i<0:
      i = 0
    elif i>=h:
      i = h-1

    if j<0:
      j = 0
    elif j>=w:
      j = w-1

    cliff = [k for k in range(1,11)]
    if i==3 and j in cliff:
      NextState = self.startstate
      Reward = -100
    else :
```

```
      NextState = [i,j]
      Reward = r

    return (NextState,Reward)

  def Policy(self,State):
    p = np.random.random()
    if p<self.epislon:
      Action = np.random.choice(self.action)
    else :
      Q_Values = self.Q_St[State[0],State[1]]
      Action = np.argmax(Q_Values)
    return Action

  def Sarsa(self,InitialState):  # One episode
      S = InitialState
      A = self.Policy(S)
      Total_reward = 0
      c = 0
      while S!=self.goalstate:
        c = c + 1
        if c==100:
          break
        (S_dash,R) = self.NextStateReward(S,A)
        A_dash = self.Policy(S_dash)
        for i in range(4):
          self.Q_St[S[0],S[1],i] = self.Q_St[S[0],S[1],i] + self.stepsize*(R + self.gamma*self.Q_St[S_dash[0],S_dash[1],i] - self.Q_St[S[0],S[1],i])
        S = S_dash
        A = A_dash
        Total_reward = Total_reward + R
      return Total_reward

  def Q_Learning(self,InitialState):  # One episode
      S = InitialState
      A = self.Policy(S)
      Total_reward = 0
      c = 0
      while S!=self.goalstate:
        c = c + 1
        if c==100:
          break
        (S_dash,R) = self.NextStateReward(S,A)
        A_dash = self.Policy(S_dash)
        Max_Q_Values_NextState = max([self.Q_St[S_dash[0],S_dash[1],i] for i in range(4)])
        for i in range(4):
          self.Q_St[S[0],S[1],i] = self.Q_St[S[0],S[1],i] + self.stepsize*(R + self.gamma*Max_Q_Values_NextState - self.Q_St[S[0],S[1],i])
        S = S_dash
        A = A_dash
        Total_reward = Total_reward + R
      return Total_reward
```

```
def Plot_Sarsa_Q_Learning_Curve(gridsize,stepsize,epislon,gamma,action,startstate,g
oalstate,numberepisodes):
  sarsa_reward = np.zeros(numberepisodes)
  qlearning_reward = np.zeros(numberepisodes)

  sar = CliffWalking(gridsize,stepsize,epislon,gamma,action,startstate,goalstate)
  ql = CliffWalking(gridsize,stepsize,epislon,gamma,action,startstate,goalstate)

  InitialState = startstate
  for i in range(numberepisodes):
    sarsa_reward[i] = sarsa_reward[i] + sar.Sarsa(InitialState)
    qlearning_reward[i] =  qlearning_reward[i] + ql.Q_Learning(InitialState)

  plt.plot(sarsa_reward,label='Sarsa')
  plt.plot(qlearning_reward,label='Q-Learning')
  plt.xlabel('Episodes')
  plt.ylabel('Sum of rewards during episode')
  plt.legend()
  plt.show()
  return sarsa_reward , qlearning_reward


gridsize = [4,12]
stepsize = 0.5
epislon = 0.1
gamma = 1
action = np.arange(4)
startstate = [3, 0]
goalstate = [3, 11]
numberepisodes = 500
sarsa_reward , qlearning_reward = Plot_Sarsa_Q_Learning_Curve(gridsize,stepsize,epi
slon,gamma,action,startstate,goalstate,numberepisodes)
```

Output :



8.

8. Consider the update equation of SARSA and Qlearning as

$$Q(S_t, A_t) = Q(S_t, A_t) + \alpha (R + \gamma Q(S_{t+1}, A_{t+1}) - Q(S_t, A_t))$$

→ For SARSA

$$Q(S_t, A_t) = Q(S_t, A_t) + \alpha (R + \gamma \max_{a_{t+1} \in A(S_{t+1})} (S_{t+1}, a_{t+1}) - Q(S_t, A_t))$$

→ For Qlearning

→ If the learning update in SARSA is greedy, then it will behave like Qlearning algorithm.

i.e. the next action $A_{t+1} = \underset{a_{t+1} \in A(S_{t+1})}{\arg\max} Q(S_{t+1}, a_{t+1})$

where the policy used is behaviour policy.

→ Yes, the ~~algorithm~~ algorithm makes exactly the same action selections and weight updates.

→ Yes, there is wrong in this type of learning.

Reasons:

- The algorithm will not explore and always take actions greedly.
- If multiple sub optimal policy exist, the algorithm will find one but might not lead to global optimal policy.
- In some setting, the policy improvement will not happen.