# CSE 546: Assignemt1 Final Submission

**Name: K Naga Kiran Reddy** **UB ID: 50432242**

# "I certify that the code and data in this assignment were generated independently, using only the tools and resources defined in the course and that I did not receive any external help, coaching or contributions during the production of this work."

**1. Describe the deterministic and stochastic environments, which were defined (set of actions/states/rewards, main objective, etc).**

In a Deterministic environment p(s ’, r|s, a) = {0, 1}

In a Stochastic environment Σ(p(s', r/s, a) = 1

Both deterministic and stochastic environments have

**Actions:** {down, up, right, left}

**States:** {s1, s2, s3, s4, s5....s16} -> 4x4 grid

**Rewards:** {-2, -1, 0,  3, 4, 100}

**Main Objective:** To run a random agent in both the environments for 30 timesteps or episodes and display the rewards acquired in the environments. And applying Q-Learning or SARSA to find the optimal path

**2. Provide visualizations of your environments**

**Starting position:** [0,0]

**Target position:** [3,3] #If agent reaches target position: Reward = 100

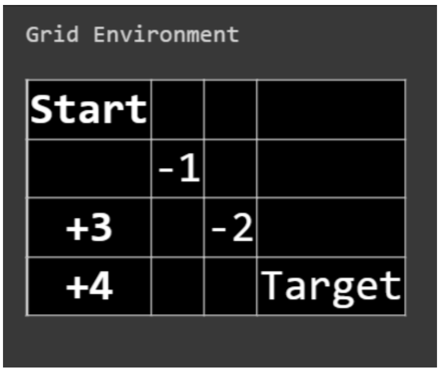
**Danger1 position:** [1,1] #If agent reaches danger1 position: Reward = -1

**Danger2 position:** [2,2] #If agent reaches danger2 position: Reward = -2

**Gold1 position:** [2,0] #If agent reaches gold1 position: Reward = +3

**Gold2 position:** [3,0] #If agent reaches gold2 position: Reward = +4

A reward of **zero** in all other cases. #Reward = 0



## Deterministic environment:

1. For all states in deterministic environment p(s', r/s, a) = {0,1}
2. So, by generating a random number between 0 - 1 and using epsilon = 0.8 I have checked if an action can be executed or not based on epsilon > rand_num: As for deterministic the probability is either 0 or 1
3. Ran the process for 30 steps, printing timesteps and rewards
4. The process of exploration is stopped if the random agent reaches target position **[3,3]**
5. Below are samples of deterministic environment with rewards: {-2, -1, 0, 3, 4, 100}

```
Timestep: 1, Reward: 0
Timestep: 2, Reward: 3
Timestep: 3, No action taken
Timestep: 4, Reward: 0
Timestep: 5, Reward: 3
Timestep: 6, No action taken
Timestep: 7, No action taken
Timestep: 8, Reward: 3
Timestep: 9, Reward: 4
Timestep: 10, Reward: 0
Timestep: 11, Reward: 0
Timestep: 12, Reward: 0
Timestep: 13, Reward: 0
Timestep: 14, Reward: -1
Timestep: 15, No action taken
Timestep: 16, No action taken
Timestep: 17, Reward: 0
Timestep: 18, Reward: 3
Timestep: 19, Reward: 0
Timestep: 20, Reward: -2
Timestep: 21, Reward: 0
Timestep: 22, Reward: 100
Target Reached!!
```

**Results:**

When random agent enters Danger1 position a reward of -1 is given and similarly if it enters Danger2 position a reward of -2 is given and as it reaches target position in $22^{th}$ timestep the process is stopped by giving agent a reward of 100.

**3. How did you define the stochastic environment?**

## Stochastic environment:

1. In stochastic environment Σ(p(s', r/s, a) = 1
2. I have chosen for every **Down action (epsilon1 = 0.7)** there is a transition probability of 0.7 to choose down state and 0.3 probability to choose up state (i.e., the state that lies above current state)
3. Examples:
   If random agent is in state s7: p(s11, -2/s7, down) = 0.7 and p(s3, 0/s7, down) = 0.3
   Σ(p(s', r/s, down) = 0.7 + 0.3 = 1
   S7: [1,2]
   S11: [2,2]
   S3: [0,2]

4. I have chosen for every **Right action (epsilon2 = 0.8)** there is a transition probability of 0.8 to choose right state and 0.2 probability to choose left state.
5. Examples:

If random agent in state s6: p(s7, 0/s6, right) = 0.8 and p(s5, 0/s6, right) = 0.2
Σ(p(s', r/s, right) = 0.8 + 0.2 = 1
s6: [1,1]
s7: [1,2]
s5: [1,0]

6. Ran the process for 30 steps, printing timesteps and rewards
7. The process of exploration is stopped if the random agent reaches target position **[3,3]**
6. Below are samples of stochastic environment with rewards: {-2, -1, 0,3, 4, 100}

```
Timestep: 1, Reward: 0
Timestep: 2, Reward: 0
Timestep: 3, Reward: 0
Timestep: 4, Reward: 0
Timestep: 5, Reward: 0
Timestep: 6, Reward: 0
Timestep: 7, Reward: 0
Timestep: 8, Reward: 0
Timestep: 9, Reward: 0
Timestep: 10, Reward: 0
Timestep: 11, Reward: 0
Timestep: 12, Reward: 0
Timestep: 13, Reward: 0
Timestep: 14, Reward: 0
Timestep: 15, Reward: 0
Timestep: 16, Reward: -1
Timestep: 17, Reward: 0
Timestep: 18, Reward: -1
Timestep: 19, Reward: 0
Timestep: 20, Reward: 3
Timestep: 21, Reward: 4
Timestep: 22, Reward: 0
Timestep: 23, Reward: 0
Timestep: 24, Reward: 0
Timestep: 25, Reward: 100
Target Reached!!
```

**Results:**

As it reaches target position in timestep 25, the process is stopped by giving agent a reward of 100.

**4. What is the difference between the deterministic and stochastic environments?**

The next state of the environment can be determined given the current state and action, such an environment is called deterministic environment. Whereas in stochastic environment the next state is random in nature as described above the next state can be determined using transition probabilities, so it’s very random for an agent to choose next state.

**5. Safety in AI: Write a brief review explaining how you ensure the safety of your environments.**

1) One way is to use clip method in the program to restrict the agent only to the given environment

2) Manually specifying the constraints on the policy’s behaviour

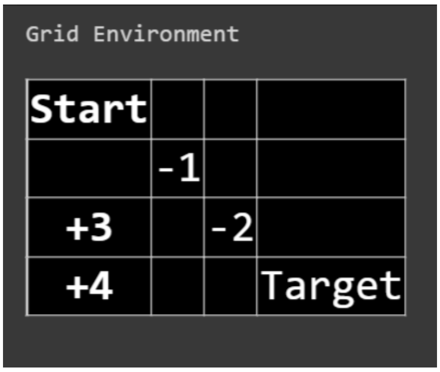
3) Choosing appropriate rewards, to avoid running an agent in loop

4) Ending the process whenever the agent reaches the target

**Part 2: Applying Tabular Methods:**

I have used **Q-Learning and SARSA tabular methods.**

I have defined my environment as below:




I have updated my checkpoint submission with 2 more rewards as I wanted to get a better understanding of the Q-Learning concept!! And to see how Q-table is updated for more rewards.

**Python code:**

**Libraries:**

```
import numpy as np
import matplotlib.pyplot as plt
import gym
from gym import spaces
from google.colab import widgets
import time
```

**Grid Environment code:**

```
class GridEnvironment(gym.Env):
    metadata = { 'render.modes': [] }

    def __init__(self):
        self.observation_space = spaces.Discrete(16) #Intializing a 4x4
 grid with 16 states: {s1, s2, s3,....s16}
        self.action_space = spaces.Discrete(4) #Intializing 4 actions:
 {0: down, 1: up, 2: right, 3: left}
        self.done = False
        #rewards = {-2, -1, 0, 3, 4, 100} defined below

    def reset(self):

        self.agent_pos = [0, 0] #start position
        self.goal_pos = [3, 3] #target position (+100)
        self.danger1_pos = [1,1] #first danger position (-1)
        self.danger2_pos = [2,2] #second danger position (-2)
```

```
        self.gold1_pos = [2,0] #First positive reward position (+3)
        self.gold2_pos = [3,0] #Second positive reward position (+4)
        self.done = False
        self.state = np.zeros((4,4))
        observation = self.state.flatten()
        return observation

    def step(self, action, deterministic = False, stochastic = False):

        #determinstic environment
        if deterministic == True:
          epsilon1, epsilon2 = 1, 1 #With probability 1 Agent chooses g
iven action

        #stochastic environment
        if stochastic == True:
          epsilon1, epsilon2 = 0.7, 0.8 #Transistion probabilites of 0.
7 and 0.8
          #Σ(p(s', r/s, down) = 0.7 + 0.3 = 1
          #Σ(p(s', r/s, right) = 0.8 + 0.2 = 1

        #Actions
        if action == 0: #down
            rand_num1 = np.random.random()
            if epsilon1 >= rand_num1:
              self.agent_pos[0] += 1
            else: #For all states in stochastic environment when Down a
ction is choosen: Agent chooses down state with a transition probabilit
y of 0.7 and up state with 0.3 transition probaility
              self.agent_pos[0] -= 1
              # print("Up state is choosen with 0.3 probability instead
 of Down")

        if action == 1: #up
            self.agent_pos[0] -= 1

        if action == 2: #right
            rand_num2 = np.random.random()
            if epsilon2 >= rand_num2:
              self.agent_pos[1] += 1
            else:#For all states in stochastic environment when Right a
ction is choosen: Agent chooses right state with a transition probabili
ty of 0.8 and left state with 0.2 transition probaility
              self.agent_pos[1] -= 1
              # print("Left state is choosen with 0.2 probability inste
ad of Right")

        if action == 3: #left
```

```
            self.agent_pos[1] -= 1

        self.agent_pos = np.clip(self.agent_pos, 0, 3) #ensuring agent 
doesn't go out of the grid #One of the ways to ensure safety in the env
ironment
        self.state = np.zeros((4,4))

        observation = self.state.flatten()

        reward = 0 #Intializing reward to zero
        if (self.agent_pos == self.goal_pos).all():
            reward = 100 #A reward of 5 if it reaches target position
            self.done = True

        #Rewards structure
        elif (self.agent_pos == self.danger1_pos).all():
            reward = -1 #A negative reward -
1 if it enters 1st danger position

        elif (self.agent_pos == self.danger2_pos).all():
            reward = -2 #A negative reward of -
2 if it enters 2nd danger position

        elif (self.agent_pos == self.gold1_pos).all():
            reward = 3 #A reward of +3 at [2,0]

        elif (self.agent_pos == self.gold2_pos).all():
            reward = 4 #A reward of +3 at [3,0]

        return reward, self.agent_pos, self.done

    def render(self):
        plt.imshow(self.state)
```

**Random Agent:**

```
class RandomAgent: #Definig the Random agent class
    def __init__(self, env):
        self.env = env
        self.observation_space = env.observation_space
        self.action_space = env.action_space

    def step(self, observation): #Just for reference I have developed a
ction space in algorithm #Random agent chooses random action every time
 we call step method
        return np.random.choice(self.action_space.n)
```

**Q-Learning:**

**Python Code:**

```
#Q-learning

def q_learning(deterministic = False, stochastic = False, evaluation_re
sults = False, qlearning_rewards = False):
  env = GridEnvironment()
  agent = RandomAgent(env)#creating a random agent to explore the given
 environments
  obs = env.reset()#resets the environment to its initial configuration

  #Displays grid in its intial configuration
  print("Grid Environment\n")
  output_grid2 = widgets.Grid(4, 4, header_row=True, header_column=True
,  style='background-color: black; font-size: 25px; color: white')
  print("\n")
  with output_grid2.output_to(0, 0):
      print("Start")
  with output_grid2.output_to(3, 3):
      print("Target")
  with output_grid2.output_to(1, 1):
      print("-1")
  with output_grid2.output_to(2, 2):
      print("-2")
  with output_grid2.output_to(2, 0):
      print("+3")
  with output_grid2.output_to(3, 0):
      print("+4")


  #Intialize parameters
  learning_rate = 0.15 #alpha
  discount_factor = 0.95 #how much weightage to put on future rewards
  det_epsilon = 0.99 # For all states in deterministic environment p(s'
, r/s, a) = {0, 1}: Either action taken or No action taken


  #Intial state
  current_state = 0 #s1
  action_val = [0,1,2,3]

  #Q table representing 16 rows: one for each state (i.e., 0,1,2,...15)
 -
> (i.e., s1, s2, s3,....s16) and 4 columns: one for each action (i.e.,
0,1,2,3) -> (down,up,right,left)
  # (0-15, 0-3) remember the dimension is one less
  q_table = np.zeros((16,4))
```

```
  #mapping next_state co-ordinates to q_table co-ordinates
  states = {(0,0): 0, (0,1): 1, (0,2): 2, (0,3): 3,
                (1,0): 4, (1,1): 5, (1,2): 6, (1,3): 7,
                (2,0): 8, (2,1): 9, (2,2): 10, (2,3): 11,
                (3,0): 12, (3,1): 13, (3,2): 14, (3,3): 15} #16 states

  #Empty lists to store values
  optimal = []
  reward_values = []
  total_timesteps = []
  epsilon_values = []
  eva_rewards = []

  done = False #signifies if agent reached terminal or not
  total_episodes = 1500
  eva_episodes = 10 #Used for evaluation
  avg_timesteps = 0
  epsilon = 1 #multiply by 0.995 for each episode(#after 20 iterations#
 or terminal state reached)
  decay_factor = (0.01/1)**(1/total_episodes)

  #For evaluation results
  if evaluation_results:
     total_episodes += eva_episodes
     print("Evaluation Results")

  for episode in range(1, total_episodes+1):

    obs = env.reset()
    current_state = 0
    total_rewards = 0
    timestep = 0

    while timestep < 20: #(i.e., considering untill the terminal is rea
ched or 20 timesteps completed)

      # deterministic: {0,1} probability for deterministic #stochastici
ty in choosing an action for stochastic
      rand_num = np.random.random()
      if det_epsilon > rand_num: #Choosing an action in deterministic e
nvironment

          #e - greedy algorithm
          rand_num = np.random.random()
          if epsilon > rand_num:
            action = np.random.choice(action_val)
          else:
```

```
            action = np.argmax(q_table[current_state]) #action in curre
nt state s with max_q value

          #Taking the action
          reward, next_state_pos, done = env.step(action, deterministic
, stochastic)

          next_state = states[tuple(next_state_pos)]

          #Choosing action with max Q value
          max_q_action = np.argmax(q_table[next_state])

          #Update function
          q_table[current_state][action] = q_table[current_state][actio
n] + learning_rate*(reward + discount_factor*q_table[next_state][max_q_
action] - q_table[current_state][action])

          if episode == total_episodes:
            optimal.append(current_state+1)

          total_rewards += reward #Captured all the rewards in each epi
sode
          timestep += 1 #Number of timesteps in each episode

          current_state = next_state #next_state is assigned to current
_state

          if done == True: #If terminal or target state reached then st
op the episode
            done = False
            break


    #Results after each episode
    avg_timesteps += timestep #Capturing all timesteps for all 100 epis
odes
    total_timesteps.append(avg_timesteps)

    reward_values.append(total_rewards) #Append rewards in every episod
e
    epsilon_values.append(epsilon) #Append epsilon values in every epis
ode


    if epsilon > 0.01: #keeping epsilon in [0.01 - 1] range as if it fa
lls below 0.01 it will exploit more: choosing best actions. We want our
 agent to explore a bit: choosing random actions
        epsilon = epsilon*decay_factor
```

```
    else:
        epsilon = 0.01


    if (episode % 100) == 0 and evaluation_results == False and qlearni
ng_rewards == False: #printing results for every 100 episodes
      print("Episode: {}, Rewards: {}, Average timesteps taken: {}, eps
ilon: {}".format(episode, total_rewards, avg_timesteps//100, epsilon))
      avg_timesteps = 0


    #evaluation results
    if evaluation_results:
      if episode > total_episodes - eva_episodes:
         eva_rewards.append(reward)


    #printing the optimal path in last episode
    if episode == total_episodes:
          print("Optimal Path: ")
          for i in optimal:
            print(i,"->", end = " ")
          print(next_state+1)


  #rewards
  if qlearning_rewards:
    return reward_values


  #Final Q - Table
  print("Q Table: \n", q_table)

  #Plotting the results
  #x, y co-ordinates
  x = [episode for episode in range(total_episodes)]
  yr = reward_values
  ye = epsilon_values
  yr_eva = eva_rewards
  x_eva = [episode for episode in range(eva_episodes)]


  if evaluation_results:
      #episodes vs rewards
      plt.plot(x_eva,yr_eva)
      plt.title("Rewards per episode")

  else:
```

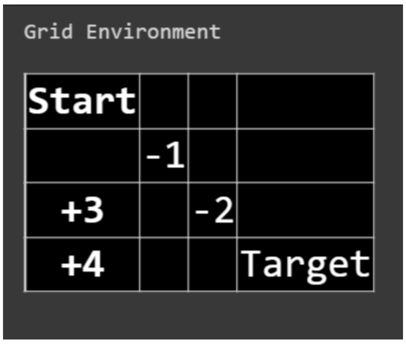
```
#Plots showing episodes vs epsilon, episodes vs rewards
fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(10,4))
#episodes vs epsilon
ax1.plot(x, ye)
ax1.set_title("Epsilon decay")

#episodes vs rewards
ax2.plot(x,yr)
ax2.set_title("Rewards per episode")
```

**Results:**

**Deterministic environment:**

```
q_learning(deterministic = True)
```

**Epsilon decay over episodes and optimal path:**



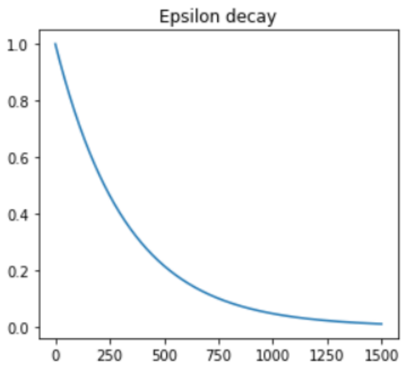

```
Episode: 100, Rewards: -8, Average timesteps taken: 17, epsilon: 0.7356422544596452
Episode: 200, Rewards: 105, Average timesteps taken: 13, epsilon: 0.5411695265464691
Episode: 300, Rewards: 99, Average timesteps taken: 10, epsilon: 0.3981071705535314
Episode: 400, Rewards: 114, Average timesteps taken: 9, epsilon: 0.2928644564625957
Episode: 500, Rewards: 105, Average timesteps taken: 8, epsilon: 0.2154434690031937 8
Episode: 600, Rewards: 107, Average timesteps taken: 7, epsilon: 0.1584893192461160 3
Episode: 700, Rewards: 107, Average timesteps taken: 6, epsilon: 0.1165914401179871 4
Episode: 800, Rewards: 107, Average timesteps taken: 6, epsilon: 0.0857695898590927 5
Episode: 900, Rewards: 107, Average timesteps taken: 6, epsilon: 0.0630957344480220 6
Episode: 1000, Rewards: 107, Average timesteps taken: 6, epsilon: 0.04641588833613003
Episode: 1100, Rewards: 107, Average timesteps taken: 6, epsilon: 0.03414548873833783
Episode: 1200, Rewards: 107, Average timesteps taken: 6, epsilon: 0.02511886431509727
Episode: 1300, Rewards: 107, Average timesteps taken: 6, epsilon: 0.01847849797422407
Episode: 1400, Rewards: 107, Average timesteps taken: 6, epsilon: 0.013593563908786174
Episode: 1500, Rewards: 107, Average timesteps taken: 6, epsilon: 0.010000000000000725
Optimal Path:
1 -> 5 -> 9 -> 13 -> 14 -> 15 -> 16
```

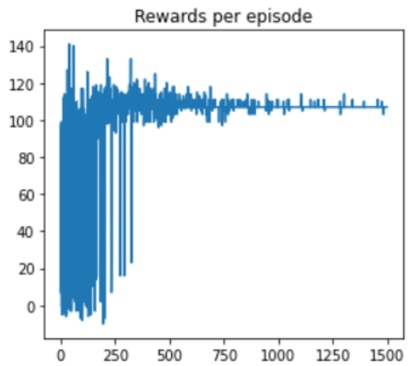
The above figure shows the agent reaching target following optimal policy. Below Q-table also depicts the same for state s1 the best action for next step is s5 and for s5 the next best action is down to s9 etc.

**Q-table:**

```
Q Table:
 [[ 83.83809375  79.64613987  75.66161841  79.64580562]
 [ 61.94830155  57.49184142  65.21250135  79.6460122 ]
 [ 79.85475586  54.9844894   49.34083253  40.89081383]
 [ 74.44315323  30.85193739  34.58632357  27.49620485]
 [ 88.250625    79.64482862  80.43913762  83.83407476]
 [ 85.7352073   68.77606301  74.32526986  80.02387753]
 [ 88.16438131  57.89723994  84.22841315  52.03936198]
 [ 94.63774927  36.80738687  73.71722697  67.43805889]
 [ 89.7375      83.83735864  85.73644087  88.24535363]
 [ 90.25        78.86898737  88.24818249  86.820698  ]
 [ 94.77606387  82.27141     94.99999319  84.3837372 ]
 [ 99.9999991   86.3209978   92.96362828  87.40484438]
 [ 89.73602083  88.24763423  90.25        89.73699335]
 [ 90.2497925   85.73730413  95.          89.737119  ]
 [ 94.99861557  88.2447925  100.          90.24952734]
 [  0.           0.           0.           0.        ]]
```

**Plots for epsilon decay and rewards per episode:**




**Observation:** Agent collect maximum rewards after training for some time as it learns optimal path that maximizes the reward**.**

**Stochastic environment**

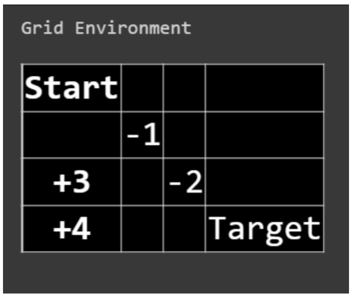
```
q_learning(stochastic= True)
```

**Epsilon decay over episodes:**

```
Episode: 100, Rewards: 1, Average timesteps taken: 19, epsilon: 0.7356422544596452
Episode: 200, Rewards: 13, Average timesteps taken: 19, epsilon: 0.5411695265464691
Episode: 300, Rewards: 54, Average timesteps taken: 20, epsilon: 0.39810717055350314
Episode: 400, Rewards: 40, Average timesteps taken: 20, epsilon: 0.29286445646252957
Episode: 500, Rewards: 58, Average timesteps taken: 20, epsilon: 0.21544346900319378
Episode: 600, Rewards: 32, Average timesteps taken: 20, epsilon: 0.15848931924611603
Episode: 700, Rewards: 70, Average timesteps taken: 20, epsilon: 0.11659144011798714
Episode: 800, Rewards: 61, Average timesteps taken: 20, epsilon: 0.08576958985909275
Episode: 900, Rewards: 58, Average timesteps taken: 20, epsilon: 0.06309573444802206
Episode: 1000, Rewards: 75, Average timesteps taken: 20, epsilon: 0.04641588833613003
Episode: 1100, Rewards: 71, Average timesteps taken: 20, epsilon: 0.03414548873833783
Episode: 1200, Rewards: 67, Average timesteps taken: 20, epsilon: 0.02511886431509727
Episode: 1300, Rewards: 59, Average timesteps taken: 20, epsilon: 0.01847849797422407
Episode: 1400, Rewards: 75, Average timesteps taken: 20, epsilon: 0.013593563908786174
Episode: 1500, Rewards: 75, Average timesteps taken: 20, epsilon: 0.010000000000000725
```

**Optimal path and Q-Table:**

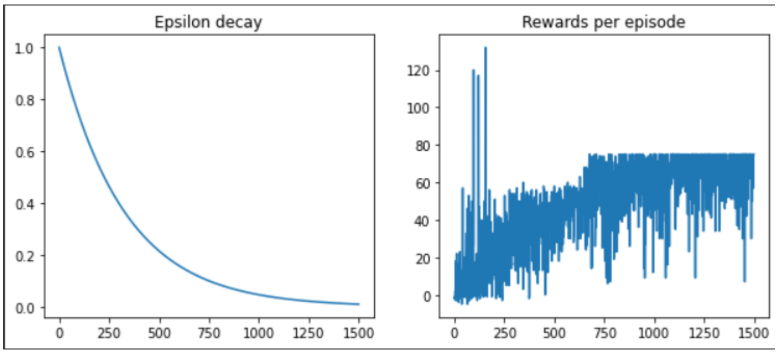
```
Optimal Path:
1 -> 5 -> 9 -> 13 -> 13 -> 13 -> 13 -> 13 -> 13 -> 13 -> 13 -> 13 -> 13 -> 13 -> 13 -> 13 -> 13 -> 13 -> 13 -> 13 -> 13
Q Table:
 [[ 7.00573194e+01  6.39357822e+01  5.76029417e+01  6.32312151e+01]
 [ 4.68566793e+01  4.52629225e+01  4.10300853e+01  6.19852377e+01]
 [ 1.98287496e+01  2.60136642e+01  1.37616897e+01  4.46787968e+01]
 [ 1.11662226e+00  5.86084617e+00  4.83606834e+00  3.07840401e+01]
 [ 7.53166920e+01  6.41931218e+01  6.36996547e+01  6.72075001e+01]
 [ 5.19063586e+01  4.53571712e+01  3.70731796e+01  6.84332389e+01]
 [ 8.31051416e+00  1.74927202e+01  1.66807982e+00  4.61372184e+01]
 [ 1.38026520e-05  7.82672352e+00  1.70086718e+00  1.63997961e+00]
 [ 7.85962008e+01  6.91984848e+01  6.89474283e+01  7.09871396e+01]
 [ 5.45713728e+01  5.00099195e+01  4.00687263e+01  7.49870597e+01]
 [ 1.56429770e+01  1.86986411e+01  3.22490533e+00  4.61607734e+01]
 [ 7.43283322e-01  0.00000000e+00 -3.00000000e-01  0.00000000e+00]
 [ 7.90421170e+01  7.63605904e+01  7.65623058e+01  8.00000000e+01]
 [ 2.15594895e+01  4.25072614e+01  1.72202893e+01  7.99999597e+01]
 [ 5.28726682e+00 -9.74303015e-04  4.41955547e+01  5.91081369e+00]
 [ 0.00000000e+00  0.00000000e+00  0.00000000e+00  0.00000000e+00]]
```



**(i.e., for reference)**

Here optimal path is different because as random agent is going to different states other than the best states because of stochastic nature of the environment.

**Plots for epsilon decay and rewards per episode:**




**Note:** Agent collect maximum rewards after training for some time as it learns optimal path that maximizes the reward. The rewards are uneven over the episodes because of stochastic nature of the environment.

**SARSA:**

**Python code:**

```
#SARSA
def sarsa(deterministic = False, stochastic = False, evaluation_results = False, sarsa_rewards = False):
  env = GridEnvironment()
  agent = RandomAgent(env)#creating a random agent to explore the given environments
  obs = env.reset()#resets the environment to its initial configuration

  #Displays grid in its intial configuration
  print("Grid Environment\n")
  output_grid2 = widgets.Grid(4, 4, header_row=True, header_column=True, style='background-color: black; font-size: 25px; color: white')
  print("\n")
  with output_grid2.output_to(0, 0):
      print("Start")
  with output_grid2.output_to(3, 3):
      print("Target")
```

```
  with output_grid2.output_to(1, 1):
      print("-1")
  with output_grid2.output_to(2, 2):
      print("-2")
  with output_grid2.output_to(2, 0):
      print("+3")
  with output_grid2.output_to(3, 0):
      print("+4")


  #Intialize parameters
  learning_rate = 0.15 #alpha
  discount_factor = 0.95 #how much weightage to put on future rewards
  det_epsilon = 0.99 # For all states in deterministic environment p(s'
, r/s, a) = {0, 1}: Either action taken or No action taken


  #Intial state
  current_state = 0 #s1
  action_val = [0,1,2,3]

  #Q table representing 16 rows: one for each state (i.e., 0,1,2,...15)
 -
> (i.e., s1, s2, s3,....s16) and 4 columns: one for each action (i.e.,
0,1,2,3) -> (down,up,right,left)
  # (0-15, 0-3) remember the dimension is one less
  q_table = np.zeros((16,4))

  #mapping next_state co-ordinates to q_table co-ordinates
  states = {(0,0): 0, (0,1): 1, (0,2): 2, (0,3): 3,
                (1,0): 4, (1,1): 5, (1,2): 6, (1,3): 7,
                (2,0): 8, (2,1): 9, (2,2): 10, (2,3): 11,
                (3,0): 12, (3,1): 13, (3,2): 14, (3,3): 15} #16 states

  #Empty lists to store values
  optimal = []
  reward_values = []
  total_timesteps = []
  epsilon_values = []
  eva_rewards = []


  done = False #signifies if agent reached terminal or not
  total_episodes = 1500
  eva_episodes = 10
  avg_timesteps = 0
  epsilon = 1 #multiply by 0.995 for each episode(#after 30 iterations#
 or terminal state reached)
  decay_factor = (0.01/1)**(1/total_episodes)
```

```
  #For evaluation results
  if evaluation_results:
     total_episodes += eva_episodes
     print("Evaluation Results")

  for episode in range(1, total_episodes+1):

    obs = env.reset() #resets the environment
    current_state = 0
    total_rewards = 0
    timestep = 0


    #e - greedy algorithm to choose s and a
    rand_num = np.random.random()
    if epsilon > rand_num:
      action = np.random.choice(action_val)
    else:
      action = np.argmax(q_table[current_state]) #action in current sta
te s with max_q value

    while timestep < 20: #(i.e., considering untill the terminal is rea
ched or 20 timesteps are completed)

      # deterministic: {0,1} probability for deterministic #stochastici
ty in choosing an action for stochastic
      rand_num = np.random.random()
      if det_epsilon > rand_num: #Choosing an action in deterministic e
nvironment

          reward, next_state_pos, done = env.step(action, deterministic
, stochastic)
          next_state = states[tuple(next_state_pos)]

          #e - greedy algorithm to choose next_action for next_state
          rand_num = np.random.random()
          if epsilon > rand_num:
            next_action = np.random.choice(action_val)
          else:
            next_action = np.argmax(q_table[next_state]) #action in nex
t state s' with max_q value

          #q-value update function for SARSA
          q_table[current_state][action] = q_table[current_state][actio
n] + learning_rate*(reward + discount_factor*q_table[next_state][next_a
ction] - q_table[current_state][action])

          if episode == total_episodes:
```

```
            optimal.append(current_state+1)

          total_rewards += reward #Captured all the rewards in each epi
sode
          timestep += 1 #Number of timesteps in each episode

          current_state = next_state #next_state is assigned to current
_state
          action = next_action

          if done == True: #If terminal or target state reached then st
op the episode
            done = False
            break


    #Results after each episode
    avg_timesteps += timestep #Capturing all timesteps for all 100 epis
odes
    total_timesteps.append(avg_timesteps)

    reward_values.append(total_rewards) #Append rewards in every episod
e
    epsilon_values.append(epsilon) #Append epsilon values in every epis
ode


    if epsilon > 0.01: #keeping epsilon in [0.01 - 1] range as if it fa
lls below 0.01 it will exploit more: choosing best actions. We want our
 agent to explore a bit: choosing random actions
        epsilon = epsilon*decay_factor
    else:
        epsilon = 0.01


    if (episode % 100) == 0 and evaluation_results == False and sarsa_r
ewards == False: #printing results for every 100 episodes
      print("Episode: {}, Rewards: {}, Average timesteps taken: {}, eps
ilon: {}".format(episode, total_rewards, avg_timesteps//100, epsilon))
      avg_timesteps = 0

    #evaluation results
    if evaluation_results:
      if episode > total_episodes - eva_episodes:
         eva_rewards.append(reward)

    #printing the optimal path in last episode
    if episode == total_episodes:
          print("Optimal Path: ")
```

```
        for i in optimal:
          print(i,"->", end = " ")
        print(next_state+1)

#rewards
if sarsa_rewards:
    return reward_values

#Final Q - Table
print("Q Table: \n", q_table)

#Plotting the results
#x, y co-ordinates
x = [episode for episode in range(total_episodes)]
yr = reward_values
ye = epsilon_values

yr_eva = eva_rewards
x_eva = [episode for episode in range(eva_episodes)]


if evaluation_results:
    #episodes vs rewards
    plt.plot(x_eva,yr_eva)
    plt.title("Rewards per episode")
    plt.xlabel('Episodes')
    plt.ylabel('Rewards')

else:
    #episodes vs epsilon
    #Plots showing episodes vs epsilon, episodes vs rewards
    fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(10,4))
    ax1.plot(x, ye)
    ax1.set_title("Epsilon decay")

    #episodes vs rewards
    ax2.plot(x,yr)
    ax2.set_title("Rewards per episode")
```

**Results:**

**Deterministic environment:**

```
sarsa(deterministic = True)
```

**Epsilon decay over episodes:**

```
Episode: 100, Rewards: -3, Average timesteps taken: 18, epsilon: 0.7356422544596452
Episode: 200, Rewards: 97, Average timesteps taken: 13, epsilon: 0.5411695265464691
Episode: 300, Rewards: 95, Average timesteps taken: 11, epsilon: 0.39810717055350314
Episode: 400, Rewards: 107, Average timesteps taken: 9, epsilon: 0.29286445646252957
Episode: 500, Rewards: 107, Average timesteps taken: 8, epsilon: 0.21544346900319378
Episode: 600, Rewards: 103, Average timesteps taken: 7, epsilon: 0.15848931924611603
Episode: 700, Rewards: 106, Average timesteps taken: 6, epsilon: 0.11659144011798714
Episode: 800, Rewards: 107, Average timesteps taken: 6, epsilon: 0.08576958985909275
Episode: 900, Rewards: 107, Average timesteps taken: 6, epsilon: 0.06309573444802206
Episode: 1000, Rewards: 107, Average timesteps taken: 6, epsilon: 0.04641588833613003
Episode: 1100, Rewards: 107, Average timesteps taken: 6, epsilon: 0.03414548873833783
Episode: 1200, Rewards: 107, Average timesteps taken: 6, epsilon: 0.02511886431509727
Episode: 1300, Rewards: 107, Average timesteps taken: 6, epsilon: 0.01847849797422407
Episode: 1400, Rewards: 107, Average timesteps taken: 6, epsilon: 0.013593563908786174
Episode: 1500, Rewards: 107, Average timesteps taken: 6, epsilon: 0.010000000000000725
```

**Optimal Path and Q-table:**

Grid Environment

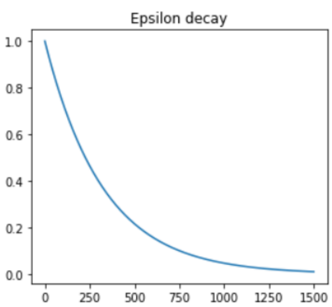
| | | | |
|---|---|---|---|
| Start | | | |
| | -1 | | |
| +3 | | -2 | |
| +4 | | | Target |

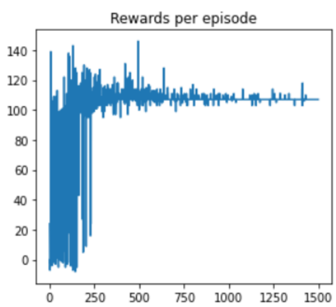
**(i.e., for reference)**

```
Optimal Path:
1 -> 5 -> 9 -> 13 -> 14 -> 15 -> 15 -> 16
Q Table:
 [[ 83.62606281  76.21456996  56.34834526  73.35950752]
 [ 67.43041883  16.4975812   12.69085235  33.10066237]
 [ 31.12392665   3.1486359    4.17796767  10.72544047]
 [ 21.53726193   2.34033437   5.04225988   1.0715774 ]
 [ 88.09751347  72.89377126  65.26958193  79.08675511]
 [ 48.44778668  18.56621409  38.75561957  80.62932866]
 [ 63.26052001   7.23324123  17.83419102   7.84249679]
 [ 62.52237997   8.03050208  24.22346806   8.38190111]
 [ 89.65692893  76.42668447  83.33641917  86.86469156]
 [ 89.0687089   32.89846484  57.32553798  55.50402685]
 [ 94.09289504  16.26533111  55.02137855  44.69032858]
 [ 99.92157894  20.6265568   59.10184044  27.23887387]
 [ 87.79045234  84.94843763  90.22253735  86.98096135]
 [ 88.68888202  82.89198301  94.24273469  86.19061613]
 [ 94.76012126  81.92777426 100.          87.17690738]
 [  0.           0.           0.           0.        ]]
```

Th agent learns the optimal policy and reaches the target following the optimal path it learned.

**Plots for epsilon decay and rewards per episode:**



**Note:** Agent collect maximum rewards after training for some time as it learns optimal path that maximizes the reward

### SARSA Stochastic environment

```
sarsa(stochastic = True)
```

### Epsilon decay over episodes:

```
Episode: 100, Rewards: 16, Average timesteps taken: 19, epsilon: 0.7356422544596452
Episode: 200, Rewards: 20, Average timesteps taken: 19, epsilon: 0.5411695265464691
Episode: 300, Rewards: 9, Average timesteps taken: 18, epsilon: 0.3981071705535314
Episode: 400, Rewards: 48, Average timesteps taken: 16, epsilon: 0.2928644564625957
Episode: 500, Rewards: 51, Average timesteps taken: 19, epsilon: 0.2154434690031937
Episode: 600, Rewards: 60, Average timesteps taken: 20, epsilon: 0.1584893192461160
Episode: 700, Rewards: 63, Average timesteps taken: 19, epsilon: 0.1165914401179871
Episode: 800, Rewards: 37, Average timesteps taken: 20, epsilon: 0.08576958985909275
Episode: 900, Rewards: 59, Average timesteps taken: 20, epsilon: 0.06309573444802206
Episode: 1000, Rewards: 70, Average timesteps taken: 20, epsilon: 0.04641588833613003
Episode: 1100, Rewards: 75, Average timesteps taken: 20, epsilon: 0.03414548873833783
Episode: 1200, Rewards: 70, Average timesteps taken: 20, epsilon: 0.02511886431509727
Episode: 1300, Rewards: 28, Average timesteps taken: 20, epsilon: 0.01847849797422407
Episode: 1400, Rewards: 61, Average timesteps taken: 20, epsilon: 0.013593563908786174
Episode: 1500, Rewards: 75, Average timesteps taken: 20, epsilon: 0.010000000000000725
```

### Optimal path and Q-table:

Grid Environment

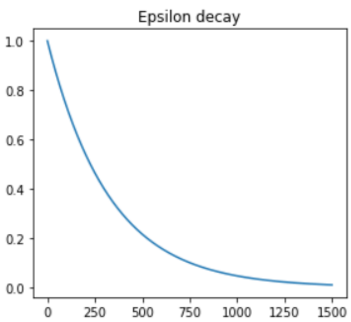
| | | | |
|---|---|---|---|
| Start | | | |
| | -1 | | |
| +3 | | -2 | |
| +4 | | | Target |

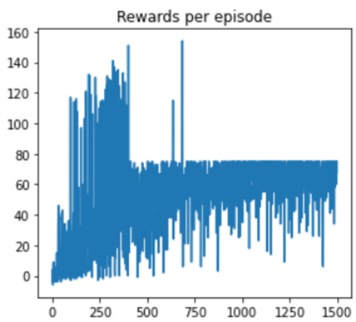
**(i.e., for reference)**

```
Optimal Path:
1 -> 5 -> 9 -> 13 -> 13 -> 13 -> 13 -> 13 -> 13 -> 13 -> 13 -> 13 -> 13 -> 13 -> 13 -> 13 -> 13 -> 13 -> 13 -> 13 -> 13
Q Table:
 [[ 6.97124147e+01  5.91617014e+01  5.54816576e+01  6.04903706e+01]
 [ 1.75762674e+01  2.08124558e+01  1.38739492e+01  5.93187198e+01]
 [ 2.64364447e+00  3.33831089e+00  3.30548162e-01  1.88830401e+01]
 [-2.20445071e-02  1.09939305e-01 -2.89712222e-02  1.76580657e+00]
 [ 7.41962276e+01  5.87823100e+01  5.73238757e+01  6.33394317e+01]
 [ 3.13245368e+01  1.99692900e+01  9.23542286e+00  6.22995558e+01]
 [ 5.43872479e+00  8.84751778e-01 -3.59020613e-02  2.12741832e+01]
 [-1.81080679e-01  5.61448261e-02 -1.72014395e-02  2.13886043e+00]
 [ 7.68547575e+01  6.35323073e+01  6.34693209e+01  7.16585476e+01]
 [ 2.42614809e+01  1.60330851e+01  2.07115390e+01  7.13812390e+01]
 [ 4.23820420e+01  1.92067612e+00  4.65117315e+00  8.19591424e+00]
 [ 2.17514263e+01 -5.76890696e-03  1.71826304e+00 -4.69674782e-01]
 [ 7.80390006e+01  7.44595350e+01  7.54402621e+01  7.83887766e+01]
 [ 5.29910215e+01  4.08247749e+01  6.77817040e+01  7.85116641e+01]
 [ 6.20500661e+01  1.45338422e+01  8.63331453e+01  3.85666903e+01]
 [ 0.00000000e+00  0.00000000e+00  0.00000000e+00  0.00000000e+00]]
```

Here optimal path is different because of stochastic nature of the environment

### Plots for epsilon decay and rewards per episode:



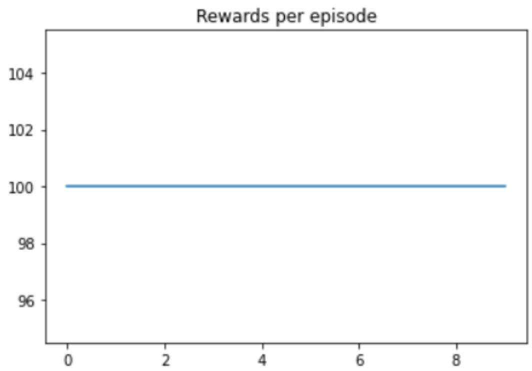



**Note:** Agent collect maximum rewards after training for some time as it learns optimal path that maximizes the reward. The rewards are uneven over the episodes because of stochastic nature.

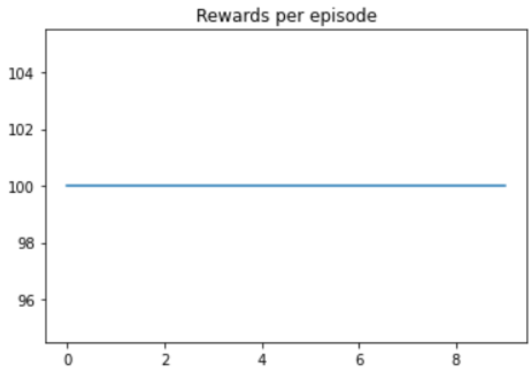
**Evaluation Results:**

**Q-Learning:**

```
q_learning(deterministic = True, evaluation_results = True)
```



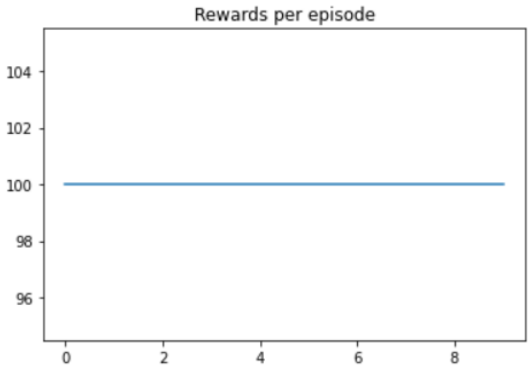

```
q_learning(stochastic = True, evaluation_results = True)
```




**Observation:** The rewards in each episode are constant because the agent has learnt an optimal path so every episode it achieves same maximum reward.

**SARSA:**

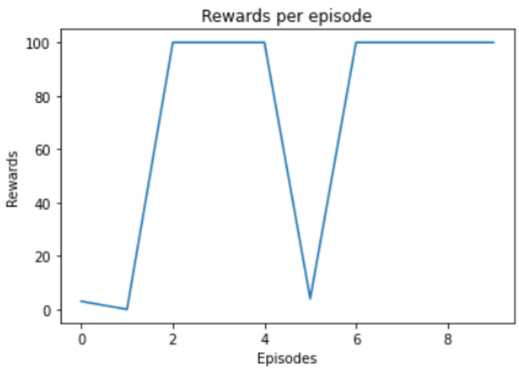
```
sarsa(deterministic = True, evaluation_results = True)
```




```
sarsa(stochastic = True, evaluation_results = True)
```

**The optimal path is different due to stochasticity:**

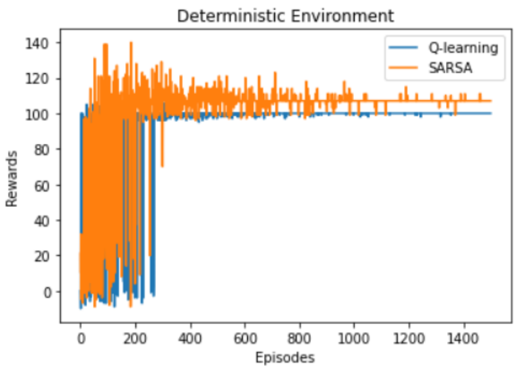
```
Evaluation Results
Optimal Path:
1 -> 1 -> 1 -> 5 -> 9 -> 13 -> 14 -> 15 -> 14 -> 13 -> 14 -> 15 -> 14 -> 15 -> 16
Q Table:
 [[ 6.63340592e+01  6.57857006e+01  5.89015494e+01  6.56912127e+01]
 [ 3.24754729e+01  2.08673705e+01  1.24518734e+01  6.38294743e+01]
 [ 3.58074590e+00  1.74016216e+00  1.53872125e-01  8.39382600e+00]
 [ 2.13598611e+01  2.57020507e+00  8.48108790e-01  2.80271322e-01]
 [ 7.43318808e+01  6.30844026e+01  6.06879670e+01  6.80160186e+01]
 [ 2.68991785e+01  1.90535201e+01  1.70860372e+01  6.74097215e+01]
 [ 5.47861129e+00 -2.01480445e-02  1.98614586e+01  7.64641061e+00]
 [ 4.43245997e+01  3.63972155e+00  4.37197943e+00  2.33706961e+00]
 [ 7.89400211e+01  6.82360713e+01  6.63998796e+01  7.37153561e+01]
 [ 5.78633299e+01  2.27481780e+01  3.71899934e+01  7.97806601e+01]
 [ 2.57567454e+01  1.20802310e+00  8.75478531e+01  1.76419436e+01]
 [ 9.79504955e+01  6.93482500e+00  2.93896081e+01  1.14283312e+01]
 [ 8.04240398e+01  7.63983011e+01  8.44863606e+01  8.07364334e+01]
 [ 7.26192698e+01  7.39149682e+01  8.95136864e+01  8.04880694e+01]
 [ 8.43461188e+01  6.48418746e+01  9.50591608e+01  8.04617094e+01]
 [ 0.00000000e+00  0.00000000e+00  0.00000000e+00  0.00000000e+00]]
```




**Observation:** In SARSA stochastic environment I intentionally ran the agent for few times to show sometimes even after learning optimal policy the agent may not collect maximum rewards in some episodes because of stochastic nature of the environment.

**2.Comparison of Q-Learning and SARSA: Deterministic Environment**

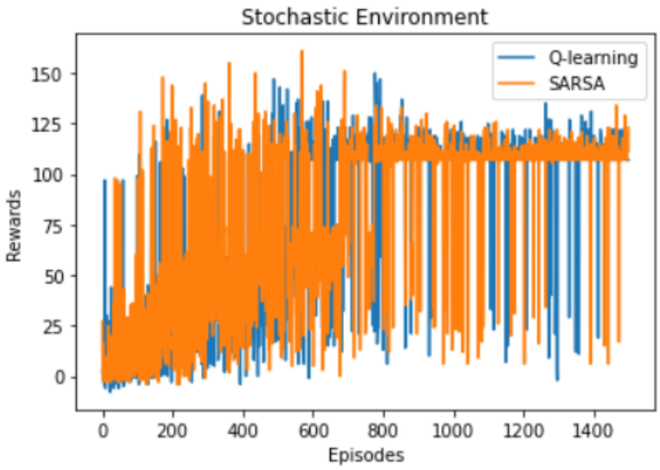
```
yq_r = q_learning(deterministic = True, qlearning_rewards = True)
ys_r = sarsa(deterministic = True, sarsa_rewards = True)
#Rewards for Q-learning and SARSA
yq = yq_r
ys = ys_r
episodes = len(yq_r)
x = [x for x in range(episodes)]
plt.plot(x, yq)
plt.plot(x, ys)
plt.xlabel('Episodes')
plt.ylabel('Rewards')
plt.legend(["Q-learning", "SARSA"])
plt.title("Deterministic Environment")
```




From the above plot it is evident that the reward collection in SARSA is varying more compared to Q-Learning because in SARSA the agent explores the grid collecting different rewards as we take next step based on greedy approach (i.e., e: random and (1-e) greedy). Whereas in Q-Learning we always choose greedy action reaching optimal path sooner. And both algorithms show increase in maximize reward over the episodes.

**3.Comparison of Q-Learning and SARSA: Stochastic Environment:**

```
yq_r = q_learning(stochastic = True, qlearning_rewards = True)
ys_r = sarsa(stochastic = True, sarsa_rewards = True)
#Rewards for Q-learning and SARSA
yq = yq_r
ys = ys_r
episodes = len(yq_r)
x = [x for x in range(episodes)]
plt.plot(x, yq)
plt.plot(x, ys)
plt.xlabel('Episodes')
plt.ylabel('Rewards')
plt.legend(["Q-learning", "SARSA"])
plt.title("Stochastic Environment")
```




Same as in deterministic environment the maximum reward collected is increasing over the episodes and the more variation is due to the stochastic nature of the environment.

**4.**

| | Update Function | Key Features |
|---|---|---|
| Q-Learning | Q(S,A) ← Q(S,A) + alpha*[R + y*maxaQ(S', a) – Q(S,A)] | It is a model free approach and off-policy TD control<br><br>Q-learning learns optimal policy faster<br><br>The next step is purely greedy choice |
| SARSA | Q(S,A) ← Q(S,A) + alpha*[R + y*Q(S', A') – Q(S,A)] | It is a model free approach and on-policy TD control<br><br>SARSA will take longer but safer route to the target<br><br>The next step is not entirely greedy (e – greedy) |

**Alpha= learning rate**

**Y = discount factor**

**References:**


https://www.javatpoint.com/agent-environment-in-ai#:~:text=Deterministic%20vs%20Stochastic%3A,determined%20completely%20by%20an%20agent.

https://arxiv.org/abs/2010.14603#:~:text=Safety%20is%20an%20essential%20component,constraints%20on%20the%20policy's%20behavior

https://www.learndatasci.com/tutorials/reinforcement-q-learning-scratch-python-openai-gym/