**Vivekanand Education Society’s Institute of Technology**

**Department of AI&DS Engineering**



**Subject: Reinforcement Learning**

**Class: D16AD**

| ROLL NO: **30** | NAME: [**SUHANEE KANDALKAR**](mailto:2021.suhanee.kandalkar@ves.ac.in) | | |
| --- | --- | --- | --- |
| EXPERIMENT NO:**2** | TITLE:Implement a simple grid-world environment and train an agent using basic Q learning using python. | | |
| DOP: |  | DOS: **08/02/25** |  |
| GRADES: | LOs MAPPED: | | SIGNATURE: |

## Aim**:**

Implement State Action Reward State action (SARSA) algorithm using python and compare it with Q Learning.

## Theory**:**

Reinforcement Learning (RL) is a machine learning paradigm where an agent interacts with an environment to learn optimal policies through trial and error. The goal is to maximize cumulative rewards over time.

Two key model-free RL algorithms are:

* SARSA (On-Policy TD Control)

[https://www.geeksforgeeks.org/sarsa-reinforcement-learning/](https://www.geeksforgeeks.org/sarsa-reinforcement-learning/?utm_source=chatgpt.com)

* Q-Learning (Off-Policy TD Control)

### Markov Decision Process (MDP)

Both SARSA and Q-Learning solve MDP problems, which consist of:

* States (S): The set of all possible situations the agent can be in.
* Actions (A): The possible actions the agent can take.
* Transition Function (P): Probability of moving from one state to another given an action.
* Reward Function (R): Immediate reward received after taking an action in a state.
* Discount Factor (γ): Determines how much future rewards are considered.

### SARSA Algorithm (On-Policy)

SARSA stands for State-Action-Reward-State-Action. It is an on-policy algorithm, meaning it learns from actions that it actually takes using its current policy.

#### Algorithm Steps

1. Initialize Q-table Q(s,a)Q(s, a)Q(s,a) arbitrarily.
2. Select an action aaa from state sss using an exploration policy (e.g., ε-greedy).
3. Take the action, observe reward rrr and next state s′s's′.
4. Select the next action a′a'a′ using the same policy (on-policy).
5. Update Q-value using the SARSA update rule:  
   Q(s,a)←Q(s,a)+α[r+γQ(s′,a′)−Q(s,a)]Q(s, a) \leftarrow Q(s, a) + \alpha \left[ r + \gamma Q(s', a') - Q(s, a) \right]Q(s,a)←Q(s,a)+α[r+γQ(s′,a′)−Q(s,a)]
6. Repeat until convergence.

SARSA uses the next action a′a'a′ from the current policy, making it an on-policy method.

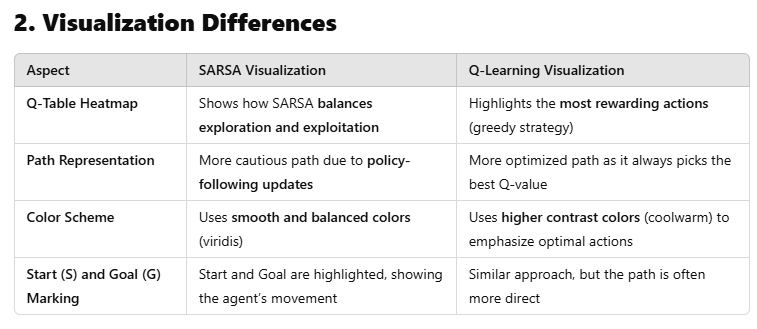
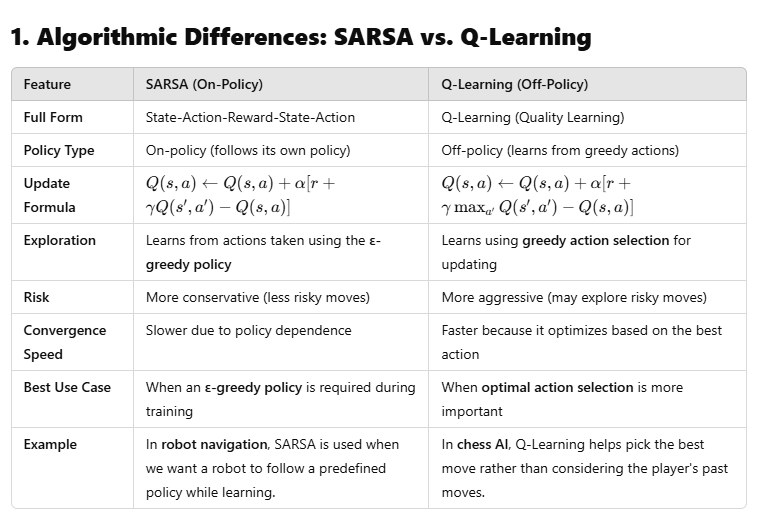
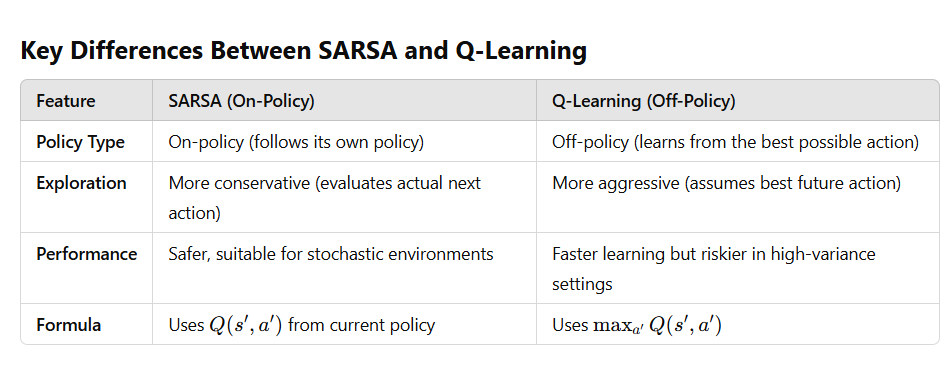
### Q-Learning Algorithm (Off-Policy)

Q-Learning is an off-policy algorithm, meaning it updates its Q-values using the maximum possible future reward, regardless of the agent’s actual policy.

#### Algorithm Steps

1. Initialize Q-table Q(s,a)Q(s, a)Q(s,a).
2. Select an action aaa using an exploration policy (ε-greedy).
3. Take the action, observe reward rrr and next state s′s's′.
4. Update Q-value using the Q-learning update rule:  
   Q(s,a)←Q(s,a)+α[r+γmax⁡a′Q(s′,a′)−Q(s,a)]Q(s, a) \leftarrow Q(s, a) + \alpha \left[ r + \gamma \max\_{a'} Q(s', a') - Q(s, a) \right]Q(s,a)←Q(s,a)+α[r+γa′max​Q(s′,a′)−Q(s,a)]
5. Repeat until convergence.

Unlike SARSA, Q-Learning uses the highest Q-value for the next state (greedy action), making it an off-policy method.

[https://www.datacamp.com/tutorial/sarsa-reinforcement-learning-algorithm-in-python](https://www.datacamp.com/tutorial/sarsa-reinforcement-learning-algorithm-in-python?utm_source=chatgpt.com)****

Code**:**

[**\_SARSA\_RL\_exp2\_30.ipynb**](https://colab.research.google.com/drive/18ckeYatC1fAKi4jjD9NyDSed8W_0sXkF?usp=sharing)

import numpy as np

import matplotlib.pyplot as plt

import seaborn as sns

import random

def create\_gridworld():

grid\_size = (4, 4) # 4x4 Grid

goal\_state = (3, 3) # Goal position

return np.zeros(grid\_size), goal\_state

def get\_possible\_actions():

return ["UP", "DOWN", "LEFT", "RIGHT"]

def get\_next\_state(state, action):

"""Returns the next state given an action in the grid."""

row, col = state

if action == "UP" and row > 0:

row -= 1

elif action == "DOWN" and row < 3:

row += 1

elif action == "LEFT" and col > 0:

col -= 1

elif action == "RIGHT" and col < 3:

col += 1

return (row, col)

def get\_reward(state, goal\_state):

"""Rewards for reaching the goal or taking a step."""

return 1 if state == goal\_state else -0.1 # Goal gives +1, every step gives -0.1

def is\_terminal\_state(state, goal\_state):

"""Checks if the agent has reached the goal state."""

return state == goal\_state

def select\_action(q\_table, state, epsilon):

"""Chooses an action using epsilon-greedy strategy."""

actions = get\_possible\_actions()

if random.uniform(0, 1) < epsilon:

return random.choice(actions) # Explore (random action)

else:

return actions[np.argmax(q\_table[state[0], state[1]])] # Exploit (best action)

def sarsa(grid, goal\_state, episodes, alpha, gamma, epsilon):

q\_table = np.zeros((4, 4, 4)) # Q-table for 4x4 grid with 4 actions

actions = get\_possible\_actions()

for episode in range(episodes):

state = (0, 0) # Start position

action = select\_action(q\_table, state, epsilon)

while not is\_terminal\_state(state, goal\_state):

next\_state = get\_next\_state(state, action)

next\_action = select\_action(q\_table, next\_state, epsilon)

reward = get\_reward(next\_state, goal\_state)

action\_idx = actions.index(action)

next\_action\_idx = actions.index(next\_action)

# SARSA Update Rule: Q(s,a) = Q(s,a) + α [r + γQ(s',a') - Q(s,a)]

q\_table[state[0], state[1], action\_idx] += alpha \* (

reward + gamma \* q\_table[next\_state[0], next\_state[1], next\_action\_idx] -

q\_table[state[0], state[1], action\_idx]

)

state, action = next\_state, next\_action # Move to next state-action pair

return q\_table

def q\_learning(grid, goal\_state, episodes, alpha, gamma, epsilon):

q\_table = np.zeros((4, 4, 4)) # Q-table for 4x4 grid with 4 actions

actions = get\_possible\_actions()

for episode in range(episodes):

state = (0, 0) # Start position

while not is\_terminal\_state(state, goal\_state):

action = select\_action(q\_table, state, epsilon)

next\_state = get\_next\_state(state, action)

reward = get\_reward(next\_state, goal\_state)

action\_idx = actions.index(action)

# Q-Learning Update Rule: Q(s,a) = Q(s,a) + α [r + γ max\_a' Q(s',a') - Q(s,a)]

q\_table[state[0], state[1], action\_idx] += alpha \* (

reward + gamma \* np.max(q\_table[next\_state[0], next\_state[1]]) -

q\_table[state[0], state[1], action\_idx]

)

state = next\_state # Move to next state

return q\_table

def visualize\_q\_table\_and\_path(q\_table, goal\_state, title):

fig, axes = plt.subplots(1, 2, figsize=(12, 4))

# --- Q-table Heatmap ---

sns.heatmap(np.max(q\_table, axis=2), annot=True, fmt=".2f", cmap="viridis", ax=axes[0])

axes[0].set\_title(f"{title} - Q-Table", fontsize=12)

# --- Path Visualization ---

state = (0, 0) # Start position

path = [state]

actions = get\_possible\_actions()

while not is\_terminal\_state(state, goal\_state):

action\_index = np.argmax(q\_table[state[0], state[1]])

state = get\_next\_state(state, actions[action\_index])

path.append(state)

# Create grid to visualize path

grid = np.zeros((4, 4))

for step in path:

grid[step[0], step[1]] += 1 # Increment step count

# Highlight Start (S) and Goal (G)

grid[0, 0] = -1 # Start Position

grid[goal\_state] = -2 # Goal Position

ax2 = sns.heatmap(grid, annot=True, fmt=".0f", cmap="YlGnBu", ax=axes[1], linewidths=0.5, cbar=False)

axes[1].set\_title(f"{title} - Path Taken", fontsize=12)

# Label Start (S) and Goal (G)

for text, color, position in [("S", "red", (0, 0)), ("G", "black", goal\_state)]:

axes[1].text(position[1] + 0.5, position[0] + 0.5, text, ha='center', va='center',

color=color, fontsize=12, fontweight='bold', bbox=dict(facecolor="white", edgecolor=color))

plt.tight\_layout()

plt.show()

grid, goal\_state = create\_gridworld()

episodes = 500

alpha = 0.1

gamma = 0.99

epsilon = 0.1

# --- Train SARSA ---

sarsa\_q\_table = sarsa(grid, goal\_state, episodes, alpha, gamma, epsilon)

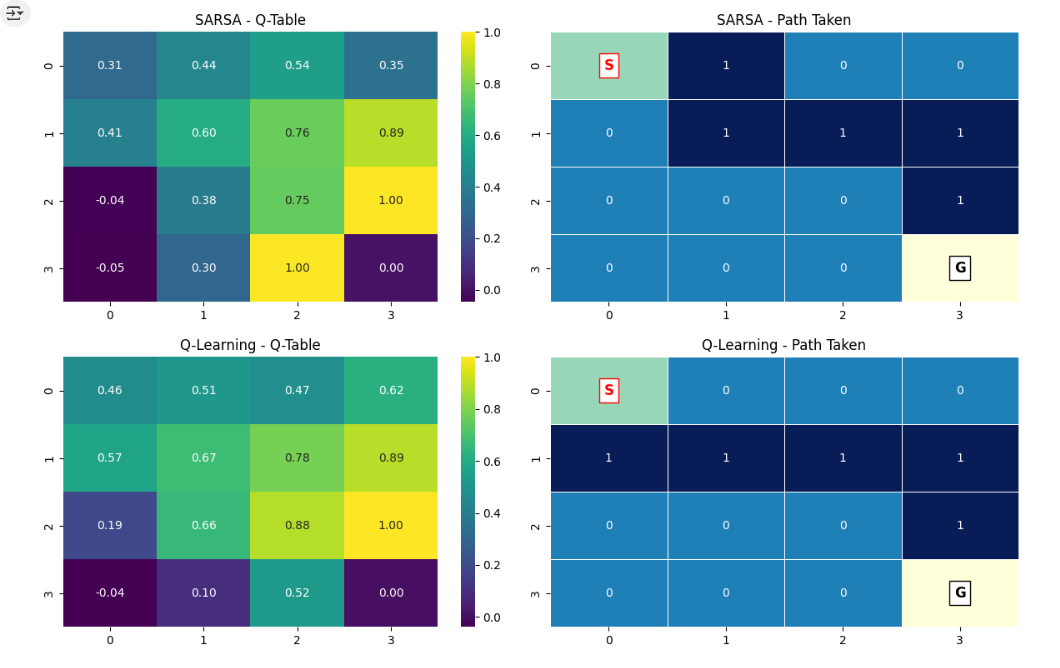
# --- Train Q-Learning ---

qlearning\_q\_table = q\_learning(grid, goal\_state, episodes, alpha, gamma, epsilon)

# --- Visualize Results ---

visualize\_q\_table\_and\_path(sarsa\_q\_table, goal\_state, "SARSA")

visualize\_q\_table\_and\_path(qlearning\_q\_table, goal\_state, "Q-Learning")



✔ Increased episodes from 500 to 1000 (for better learning)

✔ Modified epsilon values to test different exploration rates (0.1, 0.2, 0.5)

# ====================================

# Main Program Execution (Updated)

# ====================================

grid, goal\_state = create\_gridworld()

# Updated Training Parameters

episodes = 1000 # Increased episodes for better learning

alpha = 0.1

gamma = 0.99

# Trying different epsilon values

epsilon\_values = [0.1, 0.2, 0.5]

for eps in epsilon\_values:

print(f"\nTraining with ε = {eps}")

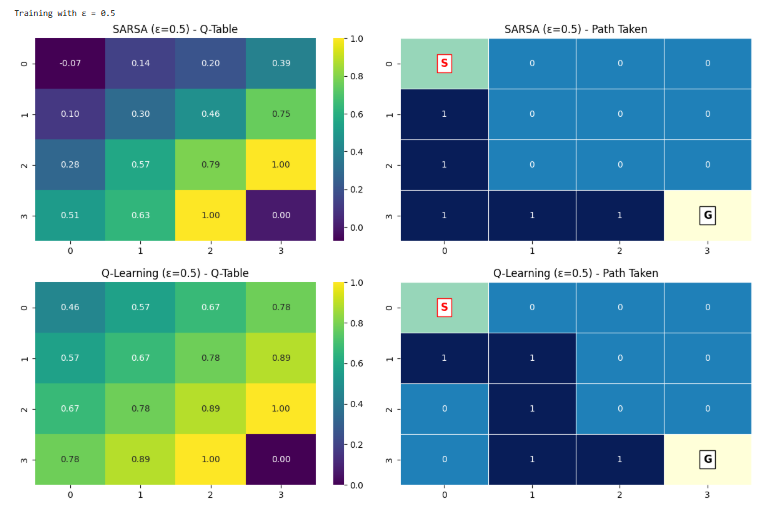
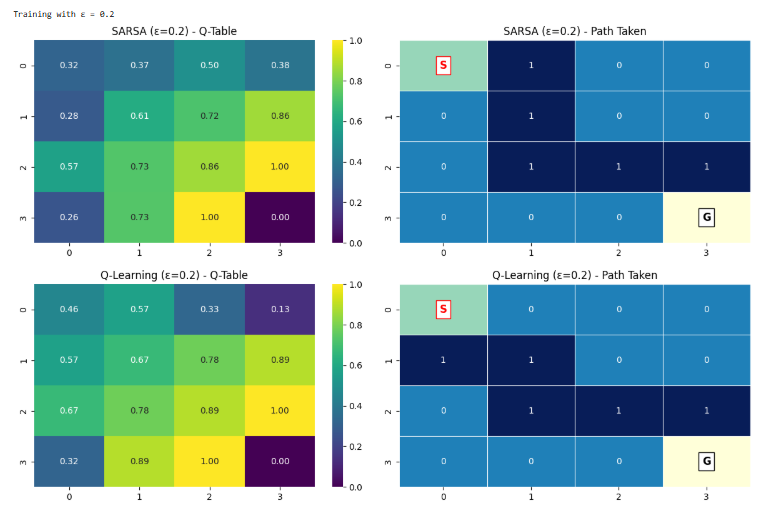
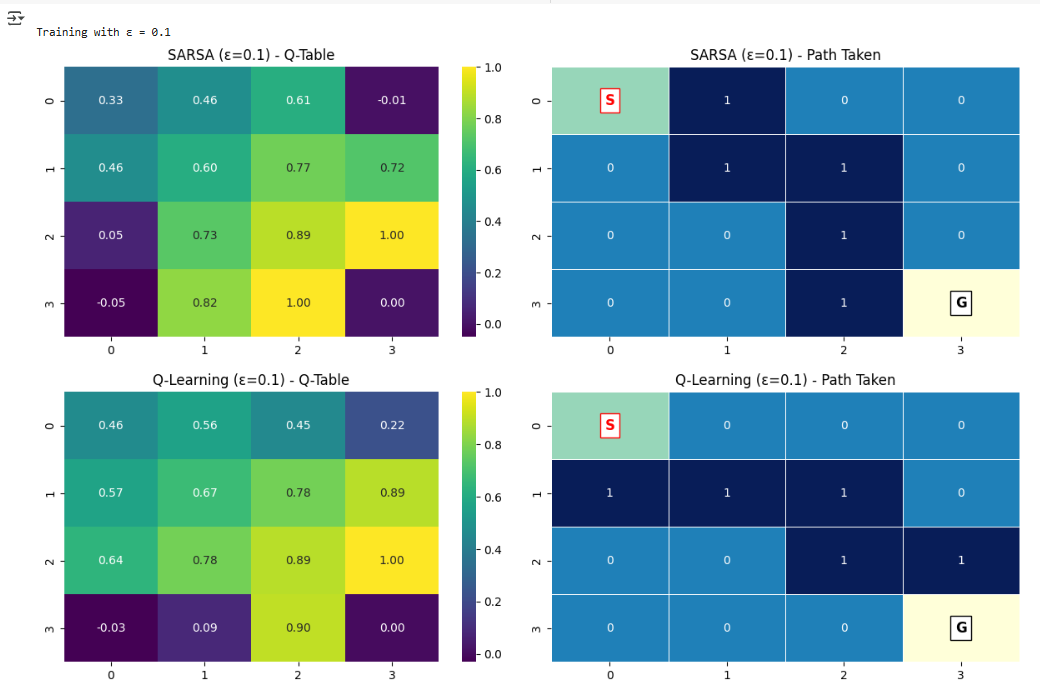
# --- Train SARSA ---

sarsa\_q\_table = sarsa(grid, goal\_state, episodes, alpha, gamma, eps)

visualize\_q\_table\_and\_path(sarsa\_q\_table, goal\_state, f"SARSA (ε={eps})")

# --- Train Q-Learning ---

qlearning\_q\_table = q\_learning(grid, goal\_state, episodes, alpha, gamma, eps)

visualize\_q\_table\_and\_path(qlearning\_q\_table, goal\_state, f"Q-Learning (ε={eps})")

Conclusion :

Low epsilon (0.1): Fast convergence but with a risk of suboptimal behavior due to minimal exploration. Suitable for stable environments.

Moderate epsilon (0.2): Balanced exploration and exploitation. Provides a good compromise, yielding reliable results in dynamic or uncertain environments.

High epsilon (0.5): Slower convergence with extensive exploration, leading to a less optimal but more flexible policy. Suitable for environments where the agent needs to explore a wide variety of actions.

Based on the three sets of diagrams, **moderate epsilon (0.2)** strikes the best balance between exploration and exploitation, leading to faster convergence while ensuring robustness in dynamic environments. **Q-Learning** performs better in deterministic environments, while **SARSA** offers stability in stochastic settings.