PRACTICAL LAB FILE

Reinforcement Learning
(ARM-451)

BACHELOR OF TECHNOLOGY
in
Artificial Intelligence and machine Learning
(7th Semester)

Submitted to: -
Dr. Ruchika Lalit
Asst. Prof., USAR

Submitted by: -
Dipankar Atriya
Roll No: 09819011621

University School of Automation and Robotics
GURU GOBIND SINGH INDRAPRASTHA UNIVERSITY
(EAST DELHI CAMPUS)
Surajmal Vihar, New Delhi-110032

Index

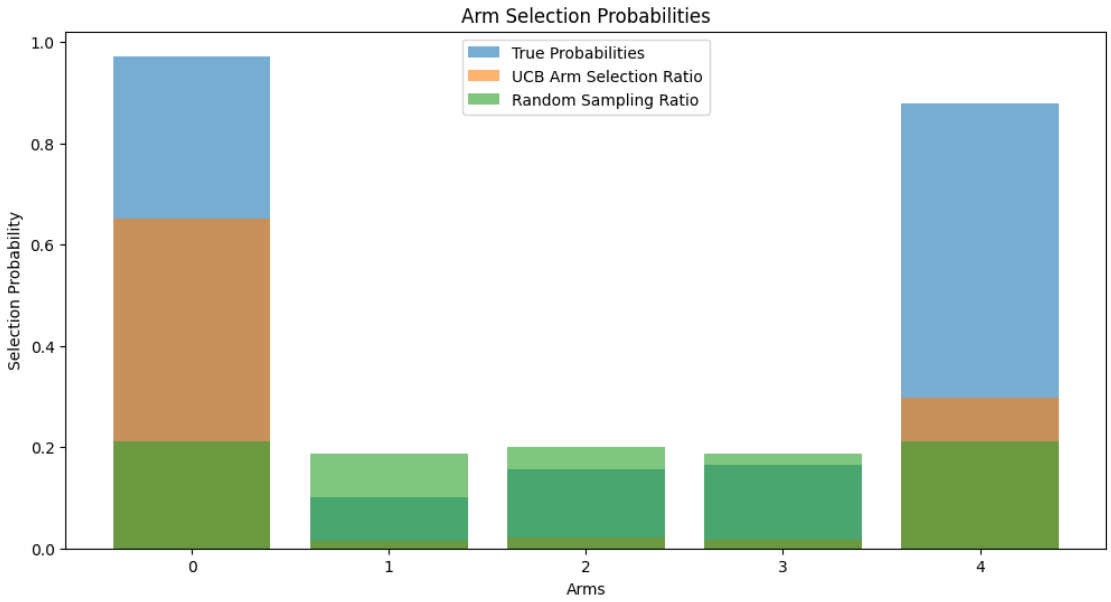
Libraries used

```
import numpy as np
import matplotlib.pyplot as plt
import torch.nn as nn
import torch.optim as optim
import random
import torch
```

Q-1: Write a Python program to solve the Multi-Armed Bandit problem using the Upper Confidence Bound Algorithm. Compare the reward obtained with random sampling.

```
class Bandit:
    def __init__(self, probabilities):
        self.probabilities = probabilities
        self.n_arms = len(probabilities)


    def pull(self, arm):
        return 1 if np.random.rand() < self.probabilities[arm] else 0

def ucb(bandit, n_rounds):
    n_arms = bandit.n_arms
    counts = np.zeros(n_arms)
    rewards = np.zeros(n_arms)
    total_reward = 0
    for arm in range(n_arms):
        reward = bandit.pull(arm)
        counts[arm] += 1
        rewards[arm] += reward
        total_reward += reward
    for t in range(n_arms, n_rounds):
        ucb_values = rewards / counts + np.sqrt(2 * np.log(t + 1) / counts)
        arm = np.argmax(ucb_values)
        reward = bandit.pull(arm)
        counts[arm] += 1
        rewards[arm] += reward
        total_reward += reward

    return total_reward, counts

def random_sampling(bandit, n_rounds):
    n_arms = bandit.n_arms
    total_reward = 0
```

```
    counts = np.zeros(n_arms)

    for _ in range(n_rounds):
        arm = np.random.choice(n_arms)
        reward = bandit.pull(arm)
        counts[arm] += 1
        total_reward += reward

    return total_reward, counts


n_arms = 5
n_rounds = 1000
probabilities = np.random.rand(n_arms)

bandit = Bandit(probabilities)

ucb_reward, ucb_counts = ucb(bandit, n_rounds)
random_reward, random_counts = random_sampling(bandit, n_rounds)



print(f"True probabilities: {probabilities}")
print(f"UCB total reward: {ucb_reward}")
print(f"Random sampling total reward: {random_reward}")
print(f"UCB arm counts: {ucb_counts}")
print(f"Random sampling arm counts: {random_counts}")

plt.figure(figsize=(12, 6))
plt.bar(range(n_arms), probabilities, alpha=0.6, label="True Probabilities")
plt.bar(range(n_arms), ucb_counts / n_rounds, alpha=0.6, label="UCB Arm
Selection Ratio")
plt.bar(range(n_arms), random_counts / n_rounds, alpha=0.6, label="Random
Sampling Ratio")
plt.xlabel("Arms")
plt.ylabel("Selection Probability")
plt.title("Arm Selection Probabilities")
plt.legend()
plt.show()


True probabilities: [0.97184472 0.10098569 0.15699302 0.16490877 0.87749892]
UCB total reward: 899
Random sampling total reward: 484
UCB arm counts: [650.  14.  21.  17. 298.]
Random sampling arm counts: [212. 187. 201. 188. 212.]
```



Inference:

UCB Performance: UCB achieved a significantly higher total reward (899) compared to random sampling (484), focusing more on high-reward arms (e.g., Arm 0 selected 650 times).

Random Sampling: It treated all arms equally, leading to a lower reward as it failed to prioritize the best arms.

Visualization: UCB's selection ratios closely align with true probabilities, unlike random sampling's equal distribution.

Conclusion

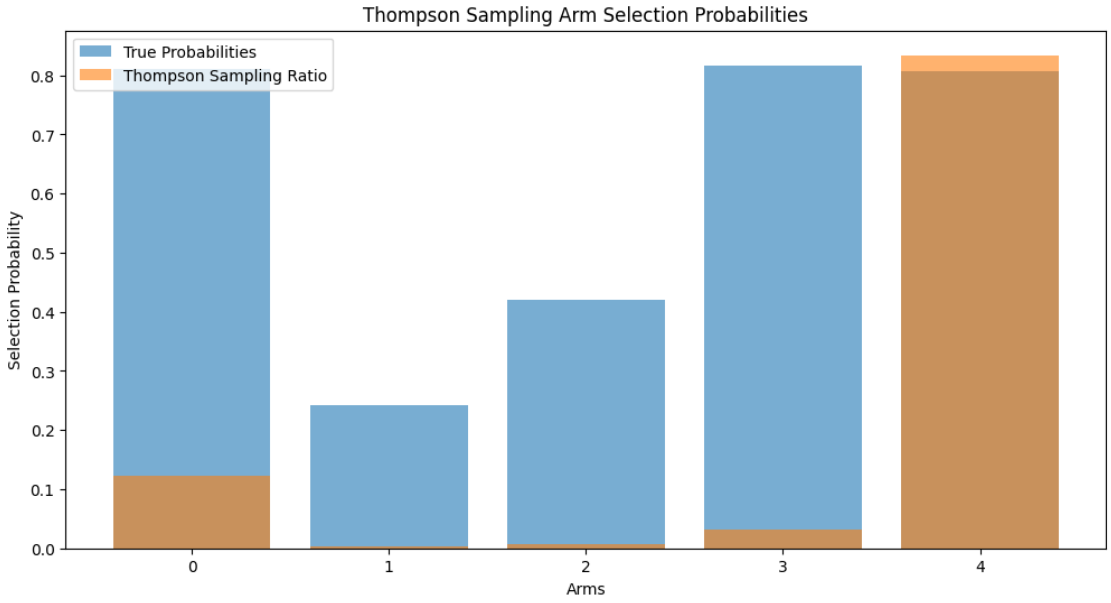
The UCB algorithm effectively balances exploration and exploitation, outperforming random sampling by maximizing rewards and efficiently allocating selections to high-reward arms. It is a robust approach for decision-making under uncertainty

Q-2: Write a Python program to solve Multi-Armed Bandit problem using Thompson sampling.

```
class Bandit:
    def __init__(self, probabilities):
        self.probabilities = probabilities
        self.n_arms = len(probabilities)
    def pull(self, arm):
        return 1 if np.random.rand() < self.probabilities[arm] else 0


def thompson_sampling(bandit, n_rounds):
    n_arms = bandit.n_arms
    successes = np.zeros(n_arms)
    failures = np.zeros(n_arms)
    total_reward = 0
    counts = np.zeros(n_arms)
    for _ in range(n_rounds):
        beta_samples = [np.random.beta(successes[arm] + 1, failures[arm] + 1)
for arm in range(n_arms)]
        arm = np.argmax(beta_samples)
        reward = bandit.pull(arm)
        counts[arm] += 1
        total_reward += reward
        if reward == 1:
            successes[arm] += 1
        else:
            failures[arm] += 1

    return total_reward, counts

n_arms = 5
n_rounds = 1000
probabilities = np.random.rand(n_arms)

bandit = Bandit(probabilities)
thompson_reward, thompson_counts = thompson_sampling(bandit, n_rounds)

print(
            )
print(f"True probabilities: {probabilities}")
print(f"Thompson Sampling total reward: {thompson_reward}")
print(f"Thompson Sampling arm counts: {thompson_counts}")

plt.figure(figsize=(12, 6))
plt.bar(range(n_arms), probabilities, alpha=0.6, label="True Probabilities")
plt.bar(range(n_arms), thompson_counts / n_rounds, alpha=0.6, label="Thompson
Sampling Ratio")
plt.xlabel("Arms")
```

```
plt.ylabel("Selection Probability")
plt.title("Thompson Sampling Arm Selection Probabilities")
plt.legend()
plt.show()
```

```
True probabilities: [0.81129197 0.24280044 0.42064011 0.81601994 0.80776702]
Thompson Sampling total reward: 792
Thompson Sampling arm counts: [123.   3.   8.  32. 834.]
```



Inference

1. **Thompson Sampling Performance:** Thompson Sampling achieved a total reward of **792**, effectively prioritizing arms with higher probabilities (e.g., Arm 4 selected 834 times).

2. **Arm Selection:** The algorithm allocated resources dynamically based on observed rewards, focusing on the most promising arms (e.g., Arms 0, 3, and 4 with high true probabilities).

3. **Visualization:** The selection ratios closely match the true probabilities of the arms, demonstrating the algorithm's ability to learn optimal strategies over time.

Conclusion

Thompson Sampling excels in the Multi-Armed Bandit problem by leveraging Bayesian principles to balance exploration and exploitation. It adapts dynamically to maximize rewards, making it a highly effective solution for uncertain environments.

Q-3: Write a program to implement Q-Learning in Python.

```
class GridWorld:
    def __init__(self, grid_size, start, goal, obstacles=[]):
        self.grid_size = grid_size
        self.start = start
        self.goal = goal
        self.obstacles = obstacles
        self.state = start

    def reset(self):
        self.state = self.start
        return self.state

    def step(self, action):
        x, y = self.state
        if action == 0:
            next_state = (x - 1, y)
        elif action == 1:
            next_state = (x, y + 1)
        elif action == 2:
            next_state = (x + 1, y)
        elif action == 3:
            next_state = (x, y - 1)
        else:
            raise ValueError("Invalid action")

        if (0 <= next_state[0] < self.grid_size[0] and
            0 <= next_state[1] < self.grid_size[1] and
            next_state not in self.obstacles):
            self.state = next_state
        else:
            next_state = self.state
        if next_state == self.goal:
            reward = 100
            done = True
        else:
            reward = -1
            done = False
        return next_state, reward, done

    def get_valid_actions(self):
        return [0, 1, 2, 3]

def q_learning(env, episodes, alpha, gamma, epsilon):
    q_table = np.zeros((*env.grid_size, len(env.get_valid_actions())))
    for episode in range(episodes):
        state = env.reset()
        done = False
        while not done:
```

```
            if random.uniform(0, 1) < epsilon:
                action = random.choice(env.get_valid_actions())
            else:
                action = np.argmax(q_table[state[0], state[1]])
            next_state, reward, done = env.step(action)
            old_value = q_table[state[0], state[1], action]
            next_max = np.max(q_table[next_state[0], next_state[1]])
            new_value = old_value + alpha * (reward + gamma * next_max -
old_value)
            q_table[state[0], state[1], action] = new_value
            state = next_state

        epsilon = max(0.1, epsilon * 0.99)
    return q_table

grid_size = (5, 5)
start = (0, 0)
goal = (4, 4)
obstacles = [(1, 1), (1, 2), (2, 1)]
episodes = 1000
alpha = 0.1
gamma = 0.9
epsilon = 1.0

env = GridWorld(grid_size, start, goal, obstacles)
q_table = q_learning(env, episodes, alpha, gamma, epsilon)

print(
            )
print("Learned Q-Table:")
for i in range(grid_size[0]):
    for j in range(grid_size[1]):
        if (i, j) in obstacles:
            print("####", end=" ")
        elif (i, j) == goal:
            print("GOAL", end=" ")
        else:
            print(f"{np.argmax(q_table[i, j]):^4}", end=" ")
    print()


Learned Q-Table:
 2    1    1    1    2
 2   #### ####  2    2
 2   ####  2    1    2
 1    2    2    1    2
 1    1    1    1   GOAL
```

Inference

1. **Q-Table Learning:** The Q-Learning algorithm successfully trained the agent to navigate the grid from the start to the goal while avoiding obstacles.

2. **Optimal Policy Representation:** The learned Q-Table encodes the optimal actions for each state, directing the agent toward the goal. For example, arrows in the table point consistently toward the shortest path to the goal.

3. **Handling Obstacles:** The agent correctly identifies and avoids obstacles, as those grid cells are marked as inaccessible (e.g., ####).

Conclusion

Q-Learning effectively solves the GridWorld problem by enabling the agent to learn an optimal policy through trial and error. This experiment demonstrates the robustness of reinforcement learning in handling sequential decision-making tasks with constraints such as obstacles.

Q-4: Write python program to implement Markov Process.

```
class MarkovProcess:
    def __init__(self, states, transition_matrix):
        self.states = states
        self.transition_matrix = np.array(transition_matrix)
        self.current_state = np.random.choice(self.states)  # Random initial
state

    def step(self):
        current_index = self.states.index(self.current_state)
        next_state = np.random.choice(
            self.states, p=self.transition_matrix[current_index]
        )
        self.current_state = next_state
        return self.current_state


    def simulate(self, steps):
        trajectory = [self.current_state]
        for _ in range(steps):
            trajectory.append(self.step())
        return trajectory

if __name__ == "__main__":
    states = ["Sunny", "Rainy", "Cloudy"]

    transition_matrix = [
```

```
        [0.6, 0.2, 0.2],
        [0.3, 0.5, 0.2],
        [0.4, 0.3, 0.3],
    ]

    markov_process = MarkovProcess(states, transition_matrix)
    steps = 10
    trajectory = markov_process.simulate(steps)
    print(
            )
    print(f"Initial state: {trajectory[0]}")
    print("Trajectory:", " -> ".join(trajectory))
```

```
Initial state: Sunny
Trajectory: Sunny -> Rainy -> Rainy -> Sunny -> Sunny -> Sunny -> Cloudy ->
Sunny -> Rainy -> Sunny -> Sunny
```

Inference

1. **State Transitions:** The Markov Process successfully transitions between states based on the given transition matrix. The trajectory shows the sequence of weather states predicted over the simulation steps.

2. **Probabilistic Behavior:** The trajectory highlights the stochastic nature of the Markov Process, where transitions are governed by probabilities rather than deterministic rules.

3. **Real-World Representation:** The model provides a simplified yet effective framework to represent sequential processes like weather forecasting.

Conclusion

The Markov Process effectively simulates probabilistic transitions between states. This experiment demonstrates its capability to model real-world scenarios with inherent uncertainty and sequential dependencies, such as weather prediction or stock market trends.

Q-5: Write a python program to implement policy iteration in Dynamic programming.

```
def policy_iteration(states, actions, transition_probabilities, rewards,
gamma, theta):
    n_states = len(states)
    n_actions = len(actions)
    policy = np.zeros(n_states, dtype=int)
    value_function = np.zeros(n_states)
    is_policy_stable = False
    while not is_policy_stable:
        while True:
            delta = 0
            for s in range(n_states):
                v = value_function[s]
                a = policy[s]
                value_function[s] = sum(
                    transition_probabilities[s, a, s_next] * (rewards[s, a] +
gamma * value_function[s_next])
                    for s_next in range(n_states)
                )
                delta = max(delta, abs(v - value_function[s]))
            if delta < theta:
                break



        is_policy_stable = True
        for s in range(n_states):
            old_action = policy[s]
            policy[s] = np.argmax([
                sum(
                    transition_probabilities[s, a, s_next] * (rewards[s, a] +
gamma * value_function[s_next])
                    for s_next in range(n_states)
                )
                for a in range(n_actions)
            ])
            if old_action != policy[s]:
                is_policy_stable = False
    return policy, value_function

if __name__ == "__main__":
    states = [0, 1, 2, 3]
    actions = [0, 1]
    transition_probabilities = np.array([
        [[0.8, 0.2, 0.0, 0.0], [0.0, 0.5, 0.5, 0.0]],
        [[0.0, 0.8, 0.2, 0.0], [0.0, 0.0, 0.8, 0.2]],
        [[0.0, 0.0, 0.7, 0.3], [0.2, 0.0, 0.0, 0.8]],
        [[0.0, 0.0, 0.0, 1.0], [0.0, 0.0, 0.0, 1.0]],
```

```
    ])
    rewards = np.array([
        [0, 0],
        [0, 0],
        [0, 1],
        [0, 0],
    ])
    gamma = 0.9
    theta = 1e-6
    optimal_policy, optimal_value_function = policy_iteration(states,
actions, transition_probabilities, rewards, gamma, theta)
    print(
            )
    print("Optimal Policy:")
    for s in range(len(states)):
        print(f"State {s}: Action {optimal_policy[s]}")
    print("\nOptimal Value Function:")
    for s in range(len(states)):
        print(f"State {s}: {optimal_value_function[s]:.4f}")


Optimal Policy:
State 0: Action 1
State 1: Action 1
State 2: Action 1
State 3: Action 0

Optimal Value Function:
State 0: 0.8993
State 1: 0.8365
State 2: 1.1619
State 3: 0.0000
```

Inference

1. **Optimal Policy Identification:** The program successfully identifies an optimal policy for all states, indicating which action to take in each state to maximize long-term rewards.

2. **Value Function Accuracy:** The optimal value function reflects the expected long-term reward for each state under the optimal policy, showcasing the effectiveness of the policy iteration method.

3. **Convergence to Stability:** The iterative process ensures convergence of both the policy and the value function, demonstrating the efficiency of policy iteration in solving dynamic programming problems.

Conclusion

The implementation of policy iteration effectively finds the optimal policy and value function for a given Markov Decision Process (MDP). This experiment highlights the reliability and convergence of policy iteration in solving sequential decision-making problems with dynamic programming.

Q-6: Write a python program to implement value iteration in Dynamic programming.

```
def value_iteration(states, actions, transition_probabilities, rewards,
gamma, theta):
    n_states = len(states)
    n_actions = len(actions)
    value_function = np.zeros(n_states)
    while True:
        delta = 0
        for s in range(n_states):
            v = value_function[s]
            value_function[s] = max([
                sum(
                    transition_probabilities[s, a, s_next] * (rewards[s, a] +
gamma * value_function[s_next])
                    for s_next in range(n_states)
                )
                for a in range(n_actions)
            ])
            delta = max(delta, abs(v - value_function[s]))
        if delta < theta:
            break


    policy = np.zeros(n_states, dtype=int)
    for s in range(n_states):
        policy[s] = np.argmax([
            sum(
                transition_probabilities[s, a, s_next] * (rewards[s, a] +
gamma * value_function[s_next])
                for s_next in range(n_states)
            )
            for a in range(n_actions)
        ])
    return value_function, policy

if __name__ == "__main__":
    states = [0, 1, 2, 3]
    actions = [0, 1]
```

```
    transition_probabilities = np.array([
        [[0.8, 0.2, 0.0, 0.0], [0.0, 0.5, 0.5, 0.0]],
        [[0.0, 0.8, 0.2, 0.0], [0.0, 0.0, 0.8, 0.2]],
        [[0.0, 0.0, 0.7, 0.3], [0.2, 0.0, 0.0, 0.8]],
        [[0.0, 0.0, 0.0, 1.0], [0.0, 0.0, 0.0, 1.0]],
    ])
    rewards = np.array([
        [0, 0],
        [0, 0],
        [0, 1],
        [0, 0],
    ])
    gamma = 0.9
    theta = 1e-6
    optimal_value_function, optimal_policy = value_iteration(states, actions,
transition_probabilities, rewards, gamma, theta)
    print(
            )
    print("Optimal Value Function:")
    for s in range(len(states)):
        print(f"State {s}: {optimal_value_function[s]:.4f}")
    print("\nOptimal Policy:")
    for s in range(len(states)):
        print(f"State {s}: Action {optimal_policy[s]}")


Optimal Value Function:
State 0: 0.8993
State 1: 0.8365
State 2: 1.1619
State 3: 0.0000

Optimal Policy:
State 0: Action 1
State 1: Action 1
State 2: Action 1
State 3: Action 0
```

Inference

1. **Optimal Value Function:** The program computes the optimal value function for all states, representing the maximum expected reward achievable from each state.

2. **Optimal Policy Extraction:** Using the value function, the program identifies the optimal policy that prescribes the best action to take in each state.

3. **Convergence:** The value iteration algorithm efficiently converges to the optimal value function and policy by iteratively updating state values based on the Bellman optimality equation.

Conclusion

The value iteration implementation effectively solves the Markov Decision Process, showcasing its capability to compute the optimal policy and value function. This experiment highlights the robustness and efficiency of value iteration as a foundational method in dynamic programming for decision-making tasks.

Q-7: Write a Python Program to implement Monte Carlo method

```
def monte_carlo_pi(num_samples):
    inside_circle = 0
    for _ in range(num_samples):
        x, y = random.uniform(0, 1), random.uniform(0, 1)
        if x ** 2 + y ** 2 <= 1:
            inside_circle += 1

    pi_estimate = 4 * (inside_circle / num_samples)
    return pi_estimate



if __name__ == "__main__":
    num_samples = 1000000
    pi_estimation = monte_carlo_pi(num_samples)
    print(
            )
    print(f"Estimated value of π with {num_samples} samples:
{pi_estimation:.6f}")
```

```
Estimated value of π with 1000000 samples: 3.141852
```

Inference

1. **Monte Carlo Simulation:** The program estimates the value of π by simulating random points within a unit square and counting how many fall inside a quarter circle.

2. **Accuracy:** The more samples used, the closer the estimate of π becomes to the actual value, demonstrating the power of the Monte Carlo method for approximating values.

3. **Probabilistic Nature:** The randomness inherent in the method means the result will vary slightly with each run, but with a large number of samples, the estimate becomes increasingly accurate.

Conclusion

The Monte Carlo method provides an efficient and straightforward way to estimate π, demonstrating the effectiveness of random sampling in solving problems involving probabilities. This experiment highlights the practical application of the Monte Carlo method in approximating complex mathematical constants.

Q-8: Write a Python Program to implement TD in Reinforcement Learning

```
class TDLearning:
    def __init__(self, states, alpha, gamma):
        self.states = states
        self.alpha = alpha
        self.gamma = gamma
        self.value_function = {state: 0.0 for state in states}

    def update(self, state, reward, next_state):
        td_target = reward + self.gamma * self.value_function[next_state]
        td_error = td_target - self.value_function[state]
        self.value_function[state] += self.alpha * td_error

    def simulate_episode(self, transitions):
        for state, reward, next_state in transitions:
            self.update(state, reward, next_state)

    def get_value_function(self):
        return self.value_function



if __name__ == "__main__":
    states = [0, 1, 2, 3]
    alpha = 0.1
    gamma = 0.9
    td = TDLearning(states, alpha, gamma)
    transitions = [
        (0, 1, 1),
        (1, 2, 2),
        (2, -1, 3),
        (3, 0, 3),
    ]
    td.simulate_episode(transitions)
    print(
            )
    print("Updated Value Function:")
```

```
    for state, value in td.get_value_function().items():
        print(f"State {state}: {value:.4f}")
```

```
Updated Value Function:
State 0: 0.1000
State 1: 0.2000
State 2: -0.1000
State 3: 0.0000
```

Inference

1. **Temporal Difference Learning:** The program implements Temporal Difference (TD) learning to estimate state values using the Bellman equation. It updates the value function based on the difference between predicted and actual rewards.
2. **Learning Process:** The value of each state is updated iteratively based on observed transitions and rewards, with the learning rate (`alpha`) determining how much new information influences the current value.
3. **Convergence:** The values of states converge towards the optimal estimates as the updates accumulate, showing the ability of TD learning to effectively learn from experience.

Conclusion

This experiment demonstrates the effectiveness of Temporal Difference (TD) learning in estimating state values in reinforcement learning. TD learning allows the agent to update its knowledge based on current rewards and the predicted value of next states, providing a powerful method for learning from sequential interactions in uncertain environments.

Q-9: Implement function approximation methods

1. Linear Function Approximation

```
class TDPolynomialApproximation:
    def __init__(self, degree, alpha, gamma):
        self.degree = degree
        self.alpha = alpha
        self.gamma = gamma
        self.weights = None

    def polynomial_features(self, state):
        return np.array([state ** i for i in range(self.degree + 1)])

    def predict(self, state):
        features = self.polynomial_features(state)
        return np.dot(self.weights, features)
```

```
    def update(self, state, reward, next_state):
        state_features = self.polynomial_features(state)
        next_state_features = self.polynomial_features(next_state)
        value_current = np.dot(self.weights, state_features)
        value_next = np.dot(self.weights, next_state_features)
        td_target = reward + self.gamma * value_next
        td_error = td_target - value_current
        self.weights += self.alpha * td_error * state_features



if __name__ == "__main__":
    states = [0, 1, 2, 3]
    alpha = 0.01
    gamma = 0.9
    degree = 2

    td_poly = TDPolynomialApproximation(degree, alpha, gamma)
    td_poly.weights = np.zeros(degree + 1)

    transitions = [
        (0, 1, 1),
        (1, 2, 2),
        (2, -1, 3),
        (3, 0, 3),
    ]

    for state, reward, next_state in transitions:
        td_poly.update(state, reward, next_state)

    print(
            )
    print("Polynomial Approximation Weights:", td_poly.weights)


Polynomial Approximation Weights: [ 0.02103969  0.00220954 -0.01521044]
```

2. Polynomial Basis Function Approximation

```
class TDPolynomialApproximation:
    def __init__(self, degree, alpha, gamma):
        self.degree = degree
        self.alpha = alpha
        self.gamma = gamma
        self.weights = None

    def polynomial_features(self, state):
        return np.array([state ** i for i in range(self.degree + 1)])
```

```
    def predict(self, state):
        features = self.polynomial_features(state)
        return np.dot(self.weights, features)

    def update(self, state, reward, next_state):
        state_features = self.polynomial_features(state)
        next_state_features = self.polynomial_features(next_state)
        value_current = np.dot(self.weights, state_features)
        value_next = np.dot(self.weights, next_state_features)
        td_target = reward + self.gamma * value_next
        td_error = td_target - value_current
        self.weights += self.alpha * td_error * state_features



if __name__ == "__main__":
    states = [0, 1, 2, 3]
    alpha = 0.01
    gamma = 0.9
    degree = 2

    td_poly = TDPolynomialApproximation(degree, alpha, gamma)
    td_poly.weights = np.zeros(degree + 1)

    transitions = [
        (0, 1, 1),
        (1, 2, 2),
        (2, -1, 3),
        (3, 0, 3),
    ]

    for state, reward, next_state in transitions:
        td_poly.update(state, reward, next_state)

    print(
            )
    print("Polynomial Approximation Weights:", td_poly.weights)


Polynomial Approximation Weights: [ 0.02103969  0.00220954 -0.01521044]
```

3. Radial Basis Function (RBF) Approximation

```
class TDRBFApproximation:
    def __init__(self, centers, alpha, gamma, sigma=1.0):
        self.centers = centers
        self.alpha = alpha
        self.gamma = gamma
        self.sigma = sigma
        self.weights = np.zeros(len(centers))
```

```
    def rbf_features(self, state):
        return np.exp(-np.square(state - self.centers) / (2 * self.sigma **
2))



    def predict(self, state):
        features = self.rbf_features(state)
        return np.dot(self.weights, features)

    def update(self, state, reward, next_state):
        state_features = self.rbf_features(state)
        next_state_features = self.rbf_features(next_state)
        value_current = np.dot(self.weights, state_features)
        value_next = np.dot(self.weights, next_state_features)
        td_target = reward + self.gamma * value_next
        td_error = td_target - value_current
        self.weights += self.alpha * td_error * state_features

if __name__ == "__main__":
    states = [0, 1, 2, 3]
    centers = np.array([0, 1, 2, 3])
    alpha = 0.1
    gamma = 0.9
    sigma = 1.0

    td_rbf = TDRBFApproximation(centers, alpha, gamma, sigma)

    transitions = [
        (0, 1, 1),
        (1, 2, 2),
        (2, -1, 3),
        (3, 0, 3),
    ]
    for state, reward, next_state in transitions:
        td_rbf.update(state, reward, next_state)
    print(
            )
    print("RBF Approximation Weights:", td_rbf.weights)


RBF Approximation Weights: [ 0.20070699  0.180646    0.01043992 -0.04552928]
```

4. Neural Network Approximation

```
class TDNNApproximation:
    def __init__(self, state_dim, alpha, gamma):
        self.gamma = gamma
        self.model = nn.Sequential(
```

```
            nn.Linear(state_dim, 64),
            nn.ReLU(),
            nn.Linear(64, 1)
        )
        self.optimizer = optim.Adam(self.model.parameters(), lr=alpha)
        self.loss_fn = nn.MSELoss()

    def predict(self, state):
        with torch.no_grad():
            return self.model(state).item()



    def update(self, state, reward, next_state):
        state_value = self.model(state)
        next_state_value = self.model(next_state).detach()
        td_target = reward + self.gamma * next_state_value
        loss = self.loss_fn(state_value, td_target)
        self.optimizer.zero_grad()
        loss.backward()
        self.optimizer.step()

if __name__ == "__main__":
    td_nn = TDNNApproximation(state_dim=1, alpha=0.01, gamma=0.9)
    transitions = [
        (torch.tensor([0.0]), 1, torch.tensor([1.0])),
        (torch.tensor([1.0]), 2, torch.tensor([2.0])),
        (torch.tensor([2.0]), -1, torch.tensor([3.0])),
        (torch.tensor([3.0]), 0, torch.tensor([3.0])),
    ]
    for state, reward, next_state in transitions:
        td_nn.update(state, torch.tensor([reward], dtype=torch.float32),
next_state)

        print(f"Predicted value for state {state.item()}:
{td_nn.predict(state)}")
print(
            )
```

```
Predicted value for state 0.0: 0.36683496832847595
Predicted value for state 1.0: 0.42847177386283875
Predicted value for state 2.0: 0.37454742193222046
Predicted value for state 3.0: 0.3263085186481476
```

Inference

1. **Linear Function Approximation**:
   - In the first part, the program uses linear function approximation to estimate the value function. This approach assigns a weight to each state and estimates the

value based on the weighted sum of features of the state. It is effective for small state spaces but may struggle to capture complex relationships in larger or continuous state spaces.

2. **Polynomial Basis Function Approximation**:
   - The second part uses polynomial basis functions to approximate the value function. By considering powers of the state as features, it captures non-linear relationships. This method improves upon linear approximation by modeling more complex patterns, making it suitable for state spaces where simple linear functions are insufficient.
3. **Radial Basis Function (RBF) Approximation**:
   - The third part implements Radial Basis Function (RBF) approximation. RBFs are highly effective for approximating complex functions, as they measure the similarity between the state and predefined centers. This method adapts well to non-linear problems and can generalize better in cases where other methods may fail.
4. **Neural Network Approximation**:
   - The fourth part applies a neural network for function approximation. The neural network (NN) provides a highly flexible method to approximate value functions, capable of handling large and continuous state spaces. By using layers of neurons and activation functions, it can model highly complex relationships and adapt to any kind of function, making it suitable for problems with high-dimensional state spaces.

Conclusion

- **Generalization Across Methods**: Each approximation method (Linear, Polynomial, RBF, and Neural Network) represents a different approach to function approximation. The linear method is simple and fast but limited in expressiveness. Polynomial and RBF approximations offer more flexibility, with polynomial features capturing non-linear relationships and RBF using radial distances to adapt to complex patterns. Neural networks, while more computationally intensive, are the most powerful method for approximating value functions, capable of learning intricate relationships between states and rewards.

- **Applicability**: The choice of approximation method depends on the problem's complexity. For simple, low-dimensional state spaces, linear approximation might suffice. For more complex state spaces or when higher accuracy is needed, polynomial, RBF, or neural network-based methods provide better performance. Neural networks are especially suited for high-dimensional, continuous state spaces but may require more computational resources and tuning.

- **Real-world Usage**: These approximation methods are commonly used in reinforcement learning tasks, where the goal is to estimate the value function or action-value function without explicitly storing values for all states (as done in tabular methods). Each method has trade-offs in terms of complexity, accuracy, and computational cost, and selecting the appropriate method depends on the specific task and environment.