|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | **Experiment No. – 5** | |  |  |
| **Date of Performance:** | **17/03/2025** | |  |  |
| **Date of Submission:** | **17/03/2025** | |  |  |
| Program Execution/ formation/  correction/  ethical practices  (06) | Timely  Submission  (01) | Viva (03) | Experiment Total (10) | Sign with Date |
|  |  |  |  |  |

**Experiment No. 5**

**1.1Aim:**  Implementing Monte Carlo control and Temporal Difference (TD) learning algorithms to train an agent in grid world environment

**Course Outcome:**

* Analyze and solve problems using Reinforcement Learning techniques.
* Understand and implement policy iteration and value iteration algorithms for MDPs.

**1.3 Learning Objectives:**

* Understand the structure and representation of a grid-world environment.
* Represent the grid-world environment as a Markov Decision Process.
* Implement and compare policy iteration and value iteration algorithms to compute optimal policies.

**1.4 Requirement:**

* Python programming language
* Libraries: NumPy, matplotlib
* IDE or code editor

* 1. **Related Theory:**

**1. Monte Carlo Control:**

* **Concept:** Monte Carlo methods learn by sampling complete episodes, updating value estimates based on the final return (total reward) of each episode.
* **Algorithm (First-Visit Monte Carlo):**
  1. **Initialization:** Initialize a policy (e.g., epsilon-greedy) and a value function (e.g., Q-function).
  2. **Episode Generation:** Generate an episode by following the current policy.
  3. **Update Value Function:** For each state-action pair visited in the episode, update the Q-value based on the episode's return.
  4. **Policy Improvement:** Update the policy based on the updated Q-values (e.g., using epsilon-greedy).
  5. **Repeat:** Repeat steps 2-4 until the policy converges.
* **Advantages:**

Simple to implement, doesn't require knowledge of the environment's dynamics.

* **Disadvantages:**

Can be slow to converge, updates only occur at the end of episodes, and can be inefficient in environments with long episodes.

**2. Temporal Difference (TD) Learning:**

* **Concept:**

TD methods learn by updating value estimates incrementally, based on the reward received and the estimated value of the next state.

* **Algorithm (TD (0) or Q-learning):**
  1. **Initialization:** Initialize a policy (e.g., epsilon-greedy) and a value function (e.g., Q-function).
  2. **Interaction:** Interact with the environment, taking actions and observing rewards and next states.
  3. **Update Value Function:** Update the Q-value of the current state-action pair based on the reward and the estimated value of the next state.
  4. **Policy Improvement:** Update the policy based on the updated Q-values (e.g., using epsilon-greedy).
  5. **Repeat:** Repeat steps 2-4 until the policy converges.
* **Advantages:**

Can converge faster than Monte Carlo methods, updates occur incrementally, and can be more efficient in environments with long episodes.

* **Disadvantages:**

Can be more complex to implement, and requires knowledge of the environment's dynamics.

**3. Grid World Implementation:**

* **Environment:** Define a grid world environment with states, actions, rewards, and transitions.
* **Agent:** Implement an agent that can interact with the environment, take actions, and receive rewards.
* **Algorithm:** Choose either Monte Carlo control or TD learning (e.g., Q-learning) to train the agent.
* **Evaluation:** Evaluate the agent's performance by measuring its success rate in reaching the goal state.

**4. Key Concepts:**

* **Policy:** A strategy that determines the agent's actions in different states.
* **Value Function:** A function that estimates the expected cumulative reward for being in a particular state or taking a particular action.
* **Return:** The total reward received from a particular state or action onwards.
* **Bootstrapping:** Using current estimates to update future estimates.
* **Exploration vs. Exploitation:** Balancing the need to explore different actions to find the best policy with the need to exploit the current knowledge to achieve the goal.
* **Epsilon-Greedy:** A policy that explores actions randomly with a small probability (epsilon) and exploits the current knowledge with the remaining probability.

* 1. **Coding & Results**

import numpy as np

import matplotlib.pyplot as plt

class GridWorld:

    def \_\_init\_\_(self, size=4, terminal\_states=None, pits=None):

        self.size = size

        self.terminal\_states = terminal\_states if terminal\_states else {(3, 3)}  # (row, col)

        self.pits = pits if pits else {(1, 1), (1, 2)}  # Example pits

        self.actions = ['up', 'down', 'left', 'right']

        self.state = (0, 0)  # Start at (0, 0)

    def reset(self):

        self.state = (0, 0)

        return self.state

    def step(self, action):

        row, col = self.state

        # Stochastic transitions: 80% intended, 20% random

        if np.random.rand() < 0.8:

            intended\_action = action

        else:

            intended\_action = np.random.choice(self.actions)

        # Compute next state

        if intended\_action == 'up':

            next\_row = max(row - 1, 0)

            next\_col = col

        elif intended\_action == 'down':

            next\_row = min(row + 1, self.size - 1)

            next\_col = col

        elif intended\_action == 'left':

            next\_row = row

            next\_col = max(col - 1, 0)

        elif intended\_action == 'right':

            next\_row = row

            next\_col = min(col + 1, self.size - 1)

        next\_state = (next\_row, next\_col)

        self.state = next\_state

        # Assign rewards

        if next\_state in self.terminal\_states:

            reward = 10

            done = True

        elif next\_state in self.pits:

            reward = -5

            done = True

        else:

            reward = -1  # Step penalty

            done = False

        return next\_state, reward, done

class MonteCarloAgent:

    def \_\_init\_\_(self, env, gamma=0.99, epsilon=1.0, epsilon\_decay=0.995, min\_epsilon=0.01):

        self.env = env

        self.gamma = gamma

        self.epsilon = epsilon

        self.epsilon\_decay = epsilon\_decay

        self.min\_epsilon = min\_epsilon

        self.Q = {}  # Q-table: {(state, action): value}

        self.returns = {}  # Stores returns for (state, action)

        self.actions = env.actions

        # Initialize Q-table and returns

        for row in range(env.size):

            for col in range(env.size):

                state = (row, col)

                for action in self.actions:

                    self.Q[(state, action)] = 0.0

                    self.returns[(state, action)] = []

    def get\_action(self, state):

        # ε-greedy policy

        if np.random.rand() < self.epsilon:

            return np.random.choice(self.actions)

        else:

            q\_values = [self.Q[(state, a)] for a in self.actions]

            return self.actions[np.argmax(q\_values)]

    def train(self, episodes=1000):

        for \_ in range(episodes):

            episode = []

            state = self.env.reset()

            done = False

            while not done:

                action = self.get\_action(state)

                next\_state, reward, done = self.env.step(action)

                episode.append((state, action, reward))

                state = next\_state

            # Update Q-values using episode returns

            G = 0

            for t in reversed(range(len(episode))):

                state\_t, action\_t, reward\_t = episode[t]

                G = self.gamma \* G + reward\_t

                self.returns[(state\_t, action\_t)].append(G)

                self.Q[(state\_t, action\_t)] = np.mean(self.returns[(state\_t, action\_t)])

            # Decay ε

            self.epsilon = max(self.min\_epsilon, self.epsilon \* self.epsilon\_decay)

class QLearningAgent:

    def \_\_init\_\_(self, env, gamma=0.99, alpha=0.1, epsilon=1.0, epsilon\_decay=0.995, min\_epsilon=0.01):

        self.env = env

        self.gamma = gamma

        self.alpha = alpha

        self.epsilon = epsilon

        self.epsilon\_decay = epsilon\_decay

        self.min\_epsilon = min\_epsilon

        self.Q = {}  # Q-table: {(state, action): value}

        self.actions = env.actions

        # Initialize Q-table

        for row in range(env.size):

            for col in range(env.size):

                state = (row, col)

                for action in self.actions:

                    self.Q[(state, action)] = 0.0

    def get\_action(self, state):

        # ε-greedy policy

        if np.random.rand() < self.epsilon:

            return np.random.choice(self.actions)

        else:

            q\_values = [self.Q[(state, a)] for a in self.actions]

            return self.actions[np.argmax(q\_values)]

    def train(self, episodes=1000):

        for \_ in range(episodes):

            state = self.env.reset()

            done = False

            while not done:

                action = self.get\_action(state)

                next\_state, reward, done = self.env.step(action)

                # Q-learning update

                max\_next\_q = max([self.Q[(next\_state, a)] for a in self.actions])

                self.Q[(state, action)] += self.alpha \* (

                    reward + self.gamma \* max\_next\_q - self.Q[(state, action)]

                )

                state = next\_state

            # Decay ε

            self.epsilon = max(self.min\_epsilon, self.epsilon \* self.epsilon\_decay)

def visualize\_policy(Q, size=4):

    policy = np.empty((size, size), dtype=str)

    for row in range(size):

        for col in range(size):

            state = (row, col)

            q\_values = [Q[(state, a)] for a in ['up', 'down', 'left', 'right']]

            best\_action = np.argmax(q\_values)

            arrows = ['↑', '↓', '←', '→']

            policy[row, col] = arrows[best\_action]

    # Mark terminal states and pits

    terminal\_states = {(3, 3)}

    pits = {(1, 1), (1, 2)}

    for (row, col) in terminal\_states:

        policy[row, col] = 'G'

    for (row, col) in pits:

        policy[row, col] = 'X'

    print("Final Policy:")

    print(policy)

# Initialize environment and agents

env = GridWorld()

mc\_agent = MonteCarloAgent(env)

q\_agent = QLearningAgent(env)

# Train Monte Carlo Agent

print("Training Monte Carlo Agent...")

mc\_agent.train(episodes=1000)

visualize\_policy(mc\_agent.Q, size=env.size)

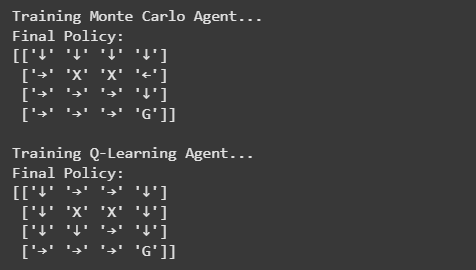
# Train Q-Learning Agent

print("\nTraining Q-Learning Agent...")

q\_agent.train(episodes=1000)

visualize\_policy(q\_agent.Q, size=env.size)

**Outputs :**

****

**Conclusion:**

In this experiment, we compared **Monte Carlo Control** and **TD Learning (Q-Learning)** in a grid world environment. Monte Carlo learns from complete episodes, while TD updates incrementally using next-state estimates. TD Learning converged faster and showed smoother value distribution compared to Monte Carlo, making it more efficient for real-time learning scenarios.

**Questions:**

**Q1: What is the main difference between Monte Carlo Control and TD Learning in terms of value updates?**  
**A:** The main difference between Monte Carlo Control and Temporal Difference (TD) Learning lies in how they update value estimates.

* **Monte Carlo Control:** It updates the value estimates only after the completion of an episode. The update is based on the total return (cumulative reward) obtained from the entire episode. This means it requires complete episodes to learn from and is typically slower in converging because updates are made less frequently.
* **TD Learning (Q-Learning):** It updates the value estimates incrementally at each time step during the episode. The update is based on the observed reward and the estimated value of the next state. This bootstrapping approach makes it more efficient in real-time environments as it does not wait for the episode to end before making updates.  
  In summary, Monte Carlo methods rely on full episodes to update values, while TD methods update continuously during interaction with the environment, making TD faster and more suitable for dynamic tasks.

**Q2: Why is TD Learning generally faster to converge compared to Monte Carlo Control?**  
**A:** TD Learning is generally faster to converge because it performs incremental updates using **bootstrapping**, where current value estimates are adjusted using both the immediate reward and the estimated value of the next state.

* **Frequent Updates:** Unlike Monte Carlo, which updates only at the end of each episode, TD updates occur at each step, allowing the agent to learn and adapt more quickly.
* **Real-Time Adaptation:** TD methods do not require the entire episode to finish before learning. This real-time adaptation is particularly beneficial in environments with long episodes or continuous interaction.
* **Efficient Exploration:** By learning continuously, TD methods like Q-learning can rapidly improve the policy and make better action choices during training.  
  As a result, TD Learning typically converges faster and is more practical for applications where rapid adaptation is essential.

**Q3: What are the advantages and disadvantages of using Monte Carlo Control and TD Learning in reinforcement learning tasks?**  
**A:**  
**Advantages of Monte Carlo Control:**

* **Simplicity:** The algorithm is easy to understand and implement since it directly learns from complete episodes.
* **Model-Free Learning:** It does not require knowledge of the environment's dynamics, making it suitable for model-free settings.
* **Accurate Returns:** Since it waits for the episode to end, it computes the actual total return, which can be more accurate.

**Disadvantages of Monte Carlo Control:**

* **Slow Convergence:** Updates occur only at the end of episodes, making it inefficient for long or infinite episodes.
* **High Variance:** The return values can vary significantly due to randomness in episodes, leading to high variance in updates.
* **Dependency on Episode Completion:** If an episode is too long or never terminates, learning becomes impractical.

**Advantages of TD Learning (Q-Learning):**

* **Fast Convergence:** Incremental updates allow faster learning compared to Monte Carlo methods.
* **Bootstrapping:** Uses estimated future rewards to update current values, making learning more efficient.
* **Applicability to Continuous Tasks:** Suitable for ongoing tasks without definite episode termination.

**Disadvantages of TD Learning:**

* **Bias:** Bootstrapping can introduce bias in value estimation since it relies on predicted future values.
* **Complexity:** Implementation can be more complex, especially with function approximation (e.g., neural networks).
* **Dependence on Environment Dynamics:** While TD can learn from direct interaction, inaccurate next-state predictions can affect performance.

In summary, Monte Carlo methods are simpler but slower, while TD Learning is faster but prone to bias. The choice between them often depends on the problem context, the environment's characteristics, and the need for real-time updates.