Drishtti Narwal
AI & ML (A2)
PRN No. 22070126037

Experiment 6

Title: Implementation of the algorithm: Q learning

Objective:

- To understand a policy that learns from experience and uses bootstrapping
- To implement Q-learning policy

Explanation/Stepwise Procedure/ Algorithm:

Q-learning Algorithm
Q-learning is a model-free reinforcement learning algorithm used to learn an optimal policy by updating Q-values, which represent the expected future rewards for a state-action pair.

Key Components:

- Q-table: A table where Q-values are stored for each state-action pair.
- State (s): The environment's current condition.
- Action (a): The possible actions the agent can take in a state.
- Reward (r): Feedback from the environment after taking an action.
- Policy: A strategy the agent follows to choose actions based on Q-values.

Q-value Update Rule:
The core of Q-learning is the update of the Q-value for a given state-action pair based on the agent's experience. The Q-value is updated using the following formula:

$$Q(s,a) = Q(s,a) + \alpha \left(r + \gamma \max_{a'} Q(s',a') - Q(s,a) \right)$$

Where:

- Q(s,a)Q(s, a)Q(s,a) is the current Q-value for state sss and action aaa.
- α\alphaα is the learning rate, which determines how much new information overrides the old Q-value.
- rrr is the reward received after taking action aaa in state sss.
- γ\gammaγ is the discount factor, which determines the importance of future rewards.
- max⁡a′Q(s′,a′)\max_{a'} Q(s', a')maxa′Q(s′,a′) is the maximum Q-value for the next state s′s's′ over all possible actions a′a'a′.

Policy Evolution:

- The agent starts by exploring the environment and learning from trial and error.
- Initially, Q-values are often set to zero, so the agent takes random actions.
- Over time, as Q-values are updated based on rewards, the agent starts favoring actions with higher Q-values, leading to more optimal behavior.
- Eventually, the agent's policy becomes more exploitative, selecting the best action based on the highest Q-value for each state.

Exploration-Exploitation Trade-off:

- Exploration: The agent tries random actions to discover new states and actions. This is important for ensuring the agent doesn't miss better actions that may not be initially obvious.
- Exploitation: The agent uses the learned Q-values to choose the best action, maximizing its cumulative reward.

In Q-learning, the epsilon-greedy policy is commonly used to balance exploration and exploitation:

- With probability ϵ\epsilonϵ, the agent chooses a random action (exploration).
- With probability 1−ϵ1 - \epsilon1−ϵ, the agent selects the action with the highest Q-value (exploitation).

As training progresses, ϵ\epsilonϵ typically decays, leading to more exploitation of the learned policy and less exploration.

Summary:
Q-learning allows an agent to learn an optimal policy through repeated interaction with the environment. By updating Q-values based on rewards and the maximum future rewards, the agent improves its decision-making process. The exploration-exploitation trade-off ensures that the agent balances discovering new strategies with optimizing known ones.
This method is powerful because it doesn't require a model of the environment and works with both small and large state spaces.

Learning Outcomes (3 to 5):

1. Understanding Q-learning: I gained a deep understanding of the Q-learning algorithm, specifically how the agent uses the Q-table to learn an optimal policy through updates based on rewards and future predictions. The update rule allows the agent to balance exploration and exploitation in a structured way.

2. Exploration vs. Exploitation: I learned how the epsilon-greedy strategy helps balance exploration (trying new actions) and exploitation (choosing the best-known actions). This trade-off is key to the agent's learning process and performance improvement over time.

3. Policy Evolution: I understood how the agent's policy evolves from random exploration to a more deterministic policy based on the learned Q-values. The agent progressively takes actions that maximize expected future rewards, converging toward an optimal policy.

4. Use of Libraries: I worked with NumPy for efficient array manipulations and Matplotlib for visualizing action-value tables and policies. These libraries helped me visualize the agent's learning progress and understand the Q-values across the state space.

5. Implementation of Q-learning: I learned how to implement Q-learning from scratch, including setting up the Q-table, defining the update rule, and designing an epsilon-greedy policy. This hands-on implementation provided a clear picture of how the agent interacts with the environment and learns optimal actions.

Assignment Questions/Practice Problems:

1. Describe the Q-value function in Q-learning. What is the role of the greedy policy in Q-learning?

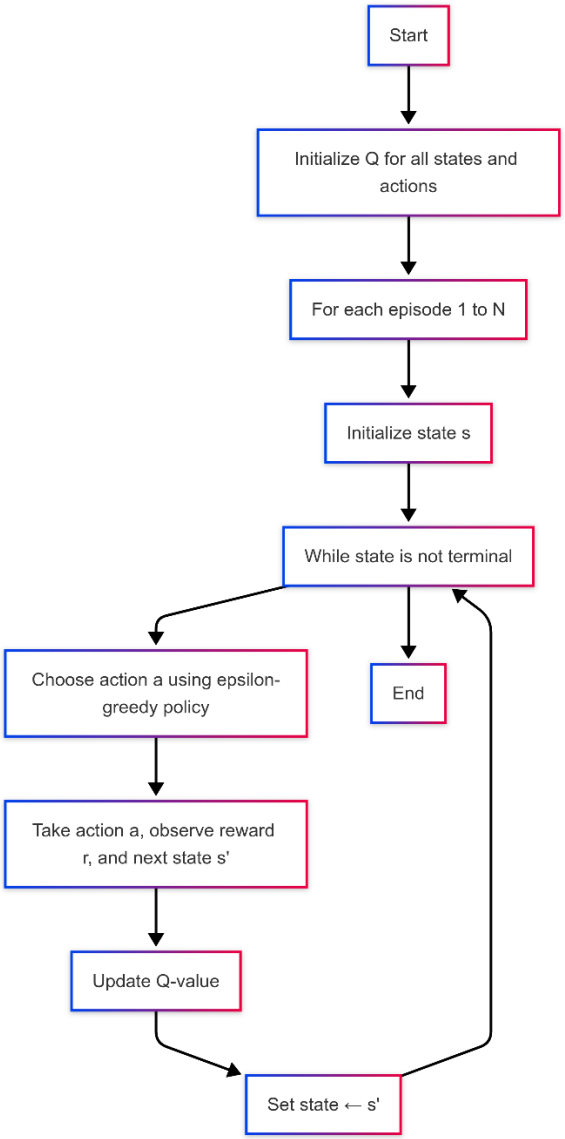
Q-value Function in Q-learning:

The **Q-value function** in Q-learning represents the expected future rewards of taking a particular action in a given state, and then following the optimal policy thereafter. It is used to guide the agent in making decisions by assigning values to state-action pairs. The higher the Q-value, the better the action is considered in that state. Over time, as the agent learns from its experiences, the Q-values are updated to reflect the most rewarding actions.

Role of the Greedy Policy in Q-learning:

The **greedy policy** in Q-learning encourages the agent to select the action with the highest Q-value in any given state, thereby maximizing its expected future rewards. As the agent's knowledge improves (through Q-value updates), the greedy policy becomes more effective in guiding the agent to the optimal solution. Initially, the agent may explore random actions to discover better options, but eventually, it relies more on the greedy policy to exploit the knowledge gained and make optimal decisions.

2. Construct and draw a flowchart for the Q-learning policy



Code:

```
import numpy as np
# NumPy is used for numerical computations, such as storing Q-values, updating them, and performing array operations.

import matplotlib.pyplot as plt
# Matplotlib is used for plotting graphs, such as visualizing the policy, action-values, or agent's performance.

from envs import Maze
# Imports the custom Maze environment, likely an instance of a grid-world or similar environment for the Q-learning agent to interact with.

from utils import plot_policy, plot_action_values, test_agent
# Utility functions:
# - plot_policy: Visualizes the learned policy, showing the best action to take at each state.
# - plot_action_values: Visualizes the learned action-value function Q(s, a).
# - test_agent: Evaluates the agent’s performance after learning, running it through several episodes.



env = Maze()


action_values = np.zeros((5, 5, 4))


def target_policy(state):
  # Defines a policy that selects the action with the highest Q-value for the given state (greedy policy).


  av = action_values[state]  # Retrieves the action-values (Q-values) for the given state.


  return np.random.choice(np.flatnonzero(av == av.max()))
  # Chooses the action with the maximum Q-value.
  # If there are multiple actions with the same maximum value, np.random.choice selects one of them randomly to break ties.
  # This ensures that the agent will always choose the best-known action (greedy approach).



def exploratory_policy(state):
  # Defines a policy that chooses a random action for the given state (purely exploratory).


  return np.random.randint(4)
  # Randomly selects an action from the action space (0, 1, 2, 3), assuming there are 4 possible actions.
  # This is the exploration aspect of Q-learning, where the agent occasionally explores random actions instead of always exploiting the best-known action.
  # The state argument is not used here because the policy is purely random regardless of the state.


plot_action_values(action_values)
# Calls the utility function to visualize the current action-value function (Q-values).
# This function likely generates a heatmap or some other graphical representation of the Q-values for each state-action pair.
# After the Q-table is initialized or updated, this visualization helps in understanding how the agent's knowledge
```

of the environment evolves.

It allows you to see which actions are preferred in each state based on the learned Q-values.

np.object = object

This line attempts to assign the built-in Python 'object' type to np.object, which was a deprecated feature in older versions of NumPy.

In recent versions of NumPy, 'np.object' is deprecated, and you should use 'np.object_' instead if you are defining arrays with the object data type.

plot_policy(action_values, env.render(mode='rgb_array'))

Calls the 'plot_policy' function to visualize the learned policy overlaid on the environment's current visual representation.

'action_values' represents the learned Q-values for state-action pairs, which the policy uses to determine the best action in each state.

'env.render(mode='rgb_array')' renders the current state of the environment as an image (likely a grid or maze), allowing for visual representation of the agent's current position and its interaction with the environment.

This visualization helps to see how the agent behaves in the environment based on its learned policy.

```
def q_learning(action_values, exploratory_policy, target_policy, episodes, alpha=0.1, gamma=0.99):
  # Implements the Q-learning algorithm for off-policy learning.
  # 'action_values': The Q-table (Q-values) storing the expected return for each state-action pair.
  # 'exploratory_policy': The exploration policy (usually epsilon-greedy) to choose actions.
  # 'target_policy': The target policy (greedy policy) used to select actions based on the maximum Q-value.
  # 'episodes': Number of episodes to train the agent.
  # 'alpha': Learning rate, determining how much new information overrides the old Q-value.
  # 'gamma': Discount factor for future rewards (usually close to 1).

  for episode in range(1, episodes + 1):
    state = env.reset()  # Reset the environment to start a new episode.
    done = False

    while not done:
      action = exploratory_policy(state)  # Choose an action using the exploratory policy (e.g., epsilon-greedy).
      next_state, reward, done, _ = env.step(action)  # Take the action and observe the next state and reward.
      next_action = target_policy(next_state)  # Select the next action using the target policy (greedy).

      # Retrieve the current Q-value for the state-action pair (s, a).
      qsa = action_values[state][action]

      # Retrieve the Q-value for the next state-action pair (s', a') using the target policy.
      next_qsa = action_values[next_state][next_action]

      # Update the Q-value for the current state-action pair using the Q-learning update rule:
      # Q(s, a) ← Q(s, a) + α * [R + γ * max(Q(s', a')) - Q(s, a)]
```

```
        action_values[state][action] = qsa + alpha * (reward + gamma * next_qsa - qsa)

        state = next_state  # Move to the next state.

# Call the Q-learning function to train the agent for 100 episodes.
q_learning(action_values, exploratory_policy, target_policy, 100)

plot_action_values(action_values)
# Visualizes the Q-values (action-values) after training the Q-learning agent.
# This function will generate a graphical representation of the Q-table for each state-action pair.
# It helps to see how the Q-values have been updated across episodes, showing which actions are preferred in each state.
# The plot might display a heatmap or similar visualization to highlight the learned action values.

plot_policy(action_values, env.render(mode='rgb_array'))
# Calls the 'plot_policy' function to overlay the learned policy on the current visual representation of the environment.
# 'action_values' represents the learned Q-values for state-action pairs, which are used to determine the best action for each state.
# 'env.render(mode='rgb_array')' renders the current state of the environment as an image (likely a grid or maze), which is used as the backdrop for the policy visualization.
# This visualization shows the agent's learned policy (best actions) on the maze or environment, indicating how the agent would navigate the environment.

test_agent(env, target_policy, episodes=5)
# Calls the 'test_agent' function to evaluate the performance of the trained agent.
# 'env' is the environment (the maze), and 'target_policy' is the learned greedy policy (maximizing Q-values).
# 'episodes=5' specifies the number of episodes the agent will run to test its performance.
# The function will run the agent through the specified number of episodes, where it follows the target (greedy) policy.
# It will likely output performance metrics like total rewards per episode, number of steps taken, or success in reaching the goal.
```

References:

1. https://medium.com/@goldengrisha/a-beginners-guide-to-q-learning-understanding-with-a-simple-gridworld-example-2b6736e7e2c9
2. https://www.geeksforgeeks.org/q-learning-in-python/