CSE - 584 HomeWork 2

Name : Raghavendra Jagirdar
PSU Id : 911938232

1. Abstract

This code implements a Q-Learning algorithm to solve a simple problem in a grid environment. The grid environment consists of a 2D grid where an agent starts at one location and tries to reach a goal location while avoiding obstacles. The Q-Learning algorithm helps the agent learn the best policy through interactions with the environment. At each step, the agent receives feedback in the form of rewards based on its actions, and it updates a Q-value table to improve its decisions. The main goal is to maximize the cumulative reward by finding an optimal policy that specifies the best action to take in each state.

2. **Annotated Code**

```python
import numpy as np

# Define the environment parameters
num_states = 16  # Number of states in the grid (4x4 grid, for instance)
num_actions = 4  # Number of actions (up, down, left, right)
q_table = np.zeros((num_states, num_actions))  # Initialize Q-table with zeros

# Hyperparameters for the Q-learning algorithm
alpha = 0.1       # Learning rate (how much we accept new information)
gamma = 0.9       # Discount factor (how much we consider future rewards)
epsilon = 0.1     # Exploration factor (probability of choosing random action)

# Function to choose an action using the epsilon-greedy strategy
def choose_action(state):
    # With probability epsilon, choose a random action (exploration)
    if np.random.uniform(0, 1) < epsilon:
        return np.random.randint(0, num_actions)
    else:
        # Choose the action with the highest Q-value (exploitation)
        return np.argmax(q_table[state, :])

# Function to update the Q-value table
def update_q_table(state, action, reward, next_state):
    # Get the maximum Q-value for the next state
    max_q_next = np.max(q_table[next_state, :])
    # Calculate the new Q-value using the Q-learning formula
    q_table[state, action] = q_table[state, action] + \
        alpha * (reward + gamma * max_q_next - q_table[state, action])

# Simulate the environment
def simulate_environment(state, action):
    # Example environment dynamics
    # Assume the agent moves to a new state based on action taken
    next_state = state  # Placeholder for the next state
    reward = -1  # Default reward is -1 (penalty for each step)

    # Example transitions (these should be customized for a real problem)
```

Ln 20, Col 44   Spaces: 4   UTF-8   LF   Julia

```python
    max_q_next = np.max(q_table[next_state, :])
    # Calculate the new Q-value using the Q-learning formula
    q_table[state, action] = q_table[state, action] + \
        alpha * (reward + gamma * max_q_next - q_table[state, action])

# Simulate the environment
def simulate_environment(state, action):
    # Example environment dynamics
    # Assume the agent moves to a new state based on action taken
    next_state = state  # Placeholder for the next state
    reward = -1  # Default reward is -1 (penalty for each step)

    # Example transitions (these should be customized for a real problem)
    if state == 0 and action == 1:  # If at state 0 and taking action 'down'
        next_state = 4
        reward = 0  # Neutral reward
    elif state == 4 and action == 1:  # If at state 4 and taking action 'down'
        next_state = 8
        reward = 10  # Reward for reaching goal state

    # Return the next state and reward
    return next_state, reward

# Main Q-Learning loop
num_episodes = 1000  # Total episodes for training
for episode in range(num_episodes):
    state = 0  # Starting state for each episode (e.g., top-left corner of grid)

    while state != 8:  # Continue until goal state (e.g., bottom-right corner)
        action = choose_action(state)  # Choose action based on epsilon-greedy
        next_state, reward = simulate_environment(state, action)  # Simulate env
        update_q_table(state, action, reward, next_state)  # Update Q-table
        state = next_state  # Move to the next state

# Display the learned Q-table
print("Learned Q-Table:")
print(q_table)
```

Ln 20, Col 44   Spaces: 4   UTF-8   LF   Julia

Output :

```
Learned Q-Table:
[[ 6.35806493  9.          6.72638768  6.47073138]
 [ 0.          0.          0.          0.        ]
 [ 0.          0.          0.          0.        ]
 [ 0.          0.          0.          0.        ]
 [ 7.4763163  10.          7.72345135  7.28964404]
 [ 0.          0.          0.          0.        ]
 [ 0.          0.          0.          0.        ]
 [ 0.          0.          0.          0.        ]
 [ 0.          0.          0.          0.        ]
 [ 0.          0.          0.          0.        ]
 [ 0.          0.          0.          0.        ]
 [ 0.          0.          0.          0.        ]
 [ 0.          0.          0.          0.        ]
 [ 0.          0.          0.          0.        ]
 [ 0.          0.          0.          0.        ]
 [ 0.          0.          0.          0.        ]]
```

Explanation of the Core Sections

Here are the core functions with detailed comments added to each line:

1) **`choose_action(state)`**
    a) Implements the epsilon-greedy strategy, balancing exploration and exploitation.
    b) With probability `epsilon`, it selects a random action to explore new possibilities.
    c) Otherwise, it chooses the action with the highest Q-value to exploit known information.
2) **`update_q_table(state, action, reward, next_state)`**
    a) Calculates the maximum Q-value for the next state, representing the expected future reward.
    b) Updates the Q-value of the current state-action pair using the Q-Learning update formula: Q(s,a)←Q(s,a)+α(r+γmaxa′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′maxQ(s′,a′)−Q(s,a))
    c) Here, Q(s,a)Q(s, a)Q(s,a) is the Q-value for the current state-action pair, α\alphaα is the learning rate, rrr is the immediate reward, γ\gammaγ is the discount factor, and maxa′Q(s′,a′)\max_{a'} Q(s', a')maxa′Q(s′,a′) is the maximum Q-value for the next state.
3) **`simulate_environment(state, action)`**
    a) Simulates environment dynamics based on the current state and action.
    b) Returns the next state and reward to be used for updating the Q-table.
    c) In this simple implementation, the dynamics are manually defined; in more complex scenarios, this function would interact with an actual environment (e.g., OpenAI Gym).