Task 2 Documentation

Overview

The Python script implements a **Q-learning** agent that learns to navigate a maze using reinforcement learning techniques. It defines a **Maze Environment** and a **Q-learning agent** and includes a training and evaluation process.

1. Maze Model Design

Explanation of the Maze Navigation Model

The maze is represented as a **2D grid** where:

- **1 represents walls** that the agent cannot pass through.
- **0 represents open paths** where the agent can move.
- The agent starts at a predefined position and must reach a goal position using a set of discrete actions (Up, Down, Left, Right).

The environment follows a **discrete state-action space** model, where each grid cell is treated as a unique state, and the agent learns an optimal policy to reach the goal while avoiding walls.

Overview of the Reinforcement Learning Approach Used

- The **Q-learning algorithm** is used to train the agent.
- A **Q-table** is maintained, where each entry corresponds to a state-action pair and stores the estimated cumulative future reward.
- The agent follows an **epsilon-greedy policy** to balance exploration (random moves) and exploitation (choosing the best-known move).
- Rewards are assigned based on the following logic:
    - **+10** for reaching the goal.
    - **-1** for hitting a wall.
    - **-0.1** for each step to encourage the shortest path.

2. Maze Environment Class

Purpose:

- Represents the maze environment where an agent navigates.
- Defines the maze layout, agent movements, reward system, and rendering.

Attributes:

- maze: A 2D NumPy array where **1 represents walls** and **0 represents open spaces**.
- start: The starting position of the agent.
- goal: The goal position.
- agent_pos: The current position of the agent.
- rows, cols: Dimensions of the maze.
- action_space_size: Number of available actions (Up, Down, Left, Right).
- state_space_size: Total number of states (maze cells).

Methods:

- reset(): Resets the agent to the start position and returns the initial state.
- get_state(): Converts the 2D agent position into a unique integer state.
- step(action): Moves the agent in the maze based on the chosen action and returns the new state, reward, and termination status.
- render(): Displays the maze with the agent's position.

3. QLearningAgent Class

Purpose:

- Implements the **Q-learning algorithm** to learn the optimal path through the maze.
- Uses an **epsilon-greedy policy** for exploration vs. exploitation.

Attributes:

- q_table: A NumPy array storing Q-values for each state-action pair.
- lr: Learning rate for Q-value updates.
- gamma: Discount factor for future rewards.
- epsilon: Probability of selecting a random action (exploration rate).
- epsilon_decay: Decay rate to gradually reduce exploration.
- action_space_size: Number of possible actions.

Methods:

- choose_action(state): Selects an action using the epsilon-greedy policy.

- update_q_table(state, action, reward, next_state): Updates the Q-values using the **Bellman equation**.
- decay_epsilon(): Decreases the exploration rate over time.

4. Training Process

The agent is trained over **1000 episodes**:

1. The environment is reset.
2. The agent selects an action based on the epsilon-greedy policy.
3. It moves in the maze and receives a reward.
4. The Q-table is updated.
5. The process repeats until the agent reaches the goal or a termination condition is met.
6. epsilon decays after each episode to reduce exploration over time.

5. Training Results and Performance Analysis

After **1000 training episodes**, the agent learns to navigate the maze efficiently. The performance is evaluated based on:

- **Success rate**: The percentage of test runs where the agent reaches the goal.
- **Convergence of Q-values**: The Q-table stabilizes, indicating the agent has learned an optimal policy.
- **Reduction in step count**: The number of steps taken to reach the goal decreases over time.

In the **evaluation phase (10 test episodes)**:

- The agent operates with epsilon = 0, meaning it only exploits learned policies.
- A **high success rate** suggests that the agent has effectively learned the optimal path.

6. Evaluation Process

- The trained agent is tested over **10 episodes** with epsilon = 0 (pure exploitation).
- Success rate is calculated as the fraction of episodes where the agent reaches the goal.
- The final success rate is displayed as output.

7. Key Features

- **Q-learning Algorithm**: Uses reinforcement learning to find the shortest path.

- **Epsilon-greedy Exploration**: Balances exploration (random actions) and exploitation (best-known actions).
- **Dynamic Learning**: Q-values improve over time based on rewards.
- **Text-based Visualization**: The render() function provides a simple way to visualize the maze and agent movement.

This implementation has provided me a strong foundation for reinforcement learning applications in grid-based environments.