Maze Navigation with Q-Learning

Introduction

The maze navigation implements a reinforcement learning system using the Q-Learning algorithm. The agent starts at a defined position and must reach the goal while avoiding obstacles and walls. Through trial and error, the agent learns an optimal policy to maximize rewards by reaching the goal efficiently.

Key Features

1. **Custom Maze Environment**:
    - The maze is represented as a grid where cells can be walkable, walls, the agent's starting position, or the goal.
2. **Reinforcement Learning (Q-Learning)**:
    - The agent learns by interacting with the environment, receiving rewards or penalties for its actions, and updating a Q-table that helps it make decisions.
3. **Visualization**:
    - The agent's training process and movements are visualized using `matplotlib`.

Dependencies

The following Python libraries are required:

- **NumPy**: For handling the grid and numerical operations.
- **Matplotlib**: For visualizing the maze and the agent's movements.

Install these libraries using:

pip install numpy matplotlib

How It Works

1. Maze Environment

The maze is defined as a 2D grid:

- `0`: Walkable path
- `1`: Wall or obstacle
- The agent starts at `(0, 0)` and must reach the bottom-right corner `(rows-1, cols-1)`.

The environment provides methods to:

- `reset`: Reset the maze and the agent to the starting position.
- `step`: Move the agent based on the chosen action, return the new state, reward, and whether the goal is reached.
- `render`: Visualize the maze and the agent's position.

2. Reinforcement Learning with Q-Learning

The Q-Learning algorithm is used to train the agent:

- **Q-Table**: A matrix where rows/columns represent the state (grid cells), and the depth represents actions (up, down, left, right).
- **Actions**:
  - `0`: Move up
  - `1`: Move down
  - `2`: Move left
  - `3`: Move right
- **Rewards**:
  - `+1` for reaching the goal.
  - `-1` for hitting a wall or invalid move.
  - `0` for valid intermediate moves.
- **Learning Process**:
  - The agent explores the environment by taking actions and updates its Q-table using the Q-Learning formula: $Q(s,a) \leftarrow Q(s,a) + \alpha [r + \gamma \max_a Q(s',a) - Q(s,a)]$ $Q(s, a) \leftarrow Q(s, a) + \alpha \left[r + \gamma \max_a Q(s', a) - Q(s, a) \right]$ Where:
    - α: Learning rate
    - γ: Discount factor
    - r: Reward
    - s: Current state
    - a: Action

3. Training

The agent trains for a specified number of episodes. In each episode:

- The maze is reset.
- The agent interacts with the environment, learning an optimal policy by exploring and exploiting actions based on the Q-table.
- The exploration rate (epsilon) decays over time to favor exploitation of learned knowledge.

4. Visualization

The training progress (total reward per episode) and the agent's navigation through the maze are visualized using `matplotlib`.

Code Structure

Classes

1. **`MazeEnvironment`**
    - Represents the maze, handles agent movement, and provides feedback.
    - Methods:
        - `reset`: Resets the agent to the start position.
        - `step`: Executes an action, updates the agent's state, and calculates the reward.
        - `render`: Displays the maze grid and agent's position.
2. **`QLearningAgent`**
    - Implements the Q-Learning algorithm.
    - Methods:
        - `choose_action`: Selects an action based on exploration or exploitation.
        - `update_q_value`: Updates the Q-table using the Q-Learning formula.
        - `train`: Trains the agent through multiple episodes.

How to Run the Code

Step 1: Install Required Libraries

Run the following command:

pip install numpy matplotlib

Step 2: Save the Code

Save the full code in a Python file (e.g., `maze_navigation.py`).

Step 3: Run the Code

Run the script from your terminal:

python maze_navigation.py

Results

1. **Training Rewards**:
    - A graph shows the total reward per episode during training. This demonstrates the agent's improvement over time.
2. **Maze Visualization**:
    - The maze with the agent's position is displayed at each step as the agent navigates toward the goal.

Customization

1. **Maze Design**:
    - Modify the `maze` array to create different mazes.
2. **Training Parameters**:
    - Adjust learning rate (`lr`), discount factor (`gamma`), and epsilon parameters in the `QLearningAgent` class for better performance.
3. **Episode Count**:
    - Increase or decrease the number of training episodes in the `train` method.

Limitations

1. **Q-Learning Scalability**:

    - Q-Learning may struggle with very large mazes due to the high dimensionality of the Q-table. For such cases, Deep Q-Learning (DQN) can be implemented.
2. **Environment Complexity**:
    - The current implementation assumes a static maze. For dynamic or more complex environments, additional modifications are needed.

Future Enhancements

1. **Deep Q-Learning (DQN)**:
    - Use a neural network instead of a Q-table for larger mazes.
2. **Dynamic Maze**:
    - Add moving obstacles or time-based challenges.
3. **Multiple Goals**:
    - Train the agent to navigate mazes with multiple goals or checkpoints.

Conclusion

This demonstrates the power of reinforcement learning using Q-Learning to teach an agent to navigate a maze. By exploring and learning from feedback, the agent progressively optimizes its path, avoiding obstacles and reaching the goal efficiently.