# Homework #2: Reinforcement Learning Code Analysis

Yu Tianrun

October 28, 2024


## Abstract

This code implements a Q-Learning algorithm to solve a maze navigation problem. The environment is modeled as a 4x5 grid where each cell represents an open path, an obstacle, or a goal. The primary objective of the reinforcement learning agent is to navigate from the starting position to the goal efficiently, minimizing the number of steps taken. The agent learns optimal policies by iteratively updating a Q-table, which estimates the value of taking specific actions in given states. The algorithm employs an epsilon-greedy strategy to balance exploration of new actions and exploitation of known high-reward actions. Throughout multiple training episodes, the agent refines its policy, enabling it to effectively traverse the maze and consistently reach the goal. This implementation demonstrates the fundamental principles of reinforcement learning, including state representation, action selection, reward processing, and policy optimization.


## Code with Detailed Comments

```python
import numpy as np
import random

# Define the maze environment
class MazeEnv:
    def __init__(self):
        # 4x5 maze grid: 0 is free cell, -1 is obstacle, 1 is goal
        self.maze = np.array([[0, 0, 0, 0, -1],
                              [0, -1, 0, -1, 0],
                              [0, -1, 0, 1, 0],
                              [0, 0, 0, -1, 0]])
        self.start_position = (0, 0)
        self.reset()

    def reset(self):
        self.position = self.start_position
        return self.position

    def step(self, action):
        # Define possible movements: right, down, left, up
        moves = [(0, 1), (1, 0), (0, -1), (-1, 0)]
        next_position = (self.position[0] + moves[action][0],
                         self.position[1] + moves[action][1])

        # Check if next position is within bounds and not an obstacle
        if (0 <= next_position[0] < self.maze.shape[0] and
            0 <= next_position[1] < self.maze.shape[1] and
            self.maze[next_position] != -1):
            self.position = next_position

        # Check if goal is reached
        if self.maze[self.position] == 1:
            return self.position, 1, True  # Goal reached
        else:
            return self.position, -0.1, False  # Step penalty

# Q-Learning parameters
alpha = 0.1    # Learning rate
gamma = 0.9    # Discount factor
```

```python
epsilon = 0.1  # Exploration rate
episodes = 500 # Number of training episodes

# Initialize Q-table for the 4x5 grid with 4 actions per state
q_table = np.zeros((4, 5, 4))

# Q-Learning algorithm
env = MazeEnv()

for episode in range(episodes):
    state = env.reset()
    done = False

    while not done:
        # Epsilon-greedy strategy for action selection
        if random.uniform(0, 1) < epsilon:
            action = random.choice(range(4))  # Random action (exploration)
        else:
            action = np.argmax(q_table[state[0], state[1]])  # Best action (exploitation)

        # Take action and observe the outcome
        next_state, reward, done = env.step(action)

        # Update Q-value for the current state-action pair
        old_value = q_table[state[0], state[1], action]
        next_max = np.max(q_table[next_state[0], next_state[1]])
        new_value = (1 - alpha) * old_value + alpha * (reward + gamma * next_max)
        q_table[state[0], state[1], action] = new_value

        # Transition to the next state
        state = next_state
```

Listing 1: Reinforcement Learning Code for Maze Navigation with Detailed Comments

# Core Section with Line-by-Line Comments

```python
# Initialize environment and Q-table for the Q-learning algorithm
env = MazeEnv()  # Instantiate the MazeEnv class to create the maze environment
q_table = np.zeros((4, 5, 4))  # Initialize a Q-table with dimensions corresponding to the maze grid (4 rows, 5 columns) and 4 possible actions

# Define Q-Learning parameters
alpha = 0.1    # Learning rate: determines the extent to which newly acquired information overrides old information
gamma = 0.9    # Discount factor: measures the importance of future rewards versus immediate rewards
epsilon = 0.1  # Exploration rate: probability of choosing a random action instead of the best-known action
episodes = 500 # Total number of training episodes

# Q-Learning algorithm: training the agent over multiple episodes
for episode in range(episodes):  # Loop through each episode
    state = env.reset()  # Reset the environment to the starting state and obtain the initial state
    done = False  # Flag to indicate whether the goal has been reached

    while not done:  # Continue the episode until the goal is reached
        if random.uniform(0, 1) < epsilon:  # Epsilon-greedy strategy: decide whether to explore or exploit
            action = random.choice(range(4))  # Exploration: randomly select an action (0-3 corresponding to up, down, left, right)
        else:
            action = np.argmax(q_table[state[0], state[1]])  # Exploitation: choose the action with the highest Q-value for the current state

        # Execute the chosen action in the environment
        next_state, reward, done = env.step(action)  # Perform the action and observe the next state, reward, and whether the goal is reached

        # Q-value update using the Q-Learning update rule
        old_value = q_table[state[0], state[1], action]  # Retrieve the current Q-value for the state-action pair
```

```
        next_max = np.max(q_table[next_state[0], next_state[1]])  # Find the maximum Q-value for the next state across all possible actions

        # Calculate the new Q-value using the Bellman equation
        new_value = (1 - alpha) * old_value + alpha * (reward + gamma * next_max)
        q_table[state[0], state[1], action] = new_value  # Update the Q-table with the new Q-value

        state = next_state  # Transition to the next state for the subsequent step
```

Listing 2: Core Q-Learning Algorithm