Student:
Syma Afsha [symaafsha.eece@gmail.com]

Lab 5: Q-Learning Algorithm

Introduction

Reinforcement Learning (RL) is a sort of machine learning in which an agent learns to make decisions by interacting with the environment around it in order to achieve a predefined goal. Learning to connect situations to behaviors in a way that optimizes a numerical reward is part of the process. RL is distinguished by a focus on trial-and-error learning, balancing the problem between exploration (identifying optimal actions) and exploitation (gaining higher rewards). It also addresses delayed rewards, where long-term advantages are frequently prioritized over immediate satisfaction. TThe objective of this practical exercise is to use a Reinforcement Learning algorithm to develop a policy that guides a robot to a desired location. The algorithm used to create a policy is the Q-learning algorithm.

Implementation

The problem is to identify the goal in a finite 2D environment that is closed and has certain obstacles. The environment has 20x14=280 states. The robot can only perform four different actions: left, up, right, or down actions. As a result, the size of the Q function will be 280x4=1120 cells.The dynamics of the robot can be found in any free cell. Unless there is a barrier or a wall in front of it, the robot will advance one cell per iteration in the direction of the action that we select. Because the goal is to get to the goal location as quickly as feasible, the reinforcement function will return -1 in all cells except in the goal cell, where the reward will be +1. The cell that contains the goal is (3,17).

Deployment of Methods

Method: `get_start()`

The `get_start` method chooses a starting position for an agent in a map at random. It checks for free spaces with no obstacles and then selects one at random. This ensures that the agent begins in a safe, obstacle-free environment. The method is shown in Figure 1

Method: step()

The step function moves the agent to the left, up, right, or down and changes its state. It employs a lexicon to convert actions into movements. The function determines whether the new position is valid and within the map's boundaries, avoiding obstacles. The agent's state is updated if the position is valid. It then checks to see if the agent has achieved its aim and rewards accordingly.

```
def get_start(self):
  # start the agent in a random position within the map:
  valid_positions=np.argwhere(self.map==0) #Find positions that are not obstacles
  #Select a random valid position as starting
  start_position= valid_positions[np.random.choice(len(valid_positions))]
  return start_position
```

Figure 1: Code of get_start() method

```
def step(self, action):
  # this function applies the action taken and returns the next state, the reward and a variable that says if the goal is reached
  # action: 0 = LEFT, 1 = UP, 2 = RIGHT, 3 = DOWN
  move_dictionary={0:(0,-1), 1:(-1,0), 2:(0,1), 3:(1,0)} #define movement for each action
  move=move_dictionary[action]#get the movement for the current action
  #calculate new state
  new_state=self.current_state +np.array(move)
  # check if new position is within map bounds and not an obstacle
  if (
      0 <= new_state[0] < self.map.shape[0]
      and 0 <= new_state[1] < self.map.shape[1]
      and self.map[new_state[0], new_state[1]] == 0
  ):
      self.current_state = new_state
  # check if the goal is reached
  goal_reach=np.array_equal(self.current_state, self.goal)

  if goal_reach: #assign reward
    reward=1
  else:
    reward=-1
  return self.current_state, reward, goal_reach
```

Figure 2: Code of step() method

Finally, the current state, reward, and goal-attainment status are returned. The following Figure 2 illustrates the method.

Class: Q-Learning

At each episode, the Q Learning class builds and trains the policy based on the information provided by the environment. In order to implement this class, we must create the `epsilon_greedy_policy()` and `episode()` methods in this report. After training, the optimal policy and value function are determined.

Method: `epsilon_greedy_policy()`

The `epsilon_greedy_policy()` method in this code determines an agent's behavior in a particular condition. It chooses an action at random, based on the epsilon value, to promote the exploration of new methods. When not deciding at random, it chooses the best-known action for that state based on the greatest Q-value from its learning. This method achieves a balance between trying new activities and applying what it has learned to be effective. The method depicts in Figure

```
def epsilon_greedy_policy(self, s):

    # with probability epsilon, choose a random action
    if np.random.random() < self.epsilon:
        return np.random.randint(0, self.env.actions - 1)
    # otherwise, choose the action with the highest Q-value for the current state
    else:
        return np.argmax(self.Q[s[0], s[1]])
```

Figure 3: Code of epsilon_greedy_policy() method

Figure 4: Code of episode() method

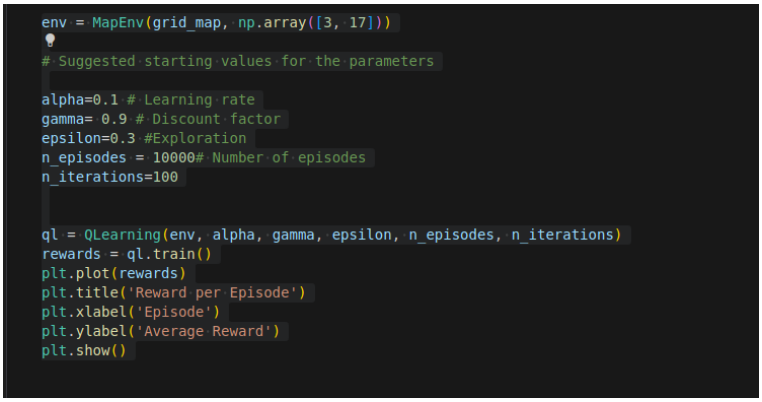

Figure 5: Training

Method: `episode()`

The `episode` method executes a single episode in a reinforcement learning environment. It initializes the state and then loops for a predetermined amount of iterations. Within each iteration, it chooses an action according on the epsilon-greedy policy, executes the action, and observes the new state and reward. The function updates the Q-value for the current state and action, accumulates the overall reward, and updates the current state. If the 'done' condition is met, signifying the end of the event, the loop is broken. The function returns the total accumulated award. The Figure 4 shows the episode() method.

Results

The code creates a Q-learning agent (ql) with a given environment (env), learning rate (alpha), discount factor (gamma), exploration probability (epsilon), number of episodes (n_episodes), and iterations per episode (n_iterations). The agent is trained across multiple episodes, with rewards recorded for each one. The training method is depicted in Figure 5 The graph displays the average reward per episode during the initial phase of training a Q-learning agent. The performance improves dramatically in the first few episodes, indicating that the agent is quickly learning from the environment. After the initial rapid ascent, the curve flattens out, indicating that the agent has reached a performance peak where additional increases are minimal or that the agent regularly achieves an equivalent amount of reward per episode. The Figure 6 illustrates the performance

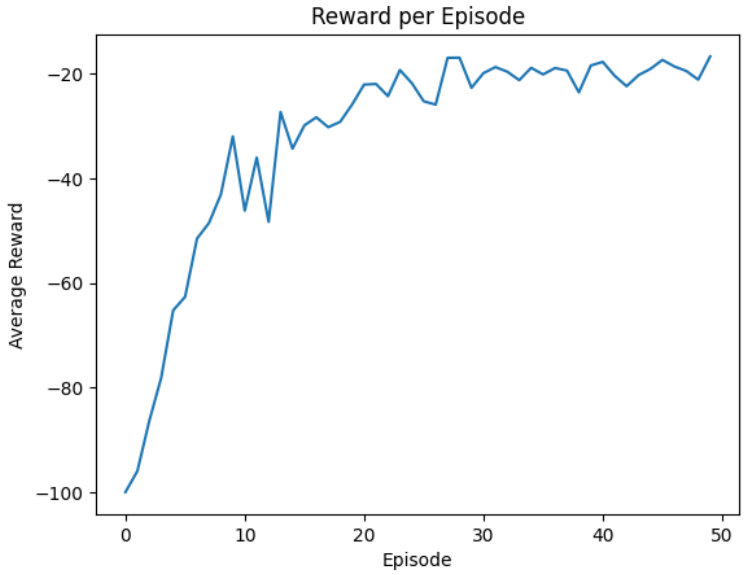

Figure 6: Performance Evaluation

evaluation. The code generates visual maps for the value function and optimal policy of the Q-learning model. The value function map depicts how likely each state is to get future rewards. The optimal policy map depicts the best action for each state (up, down, left, right). Both maps employ colors to represent different values or actions, making it easier to see the model's strategy. The Plot value function and optimal policy are shown in Figure 7 and 8. When the training is finished, we may see what the robot has learned. This code executes a single episode in a map environment, using the trained policy from the Q-learning model. It initializes the environment before entering a loop in which it continuously selects actions based on the optimal policy defined during training. At each step, it does the chosen action, updates the state, obtains a reward, and determines whether the episode is complete (done). After each action, the environment is visually rendered, displaying the agent's route through the map. This loop will run until the episode is finished (when done is True). The following figure 9 shows the test current policy.

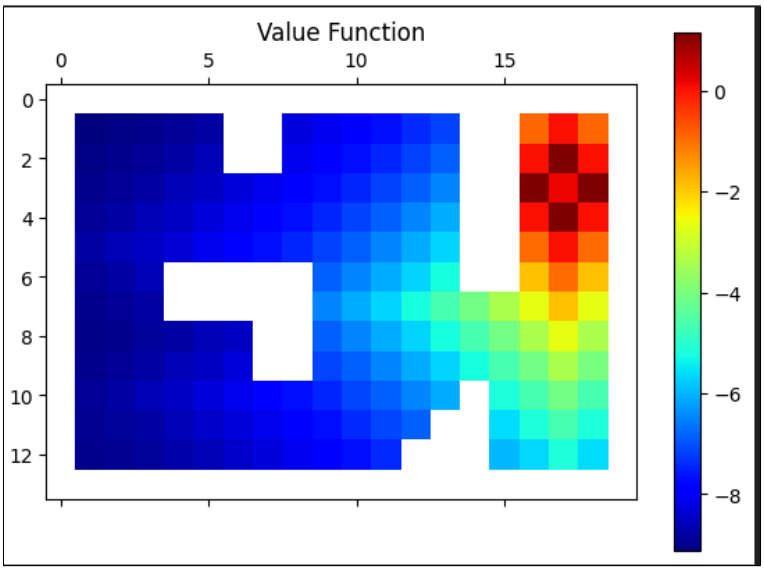

Figure 7: Value Function

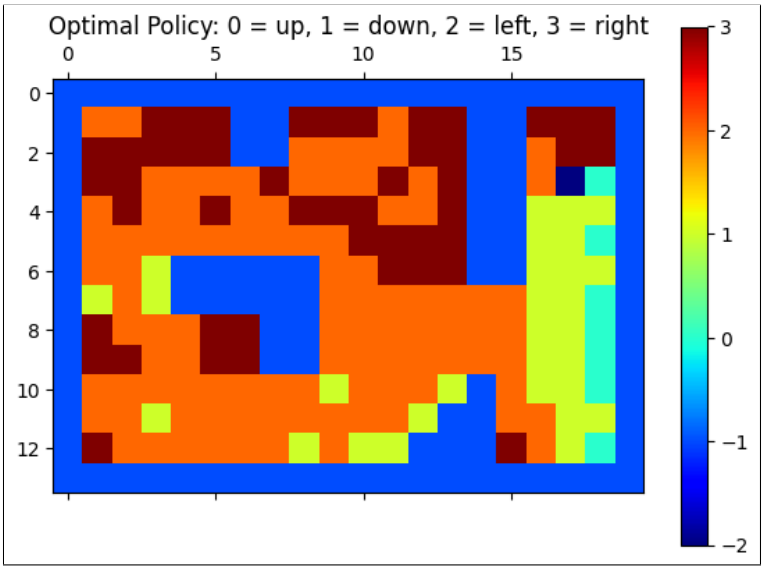

Figure 8: Optimal Policy

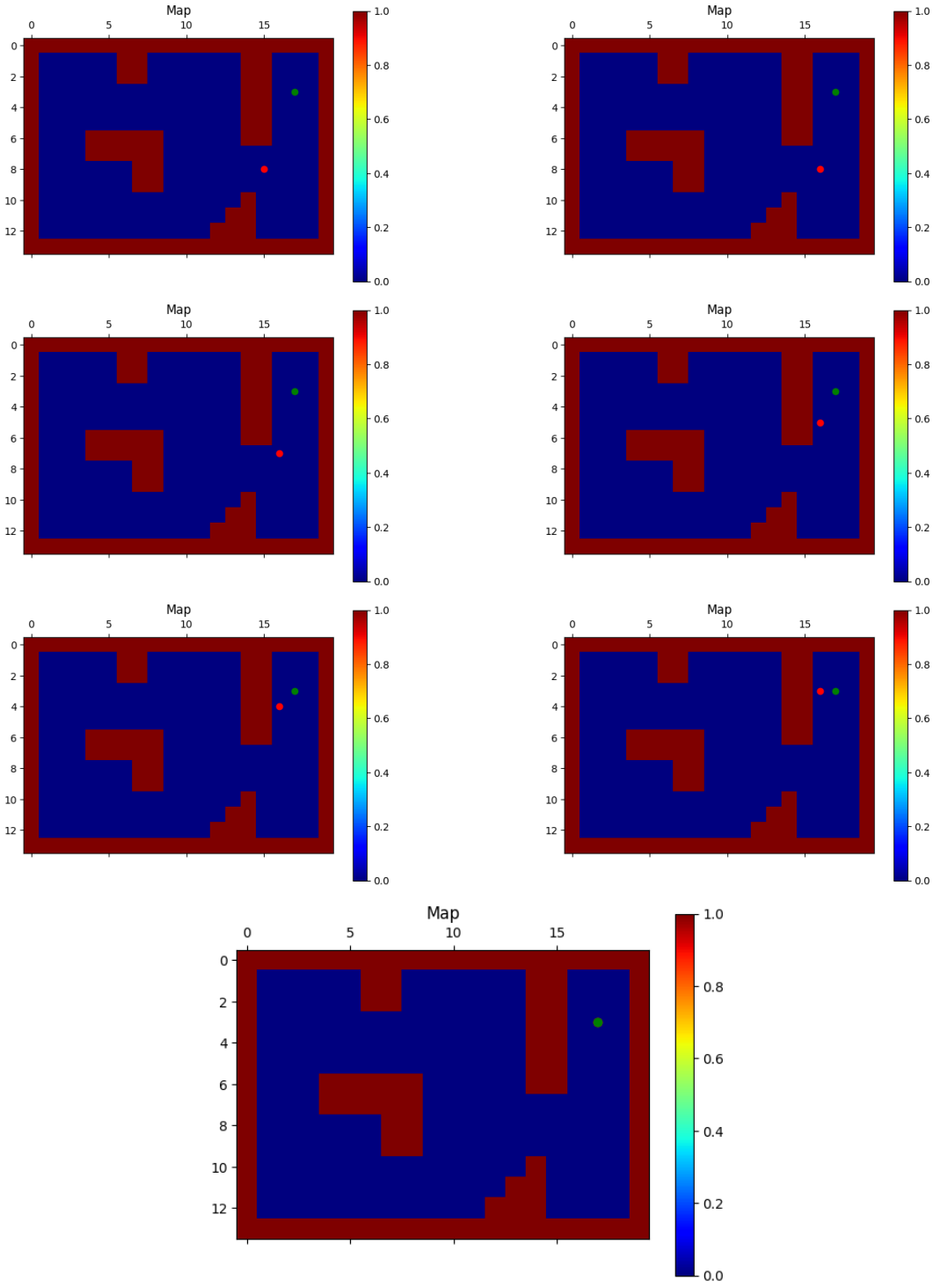

Figure 9: Test Current Policy

Discussion of how the parameters affect the training and conclusions for the results obtained

The model's learning process is affected by the parameters in Q-learning training: alpha, gamma, epsilon, n_episodes, and n_iterations. A greater alpha can result in faster learning but, if too high, might lead to unstable training. Gamma weighs the relevance of future rewards, with a higher gamma implying more strategic, long-term actions. Exploration is controlled by Epsilon; a larger value stimulates the agent to try new actions, which is essential for avoiding local optima. The number of episodes and iterations impacts the quantity of learning experience and the granularity of learning within each episode, with greater numbers generally resulting in better-informed decision-making if computational resources allow.

Problem Faced

The main issue we encountered during the installation of the Q learning algorithm was testing the performance evaluation. The problem was solved by altering the parameters.

Conclusion

The Q-learning experiment in this lab report successfully constructed a reinforcement learning algorithm to teach a robot to navigate a 2D environment containing obstacles. The robot, which could move in four directions, was designed to effectively reach a target point. The Q-learning approach entailed creating a Q table, executing training episodes, and assessing performance. The training procedure lasted 10,000 episodes, with performance reviews every 200 episodes, averaging the awards over the previous 50 episodes. This approach proved the successful use of Q-learning in complicated contexts, emphasizing its future applications in adaptive decision-making and path optimization.