Universitat de Girona
COMPUTER VISION AND ROBOTICS RESEARCH INSTITUTE (VICOROB)

Autonomous Systems

Lab Report

Q-Learning Algorithm

No.	Full Name	Student ID
1	Amine Dhemaied	u1985972
2	Joseph Oloruntoba Adeola	u1988552

Instructor: Marta Real

UDG, JANUARY 2023

1 Introduction

Reinforcement learning (RL) is a type of machine learning in which an agent learns to make decisions by interacting with an environment and receiving feedback in the form of rewards or penalties. The goal of the agent is to learn a policy, which maps states of the environment to actions, that maximizes the cumulative reward over time.

The Q-learning algorithm is a type of RL algorithm that uses a Q-table to learn the optimal action-value function, which assigns a value to each state-action pair. The Q-table is initially filled with random values, and the agent updates the values based on the observed rewards and the estimated future rewards using the temporal difference equation. The agent selects actions based on the current state, using a strategy such as greedy or epsilon-greedy exploration. Over time, the Q-values converge to the optimal action-value function, and the agent's policy converges to the optimal policy.

1.1 Objective

The objective of the lab is to implement the Q-learning algorithm for solving the problem of moving a robot to a goal position using reinforcement learning techniques. The goal is to use the Q-learning algorithm to learn an optimal policy for the robot's movement, by updating the Q-table based on the observed rewards and estimated future rewards, and ultimately, to converge to the optimal action-value function and optimal policy that guides the robot to the goal position.

1.2 Problem Statement

The problem is to find the goal in a 2-dimensional environment that has a defined size and contains obstacles. The environment is finite with a size of 20x14, equating to 280 states. The robot can take 4 different actions: left, up, right, down and cannot move diagonally. As a result, the Q function will have a size of 280x4, or 1120 cells.

The robot's location can only be in cells that are not obstacles. The dynamics of the robot's movement is straightforward, it will move one cell per iteration in the direction of the selected action, unless there is an obstacle or a wall in front of it, in which case it will remain in the same position.

The reinforcement function is designed to encourage the robot to reach the goal as quickly as possible. It assigns a -1 reward for all cells except the goal cell, which has a reward of +1. The goal cell is located at coordinates (3,17).

1.3 Definition of Terms

1. **Agent**: an autonomous entity that perceives its environment, makes decisions, and takes actions based on those decisions. In reinforcement learning, an agent is the entity that learns to make decisions by interacting with an environment and receiving feedback in the form of rewards or penalties. The goal of the agent is to learn a policy, which maps states of the environment to actions, that maximizes the cumulative reward over time.

2. **Q-table**: One of the key components of the Q-learning algorithm is the Q-table, which is a matrix that stores the Q-values for each state-action pair. The Q-table is initially filled with random values, and the agent updates the values based on the observed rewards and the estimated future rewards using the temporal difference equation.

$$Q(s,a) = Q(s,a) + \alpha(r + \gamma \times max(Q'_a(s',a')) - Q(s,a)) \quad (1)$$

3. **States**: represent the different configurations of the environment that the agent can be in. In this case, the states are the different positions of the robot in the environment.

4. **Actions**: represent the different actions that the agent can take in a given state. In the case of the robot movement problem, the actions are $\uparrow$ (up), $\downarrow$ (down), $\leftarrow$ (left), and $\rightarrow$ (right).

5. **Rewards**: are a scalar value that the agent receives after taking an action in a state. In the case of the robot movement problem, the rewards are -1 for all cells except the goal cell, which has a reward of +1.

6. **Q-Values**: The Q-values are the values assigned to each state-action pair in the Q-table. They represent the expected cumulative reward of taking a certain action in a certain state and following the policy thereafter. Q-values are updated using the temporal difference equation, which takes into account the observed reward, the maximum estimated future reward, and a learning rate.

7. **Policy**: is a mapping of states to actions. It defines the behavior of an agent, and it tells the agent what action to take in a given state.

8. **Optimal Policy**: The optimal policy is the policy that maximizes the expected cumulative reward over time. The optimal policy is the best solution to the problem of maximizing the agent's long-term reward.

9. **Epsilon-Greedy Policy**: An epsilon-greedy policy is a common exploration strategy used in reinforcement learning. It is a trade-off between exploration and exploitation, where the agent balances between taking the action that it believes will maximize the reward (exploitation) and taking random actions to explore the environment (exploration).

10. **Exploration**: Exploration refers to the process of trying out new actions in order to learn more about the environment and find better actions. This is important in Q-learning because the agent needs to explore the state space in order to learn the optimal Q-values. Exploration can be achieved by using a random action selection strategy or by using an exploration-exploitation strategy such as epsilon-greedy.

11. **Exploitation**: Exploitation, refers to the process of taking actions that are known to be good based on the current knowledge of the agent. This is important in Q-learning because the agent needs to use the knowledge it has learned in order to achieve a high reward. Exploitation can be achieved by using a greedy action selection strategy or by using an exploration-exploitation strategy such as epsilon-greedy.

2 Methodology

The Q-learning algorithm implemented in this lab uses an exploration strategy to balance between exploiting the current knowledge of the Q-values and exploring new actions. The strategy used is called epsilon-greedy, in which the agent selects the action with the highest Q-value with probability (1- ϵ) and selects a random action with probability epsilon. This way the agent will mostly select the action with the highest Q-value but still have a chance of exploring new actions.

2.1 Algorithm Implementation

In order to implement the Q-learning algorithm, the following psuedocode was followed

1 Initialize Q(s,a) to "0" n episodes Initialize s randomly in any free cell
2 For m iterations repeat
3 Choose a following -greedy policy
4 Take action a, observe r, s'
5 Q(s,a)←Q(s,a) + (r + · max$Q'_a(s', a') - Q(s, a)$)
6 s←s'
7 if the goal is achieved then finish the episode endFor endFor

The google colab file provided in the lab contains some already implemented functions. Those implemented by us are discussed below.

1. The ***get_ start()*** function that randomly initializes the agent's start position in the environment

```python
def get_start(self):
  # start the agent in a random position within the map
  start_x = np.random.randint(0, self.map.shape[0])
  start_y = np.random.randint(0, self.map.shape[1])
  while self.map[start_x, start_y] == 1:
    start_x = np.random.randint(0, self.map.shape[0])
    start_y = np.random.randint(0, self.map.shape[1])
  return [start_x, start_y]

```

   The random position is chosen within the environment size and checked to validate its a free-space in the map since the agent can never be at an obstacle position.

2. The ***step()*** function that accepts the agent's action and returns its next state, reward, and status of the goal position. While taking an action to move to the next state, we always check if the next state is free before making that position the agent's new position else the agent remains in its current position. The reinforcement function rewards -1 for free spaces and +1 for the goal position thus guiding the agent to move toward the goal position while avoiding any unnecessary actions. If the next state is the goal, the agent receives a reward of +1 and the is_goal_reached is set to true, which means the episode has ended. If the next state is not the goal, the agent receives a reward of -1 and the is_goal_reached is set to false. The current state is set to the next state and the function returns the next state, the reward, and

is_goal_reached flag.

```python
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 = up, 1 = down, 2 = left, 3 = right
    #applies the action taken and returns the next

    if action == 0:
      next_state = [(self.current_state[0] - 1), self.current_state[1]]
      if self.map[next_state[0], next_state[1]] == 1:
        next_state = self.current_state

    elif action == 1:
      next_state = [(self.current_state[0] + 1), self.current_state[1]]
      if self.map[next_state[0],next_state[1]] == 1:
        next_state = self.current_state

    elif action == 2:
      next_state = [self.current_state[0], (self.current_state[1] - 1)]
      if self.map[next_state[0],next_state[1]] == 1:
        next_state = self.current_state

    elif action == 3:
      next_state = [self.current_state[0], (self.current_state[1] + 1)]
      if self.map[next_state[0], next_state[1]] == 1:
        next_state = self.current_state

    # assigns the reward of the agent's new state
    if next_state == self.goal:
      reward = 1
      is_goal_reached = True
    else:
      reward = -1
      is_goal_reached = False

    # update the current state
    self.current_state = next_state

    return next_state, reward, is_goal_reached
```

It's important to note that the step function is a key part of the environment and it defines the dynamics of the problem.

3. The ***epsilon-greedy()*** function. The epsilon parameter is the probability of selecting a random action. The function first generates a random number between 0 and 1 using the np.random.random() function. If this number is less than epsilon, the function selects a random action from the four possible actions using the random.choice() function. Otherwise, the function selects the action with the highest Q-value using the np.argmax() method. The function returns the action to take.

```
def epsilon_greedy_policy(self, s, epsilon):
  # Epsilon greedy policy (choose a random action with probability epsilon)
  # choose a random value between 0 and 1
  x = np.random.random()
  if x > epsilon:
    # choose the action with the most promising reward from the q_value table
    action = np.argmax(self.Q[s[0],s[1]])
  else:
    # choose a random action
    action = random.choice([0,1,2,3])
  return action
```

4. The ***episode()*** function. The episode function starts by resetting the environment and initializing the current state, the is_goal_reached flag, and total_reward count. The agent then enters a loop that continues until either the goal is reached, or the number of iterations has been exhausted which means the episode has ended.

   In each iteration, the agent selects an action using the epsilon-greedy policy, takes the action and observe the next state, reward, and done flag. Then, the agent updates the Q-values using the observed state, action, reward and next state.

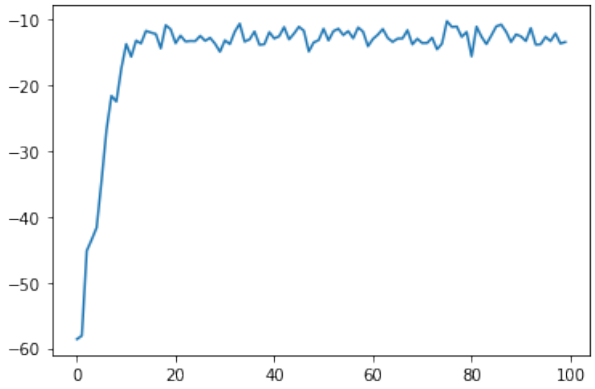
   After that, the agent sets the current state to the next state and increments the total_reward count. The process repeats until the episode is over.

   It's important to note that the episode function is the core of the Q-learning algorithm and it's where the agent interacts with the environment, learns from its experiences, and updates its Q-values. The episode function also allows the agent to explore the state space and find the optimal policy.

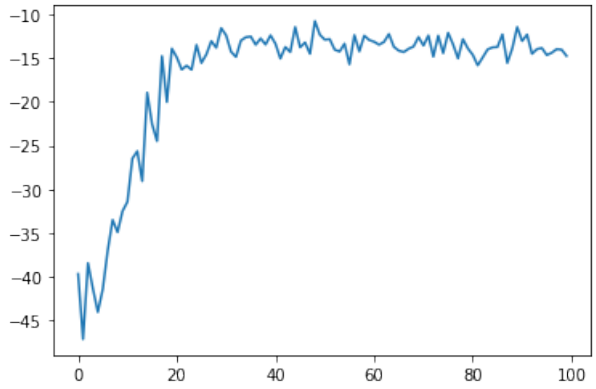
```
    def episode(self, alpha, epsilon):
      # Episode execution for n_iterations. Generate an action with
      epsilon_greedy_policy,

      #get a start position for the first episode and reset it for each episode
      using the
      state = self.env.reset()
      is_goal_reached = False
      total_reward = 0

      for i in range(self.n_iterations):

```

```
            # use the epsilon-greedy policy to choose an action
            action = self.epsilon_greedy_policy(state, epsilon)
            # print(action)

            # use the chosen action to move to the next state
            next_state, reward, is_goal_reached = self.env.step(action)
            self.env.current_state = next_state

            self.Q[state[0], state[1],action] = self.Q[state[0], state[1], action] +
            self.alpha * (reward + self.gamma * np.max(self.Q[next_state[0],
            next_state[1], :]) - self.Q[state[0], state[1],action])
            total_reward += reward
            state = next_state

            if is_goal_reached == True:
              return total_reward

        return total_reward
```

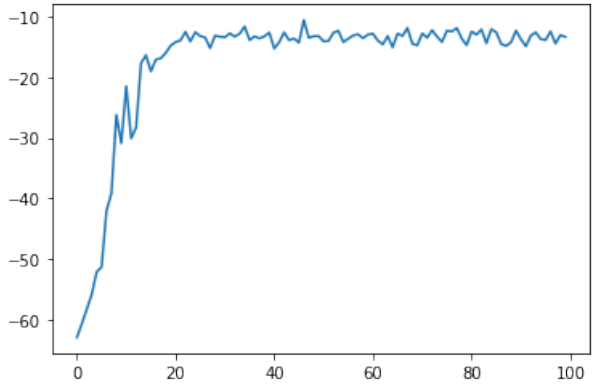
3 Results



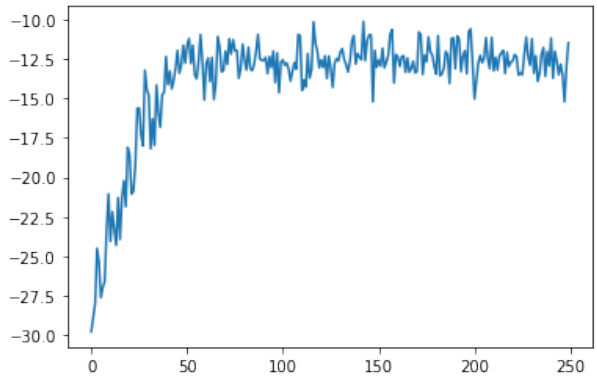
(a) $\alpha = 0.1, \gamma = 0.9, \epsilon = 0.7, n_e = 20000, n_i = 70$



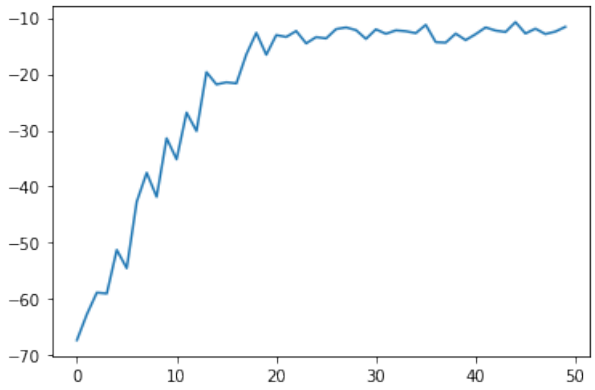
(b) $\alpha = 0.1, \gamma = 0.5, \epsilon = 0.3, n_e = 20000, n_i = 50$



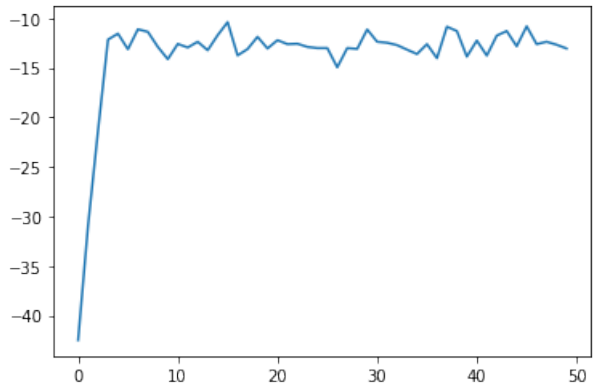
(c) $\alpha = 0.1, \gamma = 0.5, \epsilon = 0.3, n_e = 20000, n_i = 70$



(d) $\alpha = 0.1, \gamma = 0.3, \epsilon = 0.1, n_e = 50000, n_i = 30$



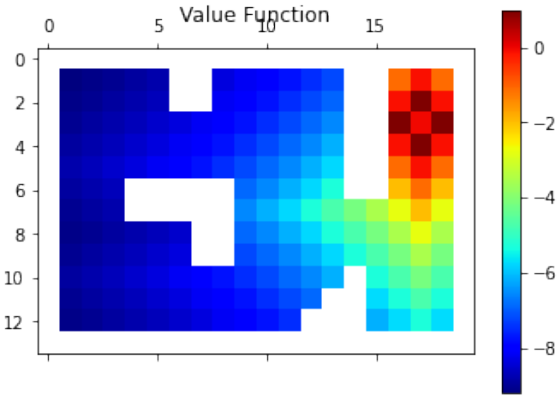
(e) $\alpha = 0.1, \gamma = 0.3, \epsilon = 0.3, n_e = 10000, n_i = 70$



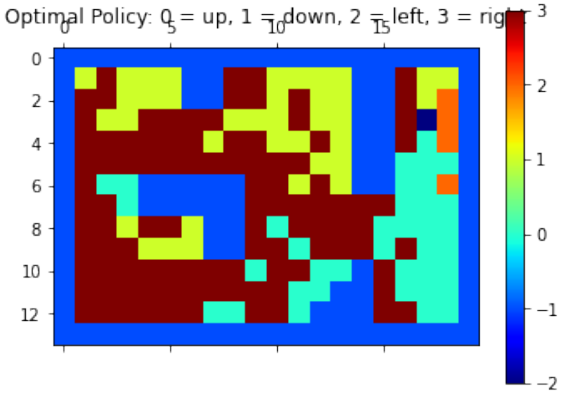
(f) $\alpha = 0.8, \gamma = 0.3, \epsilon = 0.3, n_e = 10000, n_i = 50$

Figure 1: Rewards obtained using different hyperparameters

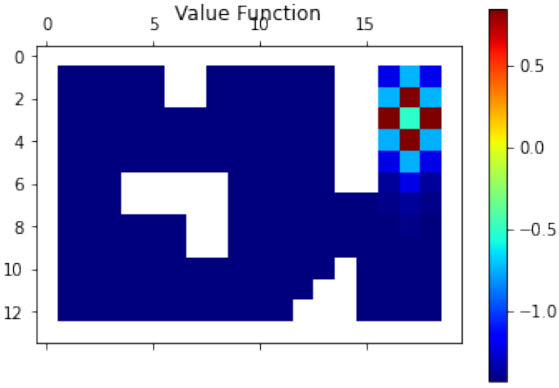
The graphs presented above represent the results obtained using different values of learning rate α, discount factor γ, exploration rate ϵ, no. of episodes n_e and, no. of iterations n_i. These results would be discussed in the next section.



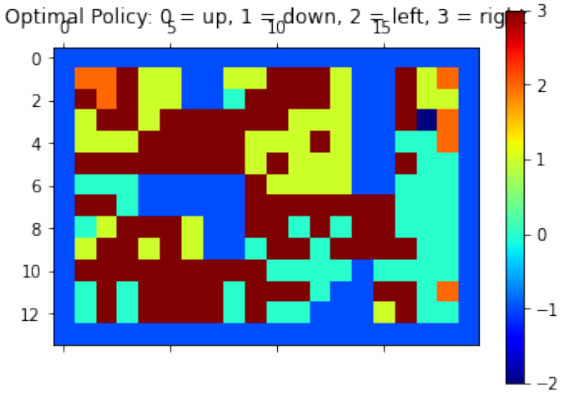
(a) Value function for graph (a) fig.1



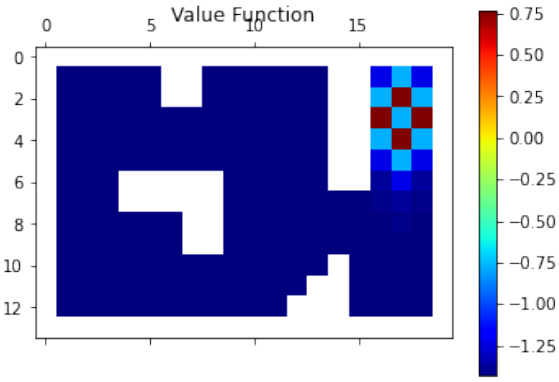
(b) Optimal policy for graph (a) fig.1



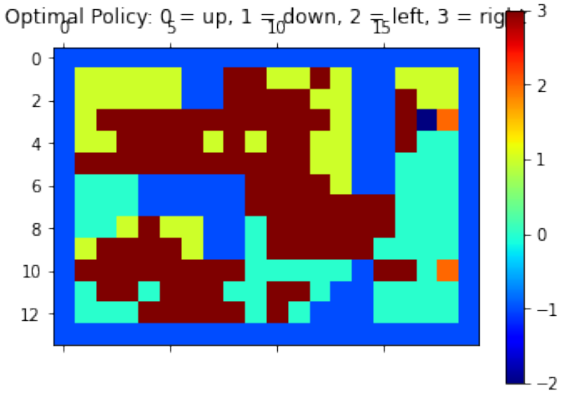
(c) Value function for graph (e) fig.1



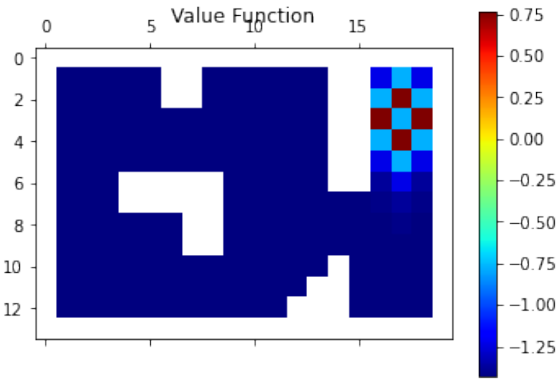
(d) Optimal policy for graph (e) fig.1



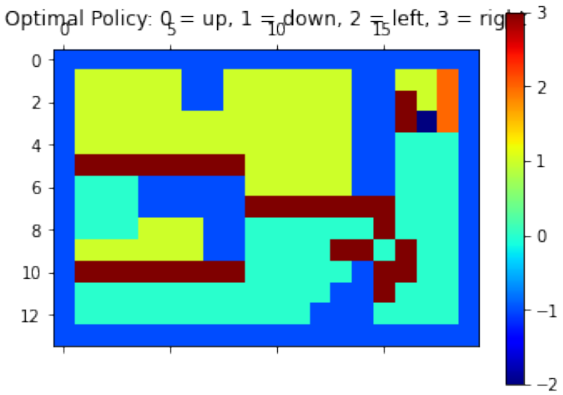
(e) Value function for graph (d) fig.1



(f) Optimal policy for graph (d) fig.1



(g) Value function for graph (f) fig.1



(h) Optimal policy for graph (f) fig.1

Figure 2: Value function and Optimal Policy

In fig.3 we show what the robot has learned to do using the optimal policy from the hyperparameters in fig.1 (a)

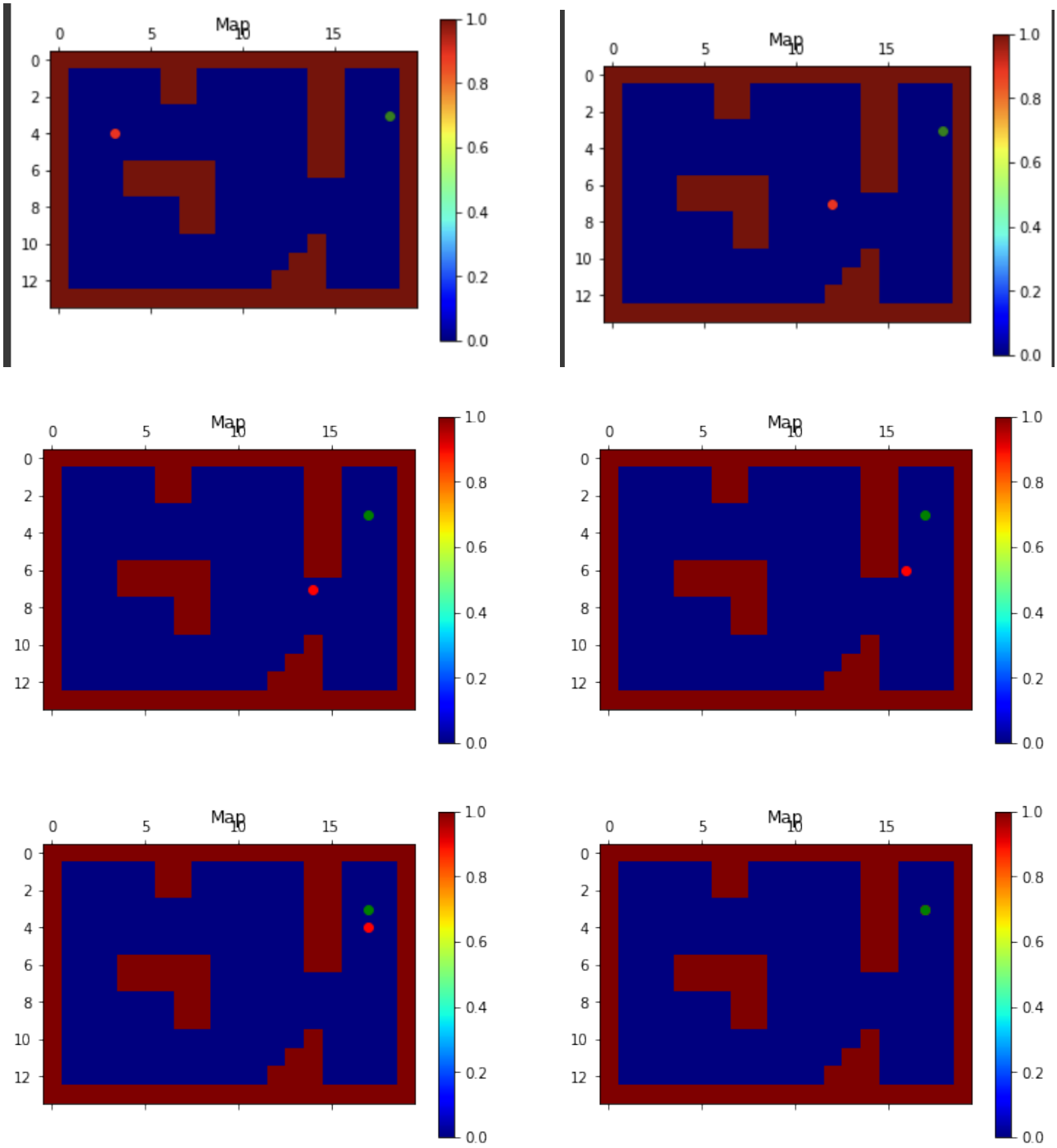


Figure 3: Robot moving from start to goal using the optimal policy in fig.2 (b)

4 Discussion

4.1 How the parameters affect the training

- The learning rate α determines the step size at which the algorithm updates the estimates of the Q-values. From the results obtained, a smaller learning rate means that the algorithm makes smaller updates, while a larger learning rate means that the algorithm makes larger updates.
- The discount factor γ determines the importance of future rewards in the Q-value estimates. A smaller discount factor makes the algorithm place less importance on future rewards, while a larger discount factor means that the algorithm places more importance on future rewards.
- The exploration rate ϵ determines the probability of the algorithm taking a random action instead of the action with the highest estimated Q-value. A higher exploration rate means that the algorithm is more likely to explore and try new actions, while a lower exploration rate means that the algorithm is more likely to exploit and take actions with known high Q-values.

After trying out several values for these parameters, we realized a small learning rate, high discount factor, and high exploration rate will make the algorithm converge to the optimal Q-values more slowly but with more certainty. A high learning rate, low discount factor, and low exploration rate will make the algorithm converge to the optimal Q-values more quickly but with less certainty. The images presented in fig.1 represent the graphs obtained while using different parameters. For the first graph in fig.1 (a), we see that a small learning rate, high gamma, high epsilon, high number of episodes, and high number of iterations in training the Q-learning algorithm will increase the chances of the agent to converge to the optimal solution but also it will increase the training time.
In fig.1 (d) we see that if you use a small learning rate, the agent's updates to its Q-values will be small, which can make the learning process slower. A small gamma value indicates that the agent places less emphasis on future rewards and more emphasis on immediate rewards, which can result in a suboptimal policy. A small epsilon value means that the agent will explore less frequently, which can limit its ability to discover optimal actions in unseen states. A small number of episodes means that the agent will have less experience to learn from, which can also slow down the learning process. A high number of iterations in a 2D environment of size 20x14 means that the agent will have more opportunities to update its Q-values and learn from its experiences. However, since the other hyperparameters are also small, it may not be able to make significant improvements in its policy due to the slow learning rate and limited exploration.
Fig.1 (f): The parameters used causes the agent to converge quickly to a suboptimal solution because it does not have enough exploration and update of Q-value.

4.2 Observations while tuning parameters

1. **Increasing the number of iterations in an episode/ Increasing the number of episodes**:
    - Increase the chances of the agent exploring more of the state space, which can be beneficial if the agent needs to explore more states to find the optimal policy.
    - Increases the chances of the agent encountering rare or unlikely events, which can be beneficial if the agent needs to learn about these events to improve its performance.
    - Increase the amount of time needed to learn the optimal policy, as the agent needs more steps/iterations to converge to the optimal Q-values. Additionally, the agent needs to update the Q-values for all states

and actions encountered during the episode, and this can become computationally expensive if the episode is too long.

2. **Decreasing the number of iterations in an episode/ Decreasing the number of episodes**:
   - Decreases the chances of the agent exploring more of the state space, which can be detrimental if the agent needs to explore more states to find the optimal policy.
   - Decreases the chances of the agent encountering rare or unlikely events, which can be detrimental if the agent needs to learn about these events to improve its performance.
   - Reduces the amount of time needed to learn the optimal policy, as the agent needs fewer steps/iterations to converge to the optimal Q-values.
   - Increases the chances of the agent not reaching the convergence and not finding the optimal policy, as the agent has fewer opportunities to learn from different episodes.

To interpret the value functions in fig.2, the deep blue regions represent states that are farther from the obstacle while areas around the obstacle have lower values. For the optimal policy presented in fig.2 (b), we see that using the parameters stated in fig.1 (a), the optimal policy obtained contains more right action as the robot moves to the goal position. While fig.2 (h) shows that with the parameters in fig.1 (f), the optimal policy we'll obtain contains more up and down actions and some right actions. Overall, the hyperparameter for fig.1 (a) seems to be the best choice for this environment since we have a 2D environment of size 20x14, there are a total of 280 states, which means that the agent has many possible states to explore, so it will require more episodes and iterations to find the optimal policy. The images in fig.3 show what the robot has learned to do using the optimal policy obtained from the hyperparameters in fig.1 (a).

5 Conclusion

In conclusion, the implementation of the Q-learning algorithm in python for a robot in a 2D environment has shown that it is a powerful tool for enabling robots to learn optimal policies for navigation. The robot was able to learn to navigate to a goal state while avoiding obstacles through the use of a Q-table that stored the rewards associated with different actions in different states. The chosen exploration strategy, such as epsilon-greedy, was able to balance between exploiting the current knowledge of the Q-values and exploring new actions. The challenge encountered while implementing the algorithm was in parameter selection. Since Q-learning can be sensitive to the choice of hyperparameters such as the learning rate and exploration rate. But after several tuning and iterations, the parameters in fig.1 (a) were chosen. Overall, the lab was very interesting.