CS 451/551 - Assignment 2

December 12, 2024

1 Introduction

In the third assignment, you must implement a reinforcement learning algorithm, Q-Learning, to develop an intelligent agent navigating through Grid-World and find the optimal path. The Grid-World is a world consisting of four types of nodes: "Flat", "Mountain", "Goal" and "Pitfall". The agent aims to navigate on the Grid-World and reach a "Goal" node while maximizing the total reward/score. It is worth noting that more than one goal node can exist in some scenarios. The agent must visit one of the goal nodes when there are multiple goals. Besides, the agent can move in four directions: 'UP', 'DOWN', 'LEFT', and 'RIGHT'. Additionally, the transition from one node to another has a reward based on the node type.

We provide a code base in Python; you should use it to implement the algorithm. The algorithm should be able to find the optimal list of actions that lead to the maximum total reward. Additionally, you will be required to compare the performance of the algorithm in various scenarios. You can use the grid generator in the code base to create various scenarios. The performance metrics you should consider include the optimality gap, computational time, the number of iterations to converge, the convergence speed, and so on.

The deadline for this homework is **7 January 2025**.

2 Environment&Implementation

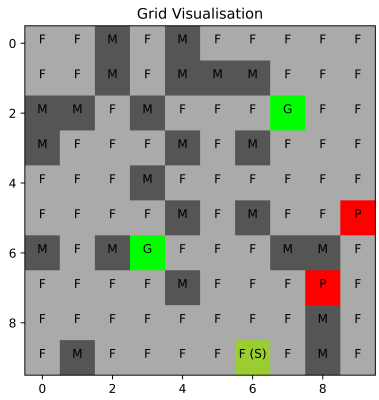
This section describes the Grid-World with transition rewards and the rules in detail. Besides, it explains the provided code base.

2.1 Grid-World

Grid-World is a square map with dimensions $n \times n$ containing n^2 nodes, as illustrated in Figure 1. The agent aims to reach the goal nodes by riding a bike and moving in the directions of 'UP', 'DOWN', 'LEFT', and 'RIGHT'. The Grid-World consists of four types of nodes, each with its own characteristics:

- **Flat**: It is easy to ride a bike.

Figure 1: Demonstration of Example Grid-World



- **Mountain**: It is hard to ride a bike.
- **Goal**: This node is the target destination for the agent.
- **Pitfall**: If the agent enters this node, it will fail.

One of the nodes in the map is designated as the starting point for the agent. Moving from one node to another provides a reward based on the type of the nodes, as shown in Table 1. The reinforcement learning (RL) algorithm aims to find the best sequence of actions to maximize the total reward.

From	To	Transition Reward
Flat	Flat	-1
Flat	Mountain	-3
Mountain	Mountain	-2
Mountain	Flat	-1
(Any)	Goal	100
	Pitfall	-100

Table 1: Transition Reward Function between two Node Types

Note: If the agent tries moving out of a Grid-World, it stays on the same node and gets a -1 reward.

For this assignment, we provide both single-goal and multiple-goal scenarios. In both cases, the agent must visit one of goal nodes without falling pitfalls by minimizing the total traveling cost. Note that the ultimate aim is maximizing the total collected score.

2.2 Implementation

This text presents the instructions for the programming assignment. The assignment involves implementing the Q-Learning algorithm in Python for a grid-world environment. The students are expected to complete the corresponding methods in the "QLearning.py" file in the given code base. The code base is implemented in Python, and Jupyter is not allowed. Students who need to learn how to code with Python can learn from available sources on the internet. The "Environment.py" file contains the necessary information about the grid world and provides moving on it and the transition reward. The "RLAgent.py" file contains an abstract class for Reinforcement Learning (i.e., RL) agents; the Q-Learning algorithm must be a sub-class. Finally, some helpful links to learn Python programming language from scratch are shared at the end of the document (Section 6).

- **Environment.py**: The "Environment" class in this file holds the necessary information about the Grid-World and provides moving on it and the transition reward. It takes the file path where the scenario data exists. The scenario data file is in Pickle format. Some essential methods in this class are listed:
  - **reset**(): The agent goes to a predefined starting point/node, then the method returns the node index of the starting point.
  - **to_node_index**(*position*): This method converts a given position (i.e., row and column indices as a list) to a node index.
  - **to_position**(*node_index*): This method converts a node index to a position.
  - **set_current_position**(*node_index*): It changes the agent's current position to the given node index.
  - **get_node_type**(*position*): It returns the node type of the given position as a string. This string is the node type name's first (upper) character.
  - **get_reward**(*previous_pos*, *next_pos*): This method provides the transition reward from the previous position (*previous_pos*) to the next position (*new_pos*) based on Table 1.
  - **is_done**(*position*): It states whether the given position ends the episode based on the current environmental state. In other words, it determines whether any termination condition is met (i.e., reaching a "Goal" node or entering a "Pitfall" node).
  - **move**(*action*): It moves the agent from the current node based on the given action. The action can be "UP" (0), "LEFT" (1), "DOWN" (2) or "RIGHT" (3). Besides, it returns the next node index, the transition reward, and whether the episode is done.
  - **get_goals**(): It finds the state indices of all goals and then returns them as a list.

- **rl_agents/RLAgent.py**: "RLAgent" class in this file is an abstract class. The Q-Learning algorithm must be a sub-class of it.
  - **Constructor**(*env*, *discount_rate*, *action_size*): This constructor takes some necessary parameters and "Environment" object.
  - **train**(***kwargs*): This abstract method trains the RL agent with the corresponding RL algorithm.
  - **act**(*state*, *is_training*): This abstract method decides on an action that will be taken based on the given "state". Note that the action decision can differ depending on whether it is called during training or validation.
  - **validate**(): This method plays on the provided environment and returns a list of actions decided by the trained policy and the total collected reward. Note that this method should be called after training.
- **rl_agents/QLearning.py**: "QLearningAgent" class in this file must contain the implementation of the Q-Learning algorithm for this assignment. Also, it is the sub-class of the "RLAgent" class. This means that you must write your whole Q-Learning approach inside of the corresponding methods (i.e., **train** and **act**). Note that you can initiate some parameters/variables in the constructor.
- **Main.py**: This file has a script to run the algorithm and print some results, such as the total scores, elapsed time in microseconds, etc. You are free to edit this file if you want to add more analyses or RL algorithms. Although we will test your code with the original script, you can share your own "Main.py" file with us. To run this script, you should enter the name of the grid file, such as "Grid1.pkl" in the console. Besides, you must define the parameters you determine in this file in the constructor of the corresponding RL agent classes.
- **grid_generator.py**: This file contains a script to generate a new scenario randomly. We provided this file so that you can test your algorithm with more scenarios. When you run this script, it will ask for some parameters in the console.

Briefly, you will implement the "Q-Learning" RL algorithm with Python programming language in the corresponding methods. Do not forget to read the comments in all methods.

3 Reinforcement Learning Algorithm

This section provides essential background information on Reinforcement Learning (RL) and the algorithm you will be expected to implement for the given

problem. RL is a machine learning category where an agent learns to make decisions in an environment by interacting with it. The agent observes the current state and selects an action, and then the environment responds with feedback called "reward" (as shown in Equation 1). The primary objective of RL is to maximize the cumulative reward (as illustrated in Equation 2).

$$s_{t+1}, r_t \leftarrow \text{Env}(s_t, a_t) \tag{1}$$

$$\textbf{maximize}(\sum_{t}^{T} r_t) \tag{2}$$

Reinforcement learning enables an agent to learn how to identify the best actions to take in different states, leading to more successful outcomes in the environment. The Q-value is a critical concept in RL, which indicates how good a particular state-action pair is. By using a Dynamic Programming approach, the Q-value function ($Q(s, a)$) can be defined as shown in Equation 3 (Bellman *Optimality* Equation), where γ is the discount rate.

$$\begin{aligned} Q(s_t, a_t) &= r_t + \gamma \times r_{t+1} + \gamma^2 \times r_{t+2} + \gamma^3 \times r_{t+3} + ... \\ &= r_t + \gamma \times \textbf{max}_{a_{t+1}}(Q(s_{t+1}, a_{t+1})) \end{aligned} \tag{3}$$

$$\pi(s_t) = \textbf{argmax}_{a_t}(Q(s_t, a_t)) \tag{4}$$

The Q-value function represents the expected cumulative discounted reward the agent can receive by taking a specific action in a particular state and following its policy afterward. Note that the policy function can be defined as displayed in Equation 4. The Q-value function is learned through the Q-learning process, where the Q-value estimates are updated based on the rewards the agent receives in response to its actions. By continually updating the Q-value estimates, the agent can make more informed decisions about which actions to take in different states, ultimately leading to more successful environmental outcomes.

Q-learning (Algorithm 1) is an off-policy algorithm that learns the optimal Q-value function by updating the Q-value of the current state-action pair using the maximum Q-value of the next state. The agent selects actions based on the highest Q-value for the current state, which may not necessarily be the action taken by the agent. This makes Q-learning an off-policy algorithm, as the agent updates its Q-values based on the optimal action, not the action it took.

On the other hand, in Q-learning, the agent's behavior policy is typically an epsilon-greedy approach during training. This means that the agent selects the best action with probability $1 - \epsilon$ (exploitation) and a random action with probability ϵ (exploration), as defined in Equation 5. The epsilon-greedy approach is crucial because it allows the agent to explore and learn from different actions. Without exploration, the agent may not encounter all possible

Algorithm 1 Q-Learning (Off-Policy)

procedure Q-Learning(Env, γ, max_iter, α, ϵ_{min}, ϵ_{decay})
    Initialize $\hat{Q}[s, a]$ as a zero-matrix
    iter $\leftarrow 0$
    $\epsilon \leftarrow 1$
    while iter $\leq$ max_iter **do**
        $s \leftarrow$ Initial state
        repeat
            $a \leftarrow \epsilon$-greedy(s)
            $s', \text{reward} \leftarrow \text{Env}(s, a)$
            $TD \leftarrow \text{reward} + \gamma \times \max_{a'}(\hat{Q}(s', a')) - \hat{Q}(s, a)$
            $\hat{Q}(s, a) \leftarrow \hat{Q}(s, a) + \alpha \times TD$ ▷ Soft update
            $s \leftarrow s'$
            if $\epsilon > \epsilon_{\text{min}}$ **then** ▷ Update ϵ
                $\epsilon \leftarrow \epsilon \times \epsilon_{\text{decay}}$
            end if
            iter $\leftarrow$ iter $+ 1$
        until Episode is done
    end while
end procedure

state-action pairs and could miss out on learning optimal policies. However, exploration also comes at a cost since the agent may choose sub-optimal actions that lead to lower rewards. By balancing exploration and exploitation through the epsilon-greedy approach, the agent can learn an optimal policy while still exploring the environment. The value of ϵ typically decreases over time as the agent learns more about the environment. In the beginning, when the agent has little knowledge of the environment, it is essential to explore more to learn optimal policies. As the agent's knowledge grows, the exploitation term becomes more important, and the exploration term can be decreased. This approach is known as annealing epsilon-greedy, where the value of epsilon is annealed or gradually reduced over time.

$$a = \begin{cases} \pi(s) & \text{with probability } 1 - \epsilon \\ \text{random action} & \text{with probability } \epsilon \end{cases} \tag{5}$$

For this assignment, the Q-Learning algorithm must be implemented; its performance must be compared in terms of computational time and total reward during the validation phase. Additionally, its performance depends on some hyper-parameters (e.g., maximum iteration, γ, ϵ_{min}, ϵ_{decay}, and $alpha$). Thus, you are expected to tune these parameters and analyze the impact on the performance of the method in

4 Requirements

The requirements of this homework are listed below:

1. You have to implementthe Q-Learning algorithm to find the maximum path to reach the "Goal" nodes in a given scenario.
2. Your algorithms must return the list of actions and the total score. They should find the optimal/sub-optima solution, the maximum total score from the starting point to the "Goal" nodes.
3. For this assignment, both single and multiple-goal examples are provided. The agent must maximize the total reward by visiting one of "Goal" nodes. You may focus on only single-goal scenarios if you struggle with multiple-goal scenarios. If you implement your RL approaches for only single-goal approaches, you can achieve a maximum of half of the implementation part. However, you can get full credit from the report part.
4. You must fill in the corresponding methods (i.e., **train** and **act**) in "QLearning.py". Please, do not change the other files (except "Main.py").
5. You must use Python programming language with the 3.10 version. Jupyter is not allowed. You are free to choose any IDE for implementation, but PyCharm is recommended. You can create a student account with your "@ozu.edu.tr" e-mail address for an educational (free) license.
6. No library except *NumPy*, *Matplotlib*, and visualization libraries (e.g., Seaborn & plotly, etc.) is allowed. Using other libraries will be penalized.
7. You must also write a detailed and well-organized report. Also, your report must be clear and in English.
8. The report must cover the implementation details and design and the comparison of the algorithms. You need to indicate your "state" representation in the report.
9. You must compare the performance of the algorithms in terms of the computational time, the total score, the output of the algorithms, the convergence speed, the optimality gap, etc. You are free to extend these analyzes. It would be best to put some graphs and tables in your report for better evaluation, as in Section 5.
10. You must also analyze the effect of the hyper-parameters (e.g., maximum iteration, γ, ϵ_{min}, ϵ_{decay} and α) on the performance of the algorithms. You need to indicate the determined parameters in a table in your report.
11. After parameter tuning, do not forget to define the determined parameters in the "Main.py" file.
12. You can also implement and analyze different stopping conditions in the literature.

13. We provide 8 different scenarios (4 for single-goal and 4 for multiple-goal) for you. Please, do not forget to test/evaluate your code with these scenarios. With "grid_generator.py", you can create more various scenarios for your evaluation. We have more scenarios we did not share to test your algorithms.

14. You can also implement other RL algorithms for better performance evaluation, such as Double Q-Learning, SARSA, Value Iteration, etc. If you implement more RL algorithms, you can also compare them in your report.

15. Do not put any screenshots and snippets of the code or console. Because, we will already examine your code and run it to test. Instead, you can provide flowcharts, pseudo-codes, detail exhalations, graph, and so on.

16. You will submit your homework as *zip* format. Other formats, such as *rar*, will not be accepted. The *zip* file must contain your report as a PDF file and your implementation as **.py* file format. The *zip* file should contain "QLearning.py", "Main.py" (with the parameters) and "report.pdf". Do not forget to write your name in your Python files.

17. File name format is **NAME_SURNAME_STUDENTID_hw2.zip**.

18. Any compiler or run-time error will be penalized **(-20pt)**.

19. Any plagiarism will also be penalized (e.g., Sharing code, copy code from the Internet, asking someone to do your homework, using any language model such as **ChatGPT**, etc.).

20. **Grading**:

    - Implementation **(40pt)**:
      - Implementation of Q-Learning **(40pt)**: Points will be awarded based on the correctness and efficiency of the implementation.
    - Report **(60pt)**:
      - Implementation details (12.5pt)
      - Analyzes the effect of hyper-parameters (12.5pt)
      - Analyzes and visualization of trained Q-Tables (12.5pt)
      - Graphs & Demonstration (12.5pt)
      - Overall (10pt)
    - Plagiarism Penalty: Any plagiarism will result in a direct deduction of **-100pt** from this assignment.

21. The assignment must be done individually. While you may discuss the homework with your peers, you are strictly prohibited from sharing any ideas or code. Failure to comply with this rule will result in a penalty for plagiarism.

22. The deadline for this homework is **7 January 2025**. Late submissions will not be allowed according to the course policy.

If you have any questions regarding the homework, we encourage you to seek help during our office hours. Our office hours are held online via Zoom to ensure accessibility for all students, and the corresponding meeting links can be found on LMS. Additionally, if you prefer to communicate via e-mail, you can contact Anil Dogru. Please keep in mind that we cannot provide any extra code or ideas before the deadline, as this would be considered a violation of academic integrity. However, we will do our best to assist you with any conceptual or technical questions.

5 Example Graphs

This section provides some example graphs to get some idea of what kind of graphs are expected in your report. Also, it would be best if you extended these graphs with more analysis and comparisons.

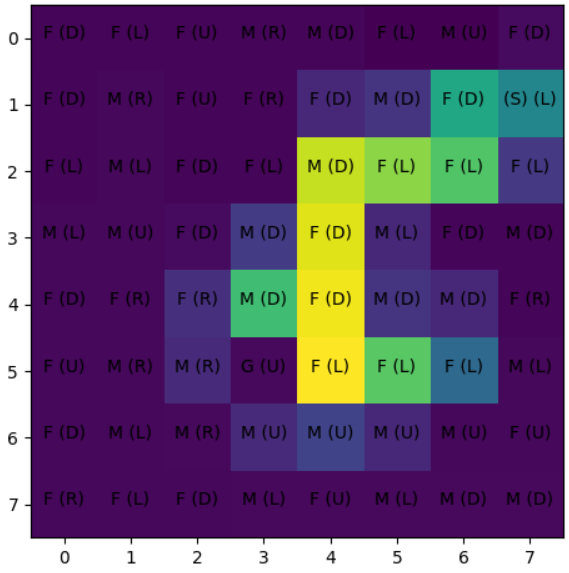


Figure 2: Max. Q-Value for each State with Optimal Action

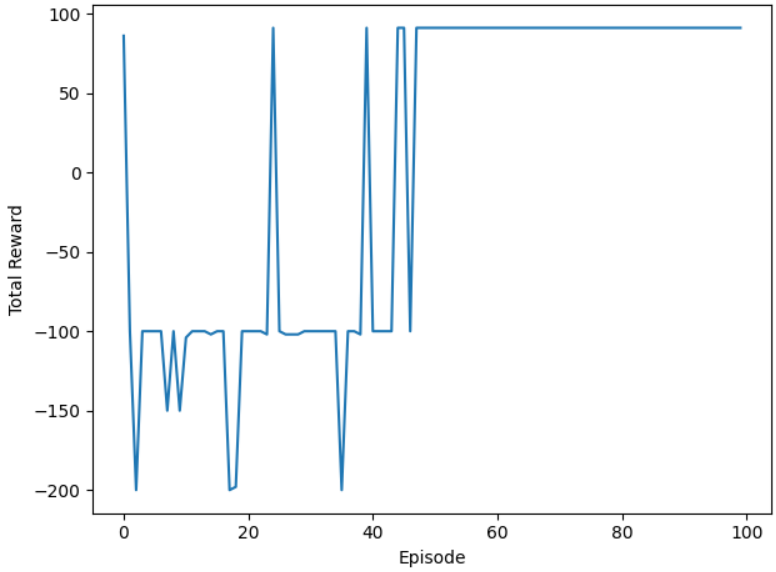


Figure 3: Total Reward Change over Episode

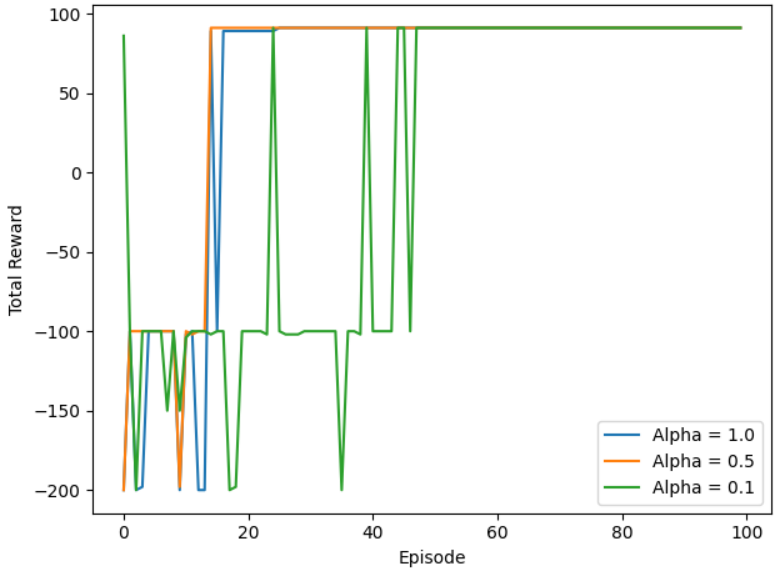


Figure 4: Effect of Alpha Hyper-Parameter on Total Reward

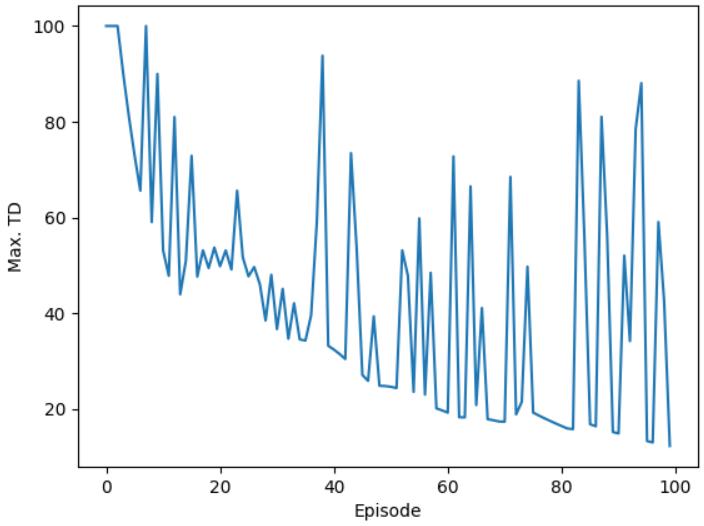


Figure 5: Max. TD on each Episode

6 Helpful Links for Python

You can learn the required Python programming with the helpful links below:

- Python Docs
- Tutorials Point
- Geeks for Geeks
- W3 Schools
- Programiz
- Learn Python