University of Dhaka

Department of Computer Science and Engineering

CSE 4116: Artificial Intelligence Lab

Assignment - 3

Submitted By:

Syed Mumtahin Mahmud, Roll: 50

Submitted On :

May 3, 2024

Submitted To :

Dr. Md. Mosaddek Khan

Contents

1 OpenAI Gym

1.1 Introduction

OpenAI Gym is an essential toolkit that plays a critical role in the field of artificial intelligence, specifically in the domain of reinforcement learning. The principal purpose of this system is to assist in the development and assessment of reinforcement learning algorithms. Fundamentally, OpenAI Gym provides an extensive collection of environments that have been carefully crafted to facilitate the exhaustive testing and validation of said algorithms. The simulation environments provided can vary in complexity from basic grid-based scenarios to advanced, real-life simulations, accommodating a wide range of research and development requirements.

In the domain of reinforcement learning, OpenAI Gym offers numerous benefits. Both developers and researchers utilize its capabilities to assess the reliability and effectiveness of their algorithms, thus cultivating an environment that promotes innovation and ongoing progress in the discipline. The environments' standardization guarantees an equitable setting for comparative analysis, allowing stakeholders to extract valuable insights regarding the merits and drawbacks of different methodologies.

A fundamental comprehension of the utility of OpenAI Gym environments requires familiarity with the concept. These simulated environments contain simulated tasks or scenarios in which agents—usually algorithms or learning agents—engage with their circumstances in order to accomplish predetermined goals through iterative learning and adaptation. In essence, every environment is characterized by a unique arrangement of states, actions, and rewards, which establish the regulations and dynamics that govern the simulated undertaking. For example, within a basic grid world environment, discrete locations on the grid may represent states, actions would be denoted by "up," "down," "left," and "right," and rewards would be contingent on achieving predetermined objectives or avoiding obstacles.

The profound variety of environments that OpenAI Gym provides further emphasizes its adaptability. Embedded in robotics simulations, classic control problems, immersive Atari games, and various other domains, every environment poses a distinct collection of complexities and difficulties. OpenAI Gym fosters an environment that encourages exploration and experimentation by providing developers and researchers with unrestricted access to a wide range of testing environments. In this environment, innovative algorithms are subjected to thorough scrutiny and refinement in a multitude of contexts.

Fundamentally, OpenAI Gym functions as an essential collaborator for the reinforcement learning community, facilitating widespread availability of state-of-the-art environments and tools while stimulating advancements and novelty in the domain. The enduring impact of this organization is not limited to the software it offers; rather, it cultivates a dynamic ecosystem that is distinguished by cooperation, innovation, and an unwavering quest for excellence.

1.2 Code Analysis

Below is the basic code of OpenAI Gym to run an environment.

```
import gymnasium as gym

env=gym.make("FrozenLake-v1", render_mode='Human' ,is_slippery='
    True')
env.reset()
# render the environment
env.render()
```

1.2.1 make()

OpenAI Gym allows users to create environments by using the gym.make() function, which requires providing the environment name and version to load the correct configuration. Users can also customize the visualization settings using the "render_mode" parameter, which can be graphical, text-based, or suppressed. Some environments in OpenAI Gym incorporate stochastic elements, introducing randomness into the agent's interactions. To enable or disable this, the "is_slippery" parameter can be used. Setting it to True introduces slippery dynamics, while setting it to False ensures deterministic behavior. By invoking the gym.make() function and providing necessary parameters, users can customize their environment setup to suit their experimentation and visualization needs effectively.

```
import gymnasium as gym

# Create an instance of the CartPole environment
("FrozenLake-v1", render_mode='Human' ,is_slippery='True')
```

1.2.2 reset()

This function resets the environment to its initial state and returns the initial observation. When the reset() function is called, the player, or agent, is effectively returned to the starting position within the environment. This action resets the environment's state, reverting it back to its initial configuration as defined by the environment's parameters.

```
# Reset the environment to its initial state
observation = env.reset()
```

1.2.3 render()

The render() function in OpenAI Gym is used to visualize the current state of the environment. It provides a graphical or textual representation of the environment, allowing

users to observe the agent's interactions in real-time.Additionally, rendering the environment may introduce overhead, impacting the performance of the reinforcement learning algorithm, especially in complex or computationally intensive environments.

When called, the render() function typically opens a window or displays output in the console, depending on the rendering mode specified during environment creation. This visualization can be invaluable for debugging purposes, as it provides insights into the agent's behavior and the dynamics of the environment.

```
# actions: left -0, down - 1, right - 2, up- 3
env.action_space
returnValue = env.step(1)
# format of returnValue is (observation, reward, terminated,
   truncated, info)
# observation (object)  - observed state
# reward (float)        - reward that is the result of taking the
    action
# terminated (bool)     - is it a terminal state
# truncated (bool)      - it is not important in our case
# info (dictionary)     - in our case transition probability


env.render()
```

1.2.4 step()

The step function takes the action the player wants and returns the value (observation, reward, terminated, truncated, info). In our case, we obtained (0, 0.0, False, False, 'prob': 0.3333333333333333). This implies that the final state is 0, the reward is 0, "False" means that the final state is not terminal (we did not arrive at the hole or the goal state), and another Boolean return value, "False," is not important for this discussion, and 'prob': 0.3333333333333333 is the transition probability from our initial state 0 to state 0. Though we chose the down step, the player did not step down; it stayed on the initial step. This happened because of transitional probability. We can check the transitional probabilities using env.P[state][action]. env.P is a dictionary containing all the transition probabilities. The following code lines illustrate transition probabilities:

```
#transition probabilities
#p(s'|s,a) probability of going to state s'
# Starting from the state, s and by applying the action, a
# env.P[state][action]
env.P[0][1] #state 0, action 1
# Output is a list having the following entries:
# (transition probability, next state, reward,  as MDPterminal
   state?)
```

The result is:

$$\begin{bmatrix} 0.3333333333333333 & 5 & 0.0 & \text{True} \\ 0.3333333333333333 & 10 & 0.0 & \text{False} \\ 0.3333333333333333 & 7 & 0.0 & \text{True} \end{bmatrix}$$

The step() function is arguably the most critical method when interacting with an environment. It allows the agent to take an action and observe the result. The method returns a tuple containing the following: **observation :** The new state of the environment after the action. **reward:** The reward received for taking the action. **done:** A boolean indicating whether the episode has ended (e.g., the agent reached a terminal state). **info:** A dictionary with auxiliary diagnostic information (not to be used for learning).

```
# Take an action in the environment and observe the next state
action = 1  # Example action (e.g., move right)
next_observation, reward, done, info = env.step(action)
```

2 Frozen Lake Environment as MDP

The Frozen Lake environment is a toy text problem. Frozen lake involves crossing a frozen lake from start to goal without falling into any holes by walking over the frozen lake. The player may not always move in the intended direction due to the slippery nature of the frozen lake.
The game commences with the player starting at [0,0] in a grid world with the goal at a distant point like [3,3] in a 4x4 setting. Holes in the ice are fixed in certain spots for predetermined maps or scattered randomly for random maps. Progression involves the player moving towards the goal, avoiding holes. The lake's slippery nature allows occasional perpendicular movements. Randomly generated maps will always offer a clear path to the goal.
MDPs are the foundation of model-free and model-based reinforcement learning (RL). Both model-free and model-based RL approaches strive to address MDP-type issues. These settings involve discrete time steps for agents to interact with the world, get rewards, and transition between states. Both forms of RL algorithms rely on the essential features of an MDP, which include states, actions, transition functions, and rewards.

2.1 States

In an MDP, a state represents a situation or a configuration of the environment at a particular time. In the FrozenLake-v1 environment, the states are represented by the position of the player on a grid. The grid is composed of different types of tiles:

Frozen (F): Safe tiles where the player can step.

Hole (H): If the player steps here, they fall into the hole and the episode ends.

Start (S): The starting point of the player.

Goal (G): The target tile for the player to reach.

Of these 4 types of fields, the starting and frozen ones are non-terminal states and the goal and hole are terminal states. The game ends when the player reaches any of the

terminal states. The state is a discrete representation of the player's location on this grid. For instance, if the grid is a 4x4 environment, there are 16 possible states (S0 to S15).

2.1.1 observation_space

```
env.observation_space
```

This line of code is used to show the total number of states in the environment.

2.2 Actions

Actions are the set of moves that the agent can make at any given state. The frozen lake environment has 4 discrete actions indicating which direction to move the player. The actions are typically defined as:

Left: Move one tile to the left.

Down: Move one tile downwards.

Right: Move one tile to the right.

Up: Move one tile upwards.

2.2.1 action_space

```
env.acttion_space
```

The action space of an environment consists of the actions a player can take to reach its goal.

2.3 Rewards

Rewards are given to the agent after it takes an action and transitions to a new state. In FrozenLake-v1, the reward structure is quite simple:

- Reach goal: +1
- Reach hole: 0
- Reach frozen: 0

```
returnValue = env.step(1)
# format of returnValue is (observation, reward, terminated,
   truncated, info)
```

Here, the second value it returns is the reward for that function. For the frozen lake environment, the reward is 0 for every step except the one that reaches the goal state. The reward for reaching the goal is 1.

2.4 Transitions

Transitions in an MDP define the probability of moving from one state to another, given an action. In the case of FrozenLake-v1, the transition probabilities can be affected by the "slippery" nature of the environment:

When the ice is slippery, the action taken by the agent does not always go as planned. For example, if the agent decides to move left, there is a chance it could actually move up, down, or stay in the same place. The transition model would then define the probability distribution over the resulting states for each action. In a deterministic version of the game, the transition probabilities are 1 for the intended move and 0 for all other moves. In a stochastic version, the probabilities might be spread out among different outcomes.

Let us now see the transition probability with an example,

```
env.P[6][1]
```

The result is:

$$\begin{bmatrix} 0.3333333333333333 & 5 & 0.0 & \text{True} \\ 0.3333333333333333 & 10 & 0.0 & \text{False} \\ 0.3333333333333333 & 7 & 0.0 & \text{True} \end{bmatrix}$$

The transition probability from state 6 and under action 1 (DOWN) to state 5 is 1/3, the obtained reward is 0, and state 5 (final state) is a terminal state.
The transition probability from state 6 and under action 1 (DOWN) to state 10 is 1/3, the obtained reward is 0, and state 10 (final state) is not a terminal state.
The transition probability from state 6 and under action 1 (DOWN) to state 7 is 1/3, the obtained reward is 0, and state 7 (final state) is a terminal state.

3 Value Iteration

The policy derived from the Value Iteration algorithm demonstrates robust performance in the FrozenLake-v1 environment. Through iterative refinement, the algorithm converges to an optimal policy that guides the agent's actions towards achieving the designated goal while avoiding perilous tiles. Despite the stochastic nature of the environment, the agent exhibits competence in reaching the goal state, underscoring the effectiveness of the derived policy.

In the context of the FrozenLake-v1 environment, the Value Iteration algorithm operates as follows:

- **Initialization**:
  - The algorithm initializes the value function for each state arbitrarily. This can be done with zeros or random values.
- **Value Iteration**:
  - The algorithm iteratively updates the value function for each state using the Bellman optimality equation.
  - It computes the value of each state by considering the immediate reward obtained from taking an action and the discounted value of the resulting state.
  - This process continues until the values converge, indicating that the optimal value function has been found.
- **Convergence**:
  - The algorithm iterates until the value function converges, meaning that the values of all states stabilize and no longer change significantly between iterations.
  - At this point, the algorithm has found the optimal policy, and the agent can use it to navigate the environment.

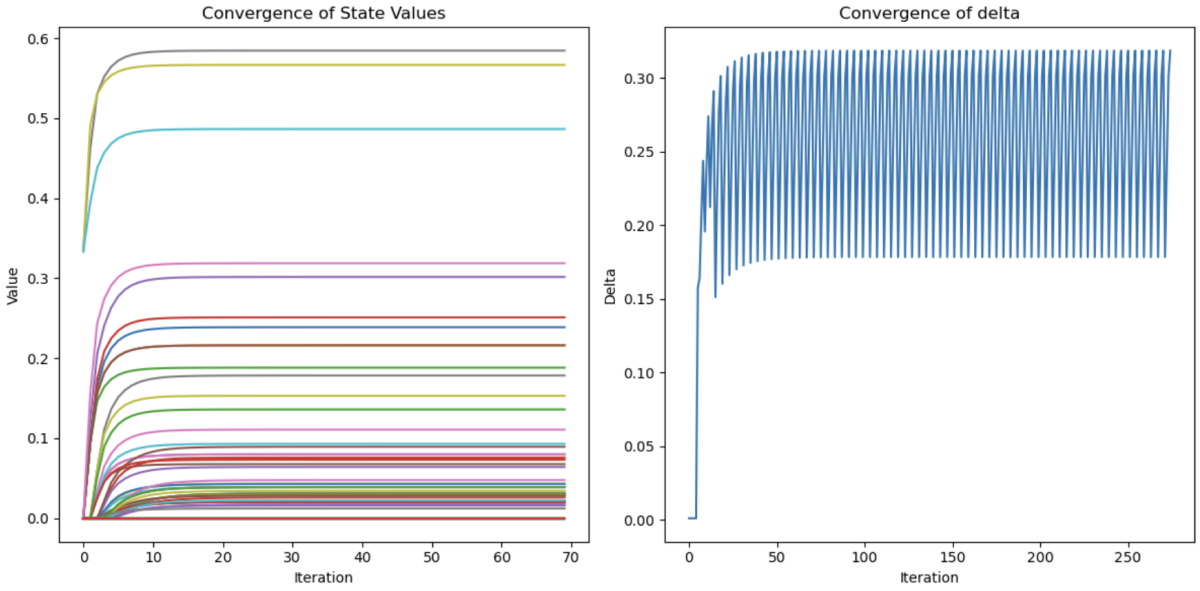

Figure 1: Convergence of the state value function.

- **Policy Execution**:
  - Once the algorithm converges, the agent follows the optimal policy derived from the value function to make decisions in the environment.
  - It selects actions according to the policy and observes the resulting rewards and state transitions.
- **Evaluation**:

- Finally, the performance of the policy is evaluated by running the agent in the environment and measuring its success rate, average reward, or other relevant metrics.
- This evaluation provides insights into how well the learned policy performs in practice.

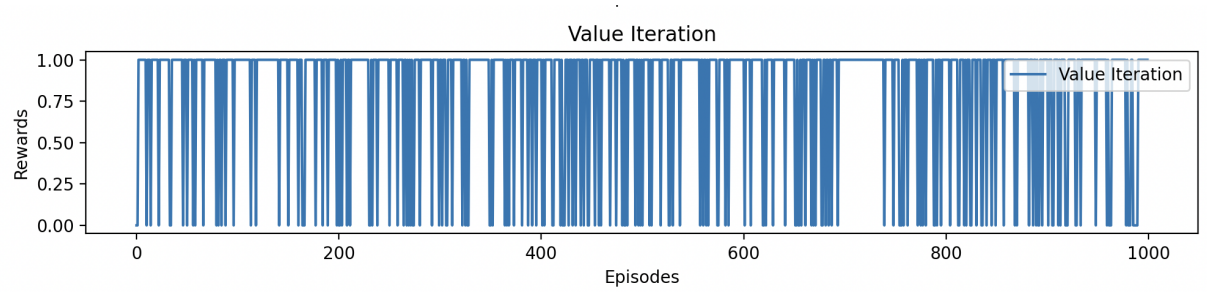

Figure 2: Reward vs Episodes for Value Iteration.

4 Q-Learning

The policy derived from the Q-learning algorithm demonstrates robust performance in the FrozenLake-v1 environment. Through a process of exploration and exploitation, the algorithm gradually refines its strategy to converge to an optimal policy. This policy effectively guides the agent's actions towards achieving the designated goal while avoiding perilous tiles. Despite the stochastic nature of the environment and the inherent uncertainty in action selection, the agent demonstrates competence in reaching the goal state. This highlights the effectiveness of the learned policy and the adaptability of the Q-learning algorithm in navigating challenging environments.

In the context of the FrozenLake-v1 environment, the Q-Learning algorithm operates as follows:

- **Initialization**:
  - Initialize the Q-table with arbitrary values, typically zeros or small random numbers, representing the expected future rewards for each state-action pair.
- **Exploration-Exploitation Trade-off**:
  - At each time step, decide whether to explore new actions (exploration) or exploit the current knowledge (exploitation) to select actions.
  - Exploration ensures that the agent discovers new strategies and avoids getting stuck in suboptimal policies, while exploitation maximizes rewards based on current knowledge.
- **Action Selection**:

- Select an action based on an epsilon-greedy policy, where with probability ϵ a random action is chosen, and with probability $1-\epsilon$ the action with the highest Q-value is selected.
- This balance between exploration and exploitation allows the agent to gradually shift from exploration to exploitation as it learns more about the environment.

- **State Transition and Reward**:
  - Execute the selected action in the environment and observe the resulting state and reward.
  - Update the Q-value of the previous state-action pair using the observed reward and the maximum Q-value of the next state.

- **Policy Extraction**:
  - Once the Q-values have converged, extract the optimal policy by selecting the action with the highest Q-value for each state.
  - This policy guides the agent's actions in the environment, ensuring that it takes the most rewarding actions at each step.

- **Evaluation**:
  - Evaluate the performance of the learned policy by running the agent in the environment and measuring its success rate, average reward, or other relevant metrics.
  - This evaluation provides insights into how well the learned policy performs in practice and allows for further refinement if needed.

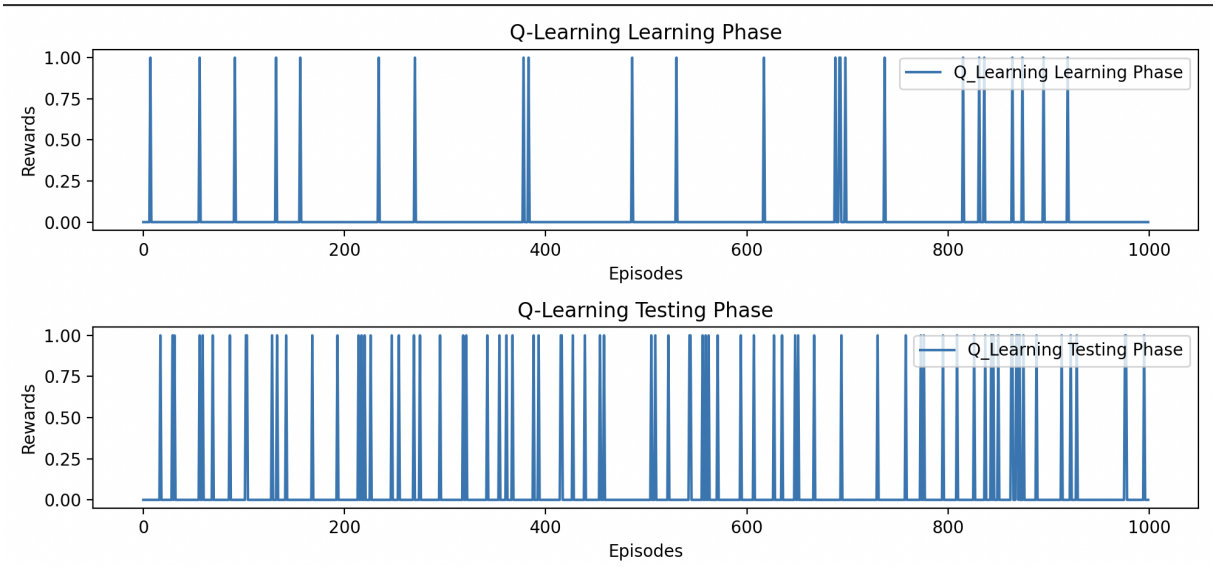

Figure 3: Reward vs Episodes for Value Q-Learning test and train.

5 Comparison of Results

A comparative evaluation was conducted to assess the performance of the learned policies derived from Q-learning and Value Iteration in the FrozenLake-v1 environment.

- **Q-learning:**
  - Average Reward: 0.124
- **Value Iteration:**
  - Average Reward: 0.777

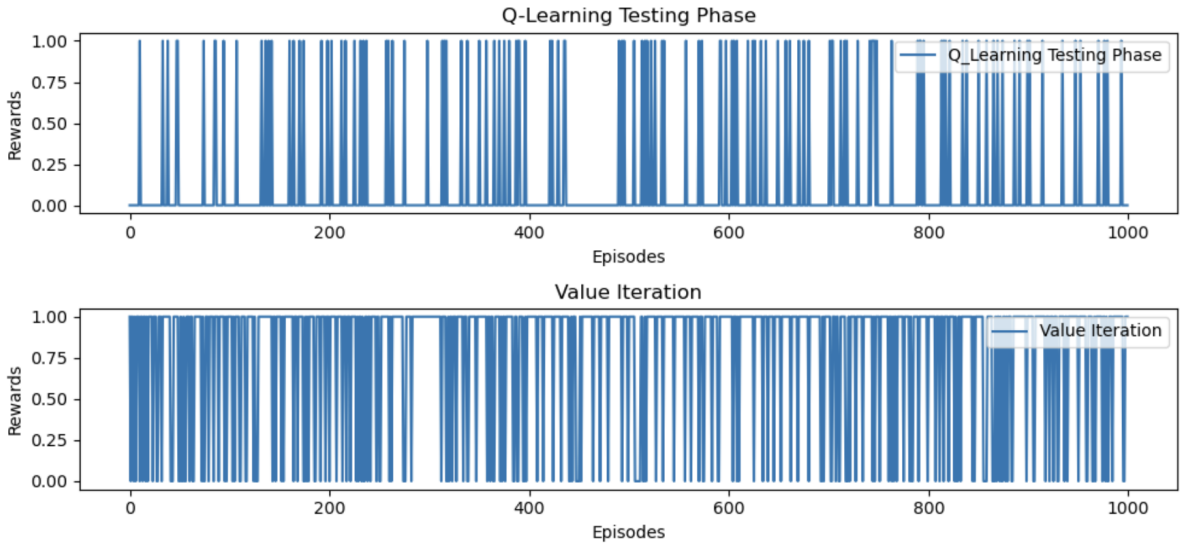

Figure 4: Result comparison between Q-Learning and Value Iteration

The results indicate that the policy derived from Value Iteration outperforms the policy learned through Q-learning, achieving a significantly higher average reward. This suggests that Value Iteration may have converged to a more optimal policy in this scenario. Further analysis and experimentation may be warranted to explore the factors contributing to the observed differences in performance.