Familiarization with OpenAI Gym for Reinforcement Learning

1. Introduction

This assignment is designed to introduce the core concepts of Reinforcement Learning (RL) using the OpenAI Gym library (now Gymnasium). The FrozenLake-v1 environment is used as a basic example to help you understand how agents interact with the environment, how actions are chosen, and how rewards are obtained.

2. Setup

Before we start with the code, we need to install the necessary libraries. Run the following commands to install the required packages.

```
# Install Gymnasium (formerly OpenAI Gym)
!pip install gymnasium

# Install numpy (if not installed already)
!pip install numpy
```

```
Collecting gymnasium
  Downloading gymnasium-1.0.0-py3-none-any.whl.metadata (9.5 kB)
Requirement already satisfied: numpy>=1.21.0 in /usr/local/lib/python3.10/dist-packag
Requirement already satisfied: cloudpickle>=1.2.0 in /usr/local/lib/python3.10/dist-p
Requirement already satisfied: typing-extensions>=4.3.0 in /usr/local/lib/python3.10/
Collecting farama-notifications>=0.0.1 (from gymnasium)
  Downloading Farama_Notifications-0.0.4-py3-none-any.whl.metadata (558 bytes)
Downloading gymnasium-1.0.0-py3-none-any.whl (958 kB)
   ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 958.1/958.1 kB 11.9 MB/s eta 0:00:00
Downloading Farama_Notifications-0.0.4-py3-none-any.whl (2.5 kB)
Installing collected packages: farama-notifications, gymnasium
Successfully installed farama-notifications-0.0.4 gymnasium-1.0.0
Requirement already satisfied: numpy in /usr/local/lib/python3.10/dist-packages (1.26
```

3. Key Concepts in Reinforcement Learning

Reinforcement Learning (RL) is a type of machine learning where an agent learns to take actions in an environment to maximize cumulative reward. The agent interacts with the environment through a cycle of observations, actions, and rewards.

- **Agent**: The learner or decision-maker (e.g., our policy that decides actions).
- **Environment**: The world the agent interacts with (e.g., FrozenLake).

- **State**: A specific situation in which the agent finds itself (e.g., a location on the grid).
- **Action**: A move the agent can take (e.g., moving up, down, left, right).
- **Reward**: A signal received after taking an action, indicating success/failure.

---

4. Environment Setup: FrozenLake

FrozenLake-v1 is a simple environment where the agent must navigate a slippery grid world to reach a goal without falling into holes.

- **Map**: A 4x4 grid where the agent starts in the top-left and the goal is in the bottom-right.
- **Objective**: Reach the goal (denoted as "G") while avoiding holes (denoted as "H").
- **Action Space**: The agent can move:
    - `0`: Left
    - `1`: Down
    - `2`: Right
    - `3`: Up

---

5. Code Implementation

Let's dive into the actual code to demonstrate the interaction between the agent and the environment.

```
import gymnasium as gym
import time
```

Step 2: Creating the FrozenLake Environment

FrozenLake is a grid-based environment where an agent starts at one corner and tries to reach the goal, avoiding "holes" in the ice. The map can be **deterministic** (where actions lead to expected results) or **stochastic** (where actions sometimes fail).

We will create a **deterministic** environment (*is_slippery=False*) to better understand how the agent behaves.

```
# Create the FrozenLake environment with a deterministic setup (no slipping)
env = gym.make('FrozenLake-v1', map_name="4x4", render_mode='human', is_slippery=False)
```

```
/usr/local/lib/python3.10/dist-packages/ipykernel/ipkernel.py:283: DeprecationWarning
  and should_run_async(code)
```

The deterministic policy is a straightforward decision-making strategy for the agent operating within the environment. Under this policy, the agent first checks if it can move to the right. If this is possible, the agent will execute the action to move right. If moving right is not an option, the agent will then choose to move down instead.

Arguments:

- **state (int or tuple)**: This represents the current state of the environment, which indicates the agent's position on the grid.

Returns:

- **action (int)**: This indicates the action the agent will take based on its current state. The action is represented as an integer, where:
    - 0: corresponds to moving left,
    - 1: corresponds to moving down,
    - 2: corresponds to moving right,
    - 3: corresponds to moving up.

```
# Define a deterministic policy: Always move right if possible, otherwise move down
def deterministic_policy(state):

    # Handle state tuple (in case reset() returns a tuple)
    if isinstance(state, tuple):
        state = state[0]

    # Determine the action based on the current state
    if state % 4 < 3:  # Can move right unless in the last column
        return 2  # Move right
    elif state < 12:  # Can move down unless in the last row
        return 1  # Move down
    else:
        return 0  # Default action, move left (to keep the agent moving)



# Simulate episodes (runs through the environment)
num_episodes = 10  # Number of episodes to run

for episode in range(num_episodes):
    # Start a new episode
    state, info = env.reset()  # Reset environment to start position
    done = False  # To track when an episode is complete
    total_reward = 0  # Total accumulated reward
    step_count = 0  # Number of steps taken in the episode

    while not done:
        # Introduce a slight delay to make the simulation easier to follow visually
        time.sleep(0.5)

        # Render the environment (visual display in a separate window)
        env.render()

        # Take action according to the deterministic policy
        action = deterministic_policy(state)

        # Execute the action in the environment and observe the results
        state, reward, done, truncated, info = env.step(action)

        # Track the number of steps and total reward
        step_count += 1
        total_reward += reward

        # If the episode ends (either reaching the goal or falling in a hole)
        if done:
            print(f"Episode {episode + 1} finished after {step_count} steps with reward {
            break

# Close the environment once all episodes are done
env.close()
```

```
Episode 1 finished after 4 steps with reward 0.0

Episode 2 finished after 4 steps with reward 0.0

Episode 3 finished after 4 steps with reward 0.0

Episode 4 finished after 4 steps with reward 0.0

Episode 5 finished after 4 steps with reward 0.0

Episode 6 finished after 4 steps with reward 0.0

Episode 7 finished after 4 steps with reward 0.0

Episode 8 finished after 4 steps with reward 0.0

Episode 9 finished after 4 steps with reward 0.0

Episode 10 finished after 4 steps with reward 0.0
```

6. Explanation of Code

6.1 Environment Creation

```
env = gym.make('FrozenLake-v1', map_name="4x4", render_mode='human', is_slippery=False)
```

- **env**: The environment instance created using Gymnasium.
- **map_name="4x4"**: Specifies the map size (4x4 grid).
- **render_mode='human'**: Enables graphical visualization.
- **is_slippery=False**: Makes the environment deterministic (no slipping).

This block of code initializes the FrozenLake environment, specifying the grid size and rendering mode. A deterministic setting is chosen to simplify the initial learning process.

6.2 Deterministic Policy

```
def deterministic_policy(state):
    if isinstance(state, tuple):
        state = state[0]

    if state % 4 < 3:  # Can move right
        return 2  # Right
    elif state < 12:  # Can move down
        return 1  # Down
    else:
        return 0  # Left
```

- This policy moves the agent **right** whenever possible and **down** otherwise. It ensures the agent attempts to reach the goal in a predictable manner.

The policy is designed to make decisions based on the agent's current state, guiding it to make optimal moves toward the goal while minimizing unnecessary left moves.

6.3 Episode Simulation

```
for episode in range(num_episodes):
    state, info = env.reset()  # Reset environment for each episode
    done = False
    total_reward = 0
    step_count = 0

    while not done:
        time.sleep(0.5)  # Optional: slows down rendering to make it visible
        env.render()  # Visual display of the environment

        action = deterministic_policy(state)  # Get action from policy
        state, reward, done, truncated, info = env.step(action)  # Take action

        step_count += 1
        total_reward += reward

        if done:
            print(f"Episode {episode + 1} finished after {step_count} steps with reward {
            break
```

```
Episode 1 finished after 4 steps with reward 0.0

Episode 2 finished after 4 steps with reward 0.0

Episode 3 finished after 4 steps with reward 0.0
Episode 4 finished after 4 steps with reward 0.0

Episode 5 finished after 4 steps with reward 0.0

Episode 6 finished after 4 steps with reward 0.0

Episode 7 finished after 4 steps with reward 0.0

Episode 8 finished after 4 steps with reward 0.0

Episode 9 finished after 4 steps with reward 0.0

Episode 10 finished after 4 steps with reward 0.0
```

- The agent interacts with the environment over multiple **episodes**.
- **env.reset()**: Resets the environment to the starting state for each episode.
- **env.step(action)**: The agent takes an action and receives feedback (next state, reward, and whether the episode is done).
- The episode finishes either when the agent reaches the goal or falls into a hole.

This section simulates the agent navigating the FrozenLake environment for a predefined number of episodes, collecting rewards and tracking performance in each run.

---

7. Key Concepts in Reinforcement Learning

- **Agent**: The policy determining the actions.
- **Environment**: The grid in FrozenLake.
- **State**: The agent's position in the grid.
- **Action**: The movement direction (left, right, up, down).
- **Reward**: The reward the agent gets upon completing an episode.

In this example:

- The **policy** is deterministic and simplistic (moving right or down).
- **Rewards** are sparse, as the agent only gets a positive reward when it reaches the goal.

---

8. Conclusion

This assignment introduces us to the basic principles of **Reinforcement Learning** and the **OpenAI Gym** library (Gymnasium). By using a deterministic policy in the FrozenLake environment, we can observe how an agent interacts with its environment and understand core RL concepts like states, actions, rewards, and episodes.

---

9. Future Work

- Implementing a more complex learning algorithm, such as **Q-learning** or **Deep Q-Networks (DQN)**, can enhance the agent's ability to learn optimal policies over time.

- Exploring the **stochastic version** of FrozenLake by setting *is_slippery=True* would introduce randomness and make the problem more challenging.

Assignment: Familiarization with OpenAI Gym for Reinforcement Learning

Topic: CartPole Environment

Libraries and Dependencies

Importing necessary libraries:

- gym: For creating and managing RL environments.
- numpy: For numerical operations (not heavily used in this basic version).
- matplotlib.pyplot: For visualizing the environment states.
- IPython.display.clear_output: To clear previous output in Jupyter for smooth visualization.

```
import gym
import numpy as np
import matplotlib.pyplot as plt
from IPython.display import clear_output
```

1. Environment Creation

Create the CartPole environment using the gym library.

CartPole-v1 is a classic control problem where the agent must balance a pole on a cart. The environment consists of states, actions, and rewards.

Key Concepts:

- **State**: The current position and velocity of the cart and pole.
- **Action**: Either move left (0) or right (1).
- **Reward**: A positive reward is given for every timestep the pole remains upright.

```
# Create the CartPole environment
env = gym.make('CartPole-v1')
```

2. Visualization Function

Function Name: *visualize_cartpole*

Visualize the CartPole environment over a specified number of episodes.

Parameters:

- env: The CartPole environment instance.
- num_episodes: The number of episodes to run and visualize (default is 5).

2.1. Episode Initialization

- Reset the environment to start a new episode.
- The done flag will indicate when the episode has ended (e.g., when the pole falls).
- *total_reward* keeps track of the accumulated reward for the episode.

2.2. Episode Loop (Actions and Environment Updates)

Continue the episode until it's done.

2.3. Environment Rendering

Render the environment to get the current state image in 'rgb_array' mode.

2.4. Random Action Selection

- In this example, the agent selects a random action (either move left or right).
- In actual RL, the agent would use a policy to decide the best action.

2.5. **Step Function (Environment Transition)**

- The *env.step(action)* function performs the action and returns:
- *next_state*: The new state after the action.
- *reward*: Reward earned for this action.
- *done*: Whether the episode has ended (True if the pole has fallen).

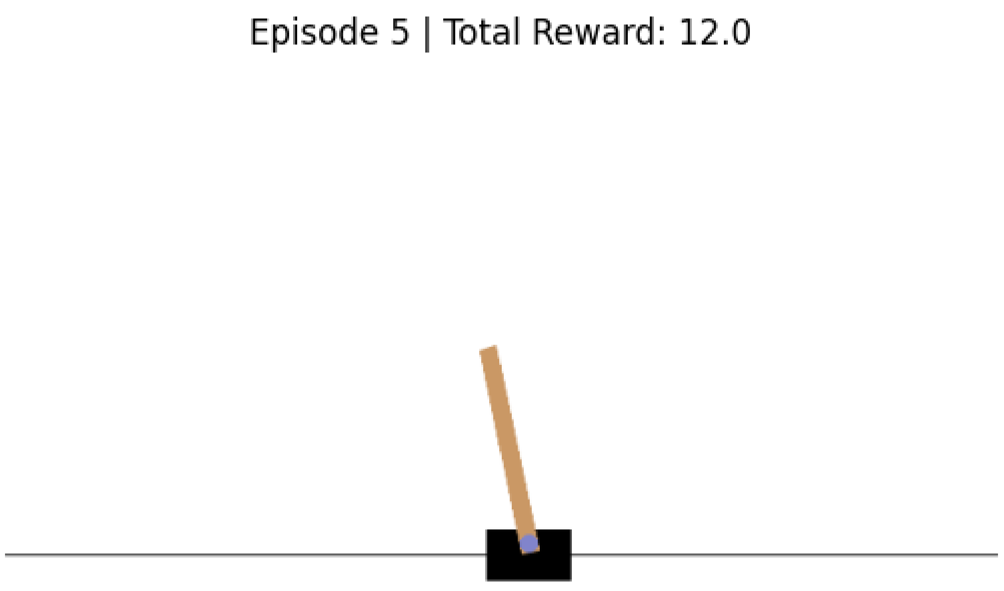
```
def visualize_cartpole(env, num_episodes=5):

    # List to store total rewards for each episode.
    total_rewards = []

    # Loop through each episode
    for episode in range(num_episodes):

        # 2.1. Episode Initialization
        state = env.reset()
        done = False
        total_reward = 0  # Total rewards accumulated in the episode

        print(f"Episode {episode + 1}:")  # Display the episode number

        # 2.2. Episode Loop (Actions and Environment Updates)
        while not done:
            # Clear previous output to make visualization smoother.
            clear_output(wait=True)
            # 2.3. Environment Rendering
            img = env.render(mode='rgb_array')

            # Display the current state of the CartPole environment.
            plt.imshow(img)
            plt.axis('off')  # Turn off the axis for better visuals
            plt.title(f'Episode {episode + 1} | Total Reward: {total_reward}')  # Display
            plt.show()

            # 2.4. Random Action Selection
            action = env.action_space.sample()

            # 2.5. Step Function (Environment Transition)
            next_state, reward, done, info = env.step(action)

            # Update the total reward for this episode.
            total_reward += reward

            # Set the current state to the next state for the next iteration.
            state = next_state

        # Store the total reward for the completed episode.
        total_rewards.append(total_reward)
        print(f"Total Reward: {total_reward}\n")  # Display total reward for the episode

    # 3. Environment Closure
    env.close()
```

4. Running the Visualization

Execute the *visualize_cartpole* function to simulate and visualize the CartPole environment.

This runs the simulation for 5 episodes by default.

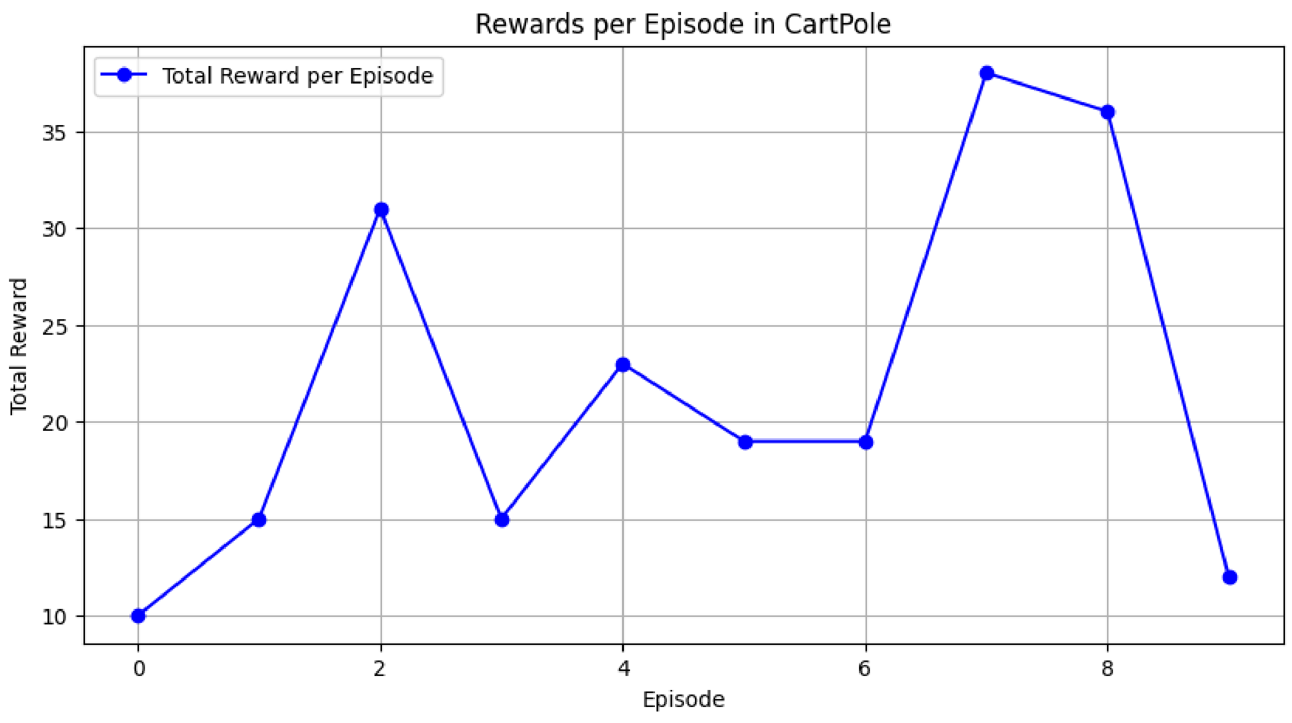
```
# 4. Running the Visualization
visualize_cartpole(env, num_episodes=5)
```



```
Total Reward: 13.0
```

```
import gym
import numpy as np
import matplotlib.pyplot as plt
from IPython.display import clear_output

# Create the CartPole environment
env = gym.make('CartPole-v1')

# Function to visualize the environment and plot rewards
def visualize_cartpole_with_rewards(env, num_episodes=10):
    total_rewards = []  # List to store total rewards for each episode

    for episode in range(num_episodes):
        state = env.reset()  # Reset the environment to start a new episode
        done = False  # Flag to check if the episode is done
        total_reward = 0  # Variable to accumulate rewards for the episode

        print(f"Episode {episode + 1}:")

        while not done:
            clear_output(wait=True)  # Clear previous output in the notebook

            # Render the environment in real-time (can be disabled for faster runs)
            env.render()

            # Choose a random action (either move left or right)
            action = env.action_space.sample()

            # Take the action and observe the next state and reward
            next_state, reward, done, info = env.step(action)
            # Update the total reward
            total_reward += reward

            # Update state for the next iteration
            state = next_state

        total_rewards.append(total_reward)  # Store the total reward for this episode
        print(f"Total Reward: {total_reward}\n")

    env.close()  # Close the environment when done

    # Plotting the rewards
    plt.figure(figsize=(10, 5))
    plt.plot(total_rewards, marker='o', linestyle='-', color='b', label="Total Reward per
    plt.title('Rewards per Episode in CartPole')
    plt.xlabel('Episode')
    plt.ylabel('Total Reward')
    plt.grid(True)
    plt.legend()
    plt.show()

# Run the interaction with the CartPole environment
visualize_cartpole_with_rewards(env, num_episodes=10)
```

```
Total Reward: 12.0
```



Start coding or generate with AI.