Group 18 - Pendulum Problem

Team Members:
B123040037 陳聖繹
B123040040 余家睿
B123040061 陳偉財

1. Introduction of the Problem

The objective of this project was to develop a Reinforcement Learning (RL) agent capable of solving the classic Inverted Pendulum control theory problem.

The Inverted Pendulum problem system consists of a pendulum that starts in a random position with one end attached to a fixed point, and the other one being free. The goal that the agent must achieve is to apply torque on the free end to swing it into a balanced upright position.

The problem presents specific challenges:

1. Continuous State Space

The pendulum's angle and angular velocity are continuous floating-point values, which cannot be directly mapped to a tabular Q-Learning approach without modification.

2. Gravity & Inertia

The agent must learn to overcome gravity to swing up, but also learn to decelerate to prevent overshooting the top position.

Our solution implements a Three-Layer Architecture that bridges the continuous physics simulation with a discrete tabular Q-Learning algorithm using Object-Oriented Programming principles.

2. Implementation Details

2.1 Agent Implementation

File: pendulum_agent.py, this layer acts as the bridge between the raw physics and our discrete learning algorithm, containing:

- **Discretization Logic:** Since Q-Learning requires a finite set of states, this class converts the continuous world into a grid. We implemented a resolution of 20 angle bins covering $[-\pi, \pi]$ and 20 velocity bins covering [-8, 8] rad/s, resulting in a total state space of 400 possible states.
- **Action Mapping:** The agent maps 5 discrete logical actions (STRONG_LEFT, WEAK_LEFT, NONE, WEAK_RIGHT, STRONG_RIGHT) to specific continuous torque values ranging from -2.0 to 2.0.
- **Visualization:** This layer manages the pygame rendering to visualize the pendulum's movement during evaluation.

2.2 Environment Implementation

File: pendulum_env.pym this layer wraps our custom agent to make it compatible with the standard Gymnasium API, containing:
- **Standardization:** It inherits from gym.Env and implements the required reset, step, and render methods.
- **Space Definition:** It formally defines the observation space as a MultiDiscrete (or Box) grid and the action space as Discrete(5). This ensures that any standard RL algorithm can interact with our agent without knowing the internal physics details.

2.3 Training Implementation
File: train_pendulum.py, this layer implements the Q-Learning algorithm using a modular, class-based design, containing:
- **Q-Table Management:** It maintains a 3D NumPy array (20, 20, 5) representing the Q-values for every state-action pair.
- **Update Rule:** The agent learns using the Bellman Equation:
$Q_{new} = Q_{old} + \alpha(Reward + \gamma \cdot maxQ_{future} - Q_{old})$
- **Episode Loop:** The ExperimentRunner class manages the training lifecycle, handling episode resets, step execution, and data collection for performance plotting.

3. Object Oriented Design & Applied Concepts
Our implementation made deliberate use of these key OOP concepts:

3.1 Abstraction
We used Abstract Base Classes (ABCs) to define clear interfaces before implementing logic.

- BaseAgent Interface: We defined an abstract BaseAgent class that enforces methods like act(), update(), and save(). This allows us to easily swap in a different algorithm (e.g., SARSA or Deep Q-Learning) in the future without breaking the main training loop.
- ExplorationStrategy Interface: The decision-making logic is abstracted behind a generic select_action() method, hiding the details of how the decision is actually calculated.

3.2 Encapsulation
We protected the internal state of our objects to prevent "spaghetti code" and accidental data corruption

- **The Q-Table:** The _q_table array is encapsulated as a protected member inside QLearningAgent. External classes (like the training loop) cannot modify Q-values directly; they must use the defined update() method.
- **Hyperparameters:** Learning rates (α) and discount factors (γ) are bundled inside the agent class rather than floating as global variables.

3.3 Polymorphism

We used polymorphism to handle the difference between Training and Evaluation behaviors cleanly.

- **Polymorphic Methods:** The ExplorationStrategy abstract class defines a polymorphic select_action() method. Subclasses EpsilonGreedy (for training) and Greedy (for evaluation) implement this method differently.
- **Runtime Substitution:** The QLearningAgent accepts any object of type ExplorationStrategy in its act() method. This allows us to dynamically swap the agent's behavior from "exploratory" to "optimal" without changing a single line of the agent's code.

3.4 Inheritance

We utilized inheritance to maximize code reuse and enforce standards.

- **Gymnasium Compliance:** PendulumEnv inherits from gym.Env, inheriting critical utility methods and ensuring our environment works with standard RL tools.
- **Agent Hierarchy:** QLearningAgent inherits from BaseAgent, ensuring it implements all necessary methods required by the system.

4. Reflections & Usage of AI Tools

4.1 Reflection

This project helps us learn about various things, such as Discretization Balance, fewer bins led to loss of control (the agent couldn't see fine movements), while more bins made the Q-Table too large to learn efficiently in a reasonable time. It also demonstrated that while Q-Learning is a powerful algorithm, the software architecture surrounding it is equally important. By using OOP principles like Strategy and Encapsulation, we created a robust, flexible codebase that is easy to debug, extend, and understand. It was also our first time using the Gymnasium library, therefore it was an important experience for us to learn something new.

4.2 Usage of AI Tools

AI tools were useful in helping to:

- Create the workflow and a balanced division of labor for the group project
- Proof reading and verifying the english grammar used in the reflection report
- Helping to understand how the Gymnasium library works