OOP Final Project Reflection Report

B123245025 李君潔 B123245026 黃昱憙 B123040060 陳泉龍

Part1:

In part 1, we only changed one line to ensure the code could run properly on our local environment. The original conditional render mode caused rendering issues on our system, so we changed `env = gym.make('MountainCar-v0', render_mode='human' if render else None)` into `env = gym.make("MountainCar-v0", render_mode="human"` to ensure stable visualization.

Part2:

Part 2 was a lot harder than we thought it would be. At first, we attempted to achieve the required success rate of 70% through parameter tuning alone, which yielded inconsistent results that often contained runs where the agent learnt nothing. We observed that it often got stuck on the left side of the map, or kept going in circles. After consulting various sources, we added a variable called min_exploration_rate, and dropped the learning rate once epsilon decayed to that value to increase stability. In addition, we experimented with various values, choosing a low/moderate learning rates, a high discount factor, and a slow decay to encourage exploration. While this indeed made the results relatively more consistent, we still did not achieve a high enough success rate, stuck on around 60%.

Then, we tried synthesizing multiple advanced reinforcement learning techniques into a single cohesive class, incorporating optimistic initialization and reward shaping based on Manhattan distance to mathematically guide the agent toward the goal, while simultaneously using UCB-inspired exploration bonuses and Softmax action selection to intelligently manage curiosity. We added a dedicated 'polish phase' that switches to pure exploitation at the end of training, but we still weren't able to boost the rate over 70%.

However, we still learnt and got to experiment with a lot of new methods we learned this semester, both from OOP and other related classes, gaining valuable experience.

Part3:

This project demonstrates an object-oriented design using abstraction, inheritance, encapsulation, composition, and polymorphism to build a flexible and extensible reinforcement learning framework.

1. agent_base.py: define Agent abstract base class, Encapsulation
    - `BaseAgent` is implemented as an abstract base class using Python's `ABC` module.
    - It specifies that every agent must implement the following behaviors:
        - `select_action(self, obs)`
        - `update(self, obs, action, reward, next_obs, done)`
    - These methods do not provide concrete implementations in `BaseAgent` and are intended to be overridden by subclasses
    - **Abstraction**: BaseAgent exposes what an agent should do without specifying how it is done.
    - **Encapsulation**: The internal decision logic of each agent is hidden behind the common interface, allowing external code to interact with agents in a uniform way.
2. agent_random.py: sub-class, Inheritance, Polymorphism

- Inherits from AgentBase
- Implements:
  - `select_action()` → randomly selects an action
  - `update()` → does nothing (no learning)
- **Inheritance** -> `RandomAgent` is a subclass of `AgentBase`
- **Polymorphism** -> overrides `select_action()` and `update()`

3. agent_qlearning.py: Q-learning, Encapsulation, Polymorphism
   - Inherited from AgentBase
   - Implements Q-learning logic by overriding:
     - `select_action()` → ε-greedy policy
     - `update()` → Q-learning update rule
   - Internally maintains:
     - Q-table
     - learning rate (`alpha`)
     - discount factor (`gamma`)
     - exploration parameters (`epsilon`, decay)
   - **Inheritance** -> `QLearningAgent` is a subclass of `AgentBase`
   - **Polymorphism** -> overrides `select_action()` and `update()`
   - **Encapsulation** -> The Q-table and learning parameters are hidden inside the agent and cannot be modified externally.
     They can only be affected through:
     - `agent.select_action()`
     - `agent.update()`
4. trainer.py: Encapsulation full trainning process
   - Encapsulates the entire training loop, including:
     - environment reset
     - action selection
     - agent update
     - reward tracking
     - performance statistics

     External users only interact with:

     - `trainer.train()`
   - **Encapsulation** -> The full training procedure is hidden behind a single public method.
   - **Polymorphism ->** Trainer does not care whether the agent is random or Q-learning, as long as it follows `AgentBase`.
5. main.py: Program entry, method selection
   - Selects which agent to use (`RandomAgent` or `QLearningAgent`)
   - Creates an environment for `trainer` to start training
   - Optional rendering
   - **Polymorphism ->** Demonstrates polymorphism in practice.
6. warehouse_robot.py: Encapsulation of Robot Model

- Encapsulates:
    - robot position
    - target position
    - movement logic
    - pygame rendering

  Exposes behavior via:

    - `perform_action()`
    - `reset()`
    - `render()`
- **Encapsulation ->** Robot state is hidden and manipulated only through methods.

7. oop_project_env.py: Environment Wrapper, Composition
    - Wraps `WarehouseRobot` into a standard RL environment.
    - Implements Gymnasium-style methods:
        - `reset(self, seed=None, options=None)`
        - `step(self, action)`
    - Defines:
        - `observation_space`
        - `action_space`
    - **Composition ->** The environment contains a `WarehouseRobot` instance.
    - **Encapsulation ->** External code interacts only with the environment interface, not the robot directly.

By separating the agent, environment, and training logic into different classes, we gained a better understanding of how abstraction defines common behaviors, how inheritance allows different agents to share the same interface, and how polymorphism enables the trainer to work with different agents without changing its code.

Through this project, we learned how object-oriented programming helps organize a complex system into clear, modular, and reusable components in practice.

We used AI to help us quickly understand the assignment requirements and the overall structure of the code. It provided useful explanations and implementation ideas, which saved us time and reduced confusion. Even though we used AI, we always reviewed and refined the code ourselves to ensure everything was reasonable, working properly, and aligned with the assignment requirements.