OOP Final Project Reflection Report

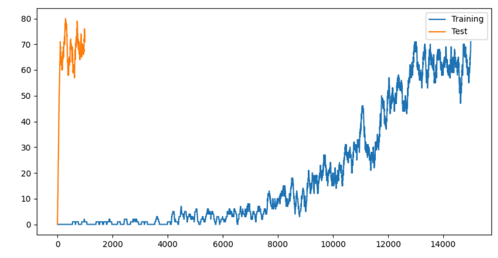
written by B124020010 黃柏瑋 & B124020028 許家瑋

Part 2：(written by B124020028)

When I first ran this file, I started by adjusting the parameters `learning_rate_decay` and `min_learning_rate`, but the success rate remained very low. Later, I began tuning `learning_rate_a` and `discount_factor_g`, which successfully increased the success rate to around 50%. After that, I experimented with modifying the reward structure—for example, giving a small penalty for each step taken to encourage the agent to reach the goal earlier—but the improvement was minimal.

I also tried using Deep Q-Networks, the SARSA algorithm, and Monte Carlo methods. However, many of these approaches required significantly longer training times, and their performance still did not surpass the original method. Therefore, I eventually returned to Q-Learning. I continued testing additional parameters, running many tests and adjusting one or two parameters at a time to study how increasing or decreasing each value affected the accuracy. I also incorporated elements such as early stopping and learning rate decay.

In the end, I chose to use learning rate decay because I found that gradually reducing the learning rate during training—rather than suddenly lowering it only when epsilon reached zero—produced better results. After several weeks of experimenting with countless parameter combinations, the best test result I achieved was 67.6% (episodes = 1000).



Throughout this process, I gained a better understanding of how this type of machine learning works and learned about different training methods and models. Although I spent a great deal of time researching, tuning, testing, and analyzing—and consulted AI tools, YouTube videos, and online articles to help me understand new training techniques and the background of the problem—I ultimately did not meet the assignment requirements. This was disappointing, but I still learned a lot. I now understand that achieving strong performance in this kind of task is very challenging when uncertain factors (such as slippery) and a limited number of episodes are involved. I believe that when I encounter similar tasks in the future, I will be able to approach them much more easily and confidently.

Part 3: (written by B124020010)
overview:

For part 3 I decided to implement a custom env from gymnasium by extending the env class from the library, the env is a warehouse with two agents, and 4 packages will spawn at random shelves(indicated by @ in the map), and the agents will have to go to that point and carry the package all the way back to a fixed point called offload

OOP concepts used:

classes and objects:

the main code is made up of three main classes:

```
class Agent
class Package
class WareHouseEnv(Env)
```

with WareHouseEnv extending the Env class from gymnasium library
with two subclasses that extends Package class:

```
class ExpressPackage(Package)
class HeavyPackage(Package)
```

and WareHouseEnv has its own Agent Objects and Package objects

encapsulation:

I used _(underscore) in front of most of the names for the class fields ex: `self._pos = np.array(start_pos, dtype=np.int32)`
, in hope to simulate private attribute, which does not exists in python, however when it comes to actually using a getter and setter for all these attributes during the environment, I determined that it will be too cumbersome to call a function every time some private fields need to be modified, so I decided to make the WareHouseEnv class a friend to both the Agent class and Package Class to access their private fields, or at least simulate this rule during the code

inheritance:
WareHouseEnv inherits from the Env class from gymnasium, and ExpressPackage and HeavyPackage are subclasses of the Package class

polymorphism:

In order for WareHouseEnv to inherit from the Env class, I need to first know the Env class from gymnasium, the Env class is an abstract base class with 4 things to implement:

```
reset()
step()
action_space #not strictly forced but often necessary
observation_space #not strictly forced but often necessary
```

with an optional function render() to implement for user GUI, So I implemented these 5 into my custom env, reset() resets the env to its initial state, step() determines what happens when the agent takes a step in an episode, action space determine the actions that the agent can take, The observation_space describes the format and constraints of the observations returned by the environment and is used to make the next move, render() uses pygame to present the map and the agents moving within them

I also specialized the Package class into the ExpressPackage class and the HeavyPackage class, these two subclasses overrides

```
def get_priority(self):
        return 0
def on_pickup(self, pickup_reward):
        return pickup_reward
def on_deliver(self, deliver_reward):
        return deliver_reward
```

to have different priorities during package assigning(basically ExpressPackage are the first to be picked up, while HeavyPackage is the last), as well as awarding different rewards when picked up and delivered

Challenges:

For the training of this environment, I decided to use the PPO algorithm from stable_baselines3, a policy-gradient method that learns by improving the policy step-by-step, I started out with only one agent, and after looking at the result an issue is apparent in that the agent will often go back and forth in the env(oscillates) between the two same spaces, so in response I have to make Agent class have more fields and detect oscillations to give out penalties, which did solve it to a degree, but it still happens from time to time, and it also made the code a lot more complicated.

Next I decided to use two agents, so the env would have a list of Agent objects instead of just one object, however, PPO or any algorithms from stable baselines_3 are not built for multi agent training(or MARL), but I decided to still give it a shot, the first issue is that the action_space and observation_space will get a lot larger, luckily it is still in a manageable state, but the big problem for this is that since PPO is not built for multiple agents learning, I have to sum the rewards at the end of step(), this causes only one agent to actually learn and performs well, but the other will be completely incompetent, this imbalance problem, is something I don't know how to solve without switching to completely new code entirely(PettingZoo), since this is just the limitation of the PPO algorithm

LLM usage:

I did use CHATGPT to help me do this part, but I did write the initial WareHouseEnv, Agent and Package class myself with online resources, then I ask it to either patch up bugs and see where it can be improved, as well as asking it to organise my code and clean up unused parts.