ME5418 Machine Learning in Robotics

Autonomous Fruit Picking Robot

Student: SUN YINING

Student ID: A0310458J

Group 7 Teammates: GAO YILIN, LIN HAOYU

College of Design and Engineering

Department of Robotics

National University of Singapore

Session 2024/2025

Introduction

1. Problem Background

In modern agriculture, fruit harvesting remains a labor-intensive task, posing significant challenges in terms of efficiency and cost, especially in large-scale orchards. Autonomous fruit-picking robots offer a promising solution, effectively reducing labor costs while significantly improving harvesting efficiency and accuracy. This advancement not only addresses the complexities of fruit distribution and diverse harvesting demands but also promotes the intelligent and sustainable development of modern agriculture.

2. Limitations of Existing Methods

Traditional approaches perform well in static and straightforward environments, as they rely heavily on predefined rules. However, this dependence limits their application in dynamic and complex scenarios, where their performance decreases significantly. In contrast, reinforcement learning (RL) enables agents to autonomously learn and optimize strategies through interaction with the environment. This eliminates the reliance on expert knowledge or preset rules, making RL more adaptable to the unpredictable and complex conditions of orchard environments. A detailed comparison is provided in the ***proposal report***, and this paper will further demonstrate the results of the A* algorithm for comparison.

3. Project Objectives

The primary goal of this project is to design and implement a RL-based autonomous fruit-picking robot in a 3D discrete grid space. The robot's task is to successfully pick a specified number of ripe fruits within the shortest possible time, while avoiding collisions with trees and staying within the boundaries of the grid.

The project employs the Proximal Policy Optimization 2 (PPO2) algorithm, integrated with the custom-designed fruit-picking environment. The agent interacts with this environment to optimize its policy iteratively, ultimately achieving efficient and accurate fruit harvesting.

Methods

This project is composed of three primary modules: our_gym.py, PPO2network.py and PPO2agent.py.

1. Our_gym

The **State** class defines and manages the state space within the 3D grid. It enables interactions between the agent and the environment, supporting decision-making and reward allocation. A detailed explanation of the class components is available in the ***gym report***, accompanied by a workflow diagram illustrating the State class's operation.

The **FruitPickingEnv** class is responsible for environment generation, action execution, state observation, and rendering. Its core function is to facilitate the complete process of agent action execution, which includes updating the environment state, calculating rewards, and checking the completion status of the task.

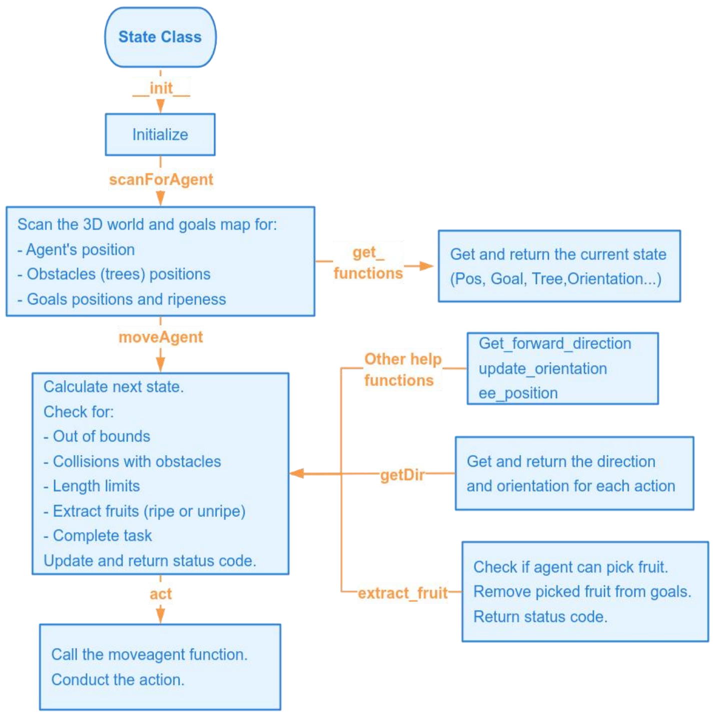
State Class
__init__
Initialize
scanForAgent
Scan the 3D world and goals map for:
- Agent's position
- Obstacles (trees) positions
- Goals positions and ripeness
get_ functions
Get and return the current state (Pos, Goal, Tree,Orientation...)
moveAgent
Calculate next state.
Check for:
- Out of bounds
- Collisions with obstacles
- Length limits
- Extract fruits (ripe or unripe)
- Complete task
Update and return status code.
Other help functions
Get_forward_direction
update_orientation
ee_position
getDir
Get and return the direction and orientation for each action
extract_fruit
Check if agent can pick fruit.
Remove picked fruit from goals.
Return status code.
act
Call the moveagent function.
Conduct the action.

Fig 1 Workflow Diagram of the State Class

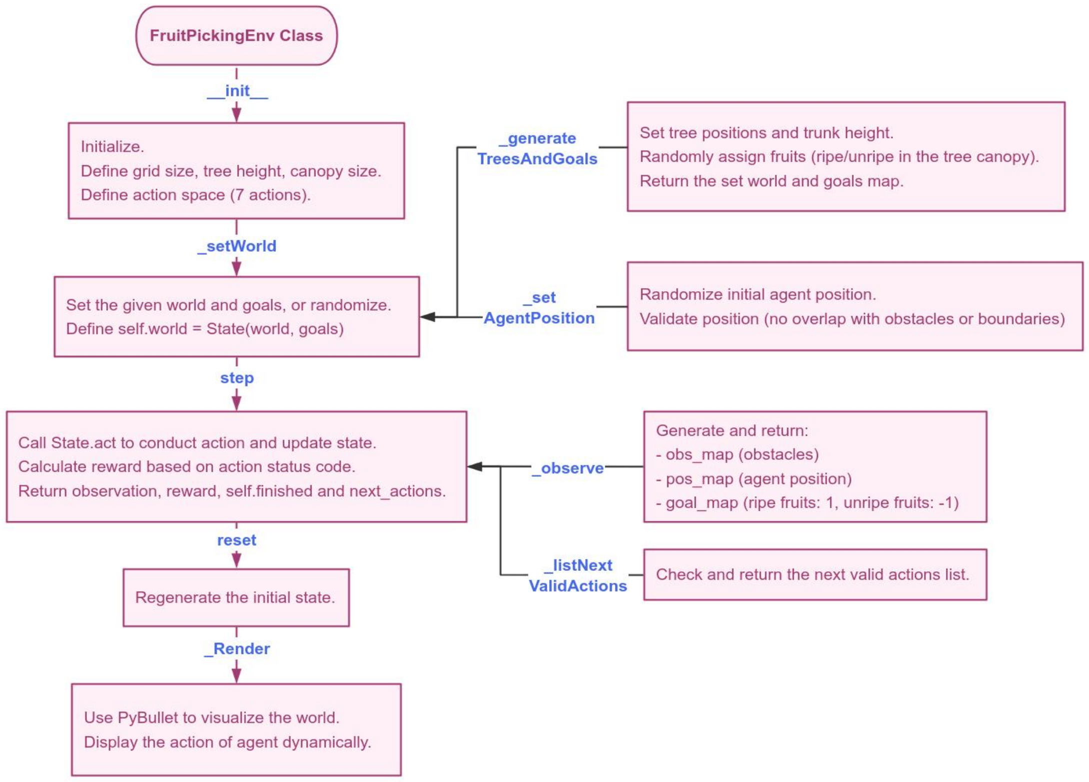
FruitPickingEnv Class
__init__
Initialize.
Define grid size, tree height, canopy size.
Define action space (7 actions).
_generate TreesAndGoals
Set tree positions and trunk height.
Randomly assign fruits (ripe/unripe in the tree canopy).
Return the set world and goals map.
_setWorld
Set the given world and goals, or randomize.
Define self.world = State(world, goals)
_set AgentPosition
Randomize initial agent position.
Validate position (no overlap with obstacles or boundaries)
step
Call State.act to conduct action and update state.
Calculate reward based on action status code.
Return observation, reward, self.finished and next_actions.
_observe
Generate and return:
- obs_map (obstacles)
- pos_map (agent position)
- goal_map (ripe fruits: 1, unripe fruits: -1)
_listNext ValidActions
Check and return the next valid actions list.
reset
Regenerate the initial state.
_Render
Use PyBullet to visualize the world.
Display the action of agent dynamically.

Fig 2 Workflow Diagram of the FruitPickingEnv Class

Compared to the initial version of the our_gym environment, several modifications have been made. Firstly, considering the limitations in computational resources and processing speed, and to facilitate better convergence of the reinforcement learning model, we reduced the world size to **10×10×10** and changed the local view to a **global view**. Second, we have adjusted the methods for randomly **generating the world and goal maps** to better fit the current environment. Additionally, both the **action and reward spaces** have been modified to reduce complexity and improve the model's performance. The current action and reward spaces are as follows:

Table 1 Action Space

Mark	0	1	2	3	4	5	6
Action	MOVE FORWARD	MOVE BACKWARD	TURN RIGHT	TURN LEFT	INCREASE HEIGHT	DECREASE HEIGHT	EXTRACT FRUIT

Table 2 Reward Space

Reward	ACTION COST	PICK NOTHING PENALTY	PICK UNRIPE PENALTY	COLLISION PENALTY	OUT OF BOUNDS PENALTY	MANIPULATOR EXTEND LIMITATION	PICK RIPE REWARD	GOAL REWARD
Value	-1	-3	-5	-5	-5	-5	15	50 END

Furthermore, we have modified and refined several functions to make them more concise and robust. Finally, the **render module** has been adjusted to ensure it correctly renders the revised **URDF** model, providing a smooth and clear visualization of the agent's actions and the changes in the state space.

2. PPO2 network

This section defines a neural network for the PPO2 algorithm, consisting of two parts. The policy network determines action probabilities, guiding the agent toward optimal decisions. The value network estimates the value of the current state. The detailed explanation of the modules is provided in the ***Neural network report***.

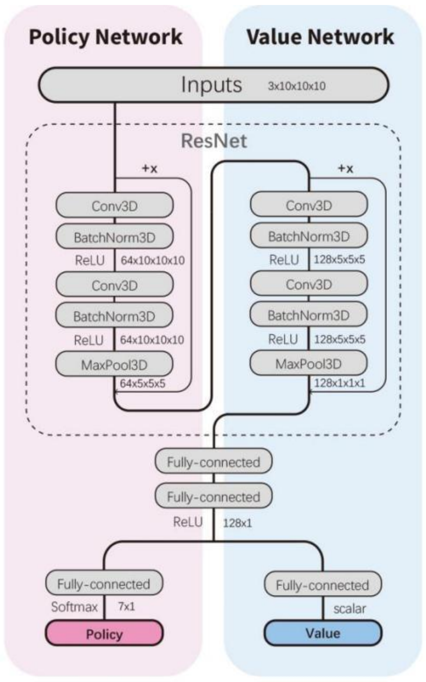

Fig 3 Network Architecture of the PPO2 Algorithm

We have made only a few modifications to the PPO2 neural network module. First, we adjusted the **input dimensions and other parameters** to align with the newly set world size, ensuring compatibility with the updated input format. Second, considering that the subsequent PPO2 algorithm already incorporates a memory module (old ACTOR), we **removed the LSTM module** from this section to reduce the overall network complexity.

3. PPO2 agent

This section implements the reinforcement learning framework based on the PPO2 algorithm[1]. The **PPO2 Actor** class updates policies using the Clipped Surrogate Objective while selecting optimal actions. The **PPO2 Critic** class evaluates state values, guiding policy optimization via MSE loss. Additionally, the ppo2_train function handles model training and saving, whereas the ppo2_test function evaluates the trained model, rendering real-time visualizations to assess policy effectiveness intuitively. This allows for an intuitive assessment of the policy's effectiveness and stability.

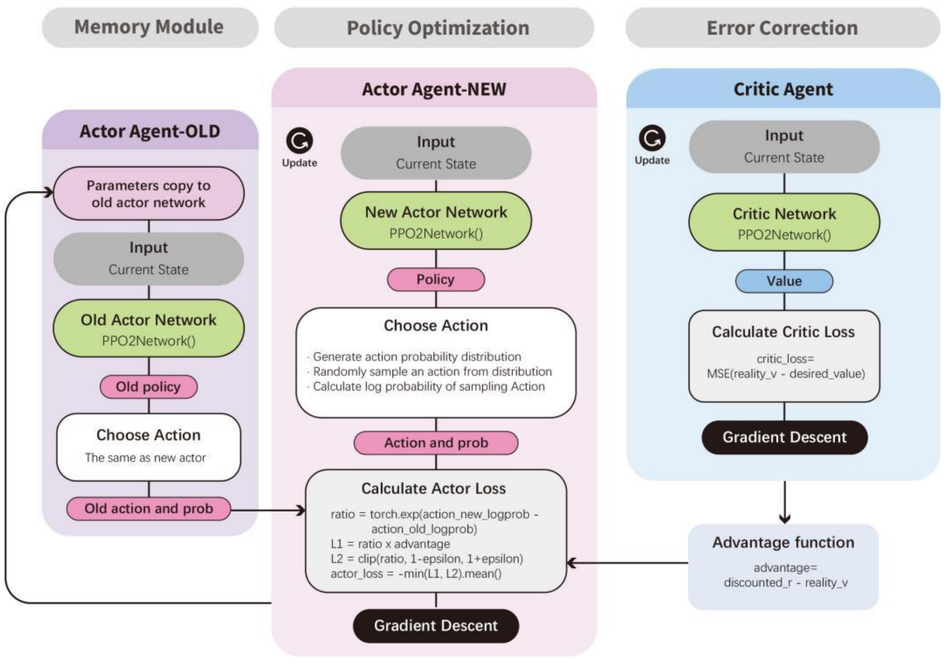

Fig 4 Architecture of the PPO2 Algorithm

We made **no significant modifications** to this section, but instead standardized the code and adjusted various parameters for experimentation. A detailed explanation can be found in the ***Learning Agent Report***.

Experiments Results

1. Experiment 1

The parameters controlling the PPO2 training process are listed in Table 3. *EP_MAX* and *EP_LEN* define the maximum number of training episodes and steps per episode, respectively, setting the training scope. *GAMMA* is the reward discount factor, balancing focus between short- and long-term rewards. *A_LR* and *C_LR* are the learning rates for the Actor and Critic networks, influencing the speed of optimization. *BATCH* specifies the training batch size, while *A_UPDATE_STEPS* and *C_UPDATE_STEPS* determine the number of updates for the Actor and Critic networks per cycle, affecting the frequency of policy and value learning.

Table 3 Experiment Parameters

GYM Environment	Goal_ number	EP_ MAX	EP_ LEN	GAMMA	A_LR	C_LR	BATCH	A_UPDATE _STEPS	C_UPDATE _STEPS
our_gym	3	1000	2000	0.6	0.0001	0.0003	128	10	10

Policy Loss and Value Loss are crucial metrics for assessing the convergence of the PPO2 algorithm. Policy Loss measures the degree of optimization in the action distribution of the policy network during the update process. A lower value indicates successful optimization of the action policy. Value Loss indicates the accuracy of state value estimation by the value network. A lower value implies enhanced learning of state values. The figure below presents the Policy and Value Loss curves in this part, illustrating the convergence trend of policy optimization.

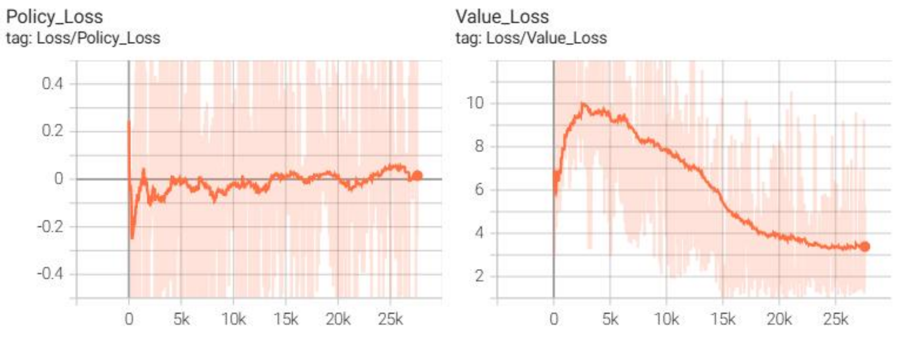

Fig 5 Policy and Value Loss Curves

The results show significant fluctuations in policy loss during the initial training phase, followed by stabilization and convergence near zero as training progresses. Similarly, value loss, initially high, shows a clear downward trend, eventually stabilizing close to zero. These curves demonstrate effective learning of policies and accurate value estimation by the PPO2 algorithm.

The **cumulative reward** represents the total reward obtained by the agent in each episode, serving as a measure of the agent’s task completion efficiency and overall performance.

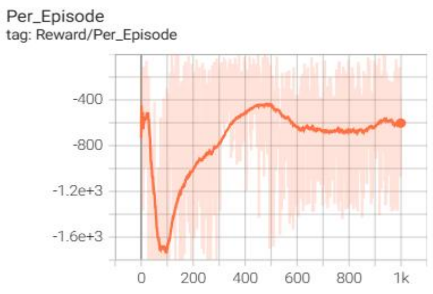

Fig 6 Total Reward Curves per Episode

As shown, the reward curve rises significantly over time. During early training, the agent’s strategy is underdeveloped, leading to inefficient actions or penalties (e.g., picking unripe fruits or collisions). As the agent gains experience, it learns efficient paths and strategies, resulting in higher cumulative rewards, which stabilize in later episodes, indicating policy convergence.

After completing the training of the policy network and value function network, we use several episodes to **test** the model, and the average value is used to evaluate the agent's performance. The **Average Number of Steps per Episode** reflects the average number of steps the agent takes to complete the task. The **Average Total Rewards per Episode** reflects the total reward the agent receives for completing the task in each episode, providing an evaluation of the overall performance of the policy.

Table 4 Test Performance

Number of test episodes	Average number of steps per episode	Average total rewards per episode
100	487.03	-592.84

The results show that the agent achieves fewer average steps and higher total rewards, suggesting efficient fruit-picking strategies with minimal unnecessary exploration. This suggests that the agent has initially mastered an efficient harvesting strategy.

Additionally, we have recorded **Video_PPO2** to illustrate the agent's behavioral trajectory, showing its ability to avoid collisions and pick fruits via optimal paths, albeit with some redundant steps.

2. Experiment 2

This experiment evaluates **three different sets of parameters** to assess their impact on the PPO2 agent's performance. The parameters tested were batch size, learning rates, and gamma. The results are compared with Experiment 1, highlighting how these parameter changes affect the agent's performance in terms of loss, reward, and efficiency.

Table 5 Experiment Parameters for Three Sets of Comparisons

GYM Environment	Goal_ number	EP_ MAX	EP_ LEN	GAMMA	A_LR	C_LR	BATCH	A_UPDATE _STEPS	C_UPDATE _STEPS
our_gym	3	500	2000	0.6	0.0001	0.0003	64	10	10
our_gym	3	1000	2000	0.8	0.0005	0.0008	128	10	10
our_gym	3	1000	2000	0.9	0.0001	0.0003	128	10	10

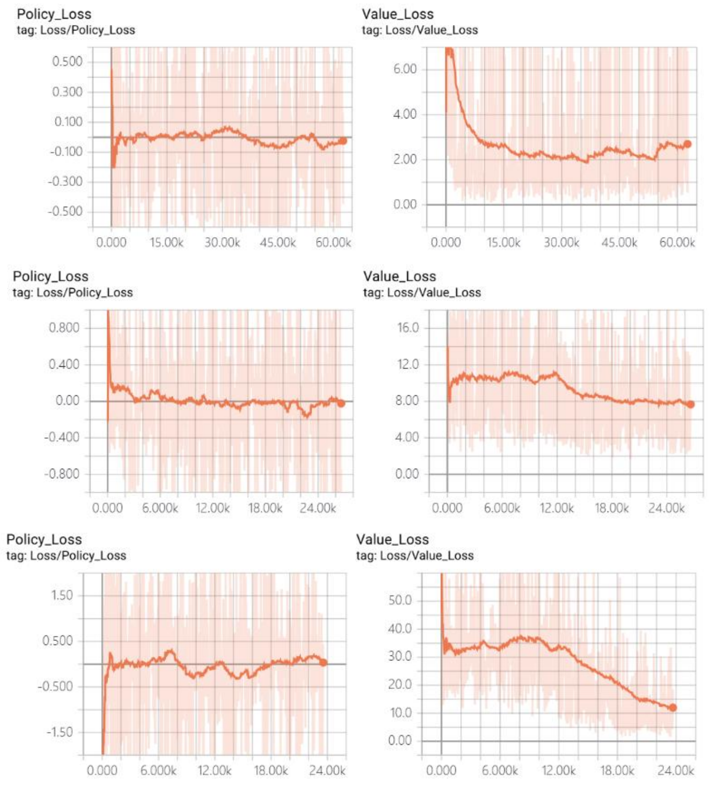

Fig 7 Policy and Value Loss Curves

Compared with Experiment 1, we observed that all three parameter settings led to higher loss values and lower rewards. This suggests that these changes resulted in slower convergence and less efficient learning. Specifically, larger learning rates in Experiment 2b caused more aggressive updates, while Experiment 2a's smaller batch size hindered stable learning. Experiment 2c, with a higher gamma, showed somewhat improved stability but still underperformed relative to Experiment 1, as the loss is totally higher.

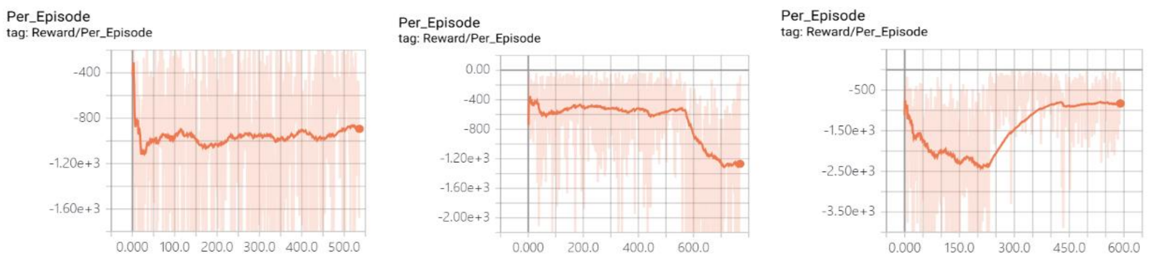

Fig 8 Total Reward Curves per Episode

In terms of rewards, Experiment 2a and Experiment 2b displayed an obvious decline compared to Experiment 1, indicating inefficient exploration and action selection. While Experiment 2c performed better than 2a and 2b, its results were below those of Experiment 1, pointing to need for further parameter tuning to optimize performance.

Table 6 Test Performance

Number of test episodes	Average number of steps per episode	Average total rewards per episode
100	872.97	-1015.11
100	935.45	-1298.21
100	974.46	-1147.05

This part of the experiment does not include any visualization and is primarily an attempt to adjust parameters, serving to clearly contrast with the results of Experiment 1.

3. Experiment 3

This experiment was conducted in a simplified environment, **our_gym_easy**, which **excludes the extract_fruit action and unripe fruits**. In this setting, fruit-picking tasks are completed as soon as the agent's end effector reaches the fruit. This reduces the problem to a basic three-dimensional pathfinding and obstacle avoidance task.

Table 7 Experiment Parameters

GYM Environment	Goal_ number	EP_ MAX	EP_ LEN	GAMMA	A_LR	C_LR	BATCH	A_UPDATE _STEPS	C_UPDATE _STEPS
our_gym_easy	5	1000	500	0.6	0.0001	0.0003	128	10	10

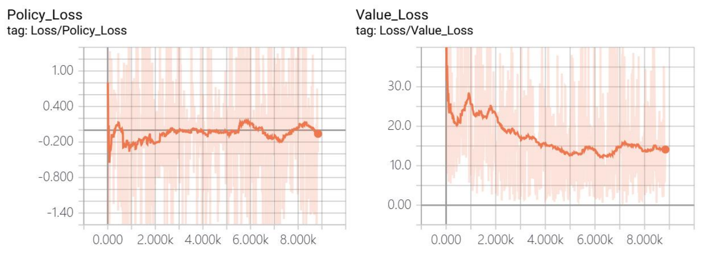

Fig 9 Policy and Value Loss Curves

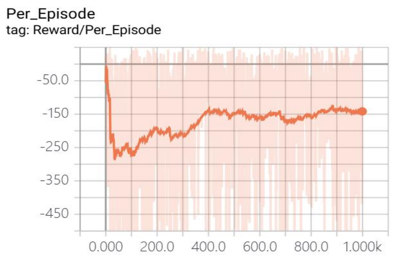

Fig 10 Total Reward Curves per Episode

The simplified environment led to significant improvements in cumulative rewards, indicating that the agent was able to employ efficient strategies for fruit-picking tasks.

Table 8 Test Performance

Number of test episodes	Average number of steps per episode	Average total rewards per episode
100	92.29	-20.33

Video_PPO2_easy demonstration highlights the agent's behavior, showing improved task completion speed compared to Experiment 1.

Compare and Conclusion

1. Experiment Based on A Trajectory Planning Method A*

This experiment evaluates the **A*** trajectory planning method to optimize the agent's path from its current position to the target location[2]. We use a simplified environment, ***a_gym***. The action space was redefined as {0:MOVE_FORWARD,1:MOVE_BACKWARD,2:MOVE_LEFT,3:MOVE_RIGHT,4:INCREASE_HEIGHT, 5:DECREASE_HEIGHT}. Additionally, unnecessary modules were removed, and the robotic arm model was excluded. The fruit-picking task was simplified by directly utilizing the base center coordinates.

In this approach, the path is initially planned in a 2D plane using a heuristic search algorithm to find the shortest route. Then the number of vertical adjustments required for the robotic arm is calculated. For each episode, a complete list of fruit-picking actions is planned, after which the actions are executed and rendered for visualization.

The **a_star** function implements the A* algorithm to compute the shortest path while avoiding obstacles. The **a_star_agent** function initializes the environment, extracts environmental information, and invokes the a_star function for path planning. It generates and executes a sequence of actions, and renders results for visualization.

Table 9 A_STAR Method

```
FUNCTION A_STAR(start, goal, obstacles):
Initialize environment and agent parameters
FOR each episode DO:
    current_position ← Get agent's current position    # Step 1: Extract current state and target
    target_position ← Get target fruit's position
    path_2D ← A_STAR_2D(current_position, target_position, obstacles)    # Step 2: Perform 2D A* path planning
    height_adjustment ← HeightAdjustment(path_2D, target_position)    # Step 3: Compute 3D height adjustment
    action_sequence ← GenerateActions(path_2D, height_adjustments)    # Step 4: Generate action sequence
    FOR each action IN action_sequence DO:    # Step 5: Execute actions and render environment
        ExecuteAction(action)
        RenderEnvironment()
    END FOR
END FOR
```

The **Video_A_star** indicates that the A* method quickly computes the shortest path from the starting point to the target. Its advantages include high computational efficiency, clear path planning, and effective obstacle avoidance. However, its reliance on predefined heuristic searches limits its ability to dynamically adapt to environmental changes.

In this project, A* was primarily employed for fixed-environment path planning, where the shortest path was determined in 2D, followed by necessary 3D height adjustments for fruit-picking. While effective for static environments, this algorithm struggles with tasks such as fruit picking and turning, and the presence of a lateral robotic arm in real-world scenarios adds additional complexity to the path planning problem.

By comparison, reinforcement learning offers greater flexibility. Although RL needs more computational resource, longer training times, and occasional convergence challenges, it allows agents to learn and refine strategies through exploration and rewards. RL demonstrates superior adaptability in dynamic or previously unseen environments, making it a more versatile solution for complex tasks.

2. Project Outcomes and Insights

This project successfully developed a reinforcement learning agent for efficient fruit-picking and compared the performance of A* path planning with the PPO2 reinforcement learning method. The experiments demonstrated PPO2's effectiveness, particularly in dynamic environments where adaptive decisions are critical.

Additionally, through multiple comparative experiments, we gained valuable insights into the importance of reward function design in guiding agent behavior and improving task efficiency. Additionally, the choice of algorithm parameters significantly impacted experimental results.

3. Issues and Directions for Improvement

Simplistic Environment: The current distribution rules for fruits and trees are relatively simple. To further enhance the training performance, we can add more randomness, obstacles, and varying fruit distribution patterns.

Network Architecture Optimization: The neural network architecture currently used is relatively simple and fixed, which limits its performance. Future work could explore more advanced architectures.

High Training Costs and Suboptimal Testing Performance: The training time and computational costs are relatively high. Moreover, although the current training has shown some progress, the testing results are still unsatisfactory, as the agent still engages in much unnecessary exploration. Further optimization is needed to improve efficiency and performance.

Reference

[1] PPO2 algorithm: https://zhuanlan.zhihu.com/p/538486008

[2] Function A*: https://github.com/while-TuRe/A-star-ShortestPath

We referred to the above two sections and made modifications to them, resulting in our own code.