

Faculty of Natural and Mathematical Sciences

# Department of Informatics

King's College London, Strand Campus, London, United Kingdom

7CCSMPRJ

Individual Project Submission 2022/23

Name: Igor Pereira Martins

Student Number: 19034213

Degree Programme: Artificial Intelligence

Project Title: A Comparative Analysis of Reward Functions for

Reinforcement learning in a Warehouse Environment

Supervisor: Dr. Maria Polukarov

Word count: 13,567

RELEASE OF PROJECT

Following the submission of your project, the Department would like to make it publicly available via the library electronic resources. You will retain copyright of the project.

I **agree** to the release of my project

I **do not** agree to the release of my project

**Signature:** (An e-signature is acceptable, i.e., attach your signature as a picture, sign using the draw function)

Date:



Department of Informatics

King’s College London

United Kingdom

7CCSMPRJ MSc Project

A Comparative Analysis of Reward Functions for Reinforcement learning in a Warehouse Environment

Name: Igor Pereira Martins

Student Number: 19034213

Degree Programme: Artificial Intelligence

Supervisor’s Name: Dr. Maria Polukarov

This dissertation is submitted for the degree of MSc in Artificial Intelligence

Abstract

This project presents an exploration of reinforcement learning (RL) agents' behaviours and performance in a warehouse automation environment. The main objective was to analyse the impact of different reward functions on the agents' learning processes and collaboration in a multi-agent system. The project utilized OpenAI's Gymnasium and PettingZoo libraries to develop the warehouse environment, enabling agents to navigate, pick up packages, and deliver them to specified goal states.

Two distinct reward functions were examined: sparse and dense rewards. The sparse reward function provided minimal feedback to agents, rewarding them only upon successful package deliveries. In contrast, the dense reward function offered more frequent and incremental feedback, guiding agents through a series of steps towards their goals. To assess the effects of these reward functions, the agents were trained independently, and their performance was compared over multiple epochs.

The results revealed that the sparse reward function allowed for faster convergence, with agents achieving an average score of 4 points out of 8. Conversely, the dense reward function led to higher overall performance, with agents reaching an average score of 36 out of 40. However, the dense reward function also introduced challenges related to volatility and potential suboptimal outcomes.

This project provides valuable insights into the impact of reward functions on RL agents' learning behaviours and collaboration in warehouse automation environments. The findings serve as a foundation for future research to enhance the adaptability and performance of RL agents in complex and dynamic scenarios.

Contents

[1 Introduction 4](#_Toc142244545)

[1.1 The Problem 4](#_Toc142244546)

[1.2 Motivations 5](#_Toc142244547)

[2 Background and Literature Review 6](#_Toc142244548)

[2.1 Overview 6](#_Toc142244549)

[2.2 Environment Creation 6](#_Toc142244550)

[2.2.1 Gymnasium 6](#_Toc142244551)

[2.2.2 PettingZoo 7](#_Toc142244552)

[2.3 Training Agents 8](#_Toc142244553)

[2.3.1 Tianshou 8](#_Toc142244554)

[2.3.2 Deep Q-Networks 8](#_Toc142244555)

[2.4 Existing Literature 9](#_Toc142244556)

[2.4.1 Autonomous Warehouse Robot using Deep Q-Learning 9](#_Toc142244557)

[2.4.2 Mobile Robot Path Optimization Technique Based on Reinforcement Learning Algorithm in Warehouse Environment 10](#_Toc142244558)

[2.4.3 Autonomous Robot Navigation in Dynamic Environment Using Deep Reinforcement Learning 10](#_Toc142244559)

[3 Objectives, Specifications and Design 12](#_Toc142244560)

[3.1 Objectives 12](#_Toc142244561)

[3.2 Requirements 13](#_Toc142244562)

[3.2.1 Functional Requirements 13](#_Toc142244563)

[3.2.2 Non-Functional Requirements 14](#_Toc142244564)

[3.3 Design 14](#_Toc142244565)

[3.3.1 Technical Specifications 14](#_Toc142244566)

[3.3.2 Environment Design 14](#_Toc142244567)

[3.3.3 Action and Observation Space 15](#_Toc142244568)

[4 Methodology and Implementation 17](#_Toc142244569)

[4.1 Overview 17](#_Toc142244570)

[4.2 Implementation 17](#_Toc142244571)

[4.2.1 Initial Environment Creation 17](#_Toc142244572)

[4.2.2 Multi Agent Environment Creation 19](#_Toc142244573)

[4.2.2 Reward Systems 22](#_Toc142244574)

[4.2.3 Larger Map 23](#_Toc142244575)

[4.3 Methodology 23](#_Toc142244576)

[4.3.1 Overview 23](#_Toc142244577)

[4.3.2 Performance Metrics 23](#_Toc142244578)

[4.3.2 Experiment Design 24](#_Toc142244579)

[4.3.3 Data Collection and Analysis 25](#_Toc142244580)

[5 Results, Analysis and Evaluation 26](#_Toc142244581)

[5.1 Independent Learning 26](#_Toc142244582)

[5.1.1 Sparse Rewards 26](#_Toc142244583)

[5.1.2 Dense Rewards 27](#_Toc142244584)

[5.2.3 Evaluation 28](#_Toc142244585)

[5.2 Combined Learning 29](#_Toc142244586)

[5.2.1 Sparse Rewards 29](#_Toc142244587)

[5.2.2 Dense Rewards 31](#_Toc142244588)

[5.2.3 Evaluation 31](#_Toc142244589)

[5.3 Larger Map 32](#_Toc142244590)

[5.3.1 Sparse Rewards 32](#_Toc142244591)

[5.3.2 Dense Rewards 33](#_Toc142244592)

[5.3.3 Evaluation 33](#_Toc142244593)

[6 Legal, Social, Ethical and Professional Issues 35](#_Toc142244594)

[6.1 Legal Considerations 35](#_Toc142244595)

[6.2 Social and Ethical Implications 35](#_Toc142244596)

[6.3 Professionalism and Codes of Conduct 35](#_Toc142244597)

[6.4 Public Well-being and Security 35](#_Toc142244598)

[7 Conclusion 36](#_Toc142244599)

[7.1 Achievements 36](#_Toc142244600)

[7.2 Limitations 36](#_Toc142244601)

[7.3 Future Work 36](#_Toc142244602)

[8 References 38](#_Toc142244603)

[9 Appendix 39](#_Toc142244604)

[9.1 Appendix A: Policy Trainer 39](#_Toc142244605)

[9.2 Appendix B: Sparse Rewards 42](#_Toc142244606)

[9.2.1 Appendix B.1: Environment 42](#_Toc142244607)

[9.2.2 Appendix B.2: Loader 47](#_Toc142244608)

[9.3 Appendix C: Dense Rewards 48](#_Toc142244609)

[9.3.1 Appendix C.1: Environment 48](#_Toc142244610)

[9.3.2 Appendix C.2: Loader 53](#_Toc142244611)

# 1 Introduction

Reinforcement learning (RL) has witnessed significant advancements in recent times and has been applied to various real-world scenarios, demonstrating its potential for solving complex problems. One of the crucial components in this field is the reward function. This function defines the metric used to quantify the success or desirability of an agent’s actions within an environment. In simple terms, it serves as a guide for the agent to discern which actions are favourable and which should be avoided while pursuing a task. This is done by assigning a numerical value to each state-action pair, indicating the immediate feedback an agent receives upon executing an action in a specific state of the environment.

One of the notable applications of RL in recent times is Warehouse Automation, where warehouses are manned by drones to enhance efficiency, productivity, and safety. These unmanned aerial vehicles (UAVs) are equipped with various sensors and capabilities, making them valuable tools for a range of tasks within the warehouse environment, such as managing inventory, transporting goods, and warehouse mapping.

Training these drones with RL often involves the use of Simulated Environments to allow for safer and more cost-effective exploration and learning, where the reward function becomes an even more important aspect as only by utilising a realistic environment can the drones be trained effectively.

There are two main ways to design a reward function: using sparse rewards, feedback signals that provide information to the agent only occasionally or at specific critical events during its interactions with the environment; or using dense rewards, feedback signals that provide continuous or frequent information to the agent during its interactions with the environment. The choice of reward type depends on the specific task, the agent’s learning capabilities, and the available information in the environment.

## The Problem

As technology continues to advance, the integration of autonomous agents and AI in warehousing is expected to expand. However, warehouse operations present intricate challenges that demand efficient optimisation strategies. One of the primary considerations when training an agent using reinforcement learning is the selection of an appropriate reward function.

The goal of this project is to investigate and compare the effectiveness of the two types of reward functions, sparse and dense rewards, in training RL agents to navigate the warehouse and perform tasks efficiently. The sparse reward function offers rewards only upon successful completion of specific warehouse tasks, such as delivering packages to goal states. The dense reward function, on the other hand, aims to provide more frequent feedback to the agents based on their actions and progress towards their goals.

The project seeks to address the following research questions:

* How does the choice of reward function (sparse vs. dense) affect the convergence speed of RL agents in the warehouse environment?
* What is the impact of different reward functions on the exploration behaviour of RL agents and their ability to adapt to varying warehouse complexities?
* How do RL agents perform in task completion efficiency under sparse and dense reward settings, and which reward function leads to better overall performance?
* How do the RL agents collaborate and coordinate in multi-agent scenarios under different reward functions, and which setting promotes efficient multi-agent collaboration in the warehouse environment?

By evaluating and comparing the performance of RL agents under sparse and dense reward functions in a series of experiments with varying warehouse sizes and package densities, this project aims to provide insights into the most effective reward structure for RL-based warehouse automation. The findings of this research will contribute to enhancing warehouse operations, improving multi-agent coordination, and guiding the development of efficient and adaptive RL strategies in complex environments.

## Motivations

The motivation behind this research lies in the growing significance of reinforcement learning (RL) and its widespread applications across diverse domains. As RL continues to witness remarkable advancements, it becomes increasingly pertinent to explore and understand the impact of key components, such as the reward function, on an agent's learning process. This project's significance stems from its focus on the critical role played by the reward function in training autonomous agents for warehouse automation. By investigating and comparing the two reward functions, this research aims to shed light on the most effective approach to facilitate efficient learning and task completion for drones in a warehouse setting. Such insights are of utmost importance as they hold the potential to revolutionize the efficiency and productivity of warehouse operations, ultimately benefiting industries reliant on smooth logistics and efficient inventory management.

The relevance of this work within the broader project topic of warehouse automation and reinforcement learning is evident in its alignment with the real-world challenges faced by warehouse managers and researchers. As the demand for warehouse automation grows, the implementation of autonomous agents, particularly drones, is increasingly becoming a viable solution to enhance operational efficiency and streamline logistical processes. However, the success of such implementations relies heavily on the design of an optimal reward function. The findings and insights from this study may directly impact the warehouse automation domain, offering valuable guidance on the selection and design of reward functions that lead to more efficient learning, robust multi-agent collaboration, and improved performance of drones within dynamic warehouse settings.

# 2 Background and Literature Review

2.1 Overview

This chapter delves into the literature relevant to this project’s focus, exploring studies that address similar areas. It also provides a detailed examination of the main technologies utilized in this project.

The first section examines existing literature concerning the creation and design of environments for reinforcement learning agents. It explores various techniques and methodologies used to model simulated environments.

The second section provides a detailed example of an environment that is like those used in this project. This environment will showcase the structure and features relevant to warehouse automation. It will encompass elements such as warehouse layouts, package locations, and goal states, reflecting the complexities faced by RL agents in real-world warehouse scenarios.

The final section focuses on literature related to the training of reinforcement learning agents. It delves into different RL algorithms and techniques, such as Deep Q-Networks

(DQNs), to understand their suitability for training autonomous agents in warehouse environments. The literature review will explore state-of-the-art methods for addressing challenges like reward sparsity, exploration-exploitation trade-offs, and efficient convergence to optimal policies. Moreover, the review will consider approaches that facilitate multi-agent training, aiming to optimize the collaboration and coordination among agents in warehouse tasks.

By exploring the literature and technologies within these three main sections, this

chapter aims to lay a strong theoretical foundation for the subsequent stages of the

project.

2.2 Environment Creation

“An in-depth understanding of the particular environment is crucial in reinforcement learning” (Farsang & Szegletes, 2021), the environment plays a pivotal role in the entire process of reinforcement learning (RL). It serves as the virtual world in which the RL agent operates, defining the agent’s goals, permissible actions, and the feedback it receives through rewards. As such, the environment serves as the foundation on which the entire RL process is built. A well-designed environment with meaningful goals, appropriate action space, and a well-defined reward system can lead to efficient learning and policy convergence. Understanding how to create a conducive environment for the research in this project is key to generating valuable insights into autonomous agent behaviour in warehouse settings and optimizing warehouse automation through reinforcement learning.

### 2.2.1 Gymnasium

OpenAI Gymnasium is “An API standard for reinforcement learning with a diverse collection of reference environments” (Gymnasium Documentation, n.d.). It provides a standardized set of environments and a user-friendly API that allows researchers, developers, and practitioners to easily develop, compare, and benchmark RL algorithms and agents.

Gymnasium is widely regarded as a fundamental tool for reinforcement learning research and has become a standard in the RL community. It has several key aspects that make it suitable for the research in this project.

Environment Collection: Gymnasium includes a diverse collection of pre-built environments that cover a wide range of RL problems. These environments are divided into several categories, such as classic control tasks (e.g., cart-pole balancing, mountain car), Atari 2600 game environments, robotics tasks, and more. Each environment is designed to simulate specific challenges and scenarios, making it suitable for studying different aspects of RL algorithms and agent performance. This provides a large pool of examples to pull from when designing the environments used in this research.

Episodic Nature: “OpenAI Gym focuses on the episodic setting of reinforcement learning, where the agent’s experience is broken down into a series of episodes.” (Brockman, et al., 2016). Each episode consists of a sequence of actions taken by the agent, leading to states, rewards, and ultimately the terminal state. The episodic nature is well-suited for tasks that have distinct starting and ending points, such as the environments that will be used to perform the research in this paper.

Custom Environment Creation: While Gymnasium provides a broad selection of pre-built environments, researchers can also create custom environments tailored to their specific research objectives. This customization capability allows for the design of complex and domain-specific environments relevant to warehouse automation and multi-agent systems, which can be beneficial for this project.

### 2.2.2 PettingZoo

PettingZoo is a powerful and versatile open-source platform developed by OpenAI, specifically designed for multi-agent reinforcement learning (RL) environments. It provides a collection of diverse multi-agent environments, facilitating the development and evaluation of multi-agent RL algorithms. PettingZoo serves as an extension of OpenAI Gym, and “was created with the goal of making research in MARL more accessible and serving as a multi-agent version of Gym.” (Black, et al., 2021) and contains several features on top of Gymnasium that are crucial to this project:

Unlike traditional single-agent RL environments, PettingZoo is dedicated to multi-agent settings, where multiple agents coexist and interact in the same environment. It enables researchers to explore the challenges and opportunities of training agents in complex multi-agent scenarios, which are highly relevant in real-world applications such as warehouse automation, traffic control, and robotics.

The platform supports both cooperative and competitive multi-agent environments, as well as a mixture of both. This allows researchers to investigate the dynamics of collaboration and competition among agents, providing insights into how different reward structures and strategies influence agent behaviour, and will allow the comparison of the reward structures in both contexts.

PettingZoo is designed to be compatible with popular RL libraries and frameworks, such as Stable Baselines, Ray Rllib, and Tianshou. This allows researchers to leverage existing RL algorithms and tools for training and evaluating multi-agent systems in PettingZoo environments and will greatly facilitate the training of the agents used in this project.

## 2.3 Training Agents

### 2.3.1 Tianshou

“Tianshou is a reinforcement learning platform based on pure PyTorch” (Welcome to Tianshou! — Tianshou 0.5.1 documentation, n.d.). It is an open-source library designed to facilitate research and development in the field of reinforcement learning (RL). Developed by Tsinghua University’s Robotics Institute, Tianshou provides a comprehensive set of tools and functionalities for training and evaluating RL algorithms, particularly in the context of deep reinforcement learning and multi-agent systems.

Tianshou is a powerful framework with a primary focus on deep reinforcement learning, making use of deep neural networks to handle complex state and action spaces. By providing support for various deep RL algorithms, including Deep Q-Networks (DQNs), Proximal Policy Optimization (PPO), and Advantage Actor-Critic (A2C), it empowers researchers to explore state-of-the-art techniques and tackle intricate real-world problems.

Beyond single-agent RL, Tianshou goes a step further by offering multi-agent support, enabling researchers to delve into the dynamics of multi-agent systems. This capability is particularly relevant for studying complex environments like warehouses, where multiple agents need to interact, collaborate, or compete. The framework allows for in-depth examination and comparison of diverse multi-agent algorithms and strategies, facilitating a comprehensive understanding of their behaviour and performance.

An essential feature of Tianshou is its flexibility when it comes to defining custom reward functions. This flexibility allows researchers to tailor the reward shaping precisely to their specific research objectives. In the context if this project, this feature is crucial for the creation of the two reward structures.

### 2.3.2 Deep Q-Networks

Deep Q-Networks (DQNs) are a groundbreaking class of algorithms in the field of deep reinforcement learning (RL). Introduced by Volodymyr Mnih et al. in their seminal paper “Playing Atari with Deep Reinforcement Learning” in 2013 (Mnih, et al., 2013), DQNs represent a significant advancement in combining deep neural networks with Q-learning for solving complex RL tasks. DQNs have become a fundamental building block for various deep RL algorithms and have been successfully applied in a wide range of domains, including robotics, gaming, and autonomous systems.

DQNs combine the power of deep neural networks with Q-learning, a classic RL technique used to approximate the action-value function (Q-function) of an RL agent. Instead of using traditional Q-tables to store Q-values, DQNs employ deep neural networks to approximate the Q-function, allowing them to handle large and continuous state and action spaces and can efficiently learn and represent intricate patterns and dependencies in the environment, allowing agents to make informed decisions based on their current state and potential actions.

They typically use epsilon-greedy exploration, where the agent chooses actions based on the highest Q-value with a probability of (1 - epsilon) and selects a random action with probability epsilon. This exploration-exploitation trade-off allows the agent to explore the environment and discover new states while also exploiting its learned policy.

DQNs play a crucial role in this project as they offer a powerful and effective approach to handling complex state and action spaces within the warehouse automation environment. DQNs are well-suited for training autonomous agents, such as drones, to navigate and make decisions in dynamic warehouse settings. The ability of DQNs to handle high-dimensional state spaces and large action spaces makes them ideal for simulating real-world warehouse scenarios. Moreover, DQNs facilitate the training of agents in multi-agent systems, enabling them to collaborate and compete effectively within the warehouse environment.

2.4 Existing Literature

This section provides a comprehensive overview of relevant research and scholarly works that have delved into the field of reinforcement learning, specifically in a warehouse environment. They serve as the foundation for understanding the current state of knowledge, methodologies, and findings related to similar projects and research areas. By examining the existing literature, key insights, gaps in knowledge, and potential areas for further exploration were identified.

### 2.4.1 Autonomous Warehouse Robot using Deep Q-Learning

One of the papers analysed was “Autonomous Warehouse Robot using Deep Q-Learning” (Peyas, et al., 2021). The paper explores the application of reinforcement learning to optimize the path of mobile robots operating in a warehouse setting with automated logistics. Through a series of experiments, the authors compared two fundamental reinforcement learning algorithms and evaluated their performance using simulations within the same environment.

The paper offers valuable insights into the characteristics and differences of reinforcement learning algorithms, particularly in the context of path optimization for mobile robots in a warehouse environment. It serves as a solid starting point for understanding the application of Q-learning in pathfinding and simulating the environment for testing purposes in warehouse automation research.

However, it is essential to note that the paper has some limitations. Firstly, it lacks coverage of multi-agent systems and how agents would communicate in a warehouse environment. Considering the collaborative nature of warehouse operations, understanding multi-agent communication and coordination is crucial for comprehensive research.

Additionally, the paper does not address the challenges posed by a dynamic environment, including the presence of obstacles or sudden changes that may impact the agents’ decision-making process. In a real-world warehouse setting, such dynamic factors significantly influence agent behaviour and performance.

Despite these limitations, the paper provides a valuable introduction to the field and lays the groundwork for exploring pathfinding using Q-learning in warehouse environments. However, as this paper served as a foundation for this project, it was essential to build upon

its insights and consider additional research that specifically addresses the challenges of multi-agent coordination and dynamic environment scenarios in warehouse automation. Doing so would allow this project to offer a comprehensive analysis and provide more realistic solutions, making the results of testing different reward functions more viable.

### 2.4.2 Mobile Robot Path Optimization Technique Based on Reinforcement Learning Algorithm in Warehouse Environment

The second paper analysed, titled “Mobile Robot Path Optimization Technique Based on Reinforcement Learning Algorithm in Warehouse Environment” (Lee & Jeong, 2021), introduces the use of Deep Reinforcement Learning (DRL) to address robot navigation and obstacle avoidance challenges in warehouse environments. Additionally, the paper explores traditional Q-learning for optimizing space utilization for product placement. The authors conduct investigations for both the single robot case and extend the approach to multi-robot scenarios by employing a strategic variation of Q-tables for multi-agent Q-learning. The model’s performance is evaluated in a 2D simulation environment for both single and multi-robot cases.

This paper offers in-depth insights into the topic of this project, delving extensively into the methodologies used to train agents with reinforcement learning. It presents detailed explanations, pseudocode, and equations for training the agents in both single-agent and multi-agent systems. The inclusion of these methodologies facilitates a comprehensive understanding of the techniques employed in optimizing robot navigation and space utilization in warehouse environments. However, it is essential to acknowledge that this paper also does not address the training of agents to adapt to dynamic environments. While it provides valuable information regarding agent training in static scenarios, the absence of consideration for dynamic environments, where obstacles and sudden changes occur, is a notable limitation.

Nonetheless, this paper remains a valuable resource for this project, offering comprehensive details on reinforcement learning techniques for training agents in both single-agent and multi-agent systems. By leveraging the insights and methodologies presented in this paper, this project can establish a solid foundation for addressing path optimization and space utilization challenges in warehouse automation.

### 2.4.3 Autonomous Robot Navigation in Dynamic Environment Using Deep Reinforcement Learning

As this project aims to explore the effects of different reward functions, it was essential to supplement this literature with research that specifically investigates training agents to adapt and react to dynamic environments.

The third piece of literature analysed is titled “Autonomous Robot Navigation in Dynamic Environment Using Deep Reinforcement Learning”. The paper explores the application of deep reinforcement learning (DRL) techniques, focusing on the use of CNNs , for autonomous control of robots in simulated dynamic environments. This paper provided valuable insights into how agents can be trained to navigate dynamic environments using reinforcement learning, an area of interest that was not extensively covered in the previous literature. Specifically, the study shed light on the effectiveness of DRL methods, in enabling autonomous robot navigation, making it a valuable reference for this project.

While the paper did not focus on multi-agent systems, it offered a deeper understanding of how reinforcement learning techniques can be employed to navigate dynamic environments, which was the missing element of the investigations undertaken before the conception of this project. Moreover, the paper introduced a novel reward shaping method, which could potentially be adapted and incorporated into your project to address the challenges of dynamic environments and optimize agent behaviour.

The findings from this paper provided broader knowledge beyond deterministic methods and emphasized the potential of reinforcement learning in tackling dynamic scenarios. Building upon the methodologies and insights from this paper, this project can explore how different reward functions affect agent navigation and behaviour in dynamic warehouse environments.

# 3 Objectives, Specifications and Design

## 3.1 Objectives

The main objectives of this project are touched upon briefly in the introduction, in this section, they will be expounded upon, those objectives are:

* How does the choice of reward function (sparse vs. dense) affect the convergence speed of RL agents in the warehouse environment?
* What is the impact of different reward functions on the exploration behavior of RL agents and their ability to adapt to varying warehouse complexities?
* How do RL agents perform task completion efficiency under sparse and dense reward settings, and which reward function leads to better overall performance?
* How do the RL agents collaborate and coordinate in multi-agent scenarios under different reward functions, and which setting promotes efficient multi-agent collaboration in the warehouse environment?

The first objective of this project is to investigate how the choice of reward function, specifically sparse and dense rewards, affects the convergence speed of RL agents in the warehouse environment. Convergence speed refers to how quickly the agents reach a stable policy, indicating efficient learning and adaptation to the environment. By conducting experiments with different reward functions and tracking the agents' learning progress over multiple epochs, this objective aims to identify which reward function facilitates more effective learning in the warehouse setting. Understanding the impact of reward functions on convergence speed is critical for optimizing training times and resource utilization in real world applications where time and efficiency are paramount.

The second objective focuses on examining how different reward functions influence the exploration behavior of RL agents and their adaptability to varying warehouse complexities. Exploration is a crucial aspect of RL, as agents need to explore their environment to discover new strategies and optimal policies. By comparing the exploration behavior under sparse and dense reward settings, this objective aims to shed light on which reward function encourages more efficient exploration and enables agents to adapt better to dynamic and challenging warehouse environments. The findings will contribute to the design of reward functions that strike a balance between exploration and exploitation, maximizing the agents' learning capabilities and adaptability.

The third objective delves into evaluating the task completion efficiency of RL agents under sparse and dense reward settings. Task completion efficiency refers to how well agents can perform their designated warehouse tasks, in the case of this project package delivery, with different reward functions. By measuring and comparing the agents' performance in completing tasks, this objective aims to determine which reward function leads to better overall performance and task completion rates in the warehouse environment. The insights gained from this analysis will inform the selection of reward functions that optimize the agents' efficiency and productivity in real-world warehouse operations.

Lastly, this project aims to study how RL agents collaborate and coordinate in multi-agent scenarios under different reward functions, particularly in the context of warehouse automation with multiple drones working concurrently. Efficient multi-agent collaboration is crucial for smooth warehouse operations and minimizing conflicts or collisions among agents. By observing and analyzing the agents' behaviors and coordination patterns, this objective seeks to identify which reward function promotes more efficient multi-agent collaboration and leads to improved overall performance in the warehouse environment. Understanding the dynamics of reward functions in multi-agent systems will provide valuable insights for designing cooperative and competitive strategies in real-world warehouse automation scenarios.

## 3.2 Requirements

### 3.2.1 Functional Requirements

The following functional requirements outline the core functionality that the environment and agents must possess at an abstract level.

Environments:

* Properly Render the Warehouse and Drone Locations: The environments must visually represent the warehouse layout and the positions of drones accurately. This visualization is crucial for evaluating training progress visually, ensuring the results are reliable and practical.
* Navigable Area: The environment should provide a navigable area where agents can move without encountering obstacles or issues. This ensures that the agents can perform their tasks effectively and without hindrance.
* Easy to Edit: The environments should be easy to edit, allowing testing to be conducted under diverse conditions. The ability to modify the environment easily enables experimentation with various warehouse setups and configurations.
* Implement Sparse and Dense Reward Metrics: The environments should implement both sparse and dense reward metrics, providing the flexibility to choose which metric to use during agent training. This allows for a comparative analysis of the two reward types.

Agents:

* Navigation Capability: The agents must be capable of navigating the environment correctly, avoiding collisions with walls and other drones. This ensures that the agents' movements are logical and realistic within the warehouse environment.
* Task Completion: The agents should be capable of performing the task required to terminate an epoch successfully. Task completion is essential for evaluating the agent's overall performance and learning progress.
* Limited by Variable: To prevent agents from running indefinitely, they should be constrained by a variable that limits their operations within a reasonable time frame.
* Collision Handling: The agents should be equipped to handle collisions, whether with walls or other drones. Proper collision handling ensures that the agents can navigate the environment safely and avoid undesirable behavior.

### 3.2.2 Non-Functional Requirements

In addition to complying with the functional requirements mentioned above, the system must fulfill a set of non-functional requirements, as follows:

* Ease of Setup: The system should have a straightforward setup process to facilitate training and testing with minimal input. This ease of setup streamlines the experimentation process and reduces the complexity of using the system.
* Model Saving: The system should automatically save the best model during the training process. Automatic model saving ensures that valuable progress and learned policies are preserved and accessible for later use.
* Hyper-parameter Customization: The system should provide the capability to easily amend hyper-parameters during training. This feature allows for efficient tweaking and fine-tuning of training parameters, enabling better optimization of the reinforcement learning process.

## 3.3 Design

### 3.3.1 Technical Specifications

The environments and agents in this project were implemented using Python 3 due to its extensive support for reinforcement learning, backed by several powerful libraries. Python provides a robust ecosystem for RL development, making it an ideal choice for this project. Some of the essential libraries used include OpenAI's Gymnasium, Torch, and additional add-on libraries that enable multi-agent systems. Python's support for both single-agent reinforcement learning, and environments served as a solid foundation for converting the environment to a multi-agent setting.

The environments were created using the Gymnasium extension called Pettingzoo, designed explicitly for building multi-agent RL environments. Pettingzoo simplifies the process of designing complex multi-agent systems by providing a high-level API and pre-defined environments that support multiple agents interacting simultaneously.

For training the models, the project employed Tianshou, a library built on top of Torch. Tianshou offers various functionalities for training multi-agent systems with diverse policies. In this project, a Deep Q-Network (DQN) policy was utilized. DQN is a popular and effective RL algorithm that is well-suited for multi-agent reinforcement learning tasks.

The choice of these libraries not only facilitated the implementation of the environments and agents but also provided the necessary tools and functionalities to train and evaluate multi-agent systems effectively.

### 3.3.2 Environment Design

The environments used in this project are based on the RWARE environment (Filippos & Schäfer, 2023), which serves as the foundation for the simulation.

The primary objective of these environments is to challenge the agents to navigate efficiently and complete tasks involving the movement of packages from their initial positions to specified goal states. The task within these environments involves a series of steps for the agents to follow:

1. Starting Point: Each agent begins at a designated starting point within the grid world.
2. Package Location: The agents must locate a package located at a specific position in the grid world. The location of the package serves as an intermediate target for the agents.
3. Package Retrieval: After identifying the package's location, the agents are required to move to the package's position and retrieve it. This step involves planning efficient paths and avoiding obstacles or other agents.
4. Goal State: Once the agents successfully acquire the package, they must navigate to a specified goal state within the grid world. The goal state represents the destination for the package and concludes the task.

The environments are represented as grid worlds, where each cell in the grid corresponds to a specific location in the warehouse. The grid provides a structured and easily navigable representation of the environment.

An example of the environment is as follows:

A diagram of a graph

Description automatically generated

Figure - Map Design

In this example, the grid world is represented by a 5x5 matrix. The letters represent the following elements:

* 'S': Starting point of the agent(s).
* '.': Empty cell, which the agents can move through.
* 'P': Position of the package that needs to be retrieved.
* 'G': Goal state where the package should be delivered.

The agents in this environment must navigate from the starting point 'S' to the package position 'P', retrieve the package, and then deliver it to the goal state 'G'.

### 3.3.3 Action and Observation Space

The action and observation spaces are critical components in designing an effective RL environment for warehouse automation. The design of these spaces will significantly influence the agents' learning capabilities, exploration behavior, and task completion efficiency. By carefully defining the action and observation spaces, the proposed warehouse environment can create a realistic and challenging setting for training RL agents to navigate, collaborate, and perform tasks effectively within the warehouse.

In the proposed warehouse environment, the action space represents the set of possible actions that an RL agent can take. These actions should include moving in different directions, picking up packages, delivering packages to specified locations, and avoiding collisions with obstacles or other agents. The action space needs to be carefully designed to allow for a diverse range of agent behaviors and decision-making while ensuring that the agents adhere to the constraints of the environment. Each action in the action space is associated with a specific numeric value, which is used to guide the RL agent's decision-making process based on the feedback received from the reward function. The initial proposed action space is such:

0: Move Up

1: Move Right

2: Move Down

3: Move Left

4: Pick Up Package

5: Drop Package

An alternative to this action space is combining the action the picks up and drops the package, like in RWARE, depending on whether the drone has a package already or not, another option could be the removal those actions all together and simply assume the drone picks up the package when it finds it and can only drop it in designated zones, this would reduce the complexity of the environment, but make it less realistic and applicable to the real world.

On the other hand, the observation space in the warehouse environment encompasses the information that the RL agent can perceive about its surroundings. This information is crucial for the agent to make informed decisions and determine the most suitable actions in each state of the environment. The observation space may include various sensory inputs from the drone, such as its current location, the positions of packages and goals, the presence of obstacles or other agents, and any relevant environmental factors that impact the agent's decision-making process. The observation space needs to be carefully designed to provide sufficient information for the RL agent to navigate and interact effectively with the environment.

The initial design for the observation space will be to give the drone access to a map, in the form of a state list, of the warehouse, allowing it to know here packages are at any time, this is in contrast to RWARE where the drone has a limited scope of vision, and this is done as this project will be unlikely to perform experiments on a continuous environment without a set end, so the drones do not need to be as adaptable to changes as the ones in RWARE.

# 4 Methodology and Implementation

## 4.1 Overview

In this section, I will provide a comprehensive overview of the implementation of my project, along with a detailed explanation of the methodology used to gather the results. The implementation and methodology are crucial components of the research process, as they define the framework and procedures employed to address the project's objectives effectively.

## 4.2 Implementation

### 4.2.1 Initial Environment Creation

During the initial stages of implementing this project, the focus was on learning how to create a custom environment using Open AI's Gymnasium. As previously emphasized, the environment is a fundamental component of reinforcement learning, serving as the foundation for training agents and conducting experiments. While Gymnasium did not provide all the necessary functionalities for the full scope of the project, it was chosen as a starting point to build a solid groundwork for the more complex environments that would be developed later.

To facilitate testing and verification, a graphical user interface (UI) was created to provide a visual representation of the environment. This UI played a crucial role in observing the agent's actions, validating results, and identifying any issues with the learning process. For instance, where the agent finds a method to gain an unexpected and excessive number of rewards, which could indicate potential loopholes or exploitable aspects of the system, it is much easier to understand what is causing these issues when once can directly see what the agents are doing.

Gymnasium provides the choice of two methods to render environments, text based in the console and using the Pygame Library, for this project, the use of Pygame was chosen as a text based UI can often times be messy and make it hard to discern exactly what’s wrong with the environment, this meant that the creation of simple and user-friendly assets to represent the agents and packages in the environment was essential. Two basic assets, representing the drone and the package, were designed for the visual representation:

|  |  |
| --- | --- |
| A black background with a black square  Description automatically generated with medium confidence  Figure - Drone Image | A brown square with a white line  Description automatically generated  Figure - Package Image |

The initial iteration of the project focused on a small-scale warehouse environment with a single agent. This environment featured a 5x5 warehouse with one drone and one package that needed to be delivered, exactly as was presented in the design portion. This setup allowed for various tests to be conducted, ensuring the environment's functionality before expanding to more complex scenarios.

A computer screen shot of a computer

Description automatically generated

Figure - Single Agent Environment

During this phase, particular attention was given to ensure the action space functioned correctly, allowing the agent to perform actions as intended. The agent was capable of standard movement, going anywhere in the warehouse and not clipping through the bounds of the five-by-five space, as well as being able to attempt to pick up a package at any point, as well as drop the package at any point.

The observation space was simply a map of the warehouse fed to the agent in list form as well as its own state, so that the agent’s decisions were motivated by where it was in the map, the assumption was that the agent would then know when to pick up and drop off packages based on what state it was in, this was later amended to contain a binary value so that the agent knew when it was holding a package or not.

Additionally, a rudimentary reward system was implemented to enable the agents to learn how to solve the environment effectively, giving the agent one point if they performed the correct action, such as picking up a package or dropping it in the right place, and taking a point away from the agent for any other action, to try and incentivize the agent to optimize its actions.

The goal in this stage was simply have the drone move towards the package, pick it up, move to the goal state and then drop it off, when this proved easy for the agent, the goal was expanded into seeing how many packages could be delivered before the agent’s battery life ran out, which was done to try and make the environment more realistic, as rarely will a drone be tasked with the delivery of a single package.

For the initial training, Stable Baselines3 was utilized as it provided a simple and straightforward implementation, despite its limitation in handling multiple agents. This allowed the project to focus on the core aspects of environment design, action space, and reward system, paving the way for further development and expansion to more sophisticated multi-agent systems in subsequent stages instead of spending time trying to figure out more complex methods of training which would be redundant if the environment didn’t work in the first place.

This version of the project came with one main issue: the agent was struggling to learn where it needed to be to pick up the box and where to drop it off safely, this would often mean that the agent needed a very long time to learn a very simple goal.

Several attempts were made to try and solve this issue, such as taking away more than one point if the agent picked up or dropped nothing, which lead to the agent learning that it was better to simply not try to pick up anything at all, or providing the agent with bonus points for reaching a state that had a package, which lead to the agent moving towards a package and simply hovering over it by moving into a wall repeatedly to gain as many points as possible.

Reducing the action space, so that action of picking up and dropping off were mapped to the same number and the effect changed depending on whether the agent had a package or not, therefore reducing the complexity of the action space, seemed to fix the issue in this version of the environment.

### 4.2.2 Multi Agent Environment Creation

Once the basic environment had been tested thoroughly, it was time to expand on it to better meet the scope and requirements set out for this project, to do this it was necessary to switch the environment from a simple single agent Gym environment to a more complex PettingZoo environment.

As PettingZoo builds upon Gymnasium, its options for rendering remained the same, so no new work needed to be done on that front, and focused could be placed on the customizability of the environment, this involved total control over the number of agents acting at the same time, to the size and shape of the warehouse, and the amount and location of packages that needed to be delivered.

Building on the foundation of the first version, the map that was used to build the warehouse, which was stored in the same format as what was given to the agents in their observation space, was changed so it could be easily edited by a user, with numbers representing where elements needed to go, allowing a user to easily create any warehouse they deemed necessary.

The environment was expanded to accommodate two agents instead of one, it was made bigger to provide space for the exploration necessary for a DQN to learn effectively, and more packages were added so that the agents did not need to fight over a single package which would lead to several unnecessary collisions that would reduce the efficiency of learning, extra goal spaces were added to allow multiple agents to deliver at the same time, and it was made so that agents would take action one after the other, this was chosen rather than then acting concurrently, as it simplified the environment, and prevent issues such as agents moving to occupy the same space from arising.

A computer screen shot of a computer

Description automatically generated

Figure - Multi-Agent Environment

In the subsequent stages of development, further changes were introduced to the action space to enhance the agents' learning process. It was observed that agents faced difficulties when dealing with multiple packages in the environment simultaneously. To streamline the learning process, the decision of picking up and dropping packages was automated. The agents were no longer responsible for these actions; instead, the environment automatically handled package pickup and delivery when an agent without a package moved onto the same space as one or delivered a package to a goal state. This adjustment allowed the agents to focus on the main objective of finding all the packages efficiently. While an alternative solution could have involved restricting pickup/drop actions to specific states, the simpler approach was chosen due to other higher-priority challenges encountered during this version.

To accommodate the presence of multiple agents in the warehouse, the observation state needed modification. The state was expanded to include information about the location and condition of each package, indicating whether a package was still in the warehouse, held by one of the drones, or already delivered. This would prevent agents from trying to find packages that weren’t there anymore because another agent was holding it or had delivered it already.

Moreover, the reward system underwent adaptation to penalize collisions between drones, discouraging drones from attempting to move into each other indefinitely. This penalty was extended to include obstacles like walls, incentivizing the drones to keep moving and explore more of the warehouse. The goal at this stage was to foster collaboration between the drones, motivating them to work together to deliver all the packages. The agents shared rewards to encourage teamwork rather than competing for packages to the detriment of their learning.

As the project progressed and expanded into a multi-agent system, the need for a more sophisticated library to handle the complex interactions between multiple agents became apparent. While Stable Baselines3 was effective for single-agent environments, it lacked native support for multi-agent systems. Thus, a transition to Tianshou was undertaken to capitalize on its extensive support for deep reinforcement learning algorithms in multi-agent scenarios.

The shift to Tianshou brought its own set of challenges, particularly in the tuning of hyperparameters to optimize the learning process. Hyperparameters play a crucial role in shaping the behavior of the agents during training, and finding the right configuration is often a laborious and iterative process. Properly tuned hyperparameters can significantly impact the convergence speed, exploration capabilities, and overall performance of the agents. Thus, a considerable amount of experimentation was conducted to fine-tune the hyperparameters to suit the specific multi-agent environment and reward functions.

Additionally, Tianshou required adaptations to the environment functions to accommodate its unique requirements for multi-agent support. As the multi-agent environment differed substantially from the original single-agent setup, modifications were made to the observation space, action space, and reward system to ensure compatibility with Tianshou's training process. Ensuring that the environment adhered to Tianshou's specifications was crucial for successful and meaningful training.

To determine the most suitable library for multi-agent training, alternative options like Rlib and CleanRL were also tested. However, Tianshou emerged as the most compatible and versatile library for this project, effectively addressing the complexities of training agents in multi-agent environments. Its extensive support for various deep reinforcement learning algorithms and multi-agent strategies proved invaluable in implementing the project's objectives.

Although the transition to Tianshou presented its fair share of challenges, the use of this library ultimately facilitated the training and evaluation of RL agents in the multi-agent warehouse environment effectively. By overcoming these hurdles, the project was able to delve deeper into the intricacies of multi-agent collaboration, coordination, and learning, thus providing valuable insights into the impact of different reward functions on the agents' performance. The successful integration of Tianshou into the project's methodology paved the way for comprehensive experimentation and a more comprehensive understanding of how reward functions influence multi-agent behavior and task completion in warehouse automation.

The shift towards multi-agent systems and the adoption of new libraries introduced novel challenges that required careful consideration and experimentation. One of the main challenges was to ensure that the agents could learn efficiently without negatively impacting each other's training progress. To address this, two distinct methods of training were developed to explore how the choice of reward system influenced agent behavior and collaboration.

The first training method involved separate training, where one drone was designated as the learning agent, while the other drone took random moves to simulate collisions and potential obstacles. This approach aimed to create a controlled learning environment for the primary agent, allowing it to explore and learn without unnecessary interference from the other agent. The separate training method was particularly useful for understanding how individual drones learned and adapted to the environment's complexities. It provided valuable insights into the agent's ability to navigate the warehouse, locate packages, and deliver them successfully.

The second training method, simultaneous training, involved both drones running simultaneously in the environment. This approach facilitated multi-agent collaboration and coordination, as the agents had to interact and work together to complete the warehouse tasks. The reward system played a crucial role in shaping the dynamics of this collaboration. By designing reward functions that encouraged teamwork and cooperation, the project aimed to enhance the agents' ability to collaborate efficiently and achieve better overall performance in completing the warehouse tasks.

### 4.2.2 Reward Systems

As the primary objective of this project was to compare the effects of reward functions on multi-agent environments, two distinct environments were developed to cater to each reward function - sparse and dense.

The sparse reward environment focused on providing minimal feedback, rewarding agents only upon successful completion of critical events, such as delivering a package to its goal state. This scarcity of feedback posed a challenge for the agents, as they had to navigate and explore the environment with limited guidance. In contrast, the dense reward environment sought to offer more frequent and detailed feedback to guide the agents through a series of incremental steps towards their goals. This approach aimed to encourage more exploratory behavior and efficient learning by providing more informative cues to the agents.

In the sparse reward environment, the agents were rewarded with one point for successfully locating a package and an additional point for delivering it to its destination. Initially, negative rewards were incorporated, deducting points for actions that led the agents to any state other than the goal state. However, this approach proved to be counterproductive, as the agents exhibited volatile learning behavior and struggled to learn optimal paths. Instead of learning where to go, the agents wandered aimlessly, occasionally stumbling upon a box, or completing a delivery by chance. To stabilize the learning curve and encourage more focused learning, the reward function was simplified to provide feedback only when the agents made progress towards their goal. All forms of negative rewards, including collisions with other agents, were removed, allowing the agents to receive feedback only when their actions brought them closer to achieving their objectives. While this improved the learning process, it was observed that even after 500 epochs, individual agents could only successfully handle up to three boxes.

In contrast, the dense reward environment aimed to provide the agents with more frequent and incremental feedback. Each step in the right direction resulted in increasing rewards, while taking the wrong step deducted more points from the agent, with penalties increasing the closer the agents were to the goal. The objective would shift dynamically between a package yet to be picked up and a goal state, guiding the agents towards their targets. Like the sparse reward environment, the dense reward function initially faced issues where agents would frequently receive low rewards but occasionally succeed. To address this, negative rewards were removed, fostering a more focused learning process. The way the agents were rewarded had to be adapted as well. Instead of a reward increment based on proximity to the goal, the agents would now receive a point each time they got closer to the goal than they had been before, ensuring a more progressive and efficient learning experience.

The development of these two distinct environments catering to sparse and dense reward functions would allow for a systematic examination of how these reward structures influenced the agents' behavior and performance.

### 4.2.3 Larger Map

Once the initial implementation of the warehouse automation environment was completed and the core features were successfully integrated, the focus shifted to conducting more comprehensive testing on larger maps to evaluate the performance of the different reward functions. A crucial aspect of this testing was the ability to customize the map, which allowed the creation of a more complex environment with an extended size.

The default warehouse environment had a size of 6 by 6, which served as a suitable starting point for initial testing and development. However, to examine how the reward functions performed in more challenging and expansive scenarios, a new map was designed with dimensions of 8 by 8. This increased size not only provided more space for the agents to explore but also introduced additional complexities and longer paths to deliver packages.

A screenshot of a computer

Description automatically generated

Figure - Larger Warehouse Map

By implementing the larger map and customizing the warehouse environment, the project aimed to assess the generalizability and scalability of the reward functions. It sought to understand whether the agents' performance in the standard 6 by 6 environments would carry over to the more extensive 8 by 8 map or if the larger space would present new challenges and require further adjustments to the reward functions. This analysis would provide valuable insights into the adaptability of the agents and the effectiveness of the reward functions in varying warehouse sizes, thus contributing to a more comprehensive understanding of the system's capabilities and limitations.

## Methodology

### 4.3.1 Overview

This section will describe the Methodology used for gathering results. This involved a systematic and rigorous approach to train RL agents in the warehouse environment, evaluate their performance, and compare the effectiveness of different reward functions.

### Performance Metrics

To comprehensively evaluate the effectiveness of the two reward functions, a range of performance metrics was established. These metrics aimed to provide insights into the agents' learning behaviors, convergence patterns, and overall performance in the warehouse environment. The main performance metrics included:

Average Episode Rewards: this was a fundamental metric used to assess the agents' performance under different reward settings. For each episode of training, the agents' actions and interactions with the warehouse environment contributed to the cumulative rewards received. By calculating the average episode rewards over multiple training episodes, it could be gauged how effectively the RL agents were achieving their goals and completing warehouse tasks under both sparse and dense reward functions. Higher average episode rewards indicated better performance and successful task completion.

Convergence speed: this metric was used to evaluate how quickly the RL agents reached a stable and optimal policy. In reinforcement learning, convergence refers to the point at which the agents' policies stop changing significantly, indicating that they have learned the most efficient strategies to navigate the environment. Monitoring the convergence speed under different reward functions allowed a comparison of how rapidly the agents adapted their policies to the warehouse environment and achieved optimal performance.

Performance over a Set Number of Episodes: To ensure robust evaluation, the agents' performance was analyzed over a predetermined number of training episodes. By observing the agents' progress and learning trajectory over these episodes, it was possible to identify any trends or patterns in their performance. This analysis helped to assess whether the agents were consistently improving their policies, stabilizing their learning, or encountering issues with convergence or oscillation.

These performance metrics were carefully analyzed and compared to draw meaningful insights into the impact of different reward functions on the RL agents' learning behaviors and overall warehouse automation performance. The comparison of these metrics under sparse and dense reward settings allowed the identification of both the strengths and limitations of each reward structure, as well as the most effective RL algorithm for training agents in the warehouse environment.

### Experiment Design

The performance metrics discussed earlier served as essential tools to design and conduct experiments that aimed to assess the effectiveness of different reward metrics in various scenarios.

The first set of experiments focused on evaluating the performance of the reward functions in the standard multi-agent environment. This involved training the agents using both separate training, where one drone learned while the other took random moves to simulate collisions, and joint training, where both drones ran simultaneously. Comparing the results obtained from both training methods had the goal or understanding how the reward functions influenced each approach. This analysis aimed to shed light on the strengths and limitations of each reward metric in promoting effective learning and collaboration in multi-agent settings.

The second set of experiments involved training the agents in a larger-sized warehouse environment. This experiment aimed to investigate how the reward functions affected the agents' learning capabilities in more complex and extensive environments. By scaling up the size of the warehouse, it could be observed how the agents adapted to larger and more challenging spaces. Understanding the impact of the reward functions in these settings was crucial for identifying which reward metric performed better in scaling up the complexity of the environment. This analysis would provide valuable insights into the generalizability and robustness of the reward functions, helping to determine their effectiveness in diverse warehouse scenarios.

### 4.3.3 Data Collection and Analysis

To gather data for the experiments, each training session was run for a predetermined number of episodes, and the agents' performance was recorded at each episode. The average episode rewards, convergence speed, and performance over the set number of episodes were collected for each experiment. The data obtained from multiple training sessions were then aggregated and analyzed to ensure statistical significance and reliability.

To ensure fair and unbiased comparisons, each experiment was repeated multiple times with different random seeds to account for the inherent randomness in reinforcement learning algorithms. This process helped to mitigate the impact of any random fluctuations and provided more robust and representative results.

In the data analysis phase, the results obtained from the experiments were carefully examined and visualized using various graphs and plots to facilitate comprehensive analysis and comparison. The use of graphs allowed for a more intuitive representation of the data, enabling researchers to identify trends, patterns, and significant differences between different reward functions and training methods.

# 5 Results, Analysis and Evaluation

In this section, a detailed analysis of the collected results will be performed, and the overall project evaluated. The focus will be on assessing the impact of different reward functions on the behaviour and performance of reinforcement learning agents in the multi-agent warehouse environment.

The analysis will involve comparing the performance metrics between the sparse and dense reward environments. We will investigate how each reward function influences the agents' learning behaviours, decision-making processes, exploration strategies, and convergence speed. Additionally, qualitative assessments of the agents' interactions and collaborative strategies within the multi-agent environment will provide valuable insights.

## 5.1 Independent Learning

The first test conducted in the evaluation phase focused on examining the impact of different reward functions on agents during their separate or independent training. In this experiment, each agent was trained individually, and their learning behaviors and performance were closely observed under both the sparse and dense reward environments.

To ensure a fair comparison, the agents were subjected to the same number of training episodes and initialized with three different random seeds to account for inherent randomness in reinforcement learning algorithms.

During the training process, the agents' actions and interactions with the warehouse environment were recorded, and their cumulative rewards for each episode were calculated. The average episode rewards for both sparse and dense reward environments were collected over multiple training sessions to establish a comprehensive performance comparison.

### 5.1.1 Sparse Rewards

The first test performed was using the sparse reward function to train the agent independently, in the context of the sparse reward environment, the maximum achievable score for an agent was eight. This score could be attained by successfully locating and delivering all four packages present in the warehouse, as each delivery would grant two points (one for locating the package and another for successful delivery). The objective for the agents was to maximize their cumulative rewards by successfully completing as many package deliveries as possible.

|  |  |
| --- | --- |
| A graph of a number of blue lines  Description automatically generated with medium confidence  **Figure 7 - Agent 1 Reward over Epoch** | A graph of blue and orange lines  Description automatically generated  **Figure 8 - Agent 2 Reward over Epoch** |

The results indicated that the agents took approximately 300 epochs to converge to a relatively stable policy. During this convergence period, the average episode rewards fluctuated, but they eventually settled at an average of 4 points per episode.

While the average episode rewards converged around 4 points, it was observed that the highest score achieved by any agent in the sparse reward environment was 6. This indicated that some agents were able to successfully locate and deliver three out of the four packages in the warehouse. However, the agents did not consistently achieve this higher score.

There is a trend of plateauing before making further progress suggesting that the agents might have encountered certain challenges or obstacles in their learning process. They could have been stuck in suboptimal strategies or local minima that prevented them from reaching the maximum achievable score of 8. However, it was encouraging to observe that the agents could overcome these challenges and eventually improve their performance after hitting a plateau.

Based on this trend, it is plausible to speculate that with more epochs of training, the agents might have the potential to achieve a full score of 8. The agents' learning process appeared to be incremental, and they learned to perform better over time. Given additional training epochs, they might have been able to further refine their strategies and explore the environment more effectively, leading to the successful delivery of all four packages.

It is essential to acknowledge that reinforcement learning algorithms' training process can often be time-consuming, and achieving optimal performance might require significant computational resources and training time. Therefore, while the results up to 500 epochs provided valuable insights, further experimentation and extended training periods could offer a more definitive understanding of the agents' ultimate performance in the sparse reward environment.

### 5.1.2 Dense Rewards

The next step in the comparative analysis was to assess the agents' performance under the dense reward function when trained independently. In this setting, the agents received more frequent and incremental rewards, providing them with more detailed feedback throughout the learning process. The maximum achievable reward in the dense reward environment was set at 40, indicating that agents had the potential to earn higher cumulative rewards compared to the sparse reward environment.

|  |  |
| --- | --- |
| A graph of different colored lines  Description automatically generated with medium confidence  **Figure 9 - Agent 1 Reward over Epoch** | A graph of different colored lines  Description automatically generated with medium confidence  **Figure 10 - Agent 2 Reward over Epoch** |

During the training process, the agents tended to converge around 400 epochs, indicating that they reached a relatively stable policy by that point. However, unlike the sparse reward environment, where the agents converged more consistently at a specific score, the dense reward environment showed higher volatility in the agents' learning trajectories. The agents' convergence patterns ranged across various scores, implying that the learning process was more varied and uncertain.

The highest reward achieved in the dense reward environment was 36, which indicated that the agents managed to pick up the last package but did not deliver it successfully to its goal state. This suggests that the agents were able to locate and pick up most of the packages but faced challenges in completing the final delivery successfully. The observation of a score that was close to the maximum reward of 40 indicated that the agents were proficient in handling multiple packages and had a high level of task completion.

The volatility observed in the agents' learning can be attributed to the nature of the dense reward function, which provided more frequent and detailed feedback. While this frequent feedback allowed the agents to learn more efficiently and make faster progress, it also introduced higher variance in their learning behavior. The agents might have encountered instances of local minima or suboptimal solutions, leading to fluctuations in their learning trajectory.

### 5.2.3 Evaluation

Upon comparison, it became apparent that the choice of reward structure had a profound impact on the agents' learning behaviors and convergence patterns. The sparse reward function, with its minimalistic feedback, encouraged faster convergence in the agents' learning process. By providing rewards only upon successful completion of critical events, such as picking up and delivering a package, the sparse reward function offered a clear and straightforward signal for the agents to follow. As a result, the agents quickly learned to optimize their actions to achieve these specific goals.

In contrast, the dense reward function, with its more frequent and informative feedback, enabled higher overall performance in the agents' learning process. The dense reward function rewarded agents for incremental progress, such as getting closer to the goal or taking steps in the right direction, in addition to rewarding successful package deliveries. This continuous feedback allowed the agents to make rapid advancements in their learning, exploring more possibilities and refining their strategies with each episode.

While the dense reward function exhibited the advantage of promoting higher overall performance, it also introduced challenges related to volatility and potential suboptimal outcomes. The frequent rewards and incremental feedback led to higher variance in the agents' learning trajectories. This volatility could be attributed to the agents' encounters with local minima or suboptimal solutions, which influenced their learning behavior. The agents' performance under the dense reward function showed a wider range of scores, reflecting the fluctuations in their learning process.

In conclusion, the trade-off between the two reward functions lies in the balance between faster convergence and higher overall performance. This comparison of the two reward functions highlights the importance of carefully designing reward structures to suit the specific goals and complexities of the environment. Both the sparse and dense reward functions demonstrated unique strengths and limitations in influencing the agents' learning behaviors and convergence patterns. Understanding the implications of these reward choices is essential for developing efficient and effective RL agents in dynamic and complex environments.

## 5.2 Combined Learning

To assess the effectiveness of the reward functions in a multi-agent environment, a test was conducted to evaluate the agents' performance when trained simultaneously. This test aimed to emulate real-world scenarios where multiple agents need to work collaboratively to accomplish tasks in a cooperative setting.

During the simultaneous training, both agents operated in the warehouse environment at the same time, interacting with each other and sharing the same space. The reward functions played a pivotal role in shaping the agents' behaviors and fostering cooperation between them.

### 5.2.1 Sparse Rewards

The first reward function evaluated in this test was the sparse reward metric. As previously described, the sparse reward function provided minimal feedback to the agents, rewarding them only when they successfully picked up and delivered a package.

|  |  |
| --- | --- |
| A graph of blue and orange lines  Description automatically generated  **Figure 11 - Combined Reward over Epoch** | A graph of blue and orange lines  Description automatically generated  **Figure 12 - Combined Reward over Epoch** |

During the evaluation the agents faced significant challenges in collaborating and working together effectively. As previously mentioned, the sparse reward function provided limited feedback to the agents, rewarding them only upon successful package pickups and deliveries. This minimalist approach aimed to encourage the agents to explore and learn autonomously, without explicit guidance.

However, the results revealed that the agents struggled to coordinate their actions and lacked cooperation. On one of the tested seeds, one agent learned that it was more advantageous to avoid the other agent entirely, presumably to prevent collisions and unnecessary competition over packages. This individualistic approach hindered effective collaboration, leading to suboptimal outcomes, and reduced overall performance.

The implementation of negative rewards in a previous version had also exhibited similar trends, where the agents learned to avoid certain actions that would incur penalties. In this sparse reward setting, the agents seemed to employ a similar strategy, but without the presence of negative rewards, their learning process became less focused and less efficient.

Additionally, in this same seed, the other agent's performance was not consistently strong during the initial 350 epochs. While one agent learned to avoid the other, the second agent did not achieve remarkable performance during this period. However, it is important to note that in the later epochs, the second agent managed to deliver all the packages successfully. This observation suggests that with more training and exploration, the second agent might improve its performance and potentially complete the given tasks.

During the evaluation using the second seed, it was observed that both agents did learn to participate in the warehouse task together, indicating some degree of collaboration. However, their performance was still suboptimal compared to when they were trained individually. While there were occasional episodes where both agents performed exceptionally well, there was never a point during training where both agents consistently achieved high performance simultaneously.

The results obtained from the evaluation of the sparse reward function underscored the challenges of fostering cooperation and effective coordination among agents in a multi-agent environment. The limited feedback posed difficulties in shaping their behaviors, resulting in individualistic decision-making and suboptimal collaboration.

### 5.2.2 Dense Rewards

|  |  |
| --- | --- |
| A graph of blue and orange lines  Description automatically generated  **Figure 13 - Combined Reward over Epoch** | A graph of a number of blue and orange lines  Description automatically generated with medium confidence  **Figure 14 - Combined Reward over Epoch** |

As a result of the dense reward function, the agents showed a greater inclination to work together and coordinate their actions. They managed to deliver packages together, albeit not with the same level of efficiency as if they were trained independently. This collaborative behavior indicated that the dense rewards facilitated better cooperation and teamwork among the agents, allowing them to accomplish certain tasks collectively.

However, it was challenging to precisely determine the distribution of package deliveries between the two agents. The dense reward function rewarded the agents regardless of whether they delivered the packages or not, leading to a situation where both agents could receive points even if one of them completed most of the deliveries. Consequently, this made it difficult to assess the individual contributions of each agent to the task.

While the dense reward function fostered more cooperative behavior, it also presented complexities related to the agents' ability to maximize task efficiency. The distribution of rewards and the lack of clarity in individual contributions raised questions about the optimal allocation of tasks and how to effectively incentivize agents to work together for the collective benefit.

### 5.2.3 Evaluation

Upon analysis, it became evident that the sparse reward function was not as suitable for training the drones simultaneously in a multi-agent environment. The minimal feedback provided by the sparse rewards hindered the agents' ability to effectively collaborate and coordinate their actions.

However, it is worth noting that the sparse reward function exhibited faster convergence compared to the dense reward function in an independent setting. This suggests that the sparse reward function may still be viable for individual training of the agents, allowing them to learn effectively in their respective environments without interference from other agents.

On the other hand, the dense reward function, while slower to converge, demonstrated potential for training multiple agents simultaneously. The dense rewards encouraged collaboration and teamwork among the agents, leading to instances where they managed to deliver packages together. This cooperative behavior indicates that the dense rewards can foster a more coordinated approach to completing tasks in a multi-agent setting.

Despite the promising collaboration observed with the dense rewards, there were challenges encountered for both functions which would suggest that, despite prior assumptions, training the agents competitively may provide better results.

## 5.3 Larger Map

Finally, the scalability of each reward function was examined by conducting tests on a larger map. This step aimed to assess how well the agents' performance could generalize to more extensive and complex environments, shedding light on the reward functions' effectiveness in accommodating increased scale and variability.

### 5.3.1 Sparse Rewards

|  |  |
| --- | --- |
| A graph of a graph  Description automatically generated with medium confidence  **Figure 15 - Agent 1 Reward over Epoch** | A graph of different types of lines  Description automatically generated with medium confidence  **Figure 16 - Agent 2 Reward over Epoch** |

In the evaluation of the sparse reward function on the larger map, notable challenges emerged during the initial stages of training. The agents exhibited slower progress compared to their performance in the smaller map environment. While they managed to deliver up to three packages in the smaller map within the same timeframe, on the larger map, they struggled to complete even a single delivery efficiently.

The graphs depicting the agents' performance under the sparse reward function on the larger map exhibited a different trend compared to the smaller map. The agents' learning curve appeared to be slower, with a gradual increase in rewards over the training epochs. The agents seemed to be facing difficulties in finding the optimal paths and navigation strategies in the more extensive and complex environment. The lack of immediate feedback in the sparse reward system proved to be a challenge, as it took the agents longer to identify and learn the correct actions that would lead to successful deliveries.

However, despite the slower initial progress, there were signs of improvement in the agents' performance as the number of training episodes increased. The graphs indicated a potential upward trend in rewards, suggesting that with more episodes, the agents could gradually improve their learning and delivery capabilities on the larger map. This implied that the sparse reward function might eventually yield favorable results with additional training time, as the agents gained more experience in navigating and exploring the extended warehouse space.

### 5.3.2 Dense Rewards

|  |  |
| --- | --- |
| A graph of different colored lines  Description automatically generated with medium confidence  **Figure 17 - Agent 1 Reward over Epoch** | A graph of blue and orange lines  Description automatically generated  **Figure 18 - Agent 2 Reward over Epoch** |

In contrast to the sparse reward function, the evaluation of the dense reward function on the larger map revealed a more promising performance trend. Despite the larger and more complex environment, the dense rewards appeared to enable the agents to maintain a comparable level of performance to what was achieved on the smaller maps. For instance, Agent One on Seed One demonstrated consistent and high rewards, delivering multiple packages efficiently even on the larger map. This indicated that the dense reward function provided more informative and frequent feedback to guide the agents, allowing them to navigate and explore the extended warehouse space more effectively.

However, despite the overall encouraging performance, the dense reward function also exhibited a common challenge – volatility. The agents' performance was observed to be more erratic and subject to fluctuations, with episodes varying significantly in terms of rewards and task completion. While Agent One on Seed One showcased remarkable performance, there were instances where the agents' rewards dropped, indicating suboptimal decision-making and less efficient deliveries.

Despite this challenge, the dense reward function showed promise in adapting to larger environments and facilitating efficient task completion. Its ability to maintain relatively high rewards and deliver multiple packages on the larger map indicated its potential scalability.

It is important to note that as the map size increases so does the potential to gain rewards, therefore the maximum possible reward for these results is higher than on the smaller map.

### 5.3.3 Evaluation

In conclusion, the experimental results have shed light on the distinct characteristics and performances of the sparse and dense reward functions in different warehouse environments. The sparse reward function demonstrated its effectiveness in training agents in smaller warehouses with fewer agents. Its faster convergence and ability to deliver satisfactory results in simpler settings made it a suitable choice for scenarios where the scale and complexity of the environment are relatively modest.

On the other hand, the dense reward function emerged as a more robust and adaptable choice, particularly in larger warehouse environments with multiple agents. While it showed slower initial progress, it eventually demonstrated its capacity to handle the increased complexity and facilitate efficient task completion even on more extensive maps. The dense reward function's ability to scale effectively with the size of the environment and accommodate multiple agents working collaboratively proved to be its key advantage.

The results indicate that the choice between the two reward functions should be influenced by the specific requirements and characteristics of the warehouse environment being addressed.

It is important to note that these findings represent a step forward in understanding the impact of reward functions on multi-agent warehouse environments. However, the research also reveals areas for further exploration and improvement. Fine-tuning the parameters of each reward function, exploring novel reward shaping techniques, and experimenting with different warehouse layouts and agent configurations could potentially yield even more optimized and effective results.

# 6 Legal, Social, Ethical and Professional Issues

Legal, social, ethical, and professional issues play a critical role in the context of any computer science project, including the one undertaken for this research. As a responsible researcher, it is essential to consider the impact of the project on various stakeholders and adhere to the codes of conduct issued by reputable professional bodies.

## Legal Considerations

Throughout the implementation of this research project, a meticulous approach was adopted to ensure strict compliance with all relevant laws and regulations. considerations regarding intellectual property rights were diligently addressed. When utilizing third-party algorithms or software libraries, careful attention was given to licensing agreements and intellectual property ownership. Proper documentation and attribution were maintained, respecting the intellectual property rights of others, and avoiding any unauthorized use or infringement.

## Social and Ethical Implications

Warehouse automation and reinforcement learning have profound social implications, particularly in the context of labour and employment. As autonomous agents become more prevalent in warehouse operations, there may be concerns about the potential displacement of human workers. Ethical considerations come into play in ensuring that the implementation of autonomous agents aligns with principles of fairness, equity, and inclusivity. Transparency in decision-making processes is essential to avoid bias or discriminatory outcomes. Moreover, responsible, and ethical deployment of autonomous agents should consider the impact on vulnerable populations, ensuring that no harm is inflicted on individuals or communities.

## Professionalism and Codes of Conduct

Throughout the project, the principles outlined in the Code of Conduct issued by the British Computer Society (BCS) were upheld. Integrity, objectivity, and professional competence in all aspects of the research were maintained with the upmost importance. Transparency in reporting methodologies, results, and any limitations or challenges encountered during the project was also practiced to the highest degree possible. Adhering to these professional standards promotes trust and credibility in the research outcomes.

## Public Well-being and Security

The successful implementation of warehouse automation can have significant implications for the public, including improved supply chain efficiency and customer service. However, ensuring the responsible deployment of autonomous agents is vital to safeguard public well-being. This entails rigorous testing and validation of the algorithms to minimize the risk of errors or malfunction. Additionally, it is crucial to address cyber security concerns to protect against potential threats or attacks that could compromise the safety and security of the warehouse system and data.

# Conclusion

## 7.1 Achievements

Throughout the project development, all functional and non-functional requirements were successfully achieved, resulting in a robust and functional system. The implemented solution met the initial project objectives, enabling RL agents to navigate a warehouse environment, pick up packages, and deliver them to their designated goal states. Additionally, the project successfully explored and compared the impact of different reward functions on the agents' learning behaviors and overall performance.

Despite the successful fulfillment of the specified requirements, certain features were not added or extensively tested during the project's execution. These features were left out due to prioritization, time constraints, or other considerations.

## 7.2 Limitations

Despite the thoroughness of the experiments and analysis, this research has certain limitations that should be acknowledged. Firstly, the complexity of real-world warehouse environments goes beyond the scope of the simulated environments used in this study. The simulated environments lack certain real-world factors, such as dynamic changes in warehouse layouts, uncertainties in package sizes and weights, and variations in environmental conditions. As such, the findings may not fully capture the challenges that would be encountered in real-world warehouse automation scenarios.

Secondly, the choice of reward functions, though comprehensive, is not exhaustive. Different combinations of reward structures and hyperparameters may lead to varying results and performance outcomes. Exploring a wider range of reward functions and tuning strategies could provide additional insights into optimizing RL agents' performance in multi-agent warehouse settings.

Additionally, the success of reinforcement learning algorithms heavily depends on the quality and quantity of training data. In this research, training was conducted within a limited number of episodes due to computational constraints. Further research could explore longer training sessions to investigate how the agents' performance evolves over extended periods and assess the impact of more extensive training on their learning capabilities.

## Future Work

This research serves as a valuable steppingstone towards the application of reinforcement learning in multi-agent warehouse automation. Building upon the insights gained from this study, future research could focus on several areas for further exploration and improvement.

One avenue for future work involves the development of more complex and dynamic warehouse environments that better mimic real-world scenarios. This could involve introducing dynamic elements such as changing layouts, varying package types and sizes, and environmental factors like traffic and congestion. Additionally, investigating the integration of real-time data from warehouse operations could provide a more realistic and dynamic training environment, allowing RL agents to adapt and optimize their strategies based on up-to-date information.

Another potential area of future research is exploring different reward shaping techniques. The choice of reward function plays a pivotal role in shaping the agents' behaviour and learning process. Investigating alternative reward shaping methods could lead to more efficient and effective training, further enhancing the agents' ability to complete warehouse tasks.

Furthermore, extending the study to include a broader range of multi-agent algorithms and approaches could provide valuable insights into the comparative strengths and weaknesses of different RL algorithms in multi-agent warehouse settings. Comparing the performance of model-free and model-based RL algorithms, as well as exploring the application of deep multi-agent reinforcement learning methods, could shed light on the most suitable approaches for different warehouse automation tasks.

Overall, this research opens numerous avenues for future exploration in the field of multi-agent warehouse automation. By addressing the identified limitations and building upon the foundational findings, future work in this area could lead to significant advancements in optimizing warehouse operations and contributing to the broader field of AI and autonomous systems in logistics and beyond.

# **8 References**

Black, B., Grammel, N., Terry, J., Jayakumar, M., Hari, A., Sullivan, R., . . . Ravi, P. (2021, October 26). *PettingZoo: Gym for Multi-Agent Reinforcement Learning*. Retrieved from arXiv.org: https://arxiv.org/abs/2009.14471

Brockman, G., Cheung, V., Pettersson, L., Schneider, J., Schulman, J., Tang, J., & Zaremba, W. (2016). *OpenAI Gym*. Retrieved from arXiv.org: https://arxiv.org/abs/1606.01540

Farsang, M., & Szegletes, L. (2021, June 28). *Importance of Environment Design in Reinforcement Learning: A Study of a Robotic Environment*. Retrieved from arXiv.org: https://arxiv.org/abs/2102.10447

Filippos, C., & Schäfer, L. (2023, May 24). *RWARE A multi-agent reinforcement learning environment*. Retrieved from Github: https://github.com/semitable/robotic-warehouse

*Gymnasium Documentation*. (n.d.). Retrieved from gymnasium.farama.org: https://gymnasium.farama.org/

Lee, H., & Jeong, J. (2021, January 28). Mobile Robot Path Optimization Technique Based on Reinforcement Learning Algorithm in Warehouse Environment. *Applied Sciences, 11*(3), 1209. doi:10.3390/app11031209

Mnih, V., Kavukcuoglu, K., Silver, D., Graves, A., Antonoglou, I., Wierstra, D., & Riedmiller, M. (2013). *Playing Atari with Deep Reinforcement Learning*. Retrieved from arXiv.org: https://arxiv.org/abs/1312.5602

Peyas, I. S., Hasan, Z., Md. Rafat, R., Musabbir, A., Raisa, A. M., & Siddique, S. (2021, December 1). *Autonomous Warehouse Robot using Deep Q-Learning*. Retrieved from IEEE Xplore: https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=9707256

*Welcome to Tianshou! — Tianshou 0.5.1 documentation*. (n.d.). Retrieved from tianshou.readthedocs.io: https://tianshou.readthedocs.io/en/master/index.html

# 9 Appendix

## 9.1 Appendix A: Policy Trainer

1. **import** os
2. **from** typing **import** Optional, Tuple
4. **import** gymnasium as gym
5. **import** numpy as np
6. **import** torch
7. **from** torch **import** nn
9. **from** tianshou.data **import** Collector, VectorReplayBuffer
10. **from** tianshou.env **import** DummyVectorEnv
11. **from** tianshou.env.pettingzoo\_env **import** PettingZooEnv
12. **from** tianshou.policy **import** BasePolicy, DQNPolicy, MultiAgentPolicyManager, RandomPolicy
13. **from** tianshou.trainer **import** offpolicy\_trainer
15. **import** matplotlib.pyplot as plt
17. **import** DroneMA\_Dense\_Load
18. **import** DroneMA\_Sparse\_Load
20. os.environ["KMP\_DUPLICATE\_LIB\_OK"]**=**"TRUE"
21. rewardsPlot **=** []
22. rewardsPlot2 **=** []
24. **class** Net(nn.Module):
25. **def** \_\_init\_\_(self, state\_shape, action\_shape):
26. super().\_\_init\_\_()
27. self.model **=** nn.Sequential(
28. nn.Linear(np.prod(state\_shape), 128), nn.LeakyReLU(negative\_slope**=**0.01, inplace**=**False),
29. nn.Linear(128, 128), nn.LeakyReLU(negative\_slope**=**0.01, inplace**=**False),
30. nn.Linear(128, 128), nn.LeakyReLU(negative\_slope**=**0.01, inplace**=**False),
31. nn.Linear(128, np.prod(action\_shape)),
32. )
34. **def** forward(self, obs, state**=**None, info**=**{}):
35. **if** **not** isinstance(obs, torch.Tensor):
36. obs **=** torch.tensor(obs, dtype**=**torch.float)
37. batch **=** obs.shape[0]
38. logits **=** self.model(obs.view(batch, **-**1))
39. **return** logits, state
41. **def** \_get\_agents(
42. agent\_1: Optional[BasePolicy] **=** None,
43. agent\_2: Optional[BasePolicy] **=** None,
44. optim: Optional[torch.optim.Optimizer] **=** None,
45. ) **-**> Tuple[BasePolicy, torch.optim.Optimizer, list]:
46. env **=** \_get\_env()
47. observation\_space **=** (
48. env.observation\_space["observation"]
49. **if** isinstance(env.observation\_space, gym.spaces.Dict)
50. **else** env.observation\_space
51. )
53. state\_shape**=** observation\_space.shape **or** observation\_space.n,
54. action\_shape**=**env.action\_space.shape **or** env.action\_space.n,
56. **if** agent\_1 **is** None:
57. net **=** Net(state\_shape, action\_shape)
58. optim **=** torch.optim.Adam(net.parameters(), lr**=**1e**-**3)
60. agent\_1 **=** DQNPolicy(
61. model**=**net,
62. optim**=**optim,
63. discount\_factor**=**0.9,
64. estimation\_step**=**3,
65. target\_update\_freq**=**320,
66. )
68. **if** agent\_2 **is** None:
69. net **=** Net(state\_shape, action\_shape)
70. optim **=** torch.optim.Adam(net.parameters(), lr**=**1e**-**3)
72. agent\_2 **=** DQNPolicy(
73. model**=**net,
74. optim**=**optim,
75. discount\_factor**=**0.9,
76. estimation\_step**=**3,
77. target\_update\_freq**=**320,
78. )
80. agents **=** [agent\_1] #, agent\_2]
81. policy **=** MultiAgentPolicyManager(agents, env)
82. **return** policy, optim, env.agents, agents

85. **def** \_get\_env():
86. **return** PettingZooEnv(DroneMA\_Sparse\_Load.env())

89. **if** \_\_name\_\_ **==** "\_\_main\_\_":
90. # ======== Step 1: Environment setup =========
91. train\_envs **=** DummyVectorEnv([\_get\_env **for** \_ **in** range(20)])
92. test\_envs **=** DummyVectorEnv([\_get\_env **for** \_ **in** range(10)])
94. # seed
95. seed **=** 2
96. np.random.seed(seed)
97. torch.manual\_seed(seed)
98. train\_envs.seed(seed)
99. test\_envs.seed(seed)
101. # ======== Step 2: Agent setup =========
102. policy, optim, agents, temp **=** \_get\_agents()
104. # ======== Step 3: Collector setup =========
105. train\_collector **=** Collector(
106. policy,
107. train\_envs,
108. VectorReplayBuffer(20000, len(train\_envs)),
109. exploration\_noise**=**True,
110. )
111. test\_collector **=** Collector(policy, test\_envs, exploration\_noise**=**True)
112. train\_collector.collect(n\_step**=**4000)
114. # ======== Step 4: Callback functions setup =========
115. **def** save\_best\_fn(policy):
116. torch.save(policy.policies[agents[0]].state\_dict(), "policy1\_Sparse\_Independant\_Large.pth")
117. #torch.save(policy.policies[agents[1]].state\_dict(), "policy2\_Dense\_Combined.pth")
119. **def** stop\_fn(mean\_rewards):
120. **return** mean\_rewards >**=** 100000000
122. **def** train\_fn(epoch, env\_step):
123. policy.policies[agents[0]].set\_eps(0.1)
124. #policy.policies[agents[1]].set\_eps(0.1)
126. **def** test\_fn(epoch, env\_step):
127. policy.policies[agents[0]].set\_eps(0.05)
128. #policy.policies[agents[1]].set\_eps(0.05)
130. **def** reward\_metric(rews):
131. agent1\_rewards **=** rews[:, 0]  # Rewards of agent 1
132. #agent2\_rewards = rews[:, 1]
133. #com\_rewards = [(x + y) for x, y in zip(agent1\_rewards, agent2\_rewards)]
134. **return** np.array(agent1\_rewards)
136. # ======== Step 5: Run the trainer =========
137. result **=** offpolicy\_trainer(
138. policy**=**policy,
139. train\_collector**=**train\_collector,
140. test\_collector**=**test\_collector,
141. max\_epoch**=**500,
142. step\_per\_epoch**=**200,
143. step\_per\_collect**=**40,
144. update\_per\_step**=**0.1,
145. episode\_per\_test**=**100,
146. batch\_size**=**128,
147. train\_fn**=**train\_fn,
148. test\_fn**=**test\_fn,
149. stop\_fn**=**stop\_fn,
150. save\_best\_fn**=**save\_best\_fn,
151. test\_in\_train**=**False,
152. reward\_metric**=**reward\_metric,
153. )
155. print(f"\n==========Result==========\n{result}")

## 9.2 Appendix B: Sparse Rewards

### 9.2.1 Appendix B.1: Environment

1. **import** math
3. **import** gymnasium
4. **import** numpy as np
5. **import** pygame
6. **from** gymnasium.spaces **import** Discrete
8. **from** pettingzoo **import** AECEnv
9. **from** pettingzoo.utils **import** agent\_selector, wrappers
10. **from** pettingzoo.utils.conversions **import** parallel\_wrapper\_fn
12. **def** find\_indices(list\_to\_check, item\_to\_find):
13. indices **=** []
14. **for** idx, value **in** enumerate(list\_to\_check):
15. **if** value **==** item\_to\_find:
16. indices.append(idx)
17. **return** indices
19. **def** calc\_dist(currentState, goalState, mapWidth):
20. **return** (abs(math.floor(goalState**/**mapWidth) **-** math.floor(currentState**/**mapWidth)) **+** abs((goalState**%**mapWidth) **-** (currentState**%**mapWidth)))
22. **def** env(**\*\***kwargs):
23. env **=** raw\_env(**\*\***kwargs)
24. env **=** wrappers.AssertOutOfBoundsWrapper(env)
25. env **=** wrappers.OrderEnforcingWrapper(env)
26. **return** env

29. parallel\_env **=** parallel\_wrapper\_fn(env)
31. **class** raw\_env(AECEnv):
32. metadata **=** {
33. "render\_modes": ["human", "rgb\_array"],
34. "name": "droneMA\_v0",
35. "is\_parallelizable": True,
36. "render\_fps": 10,
37. }
39. **def** \_\_init\_\_(self, render\_mode**=**None):
41. # Map Key
42. # 0: Empty Space
43. # 1: Start
44. # 2: Goal
45. # 3: Wall
46. # 4: Full Shelf
47. # 5: Empty Shelf
49. # self.Map = [
50. #             1,0,0,0,0,0,0,1,
51. #             0,0,0,0,0,0,0,0,
52. #             0,0,4,0,0,4,0,0,
53. #             0,0,0,0,0,0,0,0,
54. #             0,0,0,0,0,0,0,0,
55. #             0,0,4,0,0,4,0,0,
56. #             0,0,0,0,0,0,0,0,
57. #             0,0,0,2,2,0,0,0
58. #             ]
60. self.Map **=** [
61. 1,0,0,0,0,1,
62. 0,0,0,0,0,0,
63. 0,0,4,4,0,0,
64. 0,0,4,4,0,0,
65. 0,0,0,0,0,0,
66. 0,0,2,2,0,0
67. ]
68. self.MapWidth **=** 6
69. self.Delivered **=** 0
70. self.screen **=** None
71. self.clock **=** None
72. self.width, self.height **=** 100 **\*** self.MapWidth, 100 **\*** len(self.Map)**/**self.MapWidth
74. self.depots **=** find\_indices(self.Map, 4)
76. self.currentReward **=** 0
78. self.agents **=** ["drone\_" **+** str(r) **for** r **in** range(1)]
79. self.possible\_agents **=** self.agents[:]
80. self.agent\_name\_mapping **=** dict(zip(self.agents, list(range(self.num\_agents))))
81. self.start **=** find\_indices(self.Map, 1)
82. self.state **=** find\_indices(self.Map, 1)
83. self.BatteryLife **=** [100 **for** agent **in** self.agents]
84. self.Packages **=** [**-**1 **for** i **in** find\_indices(self.Map, 4)]
85. self.PackageStates **=** [i **for** i **in** self.depots]
87. self.action\_spaces **=** {agent: Discrete(4) **for** agent **in** self.agents}
88. self.observation\_spaces **=** {
89. agent: Discrete(len(self.Map) **+** 2) **for** agent **in** self.agents
90. }
92. self.render\_mode **=** render\_mode
93. self.screen **=** None
95. self.reinit()
97. **def** observation\_space(self, agent):
98. **return** self.observation\_spaces[agent]
100. **def** action\_space(self, agent):
101. **return** self.action\_spaces[agent]
103. **def** reinit(self):
104. # self.Map = [
105. #             1,0,0,0,0,0,0,1,
106. #             0,0,0,0,0,0,0,0,
107. #             0,0,4,0,0,4,0,0,
108. #             0,0,0,0,0,0,0,0,
109. #             0,0,0,0,0,0,0,0,
110. #             0,0,4,0,0,4,0,0,
111. #             0,0,0,0,0,0,0,0,
112. #             0,0,0,2,2,0,0,0
113. #             ]
115. self.Map **=** [
116. 1,0,0,0,0,1,
117. 0,0,0,0,0,0,
118. 0,0,4,4,0,0,
119. 0,0,4,4,0,0,
120. 0,0,0,0,0,0,
121. 0,0,2,2,0,0
122. ]
124. self.MapWidth **=** 6
125. self.agents **=** self.possible\_agents[:]
126. self.\_agent\_selector **=** agent\_selector(self.agents)
127. self.agent\_selection **=** self.\_agent\_selector.reset()
128. self.rewards **=** {agent: 0 **for** agent **in** self.agents}
129. self.\_cumulative\_rewards **=** {agent: 0 **for** agent **in** self.agents}
130. self.terminations **=** {agent: False **for** agent **in** self.agents}
131. self.truncations **=** {agent: False **for** agent **in** self.agents}
132. self.infos **=** {agent: {} **for** agent **in** self.agents}
133. self.BatteryLife **=** [100 **for** agent **in** self.agents]
135. self.state **=** list(reversed(find\_indices(self.Map, 1)))
136. self.Packages **=** [**-**1 **for** i **in** find\_indices(self.Map, 4)]
137. self.PackageStates **=** [i **for** i **in** self.depots]
138. self.Delivered **=** 0
140. **def** render(self):
141. **if** self.render\_mode **is** None:
142. gymnasium.logger.warn(
143. "You are calling render method without specifying any render mode."
144. )
145. self.render\_mode **=** "human"
147. **if** self.screen **is** None:
148. pygame.init()
149. pygame.display.init()
150. self.screen **=** pygame.display.set\_mode((self.width, self.height))
151. pygame.display.set\_caption("Drones")
153. **if** self.clock **is** None:
154. self.clock **=** pygame.time.Clock()
156. BLACK **=** (0, 0, 0)
157. WHITE **=** (255, 255, 255)
158. GREEN **=** (0, 255, 0)
159. BLUE **=** (0, 0, 255)
160. GREY **=** (128, 128, 128)
162. image **=** pygame.image.load("drone.png").convert\_alpha()
163. boxImage **=** pygame.image.load("CardboardBox.png").convert()
165. image\_width, image\_height **=** 90, 90
166. image **=** pygame.transform.scale(image, (image\_width, image\_height))
168. self.screen.fill(WHITE)
170. square\_size **=** self.width **//** self.MapWidth
172. **for** i **in** range(self.MapWidth **+** 1):
173. pygame.draw.line(self.screen, BLACK, (0, i **\*** square\_size), (self.width, i **\*** square\_size))
174. pygame.draw.line(self.screen, BLACK, (i **\*** square\_size, 0), (i **\*** square\_size, self.height))

177. deliverSpot **=** find\_indices(self.Map, 2)
178. **for** ele **in** deliverSpot:
179. pygame.draw.rect(self.screen, GREEN, ((ele **%** self.MapWidth) **\*** square\_size, math.floor(ele **/** self.MapWidth) **\*** square\_size, square\_size, square\_size))
181. **for** ele **in** self.depots:
182. pygame.draw.rect(self.screen, BLUE, ((ele **%** self.MapWidth) **\*** square\_size, math.floor(ele **/** self.MapWidth)**\*** square\_size, square\_size, square\_size))
184. startSpot **=** find\_indices(self.Map, 1)
185. **for** ele **in** startSpot:
186. pygame.draw.rect(self.screen, GREY, ((ele **%** self.MapWidth) **\*** square\_size, math.floor(ele **/** self.MapWidth) **\*** square\_size, square\_size, square\_size))
188. droneX **=** [5 **+** (int(i) **%** self.MapWidth) **\*** square\_size **for** i **in** self.state]
189. droneY **=** [5 **+** math.floor(int(int(i)) **/** self.MapWidth) **\*** square\_size **for** i **in** self.state]
191. **for** i **in** self.Packages:
192. **if** i < 0:
193. packageLocation **=** find\_indices(self.Map, 4)
194. **for** i **in** packageLocation:
195. package\_size **=** 60
196. boxImage **=** pygame.transform.scale(boxImage, (60, 60))
197. package\_x **=** 20 **+** (i **%** self.MapWidth) **\*** square\_size
198. package\_y **=** 20 **+** math.floor(i **/** self.MapWidth) **\*** square\_size
200. pygame.draw.rect(self.screen, BLACK, (package\_x **-** 2, package\_y **-** 2, package\_size **+** 4, package\_size **+** 4), 0)
201. self.screen.blit(boxImage, (package\_x, package\_y))
203. **for** i **in** self.Packages:
204. **if** i >**=** 0:
205. package\_size **=** 80
206. boxImage **=** pygame.transform.scale(boxImage, (80, 80))
207. package\_x **=** droneX[i] **+** 5
208. package\_y **=** droneY[i] **+** 5
210. pygame.draw.rect(self.screen, BLACK, (package\_x **-** 2, package\_y **-** 2, package\_size **+** 4, package\_size **+** 4), 0)
211. self.screen.blit(boxImage, (package\_x, package\_y))
212. **for** i **in** range(len(self.state)):
213. self.screen.blit(image, (droneX[i], droneY[i]))
215. # Update the display
216. pygame.event.pump()
217. self.clock.tick(60)
218. pygame.display.flip()
220. **def** observe(self, agent):
221. **if** self.agents.index(agent) **in** self.Packages:
222. package **=** [1]
223. **else**:
224. package **=** [0]
226. my\_list **=** self.Map **+** [self.state[self.agents.index(agent)]] **+** package
228. **return** np.array(my\_list, dtype**=**np.int64)
230. **def** close(self):
231. pygame.display.quit()
232. pygame.quit()
234. **def** reset(self, seed**=**None, options**=**None):
235. self.reinit()
237. **def** step(self, action):
238. reward **=** 0
240. **if** (
241. self.terminations[self.agent\_selection]
242. **or** self.truncations[self.agent\_selection]
243. ):
244. self.\_was\_dead\_step(action)
245. **return**
247. agent **=** self.agent\_selection
249. inx **=** self.agents.index(agent)
250. **if** action **==** 0:
251. **if** self.state[inx] > self.MapWidth**-**1:
252. **if** self.Map[int(self.state[inx]) **-** self.MapWidth] !**=** 3:
253. **if** int(self.state[inx]) **-** self.MapWidth **not** **in** self.state:
254. self.state[inx] **-=** self.MapWidth
256. **if** action **==** 1:
257. **if** self.state[inx] **%** self.MapWidth !**=** self.MapWidth **-**1:
258. **if** self.Map[int(self.state[inx]) **+** 1] !**=** 3:
259. **if** int(self.state[inx]) **+** 1 **not** **in** self.state:
260. self.state[inx] **+=** 1
262. **if** action **==** 2:
263. **if** self.state[inx] < len(self.Map) **-** self.MapWidth**-**1:
264. **if** self.Map[int(self.state[inx]) **+** self.MapWidth] !**=** 3:
265. **if** int(self.state[inx]) **+** self.MapWidth **not** **in** self.state:
266. self.state[inx] **+=** self.MapWidth
268. **if** action **==** 3:
269. **if** self.state[inx] **%** self.MapWidth !**=** 0:
270. **if** self.Map[int(self.state[inx]) **-** 1] !**=** 3:
271. **if** int(self.state[inx]) **-** 1 **not** **in** self.state:
272. self.state[inx] **-=** 1
274. **if** inx **not** **in** self.Packages:
275. **if** self.Map[int(self.state[inx])] **==** 4:
276. self.Map[int(self.state[inx])] **=** 5
278. package **=** self.PackageStates.index(self.state[inx])

281. self.Packages[package] **=** inx
282. reward **=** 1
284. **else**:
285. **if** self.Map[int(self.state[inx])] **==** 2:
287. package **=** self.Packages.index(inx)
289. self.Packages[package] **=** **-**2
290. reward **=** 1
291. self.Delivered **+=** 1
293. self.BatteryLife[inx] **-=** 1
295. **if** self.BatteryLife[inx] <**=** 0:
296. self.terminations[agent] **=** True
298. **if** self.Delivered **==** 4:
299. self.terminations[agent] **=** True
301. self.rewards[self.agent\_selection] **+=** reward
303. info **=** {}
305. self.agent\_selection **=** self.\_agent\_selector.next()
306. **if** self.render\_mode **==** "human":
307. self.render()

### 9.2.2 Appendix B.2: Loader

1. **from** DroneMA\_Sparse **import** env, raw\_env, parallel\_env
3. \_\_all\_\_ **=** ["env", "raw\_env", "parallel\_env"]

## 9.3 Appendix C: Dense Rewards

### 9.3.1 Appendix C.1: Environment

1. **import** math
3. **import** gymnasium
4. **import** numpy as np
5. **import** pygame
6. **from** gymnasium.spaces **import** Discrete
8. **from** pettingzoo **import** AECEnv
9. **from** pettingzoo.utils **import** agent\_selector, wrappers
10. **from** pettingzoo.utils.conversions **import** parallel\_wrapper\_fn
12. **def** find\_indices(list\_to\_check, item\_to\_find):
13. indices **=** []
14. **for** idx, value **in** enumerate(list\_to\_check):
15. **if** value **==** item\_to\_find:
16. indices.append(idx)
17. **return** indices
19. **def** calc\_dist(currentState, goalState, mapWidth):
20. **return** (abs(math.floor(goalState**/**mapWidth) **-** math.floor(currentState**/**mapWidth)) **+** abs((goalState**%**mapWidth) **-** (currentState**%**mapWidth)))
22. **def** env(**\*\***kwargs):
23. env **=** raw\_env(**\*\***kwargs)
24. env **=** wrappers.AssertOutOfBoundsWrapper(env)
25. env **=** wrappers.OrderEnforcingWrapper(env)
26. **return** env

29. parallel\_env **=** parallel\_wrapper\_fn(env)
31. **class** raw\_env(AECEnv):
32. metadata **=** {
33. "render\_modes": ["human", "rgb\_array"],
34. "name": "droneMA\_v0",
35. "is\_parallelizable": True,
36. "render\_fps": 10,
37. }
39. **def** \_\_init\_\_(self, render\_mode**=**None):
41. # Map Key
42. # 0: Empty Space
43. # 1: Start
44. # 2: Goal
45. # 3: Wall
46. # 4: Full Shelf
47. # 5: Empty Shelf
49. # self.Map = [
50. #             1,0,0,0,0,1,
51. #             0,0,0,0,0,0,
52. #             0,0,4,4,0,0,
53. #             0,0,4,4,0,0,
54. #             0,0,0,0,0,0,
55. #             0,0,2,2,0,0
56. #             ]
58. self.Map **=** [
59. 1,0,0,0,0,0,0,1,
60. 0,0,0,0,0,0,0,0,
61. 0,0,4,0,0,4,0,0,
62. 0,0,0,0,0,0,0,0,
63. 0,0,0,0,0,0,0,0,
64. 0,0,4,0,0,4,0,0,
65. 0,0,0,0,0,0,0,0,
66. 0,0,0,2,2,0,0,0
67. ]
68. self.MapWidth **=** 8
69. self.Delivered **=** 0
70. self.screen **=** None
71. self.clock **=** None
72. self.width, self.height **=** 100 **\*** self.MapWidth, 100 **\*** len(self.Map)**/**self.MapWidth
74. self.depots **=** find\_indices(self.Map, 4)
76. self.currentReward **=** 0
78. self.agents **=** ["drone\_" **+** str(r) **for** r **in** range(1)]
79. self.possible\_agents **=** self.agents[:]
80. self.agent\_name\_mapping **=** dict(zip(self.agents, list(range(self.num\_agents))))
81. self.start **=** find\_indices(self.Map, 1)
82. self.state **=** find\_indices(self.Map, 1)
83. self.distanceFromGoal **=** []
84. **for** i **in** self.state:
85. self.distanceFromGoal.append(calc\_dist(i, self.depots[0], self.MapWidth))
86. self.BatteryLife **=** [200 **for** agent **in** self.agents]
87. self.Packages **=** [**-**1 **for** i **in** find\_indices(self.Map, 4)]
88. self.PackageStates **=** [i **for** i **in** self.depots]
90. # Action Space
91. # 0: Move Up
92. # 1: Move Right
93. # 2: Move Down
94. # 3: Move Left
95. # 4: Pick Up
96. # 5: Drop
97. # 6: Do nothing
99. self.action\_spaces **=** {agent: Discrete(4) **for** agent **in** self.agents}
100. self.observation\_spaces **=** {
101. agent: Discrete(len(self.Map) **+** 2) **for** agent **in** self.agents
102. }
104. self.render\_mode **=** render\_mode
105. self.screen **=** None
107. self.reinit()
109. **def** observation\_space(self, agent):
110. **return** self.observation\_spaces[agent]
112. **def** action\_space(self, agent):
113. **return** self.action\_spaces[agent]
115. **def** reinit(self):
117. self.Map **=** [
118. 1,0,0,0,0,1,
119. 0,0,0,0,0,0,
120. 0,0,4,4,0,0,
121. 0,0,4,4,0,0,
122. 0,0,0,0,0,0,
123. 0,0,2,2,0,0
124. ]
126. # self.Map = [
127. #             1,0,0,0,0,0,0,1,
128. #             0,0,0,0,0,0,0,0,
129. #             0,0,4,0,0,4,0,0,
130. #             0,0,0,0,0,0,0,0,
131. #             0,0,0,0,0,0,0,0,
132. #             0,0,4,0,0,4,0,0,
133. #             0,0,0,0,0,0,0,0,
134. #             0,0,0,2,2,0,0,0
135. #             ]
137. self.MapWidth **=** 8
138. self.agents **=** self.possible\_agents[:]
139. self.\_agent\_selector **=** agent\_selector(self.agents)
140. self.agent\_selection **=** self.\_agent\_selector.reset()
141. self.rewards **=** {agent: 0 **for** agent **in** self.agents}
142. self.\_cumulative\_rewards **=** {agent: 0 **for** agent **in** self.agents}
143. self.terminations **=** {agent: False **for** agent **in** self.agents}
144. self.truncations **=** {agent: False **for** agent **in** self.agents}
145. self.infos **=** {agent: {} **for** agent **in** self.agents}
146. self.BatteryLife **=** [200 **for** agent **in** self.agents]
148. self.state **=** list(reversed(find\_indices(self.Map, 1)))#  find\_indices(self.Map, 1)
149. self.distanceFromGoal **=** []
150. **for** i **in** self.state:
151. self.distanceFromGoal.append(calc\_dist(i, self.depots[0], self.MapWidth))
152. self.Packages **=** [**-**1 **for** i **in** find\_indices(self.Map, 4)]
153. self.PackageStates **=** [i **for** i **in** self.depots]
154. self.Delivered **=** 0
156. **def** render(self):
157. **if** self.render\_mode **is** None:
158. gymnasium.logger.warn(
159. "You are calling render method without specifying any render mode."
160. )
161. self.render\_mode **=** "human"
162. **if** self.screen **is** None:
163. pygame.init()
164. pygame.display.init()
165. self.screen **=** pygame.display.set\_mode((self.width, self.height))
166. pygame.display.set\_caption("Drones")
168. **if** self.clock **is** None:
169. self.clock **=** pygame.time.Clock()
171. BLACK **=** (0, 0, 0)
172. WHITE **=** (255, 255, 255)
173. GREEN **=** (0, 255, 0)
174. BLUE **=** (0, 0, 255)
175. GREY **=** (128, 128, 128)
177. image **=** pygame.image.load("drone.png").convert\_alpha()
178. boxImage **=** pygame.image.load("CardboardBox.png").convert()
180. image\_width, image\_height **=** 90, 90
181. image **=** pygame.transform.scale(image, (image\_width, image\_height))
183. self.screen.fill(WHITE)
185. square\_size **=** self.width **//** self.MapWidth
187. **for** i **in** range(self.MapWidth **+** 1):
188. pygame.draw.line(self.screen, BLACK, (0, i **\*** square\_size), (self.width, i **\*** square\_size))
189. pygame.draw.line(self.screen, BLACK, (i **\*** square\_size, 0), (i **\*** square\_size, self.height))

192. deliverSpot **=** find\_indices(self.Map, 2)
193. **for** ele **in** deliverSpot:
194. pygame.draw.rect(self.screen, GREEN, ((ele **%** self.MapWidth) **\*** square\_size, math.floor(ele **/** self.MapWidth) **\*** square\_size, square\_size, square\_size))
196. **for** ele **in** self.depots:
197. pygame.draw.rect(self.screen, BLUE, ((ele **%** self.MapWidth) **\*** square\_size, math.floor(ele **/** self.MapWidth)**\*** square\_size, square\_size, square\_size))
199. startSpot **=** find\_indices(self.Map, 1)
200. **for** ele **in** startSpot:
201. pygame.draw.rect(self.screen, GREY, ((ele **%** self.MapWidth) **\*** square\_size, math.floor(ele **/** self.MapWidth) **\*** square\_size, square\_size, square\_size))
203. droneX **=** [5 **+** (int(i) **%** self.MapWidth) **\*** square\_size **for** i **in** self.state]
204. droneY **=** [5 **+** math.floor(int(int(i)) **/** self.MapWidth) **\*** square\_size **for** i **in** self.state]
206. **for** i **in** self.Packages:
207. **if** i < 0:
208. packageLocation **=** find\_indices(self.Map, 4)
209. **for** i **in** packageLocation:
210. package\_size **=** 60
211. boxImage **=** pygame.transform.scale(boxImage, (60, 60))
212. package\_x **=** 20 **+** (i **%** self.MapWidth) **\*** square\_size
213. package\_y **=** 20 **+** math.floor(i **/** self.MapWidth) **\*** square\_size
215. pygame.draw.rect(self.screen, BLACK, (package\_x **-** 2, package\_y **-** 2, package\_size **+** 4, package\_size **+** 4), 0)
216. self.screen.blit(boxImage, (package\_x, package\_y))
218. **for** i **in** self.Packages:
219. **if** i >**=** 0:
220. package\_size **=** 80
221. boxImage **=** pygame.transform.scale(boxImage, (80, 80))
222. package\_x **=** droneX[i] **+** 5
223. package\_y **=** droneY[i] **+** 5
225. pygame.draw.rect(self.screen, BLACK, (package\_x **-** 2, package\_y **-** 2, package\_size **+** 4, package\_size **+** 4), 0)
226. self.screen.blit(boxImage, (package\_x, package\_y))
227. **for** i **in** range(len(self.state)):
228. self.screen.blit(image, (droneX[i], droneY[i]))
230. # Update the display
231. pygame.event.pump()
232. self.clock.tick(60)
233. pygame.display.flip()
235. **def** observe(self, agent):
236. **if** self.agents.index(agent) **in** self.Packages:
237. package **=** [1]
238. **else**:
239. package **=** [0]
241. my\_list **=** self.Map **+** [self.state[self.agents.index(agent)]] **+** package
243. **return** np.array(my\_list, dtype**=**np.int64)
245. **def** close(self):
246. pygame.display.quit()
247. pygame.quit()
249. **def** reset(self, seed**=**None, options**=**None):
250. self.reinit()
252. **def** step(self, action):
253. reward **=** 0
255. **if** (
256. self.terminations[self.agent\_selection]
257. **or** self.truncations[self.agent\_selection]
258. ):
259. self.\_was\_dead\_step(action)
260. **return**
262. agent **=** self.agent\_selection
264. inx **=** self.agents.index(agent)
266. **if** action **==** 0:
267. **if** self.state[inx] > self.MapWidth**-**1:
268. **if** self.Map[int(self.state[inx]) **-** self.MapWidth] !**=** 3:
269. **if** int(self.state[inx]) **-** self.MapWidth **not** **in** self.state:
270. self.state[inx] **-=** self.MapWidth
272. **if** action **==** 1:
273. **if** self.state[inx] **%** self.MapWidth !**=** self.MapWidth **-**1:
274. **if** self.Map[int(self.state[inx]) **+** 1] !**=** 3:
275. **if** int(self.state[inx]) **+** 1 **not** **in** self.state:
276. self.state[inx] **+=** 1
278. **if** action **==** 2:
279. **if** self.state[inx] < len(self.Map) **-** self.MapWidth**-**1:
280. **if** self.Map[int(self.state[inx]) **+** self.MapWidth] !**=** 3:
281. **if** int(self.state[inx]) **+** self.MapWidth **not** **in** self.state:
282. self.state[inx] **+=** self.MapWidth
284. **if** action **==** 3:
285. **if** self.state[inx] **%** self.MapWidth !**=** 0:
286. **if** self.Map[int(self.state[inx]) **-** 1] !**=** 3:
287. **if** int(self.state[inx]) **-** 1 **not** **in** self.state:
288. self.state[inx] **-=** 1
290. **if** inx **not** **in** self.Packages:
291. **if** self.Map[int(self.state[inx])] **==** 4:
292. self.Map[int(self.state[inx])] **=** 5
294. package **=** self.PackageStates.index(self.state[inx])

297. self.Packages[package] **=** inx
298. reward **=** 1
300. self.distanceFromGoal[inx] **=** [i **for** i, n **in** enumerate(self.Map) **if** n **==** 2][0]
302. **else**:
303. **if** self.Map[int(self.state[inx])] **==** 2:
305. package **=** self.Packages.index(inx)
307. self.Packages[package] **=** **-**2
308. reward **=** 1
309. self.Delivered **+=** 1
311. **if** 4 **in** self.Map:
312. self.distanceFromGoal[inx] **=** [i **for** i, n **in** enumerate(self.Map) **if** n **==** 4][0]
314. **if** inx **in** self.Packages:
315. goalState **=** [i **for** i, n **in** enumerate(self.Map) **if** n **==** 2][0]
316. **elif** 4 **in** self.Map:
317. goalState **=** [i **for** i, n **in** enumerate(self.Map) **if** n **==** 4][0]
318. **else**:
319. goalState **=** self.start[inx]
321. currentDistance **=** calc\_dist(self.state[inx], goalState, self.MapWidth)
323. **if** currentDistance < self.distanceFromGoal[inx]:
324. reward **=** 1
325. self.distanceFromGoal[inx] **=** currentDistance
327. self.BatteryLife[inx] **-=** 1
329. **if** self.BatteryLife[inx] <**=** 0:
330. self.terminations[agent] **=** True
332. **if** self.Delivered **==** 4:
333. self.terminations[agent] **=** True
335. self.rewards[self.agent\_selection] **+=** reward
337. info **=** {}
339. self.agent\_selection **=** self.\_agent\_selector.next()
340. **if** self.render\_mode **==** "human":
341. self.render()

### 9.3.2 Appendix C.2: Loader

1. **from** DroneMA **import** env, raw\_env, parallel\_env
3. \_\_all\_\_ **=** ["env", "raw\_env", "parallel\_env"]