SYSU FinalTerm Homework of Task B

Essay Reading

The reading section of the paper primarily focuses on sharing the information I have summarized while studying the paper titled "VMAS: A Vectorized Multi-Agent Simulator for Collective Robot Learning."This includes the motivation behind the paper, its innovative aspects, and the core algorithms presented. Since I need to reproduce the three algorithms—CPPO, MAPPO, and IPPO—in the subsequent sections, the core algorithm part will mainly explain the fundamental principles and relevant applications of these three algorithms as discussed in the paper.

Motivation

With the further development of robotic applications, there is an increasing demand for robots to master more skills and accomplish a wider range of tasks, such as path planning, autonomous obstacle avoidance, and area coverage. The complexity of real-world problems necessitates the collaboration of multiple robots to achieve goals more effectively. Relying on simple heuristic learning methods to address multi-robot cooperation issues tends to be a trade-off between learning speed and optimal results, often leading to unsatisfactory outcomes. In contrast, Multi-Agent Reinforcement Learning (MARL) is a more effective learning algorithm for finding optimal solutions. In this context, robots act as agents, acquiring information and experience through interaction with the environment, and employing deep neural networks as decision-making networks to learn optimized strategies related to reward functions.
However, current MARL algorithms still face some common issues. Compared to heuristic algorithms, the inherent computational complexity of MARL algorithms leads to significantly longer training and convergence times, which is a very apparent problem. This can adversely affect the practicality of MARL algorithms when dealing with large datasets and overly complex scenarios. If the computational requirements are excessively high, it can make it difficult for typical research organizations and individual scholars to continue advancing research related to MARL, forcing them to limit their work to relatively simple problems. This reflects the current state of using MARL algorithms, where the range of complete problem scenarios that can be effectively addressed is overly restricted. As a result, many real-world factors that could influence multi-robot decision-making are sacrificed, leading to an overly idealized representation of the problem scenarios that lack meaningful relevance to real-world tasks.

Benefits

Due to the various issues associated with MARL algorithms, many organizations and individuals have proposed and developed their own multi-agent reinforcement learning environments to address the shortcomings of existing problem environments. For instance, NVIDIA's Isaac and Google's vectorized 3D physics engine Brax are notable examples. Additionally, there are some vectorized environments that utilize

single agents. Although the invention of simulators and engines that leverage real GPU acceleration has significantly alleviated the problem of algorithm convergence speed, these environments still face challenges related to the complexity and cost of scaling to different numbers of agents.
Based on the aforementioned circumstances, this paper proposes a vectorized multi-agent simulator called VMAS. VMAS, as a 2D physics simulator, is capable of supporting complex problem scenarios involving various physical conditions such as gravity, distortion, and collapse. It implements a vectorized approach using PyTorch, which allows the simulator to operate in a parallel environment with batch processing on GPUs, thereby alleviating the computational efficiency issues associated with MARL. Additionally, VMAS is compatible with OpenAI Gym and RLlib, and it incorporates a variety of reinforcement learning algorithms to facilitate ease of use for developers.
Within the VMAS framework, the paper also briefly introduces the 12 sets of complex multi-robot problem environments it proposes. These problems require the implementation of sophisticated collaborative capabilities among robots, and utilizing Graph Neural Networks (GNNs) is an effective approach to address these challenges. I will elaborate on the specific details and parameters of these 12 environments in the **Developed Scenes** section.

Core Methods

In the VMAS environment, each agent has two roles: the actor and the critic. The actor is responsible for reading the available observation information to make decisions, while the critic observes the environment and outputs the results. If an agent can synchronize and access the decision-making information from all agents as well as the environmental information observed by all agents, then the agents in this scenario are referred to as centralized. Conversely, if an agent can only observe and utilize its own information, it is classified as decentralized.
Below, we introduce the principles of three different agent-based behavior algorithms that were experimented with in the paper's experiments. These algorithms represent distinct approaches to understanding and modeling agent behavior, each contributing unique insights to the overall research.

a) The **CPPO** algorithm model employs a centralized actor and critic, which allows us to simplify the multi-agent problem addressed by CPPO into that of a super-agent. This means that instead of treating each agent as an individual entity, we can consider the collective decision-making and actions of all agents as a single, unified agent, thereby streamlining the complexity of the problem.

b) The **MAPPO** algorithm model utilizes a centralized critic and decentralized actors, which means that the agents in MAPPO share the training data obtained from their observations. However, when it comes to making decisions, each agent operates independently and does not take into account the current decisions of other agents. This structure allows for collaborative learning while maintaining individual decision-making autonomy among the agents.

c) The **IPPO** algorithm model employs decentralized actors and critics, which means that all agents share the same model parameters. Each agent independently trains on the data it observes and subsequently makes its own decisions in sequence. This approach fosters a collaborative environment where agents can learn from shared parameters while still maintaining their individual training processes and decision-making autonomy.

VMAS Code Understanding

Encountered Bugs

When following the instructions from the VMAS GitHub repository to properly install the vmas library and run the example code, you may encounter issues related to the code, similar to those I experienced. Therefore, I will outline and explain potential solutions in advance to facilitate subsequent reproduction and verification of the results. The GitHub repository links referenced and utilized here include https://github.com/proroklab/VectorizedMultiAgentSimulator and https://github.com/proroklab/HetGPPO.

Numpy Bool8 ERROR

When using the officially recommended WandB for rendering with vmas, it may indirectly lead to the version of NumPy reaching 2.0.0 or higher, which could result in the following bug: *AttributeError: module 'numpy' has no attribute 'bool8'*. This issue arises due to a conflict between the high version of NumPy and certain parts of the VMAS source code. The newer versions of NumPy no longer use the bool8 type to represent boolean parameters. To resolve this issue, you can locate a code file named tensorboardx.py through the error link and replace all instances of bool8 with bool_ to correct the code.

ModelField not Found ERROR

If you are using the VMAS-RLLIB framework code, it will involve the use of the Ray library and its related functions. However, if the version of Pydantic used in VMAS is greater than 2, it will conflict with Ray versions greater than 2.3, leading to the bug: *AttributeError: module 'pydantic.fields' has no attribute 'ModelField'*. Therefore, it is necessary to downgrade Pydantic to a version lower than 2 to avoid this issue.

WandbLoggingProcess not Pickleable ERROR

When using WandB for environment rendering, you may encounter the following bug: *RuntimeError: _WandbLoggingProcess is not pickleable. EOFError: Ran out of input.* The specific cause of this issue is related to the lower version of the Ray library. It has been noted in the GitHub issue that this problem has been fixed in Ray version 2.3.0. Therefore, upgrading to this version or later should resolve the issue.

Assert False ERROR

In the code file multi_trainer.py of another auxiliary module by the VMAS authors, rllib_differentiable_comms, there is a reference to a non-existent function parameter. To enable the model to train correctly, you need to either delete this parameter or remove the corresponding if statement that checks for it. This adjustment will allow the training process to proceed without errors.

Codebase Structure

For the VMAS GitHub repository, we will focus exclusively on analyzing the code files located within the vmas folder. We will not consider files such as mpe_comparison, which are used in the original paper for efficiency comparisons with MPE, nor will we include the code files in the tests folder that are used for testing the VMAS environment and agents with pytest. This approach allows us to streamline our analysis and concentrate on the core components of the VMAS framework.
In the example folder, the rllib.py code provides a way to utilize the trainers and renderers offered by the Ray library, along with the VMAS task environment, to conduct reinforcement learning training for a specific number of agents in designated task environments. This implementation involves two important function features, namely make_env and register_env:

```
def env_creator(config: Dict):
    env = make_env(
        scenario=config["scenario_name"],
        num_envs=config["num_envs"],
        device=config["device"],
        continuous_actions=config["continuous_actions"],
        wrapper=Wrapper.RLLIB,
        max_steps=config["max_steps"],
        # Scenario specific variables
        **config["scenario_config"],
    )
    return env
......
if not ray.is_initialized():
    ray.init()
    print("Ray init!")
register_env(scenario_name, lambda config: env_creator(config))
```

make_env is a key function for customizing new problem environments in VMAS and for modifying the hyperparameters of existing problem environments. The specific implementation is found within the make_env code. This function checks whether the environment already exists; if it does, it directly calls the corresponding code file from the scenarios directory. If it does not exist, it creates a new problem environment with the specified name and uses VMAS's Wrapper to identify whether the specific macro environment comes from RLLIB, GYM, or other components. And register_env is a functional function from the Ray library that is responsible for registering the specified custom environment with RLLIB.

In the example folder, the run_heuristic.py code is an official implementation provided by VMAS that offers an optimal solution algorithm derived from a heuristic approach for all problem environments. This allows users to conveniently compare the efficiency of their MARL algorithms against this heuristic solution. The last code file, use_vmas_env.py, demonstrates how to use the make_env function and provides code for viewing and rendering the specific behaviors of each agent.

Developed Scenes

This section primarily discusses the scenarios and scenarios_data folders within the VMAS code architecture, which contain the specific code functions for the problem environments created by the VMAS authors. These folders play a crucial role in defining and managing the various scenarios that the VMAS framework is designed to address.

a) According to the character sequence of the code, the first environment code is **balance.py**. In this scenario, n_agents are spawned uniformly spaced out along a line, upon which lies a spherical package with a mass of package_mass. The team and the line are randomly positioned along the X-axis at the bottom of the environment. The environment is influenced by vertical gravity, and the agents must carry the package to the designated goal. Each agent receives the same reward, which is proportional to the variation in distance between the package and the goal. Each agent can utilize the built-in function observation to perceive information including its own position, its own velocity, the distance relative to the package and the line, as well as the status and velocity of the package and the line. The following code represents a core part written by VMAS for creating this balance scenario:

```
# Make world
world = World(batch_dim, device, gravity=(0.0, -0.05), y_semidim=1)
# Add agents
for i in range(self.n_agents):
    agent = Agent(
        name=f"agent_{i}",
        shape=Sphere(self.agent_radius),
        u_multiplier=0.7,
    )
    world.add_agent(agent)
# Add landmarks
goal = Landmark(
    name="goal",
    collide=False,
    shape=Sphere(),
    color=Color.LIGHT_GREEN,
)
world.add_landmark(goal)
self.package = Landmark(
    name="package",
    collide=True,
```

```
    movable=True,
    shape=Sphere(),
    mass=self.package_mass,
    color=Color.RED,
)
self.package.goal = goal
world.add_landmark(self.package)
```

The three key class objects—World, Agent, and Landmark—are all defined in the underlying core package of VMAS. The code creates the balance environment and sequentially binds each agent and each environmental component into it. Finally, the environment is returned for training purposes. This structured approach ensures that all elements are properly integrated, allowing for effective interaction and learning within the environment.

b) This section of the environment consists of variants of **passage.py**, including the codes for **joint_passage_size.py, joint_passage.py, and ball_passage.py**. In the classic passage environment, a team of 5 robots is spawned in formation at a random location in the lower part of the environment. A similar formation of goals is randomly spawned in the upper part. Each robot must reach its corresponding goal. In the middle of the environment, there is a wall with n_passages, each large enough to accommodate one robot at a time. Each agent receives a reward that is proportional to the variation in distance between itself and its goal. The core of reinforcement learning is the reward system. In this case, if collisions occur between agents, those involved in the collision receive a reward of -10, which encourages agents to learn to cooperate and pass through the passages without making contact. The variant environment **joint_passage_size** connects two robots of different sizes using a rigid lever that does not account for collisions, and there are also two passages of different sizes corresponding to the sizes of the robots, complicating the cooperation problem between them. In **joint_passage**, two robots of the same size are connected, while in **ball_passage**, the robots are required to push a small ball through the passage to reach a designated location.

c) The **ball_trajectory.py** environment allows multiple agents to control a small ball, guiding it to follow a circular trajectory. The difficulty can be reduced by setting joints=True, which connects all agents to the ball. The reward function implemented in the code uses the distance between the ball and the target trajectory at the current moment compared to the previous moment, as well as the speed of the ball, to provide dynamic rewards for the agents. Meanwhile, the observation function provides information solely about the agent's own position, speed, and the distance relative to the ball. This setup encourages agents to work together effectively to maintain the ball's trajectory.

d) The **buzz_wire.py** environment features a narrow rectangular grid containing a small ball, with agents positioned outside the grid. The agents are connected to the ball, and their task is to maneuver the ball to a designated location without touching the edges of the grid. If collisions occur between agents, they receive a reward of -10. However, if any agent makes contact with the edges of the grid,

the game ends immediately. This design emphasizes precision and coordination among agents while navigating the challenging environment.

e) In the **discovery.py** environment, a team of n_agents must coordinate to cover n_targets as quickly as possible while avoiding collisions. A target is considered covered if agents_per_target agents approach it within a distance of at least covering_range. Once a target is covered, each of the agents_per_target receives a reward, and the target is respawned at a new random position. Agents incur a penalty if they collide with one another. In this environment, the overall reward is composed of the number of collisions, the number of successfully covered targets, and the time taken to achieve these objectives. This structure encourages efficient teamwork and strategic movement among the agents.

f) In the **dispersion.py** environment, n_agents and goals are spawned. All agents start at the position [0,0], while the goals are randomly positioned between -1 and 1. Agents must avoid colliding with each other as well as with the goals. Their task is to reach the goals. When a goal is reached, the team receives a reward of 1 if share_reward is set to true; otherwise, the agents that reach the goal in the same step split the reward of 1. This environment places a high demand on coordination and communication among multiple agents, ensuring that each agent selects a different task to address effectively. This requirement fosters collaboration and strategic planning within the team.

g) In contrast to the dispersion environment, the **dropout.py** environment features only a single goal. n_agents and the goal are spawned at random positions between -1 and 1. Agents must avoid colliding with each other as well as with the goal. The reward is shared among all agents, and the team receives a reward of 1 when at least one agent successfully reaches the goal. Additionally, a penalty is imposed on the team that is proportional to the sum of the magnitudes of the actions taken by every agent, which discourages unnecessary movement. This setup requires all agents to collaboratively determine which one is best suited to execute the goal, resembling the Byzantine problem in distributed systems to some extent.

h) In the **flocking.py** environment, a team of n_agents must flock around a target while staying together and maximizing their velocity. They must do this without colliding with each other or with a number of n_obstacles present in the environment. This setup emphasizes the importance of coordination and spatial awareness among the agents, as they need to navigate effectively around obstacles while maintaining their formation and speed.

i) In the **football.py** environment, a team of n_blue_agents plays football against a team of n_red_agents. As a result, the game can be viewed as either a cooperative or competitive task, depending on the context. Each agent observes its own position, velocity, relative position to the ball, and relative velocity to the ball. This information is crucial for making strategic decisions during the game, allowing agents to coordinate their movements, defend against opponents, and work together to score goals.

```python
def init_agents(self, world):
    # Add agents
    self.blue_controller = AgentPolicy(team="Blue")
    self.red_controller = AgentPolicy(team="Red")
    # Blue team
    blue_agents = []
    for i in range(self.n_blue_agents):
        agent = Agent(
            name=f"agent_blue_{i}",
            shape=Sphere(radius=self.agent_size),
            action_script=self.blue_controller.run if
                self.ai_blue_agents else None,
            u_multiplier=self.u_multiplier,
            max_speed=self.max_speed,
            color=Color.BLUE,
        )
        world.add_agent(agent)
        blue_agents.append(agent)
    # Red team
    red_agents = []
    for i in range(self.n_red_agents):
        agent = Agent(
            name=f"agent_red_{i}",
            shape=Sphere(radius=self.agent_size),
            action_script=self.red_controller.run if self.ai_red_agents
                else None,
            u_multiplier=self.u_multiplier,
            max_speed=self.max_speed,
            color=Color.RED,
        )
        world.add_agent(agent)
        red_agents.append(agent)
    # Different agents for different world
    self.red_agents = red_agents
    self.blue_agents = blue_agents
    world.red_agents = red_agents
    world.blue_agents = blue_agents
```

When creating the football environment, it is necessary to bind each group's agents to a different world due to the presence of two distinct teams of opponents. This separation is crucial because agents within the same world can communicate and coordinate their actions, while opponents should not have access to each other's internal decision-making information. This design ensures that each team operates independently while still being able to collaborate effectively among their own members.

j) In the **give_way.py** environment, two agents and two goals are spawned in a narrow corridor. Each agent needs to reach the goal corresponding to its color.

The agents are positioned in front of each other's goals, requiring them to swap places. In the middle of the corridor, there is an asymmetric opening that can accommodate only one agent at a time. Therefore, the optimal policy is for one agent to give way to the other. Unlike the football environment, where the agents are opponents, in this scenario, the two agents need to be bound within the same world to facilitate coordination. This setup emphasizes collaboration and communication between the agents to successfully navigate the corridor and reach their respective goals.

k) In the **transport.py** environment, n_agents, n_packages (default 1), and a goal are spawned at random positions between -1 and 1. The objective for the agents is to push all packages to the goal. The scenario concludes when all packages overlap with the goal. Each agent receives the same reward, which is proportional to the sum of the distance variations between the packages and the goal.

l) In the **wheel.py** environment, n_agents are spawned at random positions between -1 and 1. A line with a specified line_length and line_mass is positioned in the center. This line is constrained at the origin and is capable of rotating. The objective for the agents is to make the absolute angular velocity of the line match the desired_velocity. Consequently, it is not enough for the agents to simply push at the extremes of the line; they must coordinate their efforts to achieve, and not exceed, the desired velocity. This requirement emphasizes the importance of teamwork and precise control among the agents to effectively manipulate the line's motion.

Algorithm Reproduction

Code Analysis

According to the task objectives, it is necessary to reproduce three different algorithms: CPPO, MAPPO, and IPPO, and to select a VMAS scenario for efficiency reproduction. In this context, I have referred to the code implementation of HetGPPO, which was also authored by the creators of VMAS, for the reproduction process. Below is an analysis of the reproduced code, and my github code link is **https://github.com/Icelinea/SYSU-RLHomework**.

```
if centralised_critic and not use_mlp:
    if share_observations:
        group_name = "GAPPO"
    else:
        group_name = "MAPPO"  # 2
elif use_mlp:
    group_name = "CPPO"       # 1
elif share_observations:
    group_name = "GPPO"
else:
    group_name = "IPPO"       # 3
......
```

```
train(
        seed=seed,
        restore=False,
        notes="",
        # Model important
        share_observations=True,
        heterogeneous=False,
        # Other model
        share_action_value=True,
        centralised_critic=True,
        use_mlp=True,
        add_agent_index=False,
        aggr="add",
        topology_type="full",
        # Env
        max_episode_steps=100,
        continuous_actions=True,
    )
```

From the code, it can be observed that for CPPO, the following settings are required:

1. centralised_critic=True
2. use_mlp=True
3. share_observations=True
4. share_action_value=True

For MAPPO, the settings should be:

1. centralised_critic=True
2. use_mlp=False
3. share_observations=False
4. share_action_value=False

In contrast, for IPPO, the configurations are:

1. centralised_critic=False
2. use_mlp=False
3. share_observations=False
4. share_action_value=False

After distinguishing the variable settings between the different algorithms, the next step is to consider how to specifically implement these algorithms in terms of the architecture of the deep neural networks. This involves defining the network structure, including the layers, activation functions, and any other components that will enable

the algorithms to function effectively within the chosen framework. The design of the neural network should align with the unique requirements of each algorithm to ensure optimal performance in the multi-agent environment.

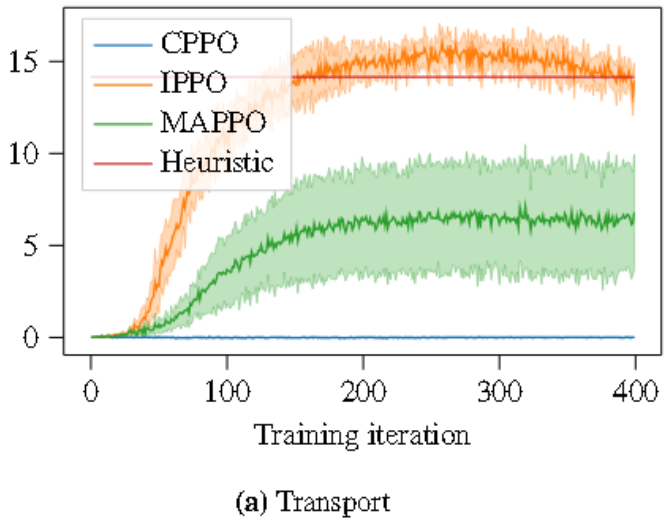
```
class GPPOBranch(nn.Module):
    def __init__(
        ...
    ):
        super().__init__()
    if self.centralised:
        # Will not get edge features
        self.centralised_mlps = nn.ModuleList(
            [
                nn.Sequential(
                    torch.nn.Linear(
                        self.in_features * self.n_agents,
                        256,
                    ),
                    self.activation_fn(),
                    torch.nn.Linear(
                        256,
                        self.hidden_size * self.n_agents,
                    ),
                )
                for _ in range(self.n_agents if self.hetero_gnns else 1)
            ]
        )
        self.gnns = None
    else:
        self.gnns = nn.ModuleList(
            [
                GNN(
                    in_dim=self.in_features,
                    out_dim=self.hidden_size,
                    edge_features=self.edge_features,
                    **cfg,
                )
                for _ in range(self.n_agents if self.hetero_gnns else 1)
            ]
        )
        self.centralised_mlps = None
```

If a centralized architecture is used, similar to the approach taken by the CPPO algorithm, the specific network layers can be implemented using a multi-layer perceptron (MLP) with two linear layers. This is because CPPO treats multiple agents as a single super agent in the context of multi-agent reinforcement learning (MARL). Therefore, when training the agents, a model that does not consider edge information from other agents can be utilized. In contrast, MAPPO and IPPO can consider using Graph Neural Networks (GNNs) to incorporate the edge information from

other agents into the training process, allowing for a more comprehensive understanding of the interactions within the multi-agent environment.

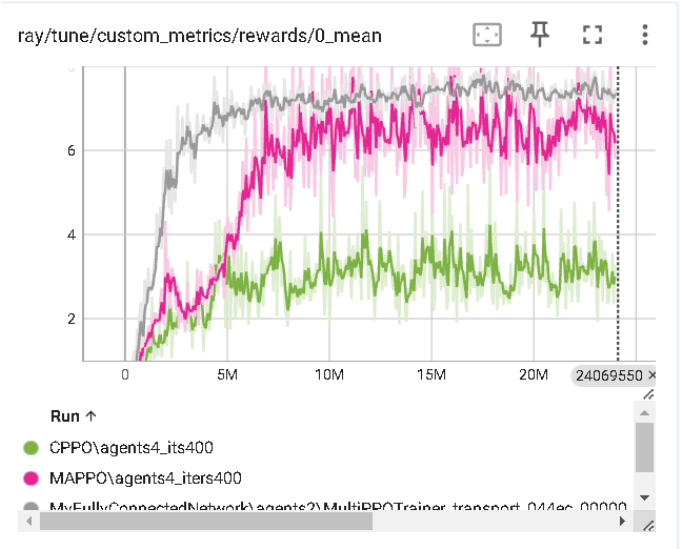
Results Comparison

In the task of reproducing training results, I used the transport scenario to demonstrate the effectiveness of the reproduction. The original paper set the number of agents in the transport scenario to 16. However, due to the limitations of my personal equipment, which only allows training on a CPU, the number of agents in the code used for reproduction was set to 4. As a result, there may be a noticeable difference in the performance when compared to the original paper's results. The images depicting the transport results from the paper and the reproduced results can be seen in Figure **??**.



(a) Transport

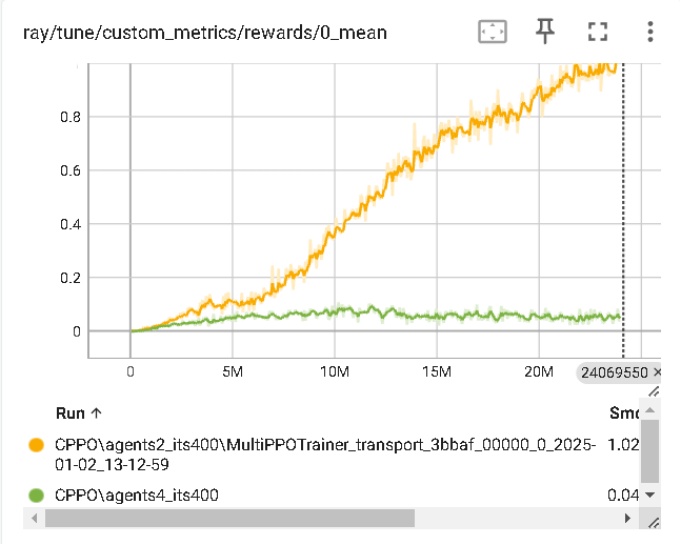
(a) VMAS Paper Outputs



(b) My Reproduction Outputs

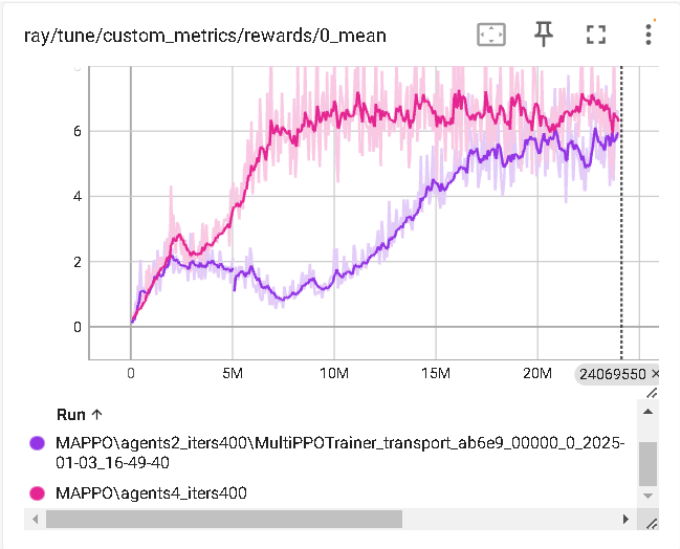
Figur 1: Reproduction Comparison with VMAS Paper

Due to the limitation of only being able to train algorithms using a CPU, I was unable to use a large number of agents while exploring the relationship between the number of agents and the three MARL algorithms: CPPO, MAPPO, and IPPO. Consequently, the number of agents compared in the reward graph below is generally restricted to 8 or fewer.
Figure **??** are the reward variation graphs for the CPPO algorithm in the transport environment, with the number of agents set to 2 and 4, respectively.



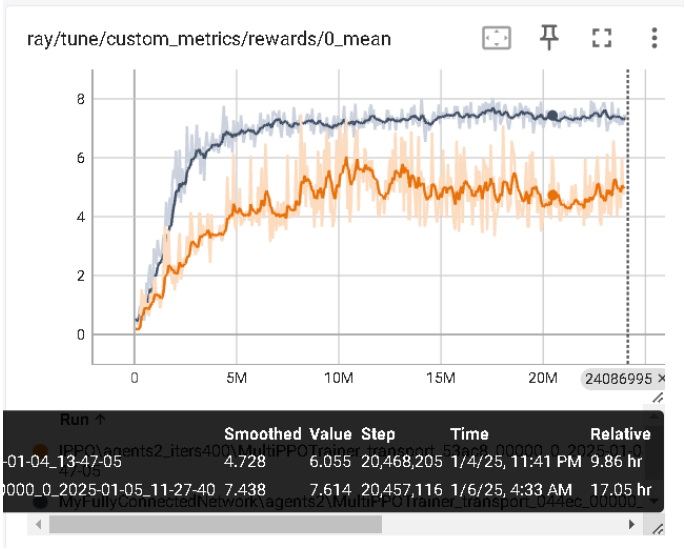
Figur 2: CPPO Algorithm Reward with Agents Number

Figure **??** are the reward variation graphs for the MAPPO algorithm in the transport environment, with the number of agents set to 2 and 4, respectively.



Figur 3: MAPPO Algorithm Reward with Agents Number

Figure **??** are the reward variation graphs for the IPPO algorithm in the transport environment, with the number of agents set to 2 and 4, respectively.



Figur 4: IPPO Algorithm Reward with Agents Number

It can be observed that for the transport environment, as the number of agents increases, the rewards for all three algorithms show an upward trend, indicating improvements in both training effectiveness and overall performance. However, if the scenario were to involve environments such as give_way or passage, simply increasing the number of agents may not necessarily lead to enhanced efficiency.

Algorithm Improvement

Code Analysis

According to the requirements outlined in the PPT, I have chosen **IPPO** as the Multi-Agent Reinforcement Learning (MARL) algorithm for improvement. The specific details of the improvements are as follows.

```
class MyFullyConnectedNetworkInner(nn.Module):
    def __init__(
        ...
    ):
        nn.Module.__init__(self)
    # Create layers 0 to second-last.
    for size in hiddens[:-1]:
        layers.append(
            SlimFC(
                in_size=prev_layer_size,
                out_size=size,
                initializer=normc_initializer(1.0),
                activation_fn=activation,
            )
        )
        prev_layer_size = size
    ...
```

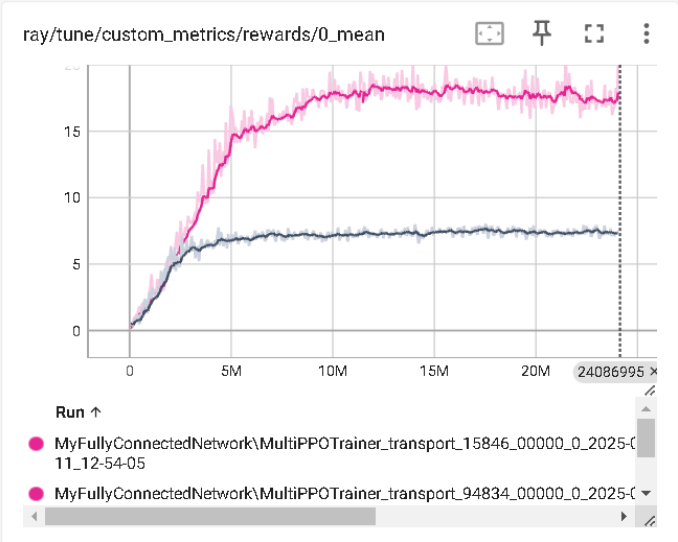
```
# Layer to add the log std vars to the state-dependent means.
if self.free_log_std and self._logits:
    self._append_free_log_std = AppendBiasLayer(num_outputs)

self._hidden_layers = nn.Sequential(*layers)
```

The SlimFC and AppendBiasLayer were implemented using the linear layers from VMAS to replace the linear layers of the old model. Additionally, the number of excessively large and numerous linear layers was simplified, resulting in faster training speeds.

Results Comparison

The reward comparison graph between the improved IPPO algorithm and the original IPPO algorithm can be seen in Figure **??**.



Figur 5: My Improvement Comparsion with IPPO Algorithm