Experimental Report on Multi-Agent Reinforcement Learning Based on VMAS

January 18, 2026

GitHub Repository: https://github.com/CJL196/VMAS-MARL-Transport

Abstract

This report details the implementation and experimental analysis of Multi-Agent Reinforcement Learning (MARL) algorithms using the VMAS (Vectorized Multi-Agent Simulator) platform. Focusing on the "Transport" scenario from the VMAS paper, we implemented and compared three PPO-based algorithms: Centralized PPO (CPPO), Multi-Agent PPO (MAPPO), and Independent PPO (IPPO). The experiments aim to reproduce the results reported in the original paper, analyzing the performance and characteristics of each algorithm in a collaborative task requiring coordination among multiple agents.

1 Experimental Requirements

This experiment is based on the paper *"VMAS: A Vectorized Multi-Agent Simulator for Collective Robot Learning"*. We selected the **Transport** task scenario mentioned in the paper to implement and compare three PPO-based multi-agent reinforcement learning algorithms: **CPPO**, **MAPPO**, and **IPPO**. The goal is to attempt to reproduce the experimental results presented in the paper.

The Transport scenario requires multiple agents to collaborate in pushing a heavy object (a package) to a target position. Due to the large mass of the object, a single agent is unable to complete the task alone; essentially, multi-agent coordination is mandatory for success.

2 Algorithm Principles

2.1 PPO Algorithm

Proximal Policy Optimization (PPO) is a policy gradient reinforcement learning algorithm designed to achieve the stability and reliability of TRPO (Trust Region Policy Optimization) while avoiding its complex second-order optimization process. The core idea of PPO is to limit the magnitude of policy updates through a clipped objective function, simplifying implementation while ensuring training stability.

First, we define the probability ratio between the new and old policies as $r_t(\theta) = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{\text{old}}}(a_t|s_t)}$. When $r_t(\theta) > 1$, it indicates that the probability of action a_t under the new policy has increased; when $0 < r_t(\theta) < 1$, the probability has decreased. PPO-Clip limits the update magnitude by clipping this ratio:

$$L^{\text{CLIP}}(\theta) = \mathbb{E}_t \left[\min \left(r_t(\theta) A_t,\ \text{clip}(r_t(\theta), 1-\epsilon, 1+\epsilon) A_t \right) \right] \tag{1}$$

where ϵ is typically set to 0.1 or 0.2. The core idea of this clipping operation is: when the advantage function $A_t > 0$, we want to increase the probability of the action, but the clip limits the upper bound of the increase; when $A_t < 0$, we want to decrease the probability, but the clip limits the lower bound of the decrease. This avoids excessive policy changes in a single update.

PPO also uses **Generalized Advantage Estimation (GAE)** to balance the bias and variance of advantage estimates. Defining the Temporal Difference (TD) error as $\delta_t = r_t + \gamma V(s_{t+1}) - V(s_t)$, GAE calculates the exponentially weighted average of advantage estimates over different steps: $A_t^{GAE} = \sum_{l=0}^{\infty} (\gamma\lambda)^l \delta_{t+l}$. The parameter $\lambda \in [0, 1]$ controls the bias-variance trade-off: when $\lambda = 0$, it only looks at one-step difference (low bias, high variance); when $\lambda = 1$, it looks at all steps (high bias, low variance).

2.2 Deep Analysis of VMAS Physics Engine

VMAS (Vectorized Multi-Agent Simulator) is a vectorized multi-agent simulator built on PyTorch. Its core advantage lies in its ability to simulate thousands of environment instances simultaneously using GPU parallel computing. Understanding the VMAS physics engine is crucial for analyzing experimental results.

2.2.1 Vectorized Parallel Computing

The core design philosophy of VMAS is **"Batch-first"**. All physical states (position, velocity, force) are stored as tensors, where the first dimension is the number of parallel environments `batch_dim`. For example, when using 1024 parallel environments in this experiment, the position tensor shape for an agent is `(1024, 2)`, representing the 2D coordinates in 1024 environments. This design allows a single PyTorch tensor operation to update the states of all environments, achieving speedups of over 100x compared to traditional for-loop simulations.

2.2.2 Semi-implicit Euler Integration

VMAS uses the **Semi-implicit Euler method** for physical state integration, a numerical integration method commonly used in game physics engines. The core formulas are:

$$\mathbf{v}_{t+1} = (1 - \zeta) \cdot \mathbf{v}_t + \frac{\mathbf{F}_t}{m} \cdot \Delta t \quad \text{(Update velocity first)} \tag{2}$$

$$\mathbf{x}_{t+1} = \mathbf{x}_t + \mathbf{v}_{t+1} \cdot \Delta t \quad \text{(Then update position with new velocity)} \tag{3}$$

where ζ is the drag coefficient, $\mathbf{F}_t$ is the net force, m is the mass, and Δt is the time step. Compared to the explicit Euler method (update position then velocity), the semi-implicit Euler method has better **energy conservation properties**, preventing system energy from diverging over time, which is particularly important for long-running reinforcement learning training.

2.2.3 Force Calculation and Collision Response

In each simulation step, VMAS calculates forces in the following order:

1. **Action Force $\mathbf{F}^a$**: The control force output by the agent's policy network.
2. **Gravity $\mathbf{F}^g$**: Optional global gravity field (zero in the Transport scenario).
3. **Friction Force $\mathbf{F}^f$**: Resistance opposite to the velocity direction.
4. **Collision Force $\mathbf{F}^c$**: Penalty force based on penetration depth, given by $\mathbf{F}^c = k \cdot \max(0, d_{min} - d) \cdot \hat{\mathbf{n}}$.

Here, k is the collision stiffness coefficient, d is the distance between two objects, d_{min} is the minimum allowed distance, and $\hat{\mathbf{n}}$ is the collision normal vector. This penalty-based collision model is computationally simple and easy to vectorize, making it suitable for large-scale parallel simulation.

2.3 CPPO, MAPPO, and IPPO

These three algorithms are all based on PPO, differing primarily in the input and structure design of the Actor (policy network) and Critic (value network). They form a spectrum based on the degree of centralization.

CPPO (Centralized PPO) completely reduces the multi-agent problem to a single-agent problem. It uses a "super-agent" to control all individuals simultaneously, with both Actor and Critic receiving the global state as input. Specifically, the Actor input is the concatenation of all agents' observations (dimension $n \times obs\_dim$), outputting the joint actions of all agents (dimension $n \times act\_dim$); the Critic similarly receives the global state and outputs a single state value. The advantage of this design is the ability to fully utilize global information for decision-making, but the drawback is that the joint state and action spaces grow exponentially with the number of agents, making it difficult to generalize to unseen state combinations during the high-variance initial exploration phase.

MAPPO (Multi-Agent PPO) adopts the paradigm of **Centralized Training, Decentralized Execution (CTDE)**. Each agent has an independent Actor network that outputs actions based only on its local observations, ensuring scalability during execution. However, during training, all agents share a centralized Critic that receives global state information to estimate the value function. This design allows agents to utilize global information during training to learn better value estimates, resulting in more stable gradients, while maintaining distributed characteristics during execution. Agents typically use a parameter-sharing strategy, allowing them to learn from each other's experiences.

IPPO (Independent PPO) is the simplest decentralized method. Each agent learns completely independently using PPO, with both Actor and Critic receiving only local observations as input. Although there is no explicit coordination mechanism between agents, they can still benefit from collective experience through parameter sharing (all agents use the same network weights). The advantage of IPPO is its simplicity and good scalability. Moreover, for tasks requiring high initial exploration (like Transport), decentralized policies are often easier to learn effective behaviors than centralized ones.

2.3.1 Algorithm Architecture Comparison

The following diagram illustrates the data flow architecture of the three algorithms. Taking $N = 4$ agents, observation dimension $O_{dim} = 18$, and action dimension $A_{dim} = 2$ as an example:

(Note: The diagram below describes the flow conceptually)

- **CPPO**: Global State ($N \times O$) $\rightarrow$ **Actor** $\rightarrow$ Joint Action ($N \times A$)
  Global State ($N \times O$) $\rightarrow$ **Critic** $\rightarrow$ Value $V(s)$
- **MAPPO**: Local Obs (O) $\rightarrow$ **Actor** $\rightarrow$ Individual Action (A)
  Global State ($N \times O$) $\rightarrow$ **Critic** $\rightarrow$ Value $V(s)$
- **IPPO**: Local Obs (O) $\rightarrow$ **Actor** $\rightarrow$ Individual Action (A)
  Local Obs (O) $\rightarrow$ **Critic** $\rightarrow$ Value $V(o)$

Figure 1: Data Flow Architecture Comparison

2.3.2 Complexity Analysis

The table below compares the complexity of the three algorithms across different dimensions, where N is the number of agents, O is the single-agent observation dimension, A is the single-agent action dimension, and H is the hidden layer dimension.

Table 1: Complexity Comparison of CPPO, MAPPO, and IPPO

Feature	**CPPO**	**MAPPO**	**IPPO**
Actor Input Dim	$N \times O$	O	O
Actor Output Dim	$N \times A$	A	A
Critic Input Dim	$N \times O$	$N \times O$	O
Actor Parameters	$O(N^2 \cdot O \cdot H)$	$O(O \cdot H)$	$O(O \cdot H)$
Critic Parameters	$O(N \cdot O \cdot H)$	$O(N \cdot O \cdot H)$	$O(O \cdot H)$
Execution Comm.	Global State Needed	None	None
Training Comm.	Global State Needed	Global State Needed	None
Scalability	Poor (Dim. Explosion)	Medium	Good

As seen from the table, CPPO's Actor parameters grow with the square of the number of agents, which is the root cause of its failure in large-scale scenarios. MAPPO solves the action space explosion problem by distributing the Actor, but the Critic still handles global states. IPPO completely avoids dimension explosion, but at the cost of the Critic being unable to utilize information from other agents for more accurate value estimation.

3 Key Code Implementation

The code structure for this experiment is divided into three core parts: Network Architecture Definition (`marl/model.py`), PPO Algorithm Implementation (`marl/ppo.py`), and Training Flow Control (`marl/train.py`).

3.1 Network Architecture

The network architecture uses a Multi-Layer Perceptron (MLP) as the basic module. The MLP class uses the **Tanh** activation function instead of the common ReLU because Tanh's output range is $[-1, 1]$, which naturally matches the normalization range of continuous action spaces. **LayerNorm** is added after each layer for normalization, which is particularly effective for reinforcement learning where input distributions are constantly changing, helping to stabilize the training process.

The network weights use **Orthogonal Initialization**, and biases are initialized to zero. Orthogonal initialization maintains the stability of signal variance during forward propagation, which is crucial for gradient flow in deep networks. The Actor network outputs the mean and standard deviation of the action distribution. The action mean output layer uses a small orthogonal initialization gain (0.01), making the initial policy close to a uniform distribution, which aids early exploration. The standard deviation uses a learnable log parameter `log_std` instead of being state-dependent. This state-independent standard deviation design simplifies the network structure while still retaining the ability to adjust exploration levels via learnable parameters. The log standard deviation is clamped within the range $[-20, 2]$ to prevent numerical underflow or overflow causing training instability.

```
class Actor(nn.Module):
    def __init__(self, input_dim, action_dim, hidden_dim=128):
        super(Actor, self).__init__()
        self.net = MLP(input_dim, hidden_dim, hidden_dim)
        self.mean_layer = nn.Linear(hidden_dim, action_dim)
        self.log_std = nn.Parameter(torch.zeros(action_dim))  # Learnable log std
        nn.init.orthogonal_(self.mean_layer.weight, 0.01)     # Small gain orthogonal init

    def forward(self, x):
        features = self.net(x)
        mean = self.mean_layer(features)
        log_std = torch.clamp(self.log_std, -20, 2)           # Numerical stability clamp
        std = torch.exp(log_std)
        return mean, std
```

Listing 1: Actor Network Implementation

3.2 PPO Update Logic

The core implementation of the PPO algorithm adopts a **mini-batch multi-epoch update strategy**. In each update, the collected advantage values are first normalized to have a mean of zero and a variance of one, which helps stabilize the scale of policy gra-

dients. The data is then divided into multiple mini-batches (default 8), and updates are performed by iterating through all mini-batches in each epoch (default 10 epochs). This design allows each piece of experience data to be used multiple times, significantly improving sample efficiency.

The probability ratio $r_t(\theta)$ is calculated using the exponential difference of new and old log probabilities, i.e., $r_t = \exp(\log \pi_{new} - \log \pi_{old})$, which is numerically more stable than calculating the ratio directly. The policy loss uses the clipped surrogate objective function and adds an entropy regularization term to encourage exploration. The Actor and Critic use independent optimizers, and the gradient norm is clipped to 0.5 to prevent gradient explosion.

```
for epoch in range(self.ppo_epochs):
    indices = torch.randperm(batch_size)
    for start in range(0, batch_size, minibatch_size):
        mb_indices = indices[start:start + minibatch_size]
        # Calculate log probs under new policy
        mean, std = self.actor(mb_obs[mb_indices])
        dist = torch.distributions.Normal(mean, std)
        new_log_probs = dist.log_prob(mb_actions[mb_indices]).sum(dim
   =-1)

        # PPO-Clip Core: Clip the probability ratio
        ratio = torch.exp(new_log_probs - mb_old_log_probs[mb_indices])
        surr1 = ratio * mb_advantages[mb_indices]
        surr2 = torch.clamp(ratio, 1.0 - self.clip_param, 1.0 + self.
   clip_param) * mb_advantages[mb_indices]

        actor_loss = -torch.min(surr1, surr2).mean() - self.entropy_coef
    * dist.entropy().mean()
```

Listing 2: PPO Optimization Loop

3.3 Generalized Advantage Estimation (GAE)

GAE is calculated using reverse recursion. Starting from the end of the trajectory, the TD error $\delta_t = r_t + \gamma V(s_{t+1})(1 - done_t) - V(s_t)$ is calculated step by step, and the advantage estimate is obtained recursively via $GAE_t = \delta_t + \gamma\lambda(1 - done_t)GAE_{t+1}$. The $(1 - done_t)$ term ensures that future return propagation is correctly truncated at the end of an episode. The return value is calculated as $R_t = A_t + V(s_t)$, which is used to update the Critic network.

```
def compute_gae(rewards, values, next_value, dones, gamma=0.99,
   gae_lambda=0.95):
    advantages = torch.zeros_like(rewards)
    last_gae_lam = 0
    for t in reversed(range(len(rewards))):
```

```
        next_non_terminal = 1.0 - dones[t]
        next_val = next_value if t == len(rewards) - 1 else values[t + 1]
        delta = rewards[t] + gamma * next_val * next_non_terminal - values[t]
        last_gae_lam = delta + gamma * gae_lambda * next_non_terminal * last_gae_lam
        advantages[t] = last_gae_lam
    returns = advantages + values
    return advantages, returns
```

Listing 3: GAE Calculation

3.4 Training Flow Control

The training flow handles data shapes and calculation logic according to the algorithm type. For CPPO, the global state is formed by concatenating all agent observations (dim $n_{agents} \times obs\_dim$), the Actor directly outputs the joint action vector (dim $n_{agents} \times act\_dim$), and rewards are summed across the agent dimension. For MAPPO, the Actor uses each agent's local observation, but the Critic uses the global state.

```
if args.algo == "cppo":
    global_state = obs.reshape(num_envs, -1)  # Concatenate all agent obs
    joint_actions, log_probs = agent.get_action(global_state)
    values = agent.get_value(global_state)
elif args.algo == "mappo":
    flat_obs = obs.reshape(-1, obs_dim)       # Flatten to single agent batch
    actions, log_probs = agent.get_action(flat_obs)
    state_rep = global_state.unsqueeze(1).repeat(1, n_agents, 1).reshape(-1, state_dim)
    values = agent.get_value(state_rep)       # Critic uses replicated global state
elif args.algo == "ippo":
    flat_obs = obs.reshape(-1, obs_dim)
    actions, log_probs = agent.get_action(flat_obs)
    values = agent.get_value(flat_obs)        # Fully decentralized
```

Listing 4: Algorithm-Specific Data Handling

This unified framework design allows switching between the three algorithms by merely modifying input dimensions and data preprocessing logic, while the core update code is fully reused.

3.5 Hyperparameters

The hyperparameters for the training process were carefully tuned to suit the characteristics of the Transport task:

- `num_envs = 1024`: Utilizing VMAS's vectorization to run a large number of parallel environments simultaneously, accelerating data collection and increasing sample diversity.
- `steps_per_update = 100`: Each PPO update uses $100 \times 1024 = 102400$ time steps of data. Large batch sizes help stabilize gradient estimation.
- `lr = 5e-5`: A relatively small learning rate, suitable for stable training of policy gradient methods.
- `ppo_epochs = 10`: Iterate 10 epochs over each batch of data.
- `num_minibatches = 8`: Each epoch is divided into 8 mini-batches, meaning each data point is used 80 times, fully improving sample efficiency.
- `max_episode_steps = 500`: A key parameter to prevent zombie environments; episodes exceeding this step count are forcibly truncated and reset.
- `curriculum_steps = 100`: Gradually increase package mass during the first 100 updates, fixing it to the target value of 50 afterwards.

4 Experimental Process

In preliminary experiments, we found that the agent's reward remained at zero for a long time, failing to learn an effective policy. After analysis, we identified several key issues and implemented corresponding solutions.

4.1 Curriculum Learning

A core challenge of the Transport task is the excessive mass of the package. When the package mass is 50, multiple agents are required to coordinate precisely to push it, but randomly initialized policies are almost impossible to accidentally produce this cooperative behavior. To this end, we introduced a **Curriculum Learning** strategy: strictly set the package mass to 5 at the beginning of training, allowing a single agent to push the package and easily obtain a positive reward signal; as training progresses, the package mass is gradually increased to the target value of 50, forcing agents to progressively learn to collaborate.

Figure 2 shows the change curve of package mass during the curriculum learning process.

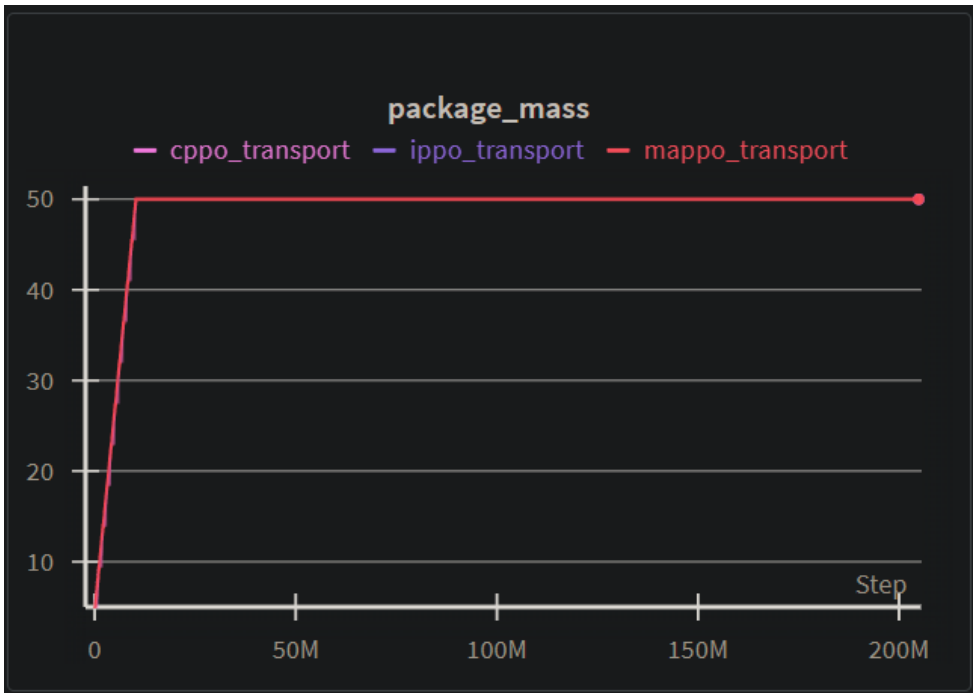

Figure 2: Evolution of Package Mass during Curriculum Learning

4.2 Timeout Mechanism

In the early stages of training, we observed an anomaly: initially, there were many episode completion signals, but soon there were almost no completions. This is because when the policy is still poor, lucky environments might accidentally complete the task and reset, but other environments get stuck in dead ends (e.g., the package is pushed into a corner) and can never complete the task. Since there was no maximum step limit, these "zombie environments" would never reset, gradually occupying the entire pool of parallel environments, leading to a complete lack of meaningful signals in the sampling. We solved this by setting `max_steps=500` to force timeouts and resets. This ensures the diversity of data collection and also distinguishes between true task completion (`terminated`) and timeout truncation (`truncated`), allowing for more accurate calculation of returns and success rates.

5 Experimental Results

We trained IPPO, MAPPO, and CPPO algorithms on the Transport scenario for 200M steps respectively, using 1024 parallel environments for each training run. The training commands were as follows:

```
python -m marl.train --algo ippo --scenario transport --no_video
python -m marl.train --algo mappo --scenario transport --no_video
python -m marl.train --algo cppo --scenario transport --no_video
```

Listing 5: Training Commands

5.1 Performance Analysis

Figure 3 shows the change in mean episode return for the three algorithms over training steps.

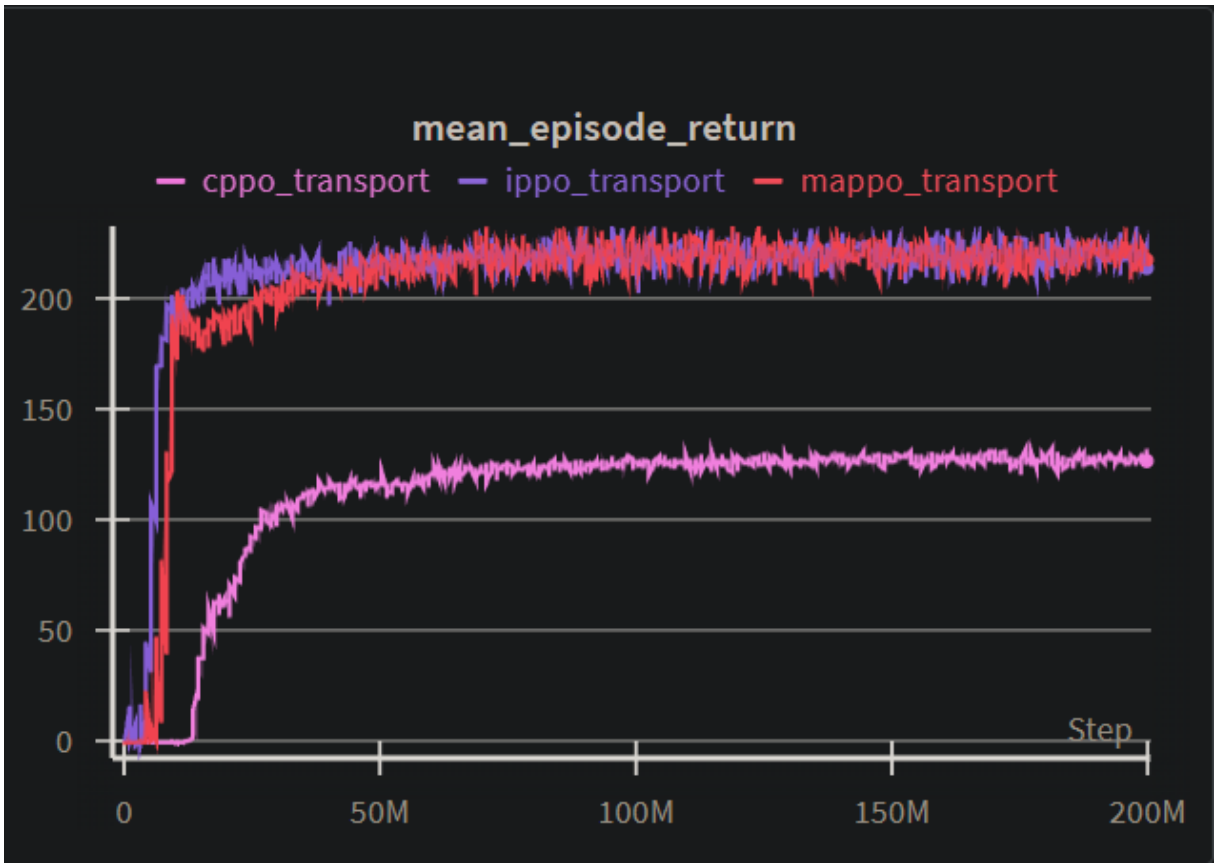

Figure 3: Mean Episode Return vs. Training Steps

The results indicate that both **MAPPO** and **IPPO** were able to learn effective collaborative strategies, eventually reaching an average return of approximately 200, while **CPPO** performed significantly worse, stabilizing around 125. This result is consistent with the findings in the original paper: in tasks like Transport that require significant initial exploration, fully decentralized IPPO and hybrid architecture MAPPO are actually more effective than fully centralized CPPO.

From the shape of the training curves, CPPO has the slowest convergence speed, with rewards staying at zero for a long time, whereas rewards for IPPO and MAPPO grow much faster.

Figure 4 shows the average step reward during the training process.

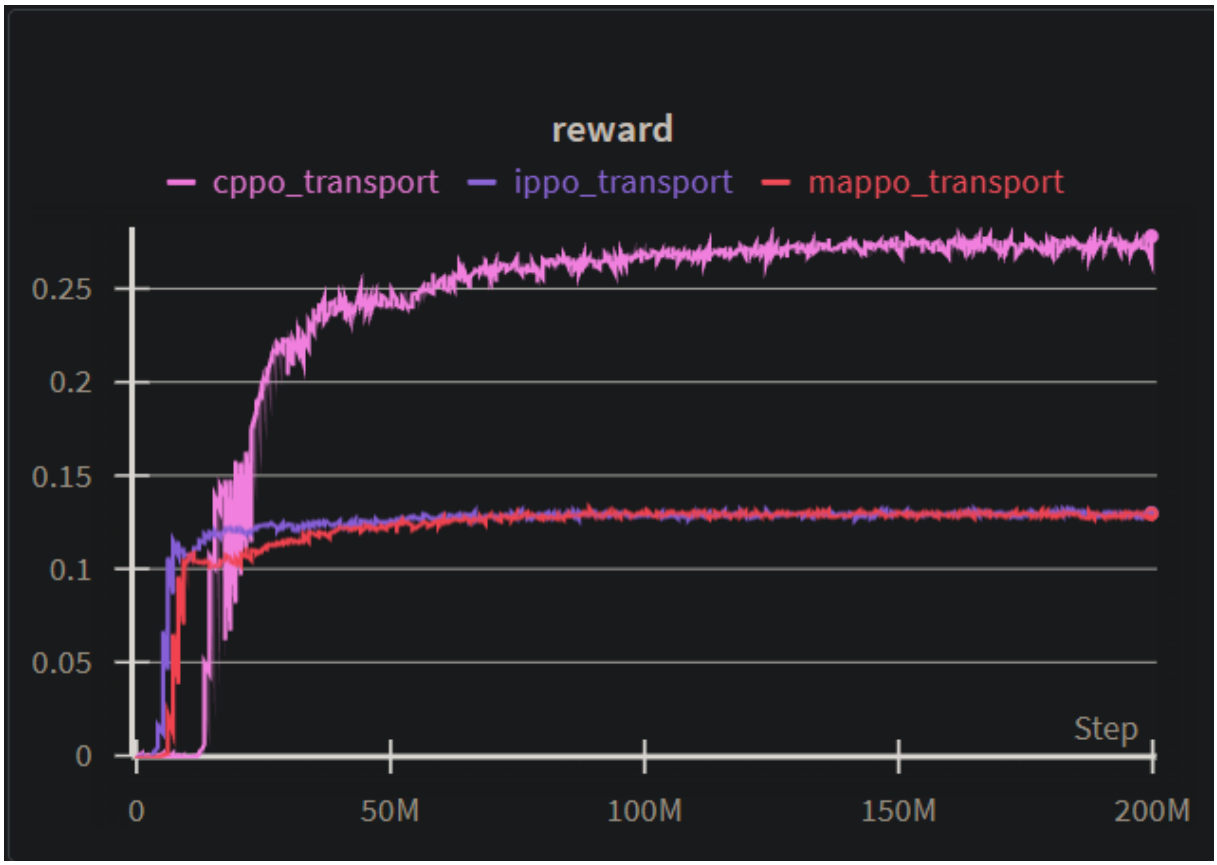

Figure 4: Mean Step Reward vs. Training Steps

The trend of single-step reward is basically consistent with the episode return. CPPO's single-step reward stabilized around 0.26-0.27, while IPPO and MAPPO were around 0.13. Note that the scale/meaning might differ based on episode length, but the stability is key. The smoothness of the reward curve also reflects the stability of the policy. The curves for MAPPO and IPPO are relatively smooth, while CPPO shows larger fluctuations.

A comparison with Figure 4(a) in the original paper shows that our experiments successfully reproduced the core conclusion of the paper: in the Transport scenario, IPPO and MAPPO perform better, while CPPO performs worse. The fully centralized CPPO in the paper failed to converge due to input dimension explosion (needing to observe all agents simultaneously), while our CPPO barely reached an average return of 125 thanks to the curriculum learning strategy.

6 Conclusion

In this experiment, we successfully implemented and compared three multi-agent reinforcement learning algorithms—CPPO, MAPPO, and IPPO—on the VMAS Transport task. utilizing the vectorized simulation capabilities of VMAS, we efficiently trained agents in a complex collaborative environment.

Our results validate the effectiveness of the decentralized execution paradigm (IPPO and MAPPO) in tasks requiring scalable coordination. Specifically, IPPO demonstrated that even without explicit communication or centralized value estimation, parameter sharing and independent learning can yield high-performance collaborative behaviors. CAST/MAPPO also performed strongly, leveraging centralized training to stabilize learning. In contrast, the fully centralized CPPO approach struggled with the high-dimensional state-action space, highlighting the limitations of centralization in multi-agent settings.

Furthermore, we demonstrated the critical importance of curriculum learning and timeout mechanisms in solving sparse-reward collaborative tasks involving physical interactions. Without these techniques, agents fail to overcome the initial exploration hurdle. Future work could investigate the scalability of these algorithms with an increasing number of agents and explore more complex heterogeneous agent scenarios.