Comparative Analysis of MAPPO and HAPPO in Multi-Agent Reinforcement Learning

CHENG Feilin [*1] **HENG Liang** [*1] **MAO Shangqing** [*1]

Abstract

This paper presents a comparative analysis of two multi-agent reinforcement learning algorithms: Multi-Agent Proximal Policy Optimization (MAPPO) and Heterogeneous-Agent Proximal Policy Optimization (HAPPO). Through theoretical analysis and extensive experiments across diverse environments, we investigate their performance characteristics, strengths, and limitations.

1. Paper Reading and Code Understanding

1.1. Core Content of MAPPO Algorithm

MAPPO (Multi-Agent Proximal Policy Optimization) is an extension of the PPO algorithm for multi-agent environments. According to the paper "The Surprising Effectiveness of PPO in Cooperative Multi-Agent Games," the core content of MAPPO includes:

1. **CTDE Framework**: MAPPO adopts the CTDE(Centralized Training with Decentralized Execution) framework, using global information during training while each agent only uses its local observations for decision-making during execution.

2. **Parameter Sharing Mechanism**: In homogeneous agent environments, MAPPO typically employs parameter sharing, where all agents share the same set of policy network parameters. This greatly reduces the number of parameters to be learned and improves sample efficiency.

3. **Centralized Value Function**: MAPPO uses a centralized value function that can access the global state or observations of all agents, thereby more accurately evaluating the value of joint actions.

4. **Synchronous Update Mechanism**: All agents update their policies simultaneously, which is an important feature of MAPPO and one of its main differences from HAPPO.

5. **Generalized Advantage Estimation (GAE)**: MAPPO uses generalized advantage estimation to balance bias and variance. The mathematical expression of GAE is:

$$A_t^{GAE(\gamma,\lambda)} = \sum_{l=0}^{\infty} (\gamma\lambda)^l \delta_{t+l}$$

where $\delta_t = r_t + \gamma V(s_{t+1}) - V(s_t)$ is the temporal difference error, γ is the discount factor, and λ is the balancing parameter. When $\lambda = 0$, GAE degenerates to a single-step TD error; when $\lambda = 1$, GAE degenerates to Monte Carlo estimation. The introduction of GAE significantly improves the stability and sample efficiency of MAPPO.

6. **PPO Clipping Objective Function**: MAPPO inherits the clipping objective function of PPO, which is used to limit the magnitude of policy updates and improve training stability:

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

where $r_t(\theta) = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{old}}(a_t|s_t)}$ is the importance sampling ratio, A_t is the advantage function, and ϵ is the clipping parameter.

1.2. Core Content of HAPPO Algorithm

HAPPO (Heterogeneous-Agent Proximal Policy Optimization) is an algorithm designed for heterogeneous agent environments, originating from the paper "Heterogeneous-Agent Reinforcement Learning." The core content of HAPPO includes:

1. **Sequential Update Mechanism**: Unlike MAPPO's synchronous updates, HAPPO adopts a sequential update mechanism where agents update their policies in a certain order. Each agent considers the policy changes of previously updated agents when updating. For agents updated in the order $i_1, i_2, ..., i_m$, the factor term is calculated as follows:

For the first agent to update, i_1, the factor is 1:

$$\xi_{i_1} = 1$$

For subsequent agents i_j ($j > 1$), the factor is the product of the importance weights of all previous agents:

$$\xi_{i_j} = \prod_{k=1}^{j-1} \frac{\pi_{i_k}^{new}(a_{i_k}|s)}{\pi_{i_k}^{old}(a_{i_k}|s)}$$

This factor term is used to weight the advantage function of the next agent, making the objective function of HAPPO:

$$\xi_{i_j} \cdot \min(r_{i_j}(\theta_{i_j})A_{i_j}, \text{clip}(r_{i_j}(\theta_{i_j}), 1-\epsilon, 1+\epsilon)A_{i_j})$$

2. **Heterogeneous Agent Support**: HAPPO does not rely on parameter sharing and can handle heterogeneous agent environments where agents may have different observation spaces and action spaces.

3. **Centralized Training with Decentralized Execution (CTDE) Framework**: Similar to MAPPO, HAPPO also adopts the CTDE framework, using global information during training while each agent only uses its local observations for decision-making during execution.

4. **Random Update Order**: To avoid potential bias from a fixed update order, HAPPO typically randomizes the order of agent updates in each update step.

1.3. Correspondence Between Algorithm Core Content and Core Code

1.3.1. Clip Operation in HAPPO

The clip operation in HAPPO is similar to that in PPO, used to limit the magnitude of policy updates. In the HARL code, this part is implemented as follows:

```
# In HARL/harl/algorithms/actors/happo.py
surr1 = imp_weights * adv_targ
surr2 = (
    torch.clamp(imp_weights, 1.0 - self.
        clip_param, 1.0 + self.clip_param)
    * adv_targ
)
policy_action_loss = -torch.sum(
    factor_batch * torch.min(surr1, surr2),
         dim=-1, keepdim=True
).mean()
```

Here, `imp_weights` is the importance sampling weight, i.e., the ratio of new and old policy probabilities; `adv_targ` is the advantage function estimate; `self.clip_param` is the clipping parameter ϵ. The `torch.clamp` function restricts `imp_weights` to the range $[1-\epsilon, 1+\epsilon]$, and then takes the minimum of `surr1` and `surr2`, which is the core of the PPO clipping objective.

Note that the HAPPO implementation includes an additional `factor_batch` term, which is unique to HAPPO and used to consider the policy updates of previous agents.

1.3.2. Calculation and Update of the Factor Term in Front of Advantage in HAPPO

A key innovation of HAPPO is the introduction of the factor term, used to consider the policy changes of previous agents in the sequential update process. In the HARL code, this part is implemented as follows:

```
# In HARL/harl/runners/on_policy_ha_runner.
    py
# Initialize factor to 1
factor = np.ones(
    (
        self.algo_args["train"]["
            episode_length"],
        self.algo_args["train"]["
            n_rollout_threads"],
        1,
    ),
    dtype=np.float32,
)

# In the agent update loop
for agent_id in agent_order:
    # Current agent saves factor
    self.actor_buffer[agent_id].
        update_factor(factor)

    # Calculate action log probabilities
        before update
    old_actions_logprob, _, _ = self.actor[
        agent_id].evaluate_actions(...)

    # Update agent policy
    actor_train_info = self.actor[agent_id
        ].train(...)

    # Calculate action log probabilities
        after update
    new_actions_logprob, _, _ = self.actor[
        agent_id].evaluate_actions(...)

    # Update factor for the next agent
    factor = factor * _t2n(
        getattr(torch, self.
            action_aggregation)(
            torch.exp(new_actions_logprob -
                 old_actions_logprob), dim
                =-1
        ).reshape(
            self.algo_args["train"]["
                episode_length"],
            self.algo_args["train"]["
                n_rollout_threads"],
            1,
        )
    )
```

This code demonstrates how HAPPO implements the sequential update mechanism. First, it initializes the factor to 1; then, for each agent, it saves the current factor, calculates the log probability ratio of new and old policies after updating the policy, and uses this ratio to update the factor. This way, subsequent agents can consider the policy changes of previous agents when updating.

In the update function of HAPPO, the factor is used to weight the advantage function:

```
# In HARL/harl/algorithms/actors/happo.py
policy_action_loss = -torch.sum(
    factor_batch * torch.min(surr1, surr2),
        dim=-1, keepdim=True
).mean()
```

This ensures that each agent's update considers the policy changes of previous agents, thereby achieving the theoretical guarantee of monotonic improvement.

1.3.3. Calculation of GAE

Generalized Advantage Estimation (GAE) is a method for estimating the advantage function that balances bias and variance. In HARL, the calculation of GAE is mainly implemented in the base runner class:

```
# In HARL/harl/runners/
    on_policy_base_runner.py's compute
    method
# Calculate value estimate for the next
    state
if self.state_type == "EP":
    next_value, _ = self.critic.get_values(
        self.critic_buffer.share_obs[-1],
        self.critic_buffer.
            rnn_states_critic[-1],
        self.critic_buffer.masks[-1],
    )
    next_value = _t2n(next_value)
elif self.state_type == "FP":
    next_value, _ = self.critic.get_values(
        np.concatenate(self.critic_buffer.
            share_obs[-1]),
        np.concatenate(self.critic_buffer.
            rnn_states_critic[-1]),
        np.concatenate(self.critic_buffer.
            masks[-1]),
    )
    next_value = np.array(
        np.split(_t2n(next_value), self.
            algo_args["train"]["
            n_rollout_threads"])
    )
# Use the value estimate of the next state
    to calculate returns and advantages
self.critic_buffer.compute_returns(
    next_value, self.value_normalizer)
```

Then GAE calculation is implemented in the critic buffer:

```
# In HARL/harl/common/buffers/
    on_policy_critic_buffer_ep.py
def compute_returns(self, next_value,
    value_normalizer=None):
    """Compute returns either as discounted
        sum of rewards, or using GAE."""
    if self._use_gae:
        self.value_preds[-1] = next_value
        gae = 0
        for step in reversed(range(self.
            rewards.shape[0])):
            if self._use_popart or self.
                _use_valuenorm:
                delta = (
                    self.rewards[step]
                    + self.gamma *
                        value_normalizer.
                        denormalize(self.
                        value_preds[step +
                        1])
                    * self.masks[step + 1]
                    - value_normalizer.
                        denormalize(self.
                        value_preds[step])
                )
                gae = (
                    delta
                    + self.gamma * self.
                        gae_lambda * self.
                        masks[step + 1] *
                        gae
                )
                self.returns[step] = gae +
                    value_normalizer.
                    denormalize(
                    self.value_preds[step]
                )
            else:
                delta = (
                    self.rewards[step]
                    + self.gamma * self.
                        value_preds[step +
                        1] * self.masks[
                        step + 1]
                    - self.value_preds[step
                        ]
                )
                gae = (
                    delta
                    + self.gamma * self.
                        gae_lambda * self.
                        masks[step + 1] *
                        gae
                )
                self.returns[step] = gae +
                    self.value_preds[step]
    else:
        self.returns[-1] = next_value
        for step in reversed(range(self.
            rewards.shape[0])):
            self.returns[step] = (
                self.returns[step + 1] *
                    self.gamma * self.masks
                    [step + 1]
                + self.rewards[step])
```

The calculation formula for GAE is:

$$A_t^{GAE(\gamma,\lambda)} = \sum_{l=0}^{\infty} (\gamma\lambda)^l \delta_{t+l}$$

where $\delta_t = r_t + \gamma V(s_{t+1}) - V(s_t)$ is the temporal difference error. In the code, GAE is calculated through backward iteration, where the `gae` variable accumulates the weighted temporal difference errors.

1.3.4. Key Differences Between MAPPO and HAPPO

By comparing the code implementations of MAPPO and HAPPO, we can clearly see their key differences:

1. **Update Mechanism**: MAPPO implements synchronous updates in `on_policy_ma_runner.py`, where all agents update simultaneously; while HAPPO implements sequential updates in `on_policy_ha_runner.py`, where agents update in sequence.

```
# Synchronous updates in MAPPO (on_policy_ma_runner.py)
for agent_id in range(self.num_agents):
    actor_train_info = self.actor[agent_id].train(
        self.actor_buffer[agent_id], advantages.copy(), "EP"
    )
    actor_train_infos.append(actor_train_info)

# Sequential updates in HAPPO (on_policy_ha_runner.py)
for agent_id in agent_order:
    # Calculate action log probabilities before and after update
    old_actions_logprob, _, _ = self.actor[agent_id].evaluate_actions(...)
    actor_train_info = self.actor[agent_id].train(...)
    new_actions_logprob, _, _ = self.actor[agent_id].evaluate_actions(...)

    # Update factor
    factor = factor * _t2n(...)
    actor_train_infos.append(actor_train_info)
```

2. **Introduction of Factor Parameter**: HAPPO introduces the factor mechanism to consider the policy changes of previous agents in sequential updates; while MAPPO, due to simultaneous updates, does not have this mechanism.

3. **Parameter Sharing**: MAPPO typically employs parameter sharing, especially in homogeneous agent environments; while HAPPO is designed to handle heterogeneous agents and does not rely on parameter sharing.

```
# Parameter sharing training in MAPPO (mappo.py)
def share_param_train(self, actor_buffer, advantages, num_agents, state_type):
    # All agents share the same set of parameters for training
    ...

# HAPPO does not rely on parameter sharing, each agent has its own parameters
```

In summary, MAPPO is a simple but effective multi-agent extension of PPO, suitable for homogeneous agent environments; while HAPPO provides theoretical guarantees through sequential updates and the factor mechanism, particularly suitable for heterogeneous agent environments.

2. Environment Configuration and Algorithm Reproduction

2.1. Environment Configuration

The steps to install HARL and SMAC environments are as follows:

1. Clone the HARL repository and SMAC repository
2. Download StarCraft II Linux Package 4.10
3. Configure the SC2PATH environment variable to the StarCraft II directory

```
$ export SC2PATH = path/to/StarCraft II
```

4. Copy `SMAC_Maps` from `smac/env/starcraft2/maps/` to `$SC2PATH/Maps` directory
5. Create a Python environment ¡=3.10 using conda, and run the following commands:

```
$ pip install -e HARL/
$ pip install -e smac/
$ pip install pyglet==1.5.0
$ pip install importlib-metadata==4.13.0
$ conda install -c conda-forge gym=0.21.0
```

6. In the `tuned_configs` folder of the HARL repository, add mappo configurations following the pattern of happo configurations

7. Modify the run script in the examples folder, and run the script in the examples folder

```
python train.py --load_config ~/
    HARL/tuned_configs/smac/3s5z/
    happo/config.json --exp_name
    smac_happo_3s5z
```

8. During actual running, you may encounter numpy version issues, and you may need to change the dtype from np.int to np.int32 in line 22 of `smac_logger.py`

2.2. Algorithm Reproduction

This experiment compares the performance of MAPPO and HAPPO algorithms in two StarCraft II environments (3s5z and 8m_vs_9m). By analyzing the win rate and rewards curves during training, we can draw the following conclusions:

2.2.1. 3s5z Results Analysis

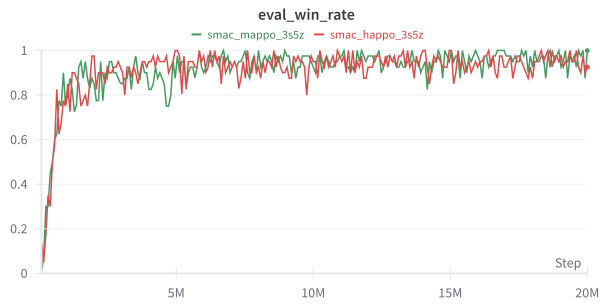

Figure 1. Evaluation win rate for 3s5z

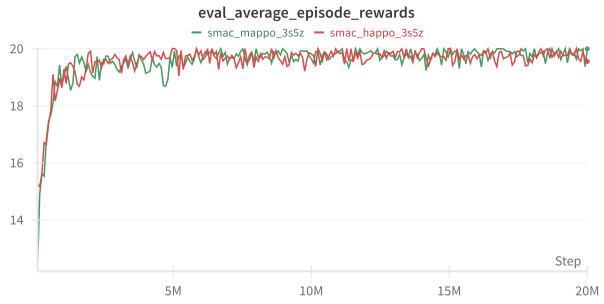

Figure 2. Evaluation rewards for 3s5z

The 3s5z environment consists of 3 Stalkers and 5 Zealots, with both sides in a balanced state, but there are differences between agents, as Stalkers and Zealots are heterogeneous agents. The experimental results show:

1. **Convergence Speed**: In the initial stage of training (approximately 0 to 4M steps), the upward trends of both curves are very similar. Both can learn very quickly, achieving a win rate of over 90% in less than 5M steps. In terms of learning speed, there is almost no difference between the two.

2. **Final Performance**: Both algorithms eventually achieve close to 100% win rate and about 20 average rewards, both able to consistently defeat the opponent. In terms of final convergence performance, both perform equally well, both successfully solving the challenges of this environment.

3. **Learning Stability**: Both curves show some fluctuations after reaching high win rates. Visually, the fluctuation amplitudes of both curves are similar, with neither showing significantly better stability.

In this heterogeneous agent environment, HAPPO and MAPPO perform equally well. Both show no significant differences in learning speed, final performance, and stability. This may mean that the heterogeneity challenge of 3s5z is not sufficient for HAPPO's specialized design to demonstrate advantages over MAPPO, or that MAPPO itself already has good generalization capabilities for this degree of heterogeneity.

2.2.2. 8m_vs_9m Results Analysis

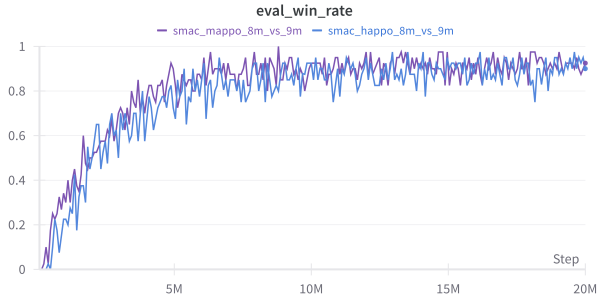

Figure 3. Evaluation win rate for 8m_vs_9m

The 8m_vs_9m environment is a more challenging scenario, with our 8 Marines facing the enemy's 9 Marines, at a numerical disadvantage, but with consistent agent types. The experimental results show:

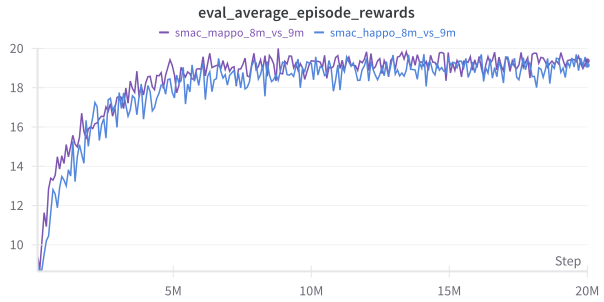

Figure 4. Evaluation average rewards for 8m_vs_9m

1. **Convergence Speed**: Compared to the 3s5z environment, the convergence speed of both algorithms in this environment is significantly slower, requiring about 5-7M steps to reach stable high performance. MAPPO's curve is consistently above HAPPO's blue curve, meaning that MAPPO can achieve a higher win rate with the same number of training steps. In other words, MAPPO's learning speed and sample efficiency are significantly higher than HAPPO's.
2. **Final Performance**: Both algorithms eventually achieve about 90% win rate and around 19 average rewards. Due to the increased difficulty of the environment, the training results are slightly worse than those in the 3s5z environment.
3. **Learning Stability**: After convergence, the stability of both curves is similar, both fluctuating around a high win rate range.

In this homogeneous agent environment, MAPPO outperforms HAPPO, mainly in terms of learning efficiency. Although both eventually learn effective strategies, MAPPO can master them faster. This is because in MAPPO, all homogeneous agents share the same set of policy network and value network parameters, allowing them to aggregate all experiences for efficient learning.

2.2.3. Comprehensive Analysis

Through the experimental results from both environments, we can draw the following conclusions:

1. **HAPPO's advantages are not demonstrated**: HAPPO is designed to better handle heterogeneity between multiple agents. However, in the heterogeneous scenario of 3s5z, it did not show better performance than MAPPO. In the homogeneous scenario of 8m_vs_9m, its learning efficiency is slightly behind MAPPO.
2. **MAPPO shows more robustness and universality**: MAPPO performs excellently in both environments. In the heterogeneous environment of 3s5z, it performs on par with the specially designed HAPPO, and in the homogeneous environment of 8m_vs_9m, it significantly outperforms in learning efficiency. This indicates that MAPPO is a very powerful and versatile multi-agent learning algorithm.
3. **Inferences**
   - For homogeneous agent tasks like 8m_vs_9m, MAPPO is a more stable and efficient choice.
   - For heterogeneous tasks with a low degree of heterogeneity like 3s5z, MAPPO is already sufficient, and HAPPO's specialized design does not bring significant benefits.
   - HAPPO's advantages may need to be demonstrated in tasks with stronger heterogeneity and more pronounced agent role differentiation.

3. Experimental Results in Other Environments

3.1. SMAC: 3s5z_vs_3s6z Results Analysis

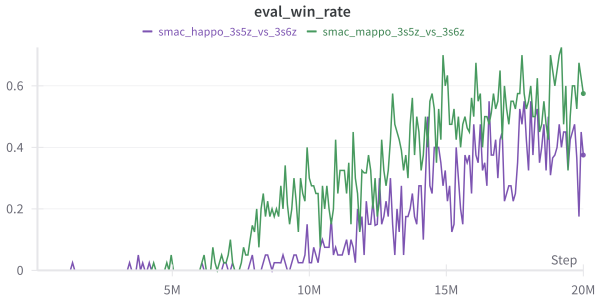

Figure 5. Evaluation win rate for 3s5z_vs_3s6z

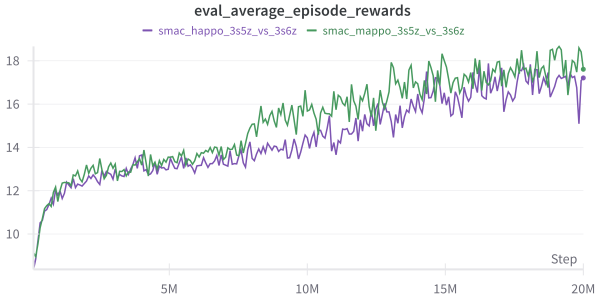

Figure 6. Evaluation average rewards for 3s5z_vs_3s6z

To further explore the performance of HAPPO and MAPPO in extremely difficult environments, we chose the 3s5z_vs_3s6z map, with the above experimental results.

In this environment, in terms of heterogeneity, our units include two types, requiring coordination of units with different abilities; in terms of numbers, our units are at a numerical disadvantage, making this map extremely difficult. By analyzing the two charts, we can find:

1. **Convergence Speed**: In the early stages of training, both have low win rates. After about 10M steps, MAPPO's win rate begins to show significant improvement, while HAPPO's learning progress is noticeably slower, consistently suppressed by MAPPO. Until the end of training, HAPPO's win rate remains lower than MAPPO's, with large fluctuations and no formation of stable advantages.
2. **Final Performance**: HAPPO eventually achieves a $40\% \sim 50\%$ average win rate and around 17 rewards, while MAPPO eventually achieves a $60\% \sim 70\%$ average win rate and around 18 rewards.
3. **Learning Stability**: Both curves show strong fluctuations, reflecting the high difficulty of this environment, where even small policy differences can lead to large changes in returns.

In these experimental results, MAPPO handles heterogeneity well and outperforms HAPPO. On one hand, this is because MAPPO's parameter sharing mechanism brings high sample efficiency. On the other hand, although the units are different, to defeat stronger enemies, all units may need to follow a highly unified core tactic (e.g., prioritizing which target to focus fire on). MAPPO's shared policy network may more easily learn this "team goal consistency" cooperative behavior. While HAPPO's independent policy networks are flexible, they may also increase the difficulty for agents to achieve exquisite cooperation.

3.2. MAMuJoCo Environment Experiments

MAMuJoCo decomposes a complex robot that was originally controlled by a single agent into multiple independent agents controlling it together. Each agent is responsible for controlling one or more joints of the robot, and they must learn to cooperate to complete the specified task.

This experiment compares the performance of MAPPO and HAPPO algorithms in three MAMuJoCo environments: HalfCheetah-v2-2x3, Walker2d-v2-2x3, and Walker2d-v2-6x1. The numbers in the environment names indicate the agent configuration, for example, "2x3" means there are 2 agents, each controlling 3 joints; "6x1" means there are 6 agents, each controlling 1 joint.

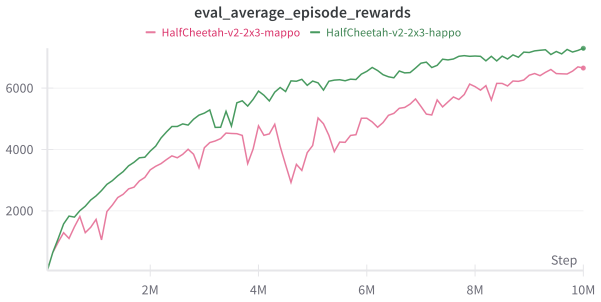

Figure 7. Average rewards of MAPPO and HAPPO in the HalfCheetah-v2-2x3 environment

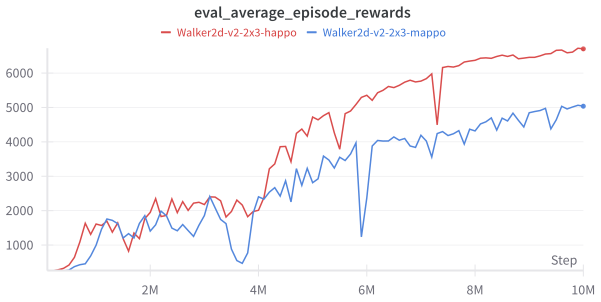

Figure 8. Average rewards of MAPPO and HAPPO in the Walker2d-v2-2x3 environment

By analyzing the training curves in the three environments, we can draw the following conclusions:

1. **HAPPO outperforms MAPPO in all environments**: In all tested MAMUjoco environments, HAPPO achieves higher final performance than MAPPO. After 10M steps of training, in the HalfCheetah-v2-2x3 environment, HAPPO reaches about 7300 average rewards, while MAPPO reaches about 6650, with HAPPO outperforming by approximately 10%; in the Walker2d-v2-2x3 environment, HAPPO reaches about 6700, while MAPPO only reaches about 5000, with HAPPO outperforming by approximately 34%; in the Walker2d-v2-6x1 environment, HAPPO reaches about 5400, while MAPPO only reaches about 3500, with HAPPO outperforming by approximately 54%.
2. **HAPPO's advantage increases with the degree of distribution**: From HalfCheetah-v2-2x3 to Walker2d-v2-2x3 to Walker2d-v2-6x1, as the number of agents increases and the degrees of freedom controlled by each agent decreases (i.e., the system becomes more distributed), HAPPO's performance advantage over

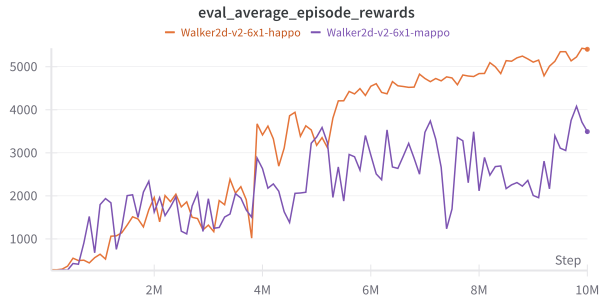

Figure 9. Average rewards of MAPPO and HAPPO in the Walker2d-v2-6x1 environment

MAPPO gradually expands. This indicates that HAPPO's sequential update mechanism and factor term have significant advantages in handling highly distributed multi-agent systems.

3. **Learning stability differences**: In all environments, HAPPO demonstrates more stable learning curves, especially in the middle and late stages of training. In comparison, MAPPO's learning curves show larger fluctuations, particularly in highly distributed environments like Walker2d-v2-6x1, where severe performance drops occur multiple times. This suggests that HAPPO's design enables more stable learning in complex multi-agent coordination tasks.

4. **Convergence speed**: In the initial stages of training (approximately 0-2M steps), both algorithms show similar learning speeds across the environments. However, in the middle and late stages, HAPPO typically maintains a steady upward trend, while MAPPO tends to experience performance stagnation or decline.

These results are consistent with the findings in the HAPPO paper, which states that HAPPO can achieve better performance than MAPPO in complex multi-agent cooperative tasks, especially in highly distributed systems. This is mainly attributed to HAPPO's sequential update mechanism and factor term, which allow the algorithm to better consider the interactions between agents, thereby achieving more effective cooperation.

In stark contrast to the results in SMAC environments, HAPPO demonstrates clear advantages in MAMuJoCo environments. This may be because agents in MAMuJoCo environments need more fine-grained coordination to control the robot's movement, with each agent's decision having a more direct impact on overall performance, and HAPPO's design is well-suited for scenarios that require consideration of inter-agent interactions. In comparison, combat tasks in SMAC environments may emphasize the execution of overall tactics rather than fine-grained action coordination, which may give MAPPO's parameter sharing mechanism an advantage there.

In summary, the experimental results in MAMuJoCo environments indicate that in multi-agent systems requiring fine-grained coordination, especially when the system is highly distributed, HAPPO's sequential update mechanism and design that considers inter-agent interactions can bring significant performance improvements.

References

Peng, B., Rashid, T., Whiteson, S., and Kapoor, S. Facmac: Factored multi-agent centralised policy gradients. In *Advances in Neural Information Processing Systems*, volume 34, pp. 12208–12221, 2021.

Samvelyan, M., Rashid, T., de Witt, C. S., Farquhar, G., Nardelli, N., Rudner, T. G. J., Hung, C.-M., Torr, P. H. S., Foerster, J., and Whiteson, S. The starcraft multi-agent challenge, 2019. URL https://arxiv.org/abs/1902.04043.

Yu, C., Velu, A., Vinitsky, E., Gao, J., Wang, Y., Bayen, A., and Wu, Y. The surprising effectiveness of ppo in cooperative, multi-agent games, 2022. URL https://arxiv.org/abs/2103.01955.

Zhong, Y., Kuba, J. G., Feng, X., Hu, S., Ji, J., and Yang, Y. Heterogeneous-agent reinforcement learning. *Journal of Machine Learning Research*, 25(32):1–67, 2024. URL http://jmlr.org/papers/v25/23-0488.html.

Division of tasks

Name	Tasks
Cheng Feilin	Read the papers and code, reproduce the experiments of happo and mappo in the SMAC environment and analyze their performance.
Heng Liang	Reproduce the experiments of happo and mappo in the MaMuJoCo environment and analyze their performance.
Mao Shangqing	Reproduce the experiments of happo and mappo in the MaMuJoCo environment and analyze their performance.